Proceedings
of the Fourth European SGI/Cray MPP Workshop
(Sep 10-11, 1998, IPP, Garching, Germany)

Editors: Hermann Lederer and Friedrich Hertweck

IPP R/46
October 1998
Proceedings
of the Fourth European SGI/Cray MPP Workshop
(Sep 10-11, 1998, IPP, Garching, Germany)

Editors: Hermann Lederer and Friedrich Hertweck

„This IPP-Report has been printed as authors’ manuscripts partially elaborated under the collaboration between the IPP and EURATOM on the field of plasma physics. All rights reserved.“
This workshop addressed computer scientists, computer engineers and end users working in the areas of application development and optimization, software support and production usage predominantly on CRAY T3E and large Origin 2000 systems installed in Europe. This event was strongly application oriented.

**Focus of this Workshop**

- MPP production applications
- Performance optimization, evaluation and prediction
- I/O optimization of large production codes
- Parallel numerical algorithms
- Load balancing strategies
- Experience with 3rd party parallel solvers/libraries

**Programme Committee:**

Laurent Colombet (CEA-Cesi - France)  
Ruediger Esser (KFA - Germany)  
Alfred Geiger (RUS - Germany)  
Friedrich Hertweck (IPP - Germany)  
Wolfgang Hiller (AWI - Germany)  
Alice Koniges (LLNL - USA)  
Hermann Lederer (RZG/IPP - Germany)  
Wolfgang Nagel (TUD - Germany)  
Alan Simpson (EPCC - United Kingdom)  
Ruediger Wolff (SGI/CRAY - Germany)

**Workshop Series:**

- First MPP Workshop: 1995 at EPFL, Lausanne, CH  
- Second MPP Workshop: 1996 (July 25, 26) at EPCC, Edinburgh, UK  
- Third MPP Workshop: 1997 (Sep 11, 12) at I.F.G. (CEA), Paris, F  
- Fourth MPP Workshop: 1998 (Sep 10, 11) at RZG/IPP, Garching, D  
- Fifth MPP Workshop: 1999 to be held at CINECA, Bologna, I

**Sponsors:**

Silicon Graphics GmbH and IPP
Contents

Performance Optimization and/or Evaluation

Stephan Seidl & Wolfgang Nagel (TU Dresden, Germany): CRAY T3E and SGI Origin2000: Merging Architectures from the User's Point of View 6 - 19
Matthias Müller & Michael Resch (HLRS, Germany): PE mapping and the contention problem on the T3E 20-28
Jim Galarowicz (SGI/Cray, US) & Bernd Mohr (FZ Jülich, Germany): Analyzing Message Passing Programs on the Cray T3E with PAT and VAMPIR 29-49
Miltos Grammatikakis, Helidon Dollani & Stefan Liesche (Uni Hildesheim, Germany): Synchronization on Cray-T3E Virtual Shared Memory 50-58
Matthias Linke (Uni Rostock, Germany): Implementation aspects and performance of some common problems on the CrAY T3E 59-68
Miltos Grammatikakis & Stefan Liesche (Uni Hildesheim, Germany): Parallel Priority Queues on Cray-T3E 69-91

MPP Production Applications

Hermann Lederer (RZG, Germany): Overview of MPP Production Applications at the Computer Center Garching 92-94
René Redler, Klaus Ketelsen, Joachim Dengg & Claus W. Bönig (AWI, Germany): A High-Resolution Numerical Model for the Circulation of the Atlantic Ocean 95-108
A. Zeiler, D. Biskamp, A. Celani, K. Hallatschek, W.C. Müller, E. Schwarz & R. Tisma (IPP, Germany): Production codes in nonlinear plasma dynamics 109-114
Tomas Wilhelmsson (KTH, S) & Josef Schüle (TU Braunschweig, Germany): Fortran memory management for parallelizing an operational ocean model 115-123
Alice Koniges, Renate Dohmen, Gesa Czapski & Bruce Scott (IPP, Germany and LLNL, US): Migration of Production Plasma Codes to the T3E 124-132
Bernadette Fritzsch, Rainer Johanni & Cornelia Köberle (AWI, Germany): Parallelization of a global ocean circulation model with coupled sea ice 133-139
Lorna Smith, Simon Jury, Peter Bladdon & Mike Cates (Edinburgh Univ., UK): Large Scale Simulation of Fluid Flow in Suspensions and Porous Media 140-149
Silke Leder (FU Berlin, Germany) & A. Beck (Uni Hannover, Germany): Parallel modelling of the exchange across the stratospheric subtropical barrier in a global 3-d model 150-157
Reiner Hegerl & Jochen Walz (MPI Biochemistry, Germany): Parallel Processing in Conjunction with Structure Analysis of Biological Macromolecules by Electron Microscopy 158-165
Programming Models

Ragnhild Blikberg & Tor Sorevik (Uni Bergen, Norway): 
*Early experiences with OpenMP on the Origin 2000*  
166-177

Parallel Numerical Algorithms

Xavier Cavin, Laurent Alonso & Jean-Claude Paul (Loria, France): 
*Experimentation of Data Locality Performance for a Parallel Hierarchical Algorithm on the Origin 2000*  
178-187

Oswald Haan (GWDG, Germany): 
*A parallel one-dimensional FFT for Cray T3E*  
188-198

Catherine Gaudart & Philippe Michallon (CEA Grenoble/Saclay, France): 
*Parallel Iterative Methods for Solving Sparse Linear Systems from Thermal-Hydraulic Codes using HPF Programming on Cray T3E*  
199-208

Support Tools

Renate Dohmen, Friedrich Hertweck & M-G Pacco-Düchs (IPP, Germany): 
*A portable and easy-to-use Fortran 90 MPP environment*  
209-224

Data Management

Kerstin Kleese (CLRC, Daresbury Lab., UK): 
*Data Management in Climate Research*  
225-231

3rd Party Solvers

Michael W. Bowers (EPCC, UK) 
*A Parallel Version of the Lagrangian-Eulerian Model of HydroGeoChemical (LEHGC) Code on the Cray T3D*  
232-241

Hartmut Häfner, Willi Schönauer & Rüdiger Weiss (Uni Karlsruhe, Germany): 
*The Parallel and Portable Linear Solver Package LINSOL*  
242-251

Simulation & Visualization

*Remote Visualization over Standard Network Connections*  
252-266

Olaf Schnapauff & Josef Schüle (TU Braunschweig, Germany): 
*Simulating and Visualizing Natural Flocking Behavior*  
267-277

Load Balancing Strategies/Methods

Paul Burton (UK Met, UK), Rupert Ford (Uni Manchester, UK) & Deborah Salmond (Cray/SGI, UK): 
*Investigation and Implementation of a number of different load balancing strategies in the UK Met. Office’s Unified Model*  
278-292
CRAY T3E and SGI Origin2000: Merging Architectures from the User’s Point of View

Stephan Seidl and Wolfgang E. Nagel
Center for High Performance Computing (ZHR)
Dresden University of Technology
D-01062 Dresden, Germany
E-mail: {seidl|nagel}@zhr.tu-dresden.de

Abstract

While the T3E is very well established as a highly parallel machine in many compute intensive environments, large Origin2000 sites still have to optimize their usage profile to get effective cycles for parallel codes even for moderate numbers of processors. The paper compares T3E and Origin2000 systems, highlighting some details with respect to parallel programming and runtime behavior of appropriate applications. The goal is not to favor one system over the other, but to give recommendations how to design applications which are able to run efficiently on both architectures.

Users are mainly faced with two differences between both systems. First, on a T3E a parallel application is statically parallel from the beginning. In case of the Origin2000, an application gets parallel during execution time when the user has control. Second, once started on a T3E, a parallel application is always running as fast as possible. On an Origin2000, this does only happen under certain circumstances. The paper will give some background about these facts and will demonstrate the strong dependence of the runtime behavior of parallel programs on different runtime situations. Comparative performance illustration of both machines will color the overall picture of the merging worlds.

1 Introduction

A few months ago, Dresden University of Technology (TU Dresden) has ordered a SGI/CRAY SN-1 machine. The system will be shipped in the second half of 1999. Part of the contract is to deliver a T3E beforehand. Since this will be a 300 MHz model, a lot of measurements have not only been done for the T3E-900. On the other hand, the TU Dresden is the largest German Origin2000 site with, over all, 56 MIPS R10k processors, 18 GBytes memory, about 300 GBytes disks, and, of course, a 21-monthed exciting story of run.

With release 6.5, the IRIX operating system comes now with a version of Miser which can also handle MPI-parallelized jobs. This means that now the task scheduler holds certain information which allow to recognize the tasks of a MPI-parallelized application as a group which is to be served as an entirety. In fact, this is very encouraging because all the prerequisites are finally fulfilled to run MPI-based parallel applications efficiently under normal batch conditions, too. The merged machine of the future will have essential characteristics of the Origin2000 concept. Nevertheless, the IRIX operating system is to be gained upon the level a user sees on a T3E, but, of course, without losing the flexibility of an Origin2000.
On a T3E, there is no fight for resources while a job is running on the application nodes, except
for pushing a packet through the connecting wires. On the Origin2000 and its successors,
respectively, this should also be true in case of Miser-controlled jobs. All the other load
coming from interactive requests should further fight for processors etc. Clearly, things like
Miser seem to be the right way here to dynamically ‘partition’ the S2MP architecture into a
critical batch job part with high priorities under strict resource control, and a remaining part
with an UNIX-like resource management.

What remains is a question which should be answered by the vendor. What does Miser do in
case of an application which performs ordinary fork() system calls, or, what is done in case
of PVM? Early investigations have shown that so far only the parent process gets the ‘batch
critical’ attribute.

2 Architecture Overview

A T3E can have up to 2048 processors over a 3D torus interconnect. The 3D torus links
have a raw bandwidth of 600 MBytes/s in each direction. One node of the system consists
of an Alpha microprocessor, a system control chip, local memory, and a network router. The
system logic runs at 75 MHz, and the processor runs with a multiple of this, i.e. at 300
MHz for the T3E which is called T3E-600 in this paper, at 450 MHz for the T3E-900, or
at 600 MHz for the T3E-1200 (delivered at a few sites in Germany over the last few weeks).
The latter has not been taken into account here. The Alpha processor is capable to do one
floating-point add and one floating-point multiply at the same time. Each processor contains
an 8 KBytes direct-mapped primary cache, an 8 KBytes instruction cache, and a 96 KBytes
unified three-way associative second-level cache. Instead of a large, board-level cache, there
is a small set of stream buffers to improve the access to stride-1 or small-stride vectors by
prefetching. The remote communication and synchronization is done between a large set of
so-called E-registers and the memory.

An Origin2000 can have up to 128 processors where up to 4 eight-vertex hypercubes are con-
ected with each other. One node board of the system consists of two R10000 microprocessors
with 4 MBytes external second-level cache each, one HUB ASIC, and local memory. Two pro-
cessors share the same memory portion through 780 MBytes/s peak. Two node boards are
connected by a six-port router unit. The processors run at 195 MHz. They are able to exe-
cute two floating-point operations per cycle. Each processor contains a 32 KBytes two-way
set associative primary cache, and a 32 KBytes two-way set associative instruction cache. The
main memory is located in a single shared address space, hence the Origin2000 is capable to
run large multi-threaded applications too. For cache coherency a directory-based protocol is
applied, using extra memory hardware which is not accessible by the user.

Picking up real-time information completely differs on the T3E and the Origin2000. On T3E,
there is a recommended intrinsic function _rtc() which returns a 64 bit integer. To get seconds,
this integer is to be divided by the return value of sysconf(_SC_CLK_TCK). On the Origin2000
the recommended code is described in syssgi(2). Its basic idea is to map a 64 bit counter into
the user’s address space via mmap(). A step unit of 800 ns is occasionally not sufficient for
performance analyzing. One gets the following results.
Table 1: Realtime timers

<table>
<thead>
<tr>
<th></th>
<th>Stepunit</th>
<th>Stepunit$^{-1}$</th>
<th>Measured overhead to access</th>
</tr>
</thead>
<tbody>
<tr>
<td>T3E</td>
<td>13.3 ns</td>
<td>75 MHz</td>
<td>&lt; 230 ns</td>
</tr>
<tr>
<td>Origin2000</td>
<td>800 ns</td>
<td>1.25 MHz</td>
<td>&lt; 320 ns</td>
</tr>
</tbody>
</table>

3 Performance Comparisons

3.1 PE Performance

To draw an exact picture of the per-PE performance of at least one well known program kernel on each of the three machines (T3E-600, T3E-900, and Origin2000), 24 variants performing matrix multiplication have been studied. These variants come from two languages, C, and Fortran, from either operating over the output matrix itself or over its transposed one, and from six possible permutations to order the loops.

Here, the results from C are of interest since they reflect some hardware properties. Typically, C compilers just generate straight code, even with `{'-03'}. On the other hand, current Fortran compilers try to recognize patterns to replace them by highly-optimized code sequences to get, at least, one floating-point instruction per cycle\(^1\). The MFLOP rates are depicted here are based on \(2n^3\) operations, where \(n\) denotes the dimension of the \(n \times n\) matrices. What we should see is that the MIPS R10k processor gets high profit from its 4 MBytes second-level cache. The curves for the transposed case look similarly, except that prefetching does not longer help in some cases.

The Fortran90 examples give an impression of the excellent T3E compilers, even so the observed compile-times are often many times longer than typical others. Bad loop orders are clearly recognized and replaced by the optimum one. One more detail is of interest here. Once started on the T3E application nodes, repeated executions exactly show the same behavior with respect to their time consumption. This is different from the situation on an Origin2000 where the reproducibility is on a lower level.

Summarizing here, the per-PE performance of a real application is not predictable. It strongly depends on the problem and on the code design. On the Origin2000 one has to specify, at least, `{'-03'` to get results which are comparable with T3E performance values. In future, program developers still should think about using the Fortran dialect Fortran90 as the implementation language to get the best performance on such HPC systems.

\(^1\)The results represented by figures 1 \ldots 12 are based on UNICOS/mk 2.0.3.23, with C 6.0.2.1, and F90 3.0.2.3, and IRIX64 6.5, with C 7.2.1, and F90 7.2.1. All the compilers have been invoked with `{'-03'` only, whereas the SGI compilers were under control of `{'abi=n32:isa=mips4:proc=r10k'`. IEEE-754 Double Precision has been used as the floating-point number format.
3.2 MPI Performance

The measurements here are based on a kernel application which interchanges messages between three processes, using the `MPI_Send()-MPI_Recv()` pair, and the `MPL_Ssend()-MPL_Recv()` pair, respectively. ‘Normalized’ denotes that the measured times have been divided by three.

The protocol for sending/receiving the messages is switched, depending on the message length. In some cases, this is pretty much optimized, sometimes not. Hence, there is still enough room for some improvement. In case of the T3Es, the important differences between the results based on `MPI_Send()` and `MPL_Ssend()`, respectively, cannot be explained in detail here. Evidently, users should use the synchronous calls wherever possible. A little bit surprising are the results for short messages, see figures 19 through 24. While the Origin2000 yields
reproducible values with small standard deviations, the round-trip times strongly varies on the T3Es. Nevertheless, the slope of the average curves of the T3Es is significantly smaller than that of the Origin2000. Using the MPI.Send()–MPI.Recv() pair, the MPI point-to-point performance of the Origin2000 is of the same order as the one of the T3E-600, except for very large message lengths. It is less than the performance of the T3E-900 at all. With respect to this pair, the values agree with [4].

Summarizing the measurement values, the great question is why the overall performance of the Origin2000 is still pretty low even if a synchronous MPI.Send() is used while the latter boosts the performance of a T3E by a factor of 2. On the other hand, on T3Es, the MPI implementation should silently change to synchronous mode in case of messages which are large enough, even if the user has invoked MPI.Send().
3.3 PVM Performance

The measurements with respect to PVM are based on a kernel application which is similar to that for MPI. It uses the \texttt{pvm.psend()} \texttt{-pvm.precv()} pair.

SGI has replaced the whole PVM in July 1998 with the result that the communication rates of the current release 3.1.1.0 with MPT 1.2.1.0 (3.3.10) are the same now as those of the open release 3.4beta6. PVM seems to be still not working together with Miser.

As shown in figure 27, one has very short startup times on the T3E-900. This is surprising because they are shorter than the MPI startup times.
Remains to give some coding hints. Interchangeable PVM codes should not contain `pvm.halt()` and `pvm_start_pvmid()` calls. On T3Es, these routines make no sense, except the application should run on the command nodes which is a bad intention. On the Origin2000, the `pvm.halt()` call does never return currently, and the group server still core-dumps, when remotely accessed, fortunately, after the application has finished. On a T3E one should branch around `pvm.spawn()` which has no meaning there. There is an appropriate stub in the library. At least on SGI hosts, MPI - as the emerging standard for message-passing applications - should be used instead of PVM wherever possible.
3.4 Communication-to-computation performance ratio

To guess the communication-to-computation performance ratio of the T3E-900 and the Origin2000, the runtime behavior of an adapted version of the well known 2D-decomposed Jacobi iteration MPI example code [3] has been investigated which comes with the VAMPIR performance analysis tool [5].

Figure 29 shows the excellent communication patterns produced on a T3E-900, running Jacobi iterations. The time line goes from left to right. Time sections where one of the processors stays in the application code are colored with blue here, yellow is for different MPI-Sendrecv() calls, and red for MPI-Allreduce(). MPI-Allreduce() sends and receives 8 bytes, i.e., very short messages, and, only MPI-Allreduce() is discussed from now. Since we need average values below a mean per-processor communication rate, $B_1$ is introduced and determined by

$$B_1 = \frac{2l(p-1)}{t_1}$$

$l$ is the message length in bytes, with $l=8$ Bytes here, $p$ is the number of processors, with $p=50$, and $t_1$ denotes the average time one of the processors spends in the MPI-Allreduce() routine. With $t_1 = 686 \mu s$ from VAMPIR one gets $B_1 = 1100 \text{ KBytes/s}$.

In case of a dedicated Origin2000, the application code runs about two times faster, and MPI-Allreduce() this needs about twice the time of a T3E-900. Although the program runs
Figure 29: One of the Jacobi iterations (NX=1000), running on 50 processors of a T3E-900

Figure 30: One of the Jacobi iterations (NX=1000), running on 50 processors of a dedicated Origin2000
Figure 29: One of the Jacobi iterations (NX=1000), running on 50 processors of a T3E-900

Figure 30: One of the Jacobi iterations (NX=1000), running on 50 processors of a dedicated Origin2000
faster at all, roughly half of the whole time is spent in the communication routines. With \( t_1 = 1590 \mu \text{s} \) from VAMPIR, \( B_1 \) equals 480 KBytes/s.

Considering only \texttt{MPI-Allreduce()} in case of the Jacobi iteration MPI example code, the communication-to-computation performance ratio of the T3E-900 is about four times the appropriate value of a dedicated Origin2000. This performance ratio is suitable to guess the parallel efficiency of a given application, using a certain number of processors.

4 Scheduling Aspects

With respect to scheduling, we should distinguish between job scheduling and task scheduling. In principle, both of them have to solve discrete optimization problems. The job scheduler has to re-order jobs with well defined resource requirements to start them preventing any resource conflict.

On T3Es, the optimization problem of the job scheduler has two variables, the number of PEs and the time. On shared memory machines, there is one more variable, which is the memory needed to execute the job. Assuming that the job scheduler has done its work, i.e. there is a plan of action, the question is how to enforce that this plan is executed in time. A common way is to remove all the interactive load from the machine. Interactive load cannot be taken into account since its resource requirements are unknown and strongly varying. Another way is to qualify the task scheduler to distinguish between task objects which are part of the plan of action or not. In figure 31, mission critical jobs are blue, green, and yellow, the red part denotes interactive load which is running under time-sharing conditions, using only resources which are not needed by the others. One should ask the question here whether it is possible to make all the pages sticky which are associated with the batch jobs.

Clearly, writing a task scheduler which is able to distribute thousands of task objects over hundreds of processors, distinguishing between mission critical tasks and non-critical ones, is one of the pretty big problems of our days. Being successful here, large application servers will win over clusters of SMP nodes, and, scientific users will win too, because they can have lots of PEs and large memory portions at the same time. Miser is a first step into this direction; nevertheless, Miser still has too many restrictions today. It will be the task of the next months for all application groups world-wide to give input about important requirements for effective scheduling issues back to the vendor to get the next generation of system software on SGI hardware ‘just right’.

Figure 31: One possible scheduling concept for large shared memory application servers
5 Sharing Cache-Lines

For the memory access conditions which a single Origin2000 processor does have far from saturation effects, see the discussion in [6]. Like the Origin, the SGI/CRAY SN-1 machine is a shared memory system, which will allow to run large multi-threaded applications too. Explicit multi-threading is not very common in the field of scientific applications. Moreover, multi-threaded applications matter little to T3Es. Even so, a kernel has been written to study performance degradations caused by concurrent write access to data which are located in the same cache line. Figure 32 shows the result of the 40 processor multi-threaded kernel in case of non-shared cache lines. It has run under Miser to enforce its execution on top of more than 120 per cent user load base.

The time runs horizontally for each thread from left to right. One color change from blue to yellow and back denotes the same portion of work. The work itself is to increment an unsigned short integer ignoring wraparounds. Unsigned short integers have been taken to ensure that one cache line is capable to contain all of them. After 25 portions of work, there is a barrier which synchronizes all the threads. Despite of some disturbances, the execution pattern is very regular. Figure 33 shows the result using the same kernel, expect that all the 40 unsigned short integers are localized in the same cache line. It seems that one can see the hardware of the Origin2000, two threads are much more faster than the others, perhaps, they are running on the processors of that node board which contains the memory where the origin of the cache line is localized. After these two processors have finished their part, all the others get able to accelerate their work. The more threads finish their work, the more the remaining threads increase in speed.

Figure 32: Runtime behavior of a multi-threaded kernel without data access collision
Figure 33: Runtime behavior of a multi-threaded kernel with maximum data access collision

Summarizing here, shared memory machines do not only allow to write slow code which is based on the message-passing parallelism, one has also the freedom to write slowly running multi-threaded applications. On the other hand, in normal situations the overhead introduced by this situation of 'wrong-sharing' data is - because of the still working principle of 'locality' in most cases - not a major problem. Nevertheless, if applications are developed, e.g. based on OpenMP, the user should carefully look to the time-consuming program parts.

6 Conclusion

We have compared the CRAY T3E, i.e. the massive parallel machine of these days, and the SGI Origin2000, a new-style shared memory system. Both systems are based on excellent hardware concepts. The merged SGI/CRAY machine of the future will be closer to an Origin2000 than to a T3E. Nevertheless, all excellent parameter values of the T3E, the transfer bandwidth, for example, should be carried on - and somewhat improved if possible - in the new architecture. IRIX, as the operating system, has done important steps to control the complicated situation of large shared memory application servers. Nevertheless, it still has to be more and more improved to ensure that, once, the job scheduling system will have the power to enforce its plan of action via task scheduling mechanisms in the presence of any interactive load.
References


PE mapping and the contention problem on the T3E

Matthias Müller* and Michael M. Resch†

August 27, 1998

Abstract

Widespread communication patterns found in most parallel programs with domain decomposition can lead to contention and thus to performance loss. We show that improved mapping of the communication topology on the physical processing elements (PEs) is one possible solution to the problem.

1 Introduction

Benchmarks are widely used to measure performance. On one hand they should be specific enough to serve as a guideline for improvements, on the other hand they should reflect the demand of real applications. The Message Passing Interface [1] (MPI) is a widespread library for communication on parallel computers. Most MPI benchmarks concentrate on point to point and global communications [2]. Some benchmarks also use kernels of applications or even complete applications [3, 4].

In the course of profiling and improving some of our parallel applications we encountered some problems that were not represented in benchmarks results available to us so far. Because we think that some of these problems are of general interest we try here to present the results and make proposals for possible new benchmarks.

One situation that occurs in many parallel application is that the algorithm performs a compute - communicate - compute loop. In this case all PEs communicate at the same time. This can overburden the communication network, the so called contention occurs.

*Institute of Computer Applications 1, University of Stuttgart, Pfaffenwaldring 27, D-70569 Stuttgart, Germany, e-mail: matthias@ical.uni-stuttgart.de
†High Performance Computing Center Stuttgart (HLRS), Allmandring 30, D-70569 Stuttgart, Germany, e-mail: resch@hlrs.de
2 PE mapping and MPI

MPI identifies a PE with its rank. In the first place there is no information about a topology associated with those ranks. On the T3E the PEs are numbered in a way that ranks with a small difference are likely to be close in the communication network. It is clear that it is impossible to provide a good solution for all situations.

MPI therefore offers to create special process topologies. One possibility is to create an n-dimensional cartesian grid with a call to MPI.Cart_create. One option allows the ranks to be reordered, this gives the implementation the possibility to remap the PEs to get the best performance with the specific hardware. The PEs of the T3E are physically organized in a bidirectional three dimensional torus. This is well suited for the cartesian grids found in applications with domain decomposition.

![Different mappings onto PEs](image)

Figure 1: Different mappings onto PEs. Left: generated with MPI.Cart_create. Right: generated with MPI.Cart_create that performs reordering.

During our investigations we found that MPI on the Cray does no reordering of PEs. We therefore tried to do an optimized reordering of PEs based on the information that we get from the sysconf call. The algorithm chosen for optimization is to sort physical coordinates that we get from sysconf according to x,y and z-coordinates. There are 6 permutations of the (x,y,z) triplet. Each
of them is tested with respect to average hop count and the optimum ordering is chosen. Results for different cartesian communicators are shown in Fig. 1.

The result of the optimization measured in reduction of average hop counts is given in Tab. 1 for varying dimensions of the cartesian communicator.

<table>
<thead>
<tr>
<th>Grid</th>
<th>without reordering</th>
<th>with reordering</th>
</tr>
</thead>
<tbody>
<tr>
<td>2x2x2</td>
<td>1.3</td>
<td>1.3</td>
</tr>
<tr>
<td>3x3x3</td>
<td>2.5</td>
<td>1.8</td>
</tr>
<tr>
<td>4x4x1</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>4x4x4</td>
<td>2.8</td>
<td>1.8</td>
</tr>
<tr>
<td>8x4x1</td>
<td>2.2</td>
<td>1</td>
</tr>
<tr>
<td>8x4x2</td>
<td>1.5</td>
<td>1</td>
</tr>
</tbody>
</table>

Table 1: Average number of hops a message has to travel.

3 Benchmarks

3.1 Pair communication

3.1.1 Description:
For contention to occur several PE have to communicate simultaneously. We choose a communication pattern where the PEs communicate pairwise with each other (see Fig. 2).

![Topology of the benchmark problem](image)

Figure 2: Topology of the benchmark problem

It’s clear that this does not reflect any real application but the design goal was to have completely independent communications. The messages are sent with `MPI.Isend` and received with `MPI.Irecv` (see Fig. 3). Because the hardware of the T3E is capable of bidirectional communication the messages could be sent
and received at the same time. We could have used MPI_Sendrecv but the use of the immediate send and receives reflects the use in application where some computations are done between the send/receive calls and the MPI_Waitany.

The topology is created with a call to MPI_Cart_create and MPI_Cart_shift. This gives the underlying MPI implementation the possibility to perform an optimized mapping onto the hardware.

MPI_Isend(sendfield, size, MPI_DOUBLE, partner, tag, comm, &request[0]);
MPI_Irecv(recvfield, size, MPI_DOUBLE, partner, tag, comm, &request[1]);
MPI_Waitall(2, request, status);

Figure 3: Code segment of the benchmark code.

We define the bandwidth as the amount of data that can be sent by one PE per second.

3.1.2 Result:

From the observed maximum bandwidth of about 200MB/s with 2 PEs (see Fig. 4) we conclude that at least to some extent bidirectional communication is performed. One link of a T3E-900 has a bandwidth of about 300MB/s for MPI.

![Bandwidth depending on message size and number of PEs. The topology is created with MPI_Cart_create.](image)

The contention is obvious in Fig. 4. The bandwidth for message sizes above
10,000 bytes drops with increasing number of PEs. For 64 PEs the bandwidth for large messages drops to less than half of the uncongested value.

![Bandwidth depending on message size and number of PEs.](image)

Figure 5: Bandwidth depending on message size and number of PEs. The topology is created with an optimized mapping.

Using the optimized mapping we can reduce congestion dramatically. Fig. 5 shows that nearly for all cases we can get better results.

3.1.3 **Extension to cyclic communication:**

To find out about the behaviour of the toroidal communication network we extended the example. Now communication is done between all neighbouring nodes in the x-direction. Each process has to talk to a left and a right neighbour. In principal we would assume that all communication can be done within one single communication step. This should also hold for cyclic communication patterns assuming that all processes are equal with respect to network topology.

For our test example we created a 8x4x1 cartesian communicator. On the one side we did tests with a periodic communicator and compared these results to tests done with a non-periodic. The tests with a periodic communicator include cyclic communication. For the non-periodic communicator no cyclic communication is done. A first guess would be that the additional communication should cost more. This would be true on machines with a standard network. However, the toroidal mesh of the T3E should show no difference. All tests were done using a fixed message size of 4MB. All measurements were done 100 times and the accumulated times are given. Without setting MPI_BUFFER_MAX one communication should take about 2.7 seconds.
First results showed a significant difference which was independent of the ordering method chosen. The overhead for cyclic communication was significant for both kinds of communicators as can be seen in Tab. 2.

<table>
<thead>
<tr>
<th></th>
<th>without reordering</th>
<th>with reordering</th>
</tr>
</thead>
<tbody>
<tr>
<td>non-periodic</td>
<td>8.2 sec</td>
<td>5.9 sec</td>
</tr>
<tr>
<td>periodic</td>
<td>8.7 sec</td>
<td>6.8 sec</td>
</tr>
</tbody>
</table>

Table 2: Time for twosided exchange communication.

The striking point here is that even for a periodic communicator we would expect that all communication could be done within two communication cycles. This should take 5.4 seconds. For the Cray T3E it takes more than 3 communication cycles. For our own method there is still some overhead.

To find out what could be done about that problem we did some more tests with different communication patterns. The result was that once cyclic communication is involved the performance of the network decreases. Further investigation made us think that we should optimize our communication pattern to make it easier for the system to cope with a fully loaded network. So we explicitly decoupled the communication and did it in two steps. For this, every second processor row started with the communication to the right neighbour. Every other second row did the same for the left one. In the next step all processors switched to the other neighbour. This way we thought that within two communication steps all communication should be finished. The results are given in Tab. 3.

<table>
<thead>
<tr>
<th></th>
<th>without reordering</th>
<th>with reordering</th>
</tr>
</thead>
<tbody>
<tr>
<td>standard</td>
<td>8.7 sec</td>
<td>5.4 sec</td>
</tr>
<tr>
<td>new</td>
<td>9.2 sec</td>
<td>5.4 sec</td>
</tr>
</tbody>
</table>

Table 3: Time for twosided exchange communication using a two step communication scheme.

Although the decoupling of communication should make the Cray MPI version faster it actually slows it down even more. We suspect that by prescribing the communication pattern we lose the possibility to optimize the communication for the bad mapping of processes. For our own mapping strategy we find that we actually can achieve peak bandwidth for that communication pattern.

In this chapter we have seen that completely independent messages can interfere with each other: the communication gets slower because the network of the parallel computer gets congested. In the next chapter we will discuss a communication pattern that can be found in real applications and show how our modified mapping can provide a solution to the contention problem.
3.2 Application benchmark

3.2.1 Description

The topology in this case is a three dimensional cartesian grid. This corresponds to a standard domain decomposition that can be found in many applications. First the communication between the neighbours of one dimension is performed. After it is completed the communication for the next dimension is started. The topology is again created with a call to MPI.Cart_create.

We defined the bandwidth as the amount of data a PE can send. Because there are two sends done simultaneously in different directions the theoretical peak bandwidth is 600MB/s for MPI.

3.2.2 Result

The observed bandwidth is always below 200MB/s (see Fig. 6). There seems to be just one send active at a time.

![Bandwidth depending on message size without reordering of PEs.](image)

Figure 6: Bandwidth depending of message size without reordering of PEs.

Again we see the congestion. In this case the bandwidth decreased from 200MB/s to 124MB/s for 128 PEs (8x4x4). The decrease is less pronounced than in the benchmark case because the different directions are affected to different degrees. A poor performance of one direction is compensated by a good performance of another one. In principle there is no reason for a congestion, the layout of the T3E is a bidirectional three dimensional torus[5, 6]. Two neighbours in a cartesian grid should have a direct connection.

However the operating system does not know whether a request for 64 PEs results in a 4x4x4 or 8x8 grid. It guarantees just a connected area in physical
PE space. This provides a first approach to local communication between PEs.
But even if the layout of the physical PEs corresponds to the requested grid
contention occurs. MPI.Cart.create does not reorder the ranks to optimize the
mapping.
Instead of calling the original MPI.Cart.create we used our own version
that performs reordering as described above. The result is shown in Fig. 7.
The bandwidth drops only to 154MB/s. We checked the quality of a mapping
by looking at the average number of hops a message has to travel. The hop
count of the grids created by our mapping was between 1 and 1.8 (see Table
1). Whenever the layout of the physical PEs corresponded to the requested grid
the hop count was one. The performance is between 2 and 40 percent better than
the unoptimized mapping for PE numbers larger than 8. Contention only occurs
when the topology is strongly disturbed. In the case of 8x8x4 PEs the domain
was practically separated into two partitions with 128 PEs. Small distortions
are handled well as can be seen in Fig. 1.

Figure 7: Bandwidth depending of message size with reordering of PEs.

4 Conclusion

We have shown that the mapping of communication topology onto the under-
lying hardware is important for benchmarks reflecting communication patterns
found in real applications. This opens a wide field of optimization. Our first
finding was that the application partition of a T3E should be configured to be
undisturbed by OS or CMD PEs in its area. This will not only provide better
mapping for the standard programs but will also enable optimized mapping algorithms to find the optimal solution. Furthermore fragmentation does not only affect the jobs you start on a system, but also the performance of the already running jobs. An optimized batch system should take this into account.

Applications which suffer under contention and communicate in a cartesian grid should use a version of `MPI_Cart_create` with reordering. The method we presented here provides such an optimized mapping. It was easy to implement and the overhead of the optimization process is negligible compared to simulation times of real applications.

5 Acknowledgements

The authors would like to thank Monika Wierse from SGI/Cray for support in using the T3E and Thomas Beisel and Rolf Rabenseifner from HLRS for useful hints and informations.

References


Analyzing Message Passing Programs on the CRAY T3E with PAT and VAMPIR

Jim Galarowicz  
Silicon Graphics/Cray Research  
655F Lone Oak Drive  
Eagan, Minnesota 55121  
jeg@cray.com

Bernd Mohr  
Forschungszentrum Jülich  
ZAM  
D-52425 Jülich, Germany  
b.mohr@fz-juelich.de

Abstract. Writing efficient parallel programs for a massively parallel system like the CRAY T3E is still a difficult task because such programs are typically very large and complex, not trivially parallelizable and their dynamic behavior is difficult to understand or predict. Therefore, runtime performance analysis tools are needed on such systems in addition to the normal programming environment tools like editors and debuggers.

For the CRAY T3E, Silicon Graphics/Cray Research implemented and provides two performance analysis tools, Apprentice and PAT. Apprentice is a profiling tool which uses source code instrumentation through compiler switches and provides statistics on the level of functions and basic blocks. PAT, the Performance Analysis Tool, is actually several tools in one. It provides profiling through sampling and access to hardware performance information. It also includes an object code instrumenter which can be used for detailed call site profiling and gathering of function level hardware performance statistics.

In a collaboration between Silicon Graphics/Cray Research and Forschungszentrum Jülich, PAT was extended to also support event tracing. In this paper, we describe how the new extended PAT and VAMPIR, an event trace browser developed by Forschungszentrum Jülich, can be used to analyze message passing programs on the CRAY T3E. The powerful trace browsing features of VAMPIR make it a perfect extension to PAT's object instrumentation and tracing functionality.

First, the features of PAT are described in detail. In order to analyze message passing programs, the message passing libraries of the CRAY T3E (MPI, PVM, and SHMEM) needed to be instrumented. This feature is described next. We then give an overview of VAMPIR and it's functionality. With two small examples, we show how the combination of PAT's object instrumentation features, the new message passing function wrapper library, and VAMPIR's trace displays can be used to analyze message passing programs on the CRAY T3E to any detail.

1 Introduction

Writing efficient parallel programs for a massively parallel system like the CRAY T3E is still a difficult task because such programs are typically very large and complex, not trivially parallelizable and their dynamic behavior is difficult to understand or predict. Therefore, runtime performance analysis tools are needed on such systems in addition to the normal programming environment tools like editors and debuggers.
Different performance indices or metrics are used to describe the performance of computer programs. In most cases, the execution time of the program or parts of the program (subroutines, procedures, functions, loops, basic blocks, ...) is measured. Depending on the computer system, operating system, and mode of operation (interactive or batch), either wall-clock time and/or time spent in user and/or system code is used. Other performance metrics include floating-point operations performed per second (FLOPS), number of cache misses, number of load and store operations, or the number of messages or amount of data transferred.

Performance analysis tools use one or both of the following techniques to measure the performance of parallel programs:

- **Profiling** is gathering statistics about the execution of the program. Profiling can be done on different levels: for the whole program, for each function/subroutine, on the level of basic blocks, or even for each statement. Typically, counting determines how often a specific part of the program was executed and how long it took. If available, other metrics like FLOPS as described above can also be recorded. Profiling is often done by sampling: at regular intervals (the sampling rate), the parallel program is interrupted and the part of the code executing is determined by looking at the program counter. This has the advantage, that the overhead for analyzing the program is very low, but with this method, the only result is the percentage of time spent in each code block. Profiling is helpful for finding those part(s) of the parallel program which consume(s) the most time or resources and therefore should be tuned. Tuning a parallel program requires obtaining more detailed statistics for the program and understanding its dynamic behavior.

- **Event tracing** is typically used to help understanding the dynamic behavior of a parallel program: the user application is instrumented (i.e., special code is inserted into the user's application) in a way so that all important events (typically function entry and exits, and sending and receiving of messages) are recorded together with the number of the processing element (PE) and a time stamp in a so-called event record. For better performance, the event records are first stored in a buffer in memory, and only later written into a file. Such a file containing a sequence of event records is called an event trace. After the execution of the program, all recorded traces are gathered and merged into a global event trace, i.e., the event records are sorted according to the time stamp.

Such an event trace describes the program on the abstraction level of the recorded events and therefore allows for reconstructing the dynamic behavior of the program. In addition, all the performance statistics can also be calculated out of the event trace. Therefore, event tracing is a much more powerful technique than profiling. However, the overhead for measuring and recording all the information is quite high. Sometimes, a mixture of both methods is used.

Both methods, profiling and event tracing, need to instrument the user program in order to be able to record the necessary information. **Instrumentation** can be done in different ways:

- The necessary changes to the user code can be done on the source code level either by the compiler or by a separate source-to-source translator which runs as a preprocessing step before the compilation. The big advantage is that it is very easy to relate the measured information or recorded events back to the source code. Also, it is possible to write source code instrumentation tools which are portable and can be used on different computer systems. On the other hand, source code instrumentation has three big disadvantages: first, for every change in the instrumentation, the user program has to be recompiled (which can be very long for complex programs), and second, it has to be implemented for every programming
language used on the computer system. If it is not done by the vendor’s compiler, it is very difficult to implement source-to-source translators for complex modern languages like C++ or Fortran90. Also, it is hard to instrument programs written in more than one programming language. Third and last, it is not possible to instrument system or 3rd-party library code which is only available as object code.

- The other option is to instrument the object code. Here, advantages and disadvantages are opposite to source code instrumentation. Object code instrumentation can automatically instrument programs written in any language, can easily handle mixed language applications, can instrument system or 3rd-party library code, and the program doesn’t have to be recompiled. On the other hand, source code mapping is very hard or impossible and the instrumentation tool is very system-specific and non-portable.

For the CRAY T3E, Silicon Graphics/Cray Research implemented and provides two performance analysis tools which cover all techniques and methods above:

- Apprentice [1] is a profiling tool which uses source code instrumentation (through compiler switches) and provides statistics (execution time, number of floating-point, integer, load, and store operations) on the level of functions and basic blocks. Because of source code instrumentation, it is not possible to get performance information on system or 3rd-party library code, and the user program has to be recompiled for the analysis.

- PAT, the Performance Analysis Tool, is actually several tools in one. It provides profiling through sampling and access to hardware performance information gathered by the DEC Alpha chip. It also includes an object code instrumenter which can be used for detailed profiling and for event tracing. Because it instruments the object code, no re-compilation of the user program is necessary and it can analyze system or 3rd-party library code.

In the following, we describe how PAT and tools developed by Forschungszentrum Jülich can be used to analyze message passing programs on the CRAY T3E. Section 2 describes all the features of PAT in detail. In order to analyze message passing programs, the message passing libraries of the CRAY T3E (MPI, PVM, and SHMEM) needed to be instrumented. This is described in Section 3. Section 4 introduces VAMPIR which was developed by Forschungszentrum Jülich and is able to analyze event traces and therefore is a perfect extension to PAT’s object instrumentation and tracing features.

2 PAT

2.1 Features of PAT

PAT has several features which enable the user to gather and display performance information for an executable program in a very lightweight manner. The user can then ask for more detailed information after viewing the information created initially. PAT also provides several displays which aid in the analysis of the data. The four major features that PAT provides are described in the following paragraphs.

2.1.1 Profile Feature with Load Balance Capability

PAT uses the profil system call to provide a program counter sampling of a program that is loaded with a special library provided with PAT (libpat.a). The user’s program does not have to be recompiled, but it does have to be linked with the special PAT library. The profil sampling data is
written to a PAT information file as the user’s program is executed. After the user’s program is finished executing and PAT is invoked, PAT associates the program counter address hits from the sampling with the addresses of the functions in the user’s executable to formulate the data to be displayed in PAT’s Profile Display window. The data displayed is the function name, the percentage of total execution time taken by the particular function, and the confidence level of that sampling. Because the program counter sampling method provides a statistical estimate of the actual time spent in a function, the tool provides a measure of confidence in the sampling estimate.

The sampling rate may be changed by the user based on the length of time the user’s executable will run. A smaller sampling rate should be used with applications that run for a short time and a larger (less frequent) sampling rate for longer running codes. An environment variable (PAT_SAMPLING_RATE) is provided to perform the sampling rate change function.

The profile data can also be displayed in a manner that shows the load balance amongst the processor elements used for the program’s execution. Here a function’s program counter sampling data can be displayed for each processor element relative to the highest usage PE, giving a load balance graph for display. This information is displayed when the Load Balance Display menu item is chosen.

2.1.2 Hardware Performance Counter Feature

PAT uses the performance counter feature of the DEC Alpha chip to gather hardware performance counter information during an user’s program execution. The data is gathered and written to a file during the user’s program execution. This performance counter information is then analyzed and displayed by PAT. This information is gathered for the whole program by default. If the user wants the information at the function level there is an interface so that individual functions may be instrumented to gather the per function information.

Because there are limits on the number of hardware performance counters for an execution of an application there is an environment variable provided allowing the user to choose the specific performance counters that will be gathered. With the environment variable PAT_SEL the user can choose from integer operations (INTOPS), floating point operations (FPOPS), memory load operations (LOADS), or memory store options (STORES) to be gathered. In addition, the number of cache misses are always recorded.

2.1.3 Call Site Analysis Feature

PAT has the capability to instrument the user’s program to gather information relating to call sites for a set of specified functions. After the program is instrumented and executed, the available information contains the names of the routines calling an instrumented function, the number of times the function was called from there, and the time spent in the function for this call site.

2.1.4 Tracing Feature

PAT provides the capability to generate trace records that show when various events occurred on different processor elements. There are two methods for obtaining trace information. First, using PAT’s object instrumenter, tracing code can be inserted at the entry and exit of functions automatically. Second the user can manually insert tracing code into the program source using PAT defined macros. There are macros supporting conditional tracing and macros to turn the tracing feature on and off. Of course, both methods can be used simultaneously. In the following sections, the tracing features of PAT are described in detail.
2.2 Event Tracing with PAT

2.2.1 Object Instrumentation of Functions

PAT has the ability to insert trace calls into a user executable. PAT inserts calls to entry and exit trace routines around calls to user or library routines. A pair of corresponding entry and exit routines is called a \textit{wrapper}. The event tracing capability allows the user to gather timing information for the multiprocessing functions themselves, output argument values, and/or gather special user specific information. The main goal of the feature is to allow the user to understand the behavior of a message passing program and then be able to tune it. The basic idea behind the tool is that every time a function is entered or left or when a message is sent and received, you record this in a trace. Then you analyze, view, and use the trace data to do the tuning.

Entry and exit trace routines can be provided in three ways:

- By the user in the form of an object (.o) or library (.a) file. These are referred to as \textit{user-defined wrappers}.
- With the generic trace routines provided by \texttt{libpat.a}. These are referred to as \textit{default wrappers}.
- With the special trace routines for MPI, PVM, and SHMEM provided by \texttt{libwrapper.a}. These are referred to as \textit{special wrappers}. They are described in detail in Section 3.1.

The format of the trace records produced are the same as the format of the source-level tracing and are displayed in the same manner.

The following rules apply to user-written entry and exit trace routines:

- Routines must be written in C.
- Routines must have the same number and same types of arguments as the routine they are tracing.
- Routines must not return a value of any kind.
- The routine name must be \texttt{name_trace_entry} for entry trace routines and \texttt{name_trace_exit} for exit trace routines.

Module(s) containing trace routines must be put in the select directive for \texttt{cld} if the modules are contained in a library (\texttt{lib+.a}). This forces \texttt{cld} to load the module(s) even though no trace routines are called and enables PAT to find trace routines in the executable file and insert calls to these routines. If the trace routines or wrapper routines are placed in an object file or set of object files, then additional \texttt{cld} select directives are not necessary. The select directives for the generic trace routines provided in \texttt{libpat.a} are already present in the default \texttt{pat.clid} file so no additional directives are needed. Also, the necessary select directives for the MPI, PVM, and SHMEM wrapper library (\texttt{libwrapper.a}) calls are already in the default \texttt{pat.clid} file so no additional directives are needed when using that library.

The following is an example for measuring the amount of data written in a C or C++ application per file descriptor. This can easily be done by writing user-defined wrappers for the functions \texttt{main} and \texttt{write}:

```c
static int ftable[OPEN_MAX];

size_t write_trace_entry (int fdindex, const void *buf, size_t nbyte) {
```
fdtable[fildes] += nbyte;
}

size_t write_trace_exit (int fildes, const void *buf, size_t nbyte) {}

int main_trace_entry ()
{
    int j;
    for (j=0; j<OPEN_MAX; ++j) fdtable[j] = 0;
}

int main_trace_exit ()
{
    int j;
    for (j=0; j<OPEN_MAX; ++j)
        if (fdtable[j]) printf ("%2d: %d\n", j, fdtable[j]);
}

If this code is linked with the application and libpat.a, and we use PAT to instrument the functions main and write, PAT ensures these user-defined wrapper functions are called before and after the corresponding functions. Execution of the instrumented version of the program will print, how many bytes were written per file descriptor.

2.2.2 Source Code Instrumentation

PAT provides interfaces which can be used to manually instrument an application program to do event tracing. Manual source code instrumentation can be used by itself (e.g., if the necessary instrumentation is simple) or in addition to the object instrumentation of PAT (if special instrumentation is needed, e.g., tracing of loop nests).

Two interfaces are available: a very simple interface (described in the next section) and a second one for advanced users, e.g., user who want to write their own user-defined wrappers (described in Section 2.2.2.2).

2.2.2.1 General Source Code Tracing

By including pat.h in your C or C++ source program, the following macros for tracing are defined:

TRACE( 'name',  p1,  p2 );

Creates a trace entry for the executing PE with name, a time stamp, and two parameters p1 and p2. The parameter name specifies an identifier for the event to be traced which must not be longer than 8 characters. Use single quotes ('), not double quotes ("), around name. The type of the parameters p1 and p2 can be any type as long it fits in one word of the T3E (8 bytes).

CTRACE( expr,  'name',  p1,  p2 );

Specifies a conditional trace; if expr is true, a trace entry is created.

TRACE_ON(), TRACE_OFF(), and TRACE_LIMIT();

Are Macros for user control of event tracing which are described in detail in Section 2.2.3.

Usage of the macros is simple as shown by the following small example:
1. Consider the following example program `progl.c`:

```c
#include <pat.h>

int main() {
    int msg_id = 31;
    TRACE( ‘sendmsg’, 1, msg_id );
    TRACE( ‘recvmsg’, 0, msg_id );
}
```

2. Compile your program:

```
$ cc -c progl.c
```

3. Relink your program with the PAT run-time library and the cld directives file:

```
$ cc progl.o -lpat pat.cld -o progl
```

4. Run your program on a CRAY T3E system to generate a `progl.pif` data file:

```
t3e> progl
```

5. Run the PAT trace command to display trace information:

```
t3e> pat progl progl.pif
=> trace
```

PAT corrects for any clock skew across PEs and displays the following trace information:

<table>
<thead>
<tr>
<th>Timestamp</th>
<th>PB(id)</th>
<th>type</th>
<th>name</th>
<th>arg1</th>
<th>arg2</th>
<th>arg3</th>
<th>arg4</th>
</tr>
</thead>
<tbody>
<tr>
<td>18.776</td>
<td>0( 1)</td>
<td>USR</td>
<td>sendmsg</td>
<td>0001</td>
<td>0x1F</td>
<td>0000</td>
<td>0000</td>
</tr>
<tr>
<td>21.939</td>
<td>0( 1)</td>
<td>USR</td>
<td>recvmsg</td>
<td>0000</td>
<td>0x1F</td>
<td>0000</td>
<td>0000</td>
</tr>
</tbody>
</table>

The trace functionality is also available in Fortran90. The Fortran90 version differs in the following ways:

- Trace macro calls are Fortran subroutine calls
- Fortran90 names use the C or C++ names prefixed by `PAT$` (for example, `TRACE` becomes `PAT$TRACE_INT` or `PAT$TRACE_CHAR` and `TRACE.ON` becomes `PAT$TRACEON`). Because of Fortran90 language considerations, calls to `PAT$TRACE_INT` and `PAT$TRACE_INT` require special attention (which are explained in detail in the PAT man page).
- Fortran90 users should include the `pat.fh` file in their source. The `pat.fh` file contains Fortran90 interface definitions required for tracing subroutines. Therefore, the include statement must be located in a place where an interface definition is legal in Fortran90.

### 2.2.2.2 Source Code Tracing Functions for Writing User-defined Wrappers

The interface described here is only available as C and C++ macros because it was designed to support the implementation of user-designed and special wrapper functions (which have to be written as C functions). It is of course possible to use them for tracing of C or C++ programs outside of wrapper functions too. By including `event_trace.h` the following macros are available to the programmer:

```c
EVENT_TRACE_REGISTER_NAME( funcid, funcname );
```
Function names are typically longer than 8 characters, so the \texttt{TRACE()} macro above cannot be used for recording function entries and exits. Therefore, \textsc{PAT} assigns each function a unique number. Numbers between 10,000 and 39,999 are reserved for system functions, so numbers lower than 10,000 and larger or equal 40,000 are available for user routines. With the macro \texttt{EVENT_TRACE_REGISTER_NAME()} the user can assign the function \texttt{funcname} the identification \texttt{funcid}. This information is stored in the event trace and is later used when the event trace is read (e.g., by the \textsc{PAT} command \texttt{trace}). The user must ensure that the function identifier \texttt{funcid} is unique and in the valid range. Each function must be registered exactly once (e.g., in the program main function or in a wrapper function for main).

\begin{verbatim}
EVENT_TRACE( funcid, type, p1, p2, p3, p4 );
\end{verbatim}

Write a trace record of type \texttt{type} for function identifier \texttt{funcid} as well as four 1-word parameters \texttt{p1} to \texttt{p4}. These parameters can be used to record additional event specific or other important information, e.g., the value of local or global variables. The type parameter is used to distinguish between different typical event types. \textsc{PAT} provides the following predefined event type constants:

\begin{verbatim}
PAT_TRACE_ENTRY          Function entry
PAT_TRACE_EXIT           Function exit
PAT_TRACE_SHORT_ENTRY    Function entry (no parameters recorded)
PAT_TRACE_SHORT_EXIT     Function exit (no parameters recorded)
PAT_TRACE GENERIC_ENTRY  Function entry (used by \textsc{PAT}'s generic wrapper)
PAT_TRACE GENERIC_EXIT   Function exit (used by \textsc{PAT}'s generic wrapper)
PAT_TRACE_SEND           Message Send
PAT_TRACE_RECV           Message Receive
PAT_TRACE_PUT            Message Put
PAT_TRACE GET            Message Get
PAT_TRACE RPUT           Message Remote Put
PAT_TRACE RGBT           Message Remote Get
PAT_TRACE USR            used by the \texttt{TRACED()} and \texttt{CTRACE()} macros
\end{verbatim}

The event types related to messages are used by the \textsc{PAT} message passing toolkit wrapper library which is described in Section 3.1.

\begin{verbatim}
EVENT_TRACE_GATE
\end{verbatim}

This is a macro for the internal flag variable used to indicate whether tracing is active. It is set and unset by the macros \texttt{TRACE_ON()} and \texttt{TRACE_OFF()}. User-defined wrapper functions should only execute tracing related code if \texttt{EVENT_TRACE_GATE} is true.

In addition, four additional macros are defined for tracing function entries and exits (which are very common). They work like calls to \texttt{EVENT_TRACE()} with specific parameters pre-set but are more efficient:

\begin{verbatim}
EVENT_TRACE_ENTRY( funcid, a1, a2, a3, a4 );
EVENT_TRACE_EXIT( funcid, a1, a2, a3, a4 );
\end{verbatim}
Like calling \texttt{EVENT\_TRACE()} with type \texttt{PAT\_TRACE\_ENTRY} and \texttt{PAT\_TRACE\_EXIT}.

\begin{verbatim}
EVENT\_TRACE\_SHORT\_ENTRY( funcid );
EVENT\_TRACE\_SHORT\_EXIT( funcid );
\end{verbatim}

Like calling \texttt{EVENT\_TRACE()} with \texttt{PAT\_TRACE\_SHORT\_ENTRY} and \texttt{PAT\_TRACE\_SHORT\_EXIT}, that is, additional parameters will not be recorded. This macro should always be used if the overhead of recording parameters is not necessary.

2.2.3 User Control of Tracing

Finally, there are macros which allow the user to control the tracing feature to some degree. The following is a list of the macros currently available in PAT:

\begin{verbatim}
TRACE\_ON();  \hspace{1cm} \text{Turns on tracing for the executing PE. This is the default value.}
TRACE\_OFF(); \hspace{1cm} \text{Turns off tracing for the executing PE.}
TRACE\_LIMIT(limit); \hspace{1cm} \text{ Stops the writing of trace entries to disk from the executing PE when the value specified by limit has been reached. Entering a value of -1 specifies unlimited trace entries. This is the default value.}
\end{verbatim}

Through the use of the macros \texttt{TRACE\_ON()} and \texttt{TRACE\_OFF()}, the user can restrict tracing to specific parts or executions of the program. \texttt{TRACE\_LIMIT()} is useful for large programs when disk space for trace data is limited.

There are also a few environment variables that can be used to control the tracing feature:

\begin{verbatim}
PAT\_PROFILE\_WHILE\_TRACING
    \text{Used by the libpat.a library to control whether or not profiling and HW performance counter information gathering will be done during execution.}

    OFF \hspace{1cm} \text{Specifies that no performance counter gathering and no profiling of the executable being run will take place.}
    ON \hspace{1cm} \text{Specifies that performance counter gathering and profiling of the executable being run will take place. This is the default setting.}

PAT\_NUM\_TRACE\_ENTRIES
    \text{Used by the libpat.a library to change the default number of trace entries in the internal PAT trace buffer. The default number of entries is 8192.}

PAT\_TRACE\_LIMIT
    \text{Used by the libpat.a library to change the default limit for the number of trace entries to be written to the trace file. The default value is -1 which indicates an unlimited number of entries.}
\end{verbatim}

2.3 Implementation of Tracing

The following paragraphs describe the implementation of the tracing functions provided by PAT. The instrumentation, tracing runtime system, and PAT Information File (PIF) access interface will be discussed.
2.3.1 Instrumentation

The user interface for the instrumentation for tracing actually has three possibilities. The user can use the batch type interface by specifying the `-B filename` option in which the file specified by `filename` contains the list of function names to be instrumented for tracing. The second interface is the interactive interface. By using the `-I` option, the user brings PAT up in interactive mode. Here, by using the `instrace` command, function names to be instrumented for tracing can be entered. The last interface option is through the graphical user interface, where a window allows the user to enter a list of function names.

PAT takes the function name entered and searches its list of functions to try to make a match on the input name with the internal function names it acquired by examining the executable. Once the match is made, PAT reads in the instructions corresponding to the function from the executable. PAT then saves the function's first instruction and replaces it with a branch to a procedure instruction whose target is a initialization code template. This code template looks like this:

```
.psect $trace_wrapper@code,code
$trace_wrapper::

; Make a copy of the callers stack, leaving space for
; the register save area for this routine.

subq fp,sp,t0         ; get size of callers stack
bis sp,sp,t1          ; save current stack pointer
subq sp,128,t2
subq t2,128,t2        ; get pointer to base of new stack frame
bis t0,t0,t4          ; save size
subq t2,t0,t2         ; point to top of new stack

st_loop:
  ldq t3,0(t1)         ; load word of old stack
  stq t3,0(t2)         ; store word into new stack
  addq t1,8,t1         ; increment to next word of old stack
  addq t2,8,t2         ; increment to next word of new stack
  subq t4,8,t4         ; decrement counter
  bne t4,st_loop       ; loop until done

subq sp,128,sp        ;
subq sp,128,sp        ; set sp past reg. save area
bis sp,sp,fp          ; set frame pointer to base of stack frame
subq sp,t0,sp         ; set stack pointer to top of stack frame

save_all_regs
bis zero,#0,a0        ; PATCH 0: function number
bis zero, #0, t0      ; upper 8 bits of function number
sll t0, #8, t0        ; shift them up 8 bits
bis t0, a0, a0        ; or the upper 8 and lower 8 together
bis zero,zero,zero    ; PATCH 1: entry trace routine
restore_all_regs
bis r1,r1,r1          ; PATCH 2: instruction zero of traced func.
jsr ra,(t7)           ; save the return value
save_return
restore_all_regs
bis zero,1,a0         ; PATCH 3: function number
bis zero, #0, t0      ; upper 8 bits of function number
sll t0, #8, t0        ; shift them up 8 bits
bis t0, a0, a0        ; or the upper 8 and lower 8 together
bis r3,r3,r3          ; PATCH 4: exit trace routine
this will also restore the return value
restore_all_regs
addq fp,128,fp        ; point to top of old stack
addq fp,128,fp        ; set stack pointer to top of old stack frame
```
There are five instructions which are then updated with the function specific information. These "patch" instructions have comments indicating that they require a patch value. The function specific information includes the function number which is an internal integer value corresponding to the function being instrumented, the first instruction of the function being instrumented which is executed in the tracer wrapper template, a branch to the entry trace wrapper function, and a branch to the exit trace wrapper function.

Once the instrumentation is complete and the new executable file has been written, the new executable image can be executed. Upon execution, the function that was instrumented gets called from somewhere in the program. Control is transferred immediately to the wrapper template, because the first instruction was patched to branch to the instrumented function's specific wrapper template. There some housekeeping work is done and control is passed to the entry wrapper routine and subsequently to the exit wrapper routine. The entry and exit wrapper routines as stated above can record various key items of information about the function that was instrumented.

### 2.3.2 Tracing Runtime System

The PAT runtime tracing system is responsible for recording the generated event records and storing them in an event trace file. To keep the influence on program behavior low, the event records are first stored in an internal trace buffer in memory on each PE and only written to disk on request or when the buffer is full. The default number of entries is 8192. As described in Section 2.2.3, the user can change the buffer size by setting the environment variable `PAT_NUM_TRACE_ENTRIES` or by using the macro `TRACE_LIMIT()` in his program. In addition, with the environment variable `PAT_TRACE_LIMIT` the size of the event trace can be controlled. The event trace data is written to the PAT Information File (PIF) like all the other performance data gathered by PAT. The name of the PIF file is the name of the generating executable plus the suffix ".pif". The final trace records are written to the PIF file when the instrumented application terminates. At termination time control is given to a PAT library exit processing routine which flushes the trace buffer to the PIF file in addition to turning off profiling and performance counter gathering. This exit routine also finalizes other information contained in the PIF file. Sorting of the trace information is delayed until the trace records are accessed for display, either through the X window display or the command line or interactive options to list the records to standard output.

### 2.3.3 PAT Information File (PIF) Access API

PAT allows the user to read the contents of an event trace either through the command line option `-t` or the `trace` command in interactive mode. The trace contents are shown one event record per line. The event records are displayed in chronological order (i.e., sorted according to the recorded timestamp).

In addition, PAT provides a C function interface to access the event trace contained in a PIF file. This interface can be used to write "custom" trace analysis tools or to implement a conversion tool which converts PIF traces into another trace format used by other trace analysis tools (e.g., the ALOG format used by Upshot [7]).

The interface is available by including `libpif.h` and linking the program with `-lpif`. It provides functions to read the event records one by one in chronological order as shown in the following small program fragment:
#include <libpif.h>

`pif_fd_t *th; /* PIF trace handle */`
`trace_record_t *ev; /* storage for 1 event record */`

`th = pif_open("progl.pif"); /* open PIF trace "progl.pif" */`
`if (th == NULL) {
    fprintf(stderr, "Error opening 'progl.pif'
    exit(1);
}

while ( pif_trace_read(th, &ev) ) {
    /* do something with event record stored in "ev" here */
    /* ... */
}

pif_close(th);

Opening a PIF event trace with `pif_open` returns a trace handle of type `pif_fd_t *` or `NULL` if an error occurred when opening the event trace. Everytime the function `pif_trace_read` is called, it reads the next event record from the trace and copies it's contents into a struct of type `trace_record_t` which has to be passed as a second argument. If there are no more event records, the routine returns 0. `trace_record_t` is declared the following way:

```c
typedef struct {
    int    func_id;  /* function identifier */
    int    type;    /* record type like described in */
    int    rtc;     /* timestamp (Real Time Clock) */
    int    pe;      /* Logical PE number */
    int    th_id;   /* Thread Id */
    int    args[NARGS]; /* Additional Arguments */
} trace_record_t;
```

The thread identifier is only of importance if multi-threaded programs are traced, otherwise it will always have the value 1. The function identifiers can be converted to function names with the help of mapping tables included in the PIF file. There are four tables which can be accessed by the following functions:

```c
typedef struct { /* struct describing a single function table entry */
    int id;     /* function id */
    char name[]; /* corresponding function name */
} trace_map;

trace_map *pif_get_mpi_map( pif_fd_t* th ); /* MPI function table */
trace_map *pif_get_pvm_map( pif_fd_t* th ); /* PVM function table */
trace_map *pif_get_sma_map( pif_fd_t* th ); /* SHMEM function table */
trace_map *pif_get_usr_map( pif_fd_t* th ); /* User function table */
```

As an example, the following code shows how to access and traverse the MPI function table and print it's contents to stdout:

```c
int i;
trace_map *mpi_map;

mpi_map = pif_get_mpi_map(th);
if (mpi_map == NULL) {
    printf("no mpi map\n");
} else {
    for (i=0; mpi_map[i].name[0]; i++) {
```
printf("id = %d name=%s\n", mpi_map[i].id, mpi_map[i].name);
}

Finally, there are two additional functions for getting the number of ticks per second for the
timestamp value (pif_get_hertz()) and for getting the number of PEs on which the traced
program was executed (pif_get_npes()).

3 Analyzing Message Passing Programs with PAT

In using the event trace features of PAT as delivered by Silicon Graphics/Cray Research for
analyzing message passing programs, it is possible to record all entries and exits to a specified set
of user and message passing functions. However, to make the analysis really useful, two things are
missing:

- No information about messages is recorded. It would be nice to know the source and
destination PE, the amount of data, and the tag and communicator (if available) for each
message transferred. This can be done by writing special wrapper functions for each function
of MPI, PVM, and SHMEM which is related to sending and receiving messages.
Forschungszentrum Jülich did exactly this and the implementation is described in what
follows.

- In addition, typical event traces from "real" message passing programs are large, so special
event trace browsers are needed to analyze them. VAMPIR, which is such an event browser
and was developed by Forschungszentrum Jülich, is described in Section 4.

3.1 Implementation of Wrappers for MPI, PVM, and SHMEM

We implemented all necessary special wrappers (as described in Section 2.2.1) for all point-to-point
and collective communication routines in the current message passing toolkit (MPT 1.2), i.e. for
MPI, PVM, and SHMEM and for both the C and Fortran interface. Fortunately for us, the Fortran
interface of the MPI and PVM functions just call their corresponding C counterparts which reduced
the amount of wrapper functions to be implemented quite a bit. Still, wrappers for 167 functions
had to be designed and implemented.

We used the following abstract event model to record the necessary information about message
transfers for MPI and PVM: functions sending messages record the following information

SEND tag destination_pe message_length communicator

while routines receiving data capture

RECV tag source_pe message_length communicator

PE numbers are always recorded as logical PE numbers (as defined by the CRAY T3E intrinsic
_my_pe). This means that the wrapper functions need to transform MPI’s rank/communicator pair
and PVM’s task identifier into logical PE numbers. The message length is recorded in bytes; this
means that the amount of data sent has to be calculated out of the size of the data type and the
number of elements transferred.

The wrappers for SHMEM routines need to be different because they implement one-sided
communications and don’t have tags and communicators. For get operations, we record the
following events on the target_pe:

RPUT source_pe message_length
... execute get ...
GET source_pe message_length

If necessary, this can be "interpreted" as a message passing send/receive pair the following way (note the switch of source_pe and target_pe when transforming the GET into a RECV):

on source_pe: SEND <no_tag> dest=target_pe message_length <no_communicator>
on target_pe: RECV <no_tag> src=source_pe message_length <no_communicator>

Wrappers for put operations are implemented accordingly:

PUT target_pe message_length
... execute put ...
RGET target_pe message_length

Again this can be "viewed" as a message passing send/receive the following way:

on source_pe: SEND <no_tag> dest=target_pe message_length <no_communicator>
on target_pe: RECV <no_tag> src=source_pe message_length <no_communicator>

Based on this design, implementation of the wrappers was straight-forward. However, the following "problems" needed to be considered:

- If a MPI or PVM receive function is called with a wildcard value for the source PE and/or the message tag, the "real" source PE and tag need to be determined and recorded.

- For MPI nonblocking and persistent communication, the wrappers need to keep track of MPI requests.

- For MPI nonblocking receives, it is impossible for a wrapper function to determine the exact time when the message was received. The best thing that can be done is to generate the necessary RECV event in the first successful MPI_Halt*, MPI_Test*, or MPI_Irecv function (worst case approximation).

- For SHMEM put operations, it is impossible for a wrapper function to determine the exact time when the data is stored in remote memory. The wrapper routines therefore write the PUT event record when the SHMEM put functions returned (best case approximation).

- If the message received with pvm_recev, pvm_nrecv, or pvm_trrecv is longer than PvmDataMax the real transfer is done during unpacking (pvm_upk*). In this case, the wrappers determine in what unpack routine the last portion of the message is unpacked and record the RECV event there.

- For collective communications we would like to record every message sent and received just like we did for the point-to-point communication routines. However, we don't know how the collective communication routines implement the collective communication (e.g., as a binary tree). So, the best thing we can do, is to record the "logical" pattern of the communication. We classified each collective communication routine as 1-to-N, N-to-1, N-to-N, or N-to-1-to-N and the corresponding wrappers generate all necessary SEND and RECV events. Because this is not the "real" thing, it is only done if the user sets the environment variable
Forschungszentrum Jülich made the source code of all wrapper routines available to Silicon Graphics/Cray Research which in turn distributes them as libwrapper.a together with PAT.

3.2 Examples

The combination of PAT’s new object instrumentation features together with the complete message passing toolkit function wrapper library (described in the last section) allow for analysis of a CRAY T3E message passing program in a much greater detail than it would be possible with other currently available performance tools (e.g., AIMS [2] or Pablo [3]). By using the default wrappers for MPI or PVM functions, but the special wrappers for SHMEM routines, we can look “inside” the former and can determine how they are implemented the necessary communication protocol in terms of the latter. Two small examples will demonstrate this.

As a first example, we want to analyze the behavior of sending a larger array of integers from one processor to another using the "standard" PVM functions. Here is the necessary source code in C:

```c
#include <pvm3.h>

int main() {
   int buf[2048];
   int tag = 4711;

   if (_my_pe() == 0) {
      /* initialize buf with useful values here ... */

      pvm_initsend(PvmDataRaw);
      pvm_pkint(buf, 2048, 1);
      pvm_send(1, tag);
   } else {

      pvm_recv(0, tag);
      pvm_upkint(buf, 1024, 1);
      pvm_upkint(buf+1024, 1024, 1);

      /* do something with the values received */
   }
}
```

In the code above, on PE 0, we first initialize the PVM send buffer with `pvm_initsend`, then use `pvm_pkint` to pack a 2048 integer array into the buffer, and finally send it with `pvm_send` to PE 1 with the tag 4711. On PE 1, we use `pvm_recv` to receive the message and (to make it a little bit more complex) unpack the receive buffer in two steps using `pvm_upkint`.

After compiling the program and linking it with PAT’s runtime library as well as the SHMEM special wrappers, we use PAT to instrument the `main` function as well as all PVM and SHMEM routines. Executing the program on two PEs results in an event trace which contents (nicely formatted) looks like this:

```
Time[ms]    PE 0          PE 1
-----------  ------        ----
37.350695    -> main
37.352144
```

43
Several things are interesting: the only information exchanged between `pvm_send` and `pvm_recv` is some header and synchronization information (done with two calls to `shmem_put64`) because the length of the message is greater than `PVMDATA_MAX`. The actual data is then transferred during the unpacking (`pvm_upkint`) using `shmem_get64`. This way unnecessary buffer copies can be avoided. However, the first time, two `shmem_get` calls are used to copy the 8192 bytes (1024 * sizeof(int)), while the second time the data is transferred with one call. This is because for the first 4096 bytes a pre-allocated, PVM internal message buffer is used, while the rest of the message is stored in a dynamically allocated buffer by PVM. The size of the pre-allocated internal message buffers is determined by `PVMMAX_PACK`. Finally, the last `pvm_upkint` uses `shmem_put64` again to signal PE 0 that the message transfer is complete.

In the second example, we want to analyze the message protocol of `MPI_Broadcast`. For generating the trace, we use the following small Fortran90 MPI test program which broadcasts four integers with value 17 from PE 0 to all other processors involved in the program.

```fortran
PROGRAM main
INCLUDE 'mpif.h'

INTEGER :: master = 0
INTEGER, DIMENSION(4) :: buf = (/ 42, 42, 42, 42 /)
INTEGER id, numprocs, sbuf, err

CALL MPI_Init(err)
CALL MPI_Comm_size(MPI_COMM_WORLD, numprocs, err)
CALL MPI_Comm_rank(MPI_COMM_WORLD, id, err)

IF (id .EQ. master) buf = (/ 17, 17, 17, 17 /)

CALL MPI_Bcast(buf, 4, MPI_INTEGER, master, MPI_COMM_WORLD, err)

IF (buf(2) .NE. 17) WRITE(0,*),'CPU: ', id, ' Bcast Error: got ', buf(2)

CALL MPI_Finalize(err)
END
```

Generating the event trace is done the same way as in the last example. However, to keep the event trace simple, we only instrument `MAIN`, `MPI_Bcast`, and the `SHMEM` functions. Figure 1 shows the logical behavior of `MPI_Bcast` when executing on 8 processors.
Initially each target PE communicates with its source PE by sending 64 bytes of configuration data (message buffer address, message size, ...).

Careful investigation of the picture reveals that the data is distributed from PE 0 using a binary tree: first 0 sends to 1, then 0 and 1 send to 2 and 3 in parallel, and finally 0, 1, 2, and 3 simultaneously transfer the data to 4, 5, 6, and 7. Each individual transfer is actually implemented with three SHMEM calls:

1. `shmem_put32` moving the actual data (4 integers here)
2. `shmem_put64` of 8 bytes setting a flag indicating to the target PE that the message transfer completed
3. and surprisingly, a `shmem_put64` of 8 bytes is used again inside the target PE to reset this flag. SHMEM is used here to bypass the cache.

Analyzing small examples like the ones described here can easily be done by looking at user readable listings of the trace file (like produced by the option -e of PAT). However, investigating real user applications which produce large trace files requires more sophisticated tools. One such tool, VAMPiR, is described in the next section.

4 Analyzing Event Traces with VAMPiR

VAMPiR (Visualization and Analysis of MPI Resources) is an event trace analysis tool [4] which was developed by the Central Institute for Applied Mathematics of Forschungszentrum Jülich and now is commercially distributed by a German company named PALLAS. For more information see http://www.pallas.de/pages/vampir.htm.
VAMPIR's main application area is the analysis of parallel programs based on the message passing paradigm but it also has been successfully used in other areas (e.g., for SVM-Fortran traces to analyze shared virtual memory page transfer behavior [5] or to analyze CRAY T3E usage based on accounting data). In the version which is commercially available, VAMPIR has three components:

- The VAMPIR tool itself is a graphical event trace browser implemented for the X11 Window system using the Motif toolkit. It is available for any major UNIX platform.

- The VAMPIR runtime library provides an API for collecting, buffering, and generating event traces as well as a set of wrapper routines for the most commonly used MPI communication routines which record message traffic in the event trace.

- In order to observe functions or subroutines in the user program, their entry and exit has to be instrumented by inserting calls to the VAMPIR runtime library. Observing message passing functions is handled by linking the program with the VAMPIR wrapper function library.

VAMPIR comes with a source instrumenter for ANSI Fortran77. Programs written in other programming languages (e.g., Fortran90, C or C++) have to be instrumented manually. Fortunately, now with the new version of PAT and the complete MPT wrapper library as described above, the situation for CRAY T3E systems was vastly improved. Forschungszentrum Jülich also implemented an event trace conversion program which reads a PAT event trace using the API described in Section 2.3.3 and generates a new event trace in VAMPIR trace format. This trace converter will be distributed with future versions of VAMPIR.

VAMPIR provides a wide variety of graphical displays to analyze the recorded event traces (see Figure 2):

- The dynamic behavior of the program can be analyzed by time line diagrams for either the whole program or a selected set of nodes (see top window in Figure 2). By default, the displays show the whole event trace, but the user can zoom-in to any arbitrary region of the trace. Also, the user can change the display style of the lines representing messages based on their tag/communicator or the length. This way, message traffic of different modules or libraries can easily be visually separated.

- The parallelism display shows the number of nodes in each function group over time. This allows for easy location of specific parts of the program, e.g., parts with heavy message traffic or I/O.

- VAMPIR also provides a large number of statistical displays. It calculates how often each function or group of functions was called and the time spent in each. Message statistics show the number of messages sent, and the minimum, maximum, sum, and average length or transfer rate between any two nodes. The statistics can be displayed as bar charts, histograms, or textual tables (see bottom 3 windows in Figure 2).

A very useful feature of VAMPIR is that the statistic displays can be linked to the selected portion of the program execution in time line diagrams. By this, statistics can be calculated for any arbitrary, user selectable part of the program execution.

- If the instrumenter/runtime library provides the necessary information in the event trace header, the information provided by VAMPIR can be related back to source code. VAMPIR
provides a source code and a call graph display to show selected functions or the location of
the send and the receive of a selected message.

![Graphical Displays of VAMPIR](image)

**Figure 2.** Graphical Displays of VAMPIR

In addition, VAMPIR provides a feature called grouping. Large parallel programs consist of several
dozens or even hundreds of functions. To ease the analysis of such complex programs, VAMPIR
arranges the functions into groups, e.g., user functions, MPI routines, I/O routines, and so on. The
user can control/change the assignment of functions to groups and can also define new groups.

## 5 Conclusion

In this paper, we described the functionality of PAT, the Performance Analysis Tool for the CRAY
T3E system, especially the recently added event trace features. These were implemented in a
fruitful cooperation between Silicon Graphics/Cray Research and Forschungszentrum Jülich. Two
properties make PAT a powerful and unique event trace analysis tool:

1. It's object instrumentation allows for analysis of programs written in any language currently
available on the T3E (Fortran, C, and C++) and is extensible to other languages. Other currently available performance tools (e.g., AIMS [2] or Pablo [3]) use source code instrumentation and only support Fortran77 and/or C. Paradyn [6] also can instrument object code but is not available for the CRAY T3E. In addition, PAT can easily instrument mixed language applications and can instrument system or 3rd-party library code, features which are not available from any other tool for the CRAY T3E. Finally, the program doesn’t have to be recompiled for instrumentation, a big advantage especially for large programs.

2. PAT allows for implementation of user-defined wrappers for any user or system function even if the source code is not available. Only the function prototype must be known. Forschungszentrum Jülich used this feature to implement a complete wrapper library for all message passing functions currently available on the T3E (MPI, PVM, and SHMEM). This library is distributed with PAT. Again, other tools do not provide this feature. There are tools for analyzing MPI or PVM programs, but not for SHMEM or using all of them in one program.

In two examples we showed how this new functionality can be used to analyze message passing programs in a detail which was not possible before. Finally, we described VAMPIR, a very powerful and highly configurable event trace browser also developed by Forschungszentrum Jülich. It displays trace files in a variety of graphical views, and provides flexible filter and statistical operations that condense the displayed information to a manageable amount. Rapid zooming and instantaneous redraw allow for identifying and focusing on the time interval of interest.

The combination of these two tools, PAT and VAMPIR, plus the availability of Cray’s second tool Apprentice, gives the CRAY T3E the most complete tool set for performance analysis currently available for message passing programming.

6 Acknowledgements

The authors want to thank Alan Mayer of Silicon Graphics/Cray Research who implemented the first prototype of the object instrumentation and event tracing features of PAT. Bernd Mohr also wants to thank Bart Theelen of Eindhoven University of Technology who implemented the wrapper functions for the collective communication functions while he did his practical studies at Forschungszentrum Jülich and especially Reiner Vogelsang of the Cray Research office Jülich who was instrumental in understanding the details of message passing communication on the CRAY T3E.

7 References

[1] CRAY Research, 
*Introducing the MPP Apprentice Tool*,
CRAY Manual IN-2511, 1994

*Performance Measurement, Visualization and Modeling of Parallel and Distributed Programs using the AIMS Toolkit*,
Software Practice & Experience, Vol. 25, No. 4, pages 429-461, April 1995

_VAMPiR: Visualization and Analysis of MPI Resources_, 
Supercomputer 63, Vol. 12, No. 1, pp. 69-80, 1996

_Performance Analysis for SVM-Fortran with OPAL_, 
Proceedings Int. Conf. on Parallel and Distributed Processing Techniques 
and Applications (PDPTA’95), Athens, Georgia, pp. 561-570, 1995

R.B. Irvin, K. Kunelthiapadam, K.L. Karavanic, and T. Newhall, 
_The Paradyn Parallel Performance Measurement Tools_, 

[7] V. Herrarte and E. Lusk, 
_Studyng Parallel Program Behavior with Upshot_, 
Technical Report ANL-91/15, Mathematics and Computer Science Division, 
Argonne National Laboratory, August 1991.

Copyright © 1998 Silicon Graphics/Cray Research and Forschungszentrum Jülich, ZAM
Synchronization on Cray-T3E Virtual Shared Memory*

Miltos D. Grammatikakis, Helidon Dollani†, Stefan Liesche‡
Institute for Informatics,
University of Hildesheim
31141 Hildesheim, Germany
{mdgramma,dollani,liesche}@informatik.uni-hildesheim.de

Abstract

We consider algorithms for implementing mutual exclusion on the Cray-T3E virtual shared memory using various atomic operations. Our implementations of Anderson's and MCS Lock minimize network contention and dramatically improve performance for any system with more than two processors. Improvements over the Cray shmem_lock library functions are above three orders of magnitude on a 64-processor T3E-900. Our results hold for both small, and large critical sections, and make the possibility of implementing concurrent data structures on the Cray-T3E virtual shared memory a viable one.

1 Introduction

In low-level (virtual) shared memory programming, processors can access local and remote memory using read, write, and atomic operations. Synchronization is needed to avoid data consistency problems, called race conditions [14]. Data races arise when threads or parallel processes access shared variables and

- at least one access is a write operation, while
- there is no synchronization as to the order of accesses to the variable.

Synchronization routines are classified into either blocking or busy-wait. While with blocking waiting processes are deferred, thereby freeing the CPU to perform other operations, with busy-wait processes repeatedly access shared variables to determine when they may proceed. Busy-wait synchronization is a better approach when scheduling overhead exceeds busy-wait time, when CPU resources are not needed by other threads, or when rescheduling is inappropriate, as in a single-user system.

Busy-wait synchronization includes acquire.lock and release.lock mechanisms for mutual exclusion, post and wait semaphores for producer-consumer problems, and non-fuzzy barriers for bulk-synchronous computations [11]. Although hardware barriers are common in NUMA supercomputers, such as the TMC/CM-5 and Cray-T3D/T3E, Fujitsu and Hitachi MPP systems, locks and special semaphores (condition variables) are generally implemented using shared memory library calls. In general, a lock is more restrictive than a semaphore. Not only it is a binary object, but also a release lock operation can only be performed by the last processor to acquire the lock. Performance of synchronization routines benefits from the availability of hardware supported atomic operations, such as Test&Set, Fetch&Increment, Fetch&Add, Fetch&Store, Compare&Swap, and Load&Linked & Store. Conditional [13].

Two of the most important and widely used busy-wait synchronization mechanisms are spin locks, and (non-fuzzy) barriers. Spin locks provide means for implementing mutual exclusion and avoiding data races. They ensure that only one processor may access (and possibly modify) a shared data structure at any given time. Spin locks protect application critical sections, and may be executed an enormous number of times with concurrent data structures, such as priority queues, FIFO queues, e.g. in OS kernel processing routines, or fault tolerance recovery techniques. Apart from concurrent implementations of linked lists and sometimes queues which can be done in a lock-free fashion, locks

---

*We acknowledge partial support from KFA, Jülich - project K2710000.
†Visiting from the Department of Mathematics, University of Tirana, Tirana, Albania. Supported by a DAAD short-term research scholarship (A/07/12720).
‡Research partially supported by EPCC/TRACS through an ICM scholarship.
are generally needed in most asynchronous implementations of shared memory algorithms, e.g., a concurrent implementation of a Gaussian solver of $AX = B$. Barriers provide mechanisms for ensuring that no process advances beyond a particular point in computation until all processes have arrived at that point. Barriers are also used for avoiding race conditions, and may be a major contributor to run time.

A concurrent algorithm is said to be lock free if it always guarantees that some process will complete in a finite number of steps. It is wait free if it guarantees that each process will complete in a finite number of steps. A hierarchy of atomic principles enabling the simulation of lower class primitives in a wait-free manner has been developed [7]. The hierarchy leads to interesting emulations of atomic operations and higher level synchronization principles.

In this paper we examine several implementations of locks on Cray-T3E virtual shared memory systems. In Section 2, we discuss the implementation of spin locks, including Test&Set-based lock, Ticket Lock, Anderson's Lock, and Mellor-Crummey and Scott (MCS) Lock for virtual shared memory. In Section 3 we evaluate spin locks, by computing the average time for acquire and release lock operations. We concentrate on both small, and large critical sections. We show that Anderson's and MCS Lock minimize network contention, and offer improved performance over all other locks for any system with more than two processors. In fact, MCS Lock is 1000 to 10,000 times faster than the Cray-T3E shmem.test.lock & shmem.set.lock instructions for 64 processors.

2 Virtual Shared Memory on Cray-T3E

Nowadays most non-uniform memory access (NUMA) supercomputers, including Cray-T3E, NEC SX-4, and to a lesser degree Hitachi SR-2201 and Fujitsu VPP700, support virtual shared memory.

The Cray-T3E implements a logically shared address space over physically distributed memories (with up to 2GB per processor). Each processing element (PE) consists of a DEC Alpha 21164 processor connected to a "shell", consisting of a control chip, a router chip and local memory. The T3E improves the memory bandwidth of the Alpha microprocessor by providing a large set (512 user plus 128 system) of external registers (called E-registers). These registers are used to support remote communications and synchronization. Because of the large number of E-registers, remote reads and writes are highly pipelined. The operations that read memory into E-registers and write E-registers to memory are called respectively, shmem.get and shmem.put. On the Cray-T3E both operations have similar performance. However, due to adaptive packet routing, successive calls to shmem.put are not guaranteed to arrive in order. Special calls to shmem.qiet/shmem_fence are needed to verify that all previous puts from/to a particular PE have executed in order.

On the Alpha µP a Load.Linked & Store.Conditional implements an atomic read-modify-write cycle; this is the only atomic operation available consistent with its RISC philosophy [3]. The Load.Linked(a) primitive returns the value $M(a)$ and sets a reservation associated with the location and the processor (but does not lock the location). A subsequent Store.Conditional(a,b) instruction checks the reservation, and either proceeds writing the value of $b$, if the value at that location has not been modified, or otherwise it fails.

The Cray-T3E provides a plethora of atomic operations on arbitrary memory locations, allowing an unlimited number of synchronization variables. The atomic operations provided by the Cray-T3E are Fetch&Inc, Fetch&Add, Compare&Swap and (masked) Swap. Notice that Fetch&Store, Compare&Swap, or Load.Linked & Store.Conditional can be used to implement Test&Set, hence Test&Set is not provided in most modern MPP systems, like the Cray-T3E.

Atomic operations on the the Cray-T3E are implemented using the Alpha µP Load.Linked & Store.Conditional primitive. The main idea of these implementations is to repeatedly execute Load.Linked & Store.Conditional cycles, until the Store.Conditional is successful, thus the operation appears to be atomic since there is no overlapping with other operations accessing the same memory location.

Barriers allow a set of participating processors to determine when all processors have signaled some event (typically reached a certain point in their execution of a program). Barriers are implemented in hardware on the Cray-T3E using combining trees.

Cache coherence is implemented on the Cray-T3E using cache-invalidate protocols. However, the T3E is not stream coherent and prefetching must be turned off to avoid data race conditions which may cause inconsistent results, program aborts, or hangs. It is also known that the T3E may reorder instructions from a given PE, e.g. local remote writes and remote read operations (WR reorder). A released consistency model has been proposed based on the Alpha microprocessor [3].

\footnote{In practice Load.Linked & Store.Conditional checks a complete cache line.}
2.1 Cray ShMem - Remote and Atomic Operations

Atomic operations are supported on all Cray MPP and PVP systems by calling Cray ShMem library functions. ShMem library calls can be used within Fortran and C/C++ programs. Since our implementations are in C only, C/C++ routines are presented.

<table>
<thead>
<tr>
<th>Generic Name</th>
<th>ShMem Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fetch&amp;Inc</td>
<td>shmem_short.finc(&amp;a,n)</td>
<td>return $M(a) = M(a) + 1$</td>
</tr>
<tr>
<td>Fetch&amp;Add</td>
<td>shmem_short.fadd(&amp;a,b,n)</td>
<td>return $M(a) = M(a) + b$</td>
</tr>
<tr>
<td>Fetch&amp;Store</td>
<td>shmem_swap(&amp;a,-1,b,n)</td>
<td>return $M(a) = M(a) + b$</td>
</tr>
</tbody>
</table>
| Compare&Swap| shmem_short.cswap(&a,b,c,n) | if $M(a) 
eq b$ then return $M(a)$ |
|             | (Masked) Swap    | else: return $M(a)$ |
| Remote Read | PEp : shmem.get(&a,b,c,n) | $M(a) = M(a)$, $M_b(b+c-1) = M(a) + b + c - 1$ |
| Remote Write| PEp : shmem.put(&a,b,c,n) | $M(a) = M(a)$, $M_b(b+c-1) = M(a) + b + c - 1$ |

Table 1: Atomic read-modify-write and remote read/write in ShMem

Atomic operations available to users and operating system programmers on the Cray-T3E are shown in Table 1. The operands are generally short (32-bit) integers; some operations allow also for other data types, such as int, long, float, and double. A definition of each operation listed in Table 1 is provided below:

**Fetch**($a,b$) operations return the memory contents $M(a)$, while storing at location $a$ the function $F(a, b)$.

**Compare&Swap**($a,b,c$) primitives atomically compare the content of memory location $M(a)$ with a replacement value $b$, and store a third value $c$ if they match; the operation also returns a condition flag indicating either success or failure.

**(masked) Swap**($a,b,c$) operations store selected bits from $c$ in $M(a)$. Selection is done by the mask $b$. It returns the previous content of $M(a)$.

**acquire_lock/release_lock**($a$) routines together provide mutual exclusion. The first call to acquire_lock($a$) returns immediately. A subsequent call returns after a release_lock($a$) corresponding to the first acquire_lock($a$) is executed. This ensures that only one process is holding the lock at a time.

Also notice, that due to adaptive routing of messages, successive calls to shmem.put are not guaranteed to arrive in order. Special calls to shmem.quit (shmem.fence) are needed to verify that all previous puts from (to) a particular PE have executed in order.

3 Mutual Exclusion on Virtual Shared Memory

In this Section, we describe five implementations of a mutual exclusion spin lock. In our busy-wait lock implementations, processors call acquire_lock whenever they desire a lock (there are no synchronization barriers involved). Acquire_lock will return only when the lock is obtained by the calling processor. A subsequent call to release_lock will make the lock available to other competing processors.

Each algorithm assumes a virtual shared memory environment that allows for atomic, remote write, and remote read operations. In our presentation, we will use the macro names: fast.finc, fadd, cswap, and swap referring to atomic operations as shown in Table 1. Furthermore, most of our lock variables will be of type short; this entails no severe limitation on the use of our routines and faster remote operations are possible.

3.1 Test&Set Lock

The simplest mutual exclusion lock employs a polling loop to access a boolean flag that indicates whether the lock is held. As the C code in Table 2 indicates, in acquire_lock each processor repeatedly
Table 2: Test&Set Lock with exponential backoff

![C++ code snippet for Test&Set Lock]

executes a Test&Set operation (via a swap) in its attempt to change the flag from false to true. The processor will then release the lock by setting the flag back to false. The main shortcoming of the Test&Set Lock is network contention for the flag, which is allocated at a given PE (usually PE0).

The total amount of network traffic caused by busy-waiting on a Test&Set Lock can be reduced by introducing a random delay at each processor between consecutive probes of the lock. The simplest method employs a constant delay; more efficient schemes adopt exponential backoff on unsuccessful probes [1]. This is also shown in Table 2.

3.2 Ticket Lock

![C++ code snippet for Ticket Lock]

In Test&Set Lock, the number of read-modify-write operations is potentially large, and each Test&Set operation would cause invalidation of all cache copies on a cache coherent system. Although every waiting processor may perform a Test&Set operation on the same flag each time, only one will actually acquire the lock. We can reduce remote cache invalidations (and possibly network traffic) using a Ticket Lock, while at the same time we ensure FIFO service, i.e. granting the lock to
processors in the order that they request for it.

A Ticket Lock consists of two counters. The request counter Ticket counts the processors that call acquire.lock, while the release counter Serve counts the times the lock has been released. A processor acquires the lock by performing

- A Fetch&Increment on Slot, a variable allocated only at PE0. Overflow of Fetch&Increment is prevented by performing a subsequent fadd to Slot
- Spinning until its request counter equals the value of the release counter.

It releases the lock by

- incrementing the release counter.

Our implementation of Ticket Lock on the Cray-T3E is shown in Table 3.

Ticket Lock causes substantial memory and network contention by polling a common location. As with Test&Set Lock, this contention can be reduced by introducing delay on each processor between consecutive probes of the lock. In this case however, exponential backoff is not a good idea. Instead proportional backoff is implemented, and the delay is made proportional to the difference between the values of the request and release counter. The constant of proportionality is the minimum time that a processor can hold the lock [11].

3.3 Anderson's Lock

```
short my_flag = -2, Lock = -1, Slot = 0, Ticket;
void acquire.lock(short *my_flag, short *Lock, short *Ticket, short *Slot, int PE, int NoPEs)
{
    short mask = MaxShort, temp, value_inv = -2;
    temp = fast.func(Slot, 0);
    *Ticket = temp % NoPEs;
    if (temp == mask)
        temp = fadd(Slot, -mask, 0);
    *my_flag = swap (Lock, PE, *Ticket);
    do {} while (*my_flag != -1);
    if (PE != *Ticket)
        shmem.short.p(Lock, value_inv,*Ticket);
    else
        *Lock = value_inv;
}
void release.lock(short *my_flag, short *Lock, short *Ticket, short *Slot, int PE, int NoPEs)
{
    short y, index_proc, value_set = -1;
    index_proc = (*Ticket+1) % NoPEs;
    y = swap (Lock, value_set, index_proc);
    if ((PE != y) && (y>0))
        shmem.short.p(my_flag, value_set, y);
    else if ((PE == y) && (y>0))
        *my_flag = value_set;
}
main: if (PE == 0) my_flag = -1; /* allow first request to acquire lock */
    acquire.lock(kmy_flag, &Lock, &Ticket, &Slot, shmem.my_pe (), shmem.uee ());
    release.lock(kmy_flag, &Lock, &Ticket, &Slot, shmem.my_pe (), shmem.uee ());
```

Table 4: Anderson's Lock

Even for Ticket Lock with proportional backoff, it is not possible to obtain an average constant number of network transactions per lock acquisition, due to the unpredictability of the length of critical sections. Anderson has proposed a locking algorithm that achieves constant network traffic on cache coherent shared memory multiprocessors that support Fetch&Increment (or Fetch&Store) operations [1]. Expanding on Anderson's idea to virtual shared memory systems, the trick is that each processor uses an atomic swap to save its address at the PE named Ticket. Then processors performing release.lock can directly inform the next processor to acquire the lock, by writing directly on its spinning variable my_flag. This idea is implemented on the Cray-T3E as shown in Table 4. Notice that, each processor spins on its local variable which can be arranged to be on a different cache line. The Fetch&Increment in Anderson's algorithm could also be avoided by using a single Fetch&Store operation (on the corresponding subfields) [6].
3.4 MCS Lock

```c
short my_flag, Lock = -1, next;

void acquire_lock(short *my_flag, short *Lock,
    short *next, int PE)
{
    short value_set = 1, value_null = -1, predecessor;
    *next = value_null;
    predecessor = swap(Lock, PE, 0);
    if (predecessor >= 0) {
        *my_flag = value_set;
        if (PE != predecessor)
            shmem.short.p(next, PE, predecessor);
        else
            *next = PE;
        do { } while (*my_flag == value_set);
    } *my_flag = value_set;
}

void release_lock(short *my_flag, short *Lock,
    short *next, int PE)
{
    short value_reset = 0, value_null = -1;

    if (*next < 0) {
        if (cswap(Lock, PE, value_null, 0) == PE)
            return;
        do { } while (*next < 0);
    }

    if (PE != *next)
        shmem.short.p(my_flag, value_reset, *next);
    else
        *my_flag = value_reset;

main: acquire_lock(&my_flag, &Lock, &next, shmem.my_pe());
release_lock(&my_flag, &Lock, &next, shmem.my_pe());
```

| Table 5: MCS Lock |

The MCS Lock, as prototyped by Mellor-Crummey and Scott, guarantees FIFO ordering of lock acquisitions, spins on local variables only, requires a small constant amount of space per lock, and works equally well on machines with and without coherent caches. We have adjusted the MCS Lock to the virtual shared memory model. The resulting code for the Cray-T3E appears in Table 5.

A Lock variable is allocated at processor 0. Its contents are either null (-1) if the lock is still available, or otherwise set equal to PE, if processor PE has acquired the lock. Each processor using the lock must allocate two queue pointers (predecessor and next) and a boolean flag (my_flag). The predecessor variable points to the previous processor requesting an acquire_lock, or is null if no PE has done so. The predecessor variable is used to notify the predecessor PE by setting its local variable next; this variable is initialized to -1 in acquire_lock. A processor issuing an acquire_lock either obtains the Lock, or spins on its local my_flag, until the Lock becomes free.

On a release_lock operation the queue pointer next is checked. If next is non-empty we pass the Lock to the next PE by setting the corresponding my_flag variable. If next is empty, a Compare&Swap enables the processor to determine whether it is the only processor in the queue. In this case, the processor simply resets Lock to -1, in a single atomic action. Otherwise, there is another processor waiting for the Lock, so we wait until this other processor sets the next pointer of this PE, so that we can inform him. This local spin in release_lock compensates for the time window between the Fetch&Store (swap) on Lock and the later assignment to next in acquire_lock.

Chains of processors holding and waiting for the lock are shown in Figure 1. In

(a) the lock is free, in
(b) PE₄ has just acquired the lock, so its my_flag equals 0, while in
(c) PE₄ is in its critical section, so its my_flag equals 1.

Upon releasing the lock PE₄ will notify the next processor (PE₂) using a remote write on its my_flag variable. Notice the corresponding next and predecessor queue pointers.

4 Performance of Lock Implementations on Cray-T3E

We now evaluate the average time each processor spends on acquire_lock and release_lock operations. The time spent in the two routines on the Cray-T3E000 is added together. In Figure 2 we show
the performance for a small synthetic critical section. For any number of processors $P \geq 2$, Anderson and MCS lock are far superior to any choice of Test\&Set or Ticket Lock. If only one or two processors are requesting a lock, and FIFO ordering of lock requests is not important, then a simple Test\&Set is the best choice. This is also true when lock is an extremely rare operation; however, in this situation, one can argue that a lock may be dropped without introducing race conditions. Furthermore, the performance of Test\&Set with exponential backoff is only slightly better than Test\&Set with no backoff. For Ticket Lock we have found proportional backoff to be generally a bad idea, especially for large critical sections.

The Cray lock, probably implemented as a Ticket Lock, since it guarantees FIFO ordering (and analysis of the assembly code indicates that PEs access a variable at a fixed PE), is extremely slow.
Figure 3: Mutual exclusion algorithms on Cray-T3E for large critical section

It is, including runtime lock validation, between 1000 to 10,000 times slower than MCS Lock for 64 processors! This would be several orders of magnitude slower on a full-scale Cray-T3E system.

Similar results for large critical sections, shown in Figure 3, verify the above observations. In this case, the critical section consists of a loop with ~1 million integer operations; large critical section, refers to the average time spent in the loop compared with the minimum time taken by an acquire_lock together with a release_lock operation.

5 Conclusions

The performance of locks is of a great importance to several problems which lie in the heart of concurrent data structures, operating systems, and fault tolerance. Our implementations of Anderson’s Lock and MCS Lock minimize internetwork and memory traffic and achieve performance at levels of several orders of magnitude better than Test&Set Lock, Ticket Lock, or Cray-T3E library shmem.lock implementations.

In several applications, alike Ticket Lock and Anderson’s Lock, there is access to centralized counter. Implementing this counter as a counting network can reduce bottlenecks. Counting networks provide a distributed mechanism for implementing Fetch&Increment, by avoiding the traffic associated with accessing a single (virtual) shared memory location [2]. Implementation of fan-in two, $O(\log^2 N)$-depth counting networks, isomorphic to Batch’s bitonic sorting network [4], or the balanced periodic sorting network [5], shows the practicality of this low-level approach [8].

Implementations of other synchronization issues, such as software barriers on the Cray T3E are interesting. For non-fuzzy barriers, we do not expect improvements, since Cray-T3E hardware barriers have almost constant runtime (2-20msc), independent of the number of PEs ($N$). This is always faster than a software barrier, whose runtime depends on the number of PEs participating in the barrier.
[11], usually as $\log_2 N$. However, fuzzy nonblocking barriers are interesting and their implementation could be useful for optimizing certain latency-bound applications.

We currently consider the effect of our lock implementations on parallel network simulation algorithms by implementing concurrent priority queues using atomic operations [12, 9]. These data structures support priority insert and delete min operations which are necessary for modeling network packet movements. Since, ~20-30% of our parallel code consists of calls to lock operations, we claim that our analysis (which will be included in the final version of this paper) will make our results even stronger [10]. We hope that improved lock/semaphore implementations in NUMA supercomputers would encourage programmers to consider concurrent, shared memory data abstraction for high performance applications.

Acknowledgments

We are greatful to Prof. Michael Scott for his helpful explanations and pointers.

References


Implementation aspects and performance of some common problems on the CRAY T3E

Matthias Linke
University of Rostock
Computing Center
matthias.linke@rz.uni-rostock.de

August 24, 1998

Abstract

The parallelization of an algorithm can be very difficult especially for novices. The main problem is the implementation of a proper data distribution and the related communication structure. Our goal is to help the programmer in this step of program development. We provide program modules that show the implementation of different communication structures using different programming environments. The interested user can download the program modules from the web-pages of the computing center. This paper gives an overview of the currently available examples, their performance behaviour and their usefulness for application programmers.

1 Introduction

The computing center at the University of Rostock runs a Cray T3E-900 since 1996. It consists of 44 DEC Alpha EV5.6 each with 128 MB main memory. The processing elements (PE) are divided into 40 Application-PEs, 3 Command-PEs and 1 Operating System-PE. So we can offer a maximum of 36 GFlops and 5 GB main memory for parallel applications. Besides using the existing software packages many users need and want to write parallel applications. It is our responsibility to support the programmer.

During the parallelization of some applications we found communication patterns that are often used. We decided to deliver the programmer a collection of modules with solutions for some common problems related to the parallelization of programs. The main problem is the implementation of a proper data distribution and the related communication structure. Our goal is to help the programmer in this step of program development. Taking simple examples the basic data distributions and communication schemes are declared. Portable implementations using MPI\(^1\) or PVM\(^2\) are given as well as system dependent implementations.

The underlying parallel system is the CRAY T3E at the University of Rostock and its linsma-library. The modules are implemented in C, yet. The interested user can download the modules from the web-pages of the Computing Center. The sample solutions come with a shell script to build executable programs. The user controls the building of the codes by several parameters. Together with the modules comes a detailed description of the implemented solutions and their performance. The following sections are an essential summary of those papers.

The first example is a parallel two-dimensional cellular automaton. Considering this algorithm one can find data distributions and communications schemes that are very common. The two-dimensional board can be divided row-wise, column-wise or block-wise. Every distribution implies the communication of a different part of the board between the processors. Given the distribution, the programming language and a parallel programming environment the performance of the communication depends strongly on the way it is implemented. In this paper

\(^1\)Message Passing Interface
\(^2\)Parallel Virtual Machine
some different optimization techniques and their impact on the performance are described. It is shown that given a distribution other than row-wise one can implement MPI programs with the performance of an implementation using the Cray specific libm4a-library. Another common problem is the distribution of independent computation steps. Given a loop that defines the number of computations the performance of the algorithm depends on its ability to avoid gaps between the end of one computation step and the start of the next. The different possibilities of MPI, PVM and the libm4a-library to control the computation are described. It is shown that a program running in the distributed PVM mode behaves like a program implemented using libm4a-routines. Performance results are given for an application from the field of environmental simulation.

2 Parallelization of a cellular automaton

2.1 Basic concepts

The concept of cellular automata was introduced by von Neumann in 1948. Cellular automata are an alternative way of formulating physical theories. The problem space is divided into small uniform cells. Each cell is in a defined state. This state is computed

- locally: The state depends on the states of the cell and its neighbours.
- globally: The state of all cells is computed at the same time.

The existing automata differ in the way they initialize the board, the number of neighbours involved in and the model of the computation of the new state.

A well-known and simple cellular automaton is the Game of Life. The problem space consists of a two-dimensional board with dimensions \(dimx \times dimy\). A cell is in one of the two states

- alive, represented by 1
- dead, represented by 0.

At each iteration the new state of a cell is determined by the state of its neighbours at the previous iteration. Taking this approach we slightly changed the rules of the known algorithm. The algorithm available for the interested user initializes the board in the following way:

\[
\text{board}(i,j) = \begin{cases} 
1 & \text{i=dimx/2 or j=dimy/2} \\
0 & \text{else}
\end{cases}
\]

The new state of a cell is determined only by the the state of its nearest neighbours. The whole board is shifted in the four directions North, East, West and South. The algorithm performs periodic boundary conditions. The new state of a cell depends on the number of alive neighbours. The rules for the evolution of the cellular automaton are:

\[
\text{board}(i,j) = \begin{cases} 
1 & \sum \text{neighbours} = 2 \\
0 & \text{else}
\end{cases}
\]

There are basically four important program steps:

1. Initialization of the board
   
   Each process determines the global position of its part of the board and performs the initialization according to the given rules.

2. Shifting the board
   
   The board is shifted in the four directions. The edges of the local parts must be shifted between the processes. The following sections show some possibilities to optimize this communication and their impact on the performance.

3. Determining the new state
   
   Each process computes the states of the cells of its part of the board. The new state is compared with the previous state to check for a final state. A global communication is necessary to compare all local states. The algorithm reaches a final state for some dimensions of the grid.
4. Printing the final state
   If the final state is reached the board is printed into a file. Communication is needed.
   The later sections discuss some optimization possibilities for this program step. Since this
   step not only depends on the communication but also on several other parameters like the
   current load of the Operating System-PE this paper includes no dedicated performance
   assertions for it.

It can be seen that the given algorithm implies a SPMD programming model\(^3\). So the first
problem is to distribute the board onto the given number of processors. This distribution can
be row-wise, column-wise or block-wise. The chosen distribution is crucial for the performance
of the algorithm. The following sections describe the communication schemes for each distribu-
tion that are implemented in our modules using MPI and the Cray-specific libmam-library
and compare their performance. It was our goal to provide the interested programmer diferent
implementations of the given algorithm for a given distribution although it is known that some
of them are not efficient. But the modules give some solutions how to optimize their inefficient
examples. So every programmer can use our modules to optimise his own special algorithm.
In this paper not all routines implemented in the modules are described, only the basic and
main optimizations. Basically there are two ways to improve the communication time. A very
common optimization is the overlapping of computation and communication. This technique
is not described especially. The second way is to minimize the number of communications to
avoid the overhead.

2.2 Row-wise Distribution

A row-wise distribution implies the communication of whole rows of the board. Each process
has to communicate the edges in the North and the South of its board. Since the algorithm
performs periodic boundary conditions this ends in a one-dimensional torus communication
scheme. A row-wise distribution is the natural way of dividing the board since the rows of
the board are stored contiguously in the memory (C-implementation). So the information that
must be given to the communication system is:

- start address of the row to communicate
- number of elements in that row (dimension \(dimy\))

Shifting the board with MPI

The rows are sent in a ring-like manner. The first process sends its row to the next and waits
for the receive of the related row from the previous process. All other processes wait for the
receive. Then they send their rows. This step has to be done twice, once for the edge in the
North and once for the edge in the South of the board.

A first approach will be the call of the standard communication routines \(MPI\).Send and
\(MPI\).Recv. Since the behaviour of these routines depends on the implementation of MPI one
has to program the ring structure to be portable. That means that basically only two processes
perform the communication at the same time, one sends and one receives. It is clear that a lack
of performance may occur because all rows are independent and therefore all processes may
perform this communication step at the same time. A simple way to achieve this behaviour is
to call the nonblocking communication routines \(MPI\).Isend and \(MPI\).Irecv. Now an additional
call is needed to ensure the completion of the communication, \(MPI\).Waitall. The performance
of both implementations shows no essential differences (fig.1a). The implementation of the
standard communication routines seems to realize a behaviour similar to the optimized version.

Shifting the board with libmam-routines

Opposite to the communication with MPI the libmam-routines allow the reading and writing
of data out of and into remote memory directly. All processes can access all memories at
the the same time. No process has to wait for receives. A process can write the rows into
the related memory using \(shm\).put or it can read the rows from the remote memory using
\(shm\).get. The difference between both routines is the necessary synchronisation to ensure
a correct completion of this communication. Looking at the performance one will find no dif-
fferences between both realizations (fig.1b). Comparing the results to MPI one can see the
much better performance of the shm-solutions.

\(^3\) Single Program Multiple Data
Printing the board using MPI
There are two basic ways to print the final state into a file. Each process prints its part of the board. A synchronisation of the processes is necessary to print the parts in the correct order. It is clear that only one process does the printing and the others are waiting for the completion. So there is a lot of idle time. To overcome this problem one could determine one process to print the final state. Then all processes have to send their parts to that process. Each process transfers its part in one communication call. The parts could be sent one after the other and printed directly into the file. But then it is pretty the same as above because only one process sends, its part is printed and all others must wait until their part is needed. So if memory is not crucial it could be better to use a temporary array to assemble all the parts before printing them. Now each process may send its part separately from the other processes. A more elegant way of collecting the parts provides the routine MPI.Gather. Another advantage is that all processes are done with their communication at the same time. This could be important if the printing is done not only at the end of the program but also at several other states.

Printing the board using libsmo-routines
Since it is not performant to let each process print its part this possibility is not considered here. The collection of the parts can be done in two ways. The printing process reads them from the remote memories or the processes write them into the memory of the printing process. Comparing a realization with a temporary field to a version where the parts are printed immediately proves that the best performance is reached when all processes write the parts into a temporary array in the memory of the printing process.

2.3 Column-wise Distribution
Contrary to the row-wise distribution the processes need to communicate a column of the board. The elements of a column are not arranged contiguously in the memory. The stride between two elements is the size of the part of the dimension dimy each process is working on.

Shifting the board with MPI
The first attempt to implement a transfer of the columns is similar to the realization described above. Because of the storage alignment the elements have to communicate separately. So there is a loop over the dimension dimz that sends and receives only one element per iteration, that means the whole communication overhead for every element. There are two approaches to solve this problem.

The elements of a column are assembled in a temporary array. This array is sent between the processes. Then the elements have to be stored in the corresponding column. The communication consists now of a single send and receive like described above. But there are additional
computations and memory. Another possibility is the use of the derived datatypes in MPI. First a derived datatype vector is declared and committed. One has to declare the number of elements and the stride between them. This datatype is used in the communication of a column. Opposite to the first approach there are no additional computations and no temporary field. Fig. 2a proves that the use of a derived datatype performs slightly better than the use of a temporary array. And it is the most elegant way to solve this problem.

Shifting the board with libsm-a-routines
The approaches given above can also be implemented using libsm-a-routines. The routines shmem_get and shmem_put perform a strided data transfer between the remote memories. Fig. 2b shows that the strided communication also performs better than the use of a temporary array. But it can be seen that there are no significant differences between the optimized versions and the basic realization.

Fig. 2: Shifting a column-wise distributed board (1500x1500)

Printing the board using MPI
Because of the storage management of C it is not easy to transfer the whole part at once. All the problems of communicating strided data mentioned in the shifting section apply. But MPI provides the derived datatypes, a means to achieve a behaviour as described in the row-wise distribution. The different parts are built in two steps. First a row is defined as MPI_Type contiguous. Then the rows are combined using MPI_Type_vector. The main parameters are the number of rows (dimx) and the stride between them (dimy/number of processes). Then the whole block is sent in one single communication call. Performance tests show that this is the fastest way to collect the different parts in a temporary array.

Printing the board using libsm-a-routines
There is no appropriate shmem-routine that allows the transfer of the whole part. So the only difference lies in the use of get or put when collecting the parts into a temporary array. And it is obvious that the best performance is achieved when all processes write their parts into the remote memory simultaneously.

2.4 Block-wise Distribution
A block-wise distribution combines the communications of rows and columns. Therefore the optimizations described in the previous sections can be applied and the performance gains shown in fig. 3 confirm the assertions. There are differences between the non-performant block-wise row-oriented and column-oriented implementation. But those differences almost vanish with the optimization.
Fig. 3: Shifting a block-wise row- (a,b) and column-oriented (c,d) distributed board (1500x1500)

**Printing the board using MPI**

Since the transfer of the parts is a combination of a row-wise and a column-wise distribution all aspects mentioned before apply. There are some problems using MPI.Gatherv because the sizes of the parts often differ. The use of dynamic groups may improve the communication time. Combining those processes with the same size of its parts into a dynamic group leads to less communication calls.

**Printing the board using libisma-routines**

There are no special optimizations for a block-wise distribution.

2.5 Speedup and scalability

The printing of the board can have enormous consequences for the runtime. The advantages of the fast libisma-communication may disappear because of delays when accessing the OS-PE or the file system. Since the maximum number of processes for parallel applications on our CRAY T3E system is limited to 16 only the results up to this limit are shown. The fig 4a and 4b compare the speedup of solutions without any optimization (-noopt) with the best optimization for each distribution (-opt).

One could achieve an almost linear speedup with the libisma-routines and a row-wise distribution. It is the fastest solution because of the minimum communication overhead. The speedup of MPI programs improves dramatically especially for column- and block-wise distributions. But not only the speedup improves also the scalability of the programs. So if
one wants to write portable programs and has no row-wise distribution he must think about optimizing the communication.

There are no real performance gains for the libsma-realizations.

A very interesting point is that besides the row-wise distribution the speedup and scalability of MPI- and libsma-realizations are nearly identical. That means one may implement a portable solution that behaves like a Cray-specific realization.

Fig. 4: Speedup (1500x1500)
3 Distributing independent computation steps

In many application fields it is necessary to run the same program several times. The reasons could be:

- Different parameters must be checked.
- The results depend on random numbers.

The individual runs are often independent of each other. That leads to a loop over the number of program runs called a *pool of tasks*. The main problem is the distribution of the individual runs especially when there are differences in the runtimes. The optimization goal is to avoid gaps between the end of one run and the start of the next. Taking an example from the field of environmental simulation different ways to achieve the desired behaviour are shown. The application is strongly based on random numbers. The program should run 100 times to get statistic reliable results. There are no dependencies between the individual runs. The implementation of a solution for this problem depends on the underlying parallel programming environment. The next sections will describe portable realizations using MPI and PVM and a Cray specific solution as well. Advantages and disadvantages are shown. The figures show the runtime of the computation and the delays between them for three example runs on 8 processors.

3.1 MPI realization

Since MPI realizes a SPMD programming model all processes do the same computations. But one process is considered to control the computation. It holds the loop over the program runs. The main optimization parameter is the point where the distribution takes place. There are two possibilities:

- after a computation run is finished,
- during the run.

The first realization is easier to implement. Let the computation be done in a subroutine then all processes perform the same subroutine. Only the main program differs. But it is obvious that there is an enormous performance loss if the individual computation runs differ heavily. Either the control process waits for the end of an other process or the other processes have to wait for the end of the control process to start the next run (see fig.5).

![Performance of MPI realization - distribution after computation](image)

**Fig. 5:** Performance of MPI realization - distribution after computation

To overcome this problem one has to move the distribution of the computations into the subroutine. Now it is necessary to choose significant points in the algorithm where the control should occur. On one hand the overall runtime behaviour should be improved. On the other hand the computation of the control process should not be interrupted too often. The right
balance depends on the actual application. At the chosen point the control process checks if there is an idle process using MPI_Lprobe. If there is a process and there are more computations in the loop the control process starts the next one. This is done until there is no idle process at this time. Then the control process continues its computation until the next check should be performed. Now the performance of the application depends on the checking intervals. Fig. 6 show that the delays diminish with a decreased checking interval. Compared to the performance in fig. 5 an improvement is achieved.

![Graphs showing performance improvement](image)

Fig. 6: Performance of MPI realization - distribution during computation

3.2 Realization with libsma-routines

Opposite to the implementation described above there is no need for a controlling process. The pool of tasks is held in the local memory of one processor. Each process has to read and change the actual content of this pool without waiting for any process. The only problem is to ensure that only one process is permitted to do this at a given time. So the reading and changing has to be done in a critical region that is saved by locks. Each process checks whether it can lock this region (shmem_set_lock), does the necessary communication and frees the region (shmem_clear_lock). Since there is no need for synchronization of the processes there are no delays of computation (fig. 7a).

3.3 PVM realization

A PVM program can be executed in two different runtime modes. A program in the stand-alone mode behaves like a MPI program and is portable only with some slight changes. Implementing the application for the distributed mode one can achieve a libsma-like behavior and a fully portable program. The application is split into two separate parts:

- Master - controls the pool of tasks
- Slave - does the computation

The master is a sequential process. It is started on a command processor. The master is responsible for starting a number of slaves (pvm_spawn) and supplies them with work. It does no computation. The master only waits for the end of a slave to start the next computation. Hence it uses almost no resources on the command processor. As shown in fig. 7b the PVM application behaves like the libsma realization. There are no delays between two consecutive runs. But opposite to the libsma-implementation the application is portable. The master and/or the slaves can run on a heterogeneous virtual machine. So it is fairly easy to run the whole application on a number of parallel computers. The environmental application mentioned
above was executed on a virtual machine consisted of the Cray T3E in Rostock and the Cray T3E at the Konrad Zuse Zentrum in Berlin. An ongoing work at our computing center deals with the distribution of a more advanced version of this application on the Cray T3E systems in Rostock, Berlin and Hannover.

Fig. 7: Performance of libsm and PVM realization

4 Conclusions

Novices in the field of parallel programming need help to develop efficient applications. Some common problems occur very often in this process. Many problems involve the distribution of a two-dimensional array. Subject to this distribution different communication schemes must be implemented. The performance of this communication depends on the underlying parallel programming environment and its ability to optimize inconvenient data layouts. This paper shows some solutions and their performance for the problem of transferring rows and columns of a two-dimensional array. Taking a given distribution and parallel programming environment different ways to improve the communication time are described. It is shown that besides a rowwise distribution there is no real difference in the performance between a MPI realization and a libsm-solution. A second example deals with the distribution of independent computation tasks. The main problem is to avoid gaps between the end of one computation and the start of the next. It can be stated that it is more difficult to solve this problem using MPI than with libsm-routines. The performance of a MPI solution depends on the control point in the algorithm and the interval between two controls. There is no delay between consecutive computations in a libsm-realization. The same behaviour can be achieved using PVM. The PVM application must run in the distributed mode. An advantage of this approach is the portability of the application. The individual computations can be distributed onto a variety of computers. So there is no problem in running this application on a number of Cray T3E systems simultaneously. Future work at the Computing Center at the University of Rostock deals with studies of the usefulness of connecting the systems in Rostock, Berlin and Hannover to run an environmental application based on the given PVM approach. The realizations described in this paper are available on the web pages of our Computing Center. The web pages are intended to be a working base for developers of parallel programs. Hence the content of these pages are always evolving. The URL is:

http://www.uni-rostock.de/rz/hware/cray/libsmain.html

Unfortunately it is in German yet. It is our goal to provide it also in English.
Parallel Priority Queues on Cray-T3E

Miltos D. Grammatikakis and Stefan Liesche
Institute for Informatics,
University of Hildesheim
31141 Hildesheim, Germany
{mdgramma,liesche}@informatik.uni-hildesheim.de

Abstract

We examine the design, implementation, and experimental analysis of parallel priority queues for network simulation. We consider: a) distributed splay trees using MPI, b) concurrent heaps using shared memory atomic locks, and c) a new, more general concurrent data structure based on distributed sorted lists, which is designed to provide dynamically balanced work allocation (with automatic or manual control) and efficient use of shared memory resources. We evaluate performance for all three data structures on a Cray-T3E900 system at KFA-Jülich. Our comparisons are based on simulations of single buffers, and a 64 x 64 packet switch which supports multicasting. In all implementations, PEs monitor traffic at their preassigned input/output ports, while priority queue elements are distributed across the Cray-T3E virtual shared memory. Our experiments with up to 60,000 packets and 2 to 64 PEs indicate that concurrent priority queues are 5-10 times faster and more scalable than distributed ones, even though the distributed implementation provides asynchronous insertions and deletions. Both concurrent implementations have comparable performance, while our new data structure uses less memory and has been further optimized. The data structures already form a parallel library. Although our concurrent implementations use the Cray-T3E ShMem, portability can be derived from Open-MP or MPI-2 standard libraries, which will provide support for one-way communication and shared memory lock mechanisms.

Keywords: Concurrent data structure, Cray-T3E, data race, distributed data structure, memory lock, priority queue, parallel simulation, virtual shared memory.

1 Introduction

In 1964, Knuth considered an abstract data type (ADT), called priority queue [15]. It consists of a set of items \( H \). Each item is assigned a priority value chosen from a totally ordered domain. A lower value means a higher priority and vice-versa. Two ADT operations Insert and DeleteMin are provided: a) \( \text{Insert}(a,H) \) inserts a new item into the set \( (H = H \cup a) \), and b) \( a = \text{DeleteMin}(H) \) returns the item with highest priority and removes it from the set \( (a = \min H; H = H - a) \).

A parallel priority queue can perform these ADT operations concurrently. The main difference of priority queues compared with other parallel applications, such as distributed databases, or dictionary machines, comes from the centralized control necessary for computing the node with the highest priority, during DeleteMin. This requirement creates hot spots and bottlenecks in this node's neighborhood. Furthermore, since applications may frequently change the data structure size, some sort of load balancing is necessary for efficient implementation.

Priority queues are common in parallel and distributed applications. They represent the optimal data structure for many problems:

- Parallel branch and bound algorithms for decision making in finance, or games are implemented using priority queues.

- In numerical iterative schemes, such as graph algorithms, priority queues are used to compute the "next" item to access; e.g. both Kruskal's algorithm for the minimal spanning tree problem (arising for example in network design) and Dijkstra's algorithm for the shortest path problem are based on DeleteMin operations.

- Heuristics for NP-complete problems, e.g. bin packing and traveling salesman require backtracking with next visited elements chosen according to size or distance metrics.
- Pattern matching or data compression techniques for quality assurance, and pattern recognition for identification purposes can use priority queues for processing signs ordered by their relative significance.

- In fuzzy search on large data sets, priority queues can be used for providing statistical information, e.g. hit lists from internet search engines, and reliability ratings for internet sites.

- The core of job schedulers in operating systems for thread or process management, and event schedulers in production and simulation applications are priority queues.

Heaps are ideally suited to implement sequential priority queues [29]. A heap is a binary tree with the property that the values stored at any node are always greater than or equal to values stored at both child nodes (if they exist). All levels, except the last, are completely filled. Several parallel priority queues, such as calendar queues, have been proposed and their parallel (PRAM) complexity has been theoretically examined [4, 5, 14, 23, 24].

The experimental evaluation of parallel priority queues consists of both distributed [18], and also concurrent implementations [13, 22]. All previous implementations have focused on different parallel systems, and thus information on the relative performance of parallel priority queues is very limited.

In this study, we aim at resolving this issue, by developing and evaluating three different parallel priority queue data structures on the same parallel platform, a Cray-T3E MPP system available to us from KFA-Jülich.

All data structures are implemented using libraries that support the programming model for which the data structure is proposed. This ensures that all data structures perform as good as possible, without any overheads induced by emulating programming models within our implementations. The concurrent data structures form a software library for further research and development on parallel simulation, targeted at:

- simulating packet communication switches.
- evaluating flow control protocols, possibly including priorities.
- simulating large communication networks, e.g. multibutterflies.

The first implementation uses MPI to develop a distributed priority queue structure (DPQ). This data structure was introduced by Mans in 1996, and targeted point-to-point multiprocessors that do not share memory [18]. A distributed data structure is allocated to different processing elements (PEs) and may be accessed by many processes simultaneously [21]. A special root PE provides the global view of a heap, with each node consisting of a splay tree of items. Insertions and deletions invoke always the root PE.

The next two implementations use the Cray ShMem virtual shared memory library to implement concurrent priority queues. In our concurrent implementations shared memory (ShMem) locks provide atomicity during parallel update operations, while barriers also help in avoiding certain race conditions.

The second implementation is based on a concurrent priority queue introduced for uniform shared memory (UMA systems) by Rau and Kumar in 1988 [22] and improved by Hunt, Michael, Parthasarathy and Scott in 1996 [13]. The logical data structure is a heap, i.e. a binary tree. Our randomly hashed concurrent priority queue (RCPQ) provides a virtual shared memory implementation. Data for each priority queue node is stored in arrays at predetermined randomly hashed PE locations.

The third implementation is the most general one. We design and implement a new dynamically balanced concurrent data structure (BCPQ) which can also play the role of a priority queue. This data structure is based on distributed circular lists. It is designed to provide balanced work allocation and efficient use of the virtual shared memory resources. Both automatic, and user control of the load balancing phase is provided.

Our comparisons of the parallel data structures are based on simulations of single buffers and also a 64 x 64 crossbar switch. The switch uses multicasting and different PEs model traffic at a number of input and output ports. Buffers are modeled by priority queues, and buffer elements are distributed across all PEs.

Our experiments with up to 60,000 packets and 2 to 64 PEs indicate that, concurrent priority queues are 5-10 times faster than distributed ones, even though DPQ implements asynchronous insertions and deletions. Furthermore, both concurrent implementations are flexible to use and have comparable performance, while RCPQ uses more memory. They can be further improved by using faster lock implementations, such as MCS lock, and they can easily be ported into MPI-2 or Open-MP, thus providing platform-wide portability.
The Cray-T3E MPP system and its virtual shared memory (ShMem) environment are described in Section 2. The data structure implementations on the Cray-T3E are provided in Section 3. Performance evaluations are provided in Section 4. Finally, several remarks and extensions are discussed in Section 5.

2 Cray-T3E and Virtual Shared Memory

Nowadays most non-uniform memory access (NUMA) supercomputers, including Cray-T3E, NEC SX-4, and to a lesser degree Hitachi SR-2201 and Fujitsu VPP700, support virtual shared memory.

The Cray-T3E implements a logically shared address space over physically distributed memories (with up to 2GB per processor). Each processing element (PE) consists of a DEC Alpha 21164 processor connected to a "shell", consisting of a control chip, a router chip and local memory. The T3E improves the memory bandwidth of the Alpha microprocessor by providing a large set (512 user plus 128 system) of external registers (called E-registers). These registers are used to support remote communications and synchronization. Because of the large number of E-registers, remote reads and writes are highly pipelined. The operations that read memory into E-registers and write E-registers to memory are called respectively, shmem.get and shmem.put. Both operations on the Cray-T3E have similar performance.

On the Alpha µP a Load.Linked & Store.Conditioned implements an atomic read-modify-write cycle; this is the only atomic operation available consistent with its RISC philosophy [3]. The Load.Linked(a) primitive returns the value M(a) and sets a reservation associated with the location and the processor (but does not lock the location). A subsequent Store.Conditioned(a, b) instruction checks the reservation, and either succeeds writing the value of b, if the value at that location has not been modified, or otherwise it fails.

The Cray-T3E provides a plethora of atomic operations on arbitrary memory locations, allowing an unlimited number of synchronization variables. The atomic operations provided by the Cray-T3E are Fetch&Inc, Fetch&Add, Compare&Swap and (masked) Swap. Fetch&Store, Compare&Swap, or Load.Linked & Store.Conditioned can be used to implement Test&Set, hence Test&Set is not provided in most modern MPP systems, like the Cray-T3E.

Atomic operations on the the Cray-T3E are implemented using the Alpha µP Load.Linked & Store.Conditioned primitive. The main idea of these implementations is to repeatedly execute Load.Linked & Store.Conditioned cycles, until the Store.Conditioned is successful, thus the operation appears to be atomic since there is no overlapping with other operations accessing the same memory location1.

Barriers allow a set of participating processors to determine when all processors have signaled some event (typically reached a certain point in their execution of a program). Barriers are implemented in hardware on the Cray-T3E using combining trees.

Cache coherence is implemented on the Cray-T3E using cache-invalidate protocols. However, the T3E is not stream coherent and prefetching must be turned off to avoid data race conditions which may cause inconsistent results, program aborts, or hangs. It is also known that the T3E may reorder instructions from a given PE, e.g. local remote writes and remote read operations (WR reorder). A released consistency model has been proposed based on the Alpha microprocessor [3].

2.1 Cray ShMem - Remote and Atomic Operations

Atomic operations are supported on all Cray MPP and PVP systems by calling Cray ShMem library functions. ShMem library calls can be used within Fortran and C/C++ programs. Since our implementations are in C only C/C++ routines are presented.

Atomic operations available to users and operating system programmers on the Cray-T3E are shown in Table 1. The operands are usually short (32-bit) integers; some operations allow also for other data types, such as int, long, float, and double. A definition of each operation listed in Table 1 is provided below:

- Fetch.Φ(a, b) operations return the memory contents M(a), while storing at location a the function Φ(M(x), b).
- Compare&Swap(a, b, c) primitives atomically compare the content of memory location M(a) with a replacement value b, and store a third value c if they match; the operation also returns a condition flag indicating either success, or failure.

---

1 In practice Load.Linked & Store.Conditioned checks a complete cache line.
<table>
<thead>
<tr>
<th>Generic Name</th>
<th>ShMem Function</th>
<th>Description</th>
</tr>
</thead>
</table>
| Fetch&Inc          | `shmem_short.finc(&a, n)` | return $M_n(a) < M_n(a) = M_n(a) + 1 >  
| Fetch&Add          | `shmem_short.fadd(&a, b, n)`    | return $M_n(a) < M_n(a) = M_n(a) + b >  
| Fetch&Store        | `shmem_swap(&a, -1, b, n)`      | return $M_n(a) < M_n(a) = b >  
| Compare&Swap       | `shmem_short.csparse(&a, b, c, n)` | if $M_n(a) \neq b : \text{return } M_n(a)  
| (Masked) Swap      | `shmem_short.mswap(&a, c, b, n)` | else : return $M_n(a) < M_n(a) = c >  
|                    |                             | return $M_n(a) < M_n(a) = c$,  
|                    |                             | $\forall i \in \{0, \ldots, 31\} : b_i = 1$  
| Remote Read        | PE : `shmem.get(&a, b, c, n)` | $< M_p(a) = M_p(a) ; \ldots ;  
|                    |                             | $M_p(a + c - 1) = M_p(a + c - 1) >  
| Remote Write       | PE : `shmem.put(&a, b, c, n)` | $< M_n(a) = M_p(b) ; \ldots ;  
|                    |                             | $M_n(a + c - 1) = M_p(b + c - 1) >  

Table 1: Atomic read-modify-write and remote read/write in ShMem

(masked) Swap($a, b, c$) operations store selected bits from $c$ in $M(a)$. Selection is done by the mask $b$. It returns the previous content of $M(a)$.

acquire_lock/release_lock($a$) routines together provide mutual exclusion. The first call to acquire_lock($a$) returns immediately. A subsequent call returns after a release_lock($a$) corresponding to the first acquire_lock($a$) is executed. This ensures that only one process is holding the lock at a time.

Also notice that, due to adaptive routing of messages, successive calls to shmem_put are not guaranteed to arrive in order. Special calls to `shmem_quiet (shmem_fence)` are needed to verify that all previous puts from/to a particular PE have executed in order. Synchronization barriers, which are hardware implemented on Cray-T3E, can also be used, but one must be careful in causing deadlock problems.

3 Parallel Priority Queues

In our presentation, minimum item (maximum item) refer to the local item with the highest (respectively, lowest) priority.

3.1 Distributed Priority Queue

The priority structure among PEs is based on a d-ary heap, or a binomial tree. For efficiency, each PE keeps its local elements ordered using a splay tree, a self-adjusting form of a binary search tree [26].

The DPQ structure, shown in Figure 1, retains the strict priority ordering of sequential heap algorithms. The distributed heap condition implies that elements at any PE have always higher priorities than their children. To preserve this condition both ADT operations start at the root PE, i.e., centralized control is needed, and proceed in top-down fashion.

Insert Operation During an Insert the invoked PE sends the new item and transfers control to the root PE. Subsequently, the root computes the next PEp in round-robin fashion, and initiates an insertion along the path to PEp. PEs on this path react by successively inserting the new item into their own splay tree, and pushing down the maximum item. PEp will finally insert the received item into its local splay tree, thus increasing its working load. If the insertion to PEp violates locally the heap condition, PEp initiates heapification; this causes further item exchanges down the heap, until the distributed heap condition is satisfied.

DeleteMin Operation A DeleteMin request also starts at the root PE. The root PE returns the minimum item from its own splay tree. If the priority queue is empty the request is queued until an Insert operation is performed. Otherwise, the root computes the next PEp in round-robin fashion in reverse order. Along the path from PEp to root each PE sends its own minimum to its parent, thus eventually decreasing the load of PEp. Upon receiving an item from a child the parent PE has to initiate heapification if the received item has a larger value than any children item.

Splay trees speed up sequences of these operations. The splaying heuristic expects that operations in a sequence often perform in the same part of the tree. Thus, the tree is transformed (splayed) after each operation. The accessed item is moved to the root of the tree and the depth of each item along
the access path becomes smaller. *Splay trees* can perform worst-case sequences of operations as fast as balanced trees [26].

In the DPQ, PEs never access the same data simultaneously, thus no synchronization operations are needed. If the data size is large, concurrency is limited only by the number of PEs ($P$). However, the DPQ does not provide a uniform PE workload. Although on the average each node contains the same number of elements, PEs at the top of the heap are accessed more often (see Figure 1). Moreover, the root PE must maintain all ADT operations and provide a (relatively prime) step for generating a round-robin sequence which allows consecutive ADT operations to proceed on different tree branches.

![Figure 1: The DPQ data structure as binary heap](image)

3.1.1 MPI Implementation of the Distributed Priority Queue

The DPQ has been implemented by Mans [18]. He kindly gave us the right to experiment and modify the DPQ code for our simulation experiments and comparisons.

Previous experiments have shown that the *d-ary heap* performs better than a *binary* DPQ [18]. Furthermore, performance tests on the Cray-T3E, which has a 3D torus topology have shown that a *d-ary heap* with degree $d = 5$ has the best performance. Thus, we used a 5-ary heap in all our experiments.

The DPQ implementation consists of five layers. DPQ procedures can not always be assigned to a particular layer, some are distributed among several layers.

- The lowest layer implements local lists as splay trees and provides functions *InsertSplay*, *DeleteMin*, *DeleteMax*, *FindMin*, and *FindMax* to access local lists in an implementation independent way.

- The *virtual topology layer* provides functions *Parent*, *Son$_i$* ($i = 1, \ldots, 5$) and *Step* that abstract from physical interconnection topology and provide the virtual 5-ary heap topology. *Step* returns an increment for the round-robin access sequence, which helps in balancing the number of elements across PEs. *Step* is either 1, if the number of elements in the priority queue is less than the number of PEs, or the relative prime number $p$, $\frac{P}{4} < p < P$.

- The *request queue layer* implements a FIFO queue to store pending *DeleteMin* operations from the *priority queue layer*. *InsertQ* stores a request into the FIFO, while *DeleteQ* returns the oldest stored request.

- ADT functions *Insert*, *DeleteMin* and *initQueue* are provided by the *priority queue layer*. This is the main ADT layer, and the only one using MPI communication operations. All PE sends are non-blocking to avoid deadlocks, while receives are blocking. MPI also provides a tag for each message which helps PEs to distinguish between different messages. Thus, all PEs (and
in particular the root PE) provide a central message server which reacts depending on the tag of a received message, and invokes local operations, or further propagate appropriate messages.

- From the application layer, applications calling the DPQ data structure are implemented.

### 3.2 Randomly Hashed Concurrent Priority Queue (RCPQ)

To simplify the presentation of our virtual shared memory implementations, **items** and **nodes** are defined. **Items** consist of data elements in the set $H$, together with special attributes required by each implementation. **Nodes** refer to actual PE positions where items are located.

![Diagram of RCPQ data structure](image)

**Figure 2:** The RCPQ data structure

The concurrent priority queue (CPQ) data structure has been proposed for a uniform shared memory environment [13]. The logical CPQ data structure is a binary heap; nodes are arranged in a complete binary tree (see Figure 2). The distributed heap condition implies that elements at any PE have always higher priorities than their children; furthermore, all PEs except the leaf PEs are nonempty. In contrast to most heap definitions, CPQ leafl need not be filled in left to right order.

Our **randomly hashed concurrent priority queue (RCPQ)** provides a virtual shared memory implementation of CPQ, by allocating priority queue nodes at predetermined randomly selected locations. The RCPQ data structure uses symmetric code distribution, and follows an asynchronous MIMD model. PEs perform local and remote operations on the shared data structure concurrently. To avoid inconsistent data or deadlocks, virtual shared memory locks and barriers protect data elements from being simultaneously modified. ADT operations for the RCPQ data structure follow closely the corresponding CPQ operations.

While concurrent operations are performed in the heap, parts of the RCPQ data structure may become inconsistent. To avoid locking the whole data structure, and still allow ADT operations to proceed without deadlocks in natural (opposite) order, items in the heap are tagged into either consistent, or transient states. A consistent state indicates an item that does not violate the heap condition. A transient state indicates an item that may violate the heap condition; this may happen while items are moved by an ADT operation performed on the heap. Transient states are indicated by storing the process identifier ($pid$) of the process moving the item. Value **Empty** is used to indicate a node that currently does not hold any item.

**Insert Operation** The Insert operation starts at a leaf and traverses the tree bottom-up. To reduce network network contention during lock acquisitions, leaf nodes are accessed in shuffled order during concurrent insert operations. This order, alike in the DPQ algorithm, allows consecutive insertions (or deletions) to traverse different subtrees. In this case, the shuffle counter uses a simple bit reversal technique to generate the access sequence. For example, for the third level of the heap, the shuffle counter would generate the following binary access sequence:
A new item is initially stored in the first free node computed (at the spot) by the shuffle counter. From here, the item is moved up level by level. During each move from the parent node, and then the child node are locked. When both nodes are successfully locked and their tags indicate consistent data, item priorities are compared. If the child has a higher priority item, items are swapped. At the end of this step both locks are released. The new item climbs up the tree, step by step, as long as the priority of the item at its parent node is lower than its own priority. When the heap condition is restored, the insert operation is complete.

**DeleteMin Operation** The `DeleteMin` operation uses the shuffle counter in reverse order, to locate the last nonempty node in the heap. It exchanges item `last` stored at the last nonempty heap position with the minimum item `r` stored at the root; to maintain consistency corresponding nodes are locked. Once items are exchanged, the last node is marked empty and its lock is released. The root node remains locked, since its item may violate the heap condition. During heapification the node holding `last` remains locked. At each step, children are locked and their priorities are compared with that of `last`. If the heap condition is violated, item `last` is swapped with the minimum child item. Only the new node holding `last` remains locked, and `DeleteMin` proceeds with the next step. `DeleteMin` completes when item `last` stops to move. Then, the lock of the node holding item `last` is released and `DeleteMin` returns item `r`.

Both ADT operations lock tree items. Since `Insert` proceeds top-down and `DeleteMin` bottom-up, cyclic deadlocks can possibly occur. To avoid this problem, previous algorithms have either locked the whole heap data structure, or made `DeleteMin` also proceed in top-down fashion [22]. However, both changes significantly limit concurrency [13]. The RCPQ algorithm increases concurrency, and avoids cyclic deadlocks by locking items top-down for both ADT operations [22]. While insert operations proceed bottom up, they always lock first the parent, and then the child node.

The RCPQ data structure is highly parallel. Concurrency is only limited by the number of items in the heap \((N)\). Consecutive inserts traverse disjoint paths to the root, thus congestion ideally occurs only at the upper levels of the heap. With an infinite number of PEs (or \(P = O(N)\)), \(O(N)\) ADT operations can perform simultaneously on the heap. Insert operations lock at most two items, while `DeleteMin` operations lock between one and three items at a time.

With the nonsaturated case \(P < N\), our randomly hashed virtual memory scheme allocates heap nodes to random PEs. As the heap size \(N\) increases, with high probability our nondeterministic heap allocation minimizes remote memory accesses for ADT operations. On certain architectures, a more complex distribution which takes into account parent-child neighborhood patterns could minimize latency during heapification. However, on most modern systems, like the Cray-T3E which use adaptive message routing, deterministic allocation of PE resources would add considerable analytical system modeling complexity and has many possible pitfalls.

### 3.2.1 ShMem Implementation of RCPQ

In a virtual shared memory programming model, the heap is randomly distributed among all PEs. An index list, generated during heap initialization at one PE and subsequently broadcast to all PEs, maps each heap node to the PE that stores this node. This index enables each PE to remotely access any element in the heap. At PE0 the index list also provides a lock for each node.

Our RCPQ implementation consists of three software layers: the lower `data access layer`, the middle `data organization layer`, and the upper `application layer`. Function calls are shown by connecting the bottom (south) of the box representing the caller routine to the top (north) of the box respective to the called routine. No recursive calls are involved.

- The lower layer uses ShMem functions and local memory operations to abstract access to the distributed data structure. Functions `setItem` and `getItem` provide remote or local access to entire items (including priority values, tags, and data). Functions `getData` and `setData` provide remote or local access to the data field.

- The `data organization layer` can now access both items, and data elements as in a physical shared memory environment. This layer implements the RCPQ ADT functions `initQueue`, `Insert`, and `DeleteMin`.

- At the `application layer` parallel applications using `Insert` and `DeleteMin` can be developed.
3.2.2 MPI Implementation of RCPQ

The next version of the MPI library (MPI-2) provides one-sided communications along with synchronization routines [19]. MPI-2 routines such as `MPI_Put`, `MPI_Get`, `MPI_Win_Lock` and `MPI_Win_Unlock` support a shared memory programming paradigm. Our ShMem implementation can become portable by using the MPI-2 library. At present, there is no MPI-2 implementation on the Cray-T3E, but implementations on various platforms will be available soon.

3.3 Dynamically Balanced Concurrent Priority Queue (BCPQ)

Our concurrent implementation focuses on virtual shared memory NUMA supercomputers, focusing on the Cray-T3E. In NUMA systems, access to remote data is much slower than access to local data, and lock operations can be time consuming [9, 20]. The aim of our new dynamic data structure implementation is to reduce remote data access and lower the number of locks needed to maintain data consistency. The data structure could easily be adapted on UMA systems, supporting physical shared memory, by modifying (or eliminating) the lower layer remote calls (see layer description in Section 2.3).

The BCPQ data structure is essentially a dynamically distributed sorted list. PE₀ stores items with the highest priority. Subsequent items are stored at PE₁,...,PE₀−₁ in increasing order (see Figure 4). For simplicity, we refer to the list stored at PEᵢ as list i. Thus, BCPQ operations are based on binary searching distributed sorted lists.

**Insert Operation** The Insert operation starts with a binary search for the local list, in which the new item must be inserted. The range of nonempty PEs [0, PE.max] is binary searched either remotely, or locally. For local searching each PE needs the minimum value (maximum priority value) from every other PE; this can be accomplished with a multidest broadcast.

A remote binary search proceeds as follows. Initially, list m, in the middle of the search list is locked, and the priority of the new item is compared with the minimum and maximum priorities in this list. If the new priority is between these priorities, the target list is m. Otherwise, the lock is released and the search list is halved into two new sublists. If the new priority is smaller than the minimum priority of list m, the search proceeds with the right sublist, otherwise, it continues with the left sublist. The same procedure is repeated until the target list is found. Special care is needed to to avoid an infinite loop if the new priority falls between two lists.
After the PE binary search, a second binary search locates the target position within the target list. The new item is inserted at this position and the insert operation completes by releasing the PE's lock. To avoid deadlocks, the insert operation holds the lock of at most one list at a time. Before locking a new sublist, the lock on the previous list is released.

**DeleteMin Operation** The *DeleteMin* operation first tests if the priority queue is empty, and in that case it returns unsuccessfully. Otherwise, *DeleteMin* must return the minimum item (highest priority) in the first nonempty list. First, list 0 is locked. If it is empty, list 0 is released and the next list is locked. This process is repeated until a nonempty list is found. The minimum item from this list is deleted and returned.

The BCPQ concurrent data structure allows implementing *DeleteMax*, and with minor modifications *delete/P穏value* as efficiently as *DeleteMin*. *DeleteMax* returns the item with lowest priority. Since the distributed search list remains sorted at all times, one only has to search the item with the largest index in this list. Find operations can also be easily implemented.

In the BCPQ, each PE stores a part of the list, protected by a single lock. If the number of items stored in the list is large and uniformly distributed to PEs, concurrency is expected to match the number of PEs, due to the low number of access conflicts.

### 3.3.1 Load Balancing the BCPQ data structure

*Insert* and *DeleteMin* operations may lead to load imbalance in the BCPQ data structure. The target list, on which new items are inserted depends on the exact item priorities. *DeleteMin* operations delete items from local lists with small index numbers. Local list minima and maxima are influenced by the local number of items and distribution of priority values, and may reduce concurrency by making insert operations time consuming. Thus, a load balance operation, which can also be called from the application layer is necessary.

To simplify and improve performance of our BCPQ implementation, we assume that the following three conditions, representing the state of our distributed list, hold prior to any inserting operation:

1. The distributed list is compact. (All local lists to the right of an empty local list are empty.)
2. In all local lists, there is enough memory space to store new items.
3. The distributed list is sorted.

Non-random *Insert* operations tend towards violating condition 2, while *DeleteMin* operations tend towards violating condition 1. The load balance operation is responsible for maintaining these two conditions. Load balance should only exchange items between neighbors, otherwise condition 3 could be violated. From condition 1 it follows, that any empty lists may exist only at the "extreme right" PEs, the ones with the highest index numbers.

BCPQ uses a deterministic prefix-based load balance operation, that consists of two phases: a computation phase, followed by a communication phase. Prior to load balance, let the length of local list \(i\) be \(L_i\), \(0 \leq i < P\).
In the computation phase $PE_i (0 \leq i < P)$ computes the number of items $M_i$ that have to be moved from $PE_i$ to maintain load balance. First, the total number of items $N$ is computed at all PEs, by summing all local counters; in our implementation we have a total counter allocated at $PE_0$. The average number of items per local list is $\frac{N}{P}$. To equally distribute $N$ items, $H_i$ items must be stored in list $i$, where

$$H_i = \begin{cases} \left\lfloor \frac{N}{P} \right\rfloor & : i \geq N \text{ mod } P \\ \left\lceil \frac{N}{P} \right\rceil & : i < N \text{ mod } P \end{cases} \quad (0 \leq i < P) \tag{1}$$

The number of items $PE_i$ has to move to get $H_i$ items is $|K_i|$, where $K_i = H_i - L_i$. A positive value of $K_i$ means that $PE_i$ holds very few items and must get items from its neighbors. A negative value means $PE_i$ has to move items to its neighbors.

Consider prefix list $P; (0 \leq i < P)$ formed by concatenating all local lists with index smaller or equal to $i$. Using the previously computed values for $K_i$, the number of items $PE_i$ must move/get to achieve load balance.

$$M_i = \sum_{j=0}^{i} K_i \quad (0 \leq i < P) \tag{2}$$

If $M_i > 0$, then $PE_i$ has to get $M_i$ items from the PEs to the right. Otherwise, it has to put $|M_i|$ items to the PEs to the right.

Hence, in the communication phase, all PEs access their right neighbor’s memory. Depending on the value of $M_i$, items are moved either from, or to each local list. At each step as many items as possible are moved, while remaining items are moved in subsequent steps (pipelining). Neighbor to neighbor movements are repeated, until load balance is achieved. For moving items, two local lists must be updated. This means competition in locking local lists. Network congestion is reduced and possible data races [25] (or deadlocks) are avoided by dividing each communication step into two steps; first, only PEs with even index numbers are active moving items, then odd PEs are active (odd-even pattern).

In Figure 5, an example illustrates the computation and communication phase for load balancing; inactive PEs are shown in gray. Also notice that, the last PE ($PE_{5}$) in our load balancing algorithm does not have to perform any local operations on its own memory.

3.3.2 ShMem Implementation of BCPQ

Since the BCPQ data structure is new and also more general than either DPQ, or RCPQ, we provide considerable more details of our implementation.

The BCPQ implementation consists of four layers, the application layer, the data organization layer, the list access layer, and the data access layer (see Figure 6).

- Similar to RCPQ, the lowest layer uses ShMem functions and local memory operations together, to abstract from the virtual shared memory data distribution. Thus, local and remote accesses are disguised under the same function calls. Functions getItems and setItems access remote or local items. Functions getData and setData access remote or local data elements. Functions getCounter and setCounter access or update the total number of items stored at $PE_0$. Functions setNonempty and resetNonEmpty set or reset local list flags.

- The list access layer abstracts from the implementation of local circular lists. Function cutItems cuts a sequence of items from a local circular list, pasteItems stores a sequence of items to a given circular list. To examine or update the state of a list, this layer also provides macros, such as isEmpty, isFull, cyclicAdd, cyclicDec, Size. Binary search functions findPE and findPos, used during the insert operation, are also provided at this layer.

- ADT functions initQueue, insert, deleteMin and loadBalance are provided within the data organization layer.

- On the application layer, user applications calling the BCPQ data structure can be implemented.

The data structure type definitions are shown in Table 2. Items are represented by their priority, a data pointer, and the PE on which the data is stored. Items are arranged in cyclic local lists, that allow appending items to the head or to the tail. Local lists are symmetric arrays across PEs (C static variables), and thus remotely accessible by ShMem operations. A binary variable nonempty is associated with each list; it is set to 1, if the local list contains items, otherwise it is set to 0.
\[ N = \sum L_i = 474 \]

<table>
<thead>
<tr>
<th>i</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
</tr>
</thead>
<tbody>
<tr>
<td>( L_i )</td>
<td>0</td>
<td>15</td>
<td>10</td>
<td>33</td>
<td>20</td>
<td>65</td>
<td>89</td>
<td>3</td>
<td>100</td>
<td>67</td>
<td>22</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>( H_i )</td>
<td>30</td>
<td>30</td>
<td>30</td>
<td>30</td>
<td>30</td>
<td>30</td>
<td>30</td>
<td>30</td>
<td>30</td>
<td>30</td>
<td>30</td>
<td>30</td>
<td>30</td>
<td>29</td>
<td>29</td>
<td>29</td>
</tr>
<tr>
<td>( H_i \cdot L_i )</td>
<td>30</td>
<td>15</td>
<td>20</td>
<td>-3</td>
<td>10</td>
<td>-35</td>
<td>-69</td>
<td>27</td>
<td>-70</td>
<td>-37</td>
<td>7</td>
<td>19</td>
<td>19</td>
<td>19</td>
<td>19</td>
<td>19</td>
</tr>
<tr>
<td>( M_i )</td>
<td>30</td>
<td>45</td>
<td>65</td>
<td>62</td>
<td>72</td>
<td>37</td>
<td>-22</td>
<td>5</td>
<td>-65</td>
<td>-102</td>
<td>-95</td>
<td>-76</td>
<td>-57</td>
<td>-38</td>
<td>-19</td>
<td>0</td>
</tr>
</tbody>
</table>

\[ \begin{array}{cccccccccccccccc}
15 & 0 & 43 & 0 & 85 & 0 & 67 & 25 & 35 & 132 & 0 & 32 & 0 & 20 & 0 & 20 \\
0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 & 14 & 15 \\
15 & 45 & 32 & 62 & 7 & 37 & 0 & 5 & 0 & -102 & -73 & -76 & -47 & -38 & -9 & 0
\end{array} \]

\[ \begin{array}{cccccccccccccccc}
15 & 43 & 0 & 62 & 23 & 37 & 30 & 30 & 30 & 30 & 30 & 102 & 0 & 32 & 0 & 20 & 0 \\
0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 & 14 & 15 \\
15 & 2 & 32 & 0 & 7 & 0 & 0 & 0 & 0 & 0 & -73 & -44 & -47 & -18 & -9 & 0
\end{array} \]

\[ \begin{array}{cccccccccccccccc}
30 & 28 & 32 & 30 & 30 & 30 & 30 & 30 & 30 & 30 & 29 & 73 & 0 & 32 & 11 & 29 \\
0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 & 14 & 15 \\
0 & 2 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -44 & -15 & -18 & 0 & 0
\end{array} \]

\[ \begin{array}{cccccccccccccccc}
30 & 30 & 30 & 30 & 30 & 30 & 30 & 30 & 29 & 29 & 44 & 14 & 29 & 29 \\
0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 & 14 & 15 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -15 & 0 & 0 & 0 & 0
\end{array} \]

\[ \begin{array}{cccccccccccccccc}
0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 & 14 & 15
\end{array} \]

Figure 5: Example of load balance on 16 PEs, using odd-even communication.

lists are protected by locks. The total number of items in the distributed data structure is stored at PEO in a global counter numitems, protected by a lock numlock.

At each PE, symmetric variables are used to store the state of the local cyclic list (see Figure 7). First stores the index of the smallest item in the circular list (in Figure 7, first=5), while last stores the index of the largest item (in Figure 7, last=1). Items are sorted in anti-clockwise direction. To distinguish between full and empty state, the position before first is always empty. In empty lists, last points to this index. A full list is indicated by last pointing two positions before first. Macros IsEmpty, IsFull, and Size provide the status of a cyclic list. Macros cyclicInc, cyclicDec, cyclicAdd and cyclicSub cyclically increase and decrease list counters.

Function cutItems(pos, n, buffer) cuts n items at index pos from the local list in anti-clockwise direction and stores them into buffer; buffer is a node_t type array. For instance, cutItems(7, 3, buffer) called on the list in Figure 7 removes the items \{(17, 25, 30)\} and stores them into buffer. Function pasteItems(pos, n, buffer) copies n items from buffer in anti-clockwise direction into the list, starting at pos. In Figure 7, pasteItems(8, 2, (28, 29)) would replace 25 with 28 and 30 with 29. Notice that in our small example, only priority values are given.

The Insert function inserts item prio by first searching for the local list, then for the index where the new item has to be inserted. Before calling Insert the list must be compact and enough
Figure 6: Layers of BCPQ implementation

Table 2: BCPQ data structure definitions

space must be available in all local lists. This is handled by load_balance. The Insert function is shown in Table 3. First, the global counter is incremented (lines 6–7). Parameter PEn represents the last nonempty list. It is computed prior to calling Insert using a global sum on the symmetric
Table 3: BCPQ Insert function

```c
int Insert (long prio, data_t data, int *PEm, queue_t *bcpq) {
    static node_t buffer[LocalSize];
    static pos_t pos;
    static cycle_t cycle, righte, lefte;
    static ulong anzright, anzleft;
    static short numitem;
    numitem = fast_shmem.short.fine(&(bcpq->numitems), 0);
    if (numitem >= QueueSize) return -1; /* list is full */
    pos.pe = findPE(prio, 0, FEm, bcpq);
    getLocalSize(pos.pe, cycle, bcpq);
    if (!isEmpty(cycle)) {
        pos.pos = cycle.first;
        buffer[0].prio = prio; buffer[0].data = data;
        setItems(pos, &buffer[0], 1, bcpq);
        cycle.last = cycle.first = pos.pos;
        setLocalSize(pos.pe, cycle, bcpq);
        setNonEmpty(bcpq, pos.pe);
        ClearLock(&(bcpq->lock[pos.pe]));
        return 0; /* success */
    } else { /* append to last */
        pos.pos = findPos(prio, pos.pe, cycle, bcpq);
        if (pos.pos == cycle.last) {
            buffer[0].prio = prio; buffer[0].data = data;
            pastItems(pos, 1, &buffer[0], bcpq);
            cycle.last = cyclicInc(cycle.last);
        } else if (pos.pos == cycle.first) {
            pos.pos = cyclicDec(pos.pos);
            buffer[0].prio = prio; buffer[0].data = data;
            pastItems(pos, 1, &buffer[0], bcpq);
            cycle.first = cyclicInc(cycle.first);
        } else { /* insert item */
            left.first = cycle.first;
            left.last = cyclicDec(pos.pos);
            anzleft = Size(lefte);
            right.first = pos.pos;
            right.last = cycle.last;
            anzright = Size(anzright);
            if (anzright < anzleft) {
                cutItems(pos, anzright, &buffer[1], bcpq);
                buffer[0].prio = prio; buffer[0].data = data;
                pos.pos = cyclicInc(pos.pos);
                pastItems(pos, anzright - 1, &buffer[0], bcpq);
                cycle.last = cyclicInc(cycle.last);
            } else {
                pos.pos = cyclicInc(pos.pos, anzleft);
                cutItems(pos, anzleft, &buffer[0], bcpq);
                buffer[anzleft].prio = prio; buffer[anzleft].data = data;
                pos.pos = cyclicDec(pos.pos);
                pastItems(pos, anzleft + 1, &buffer[0], bcpq);
                cycle.first = cyclicDec(cycle.first);
            }
            setLocalSize(pos.pe, cycle, bcpq);
            ClearLock(&(bcpq->lock[pos.pe]));
            return 0; /* success */
        }
    }
}
```
searches only nonempty local lists 0 to PEi, locking PEs while examining them as potential target lists. If the list is empty the new item is inserted using a sequence of set operations (lines 9–19). Otherwise, function FindPos determines the index in the circular list of target PE, where item prio is to be inserted, in the circular list of the target PE. To minimize moving of items the distance to the start/end of the list is first computed (lines 21–45). If the item is closer to the head, the head sublist is moved one position in front, otherwise tail is moved. Moving sublists is done by copying corresponding items into a local buffer (lines 38 and 45). Item prio is appended to this list and the new list is stored back at the new index. The local list counter is updated, the lock is released and Insert is complete.

<table>
<thead>
<tr>
<th>BCPQ DeleteMin Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>int DeleteMin (long *priority, data *data, queue *vsq);</td>
</tr>
<tr>
<td>static pos_t pos;</td>
</tr>
<tr>
<td>static cycle_t cycle;</td>
</tr>
<tr>
<td>static node_t min;</td>
</tr>
<tr>
<td>static short numitem, value,dec = -1;</td>
</tr>
<tr>
<td>numitem = shmem.short.fadd(k(vsqq-&gt;numitems), value,dec, 0);</td>
</tr>
<tr>
<td>if(numitem &lt; 1) /* list is empty */</td>
</tr>
<tr>
<td>fast.shmem.short.finc(k(vsqq-&gt;numitems), 0);</td>
</tr>
<tr>
<td>return -1;</td>
</tr>
<tr>
<td>pos.pe = 0;</td>
</tr>
<tr>
<td>do</td>
</tr>
<tr>
<td>find PE with nonempty list */</td>
</tr>
<tr>
<td>SetLock(k(vsqq-&gt;lock[pos.pe]));</td>
</tr>
<tr>
<td>pos.size = LockSize(pos.pe, kcycle, vsq);</td>
</tr>
<tr>
<td>if[kEmpty(cycle)];</td>
</tr>
<tr>
<td>ClearLock(k(vsqq-&gt;lock[pos.pe]));</td>
</tr>
<tr>
<td>pos.pe = (pos.pe + 1) % NPEs;</td>
</tr>
<tr>
<td>while[kEmpty(cycle)];</td>
</tr>
<tr>
<td>/* cycle if active insert */</td>
</tr>
<tr>
<td>if pos = cycle.first;</td>
</tr>
<tr>
<td>getitems(pos,min,1,vsq);</td>
</tr>
<tr>
<td>*priority = min.prio;</td>
</tr>
<tr>
<td>+data = min.data;</td>
</tr>
<tr>
<td>cycle.first = cycle.next(cycle.first);</td>
</tr>
<tr>
<td>setLocalSize(pos.pe, cycle, vsq);</td>
</tr>
<tr>
<td>if[kEmpty(cycle)];</td>
</tr>
<tr>
<td>ClearLock(k(vsqq-&gt;lock[pos.pe]));</td>
</tr>
<tr>
<td>return 0;</td>
</tr>
<tr>
<td>/* minexist */</td>
</tr>
</tbody>
</table>

Table 4: BCPQ DeleteMin function

The DeleteMin function is shown in Table 4. First the central counter on PEb is updated (lines 6–7). Then, starting with list 0, local lists are successively locked and their state examined until the first nonempty list is found (lines 8–16). If all local lists are found empty, the Insert operation of an item is not finished. DeleteMin continues checking again all local lists, until the Insert operation is done and DeleteMin can find the item. The minimum item is always stored at the first position in the local list. It is deleted from the list and the head pointer and nonempty flag are updated (lines 17–23). DeleteMin is complete. This implementation of DeleteMin can be called at any point within an application.

The implementation of load balance is shown in Tables 5 and 6, for the computation and communication phase, respectively. In the computation phase, the difference Xi between the number of items stored at PEi and the desired number of items H is computed (lines 9–12). The result is stored in the remotely accessible variable calc. ShMem does not provide scan routines, hence the computation of the partial sums Mi (Equation 2) was implemented within BCPQ using a recursive doubling communication pattern (lines 14–21). The algorithm used for P PEs consists of log(P) steps. In step j (0 ≤ j < [log P]) PEi is getting the partial sum from PEi−2j (line 16), if this PE address is valid (tested by ProcExists macro) calc is added to the received value at step (line 18). The barriers (line 17 or 19) are needed to avoid a data race between local writes and remote reads from calc. After this barrier, the cache line for calc is also invalidated, so that the register value can be used. After all PEs complete the scan operation (line 22), partial sum Mi is stored in partial_sum at PEi, and the absolute value is stored in movNo.

As shown in Figure 8, a parallel scan operation (shown as a partial sum instantiation) can be
Table 5: BCPQ load_balance function (computation phase)

<table>
<thead>
<tr>
<th>(a)</th>
<th>Recursive Doubling</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>8</td>
<td>9</td>
</tr>
<tr>
<td>10</td>
<td>11</td>
</tr>
<tr>
<td>12</td>
<td>13</td>
</tr>
<tr>
<td>14</td>
<td>15</td>
</tr>
<tr>
<td>16</td>
<td>17</td>
</tr>
<tr>
<td>18</td>
<td>19</td>
</tr>
<tr>
<td>20</td>
<td>21</td>
</tr>
<tr>
<td>22</td>
<td>23</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>(b)</th>
<th>Divide and Conquer</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>8</td>
<td>9</td>
</tr>
<tr>
<td>10</td>
<td>11</td>
</tr>
<tr>
<td>12</td>
<td>13</td>
</tr>
<tr>
<td>14</td>
<td>15</td>
</tr>
<tr>
<td>16</td>
<td>17</td>
</tr>
<tr>
<td>18</td>
<td>19</td>
</tr>
<tr>
<td>20</td>
<td>21</td>
</tr>
<tr>
<td>22</td>
<td>23</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>(c)</th>
<th>Odd-Even Reduction</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>8</td>
<td>9</td>
</tr>
<tr>
<td>10</td>
<td>11</td>
</tr>
<tr>
<td>12</td>
<td>13</td>
</tr>
<tr>
<td>14</td>
<td>15</td>
</tr>
<tr>
<td>16</td>
<td>17</td>
</tr>
<tr>
<td>18</td>
<td>19</td>
</tr>
<tr>
<td>20</td>
<td>21</td>
</tr>
<tr>
<td>22</td>
<td>23</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>(d)</th>
<th>Recursive Broadcasting</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>8</td>
<td>9</td>
</tr>
<tr>
<td>10</td>
<td>11</td>
</tr>
<tr>
<td>12</td>
<td>13</td>
</tr>
<tr>
<td>14</td>
<td>15</td>
</tr>
<tr>
<td>16</td>
<td>17</td>
</tr>
<tr>
<td>18</td>
<td>19</td>
</tr>
<tr>
<td>20</td>
<td>21</td>
</tr>
<tr>
<td>22</td>
<td>23</td>
</tr>
</tbody>
</table>

Figure 8: Scan related communication patterns on 8 PEs

performed using several approaches, such as (a) recursive doubling, (b) divide and conquer, (c) odd-even reduction, and (d) recursive broadcasting. Three of the patterns require one local variable
per processor, while divide and conquer requires two. The change of values during the communication/computation phase of these patterns is also shown in the figure. The best pattern to realize a scan operation clearly depends on the topology and the relative cost of arithmetic, send/recv, remote copy, and broadcast operations. The aim is to keep all processors busy, while minimizing the latency costs. For the Cray-T3E we have found that a recursive doubling communication pattern is faster; other centralized based schema quickly degrade with the number of PEs.

```c
int load_balance(queue_t *vq) {
    < Load Balance Computation Phase >
    int turn = 0;
    while (go.on) {
        if (movNo > 0) &
            (MyId % 2 == turn)) {
            /* get elements from the right */
            pos.pe = MyId + 1;
            movNo.now = getLocalSize(pos.pe, cycle, vq);
            if (Size(cycle) > movNo) movNo.now = movNo;
            if (movNo.now > 0) {
                pos.pos = cycle.first;
                cutItems(pos, movNo.now, &buffer[0], vq);
                cycle.first = cycleAdd(cycle.first, movNo.now);
                setLocalSize(pos.pe, cycle, vq);
                pos.pe = MyId;
                getLocalSize(pos.pe, cycle, vq);
                pos.pos = cycleAdd(cycle.last);
                pasteltems(pos, movNo.now, &buffer[0], vq);
                cycle.last = cycleAdd(cycle.last, movNo.now);
                setLocalSize(pos.pe, cycle, vq);
                movNo = movNo - movNo.now;
                if (movNo == 0) done = fast_shmem_short_line(&count.done, 0);
            }
        } else if (partial_sum < 0) {
            /* put elements to the right */
            pos.pe = MyId;
            movNo.now = getLocalSize(pos.pe, cycle, vq);
            if (Size(cycle) > movNo) movNo.now = movNo;
            if (movNo.now > 0) {
                pos.pos = cycleSub(cycle.last, movNo.now - 1);
                cutItems(pos, movNo.now, &buffer[0], vq);
                cycle.last = cycleAdd(cycle.last, movNo.now);
                setLocalSize(pos.pe, cycle, vq);
                pos.pe = MyId + 1;
                getLocalSize(pos.pe, cycle, vq);
                pos.pos = cycleSub(cycle.first, movNo.now);
                pasteltems(pos, movNo.now, &buffer[0], vq);
                cycle.first = cycleAdd(cycle.first, movNo.now);
                setLocalSize(pos.pe, cycle, vq);
                movNo = movNo - movNo.now;
                if (movNo == 0) done = fast_shmem_short_line(&count.done, 0);
                turn = 1 - turn;
                done = shmem_short(&count.done, 0);
                if (done == (short) NPEs) go.on = 0;
            }
        } getLocalSize(MyId, cycle, vq);
        if (!isEmpty(cycle)) vqq = nonempty = 0;
        else vqq = nonempty = 1;
        return 0; /* success */
    }
}
```

Table 6: BCPQ load_balance function (communication phase)

In the communication phase, depending on the sign of partial_sum, PEs move items either from local memory to the memory of their right neighbor (lines 45–62), or vice-versa (lines 27–45). The number of elements PEₙ has to move is stored at movNo locally.

Items are moved, in a pipelining fashion. If the number of elements available in the source list is smaller than movNo (line 30 and 48) all items from the source list are moved. Remaining items are moved in consecutive steps. To decrease congestion and avoid data races in local lists, communication is split into odd and even steps (line 26). Since synchronization of PEs in odd and even phases is done with a barrier (line 63), all PEs must loop until all items are moved. Variable done is used to count the number of PEs which have finished moving items (lines 46 and 64). Variable go.on, initialized to 1, is set to 0 when all PEs have finished moving items (line 66). Then, all PEs can exit from the pipelining loop (line 25).
4 Parallel Simulation using Priority Queues

Serial switch simulation is considerably slow and suffers from insufficient memory. Provided that suitable data structures exist, switch simulation offers natural parallelism. Trace driven traffic on different input ports is completely independent. These ports can be scheduled to different processors without any conflicts. Coordination is only needed to handle output port congestion.

Priority queues can be used for event scheduling. They represent the core of the event queue data structure. Events are inserted to the queue, and the next event to execute is obtained by calling DeleteMin. Execution of this event may lead to other events being created, and inserted to the event queue. A cycle-level precise ATM switch simulator, using a priority queue as an event queue data structure is presented in [8]. In this study, we present a simpler simulation model for evaluating the performance of our parallel data structures.

4.1 Simple Model of Crossbar Switch

We model a $64 \times 64$ output buffered crossbar packet switch which supports multicasting. For the DPQ, we will only show results for single buffer systems, due to data structure limitations on both memory space, and especially execution time.

The switch uses a central clock. Routing is synchronous and all ports operate at the same frequency. During each clock cycle, an incoming packet at each input port is routed into corresponding output buffers. Since the switch supports multicasting, a single packet arrival may trigger multiple packet departure events from different output buffers. If an output buffer is full, packets routed to this buffer are discarded. At the end of each clock cycle, the packet with the highest priority at each output buffer (if any) departs the switch. Hence, multicast copies may depart from output ports at different times. Since, packet departures at different output buffers are independent events, each output buffer can be modeled separately by a priority queue.

Simulation experiments were run on $N$ ($N \in \{2, 4, 8, 16, 32, 64\}$) PEs on one of the Gray-T3E900 systems available at KFA-Jülich. Each PE simulates $\frac{64}{N}$ input ports and supervises $\frac{64}{N}$ output ports. In Figure 9, the multicast packet arriving at input port 1 is copied to output buffers 1, 3, 4, and 64. Figure 9, also shows how 32 PEs can be used to model the $64 \times 64$ switch. Input ports 1 and 2 are managed by PE0, input ports 2 and 3 are managed by PE1, and so on. Finally, input ports 63 and 64 are assigned to PE31; Output ports are similarly assigned to PEs. In contrast, elements for each buffer are distributed across all PEs according to our parallel data structures. Although PEs issue sequentially insert/delete requests to their statically assigned input/output ports, uninterrupted inserts or deletes (e.g. in code blocks separated with barriers) leave each parallel priority queue at a state which is uniquely identified by the ADT operations.

Our experiments were designed to focus on data structure performance, independent of initialization overheads, e.g. arising from initQueue. Thus, for all our tests, RCPQ and BCPQ allocate and initialize similar amounts of main memory. The DPQ implementation uses dynamic data allocation (and initialization) by calling malloc and free memory management operations.

For BCPQ, we initially focused on a flexible and correct implementation. Results from an improved BCPQ version (called BCPQ Opt.) will also be shown. BCPQ Opt applies both algorithmic, and runtime optimizations. It uses a local fprintf routine for finding the target list, thus avoiding locking target lists. The new fprintf implementation is based on broadcasting maximum priorities of all local lists to all PEs. This all-to-all broadcast is coded using shmem_int_p and shmem_qwreat, which is
more efficient than using the Cray MPP shmem_int_bcast routine. Other algorithmic optimizations have been directed at shared memory locks with improvements over the Cray shmem_lock library functions at above three orders of magnitude on a 64-processor T3E-500 [9]. Finally, BCPQ Opt implements the PEn computation using node-to-neighbor communication (nextShmem_int.p) and parallel searching of the distributed nonempty field patterns. For example, a distributed nonempty field of 1, 1, 1, 1, 0, 0 implies PEn = 4, since PE4 is the last nonempty PE (condition 1 cf. Section 3.3.1). Both a Cray library eureka- or shmem_event-based parallel searching approach, and a direct shmem_int_sum_to_all based implementation were slower alternatives for the PEn computation.

We compiled all implementations with the highest optimization level provided by the Cray C/C++ compiler, compiler switch -O3. For time measurements, we use the Cray intrinsic function _rtcQ. The Cray C/C++ compiler generates inline code for each call, reducing overhead introduced by timers. We measured a) the total execution time, b) time for all insert operations, c) time for all DeleteMin operations, and for the BCPQ, d) time for all load balance operations. The latter times are easily converted to times per operation by dividing with appropriate counters.

4.2 Single Buffer Experiments

A synthetic priority queue application was used for our first comparisons of DPQ, RCPQ, and BCPQ data structure implementations. Insert and DeleteMin operations are performed on a single buffer modeled by a parallel priority queue. During each cycle, each PE performs one or two Insert operations, followed by one DeleteMin operation. For BCPQ, each cycle ends with a call to load balance.

Notice that in the RCPQ and BCPQ implementations, Insert and DeleteMin are synchronized by a barrier. The DPQ implementation is more asynchronous and does not allow one to apply exactly the same level of synchronization, instead Insert and DeleteMin cycles are performed in this order, but independently by all PEs.

![Figure 10: Runtime for single buffer: a) 1 Insert and DeleteMin, b) 2 Inserts and 1 DeleteMin per cycle](image)

In Figure 10 we show the total execution time vs. the number of cycles for 64 PEs. Similar results have been obtained for other numbers of PEs. From Figure 10 (a) notice that, for an equal number of Insert and DeleteMin operations, the DPQ implementation is \( \sim 3.5 \) times or more slower than our virtual shared memory implementations. For up to 150 cycles (9,600 Insert and 9,600 DeleteMin operations) BCPQ performs better than the RCPQ implementation. For a greater number of iterations the RCPQ implementation is the fastest one, but execution times of both concurrent implementations are comparable. The optimized version of BCPQ is \( \sim 5\% \) faster.

Increasing the number of Insert operations per cycle makes DPQ perform much worse. It is \( \sim 6 \) times slower than our RCPQ and BCPQ implementations (see Figure 10 (b)). Since the initial implementation of Insert function in the BCPQ is slower than the corresponding RCPQ implementation, RCPQ is slightly faster. The optimized BCPQ version with a local findPE routine is \( \sim 25\% \) faster than RCPQ.

Figure 11 compares times per Insert, DeleteMin, and load-balance operations for the RCPQ and BCPQ implementations, for 2 and 64 PEs; typically, typically 500 iterations were used. Both implementations show comparable per operation execution times. The optimized version of BCPQ executes
DeleteMin operations faster than RCPQ, because DeleteMin operations on heaps require reheapification. For 64 PEs, Insert time for the optimized BCPQ version decreases, and approaches that of the RCPQ implementation. For BCPQ, load balance is faster and scales better than Insert and DeleteMin operations. The relative time spent for load balance on 2 PEs is relatively larger, compared to the other two ADT operations, due to a constant computation overhead which shows a stronger impact if the total execution time is smaller (notice that the scales are different).

Thus, results from our synthetic application show that DPQ is always much slower than RCPQ and BCPQ. Using MPI instead of low level shared memory libraries significantly deteriorates performance. The decision for a programming model together with an appropriate library has a major impact on application performance. Before deciding on a library, one must also consider if portability is really needed, or expected.

4.3 Multicast Crossbar Switch Experiments

For our switch experiments, we used a constant input traffic model with fixed average multicasting rate. The arrival rate was set to one packet per cycle. Independently chosen random values for priorities and packet destinations were used.

After each clock cycle, the BCPQ data structure is dynamically load balanced. Barriers are used after each operation to satisfy our synchronous switch requirement. Users can further improve BCPQ performance by providing a threshold condition for load balance. This condition will normally depend on the application, and can be implemented either as fuzzy logic, or by using neural network learning techniques. Notice that, no load balancing degrades BCPQ performance severely; in our experiments, a BCPQ which does not invoke load balance performs ~5 times slower.

Performance and scalability experiments for the $64 \times 64$ crossbar switch, with multicasting rates of 1 (unicast) and 64 (broadcast) are shown in Figures 12 and 13.

In Figure 12, we show the total execution time vs. the number of simulation cycles for unicasting or broadcasting with 64 PEs; similar results have been obtained for other numbers of PEs and multicast rates. Both concurrent implementations perform almost linearly to the number of simulation cycles.

By comparing the two graphs, we see that the total execution time slightly increases with the number of items stored in the priority queue.

For unicasting, as many items are removed from the priority queues as are inserted, thus keeping priority queues always empty between consecutive simulation cycles; this would correspond to circuit switching, or hot-potato routing protocols, e.g. for simulating optical communication networks. Thus, in this case, load balancing the BCPQ is not needed. As shown in Figure 12 (a), for all implementations the execution time scales well with the number of simulation cycles. Also, RCPQ outperforms both
Figure 12: Runtime for crossbar switch (unicasting and broadcasting)

BCPQ versions. This is partially due to an overhead associated with maintaining maximum list priorities for the `findPE` computation; for uncasting, this extra work is largely unnecessary, since on the average buffers contain very few elements. To speed up BCPQ, one can rely on the fact that a simpler `DeleteMin` implementation is now possible; the item with the highest priority for each queue is always stored at the first position at the local list of `PE0`. Furthermore, one can pre-compute a random assignment of our distributed queues to PEs, so that the “minimum” local list for a given queue is shuffled across all PEs. This would remove network and memory contention at `PE0` during concurrent `DeleteMin` operations, at the expense of some local only computation.

For broadcasting, although the number of items at each queue increases a lot between consecutive simulation cycles, the total time changes only slightly. The optimized BCPQ version exhibits better scalability and superior performance to RCPQ for any reasonable number of simulation cycles. Thus, with the dynamic load balanced BCPQ data structure, the overhead of handling priority queue ADT operations on tens of thousands of data items is relatively small.

Figure 13: Scalability for crossbar switch (unicasting and broadcasting)

For a fixed problem size, Figure 13 reveals interesting scalability issues of our concurrent implementations. We consider uncasting and broadcasting on a 64 x 64 crossbar switch, with 2, 4, 8, 16, 32, and 64 PEs. Figure 13 shows an optimal number of PEs for both BCPQ implementations, around 16 PEs. Before this point, the total execution time is larger because each PE must simulate more inputs and output ports. After this point, the execution time increases due to a shift from computation towards communication work; for the optimized BCPQ this trend is obviously less profound. For broadcasting, the optimal number of PEs decreases to approximately 8, thus indicating a dependence on the number of insert operations. For both cases, RCPQ seems to saturate faster, possibly because of increased network contention.

Hence, our experimental analysis based on packet switch simulation experiments exhibits good
5 Conclusions and Extensions

We have provided fast and flexible implementations of parallel priority queues that would form a library for further research on parallel network simulation, and contribute to the experimental analysis of dynamic distributed data structures. Our implementations consist of ~ 6,000 lines of documented C code (~ 2,050 DPQ, ~ 1,100 RCPQ, ~ 1,750 BCPQ and ~ 1,000 switch simulator application).

We have simulated single buffers and also 64 x 64 multicast-based switches using up to 60,000 packets and 2 to 64 PEs. Our experiments on a Cray-T3E system at KFA-Jülich indicate that concurrent priority queues are 5 to 10 times faster than distributed ones, even though the distributed DPQ implementation uses asynchronous insertions and deletions.

Deciding for a parallel programming environment is a trade-off between portability and performance. Portable libraries are still much slower than platform dependent libraries. Our two concurrent virtual shared memory implementations, based on predetermined random mapping of a heap data structure (RCPQ), and dynamically load balanced distributed sorted lists (BCPQ) achieve comparable performance. However, BCPQ is more efficient in assigning memory to PEs.

Our work raises interesting practical and theoretical questions on all data structures.

The distributed priority queue (DPQ) is an interesting asynchronous data structure only on machines that do not support shared memory programming. An improvement to DPQ could reduce the workload imbalance which exists at root PE (and the upper heap levels) during ADT operations. While this could improve fault tolerance, software complexity would also increase considerably.

Our RCPQ and BCPQ priority queue implementations are easy to use from the application layer. There are no application restrictions or side effects arising from the core data structure, and they can both be extended and improved in various ways.

To reduce network contention, the centralized total items counter (which is accessed from ADT operations) could be implemented by emulating a counting network. Counting networks provide a distributed mechanism for implementing Fetch&Increment, by reducing bottlenecks associated with accessing a single (virtual) shared memory location [2, 10].

Our RCPQ implementation uses a random heap node distribution to PEs. A complex distribution which takes into account parent-child communication patterns could reduce latency during heapification. However, one must notice that the Cray-T3E uses adaptive message routing, and thus deterministic allocation could have its pitfalls.

The BCPQ implementation also allows low level runtime optimizations. The communication overhead can still be reduced by experimenting with different ShMem subroutines, access patterns, and physical PE allocation schemes. We have noticed an interesting tradeoff between keeping a map up to date with all system variables and minimizing program latency. In that respect, the minimum amount of redundancy which provides for a consistent and efficient data structure is an open question.

In some processors, e.g. the MC680x0 series, a double Compare&Swap atomic operation is available. Two separate words are compared with test values, and if both words match the test values they are replaced with new values. The BCPQ data structure can be transformed to be lock free [11]. The double Compare&Swap operation allows updating both head, and tail pointers at once, testing at the same time if these pointers remained unchanged.

Although, there is already upward compatibility of ShMem routines on new Cray systems, it is expected that MPI-2 will open new horizons for extensions of this work. While the data access layer will have to be rewritten, most of the main part of the code can remain unchanged. An MPI-2 version would essentially make our source code portable on a variety of hardware platforms.

Since our work is experimentally oriented, theoretical analysis lies outside its scope. More work could be performed on defining runtime semantics for asynchronous RCPQ and BCPQ ADT operations. Theoretical extensions can also focus on evaluation of space and time complexity, and asymptotic performance. To obtain any realistic results, a model on the cost of virtual shared memory accesses for each different operation is needed. Reverse profiling could prove a valuable tool in this direction [12].

Other extensions mainly involve applications based on priority queues. Performance of a parallel network simulator can be improved by changing sorting algorithms to the BCPQ data structure [6]. A sequential ATM switch simulator can also be parallelized using different priority queues for event scheduling [8]. In this case, internal timing delays can be accurately modeled.
In retrospect, there are several potential extensions for device simulation:

- Implementation of more realistic protocols, including flow control. Priorities can be used in implementing various routing protocols.
- Accurately model, possibly at the flit level, internal events in crossbar switches.
- Simulate large input or central buffered ATM switches [28].
- Simulate asynchronous switches or networks of switches, like butterflies or multibutterflies [1, 16, 17].
- Extend traffic models to incorporate hot spot and bursty traffic.

Furthermore, code from our layered BCPQ data structure can be reused in other parallel applications with relatively small changes.

- The list access layer can form the basis for dynamic parallel insertion sort (or bucket-sort) implementations.
- The list access layer can be extended to dictionary machines, by additionally supporting find_k, find_min and find_max operations. Since BCPQ is based on sorted lists, these operations can be implemented very efficiently.
- Set operations can be implemented efficiently using sorted lists. The list access layer can be extended to support such operations.
- Our dynamic load balance implementation is deterministic, flexible, and efficient. It could be reused in applications operating on distributed sorted lists or with diffusion based scenarios [27].
- Many dynamic applications process continued incoming data, e.g. convex hull in military applications. In such situations, either partial, or total order of elements must be maintained. The BCPQ is a flexible data structure for direct engagement in such problems.

Acknowledgments

Our research has been supported by KFA-Jülich (project INTENS). We also acknowledge partial support from EPCC (project TMR TRACS).

We are grateful to Prof. Michael Scott and Dr. Bernard Mans for their critical help at initial stages of this work. They kindly provided access to CPQ pseudocode and preliminary DPQ code.
References

Overview of MPP Production Applications at the Computer Center Garching

Hermann Lederer

RechenZentrum Garching der Max-Planck-Gesellschaft
Max-Planck-Institut für Plasmaphysik
D-85740 Garching
E-Mail: lederer@rzg.mpg.de, URL: http://www.rzg.mpg.de

Abstract.
Scientists from the Max Planck Society have gained experience with emerging MPP technology since the early nineties. After installation of a Cray T3D system at the Computer Center Garching of the Max Planck Society (RZG) in 1995 and its replacement by a Cray T3E system in 1996 with now 784 processors and 98 GB of memory, various projects have started to extensively use the MPP technology for large scale simulations and virtual experiments. The disciplines where applications have been parallelized and optimized for efficient usage of up to several hundred nodes of the T3E include Material Science, Theoretical Chemistry, Solid State Physics, Polymer Research, Biochemistry, Plasmaphysics, Laser Physics, Astrophysics, Astronomy, and General Relativity. This talk gives an overview of production codes from these scientific areas.

The full paper on the work presented here has already been published in:

Supercomputer 1998 - Anwendungen, Architekturen, Trends, p 68-81

References


A high-resolution numerical model for
the circulation of the Atlantic ocean

René Redler
Alfred-Wegener Institut für Polar- und Meeresforschung
Bremerhaven

Klaus Ketelsen
SiliconGraphics GmbH
Munich

Joachim Dengg and Claus W. Böning
Alfred-Wegener Institut für Polar- und Meeresforschung
Bremerhaven

Email: rredler@awi-bremerhaven.de

Abstract
The basin wide circulation of the Atlantic ocean, driven by heat and freshwater fluxes and its
reactions on fluctuations of the atmospheric forcing strongly depends on a variety of small scale
processes. Towards a realistic representation of the general circulation it is necessary to increase
the spatial resolution. Desired integration times on a decadal to centennial time scale therefore
require the use of high-end machines. An existing numerical code for solving time-dependant
hydrodynamical problems concerning the 3-dimensional ocean circulation has been recoded for
use on CRAY T3E systems. The CRAY-specific SHMEM library is used for message passing in
our 2-dimensional domain decomposition.

Contents

1 Introduction 2

2 Model configurations 3
  2.1 Atlantic Model ................................................. 3
  2.2 North Atlantic Model ........................................... 5

3 Software 5
  3.1 Parallelization Strategie ...................................... 6
  3.2 Optimization .................................................... 8
  3.3 Performance ..................................................... 10

4 Results 11

5 Conclusion 13
1 Introduction

Improving our understanding of the dynamical processes that make up the world ocean circulation is an effort that is severely hampered by the scarcity of available observational material. Numerical ocean models are used to get more insight into the dynamics of this system, which is a vital component of the global climate.

Ocean modeling is costly due to the large range of scales involved: The ocean's large-scale circulation is driven by wind, heat and freshwater fluxes, and its reaction to fluctuations of this atmospheric forcing strongly depends on a variety of processes covering a broad spectrum of time and length scales. While the climate response of the deep ocean takes place on a time scale of order 1000 years, the response of surface currents to changes of the local wind forcing occurs on time scales of a few hours. The length scales reach from the global scale (several thousand kilometers) down to the order of millimeters when small scale turbulent mixing is taken into account.

Figure 1: Lower left: Distribution of temperature at a depth of 100 m in a high-resolution numerical model of the North Atlantic. The magnification shows an excerpt from the tropical Atlantic with overlayed horizontal velocities. To reproduce some of the small scale features indicated in the figure, high resolution is required in numerical models.

Thus a compromise has to be made between the reduction of computational costs of very high-resolution and an explicit calculation of all physical processes involved. For typical basin wide ocean models this broad range of scales is restricted due to approximations of the physics involved to time scales reaching from hours up to 100 years, while the length scale is limited to values above the km-scale by the choice of grid resolution. Sub-grid
scale features have to be parameterized. The model is able to simulate phenomena like e.g. boundary currents, i.e. narrow energetic streams that connect ocean basins thousands of kilometers apart, or regional features, like e.g. the Equatorial Current, down to smaller scale features, i.e. eddies or equatorial waves with horizontal length scales of 10 to 100 km (Figure 1).

A numerical experiment covering decades to centuries should be practicable in a real-time scale within a time frame of weeks. This requires the use of high end machines like e.g. CRAY T3E systems. Integration over a one year period of a high-resolution model for the Atlantic Ocean like that described in Chapter 2.1 requires 170 hours of CPU time and approximately 90 MW memory on a CRAY C90. In this paper we will describe the model configurations and the MOM software package (Pacanowski, 1995) used. Optimization of the code was necessary to achieve an acceptable scalability for integration on large numbers of processing elements (PE). The paper is completed by results addressing the needs for high spatial resolution and long time integrations.

2 Model configurations

To allow a systematic investigation of different oceanic processes based on the same numerical code, the framework of FLAME (Family of Linked Atlantic Modeling Experiments, sketched in Figure 2) has been set up. Here, the model configurations of two of these models are described. A description of the software will follow in section 3.

2.1 Atlantic Model

The backbone of the model system is a so-called “eddy-permitting” model of the North and South Atlantic, situated between 70°N and 70°S with open boundaries across the Antarctic Circumpolar Current in the Drake Passage and south of Africa at 30°E. “Eddy-permitting” means that the horizontal grid resolution is high enough to allow formation of oceanic rings and eddies, but not yet high enough to resolve their dynamics in detail. The horizontal resolution of the locally rectangular grid is 1/3° in longitude and 1/3° × cos(ϕ) in latitude (ϕ), i.e. a grid with 37 km mesh size at the equator and decreasing to 12.5 km at the subpolar boundaries. This resolution results in 392 x 600 grid points. Grid points located in the Pacific are set to land, as are those located in the Mediterranean and Baltic Seas. To reduce the number of inactive grid points, advantage is taken of the Atlantic’s shape, which allows a “folding” of the model domain: As shown in Figure 3, ocean points located east of 5°E are “copied” onto land values on the American continent or the former Pacific, while land values east of 5°E can be ignored. This reduces the number of grid points required in zonal direction by more than 20% to 308. Cyclic boundary conditions are now needed to maintain continuity between adjacent ocean points that got separated due to the folding.

The vertical is discretized in 45 levels, with a grid spacing of 10 m in the 4 top levels increasing to a constant value of 250 m below 2500 m. The high near-surface resolution is intended to allow a more detailed representation of mixed-layer processes.
The FLAME Models

Figure 2: Overview of the FLAME model configuration. Gray shaded areas are set to land. Lateral boundaries to the Pacific and Indian Ocean are treated as open boundaries. Boundaries to the Arctic (70°N) and Antarctic Ocean (70°S) as well as to the Mediterranean Sea are closed.

Atmospheric boundary conditions are prescribed from the ECMWF (European Center for Medium Range Weather Forecast) model (Barnier et al., 1995). Wind stress, wind work and thermal boundary conditions are used to calculate monthly mean fields, which are then linearly interpolated by the ocean model to the actual time step. Freshwater fluxes are included via restoration of sea surface salinity to climatological values of Levitus et al. (1994).

To maintain water mass properties entering the model region from the Arctic, Antarctic and Mediterranean Sea, along artificially closed boundaries in the Nordic Seas between Greenland and Norway, in the Gulf of Cadiz and the Wedell Sea Newtonian damping keeps temperature and salinity values close to observations compiled by Levitus et al. (1994). For the open boundaries along sections connecting the continents of Africa and America with Antarctica a scheme allowing in- and outflow has been devised that follows the approach developed by Stevens (1991). For further details the reader is referred to Redler and Böning (1997).

A coarse resolution model version of this model (4/3° resolution, folded, 78 times 150 grid points in the horizontal) is used for short tests to define constraints for lateral boundary conditions, tests of the numerical code as well as climate sensitivity studies over long integrations periods.
2.2 North Atlantic Model

A variant of the high-resolution model concentrates on the North and Equatorial Atlantic (18°S to 70°N and 10°W to 16°E), with open boundaries along 18°S. The horizontal resolution and vertical discretization exactly corresponds to the Atlantic model; due to the smaller model area only 348 grid points in zonal and 356 grid points in meridional direction are required. Similarly to the Atlantic model the computational amount was reduced by nearly 25% due to the folding strategy. In this version ocean points located east of 10°W are transformed to land values on the American continent, land values east of 10°W are skipped. 263 grid points remained in zonal direction. The vertical resolution is the same as in the Atlantic model.

3 Software

The software used for the modeling effort described in this paper is based on the Modular Ocean Model (MOM), developed at the Geophysical Fluid Dynamics Laboratory (GFDL) Princeton (Pacanowski, 1995).

MOM is a FORTRAN code based on the so called "primitive equations" (Müller and Willebrand, 1989) which are derived from the Reynolds-averaged Navier-Stokes Equations under various assumptions, like the Boussinesq-, spherical and hydrostatic approximation. From these equations MOM calculates prognostic values of the 3-dimensional velocity, temperature (T) and salinity (S) distribution, density and pressure are diagnosed from the T- and S-fields.

For solving the momentum equation, the horizontal velocities are split up into a vertical
average and its deviations. The vertically averaged transport can be expressed by a 2-
dimensional stream function, which is described by a Poisson equation. This is solved
using an iterative conjugate-gradient method. Additional constraints are needed to keep
the values around the islands constant during every iteration step. Due to the hydrostatic
approximation the vertical component of the momentum equation degenerates and vertical
transports due to hydrostatic instabilities (convection) have to be parameterized.

The remaining 2nd and 4th order non-linear partial differential equations for temperature,
salinity and the vertical deviation of horizontal momentum are solved on regular 3-D
(longitude, latitude, depth) grids using the finite volume/difference approach. Central
differences are used in space and time, and an Euler forward time step is used regularly
to damp the computational mode. The 2nd and 4th order 2d algorithms require access
to nearest neighbors or the two next neighbors respectively. The neighbor values at the
boundary of the model are separately calculated or read in from external files.

3.1 Parallelization Strategy

To achieve as realistic as possible a representation of the oceanic circulation and to im-
prove the numerical representation of physical processes, high resolution in space and
time is necessary. Those requirements increase the amount of CPU time and memory
tremendously. The original sequential version of MOM2.1 was designed for workstations
and vector computers. Running high-resolution models over long simulation periods,
however, requires the use of large massive-parallel architectures like e.g. the CRAY T3E.
Effective use of large MPP systems will be made if the following problems can be solved.

- Work Sharing
- Data Distribution

A parallel version of MOM2.1 (see Fritzsch et al., 1998) has been created using two-
dimensional domain decomposition. Work sharing and data distribution has been imple-
mented, data exchange between Processing Elements (PE’s) is done via CRAY’s explicit
shared memory software (SHMEM library).

Figure 4 shows an example of the distribution of one layer in zonal (east-west) and merid-
ional (north-south) direction. The majority of work is distributed in meridional direction,
the zonal direction is split up into 2 up to 4 PE’s only. This examples shows a (2x7)
distribution, for a total of 14 PE’s.

Every PE is working on a subset of grid points. Inner boundaries have been defined to
enable the exchange of data between neighboring PE’s. One or two boundary lines are
necessary in each direction, depending on the use of 2nd or 4th order algorithms mentioned
above.

The main computation time in this code is typically spent in 3 folded loops. Figure 5
shows a simplified example of the structure of these loops in the sequential code.
Porting these loops to a domain decomposed parallel code, every PE's gets its individual part of the loops, as can be seen in Figure 6. The notation $i_{\text{pe}}$ defines the starting index on each PE, and $e_{\text{pe}}$ the ending index.

Running 3-D loops with PE dependent start and end index distribute the work but still require full 3-D arrays on all PE's because on some PE's $j_{\text{pe}}$ equals JM or $e_{\text{pe}}$ equals IM. As mentioned above running big models also requires data distribution. Figure 5 and Figure 6 describe an easy way of implementing data distribution. Using FORTRAN 90 dynamic memory allocation method allows allocation at runtime, when the individual size of the sub-grid is known to the PE's.

If, like in MOM, all declarations of arrays are done in include files, porting a code to distributed memory can be done with minor modifications. Only include files have to be changed and an additional initialization subroutine has to be called to do the allocation with the actual number of PE's and model size. Because of allocation at runtime a model can run on different numbers of PEs without rebuilding the binary.

Fritzsche et al. (1998) describe the parallelization of MOM's concept of a “memory window” for data distribution in meridional direction. The size of the memory window (JM) is an important value for the amount of memory because most of the 3-D arrays are only defined by JM in meridional direction, whereas the complete model has a dimension $J_{\text{MT}} \gg JM$. 

101
real, dimension (IM, JM, KM) :: a, b, c

do k=1, KM
  do j=1, JM
    do i=1, IM
      a(i, j, k) = b(i, j, k) + c(i, j, k)
    end do
  end do
end do

Figure 5: Sequential code version

real, allocatable, dimension (::,::,::) :: a, b, c

allocate (a(is_pe:ie_pe, js_pe:je_pe, KM))
allocate (b(is_pe:ie_pe, js_pe:je_pe, KM))
allocate (c(is_pe:ie_pe, js_pe:je_pe, KM))

do k=1, KM
  do j=js_pe, je_pe
    do i=is_pe, ie_pe
      a(i, j, k) = b(i, j, k) + c(i, j, k)
    end do
  end do
end do
end do

Figure 6: Parallel code version

Figure 7 shows the Memory/PE required by the North Atlantic model for different JM. The minimum value is JM=2. In case of JM=10 all grid points are within the window for 72 PEs or more. As also described in the paper mentioned above, the amount of computation and data movements decreases with larger JM. Thus the performance increases and production runs should be done in configurations where all points fit into the memory window.

3.2 Optimization

The original version of MOM was designed to run on vector computers. The optimization strategy for these systems is to vectorize as many loops as possible. If this can be
accomplished with long vector lengths the code will perform well.

RISC processors, however, require a different optimization strategy. They have a small memory bandwidth but different cache layers to overcome this problem. Long loops are not required. The optimization strategy here is to reduce the traffic to central memory by reusing data whenever it is possible.

T3E systems have another aspect to get better performance. To compensate for small secondary cache, the T3E can establish six parallel data streams from/to memory. The best performance will be achieved in loops which use exactly six data streams.

In MOM, there is no central subroutine which does the main part of the computation. The work is done in several different modules with a lot of mathematics involved. To get better performance on the overall code, all these modules have to be changed following some general rules:

- Reduce three-dimensional scratch arrays to one dimension or scalar.
  A typical computation subroutine in MOM contains series of 3-D loops. Data are passed from one loop to the next using 3-D arrays. Some of these arrays are local, i.e. they are only used to pass data between loop blocks in this program unit. Memory traffic can be reduced by collapsing the outer two loops and pass data in one dimensional arrays or scalars. This kind of data stays in cache.

- Swap loops or array indexing to get stride one.
  In MOM, there is no fixed order of array or loop indices. The best performance can be obtained by accessing as many arrays as possible with stride one. Reordering of loops or array indices is done.
• Collapse or split loops to get 6 streams from/to memory.

The code has been reordered to get loops which use exactly six streams from/to memory. That means, small loops will be collapsed and big loops will be split. Only the inner loop must be split and the data transfer will be done from one loop to the next using one-dimensional arrays. These arrays stay in cache and do not create additional streams.

After treating all major computation subroutines, a speedup of 2 has been achieved compared with the original version MOM2.1. The overall performance after optimization can be seen in the next chapter. Most of these optimizations give better performance on all RISC platforms, some (especially the stream optimization) are valid on T3E systems only. There is a tradeoff in creating hardware depending code and getting performance. The total amount of computation time spent in long term simulations with high resolution justifies the effort to make optimal use of the available resources.

3.3 Performance

On a coarse North Atlantic model, performance has been compared between a CRAY T3E-600 at AWI and a CRAY C90. Table 1 shows the elapsed time and overall CPU time of a 260 day simulation.

<table>
<thead>
<tr>
<th>TYPE</th>
<th>CPU’s</th>
<th>Elapsed-Time [s]</th>
<th>CPU-Time [s]</th>
</tr>
</thead>
<tbody>
<tr>
<td>C90</td>
<td>1</td>
<td>20784</td>
<td>7335</td>
</tr>
<tr>
<td>T3E-600</td>
<td>4</td>
<td>15319</td>
<td>60105</td>
</tr>
<tr>
<td>T3E-600</td>
<td>8</td>
<td>8225</td>
<td>64605</td>
</tr>
</tbody>
</table>

Table 1: Computation time on T3E versus C90

Times on the C90 have been measured on a loaded system. Assuming optimal conditions, elapsed time can be in the same order as CPU time and approximately 9 T3E-600 PEs show the same performance as one C90 CPU. These number clearly state the need of parallel systems to run big models, because there is neither a multitasking version of MOM available, nor can the required amount of memory be supplied on vector systems.

Overall performance and scalability is shown in Table 2 by running five days of the high-resolution North Atlantic model on different numbers of PEs. The model contains 356*260*45 grid points and has been computed on the T3E-900 at ZIB in Berlin.

<table>
<thead>
<tr>
<th>TYPE</th>
<th>NPEs</th>
<th>Time [s]</th>
<th>GFLOP</th>
<th>Mflop/PE</th>
<th>Factor</th>
</tr>
</thead>
<tbody>
<tr>
<td>T3E-900</td>
<td>18</td>
<td>932</td>
<td>1.16</td>
<td>64</td>
<td>1.0</td>
</tr>
<tr>
<td>T3E-900</td>
<td>36</td>
<td>535</td>
<td>2.18</td>
<td>59</td>
<td>1.7</td>
</tr>
<tr>
<td>T3E-900</td>
<td>72</td>
<td>348</td>
<td>3.58</td>
<td>50</td>
<td>2.7</td>
</tr>
</tbody>
</table>

Table 2: Performance of a 5 day North Atlantic model run on T3E-900. The measurement were done using the CRAY performance analysis tool PAT
Besides the required boundary exchanges between time steps, linear scalability is prevented by the 2-D conjugate-gradient (CG) solver described above. A CG-solver requires communication in every iteration step, and the summation around the islands, mentioned in Section 2, creates both, load imbalance and additional communication.

4 Results

All model configurations introduced in Section 2 have been run successfully on CRAY T3E systems on 8 to 128 processing elements, depending on the number of grid points. Due to the use of cpp ifdef - options for the parallelization, the code is still flexible and available to be used on both vector and parallel systems.

Several simulations of a North Atlantic model in 1/3° resolution have been done on the T3E system at ZIB in Berlin. Simulation periods of 30 and 15 model years have been processed on 72 and 60 PEs, respectively. The overall performance of 3.5 gigaflop enables the user to run simulations of this size in a couple of weeks time frame.

<table>
<thead>
<tr>
<th>Model</th>
<th>years</th>
<th># grid points</th>
<th>NPEs</th>
<th>PE-hours</th>
<th>Data [GB]</th>
</tr>
</thead>
<tbody>
<tr>
<td>NA</td>
<td>30</td>
<td>356<em>263</em>45</td>
<td>60 - 72</td>
<td>14000</td>
<td>2</td>
</tr>
<tr>
<td>4/3°Atl.</td>
<td>50</td>
<td>150<em>79</em>45</td>
<td>8 - 30</td>
<td>1400</td>
<td>0.2</td>
</tr>
<tr>
<td>1/3°Atl.</td>
<td>Test</td>
<td>600<em>308</em>45</td>
<td>16 - 120</td>
<td></td>
<td>3</td>
</tr>
</tbody>
</table>

Table 3: Overview of experiments carried out on T3E-900 and T3E-600 systems.

Some results will be presented addressing the effect of different horizontal resolution on the basin-wide circulation. To incorporate sub-grid scale features in coarse resolution models, i.e. turbulent mixing, these processes have to be parameterized. The resulting high explicit diffusion of T and S and the dissipation of kinetic energy tend to reduce the variability of ocean currents by an order of magnitude or more (Beckmann et al., 1994). Although large-scale circulation patterns and water mass properties are reproduced quite well, on a more regional scale deviations between model results and observations can be substantial. While in the coarse-resolution run (Figure 8) the North Atlantic Current appears to be a rather broad and sluggish stream, the high-resolution calculation (Figure 9) displays a highly energetic flow with intense eddy activity. Similarly, the observed high velocities (not displayed in the figure) of boundary currents in the East Greenland and Labrador Sea only show up in high-resolution models. Increasing the resolution and thus reducing the amount of explicit diffusion and dissipation increase the modeled variability to values much closer to observations.
Figure 8: Horizontal velocities at a depth of 100 m taken from the coarse resolution Atlantic model. Shown here is the North Atlantic Current.

Figure 9: as in figure 8 but from the high-resolution North Atlantic model.

To give a second example, an important aspect of today's discussion on climate change is the ability to distinguish between long-term trends and short-term anomalies. If climate models are to make a contribution, they have to be integrated for long time-spans to allow this differentiation.
Deep winter time mixing due to atmospheric surface cooling in the Labrador Sea, located between Canada and Greenland (see Figure 8 for geographical information), serves as an indicator for climate variability (Dickson et al., 1996). To improve our understanding of the dynamics that drive the ocean circulation and the climate system, systematic process studies are essential that allow parameter studies or the investigation of different mathematical and numerical formulations of the physical processes involved. Anomalies relative to a time mean of the modeled depth of winter time vertical mixing in the Labrador Sea are displayed for years 1958 to 1996 in Figure 10 (upper panel). If process studies are done over only a short period of time, this variability does not show up. Instead results of short time integrations may give the impression of a trend (Figure 10, lower panel).

![Anomalies of mixed layer depth in Labrador Sea](image)

Figure 10: Variability of the climate system may appear as a trend in short time integrations. (Courtesy of Carsten Eden, IfM Kiel)

## 5 Conclusion

This paper introduces a series of linked Atlantic models to enable experiments covering decades of numerical simulations aiming at a better understanding of the Atlantic circulation. For a systematic investigation of the dynamics of ocean currents it is not sufficient to have only one realization of a model integration - even at the highest resolution - at hand. Process studies, integrated over several years up to decades, based on one model configuration but differing in the numerical and mathematical implementation various physical processes, are the key towards an understanding of the ocean system.

Integrating ocean models in high-resolution require computation resources which can only
be supplied on large MPP systems. A parallel version of the MOM software using 2D domain decomposition enables long-term runs in an acceptable amount of time. A simulation of e.g. the North Atlantic in 1/3° resolution over a 30 model year period was accomplished in less than one month on the T3E at ZIB in Berlin. Future plans are long-term simulations of the 1/3° Atlantic model on at least 120 Processing Elements and integrating a 1/6° Atlantic model. Although the model setup as well as initial and boundary conditions are already existing, running a century of this new model requires computation resources which can only be supplied with the next generation of MPP systems.

References


Production codes in nonlinear plasma dynamics

A. Zeiler*, D. Biskamp, A. Celani, K. Hallatschek, W. C. Müller, E. Schwarz, R. Tisma

Max-Planck-Institut für Plasmaphysik, Germany

Abstract

Numerical codes, that are used on a T3E massively parallel system to simulate turbulence, are discussed. All four codes were originally developed on a vector computer and afterwards ported to the massively parallel platform. The achieved performance permits to address problems far beyond the capabilities of the previous system.

1 Introduction

The work of the nonlinear plasma dynamics group at Max-Planck-Institut für plasma physics is focused on two main subjects: plasma and fluid turbulence in a general sense and the turbulent transport in nuclear fusion research. Our general turbulence research covers such widely investigated systems as Navier-Stokes turbulence, magnetohydrodynamic (MHD) turbulence, thermal convection, and electron magnetohydrodynamic (EMHD) turbulence, with applications mainly in magnetically confined fusion research, astrophysics, and magnetospheric physics. Investigations in these areas require high-resolution nonlinear simulations, typically in three-dimensional space. Since most of the investigated basic phenomena are independent of particular boundary conditions and the special geometrical setup, the preferred numerical algorithms typically involve Fourier-transformations in all space directions. Thus, on a distributed memory architecture the performance is crucially controlled by the large amount of communication arising in the two- and three-dimensional Fast-Fourier-Transformation (FFT) algorithms. In nuclear fusion devices the turbulent transport controls the energy confinement and therefore determines, under which conditions a self-sustained fusion reaction can be attained. Thus, large experimental and theoretical efforts aim to understand the turbulence and to discover ways to regulate it. Since the drive mechanisms, that are responsible for the generation of the turbulence,

*email: asz@ipp.mpg.de
are closely linked to the particular three-dimensional geometry of the confining magnetic field, special nonstandard numerical techniques have been developed to efficiently perform numerical simulations. Unfortunately this excludes an extensive use of optimized library routines, hence porting the codes from vector computers to the distributed memory architecture is a challenging task.

In this paper we discuss the codes, that are currently under operation in our group, describing the underlying partial differential equations, the numerical algorithms, and the implementation on the T3E-system.

2 Three-dimensional Navier-Stokes, MHD and EMHD turbulence: the code EMHD-3D

The general fluid description of a fully ionized plasma consists of an electron fluid and an ion fluid, which selfconsistently give rise to electric and magnetic fields due to their opposite electric charges and thus interact with each other. On very short time scales the ions cannot keep up with the electrons, since the ion inertia is three orders of magnitude larger than the one of the electrons. Therefore, the electron fluid evolves on a background of homogeneously distributed stationary ions. Such a system is described by the electron magnetohydrodynamic equations

\[
\frac{\partial \mathbf{B}}{\partial t} + \nabla \times (\mathbf{j} \times \mathbf{B}) = 0, \tag{1}
\]

with

\[
\mathbf{B} = \mathbf{B} - \delta_e \nabla \nabla \mathbf{B}, \quad \mathbf{j} = \nabla \times \mathbf{B}, \tag{2}
\]

and \(\delta_e\) the electron collisionless skin depth. These equations are evolved in time by the three-dimensional code EMHD-3D. Fourier-transforming in all space dimensions allows a very accurate evaluation of the \(\nabla\) differential operators, but converts the nonlinear product of the form \(\mathbf{j} \times \mathbf{B}\) into a convolution. The most efficient method to deal with such terms (the pseudo-spectral method) consists of evaluating \(\mathbf{j}\) and \(\mathbf{B}\) in Fourier-space, transforming them to real space, where the product is taken, and back-transforming the result of the multiplication to Fourier-space. Once the second term in equation (1) has been determined, \(\mathbf{B}\) can be updated easily by discretizing the time derivative \(\partial / \partial t\) according to the trapezoidal leapfrog scheme. Thus, the main computational work consists of Fourier-transforming the complete three-dimensional vector field back and forth once per time step. Since the three-dimensional FFT is based on subsequent one-dimensional transformations in all three spatial directions, and since for a one-dimensional transformation all data need to be in the local memory of the processing element, that performs the transformation, the three-dimensional data set needs to be completely redistributed in memory several times per three-dimensional transformation. In the code EMHD-3D the FFTs including all communication are performed by the routines pscfft3d (configuration space to Fourier
space) and pcsfft3d (Fourier space to configuration space), that are available in the scientific library. To check the parallel efficiency we performed tests at a resolution of $128 \times 128 \times 128$ and $256 \times 256 \times 256$ modes, and observed almost perfect scalability up to 512 processors. The excellent performance properties enabled us already to carry out simulations with $512 \times 512 \times 512$ modes, a resolution that never before was reached in EMHD simulations. Due to the large resolution and the resulting amount of data special effort had to be made to obtain high I/O performance. Unfortunately the FFT routines from the scientific library require that the number of involved processors is a power of two, hence we cannot use all 784 processing elements. Since the number of modes must be a power of two in all three directions, the memory is insufficient to run at even higher resolution, which in principle would be possible as far as the performance of the code is concerned.

The electron magnetohydrodynamic equations are very similar to the \textit{incompressible Navier-Stokes equations}, which describe the turbulence of a neutral fluid. They can be written in the form

$$\frac{\partial w}{\partial t} - \nabla \times v \times w = 0,$$

$$w = -\nabla^2 A, \quad v = \nabla \times A.$$  \hspace{1cm} (3, 4)

Thus, a Navier-Stokes version of the EMHD-3D code is trivially obtained by letting $\delta_c = 1$ and dropping the additional $B$ in the definition of $\hat{B}$. In this version the code has been used recently to explore the statistics of the turbulence at the highest Reynolds-number attained so far (Ref. [1]).

Magnetohydrodynamic turbulence occurs on time scales that are long compared to the EMHD case and related to the ion inertia, whereas the electrons respond instantaneously to the fluctuations of the electric field. The \textit{incompressible magnetohydrodynamic equations} can be written in the form

$$\frac{\partial B}{\partial t} = \nabla \times v \times B,$$

$$\frac{\partial w}{\partial t} = \nabla \times v \times w - \nabla \times B \times j,$$

$$w = \nabla \times v, \quad j = \nabla \times B.$$  \hspace{1cm} (5, 6, 7)

Though the equations are more complicated than in the EMHD or in the Navier-Stokes case, we can use the same numerical approach as before, since the central task again consists of evaluating the nonlinear terms at the right hand side of equations (5) and (6). This version of the code is currently used to investigate statistical and spectral properties of high-resolution MHD turbulence.
3 The two-dimensional code EMHD-2D

The two-dimensional problems, that correspond to the previously discussed three-dimensional equations, still attract a lot of attention. To deal with these systems we have developed a two-dimensional counterpart to the EMHD-3D code with derived versions to solve the EMHD, the MHD, and the Navier-Stokes equations and to treat thermal convection. The numerical algorithm is completely analogous to the three-dimensional case, with the two-dimensional FFTs being computed with the routine pccfft2d. At present the code is used on up to 512 processing elements at a resolution of up to 16384 × 16384 modes. See Ref. [2] for recent results.

4 Anomalous transport in fusion devices

To investigate the turbulent transport in fusion devices in the collisional regime the complete system of two-fluid equations, which includes fluctuations of the particle density, the electrostatic potential, electron and ion temperature, the magnetic field, and the parallel fluid velocity, needs to be treated in a three-dimensional anisotropic and inhomogeneous geometry. Since the resulting equations are rather complicated we restrict ourselves to a brief description of the numerical approach without displaying the full equations (for a discussion of the physical model compare Ref. [3] and citations therein). Using the so-called local approximation we can regard the system as isotropic in the plane perpendicular to the magnetic field and apply a pseudo-spectral approach using FFTs as described before for these two dimensions. In the direction along the magnetic field, by contrast, the anisotropy of the magnetic field excludes this kind of algorithm, thus, we have to solve the third dimension in configuration space. An additional complication arises from the fact, that the equations comprise diffusive, parabolic contributions in the direction along the magnetic field, which need to be solved implicitly to avoid a severe time-step restriction. Thus, the numerical algorithm is set up in the following way: The equations are treated in Fourier-space in the plane perpendicular to the magnetic field. Since the nonlinear terms contain no derivatives in the direction along the magnetic field, they are of the same structure as the ones discussed in the previous sections, and can be treated in the same way. The nonlinearities and most other terms can be treated explicitly using a trapezoidal leapfrog discretization of the time derivatives. The implicit treatment of the diffusive terms in the direction along the magnetic field, however, requires to solve a tridiagonal system of equations.

Due to the nonstandard nature of our algorithm the use of library routines is prohibited. At first glance one would expect, that the two-dimensional FFTs that are involved in the nonlinearities can be directly taken from a library. Unfortunately this fails since the routines in the scientific library only handle the
situation, where the data to be transformed are distributed over the complete processor array, but not over part of it, as it is the case here. Since except for the nonlinearities the different Fourier-modes in the plane perpendicular to the magnetic field decouple the data are distributed in these dimensions, whereas the direction along the magnetic field resides completely on one processing element. The tridiagonal matrix solver, therefore, operates on local data only. To evaluate the nonlinearities the data need to be redistributed completely. Here the data are distributed over the direction along the magnetic field since the different planes decouple. The two-dimensional FFTs are computed by subsequent calls to the single processor one-dimensional routine ccfiff with an intermediate redistribution of the data. With the typical resolution of 96 × 96 modes in the perpendicular plane, and 90 collocation points along the magnetic field we achieve between 3.5 and 4 GFlops on an array of 128 processors. The most recent results obtained from our simulations can be found in Ref. [3].

To avoid the restrictions associated with the local approximation, a new numerical approach has been developed, that treats two spatial directions in configuration space and uses Fourier-transformations only in the direction perpendicular to the magnetic field and perpendicular to the equilibrium gradients. This algorithm allows to solve the full inhomogeneous problem. To avoid the previously discussed time step constraint the algorithm is fully implicit requiring to use a band-diagonal matrix solver for the two dimensions that are treated in configuration space. This is done by using the NAG routines f07bef (LU decomposition of complex band diagonal matrix) and f07bsf (LU back-substitution), where we need to recalculate and decompose the matrix only every \(10^3\) time steps, since it evolves slowly. The FFTs in the one dimension that is treated in Fourier space are computed with the scientific library routines csfft and scfft. The code typically runs on 256 processors with about 20\% overhead due to communication. For a discussion of the first results, that have been achieved with this code, compare Ref. [4].

5 Conclusions

Four different codes, that were developed and optimized on vector computers without caring for the special needs of a distributed memory architecture, were successfully ported to the T3E massively parallel system without any change of the numerical algorithms. The achieved speedup allows us to do simulations that are far beyond the capabilities of the vector versions, either with respect to the overall length of the simulation, or even more important, with respect to the total memory requirement.
References


Fortran Memory Management for Parallelizing an Operational Ocean Model

Tomas Wilhelmsson  
Department of Numerical Analysis and Computing Science  
Royal Institute of Technology  
S-100 44 Stockholm, Sweden  
towil@nada.kth.se

Josef Schüle  
Institute for Scientific Computing  
Technische Universität Braunschweig  
D-38092 Braunschweig, Germany  
j.schuele@tu-bs.de

9th September 1998

Abstract

The Swedish Meteorological and Hydrological Institute (SMHI) makes daily forecasts of currents, temperature, salinity, water level, and ice conditions in the Baltic Sea. These forecasts are based on data from a High Resolution Operational Model for the Baltic (HIROMB). This application has been parallelized and ported from a CRAY C90 to a CRAY T3E.

Our parallelization strategy is based on a subdivision of the computational grid into a set of smaller rectangular grid blocks which are distributed onto the parallel processors. The model will run with three grid resolutions, where the coarser grids produce boundary values for the finer. A set of F90 modules manage the different grids and blocks while hiding parallelization details. Only one grid block at a time is presented to the original F77 code. The linear equation systems for water level and ice dynamics are solved with a distributed multi-frontal solver.

We find that the production of HIROMB forecasts can successfully be moved from C90 to T3E while increasing resolution from 3 to 1 nautical mile. However, speedup and load balance could be further improved.

1 Introduction

The Swedish Meteorological and Hydrological Institute (SMHI) makes daily forecasts of currents, temperature, salinity, water level, and ice conditions in the Baltic Sea.

These forecasts are based on data from a High Resolution Operational Model of the Baltic Sea (HIROMB). Within the HIROMB project [1], the German Federal Maritime and Hydrographic Agency (BSH) and SMHI have developed an operational ocean model, which covers the North Sea and the Baltic Sea region with a horizontal resolution from 3 to 12 nautical miles (nmi). This application has been ported from a CRAY C90 parallel shared memory vector computer to the distributed memory parallel CRAY T3E. The memory and speed of the T3E will allow the grid resolution to be refined to 1 nmi while keeping the execution time within limits.
2 HIROMB: A High Resolution Operational Model for the Baltic Sea

The operational HIROMB code is loosely coupled via disk I/O with the atmospheric model HIRLAM [2]. Atmospheric pressure, velocity and direction of wind, humidity and temperature, all at sea level, together with cloud coverage are input, while sea level, currents, salinity, temperature, and coverage, thickness and direction of ice are output. HIROMB is run once daily and uses the latest forecast from HIRLAM as input. There are plans to couple the models more tightly together in the future.

The influence of the water level in the North Atlantic is accounted for by specifying tidal level at sea level at the open boundary between the North Sea and the North Atlantic. The sea level is provided by a storm surge model covering the North Atlantic. Fresh water inflow is given at 80 major river outlets.

2.1 Grids

The 3 nm grid, see Figure 1, covers the waters east of 6°E and includes the Skagerrak, Kattegat, Belt Sea and Baltic Sea. Boundary values for the open western border is provided by a coarser 12 nm grid covering the whole North Sea and Baltic Sea region, see Figure 2. In the vertical, there is a variable resolution starting at 4 m for the mixed layer and gradually increasing to 60 m for the deeper layers.

All interaction between the two grids is taking place at the western edge of the finer grid where values for flux, temperature, salinity, and ice properties are interpolated and exchanged.
2.2 Model Description

We can essentially identify three parts in the model, the baroclinic part, the barotropic part, and the ice dynamics.

2.2.1 Baroclinic Part

Water temperature and salinity are calculated for the whole sea including all depth levels. Explicit two-level time-stepping is used for horizontal diffusion and advection. Vertical exchange of momentum, salinity, and temperature is computed implicitly. Temperature and salinity both obey the same physical laws and are advected using the same subroutine tfLOW.

2.2.2 Barotropic Part

A semi-implicit scheme is used for the vertically integrated flow, resulting in a system of linear equations (the Helmholtz equations) over the whole surface for water level changes. This system is sparse and non-symmetric, reflecting the 9-point stencil used to discretize the differential equations over the water surface. It is factorized with a direct solver once at the start of the simulation and then solved for a new right-hand side in each time step.

2.2.3 Ice Dynamics

Ice dynamics include ice formation and melting, changes in ice thickness and compactness, and is taking place on a very slow time scale. The equations are highly nonlinear and are solved with Newton iterations using a sequence of linearizations. A new equation system is factorized and solved in each iteration using a direct sparse solver. Convergence is achieved after at most a dozen
iterations. The linear equation systems are typically small, containing only surface points in the eastern and northern part of the Baltic Sea. In the mid winter season however, the time spent in ice dynamics calculations may dominate the whole computation time.

3 Reorganization of the Code

As an initial effort it was decided to keep all algorithms and numerics in the parallel version the same as in the serial version of HIROMB. This was possible since we had access to a direct sparse solver of unsymmetric matrices for distributed memory machines, see Section 3.1. Our parallelization strategy is based on a subdivision of the computational grid into a set of smaller rectangular grid blocks which are distributed onto the parallel processors, see Section 3.3.

Serial HIROMB is written in Fortran 77. All new code added for parallel HIROMB make use of Fortran 90 features while much of the remaining original code is still Fortran 77. Fortran 90 pointers are used to switch context between blocks and the Message Passing Interface (MPI) library [3] is used to for updating block boundaries, see Section 3.4.

Software tools for for handling grids divided into blocks in this manner are available, notably the KeLP package from UCSD [4]. KeLP is a C++ class library providing run-time support for irregular decompositions of data organized into blocks. Unfortunately, we were not aware of this package when the HIROMB parallelization project started.

3.1 Parallel Direct Sparse Matrix Solution

In the original, serial version of HIROMB, the linear systems of equations for the water level and the ice dynamics were solved with a direct solver from the Yale Sparse Matrix Package (YSMP) [5]. For parallel HIROMB we employ a distributed multi-frontal solver written by Bruce Herndon at Stanford University [6]. This solver was originally used in a semiconductor device simulator.

The multi-frontal method seeks to take a poorly structured sparse factorization and transform it into a series of smaller dense factorizations. These dense eliminations can then efficiently be done by well known methods for dense systems. Many distributed sparse solvers have their own preferred decomposition. An advantage of Herndon’s solver is that it accepts and uses the application’s decomposition directly. Even with this non-optimal decomposition, the solver provides excellent solution capabilities. A disadvantage of the solver, however, is that it that the number of processors has to be a power of two.

Routines interfacing HIROMB with Herndon’s solver have been written that assemble the equations of every block in each processor into a local matrix partition. This local matrix partition is then extended with a set of shared equations before it is passed to the solver.

3.1.1 Distributed Multi-Frontal Factorization

The distributed factorization is based on a power-of-2 decomposition of the matrix. Equations that refer to values on remote processors are identified and ordered hierarchically into a global elimination tree as shown in Figures 3 and 4. These so called separator equations are shared among all processor located below in the hierarchy of separator levels.

The processors factorize their own local portions of the matrix independently and then cooperate with each other to factorize the shared portions of the matrix. One half of the processors will take part in factorizing shared equations at the lowest separator level, then a a quarter of them will factorize the next separator level, and so on, until only one processor factorizes the remaining equations at the root of the elimination tree.

If the fraction of shared equations is large, performance will suffer due to lack of parallelism when factorizing shared equations at the higher separator levels. New software is becoming available which parallelizes this stage further. The PARASOL library [7] uses standard dense parallel solvers to factorize the equations remaining at the root of the elimination tree. PARASOL will be available in the public domain by the end of 1998.
Figure 3: This four processor (PE) example shows how equations generated from a nine point stencil are classified. Equations on the first half of the PEs that involve equations on the second half are denoted level one separators. Recursively, equations on PE 1 involving PE 2 are level two separators as are equations on PE 3 involving PE 4. All other equations are local.

Figure 4: The separator equations are arranged hierarchically into an elimination tree.

3.2 Rectangular Arrays

The original serial vector code uses an indirect addressing storage scheme to store and treat only the grid points containing water. While efficient in terms of memory, this scheme gives low computational performance due to its irregular memory access pattern. As a starting point for parallelization, this scheme was replaced with a direct scheme storing the whole three-dimensional grid block. Land points are marked as inactive.

3.3 Decomposition into Blocks

In a rectangular block covering the whole 3 km grid only 25% of the surface points and less than 10% of the volume points are active. By decomposing the grid into a set of smaller rectangular blocks and discarding all blocks with no water points, it is possible to reduce the fraction of inactive points significantly. These remaining blocks are assigned to the processors in a parallel environment. We only cut the grid along horizontal dimensions and not between the surface and the sea bed.

The geometry of the Baltic Sea forces the blocks to have different sizes and shapes in order to obtain a good load balance and a minimal number of inter-block communication points. The more blocks the grid is decomposed into, the more inactive points can be discarded. On the other hand, overhead from switching block context and updating block boundaries will increase as the number of blocks increase and their size decrease. Hence there is an optimal number of blocks which minimizes the total execution time.

We use a domain decomposition package written by Jarno Rantakokko at Uppsala University [8]. The algorithm is summarized below:
Figure 5: This is a decomposition of the 3 nautical mile grid into 24 blocks distributed onto 16 color coded processors. No ice was present in this case.

1. The domain is first divided into equally sized blocks. Completely land filled blocks are removed in order to reduce the number of inactive points.

2. Blocks with a heavy work load are split further in order to get a more even workload. It is easier to obtain a good load balance and at the same time minimize the number of blocks if the blocks have approximately the same workload.

3. The blocks are distributed onto the processors using a data distribution algorithm like recursive spectral bisection (RSB) or recursive coordinate bisection (RCB).

4. Where it is possible, blocks on the same processor are combined and merged together in order to reduce the number of communication points within each processor.

5. As a final step, the number of inactive points are reduced further by shrinking the blocks where it is possible.

The estimated workload in each block is based on a performance model that takes into account both the number of surface as well as volume grid points. In the winter the fraction of ice coverage is included as well. As we currently use the same decomposition for both water and ice calculations, the decomposition during winters will have to be a compromise with unavoidable load imbalances.

An example of the 3 nm grid decomposed with this algorithm into 24 blocks on 16 processors is shown in Figure 5. One layer of ghost points is added around each block.

3.4 Multi-Block Memory Management

The original HIROMB code only deals with one block per grid. For parallel HIROMB we wanted to introduce several blocks per processor while still keeping as much as possible of the original code intact. This is done by replacing all fields in the original code by Fortran 90 pointers. By reassigning these pointers, the current block context can be switched.
Within the computation routines the array indices still refer to the global grid. Unfortunately, Fortran 90 pointers do not support lower array bounds other than the default 1. Therefore global grid indices cannot be used in the same routine where block context is switched. However, this proves to be of no major concern since block switching only have to take place in a few top level routines.

Below is an example of a basic code block in the parallel version of HIROMB:

```fortran
! Do first part of 'flux' routine
!
do b = 1, no.blocks
   if(.not. MyBlock(b)) cycle
   call SetCurrentBlock(b)

   call flux1(m1, m2, n1, n2, m0, mmax, n0, nmax, kmax, &
            fxe, fye, dt, khu, khv, casus, eisco, &
            ues, un, veis, vn, h, hy)
   end do

   call ExchangeBoundaries(/fxe_id, fye_id/)
```

When ghost points need to be updated within an original routine, the routine is split into several parts. This has taken place in the above example where the large routine `flux` had to be split into six parts with calls to `ExchangeBoundaries` in between. Note that the `flux1` routine does not know of more than one block at a time and may still be compiled with a Fortran 77 compiler.

The `ExchangeBoundaries` routine uses a list of neighbor blocks on order to update the ghost points. If a neighbor block is on the same processor, boundary values are copied without communication calls. For remote neighbor blocks we use buffered sends followed by receives to avoid dead locks.

The fields to be updated with `ExchangeBoundaries` are selected with a list of field id-numbers. All fields with the same dimension `n` and type, e.g., 3-dimensional reals, are stored together in an array with dimension `n+1`. This allows us to minimize the number of MPI calls when exchanging boundaries between blocks without reverting to explicit packing of messages. In the above example, the boundaries of both the field `fxe` and `fye` are transferred in the same MPI call. The MPI derived type mechanism could also have been used if separate data types are created for each combination of fields present in the calls to `ExchangeBoundaries`.

### 3.5 Multiple grids

Original HIROMB used two sets of arrays with different names, one for each grid. In parallel HIROMB the routine `SetCurrentGrid` switches context between grids in a manner analogous to how `SetCurrentBlock` switches block context. Data exchanges between different grids are currently performed via the master processor. The decompositions for each grid are completely independent and do not regard geographical localizations. More grids can be added with just minor code changes.

### 3.6 Distributing input data and assembling output data

Parallel HIROMB uses the same input and output files as serial HIROMB. File I/O is performed by the master processor and is to a large extent overlapped with computations by the workers. This is very important to achieve parallel efficiency. Due the constraints of Herndon's solver (see Section 3.1) the workers need to be a power of two. Hence, including the master, $2^n + 1$ processors are used in the simulations.

The master process also updates and distributes external influences to worker data. This prevents a complete overlapping of file I/O with computations by the workers.
Table 1: These timings were done for a full 12 hour simulation with data output every 3 hours using a CRAY T3E-600. At least 4 worker processors (5 with master) were necessary due to memory constraints. The coarse 12 nm grid was distributed onto no more than 8 processors on which it takes 0.49 seconds per time step. No ice was present in this data set from August 11, 1998.

<table>
<thead>
<tr>
<th>Processors</th>
<th>One time step only</th>
<th>Full simulation with I/O</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Time (sec)</td>
<td>Speedup</td>
</tr>
<tr>
<td>5</td>
<td>7.74</td>
<td>1.0</td>
</tr>
<tr>
<td>9</td>
<td>4.32</td>
<td>1.8</td>
</tr>
<tr>
<td>17</td>
<td>2.76</td>
<td>2.8</td>
</tr>
<tr>
<td>33</td>
<td>1.85</td>
<td>4.2</td>
</tr>
<tr>
<td>65</td>
<td>1.31</td>
<td>5.9</td>
</tr>
</tbody>
</table>

4 Timings and Results

Table 1 illustrates the performance obtained so far on a CRAY T3E-600. These timings were done for a full 12 hour simulation with data output every 3 hours. No ice was present in this data set.

The original serial HIROMB code runs the same example at 110 Mflop/s in 20.2 minutes on one C90 processor. This rather poor Mflop/s rate is partly due to the indirect addressing scheme employed in the original code. Furthermore, during the course of porting to the T3E, we were able to optimize one of the dominating routines in the original HIROMB version, tflow, making it eight times faster. On a SGI Power Challenge XL, this reduced the total simulation time by 50%. This optimization has not been introduced in HIROMB on the C90.

Table 1 shows that parallel HIROMB on 5 T3E-processors runs twice as fast as the original code on one C90 processor. However, the speedup is far from ideal. This has several causes. The most important one is that the parameters in the decomposition algorithm has not been tuned optimally yet which creates a non-optimal load balance. During wintens the inhomogeneous workload from the ice dynamics makes the situation even worse, since ice formation is normally confined only to the eastern and northern part of the Baltic Sea. Furthermore, the coarse 12 nm grid is only distributed onto 8 processors and take 0.49 seconds per time step. In the 65 processor case, this amounts to 37% of the total computation time. Finally, I/O operations on the master processor is currently not fully overlapped with computations on the worker processors. This is obvious when comparing the speedups with and without I/O.

5 Conclusions and Future Work

Moving the production of HIROMB forecasts from one C90 processor to 17 T3E-processors will reduce the elapsed time for making a forecast by a factor of five. The timings obtained so far indicate that a 1 nm grid computation can be successfully run in production on the T3E. There are still many possible enhancements that can be considered, for example:

- A better workload model for the decomposition algorithm. Timings have shown that the inactive grid points incur a computational cost that cannot be ignored to get a good load balance. The current work load model only considers active points.
- Introduce a separate distribution of the ice coverage to get a good load balance also for the ice dynamics calculation. This could be done using a small extension of the current block handling framework.
- Switch to the PARASOL sparse solver when it becomes available.
- Extend parallelism by running the different grids concurrently instead of sequentially.
Acknowledgments

The authors would like to thank Lennart Funkquist at SMHI and Eckhard Kleine at BSH for their help with explaining the HIROMB model. Computer time was provided by the National Supercomputer Centre in Linköping, Sweden and by the Institute for Scientific Computing in Braunschweig, Germany.

References


Migration of Production Plasma Physics Codes to the T3E

Alice E. Koniges\(^1\), Renate Dohmen, Gesa Czapaski, Bruce Scott

Max-Planck Institut für Plasmaphysik, Garching

Abstract:

The availability of an MPP platform is becoming the standard at most world class computer centers such as the Computer Center Garching where a Cray/SGI T3E offers the majority of high-end computing power. The production user is thus faced with the question of how best to use the resource for existing codes. These existing codes were designed for serial and parallel vector platforms, without anticipation of the MPP platform. We survey the conversion and utilization of the MPP resource for production computing. Our examples come from plasma turbulence, transport, MHD, and wave modeling. These examples are typical of production plasma physics codes, yet are not simplistic to parallelize (in contrast to plasma particle-in-cell codes which are nearly embarrassingly parallel). The original programming language of the codes is FORTRAN. We include experiences with library routines such as PETSc.

Introduction: The Resources for Plasma Physicists

Plasma Physicists in both the US and Germany have access to some of the most powerful computers in the world, and coincidentally, nearly identical architecture and size. In fact, their highest performing computers rank next to each other on the current Top 500 list\(^1\) in positions 7 and 8. (The Top 500 list categorizes parallel computers based on the Linpack Parallel Benchmark series for the best performance on a large compute intensive problem.) At the Rechnung Zentrum, Garching, the computer center of the Max-Planck Society of which plasma physics is an institute, there is sited a Cray/SGI T3E-600 with 784 applications processors. It has an \(R_{\text{max}}\) of 342800 Gflops, and an \(R_{\text{peak}}\) of 470400 Gflops. Similarly, most of the plasma physicists in the United States have access to the National Energy Research Scientific Computing Center’s 512-processor Cray/SGI T3E-900 with an \(R_{\text{max}}\) of 321100 Gflops, and an \(R_{\text{peak}}\) of 460800 Gflops.

Because of these sophisticated resources, all plasma physicists doing computational work today are faced with the decision of if and when to “go parallel.” Particle codes migrated naturally to the early parallel machines. This was due to two essential reasons. Firstly, the codes were memory-bound on conventional smaller memory vector supercomputers and readily used the enhanced memory capability of the new MPP machines. (For instance, at NERSC the memory of the vector C90 was roughly 1/10 of the memory of the first production MPP machine, a CRI T3D MPP.) Secondly, the problem has a natural parallelism, one simply assigns particles to a processor. Of course, since the particles move and interact with the field equations, there are many factors to be considered in PIC “particle-in-cell” codes on parallel machines, but these were dealt with from the very beginning. However, for the average computational plasma physicist, the overriding need to port or re-design for the MPP is not as essential. Since parallel computing is complicated for the production user to learn, and existing codes are often designed primarily for serial and vector machines, the decision about going parallel is more involved. However, given the resources and the available tools, e.g., parallel software (libraries, message passing standards, debuggers, etc.) that are now available many codes are considered for the move to the MPP platform. In this paper, we detail a few of these efforts. We consider five specific codes, the B2 transport simulation (R. Schneider), the UEDGE transport simulation (T. Rognliein), Toric ion cyclotron wave simulation (M. Brambilla), the NIMROD non-linear MHD

\(^1\) On leave from Lawrence Livermore National Laboratory.
with rotation (C. Sovinec), and the DALF 3D drift-fluid turbulence (B. Scott). Most of these codes are multi-person efforts; therefore we denote a contact/primary developer in parentheses. Each of these codes solves a complicated series of nonlinear partial differential equations for a number of plasma parameters.

**Characteristics of the Plasma Codes and the Plasma Geometry**

The major goal of these plasma codes is to simulate tokamaks and stellarators. These fusion devices are characterized by a variety of length and time scales, which differ by orders of magnitude. This makes problems in plasma physics generally ill conditioned. Additionally, there is often a complicated interaction of species (multi-species ions and electrons) and field effects coupled to fluid behavior. Traditionally, plasma physicists use a broad range of approximations to form a tractable numerical problem in a complex 3D geometry, which is shown in Fig. 1 for a tokamak.

![Figure 1. Diagram of a tokamak. Physics of the device leads to a natural decomposition for parallel direction, and much slower variation in the toroidal direction. This physics has implications for the parallel decomposition of plasma simulations. We show how it is often advantageous to use domain decomposition in the poloidal plane, where a large number of points are required to simulate the rapid variation. For a given portion of the poloidal plane, the variation in the toroidal direction can remain on the single processor using actual geometry, field-line geometry or Fourier modes.](image)

**B2 and UEDGE Transport Codes**

One fundamental problem in plasma physics is the description of transport in the poloidal plane. Transport can be modeled by a coupled set of fluid equations (essentially mass, momentum and energy conservation) for the various species of ions and electrons in the plasma[2]. Figure 2 shows a cross-section of the poloidal plane. Fully implicit solution of transport equations leads to a large linear system[3]. One successful solution technique is to use a pre-conditioned Krylov solver method. Because of the fully-implicit formulation, the problem is memory-bound on conventional parallel vector supercomputers. In the UEDGE code, a parallelization
effort has been successful based on the following components, domain decomposition in poloidal plane and simple pre-conditioning with no ghost cells[3].

The end result of the UEDGE project gives reasonable scaling on T3E, but it saturates at 16 processors with a speed-up of about 8. Of course the new code has the ability to solve larger problems. The project went from a serial code to a parallel code in a little more than one year. The was facilitated by the fact that the serial code was based on a library solution. This library, maintained by a parallel-computing group outside of plasma physics, was updated for parallel machines, allowing the plasma code to incorporate the new parallel library with a relatively low amount of effort. A primary stumbling block for the developers was the fact that the original (serial) code coupled to a scripting language known as BASIS[4]. BASIS is not suitable for a distributed memory MPP such as the T3E, so the new parallel code must be run either without the pre-processing scripting language or with the BASIS code running separately on a workstation. Note: some communication between the BASIS code and the T3E is possible, see “Remote Visualization over Standard Network Connections,” this volume, for an interesting discussion of communications between workstations and the T3E[5]. The developers are also looking into using Python[6] as a replacement for BASIS.

The B2 code solves the same set of coupled fluid equations by a different method[2].

B2 solves each equation (i.e., parallel velocity, electron temperature,...) sequentially, iterating through the coupled set of equations a certain (small) number of times until convergence. This leads to much smaller matrices to invert, -- corresponding to each evolution equation. B2 is thus not memory bound, but the parallelization focus is on problem speedup. For B2, the following routes to parallelization were investigated:

- Case A: parallelize over the species
- Cases B, C: parallelize the solver, either with or without species parallelization
- Case D: domain decompose in poloidal plane

Additionally, to speed the process, the parallel library PETSc [7] was considered. These are considered diagrammatically in Fig. 3 for cases A-C. In the species parallelization case, Fig. 3A, we considered functional parallelism as opposed to data parallelism. In other words, the parallelism is based on giving a species to each processor rather than decomposes the data. At the next level, Fig. 3B, we show the parallelization being increased as the individual matrices to be inverted are selected for data parallelization. Neither of these techniques proved successful for improving code performance as is indicated in Table 1. Firstly, the amount of communication between species for source terms requires too much communication to make the parallel division by species efficacious. Secondly, the small size of the matrices to be inverted limits the effectiveness of parallelizing the solvers.
A) Species Parallelization (up to ~20)
B) Solver parallelization ~4000 equations
C) Solver parallelization ~ 4000x20 unknowns

Figure 3. B2 Parallelization choices.

<table>
<thead>
<tr>
<th>nprocs</th>
<th>1</th>
<th>2</th>
<th>4</th>
<th>8</th>
<th>11</th>
<th>16</th>
</tr>
</thead>
<tbody>
<tr>
<td>time[s]</td>
<td>2.172</td>
<td>1.288</td>
<td>0.850</td>
<td>0.634</td>
<td>0.678</td>
<td>0.565</td>
</tr>
<tr>
<td>Speedup</td>
<td>1.680</td>
<td>2.560</td>
<td>3.430</td>
<td>3.200</td>
<td>3.840</td>
<td></td>
</tr>
<tr>
<td>Efficiency</td>
<td>84%</td>
<td>64%</td>
<td>43%</td>
<td>29%</td>
<td>24%</td>
<td></td>
</tr>
<tr>
<td>time for MatSerVal</td>
<td>0.644</td>
<td>0.320</td>
<td>0.172</td>
<td>0.098</td>
<td>0.080</td>
<td>0.062</td>
</tr>
<tr>
<td>time for MatAssembly</td>
<td>0.172</td>
<td>0.074</td>
<td>0.055</td>
<td>0.050</td>
<td>0.047</td>
<td>0.040</td>
</tr>
</tbody>
</table>

Table 1. B2 Parallelization Timings for solver parallelization with approximately 4000 equations, corresponding to Case 3B.

In order to increase the potential parallelization, timings were performed on parallelizing just the matrix solve without decomposition by species. This is diagrammed in Fig. 3C and the results are given in Table 2. Here again, issues associated with Amdahl’s law limit the performance.

<table>
<thead>
<tr>
<th>nprocs</th>
<th>1</th>
<th>2</th>
<th>4</th>
<th>6</th>
<th>8</th>
<th>10</th>
<th>11</th>
</tr>
</thead>
<tbody>
<tr>
<td>time [s]</td>
<td>2.17</td>
<td>1.19</td>
<td>0.65</td>
<td>0.52</td>
<td>0.50</td>
<td>0.50</td>
<td>0.38</td>
</tr>
<tr>
<td>Speedup</td>
<td>1.62</td>
<td>3.34</td>
<td>4.17</td>
<td>4.34</td>
<td>4.34</td>
<td>5.71</td>
<td></td>
</tr>
<tr>
<td>Efficiency</td>
<td>91%</td>
<td>88%</td>
<td>70%</td>
<td>54%</td>
<td>43%</td>
<td>52%</td>
<td></td>
</tr>
</tbody>
</table>

Table 2. B2 Parallelization timings for solver parallelization with approximately 4000 equations and 11 species, corresponding to Case 3C.

As a final try for reaching a parallel decomposition, we give projected efficiencies for domain decomposition in the poloidal plane. These estimates are based on theoretical analysis of the arithmetic operations and coordination with Amdahl’s law for efficiency. These give reasonable predictions for code improvement, although they have not yet been implemented.

<table>
<thead>
<tr>
<th>ns</th>
<th>#PE</th>
<th>B2-Speedup</th>
<th>Total-Speedup</th>
<th>Efficiency</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>16</td>
<td>5.88</td>
<td>7.67</td>
<td>98%</td>
</tr>
<tr>
<td>64</td>
<td>8</td>
<td>7.04</td>
<td>13.88</td>
<td>87%</td>
</tr>
<tr>
<td>64</td>
<td>16</td>
<td>8.28</td>
<td>35.41</td>
<td>55%</td>
</tr>
<tr>
<td>64</td>
<td>64</td>
<td>8.28</td>
<td>20.30</td>
<td>32%</td>
</tr>
</tbody>
</table>

Table 3. Theoretical predictions for speedup and efficiency of B2 parallelization based on domain decomposition in the poloidal plane.

GLOA/TORIC Wave Code

Our third example is the TORIC wave code. This code solves the finite Larmor radius wave equations in the ion cyclotron range of frequencies in arbitrary axisymmetric toroidal geometry [8]. Numerical solution is based on the spectral representation of the solution in the poloidal angle and cubic finite elements in the radial variable (2D)
Analytical simplifications lead to the solve of a very large block-banded complex matrix.

The code is memory limited on the J90, leading to the desire for a move to the parallel machine. In this case, we started with a legacy Fortran code, and considered the use of library routines for parallel implementation. A timetable of results is given below.

- 3/98 Started port of J90 to T3E
  - replaced FFT with new routines
  - converted to F90 array operations
  - added F90 allocations
  - replaced solver calls with LAPACK routines
- 5/98 Started experiments with PETSc Library
  - problems with FORTRAN interface
  - difficult to benchmark, since standard cases will not fit on one processor
- 9/98 Testing/Validation stage for code with parallel PETSc solver

**NIMROD 3D MHD Code**

NIMROD is a finite-element based magnetohydrodynamic (MHD) fusion energy code. A team of scientists who are physically located at several different laboratories is developing the code. The NIMROD effort is not a code conversion, but rather the code was designed from the onset to be parallel. NIMROD’s parallel algorithm has good scaling on the T3E [9]. We use it here to point out that the “natural parallel decomposition” of the plasma problem in the poloidal plane is not without pitfalls.

NIMROD represents the poloidal simulation plane as a collection of adjoining grid blocks; the toroidal discretization is pseudo-spectral. Within a single poloidal block the grid is topologically regular to enable the usual 2-D stencil operations to be performed efficiently. Blocks join each other in such a way that the individual grid lines are continuous across block boundaries. Within these constraints, quite general geometries can be gridded, and parallelization is achieved by assigning one or more blocks (with their associated toroidal modes) to each processor. In parallel, the only interprocessor communication that is then required is to exchange values for block-edge or block-corner grid points shared by other processors. For general block connectivity, this operation requires irregular, unstructured interprocessor communication. We precompute the communication pattern and then exchange values asynchronously. This enables the block-connection operation to execute efficiently and scalably on any number of processors.

Spatial discretization in the poloidal plane is accomplished with finite elements and a non-orthogonal Eulerian grid. The toroidal coordinate is treated with FFT’s which live wholly on a single processor. The non-orthogonal poloidal grid consists primarily of unstructured blocks of structured quadrilaterals (RBLOCKs). In the poloidal plane using the RBLOCK representation, the dependent variables are approximated with bilinear elements, and metric elements are treated as bi-cubic splines. In the core of the plasma the grid is chosen to be closely aligned with the initial axisymmetric (n = 0) flux surfaces. (Occasional regridding may be required as the axisymmetric component evolves.) Non-conforming RBLOCKs can be joined with blocks of unstructured triangular elements (TBLOCKs). A TBLOCK is also introduced to represent the region around the magnetic axis, thus avoiding the coordinate singularity inherent in pure flux coordinates. The initial choice to avoid the singularity while avoiding bottlenecks in the parallel decomposition was to overlay a patch of RBLOCKs at the axis. This is shown in Figure 4. This led to numerical inaccuracies, however, and was replaced by a pie-shaped collection of triangle elements, a TBLOCK, as shown in Figure 5.
Although the scalability of the code is excellent (nearly linear up to 64 processors), when compared with the "best sequential code performance," improvement is required. Thus the NIMROD developers are looking to more efficient parallel algorithms for the basic solve routine, possibly incorporating parallel libraries.

Figure 4. Quadralateral Grid, overlayed on magnetic axis in poloidal plane led to inaccuracy at corners of patch.

Figure 5. Patching a triangular block to the quadralateral grid removed the numerical inaccuracies associated with the corners. Allowing the entire Tblock to live but the communication between grid cells across processors is more complicated.

DALF Turbulence Code

The DALF turbulence code represents a highly anisotropic nonlinear system with fluid drift perpendicular to B (xy-plane), Alfvén wave propagation parallel to B (s-direction), and field solves in x and y [10], in a coordinate system aligned to B [11]. The xy-plane requires high resolution. The code is domain decomposed over the processors in the s-direction. A second order upwind method is used to solve the CFD-like equations, and 2 ghost cells are retained on each processor boundary point. This is another example of a code where the successful parallel implementation was envisioned from the start of code development. Since the code represents turbulence, it also uses the concept of functional parallelism. Namely, many realizations of the turbulence are required for certain problems based on the initialization of the field. Dividing the realizations among
the processors yields additional parallel performance improvement. This parallel implementation was done entirely in MPI, making use of some of the Cartesian processor grouping routines to speed development. Figure 6 shows planes of turbulence. Each xy-plane resides on a single processor. Speedup is currently measured at 15.5 for 16 CPUs.

A timetable for DALF development is given below.

- 6/1996: Electrodynamics added
  - fully explicit on T3D
  - message passing: SHMEM
  - Problems: I/O (front end), radiative solve for electrodynamics is not very accurate
- 8/1996: Electromagnetic DALF on J90
  - production code for approximately 1 year, difficulty getting autotasking to emulate message pass
- 1/1998: Port of DALF to T3E
  - parallelize Alfven dynamics, message passing with MPI
  - Cartesian processor grouping routines prove very useful for exchanges
- 9/1998: Parallelization of field solvers (MG, Tri-D) in progress

**Conclusions**

We start the conclusions with a list of the obstacles that we found in converting codes for the parallel machine:
• I/O not confined to single place in code
• Significant serial code in hard-wired algorithm
• Missing/outdated libraries, but, fixing this can lead to much better performance
• Outdated pre-processors or scripting languages (BASIS)

In our 5 examples, 2 codes explicitly implemented MPI calls and 3 used library routines for the solution of a system. Most mathematical parallel software libraries are written in C. This can lead to some problems with production Fortran codes.

Consider the following typical interchange:

USER: “We are trying to use the PETSc library to solve a complex linear system. We have installed the petsc-2.0.2 library on our T3E ... and we get the following error ...”

PETSc Response: “Yes, we understand the problem. The next release of PETSc due in a few days will resolve the problem (though sometimes with a performance penalty). The problem is with the call to VecGetArray() from Fortran. The Fortran compiler is not complex aligning its complex number arrays... Meanwhile, you can get around it by... ”

This is not to imply that libraries are unsupported or not well written. It just points out that they require a certain amount of sophistication to use properly, especially when coupling between FORTRAN codes and C-libraries is involved.

Also, we note that parallel software libraries are often updated and require some expertise and time to install. Thus we do not recommend their use for average users unless they are supported by the computer center which administers the machines. Codes using “homegrown” libraries had the easiest time, since the library developers were on site and tuned to the particular application.

In summary, most plasma codes, which do not fit on a workstation, are being run on or converted to MPP architectures. Generally, it takes one year or more to convert a code. Codes designed with parallel processing in mind avoid the bottlenecks associated with Amdahl’s law. Loop parallelism was not utilized in any of the examples; message passing was the overwhelming choice. Memory limitation is as important as speed-up concerns in making the decision to go parallel. In general, we predict that more and more plasma codes will go 3D for designing new machines and thus memory requirements will make the MPP platform very inviting. Finally, we predict that analytic approximations for the sole purpose of making the governing plasma equations numerically tractable (e.g. approximating an integral) will decrease as the compute power continues to lessen their advantages. The majority of plasma/fluid codes will use domain decomposition in the poloidal plane, with each toroidal mode living on a processor as the route to parallelism.

References

Parallelization of a global ocean circulation model with coupled seaice

B. Frittsch\textsuperscript{1}, R. Johanni\textsuperscript{2}, and C. Köberle\textsuperscript{1}

Alfred Wegener Institute for Polar and Marine Research, Bremerhaven
SGI/CRAY

1 Introduction

Oceanographic numerical models require a large amount of computational resources, even more when coupled with seaice and/or atmospheric models. A typical experiment in climate resource would need several hundreds of modelyears to reach a stationary state. One way to reduce the wallclock time of such an experiment is to use parallel computers, which in turn require special programming techniques and specifically designed numerical algorithms.

In this paper the parallelization of a global ocean model with incorporated seaice for a CRAY T3E computer is discussed. The code of the ocean model is based on Modular Ocean Model (MOFM). It was developed at GFDL in Princeton and is used by a large community of ocean modellers. The seaice model based on a model of Hibler (2).

For parallelizing the program a data decomposition technique is used which leads to a data parallel programming model. In MOM the memory window technique is exploited, which slices the three dimensional arrays along circles of constant latitude. It was originally implemented in the scalar code to reduce the requirements on main memory. To get scalability up to higher number of processing elements a further splitting of these slices is done resulting in a two dimensional data distribution. The communication between processing elements is made via shmem library of CRAY.

Parallelization of the program is done widely transparent to ensure acceptance of the parallel code for many users.

2 Parallelization of MOM

The main part of the Modular Ocean Model MOM is the time stepping loop, which contains two relatively separated tasks:

- Solving the baroclinic and tracer equations
- Solving the barotropic equation

The baroclinic part involves computations over the whole three dimensional area, whereas the barotropic part consists mainly of the solution of the two dimensional elliptic stream function equation. Depending on the number of depth levels and the number of tracers used, the barotropic part needs significant less
CPU-Time than the baroclinic part (about 2 - 15% of the total time). Never- 
theless care has to be taken to parallelize the barotropic part, too, otherwise 
Amdahl Law would limit the number of usable processors to about 10 to 20.

For the same reason other parts in the ocean model like initialization, forcing, 
diagnostics, and output of results have to be regarded in parallelization, too.

2.1 Baroclinic and tracer part

When solving the baroclinic and tracer equations in MOM, the so called "memory 
window" approach is used. This technique was originally chosen due to the 
lack of computer memory for bigger 3-dimensional models. The basic idea is to 
keep the state variables (i.e., velocities, temperature, salinity, and tracers) on disk 
and to perform the calculations only on a small strip in east-west-direction (the 
"memory window") at a time. After solving the baroclinic and tracer equations 
for this strip the newly calculated variables for the next timestep are written 
to disk and the adjacent northeasterly memory window is loaded into memory until 
the whole area is processed.

For calculating one row of variables at the next time step, the neighboring 
rows of the former time steps are also needed, so the memory window has to 
contain more rows than those actually to be computed. In the original version 
of MOM, if n rows of variables are calculated, there has to be n+2 rows in the 
memory window (one additional row at both borders), but in the actual versions 
higher order schemes are employed and things got more complicated.

Since computers nowadays have much more memory than in the times when 
MOM was designed originally, the time consuming disk I/O is now normally 
replaced by using a ram-disk. This is just a big array where all the state vari- 
ables are stored within the program. However the basic concept of the memory 
windows is still used in MOM. The reason is that the memory window tech- 
nique provides a big memory saving, even when using a ram-disk: There are a 
lot of arrays needed for intermediate values. All these arrays have to be dimen- 
sioned only to the size of the memory window and not to the size of the whole 
computational area.

At a first glance it looks like this approach would be easy to parallelize and 
the documentation of MOM states that in former times there existed a parallel 
version of MOM (for shared memory systems) based on the parallel computa- 
tion of memory windows.

In the current version of MOM, such a simple approach to parallelization can 
not be made because of the following two reasons:

Many intermediate computation results at the northern rim of one memory 
window are also needed at the southern rim of the following memory window. 
Instead of computing these results anew, they are just shifted from the north to 
the south before the computation of the next memory window.

Things get even more complicated when higher order schemes are used for 
solving the baroclinic equations (i.e. when more than one neighboring row is 
needed to calculate the values of a certain row at the next time step). Due to a 
clever usage of the shifting technique for intermediate results described above,
not even all rows needed to calculate a certain row at the next time step are kept in the memory window at a time. This implies that the memory windows have to be calculated in a certain order, namely strictly from the south to the north.

Furthermore, a parallelization only on the base of the memory windows would mean a 1-dimensional domain decomposition which limits the scalability of the code much more than a 2-dimensional decomposition.

Therefore the following approach has been used in the parallelization of the baroclinic and tracer equations:

A 2-dimensional decomposition in North-South and East-West direction is used (a parallelization over the depth layers is nearly impossible due to certain physical characteristics of the ocean model). This leads to a scalability up to a much higher number of processors than a 1-dimensional decomposition. Nevertheless, the number of processors in the East-West direction should be kept smaller than the number of processors in the North-South direction, since all inner loops of the program are running in East-West and if the length of these inner loops is getting too small it degrades the overall computational performance.

Within the computational region of each processor, the memory window technique is used to reduce the memory needed for intermediate result arrays. The size of the memory windows may be chosen freely and completely independent of the size of the computational region. With the exception of the processors at the southern rim, the first memory window has to start a certain number of rows south of the real computational area in order to set correct results with higher order schemes. For the same reason the last memory window must end a certain number of rows north of the real computational area (some special treatment necessary for the northern rim). The computations in East-West direction are overlapping in the same way.

Each processor keeps a ram disk of its own region plus a certain number of neighboring cells. The exact number of those boundary lines depends on the order of the integration scheme used. For a second order scheme one boundary line is needed at every side, for a fourth order scheme two boundary lines are needed. The "Flux-Corrected Transport Scheme" ([9]) in MOM needs even one more boundary line.

In every time step, before starting to solve the baroclinic and tracer equations, the boundary values of the ram disks are exchanged between neighboring processors. No further communication is needed during the calculations for this time step.

Diagnostics are gathered on a per processor basis. When output of the diagnostics is requested, global sums are calculated over the per processor results. Special care has to be taken that the overlapping cells at processor boundaries are counted only once.

When output of the results is requested (snapshots), it has to be done immediately after a latitudinal row is calculated, since the output files contain also intermediate data, which are no longer available after a new memory window is
loaded. In the parallel version, all processors working on the same latitudinal row are synchronizing after this row is calculated. The data are gathered on the first processor of this row and output onto an own file for each writing processor. After completing the calculations over the whole model area there exist as many files as subdivisions in North-South direction. These files have to be copied to one single output file in a separate step.

After the computation of the baroclinic and tracer equations, all processors have to synchronize at a barrier. The only variable needed for the following barotropic part (the solution of the stream function equation) is the forcing (right hand side) for that equation. Every processor broadcasts its part of that array to all others. This is only necessary, because there exists an option for testing purposes that every processor should solve the complete equation. In this case, the complete array is needed, of course.

2.2 Barotropic equation

The barotropic equation is solved using a conjugate gradient solver. Parallel solution techniques for CG solvers are well known and need not to be described here in detail. A boundary exchange is needed at each step and the dot products used in the CG method have to be build over all processors using global sums.

At each step there has to be built a boundary integral of the residual around every island. In the parallel version, a processor sums up only the boundary points in its own region, a global sum has to be made to get the complete boundary integral.

3 Seaice Model

Quality of ocean models depends strongly on a realistic forcing. Mainly in polar regions the available data (e.g. surface temperature) show major uncertainties. This results in an incorrect physical behaviour of the model. An improvement is achieved by coupling a seaice model to the ocean.

The parallelization strategy is a two dimensional domain decomposition, too.

When applying the icemodel over the complete area where the ocean model works the ice model needs about 10 - 20 % of cpu time. Thus the region for icemodel should be limited to only that areas where ice can occur (see fig. 1). For a global model, the two polar regions are disconnected areas, in which the icemodel works. This means that regions for each processor can differ in icemodel and ocean model. In the coupled model, the submodels are treated separately from each other, as far as possible. Only some coupling routines have information (like dimension) from both models. Therefore the exchange of data between ice and ocean and vice versa will be made only by one processor which broadcasts the data dependend on the actual domain allocation of each PE in the submodel actually under consideration.

In parallelizing the ice model we tried to keep the modifications in the original code as small as possible. The modifications in the model are concerning the following parts:
Fig. 1. Two separate ice regions in the global coupled model.

- A setup routine was written, where the domain decomposition for the ice model is done. All local values such as loop ranges and IDs of neighbouring processors are calculated for each processor. In the global model with two separate ice regions the communication between processors working on different regions must be cut off.
- Interface between ice and ocean. Only the master PE gets the information from the other submodel and broadcasts input data to the other PEs and gathers the output.
- In the model code itself, only some calls to interprocessor communication routines were inserted. In addition, only a few loop ranges had to be modified where only inner points are considered.
- In the original code a red black relaxation is used to solve the momentum equation. Unfortunately a coupling between points of same colour exists, which makes it impossible to parallelize this algorithm. A four colour relaxation was implemented to ensure equal results for different numbers of processors.

4 Results

In coupled models with the previously discussed submodels for ocean and ice we find following routines which are time critical for the whole model:

- in ocean
• TRACER: Solving the tracer equation
• CLINIC: Solving the baroclinic equation
• TROPIC: Baroclinic part
  in ice
• DRIFT: Solve momentum equation
  coupling between submodels and input of atmospheric forcing

The percentage of CPU time each routine consumes depends on the configuration of the model (dimension, number of tracers, kind of advection scheme etc.). For an example we consider a global coupled model with 194 x 92 points, 29 depth layers. Beside temperature and salinity three passive tracers are introduced. For advection the fourth order FCT is used.
For our models we have following approximate division of time:

<table>
<thead>
<tr>
<th>Submodel</th>
<th>Percentage</th>
</tr>
</thead>
<tbody>
<tr>
<td>ocean</td>
<td>TRACER 65%</td>
</tr>
<tr>
<td></td>
<td>CLINIC 8%</td>
</tr>
<tr>
<td></td>
<td>TROPIC 15%</td>
</tr>
<tr>
<td>seaice</td>
<td>DRIFT 7%</td>
</tr>
<tr>
<td>coupling and forcing</td>
<td>2%</td>
</tr>
</tbody>
</table>

The most time consuming parts are in the ocean model. Thus most attention is payed to the optimization of these routines (see also [4] at this workshop). The performance data measured for 16 PEs on a T3E-600 at AWI are satisfactorily for the ocean model.

<table>
<thead>
<tr>
<th>Region</th>
<th>Performance</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>total (Gflop) per PE (MFlop)</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>TRACER</td>
<td>1.07</td>
</tr>
<tr>
<td>CLINIC</td>
<td>0.70</td>
</tr>
<tr>
<td>TROPIC</td>
<td>0.50</td>
</tr>
<tr>
<td>DRIFT</td>
<td>0.29</td>
</tr>
</tbody>
</table>

In our configuration the scalability of the total model is limited by the barotropic part in the ocean and the icemodel whereas the baroclinic and tracer part scale well. The performance of the icemodel is low. Here the algorithm for solving the momentum equation has to be improved.
For moderate number of processors the limitations due to the exchange of data between submodels via one PE are not important, but they will be in the case of higher numbers. For that case the interface routines has to be modified.
Fig. 2. Speedup for main routines in a coupled model, namely TRACER (---), CLINIC (-----), TROPIC (o-o-o), DRIFT (---) and for the total program (—)

References

Large Scale Simulation of Fluid Flow in Suspensions and Porous Media

Lorna Smith

Edinburgh Parallel Computing Centre
Simon Jury, Peter Bladdon and Mike Cates
Department of Physics and Astronomy, Edinburgh University

August 21, 1998

1 Introduction

The Maxwell Institute is a new international multi-disciplinary centre devoted to the use of computer simulation in science, set up by the University of Edinburgh. Edinburgh Parallel Computing Centre (EPCC) is closely involved, collaborating in providing high performance computing consultancy and support.

The initial research program has involved the development of a dissipative particle dynamics (DPD) code to study the hydrodynamic and rheological properties of complex fluids. For example, the phase separation of a binary fluid mixture can be studied using this technique. Originally developed by the EPCC-supported HPCI Colloid Hydrodynamics Consortium, the code has been extended to consider dimers, with a view to studying surfactants, and to consider the effect of shear using Lees-Edwards boundary conditions. With the final code, including dimers and the effect of shear, using one million particles, a parallel efficiency of 80 percent was achieved going from 64 to 256 processors on the T3D.

2 Theory

Previously, some researchers have used molecular dynamic techniques to simulate fluid systems. These studies have been limited by computational cost to studies in two dimensions and relatively simple fluid systems. Alternatively, lattice gas automata type simulations offer a more computationally efficient method of simulating fluid flow. This technique involves less operations per particle and per time step with the time steps usually orders of magnitude larger than molecular dynamics simulations. However this technique suffers from a number of drawbacks. Firstly, by restricting the particles movements to a regular lattice both isotropy and Galilean invariance are broken. Secondly, as more complex fluids are studied the number of possible states per lattice site increases dramatically and the model loses its efficiency and simplicity.

Dissipative particle dynamics (DPD) involves the introduction of lattice-gas automata-type time-stepping into a molecular dynamics simulation. This model provides a much faster scheme than molecular dynamics for the simulation of complex fluids, whilst maintaining isotropy and Galilean invariance. A set of particles in a continuum act as momentum carriers whose behaviour is governed by a set of stochastic differential equations\(^1\). Each particle interacts with its neighbours via three two-particle forces, all of which conserve mass and momentum. These forces comprise a conservative, dissipative and random

force. The conservative force conserves energy and is similar in nature to the force involved in a molecular dynamics simulation. The dissipative force acts to slow the particles and remove energy whilst the random force has the effect of adding energy to the system. The latter two forces act to drive the system towards an equilibrium state. DPD "particles" have a mesoscopic description, and do not represent individual molecules or atoms within a fluid.

Like lattice gas automata techniques, each time step consists of a collision phase and a propagation phase. The stochastic differential equations used to represent the particle motions are:

\[
\frac{dp_i}{dt} = \sum_{ij} \Omega_{ij} \hat{r}_{ij}, \quad \frac{dr_i}{dt} = p_i
\]

for the collision and propagation phases respectively. \( p_i \) represents the momenta, \( \hat{r}_{ij} \) the particle separation vector (where \( r_{ij} = r_i - r_j, r_{ij} = |r_{ij}|, \hat{r}_{ij} = r_{ij}/|r_{ij}| \)) and \( \Omega_{ij} \) represents the force (conservative, dissipative and random).

2.1 Lees-Edwards boundary conditions

The behaviour of these fluids under a shearing force is of great interest and in a desire to simulate the fluid behaviour under a shearing force, Lees-Edwards boundary conditions have been implemented\(^2\). Lees-Edwards boundary conditions attempt to simulate the actual sheared motion directly. The simulation box is surrounded by periodic images and the infinite periodic system is subjected to a uniform shear in the xy plane (i.e. a constant velocity gradient is set up in, for example, the y direction).

Particle in the simulation box are given an extra velocity \( \Delta V \) in addition to the velocity given to the particles in the original simulation. For a particle at position \( (X,Y,Z) \), \( \Delta V \) (say for the x direction) is obtained from:

\[
\Delta V = V_d \left( \frac{X}{L} - \frac{1}{2} \right)
\]

where \( L \) is the length of the simulation box and \( V_d \) is the specified speed in which the neighbouring cells are made to drift.

Figure 1 demonstrates the situation. Box A is the simulation box whilst images at positions \( (+/-L,0), (+/-2L,0) \) etc. are taken as stationary. Boxes in the layer above are moving at a certain speed in the positive direction whilst boxes in the layer below are moving at the same speed in the negative direction. Lees-Edwards boundary conditions reintroduce a particle into the cell with the velocity components altered by the shear and with the particle coordinates modified to account for the "slipping" of the cells with respect to each other.

3 Code structure and parallelisation

The DPD code has a similar construction to that of a molecular dynamics code, the principle difference occurring within the force calculation. The code has the following structure:

- generation of initial particle positions and velocities
- calculation of forces
- update of particle positions
- calculation of desired observables and return to force calculation or continue.

calculation of average desired observables

Although DPD provides a much faster scheme for the simulation of complex fluids than molecular dynamics, computational cost is a limiting factor, restricting the size and complexity of the simulations. This computational restriction made the DPD code an ideal subject for parallelisation. The code has been parallelised using a traditional spatial domain decomposition method using the message passing standard MPI. The simulation grid was divided into a grid of sub-boxes, each box being assigned to a specific processor with that processor responsible for the particles within its sub-box.

Before the force calculation is carried out, processors perform boundary swapping with their neighbours, passing information about particles which lie a cut-off distance \( r_c \) from the boundaries. Rather than passing only the particle positions, as would be the case in an M.D. simulation, the particle velocities also need to be exchanged, as this information is required in the force calculation.

In a normal molecular dynamics simulation the forces \( F_{ij} \) and \( F_{ji} \) are equal and opposite and can thus be calculated on different processors. Each processor can calculate the forces for the particles in its own sub-box without any extra communication. For DPD however, the random component of the force is different for each particle. This is overcome by assigning a unique integer label to each particle (\textit{particle-ident}) and only considering the interaction between two particles \( i \) and \( j \) if \textit{particle-ident}(i) is greater than \textit{particle-ident}(j). This identifier remains with the particle at all times and ensures that the interaction is only considered once on a single processor. Once the force calculation has been carried out an extra set of communications is required to pass the random forces back to the correct processor. Thus the force calculation involves the following steps:

- **Boundary-swapping** of particle positions, velocities, particle-ident etc
- **Force calculation**
- **Boundary swapping** of random force component
- **Random force component addition** to other forces

After the forces have been integrated and the new particle positions determined, a round of communication is carried out to exchange any particles which have moved out of the domain of each processor. The final set of communications present in the code occur during the calculation of the velocity fields where each processor sends information back to a master processor.
4 Lees-Edwards boundary conditions

In an effort to simulate the effect of shear on the system, and to measure shear viscosity, Lees-Edwards Boundary conditions have been implemented into the code. The theory of Lees-Edwards Boundary conditions has been described previously, in this section the code modifications will be described.

The first variation from the original DPD code occurs during the boundary swapping between neighbouring processors carried out before the force calculation. In the code, the velocity gradient exists in the y direction and the actual shear is applied in the x direction. Previously, boundary swapping in the y direction for one processor occurred between itself and the processors above and below. For the Lees-Edwards code, the shear may cause the particles to be swapped to other processors. Hence the first stage involves deciding which processors to communicate with. For the particles which lie a distance $r_c$ from the boundaries, two possible situations can arise. If the particles are located a distance $r_c$ from the boundaries of the global cell (not just the processors sub-cell) then shear is considered when carrying out boundary exchanges. Otherwise boundary exchanges occur as normal.

For the boundary exchanges involving shear, the global coordinate are determined and the strain added to this value. This new strain modified global coordinate is then utilised to determine the correct processors to send the information. The information is sent to the relevant processor (e.g. the (modified) velocities, coordinates and particle-ident), allowing the halo region created around the processors enough information to perform the force calculation for all the particles within its domain.

Once the force calculation has been carried out an extra set of communications is required to pass the random forces back to the correct processor. Once again shear needs to be considered and the reverse is carried out, removing the shear and sending the necessary particle information back to its original processor. Particles located within the corners of the processors can be passed back by two different routes, causing the forces not to sum to zero. Thus a transfer_flag is defined, with exchanges only occurring when the transfer_flag is .false.. The transfer_flag is only set to .true. for exchanges in the x direction.

After the forces have been integrated and the new particle positions determined a round of communication is carried out to exchange any particles which have moved out of the domain of one processor. This communication is similar to the communication carried out before the force calculation.

5 Dimers

Dimers are particle pairs separated by a fixed distance. The DPD code has been extended to cope with dimer systems, with an aim to simulate surfactants. Initially the code is similar to that of the original DPD code, the boundary exchange of particle positions and velocities occurring before the force calculation can be carried out. During this exchange however an extra type needs to be passed together with the positions and velocities. Each particle of a dimer has a partner value which corresponds to the other particle of the dimer. If the particle is a monomer this value is set to -1.

Initially the forces are calculated in an identical way to the original DPD code, and the random component exchanged once the calculation is complete. An extra routine is then required to ensure that all the particles have a copy of their corresponding partner information on the same processor. This involves all the particles located within one dimer length of the boundary being exchanged in a boundary swap.

Following this, a constraints routine is called which calculates the new positions of the dimer molecules. The positions are updated in an identical way to the verlet routine, however the positions are then moved to ensure that the dimer molecules are separated by the correct dimer length. The verlet routine is then called and the new positions of the monomer molecules generated. A round of communication is carried out (in an identical manner to the original DPD code) to exchange any particles which have moved out of the domain of one processor.
6 Hardware

The three EPCC machines utilised are summarised below, together with the optimisation flags utilised during the study. The majority of the work was carried out on the T3D, with the Hitachi and the T3E being used only for machine timing comparisons.

**Cray T3D** This comprises 512 processors, each clocked at 150 MHz and with 64Mb of memory. The compiler utilised was the Cray CF90 (Version 3.0.0.2) compiler. In the case of codes compiled on the T3D, the flags -O3, -Ounroll2 and the read-ahead buffer were all enabled. -O3 provides aggressive optimisation, where characteristics include a potentially larger compile size, longer compile time, global array optimisations and possible loop nest restructuring. -Ounroll2 provides the highest level of loop unrolling carried out by the compiler. rdaahead provides an improvement in the on-chip memory bandwidth.

**Cray T3E-900** Consists of 328 processors, each running at 450 MHz with a mix of 64, 128 and 256 Mbytes of memory. The compiler utilised was the Cray CF90 (Version 3.0.0.1) compiler. Optimisation flags -O3 and -Ounroll2 were also used on the T3E, however streams enabled (which is turned on by default) replaced read-ahead in-order to improve the memory bandwidth.

**Hitachi SR2201** The Hitachi SR2201 is a distributed memory parallel system with 8 processors, each of which features 256Mb of memory. The compiler utilised was the HI-UX/MMP optimising FORTRAN90 compiler, OFORT90. The flags -W0,'opt(o(s))' and -W0,'opt(o(s))' were utilised. The flag -W0,'opt(o(s))' comprises a range of optimisation options set up to enhance the execution speed. The second of the flags, -W0,'opt(o(s))', performs all the optimisations of -W0,'opt(o(s))' and pseudo vectorisation.

7 Timings for DPD

Timings runs have been carried out for a number of particles with various densities. Calculations have been carried out on processor combinations ranging from 4 to 128. Four data sets were considered, Set 1 involved 10000 particles with a density of 3, Set 2 100000 particles and a density of 10, Set 3 50000 particles and a density 10 and Set 4 involved 100000 particles and a density of 10. The results for each of the four codes are given in table 1. Timings have been carried out using Apprentice (the Cray profiler) and timings for each of the principal subroutines within the four codes reported.

The routine cycle is called from within the main program and carries out the MD cycle, 
cycle is called from within cycle and itself calls the force calculation routine DPD_forces. The routines export and export2 carry out the relevant boundary exchanges, and the routine verlet to carry out the integration of the equations of motion. gsync is a barrier call which causes the processors to synchronise and timing routine (which itself calls gsync). The Lees-Edwards code has the routines LE_low1 and LE_up1 present, which exchange particles between the simulation cell boundaries after the force calculation, and the routines LE_low2 and LE_up2, which exchange information between the simulation boundaries after the particle position updates. For the dimer code, extra routines include pass_dimer, d_stats, and d_export, which are all primarily communications routines (d_stats calls pass_dimer), and the constraints routine which causes the dimer bond lengths to be returned to a reasonable value after the integration of the equations of motion. The Lees-Edwards Dimer code contains two extra routines (LE_d1 and LE_d2) required to send dimer partners to the same processor which are undergoing shear.

An arbitrary choice of 32 processors was taken to study the effect of the different data sets on timings. Table 2 shows the principle subroutines for the four codes for 100 iterations with the four data sets whilst table 3 shows the added subroutines present for the Lees-Edwards, dimer and dimer Lees-Edwards codes. Parallel Speed-up, the ratio of the execution time of the parallel code on a single processor to the execution time of the same code on P processors, was used to measuring the performance of the codes on parallel machines. Figure 2 shows plots of parallel speed-up vs processor number for each of the data sets for each code normalised to the timings on four processors. The figure demonstrates that the scaling decreases with increasing number of processors and that the larger data sets scale more effectively than the smaller data sets. The more particles involved in the simulation, the better the scaling.
<table>
<thead>
<tr>
<th>Procs</th>
<th>Combs</th>
<th>Set 1</th>
<th>Set 2</th>
<th>Set 3</th>
<th>Set 4</th>
<th>Set 1</th>
<th>Set 2</th>
<th>Set 3</th>
<th>Set 4</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Original DPD</td>
<td>Lees-Edwards DPD</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>2x2x1</td>
<td>81.99</td>
<td>217.36</td>
<td>785.41</td>
<td>1582.79</td>
<td>81.22</td>
<td>199.72</td>
<td>714.50</td>
<td>1403.63</td>
</tr>
<tr>
<td>8</td>
<td>2x2x2</td>
<td>48.90</td>
<td>123.86</td>
<td>435.09</td>
<td>853.11</td>
<td>47.78</td>
<td>109.58</td>
<td>372.04</td>
<td>717.45</td>
</tr>
<tr>
<td>16</td>
<td>4x2x2</td>
<td>33.67</td>
<td>71.05</td>
<td>232.73</td>
<td>469.12</td>
<td>35.55</td>
<td>73.11</td>
<td>207.01</td>
<td>382.46</td>
</tr>
<tr>
<td>32</td>
<td>4x4x2</td>
<td>25.56</td>
<td>43.04</td>
<td>126.48</td>
<td>234.10</td>
<td>26.61</td>
<td>45.14</td>
<td>117.05</td>
<td>208.77</td>
</tr>
<tr>
<td>64</td>
<td>4x4x4</td>
<td>20.67</td>
<td>29.16</td>
<td>71.54</td>
<td>127.84</td>
<td>29.48</td>
<td>31.00</td>
<td>68.11</td>
<td>117.35</td>
</tr>
<tr>
<td>128</td>
<td>8x4x4</td>
<td>19.08</td>
<td>22.75</td>
<td>48.84</td>
<td>83.03</td>
<td>24.16</td>
<td>31.45</td>
<td>55.21</td>
<td>82.33</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Dimer DPD</td>
<td>Dimer Lees-Edwards DPD</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>2x2x1</td>
<td>93.23</td>
<td>215.97</td>
<td>761.93</td>
<td>1481.33</td>
<td>99.61</td>
<td>228.03</td>
<td>775.63</td>
<td>1508.43</td>
</tr>
<tr>
<td>8</td>
<td>2x2x2</td>
<td>56.58</td>
<td>120.73</td>
<td>396.34</td>
<td>758.90</td>
<td>58.73</td>
<td>123.54</td>
<td>406.31</td>
<td>774.93</td>
</tr>
<tr>
<td>16</td>
<td>4x2x2</td>
<td>46.83</td>
<td>76.34</td>
<td>225.43</td>
<td>416.66</td>
<td>41.28</td>
<td>76.50</td>
<td>227.08</td>
<td>421.78</td>
</tr>
<tr>
<td>32</td>
<td>4x4x2</td>
<td>30.28</td>
<td>47.04</td>
<td>125.22</td>
<td>223.69</td>
<td>32.39</td>
<td>49.08</td>
<td>131.71</td>
<td>234.01</td>
</tr>
<tr>
<td>64</td>
<td>4x4x4</td>
<td>38.03</td>
<td>40.70</td>
<td>76.68</td>
<td>129.51</td>
<td>30.20</td>
<td>40.37</td>
<td>80.76</td>
<td>133.44</td>
</tr>
<tr>
<td>128</td>
<td>8x4x4</td>
<td>28.20</td>
<td>30.89</td>
<td>56.12</td>
<td>85.06</td>
<td>32.54</td>
<td>39.38</td>
<td>68.37</td>
<td>106.26</td>
</tr>
</tbody>
</table>

Table 1: Timings (seconds) obtained for the four DPD codes with various processor combinations (combs) after 200 iterations.

<table>
<thead>
<tr>
<th>Set</th>
<th>main</th>
<th>cycle</th>
<th>plefor</th>
<th>DPD_forces</th>
<th>export2</th>
<th>gsync</th>
<th>timchk</th>
<th>verlet</th>
<th>export</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Original DPD</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Set 1</td>
<td>502.33</td>
<td>457.54</td>
<td>183.05</td>
<td>109.40</td>
<td>15.70</td>
<td>236.07</td>
<td>219.96</td>
<td>18.86</td>
<td>17.02</td>
</tr>
<tr>
<td>Set 2</td>
<td>869.15</td>
<td>817.49</td>
<td>530.87</td>
<td>339.04</td>
<td>22.60</td>
<td>234.83</td>
<td>221.87</td>
<td>19.36</td>
<td>23.31</td>
</tr>
<tr>
<td>Set 3</td>
<td>2218.69</td>
<td>2172.93</td>
<td>1826.84</td>
<td>1350.63</td>
<td>44.37</td>
<td>263.53</td>
<td>234.29</td>
<td>55.35</td>
<td>42.25</td>
</tr>
<tr>
<td>Set 4</td>
<td>3974.56</td>
<td>3924.32</td>
<td>3479.95</td>
<td>2689.47</td>
<td>61.12</td>
<td>260.68</td>
<td>236.62</td>
<td>98.42</td>
<td>78.48</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Lees-Edwards DPD</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Set 1</td>
<td>619.76</td>
<td>566.92</td>
<td>251.86</td>
<td>158.66</td>
<td>21.63</td>
<td>246.52</td>
<td>223.54</td>
<td>14.05</td>
<td>22.23</td>
</tr>
<tr>
<td>Set 2</td>
<td>841.56</td>
<td>788.03</td>
<td>499.63</td>
<td>357.84</td>
<td>32.61</td>
<td>237.59</td>
<td>221.19</td>
<td>13.61</td>
<td>28.78</td>
</tr>
<tr>
<td>Set 3</td>
<td>2126.82</td>
<td>2061.75</td>
<td>1622.63</td>
<td>1292.49</td>
<td>66.55</td>
<td>258.73</td>
<td>234.03</td>
<td>43.03</td>
<td>71.62</td>
</tr>
<tr>
<td>Set 4</td>
<td>3560.19</td>
<td>3488.87</td>
<td>3012.25</td>
<td>2518.35</td>
<td>95.72</td>
<td>290.04</td>
<td>243.23</td>
<td>79.13</td>
<td>113.80</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Dimer DPD</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Set 1</td>
<td>653.16</td>
<td>588.34</td>
<td>221.22</td>
<td>150.79</td>
<td>16.73</td>
<td>268.70</td>
<td>241.04</td>
<td>16.23</td>
<td>19.72</td>
</tr>
<tr>
<td>Set 2</td>
<td>933.11</td>
<td>867.94</td>
<td>482.66</td>
<td>362.58</td>
<td>24.63</td>
<td>279.01</td>
<td>246.39</td>
<td>16.23</td>
<td>26.92</td>
</tr>
<tr>
<td>Set 3</td>
<td>2134.51</td>
<td>2076.72</td>
<td>1522.22</td>
<td>1287.62</td>
<td>46.79</td>
<td>275.19</td>
<td>242.02</td>
<td>52.27</td>
<td>62.36</td>
</tr>
<tr>
<td>Set 4</td>
<td>3596.58</td>
<td>3534.08</td>
<td>2833.53</td>
<td>2497.36</td>
<td>64.65</td>
<td>283.42</td>
<td>238.49</td>
<td>95.82</td>
<td>94.95</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Lees-Edwards Dimer DPD</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Set 1</td>
<td>709.81</td>
<td>646.66</td>
<td>238.28</td>
<td>138.97</td>
<td>20.79</td>
<td>277.02</td>
<td>238.63</td>
<td>15.95</td>
<td>23.81</td>
</tr>
<tr>
<td>Set 2</td>
<td>974.00</td>
<td>914.44</td>
<td>511.49</td>
<td>350.31</td>
<td>30.58</td>
<td>266.21</td>
<td>228.69</td>
<td>15.38</td>
<td>31.90</td>
</tr>
<tr>
<td>Set 3</td>
<td>2364.34</td>
<td>2306.55</td>
<td>1608.26</td>
<td>1273.18</td>
<td>62.95</td>
<td>287.51</td>
<td>241.55</td>
<td>51.51</td>
<td>78.37</td>
</tr>
<tr>
<td>Set 4</td>
<td>4006.01</td>
<td>3944.78</td>
<td>2965.59</td>
<td>2465.57</td>
<td>90.47</td>
<td>285.55</td>
<td>242.06</td>
<td>96.79</td>
<td>123.50</td>
</tr>
</tbody>
</table>

Table 2: Apprentice timings (seconds) for the various subroutines for the four DPD codes with 32 processors and 100 iterations.

For all the codes, the majority of the time taken within the M.D. cycle is within plefor which carries out the boundary exchanges and force calculation. The force calculation (DPD_forces) dominates, with the majority of the remaining time involved in boundary exchanges and synchronisation. By increasing the number of particles, the number of particles per processor (and per link cell) increases and hence more force calculations are required. By increasing the density, the total volume of the cell decreases and hence decreases the size of the subcell assigned to each processor. The particles become more densely packed and hence the number of particles within each link cell increases, creating more force calculations. Although the total timings reduces with lower numbers of particles and lower density the time taken in
Table 3: Apprentice timings (seconds) for the added subroutines for the three extra DPD codes with 32 processors and 100 iterations.

synchronisation (i.e. barrier calls, routine gsync and routine timchk) remains similar across the board.

Figure 2: Parallel speedup over 4, 8, 16, 32, 64 and 128 processors for DPD (top left), for DPD Lees-Edwards (top right), for dimer DPD (bottom left) and for Lees-Edwards dimer DPD (bottom right). The dashed line represents linear (perfect) speedup with the solid lines illustrating the actual speedup. d represents particle density whilst no represents the number of particles.

The DPD Lees-Edwards code is slower and the scaling poorer than the Original DPD code. This is demonstrated clearly in figure 3 where the parallel speed-up for both codes and for all the data sets are plotted on the same graph. This is primarily due to the extra communications involved in the Lees-Edwards code, in a lot of cases the strain causes non-nearest neighbour communications to occur. At lower processor combinations however the timings are similar due to fewer communications occurring overall.

Comparison of the original code timings with the dimer code demonstrates that the introduction of dimers has slowed the code down. The implementation of dimers into the code has resulted in a number

146
of extra routines being added, the most time consuming being the pass_dimer routine which exchanges particles lying within a dimer length or less from the sub-cell boundaries to neighbouring processors (to ensure that every particle has its dimer partner present on the same processor). Figure 3 shows the parallel speed-up for both the original and dimer codes, clearly demonstrating that the original code scales more effectively than the dimer code.

![Graphs showing parallel speed-up for different processor counts for DPD and Lees-Edwards dimer codes.](image)

Figure 3: Parallel speedup over 4, 8, 16, 32, 64 and 128 processors for DPD and DPD Lees-Edwards (top), DPD and dimer DPD (middle left), for DPD and dimer Lees-Edwards DPD (middle right), for Lees-Edwards DPD and dimer Lees-Edwards DPD (bottom left) and for dimer DPD and dimer Lees-Edwards DPD (bottom right). The dashed line represents linear (perfect) speedup. The solid lines illustrate the actual speedup for the original DPD code and the dot-dashed lines represent the actual speedup for the dimer DPD code. Triangles represent data set 4, circles represent data set 3, diamonds data set 2 and squares data set 1.

It is unsurprising to observe that the dimer Lees-Edwards code is the slowest code and shows poorer scaling (see figure 3). Comparison with the original apprentice times demonstrates that Lees-Edwards routines and dimer routines are the principle cause of the poor scaling. Figure 3 also shows the scaling for the DPD Lees-Edwards dimer code in comparison to the DPD Lees-Edwards and DPD dimer codes respectively. Considering the apprentice timings it is clear that the poorer scaling results from a combination of the factors mentioned previously for the individual codes and from the additional dimer Lees-Edwards routines. These extra routines increase both the amount of communications occurring
and the amount of synchronisation. However in a more realistic simulation with one million particles, even the DPD Lees-Edwards dimer code scales remarkably well, demonstrating parallel efficiency of 80 percent going from 64 processors to 256 processors.

8 Optimisation effects for the four codes

8.1 Compiler optimisations and machine comparisons

Table 4 presents the timings for the four DPD code on a variety of machines and with various optimisation flags. Timings are for 4 processors. The machines and optimisations have been explained previously. Further timings have been carried out on the T3E on various processor combinations and the results are given in Table 5.

<table>
<thead>
<tr>
<th>Machine</th>
<th>Optimisation Level</th>
<th>Original</th>
<th>LE</th>
<th>Dimer</th>
<th>Dimer LE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hitachi</td>
<td>no opt</td>
<td>399.67</td>
<td>398.73</td>
<td>518.40</td>
<td>469.68</td>
</tr>
<tr>
<td>Hitachi</td>
<td>s opt</td>
<td>384.55</td>
<td>447.91</td>
<td>551.15</td>
<td>515.68</td>
</tr>
<tr>
<td>Hitachi</td>
<td>ss opt</td>
<td>352.78</td>
<td>362.49</td>
<td>472.22</td>
<td>447.77</td>
</tr>
<tr>
<td>T3D</td>
<td>no opt</td>
<td>785.41</td>
<td>714.50</td>
<td>761.93</td>
<td>775.63</td>
</tr>
<tr>
<td>T3D</td>
<td>O3</td>
<td>776.52</td>
<td>714.92</td>
<td>760.01</td>
<td>774.28</td>
</tr>
<tr>
<td>T3D</td>
<td>O3,unroll2</td>
<td>777.24</td>
<td>712.94</td>
<td>752.84</td>
<td>778.76</td>
</tr>
<tr>
<td>T3D</td>
<td>O3,unroll2,readahead=on</td>
<td>777.38</td>
<td>713.20</td>
<td>752.31</td>
<td>765.16</td>
</tr>
<tr>
<td>T3E</td>
<td>no opt</td>
<td>217.80</td>
<td>210.35</td>
<td>229.12</td>
<td>262.00</td>
</tr>
<tr>
<td>T3E</td>
<td>O3</td>
<td>216.90</td>
<td>210.64</td>
<td>226.29</td>
<td>233.51</td>
</tr>
<tr>
<td>T3E</td>
<td>O3,unroll2</td>
<td>215.28</td>
<td>210.28</td>
<td>226.31</td>
<td>228.73</td>
</tr>
</tbody>
</table>

Table 4: Timings (seconds) obtained for the four DPD codes with four processors after 200 iterations. Data set 3. LE - Lees-Edwards

<table>
<thead>
<tr>
<th>No of Procs</th>
<th>Combination</th>
<th>Original</th>
<th>LE</th>
<th>Dimer</th>
<th>Dimer LE</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>2x2x1</td>
<td>217.80</td>
<td>210.35</td>
<td>229.12</td>
<td>262.00</td>
</tr>
<tr>
<td>8</td>
<td>2x2x2</td>
<td>114.79</td>
<td>106.14</td>
<td>116.37</td>
<td>118.93</td>
</tr>
<tr>
<td>16</td>
<td>4x2x2</td>
<td>60.33</td>
<td>57.64</td>
<td>64.19</td>
<td>65.40</td>
</tr>
<tr>
<td>32</td>
<td>4x4x2</td>
<td>33.36</td>
<td>32.68</td>
<td>37.50</td>
<td>39.62</td>
</tr>
<tr>
<td>64</td>
<td>4x4x4</td>
<td>18.50</td>
<td>20.82</td>
<td>23.43</td>
<td>25.22</td>
</tr>
</tbody>
</table>

Table 5: Timings (seconds) obtained for the four DPD codes with various numbers of processors with data set 3 on the T3E.

Table 4 allows a direct comparison of the performance of the 3 machines to be made for the four DPD codes. The T3D performs poorly with and without optimisation in comparison to the Hitachi and T3E, the T3E giving the fastest timings. A significant reduction in the overall runtime on the Hitachi has been observed using the various optimisation flags. The T3D and T3E however demonstrate only a minimal reduction in runtime with optimisation. The effectiveness of the flag -WO,'opt(o(ss))' (on the Hitachi) is clearly illustrated with a considerable reduction in the runtime observed. This flag enables pseudo vectorisation, a pipelining technique which allows data to be loaded directly from main memory into floating point registers without blocking the execution of subsequent instructions. Also of interest is the effect of the flag WO,'opt(o(s))' which has in three out of four cases reduced the speed of the code.

Table 5 shows timings for the T3E across a number of processors, comparison of these values with the corresponding T3D values demonstrates that the T3E is considerably faster than the T3D across a number of processors and that the scaling is slightly better to that of the T3D.
8.2 Performance optimisation

The performance of the original DPD code was considered and modifications made to enhance the speed of the code. The apprentice analysis highlighted three areas which dominate the code performance, the force calculation (routine \texttt{DPD\_forces}), the time spent carrying out synchronisation and the time spent carrying out communications. Of these the force calculation was the most considerable, the performance of which is limited by the single processor performance. The most expensive part of this routine is the time spent within a set of nested do loops. The link cell method is used and the loops are carried out over link cells. Initially the partner link cell is determined, and the link cell arrays used to sum between the relevant particles. Following this, the separation between the x, y and z coordinates are determined and used to calculate the new forces and hence update the particle forces and velocities. The head-of-chain array (created previously) contains the identification number of one of the molecules within a link cell. This number is used to address the element of a linked-list array which contains the number of the next molecule within the cell. The list array element for that molecule is the index of the next molecule in the cell and so on. Eventually an element of this second array is reached which is zero, indicating there are no more molecules in the linked cell and another linked cell can be considered.

Apprentice revealed that the largest amount of time is spent calculating the separation distances and velocity differences between the particles and calculating the forces.

Had the force calculation been more simplistic, calculating the separation distances and velocity differences for all the particles, the optimisation techniques loop unrolling and/or loop splitting may have been practical options. However as the loop is over link cells and the elements of an array are used to index the next particle, these becomes impractical. Performance enhancement was however obtained by modifying the force calculation to allow for some form of pipelining to occur where previously this was prevented. Although not one of the major time consuming pieces of the code, the addition and/or subtraction of the new force from the force on the particles still utilises a certain amount of the code's time. The T3D can execute the true clause of an if statement more efficiently than the false (else) clause. The logic of certain pieces of code was reordered to account for this, giving some small performance increase. For an arbitrary test example of 8000 particles and 50 iterations (density = 10) on 8 processors a total time of 141.86s was obtained, where the time before optimisation was 166.58s.

The majority of the synchronisation routines are required for correctness, however possible savings may be made by removing a number of the timing routines which incur additional synchronisation.

The code carries out boundary exchanges between the processors using MPI. On the T3D and T3E the SHMEM library can be used instead. Although this is not portable, SHMEM offers better performance than MPI on the Cray systems, particularly in terms of latency. By replacing the \texttt{MPI\_SSEND} routines with \texttt{SHMEM\_PUTs} the boundary exchanges can be carried out in a more efficient manner. All other MPI communications were unaltered. Cache coherency is not guaranteed on the T3D and hence calls to explicit cache management routines were required. Taking an arbitrary example of 10000 particles and density 10 for 200 iterations on 8 processors, the time reduced from 123.86 seconds to 106.27 seconds.

9 Summary

A dissipative particle dynamics (DPD) code has been developed to study the hydrodynamic and rheological properties of complex fluids. The code has been extended to consider dimers, with a view to studying surfactants, and to consider the effect of shear using Lees-Edwards boundary conditions. As a final test, simulations were performed on one million particles, with both dimers and shear effects and a parallel efficiency of 80 percent was achieved going from 64 to 256 processors on the T3D.
Parallel modelling of the exchange across the stratospheric subtropical barrier in a global 3-d model.

S. Leder, Institut für Meteorologie,
Carl-Heinrich-Becker-Weg 6–10, D-12165 Berlin, Germany.
A. Beck-Ratzka, Regionales Rechenzentrum für Niedersachsen,
Schloßwenderstraße 5, D-30159 Hannover.

ABSTRACT

The stratospheric subtropical barrier is investigated with the help of a global 3-d gridpoint model on the MPP system T3E. We have calculated the transport of isentropic Potential Vorticity (PV) during a 1000 day perpetual January simulation with a steady state forcing at the lower model boundary near the tropopause by observed values of planetary wave amplitudes. Large flux from the tropics into the subtropics occurs in combination with an enhanced activity of planetary waves. To represent the subtropical barrier a horizontal resolution of 1.4\textdegree x1.4\textdegree is necessary. This can only be achieved in an acceptable manner by running the model on a MPP system.

1. INTRODUCTION

The knowledge of mass exchange between different parts of the atmosphere is essential to understand the transport and the distribution of long-lived atmospheric trace gases, e. g. ozone or atmospheric aerosols. It is well accepted that the horizontal mass exchange in the stratosphere of the winter hemisphere is restricted between the inner part of the polar vortex and midlatitudes (e. g. Chen 1994;). Beside the polar vortex edge as a transport barrier, another barrier is located in the subtropics. The most direct evidence for weak transport out of the tropics comes from the longevity of tropical maxima of aerosols produced by tropical volcanic eruptions. This effect has been documented nicely after the eruption of Mount Pinatubo (15\textdegree N, 120\textdegree E) on June 14–15, 1991 (Trepte et al., 1993). One-level model studies (e. g. Polvani et al. 1995) also show the existence of the subtropical barrier. The representation of the subtropical barrier and other transport barriers needs a relatively high horizontal resolution. The presented study investigates the behaviour of the subtropical barrier at a disturbed winter stratosphere with a high–resolved global 3-d mechanistic gridpoint model using a horizontal resolution of 1.4\textdegree x1.4\textdegree. This leads to large requests of primary memory, which are not available on scalar computers. Therefore our model has to be ported to MPP systems.

150
2. RESULTS

A previous model study of (Beck, 1998) has investigated the exchange of tracers across the subtropical barrier at single events. To get more informations about the dynamic conditions which will lead to flux from the tropics into the extratropics of the winter hemisphere, it is not sufficient to look at single events. Therefore long term model runs were performed. In order to be closed to the observations we have started a perpetual January experiment with a mean climatological zonal mean zonal wind and temperature field for January. At long term runs, the lower model boundary is located at the top of the troposphere, while the upper boundary lies near 120 km, above the height region with largest wind speeds. In the vertical direction the model is equally spaced on 80 levels, which leads to a vertical gridpoint distance of 1.366 km. A realistic “disturbed” winter stratosphere, as it usually occurs at Northern hemisphere winter, is obtained by forcing the model at the lower boundary at 200 hPa with observed geopotential height values of the planetary waves 1 to 4. The forcing pushes the model away from the initial zonal mean conditions, while radiative processes, represented in the model by the Newtonian Cooling approximation (Dickinson, 1973), relaxes the model back to the zonal radiative equilibrium field.

![Figure 1: Zonal mean zonal wind in the ENSO warm event run (left) and the ENSO cold event run (right) in 30 Km.](image)

The dynamic of the stratosphere is primarily driven by upward propagating tropospheric planetary waves, so changes in the forcing field at the lower boundary will lead to different dynamic evolutions in the stratosphere. Therefore it is possible to compare the behaviour
of stratospheric dynamics under different climatological conditions, by taking the observed geopotential height fields of those different climatological conditions at 200 hPa as forcing fields at the lower boundary. We have performed two 1000 day model simulations with an 'ENSO (EL Niño/Southern Oscillation) cold' and an 'ENSO warm' forcing (Labitzke et al. 1997) and compare the exchange between the tropics and midlatitudes of the winter hemisphere in the different model runs. In the years of warm events in the Southern Oscillations the polar vortex is weak and disturbed, during cold events the polar vortex is deep and cold. The overall evolution of the model stratosphere is in accord with the observations of ENSO cold and ENSO warm years.

Due to the concurrent effects of the steady state forcing and the radiative forcing by Newtonian Cooling, the model shows large vacillations, irregular in time and intensity. Fig. 1 shows the zonal mean zonal winds at 10 hPa in dependence of latitude and time, for ENSO cold and ENSO warm events, clearly indicating the irregularity of the oscillations. In periods of only weak westerlies, the stratospheric winter polar vortex is usually displaced from the Pole because of the enhanced planetary wave activity in these periods. The westerlies remain more stable during ENSO cold events in good agreement with observations.

The model simulates quite well the difference between the ENSO warm and ENSO cold climatological state. Fig. 2 shows the difference of the climatological mean geopotential heights for January, on the left for the data of the Free University of Berlin (FUB) (Pawson et al. 1993), on the right for the model simulations. There is an overall good agreement between the observations and the simulation. The model underestimates the wave 2 component, what leads to the absence of the negative difference over southwestern Europe.

Figure 2: Geopotential height difference (gpdm) between ENSO warm event and ENSO cold event at 10hPa in the FUB data (left) and in the model (right).
Figure 3: North polar stereographic projections of the Potential Vorticity (PV) at 850 K, from day 634 to day 644 in the warm event run.
As we know from several real data studies (e.g. Chen, et al. 1994; Waugh 1996), the exchange across the subtropical barrier is large during periods with an enhanced activity of planetary waves. If the model is able to capture these exchange processes in a realistic manner, it must show exchange across the subtropical barrier when the wave activity is large. At first, we like to look at a time series of PV on the isentropic level at 850 K. Fig. 3 shows the PV from day 634 to 644 with a two day interval. Under the adiabatic approximation, PV is a conservative variable and therefore well suited to investigate quasi-horizontal transport processes on isentropic surfaces, because the quasi-horizontal transport on these levels is of adiabatic nature. In Fig. 3, tropical PV (recognizable at low PV values) is grey shaded. Low PV values coming from North Africa are located in a narrow tongue. This so-called streamer is moving northeastswards, transporting the tropical air in middle and high latitudes.

![Image](image.png)

Figure 4: Amplitude of the planetary wave 1 and the zonal PV minima at 850 K in the UKMO data set (left) and in the model run (right).

But which dynamic process is really responsible for the exchange across the subtropical barrier? A first hint is given by Fig. 4, where the amplitude of the planetary wave 1 and the tropical PV are plotted for a streamer event. The left part shows the March 1993 in the UKMO data set, the right part the results of our model run. Both figures demonstrate the developing of a streamer in low latitudes when the planetary wave 1 has a maximum. The low PV values are reaching high latitudes after a few days. The tongue in the model simulation is smaller then in the observations. The more narrow structure of the model streamer is not unrealistic. The observations suffer from the relatively low resolution. Therefore the subtropical barrier was not discovered until the relatively high resolved satellite observations of the Mount Pinatubo eruption aerosol cloud (Trepte et al. 1993). Before these observations, the existence of a subtropical barrier was unknown as well at the observations as at the global simulation models. New satellite observations of streamers (Offermann, 1996) indicate a more narrow streamer structure, as shown in our model experiment.

Finally we like to present some statistics about the link between enhanced planetary wave
Figure 5: Timeseries of the zonal PV minima (solid) and the amplitude of the planetary height wave 1 (dotted) for the warm event case at 475 K (left) and 850 K (right).

activity and the occurrence of streamers with tropical air in the midlatitude of the hemisphere. We estimate the lowest PV value along a latitude circle (LPV), keeping in mind that tropical air has low PV values near or below zero. Fig. 5 shows time series of LPV at 60° N and the amplitude of the planetary wave 1 at 40° N for the isentropic levels 475 K (located at the lowermost stratosphere) and 850 K (middle stratosphere). At both levels, LPV is small and even sometimes below zero during periods with large wave amplitudes. Estimating the correlation yields a value of -0.82 at 475 K, and -0.62 at 850 K. So at least at 475 K, the lowermost stratosphere, we have a well pronounced correlation between enhanced planetary wave activity and the occurrence of streamers.

3. SOFTWARE SOLUTION

The simulation of the model described in the previous section is done with the STRATO code. STRATO was developed at the “Institut für Meteorologie” of the FU Berlin and is based on the primitive equations in flux form. The initial version of STRATO was running on Cray vector computers. After the installation of a Cray T3E parallel computer at the “Konrad Zuse Zentrum fuer Informationstechnik Berlin (ZIB)”, a parallel version of STRATO was created. The computation speed and the available memory of many Processing elements (PE’s) enables the users to run simulations with the required high resolution in a reasonable amount of time. A domain decomposition has been used for the parallelization of STRATO. The latitudes have been distributed across the processing elements. The communication is done with Crays explicit shared memory message passing software (Shmem Technical Note, publication sn–2516, CRAY Research Inc., 1994). Inner boundaries have been introduced. Every PE stores the first and last latitude of the respective neighbor PE as boundary latitude.
<table>
<thead>
<tr>
<th>model</th>
<th>PE's</th>
<th>time</th>
<th>Mflops</th>
<th>Mflops/PE</th>
</tr>
</thead>
<tbody>
<tr>
<td>80<em>128</em>64</td>
<td>8</td>
<td>66.5</td>
<td>223</td>
<td>27.9</td>
</tr>
<tr>
<td></td>
<td>16</td>
<td>32.9</td>
<td>451</td>
<td>28.1</td>
</tr>
<tr>
<td></td>
<td>32</td>
<td>18.6</td>
<td>798</td>
<td>24.9</td>
</tr>
<tr>
<td>80<em>256</em>128</td>
<td>32</td>
<td>65.2</td>
<td>804</td>
<td>25.1</td>
</tr>
<tr>
<td></td>
<td>64</td>
<td>35.4</td>
<td>1480</td>
<td>23.1</td>
</tr>
</tbody>
</table>

Table 1:

After modification of values of 3-d arrays a boundary exchange has to take place. The main communication is done during boundary exchange. The only other significant communication is done in the transport routine. After interpolation, arrays are accessed randomly. Some elements are located on other PE’s and have to be fetched via communication software. The resolution described in chapter 2 result in a grid of 80*256*128 (high, longitude, latitude). The program requires at least two latitudes on 64 PE’s, that means it can run on up to 64 PE’s due to the required primary memory of approximately 2.2 GB. STRATO needs at least 32 PE’s to run this model.

Showing the performance and scalability of STRATO, Table 1 lists the computation time and Megaflop rate of 32 time steps of two models with a resolution of 80*128*64 and 80*256*128, respectively. The current version of STRATO runs in parallel. Due to the restriction of 2 latitudes per PE the program is not yet capable to run massively parallel. There are plans to do a two-dimensional parallelization across the latitudes and longitudes, which will allow to run on a couple of hundred PE’s. The version will allow the users to do long term simulations as well as higher resolutions.

REFERENCES


Parallel Processing in Conjunction with Structure Analysis of Biological Macromolecules by Electron Microscopy

Reiner Hegerl and Jochen Walz
Max-Planck-Institut für Biochemie
D-82152 Martinsried, Germany

Introduction

Electron microscopy is a powerful method to investigate the structural organisation of biological macromolecules or large molecular complexes with the final aim to understand their function. From a broad palette of different microscopic techniques, electron crystallography has achieved atomic resolution and, therefore, may become a competitor to X-ray crystallography, the classical method in the field of structure analysis. In contrast to X-ray diffraction, electron microscopy can also be used to image and analyse individual molecules not accessible to crystallization. The main problems with this approach when applied to biological objects arise from the low contrast and from the sensitivity against electron irradiation. In order to minimize radiation damage, electron micrographs must be recorded with a very low electron dose and, consequently, suffer from extremely high noise levels. The basic idea to overcome this problem is to collect data from a large number of individual molecules and to extract their common features by adequate averaging procedures. Obviously, extensive computing is an essential part of this approach.

Principles of electron tomography and image processing

For physical reasons, any image recorded by a transmission electron microscope represents a parallel projection of a volumetric entity which is related to the electric potential distribution of the object or, loosely speaking, to its three-dimensional structure. Similar to computer tomography in medical imaging, electron tomography is a nondestructive technique which can be used to create different projection views and to reconstruct the volumetric data from these projection images. The typical experimental setup for the investigation of macromolecules is as follows: The molecules are embedded in a thin film (e.g. 20 nm) of amorphous ice spanning the holes of a thin carbon film which is deposited on a copper grid. Using a special specimen holder, the grid can be rotated in the microscope around an axis perpendicular to the
electron beam. The experiment results in a so-called tilt series which consists of a set of projection images taken at different rotational positions of the specimen holder. Fig. 1 shows one image of such a tilt series of ice-embedded thermosomes (enzymes, involved in protein folding), mainly in one preferred orientation with respect to the specimen support. In Martinsried, most operations needed to record a tilt series (focus setting, keeping the object area within the field of view, image recording) are performed automatically in order to reduce the electron dose to a minimum (Koster et al., 1996). For the data recording,

Fig. 1: Zero degree projection of a tilt series. The image shows a thin film of amorphous ice (large bright area) spanning the hole of a carbon film (dark area at left hand). Thermosome molecules embedded in the ice are almost invisible due to the low electron dose. The small dark spots are gold clusters which are used as markers for the alignment.
a CCD camera is used. At present, a typical tilt series consists of 20-50 images with a size of 1024x1024 pixels and of 2 bytes per pixel. About 5 tilt series may be taken per day with one microscope.

The first step in a long chain of image processing procedures is to relate all images of a tilt series to a common coordinate system. Mainly caused by the tilt operation but also by a possible drift of the specimen holder, the object appears to be shifted when comparing any two projection images taken at different tilt angles. The shift correction is based on correlation methods and may be supported by a special specimen preparation technique, e.g., the inclusion of tiny gold clusters in the ice film to be used as marker points. In the next step, a volume is reconstructed from the well aligned projection images using the filtered back projection algorithm. A sampling grid with 1024x1024x128 voxels, 4 bytes per voxel, is usually required to represent this volume with the full resolution of the CCD-camera. Depending on the concentration during specimen preparation, 20-100 molecules may be included in the volume. In the third step, these molecules have to be identified and extracted from the large volume. At present, this operation is done manually. Combining the data of several tilt series taken at different areas of the specimen, up to 500 cubes with $64^3$ voxels may be obtained, each of them representing the density distribution of an individual molecule.

Due to the high noise level mentioned above, these data cannot be interpreted. However, assuming that all molecules have the same structure and also assuming additive noise, averaging should reveal the "true" structure hidden in all reconstructions of individual molecules. In general, the molecules are embedded in the ice film with arbitrary orientation. Therefore, averaging requires that all molecule images are aligned with respect to a reference image. Six parameters have to be determined for each molecule, 3 shift parameters and 3 rotation angles (Eulerian angles) to correct for translation and orientation, respectively. While the shifts can be derived from three-dimensional cross-correlation functions, the orientational parameteres can only be determined by searching the angular space defined by the Eulerian angles for maximum correlation. Once the alignment is done, an average image is calculated from the molecules just aligned and used as reference in a new alignment step. Repeated application of alignment and reference update aims to remove a possible bias of the first reference. For more details, the reader is referred to Walz et al. (1997). It may be a problem to obtain a suitable first reference. In principle one could select one of the molecules obtained by the tomographic reconstruction but, in most cases, the data are too noisy. In practice, there is information from other experiments which can be used to construct a first reference synthetically.
Parallel processing and software development

The three-dimensional alignment of molecule images with respect to a reference image is an ideal task for parallel computing because the same job has to be done independently for each molecule. The strategy is obvious: Each processor element (PE) performs a complete alignment of one molecule. After completion, a new molecule not yet aligned is submitted to the PE as long as there are unaligned molecules. Otherwise, the PE waits for completion of all other PEs. After calculating a new reference image a new cycle can be started.

Since many years, a software package named "EM" is used at the MPI für Biochemie (Hegerl, 1996) to perform all image processing tasks necessary for quantitative electron microscopy. Acting as a command interpreter, it can be used interactively or in batch mode. The set of commands comprises operations generally needed in image processing, e.g., input, output, storage and graphic display of data, image manipulations and transformations like Fourier transformation, but also special operations like the correction for the contrast transfer function involved in the imaging process of the electron microscope. Since there are also commands to control the flow of operations, complex procedures can be developed like those for tomographic reconstruction and subsequent averaging. To perform these procedures in parallel on the T3E, only a few modifications of the source code had to be made. At first, the set of control commands was extended to identify and synchronize the PEs, using SHMEM_MY_PE, SHMEM_N_PES, and SHMEM_BARRIER. In the EM language, image data are identified by a name and a number. Assuming a consecutively numbered set of molecule images, all with the same name, each image is assigned to a processor identifier via its number. Since there are more images than PEs, the image numbers are incremented by the total number of processors. Once the alignment is performed for all molecules, the processors are synchronized to calculate a new reference image. Additional software modifications were made to speed up the whole process. The broadcast function SHMEM_BROADCAST is used to supply the reference data read by one PE to all PEs available, and the global sum function (SHMEM_REAL8_SUM_TO_ALL) provides useful facilities for the reference update. Special care was taken for the FFT which is the operation most frequently used in the context of alignment and averaging. Replacing the standard algorithm of EM by library functions available on the T3E (TFFT3D, SCFFT3D, CSFFT3D) has reduced the CPU-time needed for this operation by a factor of 2.5. It may be interesting to mention that the adaption of the partially old FORTRAN code of EM to the 8 byte architecture of the T3E took more time than the modifications necessary for parallel processing.
Example and first experiences

The thermosome is a member of the group of so-called chaperones, large protein complexes whose function is to assist the folding of other proteins. The name "thermosome" points to the fact that enhanced occurrence of this macromolecule is observed in bacteria after exposition to heat shock conditions. Present investigations at the MPI für Biochemie aim to support the hypothesis that two different structural conformations ("open" and "closed") are connected with the function of the complex (Nitsch et al., 1998). Three tilt series of ice-embedded thermosomes were recorded automatically in a computer-controlled electron microscope and evaluated as described above. In total, 307 molecules could be extracted from the large volumes obtained by tomographic reconstruction of the 3 tilt series. The gallery of Fig. 2 represents horizontal sections through the density map of one of these molecules

![Fig. 2: Density map of an individual thermosome molecule obtained by tomographic reconstruction under low dose conditions. The map is represented by slices parallel to the specimen plane. The distance between two adjacent slices corresponds to the doubled sampling distance in x and y of 0.4 nm. The inset (right hand down) shows the sum (projection) over all slices.](image)

sampled by $64^3$ voxels. Due to the high noise level, it is difficult to recognize the molecule. Only the sum of all sections - inset in Fig. 2 - gives a rough idea of the particle shape. The molecules were subjected to the alignment and averaging procedure now available on the T3E. The density map of the final average is shown in Fig 3 again by a gallery of sections and, in Fig. 4, as an isosurface representation (using AVS). The threshold for the surface was chosen such that the enclosed volume corresponds to the molecular weight of 930 kDa thereby assuming an average density of the protein of 1.3 g/cm$^3$. Averaging revealed the thermosome as a barrel-shaped particle with a diameter of 15 nm and a height of 18 nm. A resolution of 2.5 nm may be estimated which is enough to assign the structural features to the "open" conformation.
In the course of averaging, the data were subjected to symmetry operations as shown in Fig. 3. From the biochemical analysis it is known that this type of thermosome is composed of 16 identical subunits. Additionally, a symmetry of P82 could be derived from two-dimensional image analysis. Generally in the case of symmetry, the analysis is facilitated by the fact that the orientational search can be restricted to the angular space corresponding to the asymmetric subunit of the molecular complex.

The calculations gave a first idea of processing time. Using 128 PEs, 5 cycles of alignment and reference updating could be carried out within a run-time of about 4 hours. The CPU-time per PE is less but does not fall below 70% of the run-time. Before EM has been implemented on the T3E, about 4 weeks were necessary to perform a comparable processing task on a current workstation. Even when considering the queuing time on the T3E, there is a considerable gain in time between data recording and the visualization of an averaged image.

Future developments

Improving the resolution is a general aim in the development of methods for structure analysis. Regarding electron tomography of macromolecules, the number of molecules is critical in this context mainly for two reasons: First, the significance of the final average image increases with the square root of this number. Second, there is a variety of structural features in any set of molecules though one "true" structure must be assumed for averaging. The larger the variety the lower is the resolution.
The number of molecules can be increased either by recording more tilt series or by enlarging the imaged specimen area. With the end of this year, one of the microscopes in Martinsried will be equipped with a CCD-camera consisting of 2048x2048 pixels. Compared to the present installation, the image area increases by a factor of 4 when using the same optical magnification. Assuming constant distribution of molecules their number will increase accordingly. With these experimental uptakes also more computing power is required. A problem may arise when the calculations have to be done with a finer sampling grid to achieve better resolution. At present, the density maps of molecules are sampled with $64^3$ voxels. With a sampling of $128^3$ voxels, however, the
actual alignment procedures have to be modified due to memory limitations of the PEs.

Concerning the variety of molecules, methods are known to reduce the structural variance by subdividing a set of images into subsets of higher homogeneity. This means that images have to be sorted according to their similarity by classification techniques such as feature extraction based on singular value decomposition or - more fashionable - neural networks. These methods are in use for two-dimensional image analysis and can easily be extended to three-dimensional applications (Walz et al., 1997). It is an interesting question how parallel processing can be used to accelerate these methods.

Acknowledgements
We would like to thank Dr. H. Lederer and Mr. J. Pichlmeier for their help in programming.

References


Early experiences with OpenMP on the Origin 2000

Ragnhild Blikberg*  Tor Sørevik†
Parallab, University of Bergen,
NORWAY  Parallab, University of Bergen,
NORWAY

September 7, 1998

Abstract

OpenMP has been marketed as THE emerging standard for shared memory parallelism (SMP). The first compiler for OpenMP is now available on the Cray Origin 2000. In this paper we report on some early experiences with this compiler on a (quasi-)application code, an implementation of the NAS, BT benchmark.

OpenMP includes, of course, the traditional do-loop parallelization. For programmers familiar with Power Fortran this is simply a question of adapting to a slight modification of syntax. Not surprisingly the performance equals that of Power Fortran. This part is easy and straightforward. The interesting thing about OpenMP is that it goes beyond the simple do-loop parallelization and opens possibilities for a more course grain parallelism by splitting the programming in different independent tasks (Parallel sections) and/or creating parallel regions. Combining this with the more fine grained do-loop parallelism keeps the promise of higher scalability than one usually find in SMP-programming. Allowing for nested parallelism, OpenMP gives us this opportunity. With appearance of SMP-system scaling beyond 128 processors this is very important new features.

In this paper we focus on those features of OpenMP which extend the model for parallel computing already available with Power Fortran. To get a feel for how it works on real application we have applied the techniques to a BT-code for the NAS parallel benchmark and report results on short coming of this first release of the compiler as well as speed-up and scalability results. Comparison with MPI will also be provided.

Keywords: OpenMP, SMP-parallelism, parallel benchmark

1 Introduction

One of the major obstacles to progress of parallel computing for large scale scientific problems has been the lack of standards. Users have for good reasons been reluctant to take the burden of parallelizing their codes when they can't be sure that the investment will pay off when moving to a different parallel computer.

This problem has been overcome for the message passing programming model with the introduction of MPI [GL94] and for data parallel programming with HPF [KLS+94]. For directive based Shared Memory Programming (SMP-programming) no such standard exists. Efforts have been made, most noticeable the ANSI X3H5 standardization effort[Le94]. But their proposal was never formally approved and no successful implementation ever reached the market. It is therefore very welcome in the user community that a vendor driven group now tries to agree on an industry standard for SMP-programming. The new standard is called OpenMP.

*http://www.parallab.uib.no/~ragnhild
†http://www.parallab.uib.no/~tors
The OpenMP group released its Fortran standard in October 1997 [whp97], [omp97], and the first commercial available implementation of this was available on SGI's Origin 2000 in the spring of 1998. In this report we will describe some of the features of OpenMP and some early experiences with it on our Origin 2000 system. In particular we focus on features which are new compared with SGI's old parallelizing Power Fortran.

This paper is organized as follows. First we give a short description of some of the main features of OpenMP and comments on some of the features we miss most. Then we give a brief description of our target test application, the NAS-BT parallel benchmark. In section 4 we report on our experience using OpenMP on this application. Speed-up is measured and compared with MPI.

2 The OpenMP standard

The OpenMP API (Application Programming Interface) is a set of directives to instruct the compiler when to fork multiple threads and where to join these threads again. An associated runtime library and a set of environmental variables are also defined. Here we only give a very brief overview of the most important directives and their scope.

The key directive is _C$OMP PARALLEL_ which defines a parallel region. A parallel region is a block of code which is to be executed by multiple threads. It ends by _C$OMP END PARALLEL_. Different clauses might be added to the _PARALLEL_-directive. In particular one might want to define data as PRIVATE or SHARED. It's also possible to specify the reduction of SHARED data.

The most obvious way of specifying work to different threads is distributing independent iterations of do-loops to the different threads. This can be specified by the _C$OMP DO_ and _C$OMP END DO_ directives, embedded in _C$OMP PARALLEL_ and _C$OMP END PARALLEL_. There is also the shorthand notation of _C$OMP PARALLEL DO_ which is handy in cases where the parallel region contains only one do-loop.

Since the dawn of parallelizing compilers, all directive based parallelizing compilers have had the ability of parallelizing do-loops, and it is still, by a large margin, the most used feature. For many applications, however, parallelizing the outer do-loop is not sufficient. When the number of iterations in the outer loop is to small, it only allows for limited parallelism. In other cases parallelizing do-loops only allow for relatively fine grain parallelism, although the problem itself might posses a coarse grain parallelism on a higher level. To assist with the first problem, OpenMP allows for nested parallelism. This can be applied to a set of nested do-loops.

A way of specifying independent tasks, not specified by different iteration of a do-loop, is provided by the _C$OMP SECTION_ directive, as in:

```c
C$OMP SECTIONS [clause[,...]clause],...
C$OMP SECTION
  a block of code
C$OMP SECTION
  a block of code
```

............... 
C$OMP END SECTIONS

This will cause the different blocks of the code to be executed in parallel. In cases where a block of code contains parallel tasks, nested parallelism might be used in this case too. We would claim that to allow for flexibility in the choice of the number of processors to use, it is necessary to use nested parallelism in combination with _C$OMP SECTION_, as the _C$OMP SECTION_ specifies a fixed number of parallel tasks.

To ensure control in multi threaded programming, means of synchronization and exclusive control for one thread, are necessary. A handful of handy directives like _C$OMP BARRIER_, _C$OMP MASTER_, _C$OMP CRITICAL_ and _C$OMP ATOMIC_ are available.

The runtime libraries provide functions for, at runtime, finding the total number of threads running and the specific number of one particular thread. You are also allowed to specify whether you want dynamic or static adjustment of thread numbers and to set locks enabling synchronization of threads.
To a large degree OpenMP is only standardizing syntax for features available on most parallelizing compilers, but some features are not universal available on all parallelizing compilers. The two most significant extensions are the **C$\text{OMP SECTIONS}$** and the possibility of nested parallelism.

### 2.1 Missing features

The most notable feature we are surprised not to find, is the lack of support for parallel directives applicable to Fortran 90/95 array syntax. Many of the most obvious parallel do-loops translates into the neat array syntax when programmed in F90-style. It seems awkward that an API for Fortran, designed in 1997, does not address the current Fortran-standard, but relates to non-standard Fortran!

We did also miss the opportunity to do reduction on arrays. For reasons unknown to us, the standard explicitly states that reduction variables must be singular. You are however, allowed to have a list of them. Thus the following directive is legal:

```fortran
C$\text{OMP PARALLEL DO REDUCTION(+:: var1, var2, var3, var4, var5)}
```

while

```fortran
C$\text{OMP PARALLEL DO REDUCTION(+:: var)}
```

where `var` is declared

```fortran
DOUBLE PRECISION, dimension(5):: var
```

is illegal.

### 3 The NAS-BT benchmark

The NAS-BT parallel benchmark is a pseudo CFD application [BHW+95]. It simplifies a full application code by simplifying the boundary conditions and I/O. It also applies a regular structured grid, which is not always the case for real application codes.

It sets out to solve a formulation of the 3-d Navier-Stokes equations, using a finite difference scheme in the spatial discretization. Time integration is done by an implicit scheme. This implies the solving of a large system of algebraic equations in each time step. The non-linearity of the partial differential equations is inherited by the discretized system. The nonlinear terms are however, expanded in Taylor series and truncated after the linear terms. Thus effectively we are left with a linear system.

Each grid point holds 5 variables. Their solution is coupled, implying that all over the place we have 5x5 blocks of matrices. In each time step the coefficient matrix (The Jacobian) as well as the right hand side are updated. The same problem are solved for all the 3 NAS-application benchmarks, but three different techniques are applied to solve the linear system.

In the NAS-BT we apply an iterative scheme, the SSOR method. In this case we split the original system matrix in an upper and a lower block triangular part. A new iterate is obtained by solving a lower block triangular system followed by an upper block triangular system.

The algorithm for one time step is then:

1. Compute RHS (right hand side).
2. Form and solve the lower block triangular system.
3. Form and solve the upper block triangular system.
4. Update $U(n+1) = U(n) + \Delta t$.

These steps are sequential in nature, thus we need to search for the parallelism within each step. Luckily there is a lot of it. Step 1 and 4 are embarrassingly parallel and so is building the matrices in step 2 and 3. The solution of the triangular systems turns out to separate nicely into independent block tridiagonal systems.

All data is stored in 5-d arrays. 5x5 blocks of 3-d data. The data access pattern shifts as we go along. Typically one access 3 5x5 blocks which are consecutive along one of the spatial dimensions.
4 Experiments

We have taken the public available sequential implementation of the NAS-BT benchmark, inserted the necessary compiler directives and let the F77-compiler on the Origin 2000 parallelize it. At some places the code needs careful examination in order to disclose which variables to declare as private and which ones should be shared. As usual the key thing is to find the outer do-loops available for parallelization. The code consists of 30 routines with a total of 3400 lines of Fortran. The initial porting to standard do-loop parallelism took only 2 days. Most of the time was spent in examining the body of the do-loop to determine which variables could be SHARED and which one had to be PRIVATE.

Our experiments focus on testing the reliability and functionality of SGI’s OpenMP compiler, as well as its efficiency. We also played around with the $OMP SECTIONS directive and will report on our experience with this. Results reported in this paper are run on a system in full production mode with a relative high load. Resource allocation is managed by the system and occasionally we observe unpredictable slow down, only possible to explain by shortage of resources due to the systems allocation policy. All codes are compiled with -O3 for optimization.

4.1 Reliability and functionality

To be a first release the compiler is pleasantly reliable. So far we have not encountered any obvious bug in the compiler. We are missing one of essential feature; The possibility for nested parallelism.

4.2 On parallel sections

Updating RHS is done in 6 different routines. Each routine computes its contribution to the new right hand side. The computation of the different contributions are totally independent, and as such a perfect case for the parallel section technique. Ideal, all one should need to do would be inserting the $OMP PARALLEL SECTIONS directives with the REDUCTION clause as shown below

```c
$OMP PARALLEL SECTIONS REDUCTION (+: rsd)
$OMP SECTION
    call rhs_xa (nx, ny, nz, ru, q, rsd, u)
$OMP SECTION
    call rhs_xb (nx, ny, nz, rsd, u)
$OMP SECTION
    call rhs_ya (nx, ny, nz, ru, q, rsd, u)
$OMP SECTION
    call rhs_yb (nx, ny, nz, rsd, u)
$OMP SECTION
    call rhs_za (nx, ny, nz, ru, q, rsd, u)
$OMP SECTION
    call rhs_zb (nx, ny, nz, rsd, u)
$OMP END PARALLEL SECTIONS
```

The use of the REDUCTION clause is, however, not allowed in this case as rsd is an array. Thus we were forced to extend rsd by an extra dimension, of size 1:6, and store each local contribution separately. An extra piece of code was added to do the reduction over this dimension after all the parallel sections were done. These changes to the code, violate one of the most attractive feature of directive based SMP-parallelization. That only one code is necessary for running in parallel as well as sequential mode.

The number of arithmetic operations is exactly the same in the 3 routines rhs_*a, but different from the numbers of arithmetic operations in the routines rhs_*b. The data is, however, accessed in different order in the x, y and z cases. This makes the execution time different for the 6 sections. Timing for a typical iteration is shown in Figure 1, and as shown here a static sectioning leads in this case to an inevitable load imbalance.
Figure 1: Execution time as recorded by VAMPIR for parallelizing the code above using \texttt{C\texttt{SOMP PARALLEL SECTIONS}}.

As noted above, we were forced to add an extra dimension to our arrays to apply the \texttt{PARALLEL SECTION} directives, due to the missing functionality of the \texttt{REDUCTION} clause. On a Distributed Shared Memory (DSM) system, like the Origin performance obviously is influenced by the distribution of the data. Done correctly we are able to force the system to allocate the data at the processor who are responsible for calculating the corresponding contribution to the right hand side. However, this is the worst possible distribution for the reduction step as the data now is distributed across all the participating processors. As seen in Figure 1 the time for doing the reduction is about 20\% of the time calculating the different contribution. A simple operation count does however indicate that it only amounts to 5 \% of the arithmetic work. We claim that this situation is not unique to our example, but is quite likely to happen in situations like this, where different processors computes independent contribution to overall results, and a global reduction is needed to accumulate the ‘grand total’.

Another non-desirable feature is also illustrated in this example. Dividing into a fixed number of sections allows for little flexibility in the number of processors to parallelize over. In this case the only reasonable choices are 2, 3 or 6 respectively. Since each of these tasks are rich in fine grained loop level parallelism, this form of parallelism could be applied as well. For OpenMP we could then apply an outer coarse grain parallelism using \texttt{C\texttt{SOMP SECTION}} and an inner fine grain do-loop parallelism. By combining the two level of parallelism we should be able to obtain the best possible scalability combined with good load balancing. This strategy is however, not yet available to us as the current implementation of SGI’s OpenMP does not support nested parallelism.

### 4.3 Efficiency of do-loop parallelization

In this section we report on our experience using \texttt{C\texttt{SOMP PARALLEL DO}} directives on the do-loops. Standard SGI profiling, \texttt{asan} and \texttt{prof} (pc-sampling) [Gra], shows that 99.7\% of the code is being parallelized.

Our analysis of the runtime behavior of the program is based on the use of the VAMPIR (Visualization and Analysis of MPI Resources) performance visualization tool [NAW+96] [Gmb], which has been rebuild by Zentrum für Hochleistungrechnen (ZHR) at the Technical University of Dresden to be able to analyse OpenMP programs as well as MPI programs.

The outer loop of the code is representing time stepping. As always, time is sequential, thus parallelism
is achieved by distributing the update of the different grid points to processes within a time step. By using VAMPIR it is easy to confirm that the behavior of a parallelized loop is similar from one time step to another within the same run. Therefore, wishing to investigate a parallelized loop, it is sufficient to pick one arbitrary time step.

For all of the parallelized loops in this code the operation count is exactly the same in all iterations. Thus provided the number of threads divides the number of loop-iterations, the workload should be perfectly balanced. Observing the execution time of a sequence of parallelized loops, we do, however, see that the execution time is far from evenly distributed across the threads. Typically the fastest threads for one loop will be the fastest threads for the other loops as well. A typical scenario is illustrated in Figure 2.

![Figure 2: Work performed by 25 threads in one time step as displayed by VAMPIR.](image)

Here we display the work done for each thread during one time step on a run for 25 threads. The time line shows the wall clock time in seconds. The different loops are showed with different colors. The numbers showing on some of the loops are the loop identity numbers, and one number represents one loop iteration. These are showing if there is room for them. If there is even more room, the name of the loop is showing instead of the loop identity number. White areas represents no act and the red area on thread 1 is the time spent waiting for the rest of the threads to finish.

By inspection we can see that the pattern is repeating itself. The fastest threads in one time step, continue to be the fastest in the next and so forth.

Figure 3 zooms in on one loop. For a high number of threads, thread 1 and 2 are really working fast (3ms each iteration), while the rest of the threads are working much slower (19ms for their first iteration). Since differences in execution time can not be explained by differences in arithmetic work, and we know that each thread had the exclusive attention of one CPU, our prime suspect for this huge differences in execution time is differences in data access. The suspicion is under pinned by an inspection of the code.
which shows that to do 12 flops a CPU needs 6 double precision numbers, and no reuse of data is possible. Thus this piece of code is very data intensive.

The CRAY Origin 2000 memory is physically distributed across nodes, with two processors located at the same node, having equal access to this memory. Remote CPUs also have direct access to this memory, but have a longer way to go. Moreover, if many processors compete for access to one node's memory simultaneously, the single access channel to this memory is doomed to be a bottleneck.

The fast work of thread 1 and 2 indicates that these two are located at the same node as the data needed for the computation. While the slow working threads do not have the data located at their node. Therefore we can conclude that the speed of the threads depends strongly on the location of memory. References to location in the remote memory of another processor takes longer to complete than references to locations in local memory. This can severely affect the efficiency of a program that suffer from a large number of cache misses.

When most of the threads are finished, the threads still working starts to work faster (6ms each iteration), showing how the speed depends on the number of threads accessing data from the same memory location. Note that the time from start to end for this loop is 41ms.

We also found that the forking and joining of the loops (parallel overhead) is not time-consuming, and can not be used as an explanation for the poor speed-up seen in this example.

### 4.4 Data distribution

To get good performance in programs which is bandwidth limited, it is important to distribute data across the memory modules to increase the aggregated bandwidth, and then try feeding the caches of individual processors from local rather than remote memory. On a Distributed Shared Memory system like the Origin 2000 it therefore makes lot of sense to distribute not only work, but data as well. This is acknowledged by SGI in their Power Fortran were they provide directives controlling data distribution,
as well as the AFFINITY clause to the DOACROSS directive, which enable us to glue the thread working on a piece of data to a processor sitting on the same node as the memory keeping the data.

The data distribution directives is not a part of OpenMP, but is available on the Origin Fortran compiler. To apply the affinity clause, one need however to replace C$OMP PARALLEL DO with Power Fortran's C$DOACROSS, so this was not used.

C$OMP PARALLEL DO PRIVATE(i, j)
   do j = 1, ny
      do i = 1, nx
         A(i, j) = i * j + A(i-1, j)
      end do
   end do
end do

If A is declared as A(nx, ny), this loop suggests a distribution
C$DISTRIBUTE A(*, block)

This distribution directive divides the data in blocks of size nb = \( \left\lfloor \frac{n_b}{P} \right\rfloor \) and tells the compiler to allocate memory for column i on the same node as processor \( \left\lfloor \frac{i}{P} \right\rfloor \). Note that in case ny mod P \neq 0, some of the processors will get fewer or no columns. For the example displayed in Figure 4, ny = 64 and P = 25, we notice that only the 22 first processors get any work, and the last only perform one iteration over the columns. Except for this feature we now get a much better distribution of work and for this particular case the time from start to end for this loop is only 13ms compared to 41 ms for the previous one!

4.5 Scheduling

Another possibility for evening out the load imbalance due to differences in speed of the different threads, is applying a different scheduling of tasks to threads by using one of OpenMP's environment routines. SCHEDULE(DYNAMIC, chunk) specifies another scheduling; the loop iterations are broken into pieces of a size specified by chunk. As each thread finishes a piece of the loop space, it dynamically obtains the next set of loop iterations. When no chunk is specified, it defaults to 1. We added SCHEDULE(DYNAMIC) to the C$OMP PARALLEL DO directive for the same loop as referred to earlier. Now, when each thread finishes a loop iteration, it dynamically obtains a new loop iteration. The result is seen in Figure 5. Now the different threads were kept working almost equally long, but almost none reduction in the overall running time was observed. This comes as no surprise as the scheduling is not any cure for fixing the fundamental problem, the memory bandwidth bottleneck.

We also tried combining data distribution with dynamic scheduling. This kept the processors busy, but the overall execution time went up compared to only using data distribution. We strongly believe the reason for this is that the dynamic scheduling breaks the affinity of threads to local data, and thus force the processors to non-local data access.

4.6 Comparing OpenMP with MPI

We wanted to compare the OpenMP version with the public available MPI version of the same NAS-BT as used above. One drawback of this version is that it is only possible to run when the number of processors is a quadratic number (1, 4, 9, 16, 25, ...).

Comparing MPI with OpenMP, the most obvious result is the much better speed-up for MPI than for OpenMP. This is seen in Figure 6.

OpenMP and MPI both achieve their parallelism by dividing the computation of the spacial gridpoints to the processors. Besides applying different programming model, the parallelism differs in how to divide data and the parallel granularity. To minimize the communication the MPI code applies a 2 dimensional partition of data and thus achieve a better 'surface-to-volume' ratio than the OpenMP, which due to the
lack of nested loop-parallelism has no choice, but slicing the data in 1 dimension only. A second difference is the parallel granularity. In the MPI code there are far less synchronization points within a time step than for the OpenMP code. Large chunks of data is exchanged between processors up front, giving each of them a juicy piece of work to chew on before further synchronization is needed.

Apparently the same coarse grain parallelism should be possible to obtain in OpenMP by letting C$\text{OMP PARALLEL}$ encapsulate multiple do-loops and allowing a thread to continue on the next loop as soon as the first is finished. This is accomplished using a NOWAIT clause to the C$\text{OMP END DO}$ like this:

```
C$\text{OMP DO}
  do-loop
C$\text{OMP END DO NOWAIT}
```

Inserting NOWAIT clauses turned out to be an error prone process. To do it correctly the programmer needs to do the same analysis of dependency between different loops, as needed to determine when and where to communicate when programming with MPI. Finding bugs introduced when adding NOWAIT was a painful job. The reward was also small in terms of speed-up in our case, as the memory bottleneck still remains, and the thread running slowest for one particular loop turned out to be the slowest one in the next loop as well.

Probably the greatest advantage performance wise of using MPI is that the correct distribution of data to processors for optimizing cache performance, comes as a free bonus with the MPI implementation. We find this, for the programmer, hard to control in OpenMP.
5 Conclusion

OpenMP is a promising step towards a standard for SMP-programming. To be successful it is important that reliable and efficient solutions become available on most vendors platform soon. A similar OpenMP API for C, should also be made available soon.

SGI should work hard on implementing the entire OpenMP standard on their compiler. Integration with the automatic parallelizer should also be done. On a physically distributed memory system like the Origin 2000, the data distribution directives of Power Fortran have proved valuable. We would like to see these as a part of the OpenMP compiler.

Acknowledgment

We greatfully acknowledge the assistance of Prof. Wolfgang Nagel, Stephan Seidl and the rest of the staff at ZHR at TU-Dresden in this project. The use of their VAMPIR prototype for OpenMP has been of great help to us, and so has the free use of computer time on their Origin system been. The first author also wants to thank for their kind hospitality and financial support at her visit in Dresden.

References

[BHW+95] David Bailey, Tim Harris, Saphir William, Rob van der Wijngaart, Alex Woo, and Maurice Yarrow. The nars parallel bench-
Figure 6: Speed-up for the NAS-BT benchmark class A, for the MPI-program and the two versions of OpenMP.


Experimentation of Data Locality Performance for a Parallel Hierarchical Algorithm on the Origin2000

Xavier Cavin, Laurent Alonso and Jean-Claude Paul

LORIA - INRIA Lorraine
615 rue du Jardin Botanique, BP 101,
F-54600 Villers-lès-Nancy, France

August 20, 1998

Abstract

Hierarchical algorithms form a class of applications widely being used in high-performance scientific computing, due to their capability to solve very large physical problems. They are based on the physical property that the further two points are, the less they influence each other. However, their irregular and dynamic characteristics make them challenging to parallelize efficiently. Indeed, two conflicting objectives have to be taken into account: load balancing and data locality.

It has been shown that the message passing paradigm was not well suited for this kind of applications, because of the intensive communication they introduce. Implicit communication through a shared address space appears to be better adapted. Particularly, the ccNUMA architecture of the Origin2000 can help us getting the desired data locality through its memory hierarchy.

We have experimented a parallel implementation of a well known computer graphics hierarchical algorithm: the wavelet radiosity. This algorithm is a very efficient approach to compute global illumination in diffuse environments but still remains too much time and memory consuming when dealing with extremely complex models.

Our parallel algorithm focuses on load balancing optimization and heavily relies on the ccNUMA architecture efficiency for data locality. Load balancing is handled with a general dynamic tasking mechanism with specific improvements. Minimal efforts are made towards memory management (like the writing of thread-safe non-blocking malloc/free C functionalities) and the Origin2000 proves all its capabilities to efficiently handle the natural data locality of our application.

The implemented algorithm allows to compute the illumination of a complex scene (a cloister in Quito, composed of 54789 initial surfaces and leading to 600000 final meshes) in 2 hours 41 minutes with 24 processors. To the knowledge of the authors, this is the most complex "real world" scene ever computed.

1 Introduction

As scientific and engineering computing requires more and more computational power, parallelism appears to be one, if not the only, efficient response. Among scientific applications, a class of solutions, namely hierarchical algorithms, is widely being used in order to solve very large physical problems. These include fluid dynamics, chemistry, structural mechanisms, semiconductor and circuit simulation, oil reservoir simulation [4], or more recently computer graphics [8], and so is representative of a large class of problems. That is the reason why they greatly
prompted the development of supercomputers and are now likely to get their benefits.

The design of an efficient parallel hierarchical algorithm, however, is challenging, because it involves dealing with complex issues. In particular, simultaneously management of load balancing and data locality quickly becomes a headache, both because of the non-uniformity of the physical domain being simulated and because of the dynamic and unpredictable changes of workload and communication across the computation of the solution.

Chan, in [4], has pointed out that parallel (hierarchical) architectures should be designed especially to support the hierarchical communication and synchronization needs of these algorithms. Singh et al. focused in [12] on the implications of the hierarchical $N$-body methods and concluded to the necessity of an efficiently supported shared address space versus the message passing paradigm for this kind of applications.

In another paper [13], they studied load balancing and data locality for the hierarchical radiosity application, on the 48 processors Stanford DASH Multiprocessor, and confirmed the appropriateness of cache-coherent, distributed shared memory (i.e. DSM) supercomputers for this kind of algorithms. However, their experiments were limited to a very small model (94 input polygons) and they had to "extrapolate" their conclusions. Indeed, dealing with much larger environments (typically hundreds of thousands of polygons) changes the amount of communication, and so the dimension of the problem. That is why it seems interesting to us to run these experiments, with a more recent algorithm (the wavelet radiosity), on "real world" scenes and on a commercial shared-memory supercomputer, the SGI Origin2000.

Section 2 of this paper discusses hierarchical methods in general, detailing hierarchical $N$-body methods and hierarchical radiosity, and the issues involved in their parallelization. Then, Section 3 presents our parallel algorithm and some interesting implementation details. Experiments are described in Section 4 and results are commented in Section 5. Finally, Section 6 concludes and presents future works.

2 Hierarchical Methods

Many physical processes, described by the mathematical models of the physical laws, are hierarchical by nature. That is, they exhibit a large range of scales, both in space and time. A given point in the physical domain is progressively less influenced less frequently by parts of the domain that are further away from it. Hierarchical algorithms efficiently use the range of length scale to capture the global features of the solution. Here are some examples of such algorithms: multi-grid methods, domain decomposition methods, adaptative mesh refinement algorithms, wavelet basis, $N$-body methods, hierarchical radiosity. All these algorithms have common features: they result in near optimal sequential complexities and possess high potential of parallelism.

The remaining of this Section focuses on classical hierarchical $N$-body methods and on hierarchical radiosity, and presents the issues of their effective parallelization.

2.1 Hierarchical $N$-body Methods

The $N$-body problem studies the evolution of a system of $n$ discrete particles (or bodies) under the influence exerted on each particle by the whole ensemble. The naive algorithm which computes all interactions between each pair of particles suffers from a $O(n^2)$ complexity. Fortunately, this complexity can be reduced to $O(n \log n)$ by hierarchical algorithms [1, 2].

Hierarchical $N$-body methods are based upon a physical property that dates back to Isaac Newton (1687): “If the magnitude of interaction between particles falls off rapidly with distance (as it does in most physical interactions, such as gravitation or electrostatics with their $1/r^2$ force laws), then the effect of a large group of particles may be approximated by a single equivalent particle, if the group is far enough away from the point at which the effect is being evaluated.”, as shown by Figure 1 [12].

Actually, hierarchical $N$-body methods can be applied to many physical domains, including astrophysics, plasma physics, molecular and fluid dynamics or computer graphics. They so represent a large class of supercomputers potential users.
2.2 Hierarchical Radiosity

Radiosity methods are an efficient approach to compute global illumination in Lambertian (i.e., diffuse) environments. They are based on the physics of light transport and capture both direct illumination (by light sources) and indirect illumination (through multiple inter-reflections), resulting in a view-independent solution.

The radiosity - power per unit area \([W/r^2]\) - on a given surface is defined as the light energy leaving the surface per unit area. Let \(\mathcal{M}\) denote the collection of all surfaces in an environment. Let \(\chi\) be a space of real-valued functions defined on \(\mathcal{M}\), that is, over all surface points. Given the surface emission function \(g \in \chi\), which specifies the origin and directional distribution of emitted light, we wish to determine the surface radiosity function \(f \in \chi\) that satisfies\(^1\):

\[
f(x) = g(x) + k(x) \int_{\mathcal{M}} G(x', x) f(x') \, dx'
\]

(1)

where \(k(x)\) is the local reflectance function of the surface (i.e., we suppose that the surfaces are ideally diffuse), and \(G(x', x)\) is a geometry term defined by:

\[
G(x', x) = \frac{\cos \theta_{x'} \cos \theta_x}{\pi r_{x'x}^2} v(x', x)
\]

(2)

and consisting of the cosines made by the local surface normals with the vector connecting the two surface points \(x\) and \(x'\), of the distance \(r\) between these two points (see Figure 2), and of the visibility function \(v(x', x)\), whose value is in \(\{0, 1\}\) according to whether the line between the two points \(x\) and \(x'\) is respectively obstructed or un-obstructed.

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]
over, the algorithm in itself does not proceed over hundreds of time-steps, but either by a few gathering passes (each surface gathers energy from all others) or many shooting iterations (the most energetic surface propagates its energy to all others).

The wavelet radiosity algorithm, introduced by [7] and [11] is a generalization of the hierarchical radiosity: the radiosity function is no longer approximated as being uniform on a surface element but is now projected on wavelet bases, thus allowing a better subdivision and error control.

2.3 Parallel Issues

As pointed out in [4], hierarchical algorithms possess relatively high degrees of parallelism. If the parallelization of hierarchical N-body methods has been well studied [12, 13], the design problem of parallel hierarchical radiosity has not yet been completely addressed [3, 6, 13, 14]. However, all works seem to prove that the message-passing paradigm is not well suited for the dynamic, irregular and unpredictable behavior of hierarchical algorithms. On the contrary, cache-coherent DSM supercomputers seem to be able to efficiently support the hierarchical constraints through their hierarchy of memories.

Unfortunately, choosing a shared memory supercomputer (randomly, the SGI Origin2000) is not sufficient to get an efficient parallel application! Indeed, as in many scientific algorithms, designers have to deal with two key but conflicting problems: load balancing and data locality.

Related to these problems is the choice of the granularity used for the decomposition of the algorithm into elementary tasks. For the hierarchical radiosity method, the granularity can go from very coarse (a task is the set of interactions between a surface and all other surfaces) to very fine (surface-surface element interaction or even surface-element-surface element interaction) through intermediate (surface-surface interaction). Obviously, the smaller the granularity, the easier the load balancing but the worse the data locality (and vice versa).

3 Parallel Wavelet Radiosity

We focus in this Section on our work for the parallelization of a wavelet radiosity algorithm. We start with the description of the sequential version we use, then we present our parallel algorithm. Finally, we explain some interesting implementation details.

3.1 Sequential Algorithm

Our work takes place inside a large project, named Candela, which has been designed to provide a flexible architecture for testing and implementing new radiosity and radiance algorithms [10]. It was at the same time intended to be able to deal with real input data and to compute physically correct results.

Several sequential wavelet algorithms have already been implemented and experimented in [5]. Our sequential experiments have shown that the wavelet coding of the radiosity function is very important in order to obtain correct results. Moreover, one particular instance of our wavelet algorithms came out as the most effective one to deal with large architectural scenes: the progressive shooting wavelet radiosity without link storage.

This algorithm sequentially handles the most energetic emitter (either a direct light source or a reflecting surface) and propagates its energy to all scene receivers (surfaces). Each energy propagation (i.e.
interaction) is done “hierarchically”: an oracle function decides (on energetic or geometrical considerations), which of the two surfaces (maybe none of them) should be subdivided to enhance the precision of the solution; then the energy propagation is recursively done between the non-subdivided surface and each of the children of the subdivided one (see Figure 3). In traditional hierarchical radiosity methods, links are created between surface elements not involving subdivision; because of the huge amount of memory this requires, our algorithm does not store the links.

3.2 Parallel Algorithm

Despite all the qualities of our sequential algorithm, time and memory requirements still remain too high for effective use on a single workstation, especially when the scenes to simulate are very large (the common case for architectural simulations). Thus, parallelism appears to be the alternate way to bypass the single workstation limitations.

A complete study of our choices and algorithms can be found in [3]. We just give here a short description of the important points.

Basically, we have chosen to decompose the sequential algorithm with the intermediate granularity (cf. Paragraph 2.3), that is to say, we have defined a task as a standard surface-surface interaction. We hoped this would be fine enough to enable an efficient load balancing (at least in the case of large scenes), but not too much to get some data locality (keep in mind that we heavily rely on the Origin2000 architecture to help us on this quest).

The parallel algorithm thus consists in distributing these tasks to processors, in the best load balanced way. Our experiments showed that a simple dynamic tasking algorithm with a single centralized tasks queue can give good results, at least at moderate processors scale (up to 40 processors). When scaling to a much larger number of processors, a distributed tasks queue algorithm, associated with a task stealing mechanism, may become necessary to avoid the single queue bottleneck.

3.3 Implementation

The Candela libraries consist of approximately 360 C++ classes. Even if it can seem challenging to parallelize C++ algorithms inside such a large platform, it is really important not to move apart from the sequential part in order to be able to make comparisons and to take advantage of its last enhancements.

Furthermore, Candela is built over the SGI Open Inventor library and intensively uses the scene graph structure and its associated nodes. Hence, we absolutely have no control over the storage and manipulation of the data structures. It is even worse in parallel, because of the non thread-safe behavior of the Open Inventor library: a catastrophic error (i.e. a core dump) may occur if several processes manipulate the scene graph at the same time.

Finally, the many dynamic memory allocations implied by the hierarchical wavelet algorithm have to be done in parallel in a non-blocking way, in order to avoid memory allocations congestion. Unfortunately, this is not the case with the standard IRIX memory allocation package, which has been made thread-safe by serializing the parallel allocations.

Consequently, we have developed an independent, general purpose C/C++ framework, which aims to facilitate the design of parallel programs, at least on the SGI Origin2000. Here are some of the available functionalities. First, it provides encapsulated calls to specific MP routines, such as processes management (we use m.fork’ed processes) and synchronization methods (locks, barriers). This allows us to automatically monitor the synchronization times and to consider porting the application under other operating systems and/or architectures.

Then, a set of preprocessor macros allows to transparently transform a variable which could potentially be modified by several processes at the same time (typically static class variables) into an array of variables (one per process). Finally, a new C memory allocation package (i.e. overloaded malloc, free, realloc, calloc functions) has been written. It allows fully parallel, contention free, memory operations without any modifications to the source code.
4 Protocol Considerations

This Section is intended to give a full description, as complete as possible, of our experiments protocol. We first describe the hardware/software configuration. Then we mention the measures we perform. Finally, we present our architectural test scenes.

4.1 Hardware and Software

We performed our experiments on the Silicon Graphics Origin2000 installed at the Centre Charles Hermite\footnote{See the web page: http://cch.loria.fr}, in France.

The machine is equipped with 64 processors organized in 32 nodes. Each node consists of two R10000 processors with 32 KBytes of first level cache (L1) of data on the chip, 4 MBytes of external second level cache (L2) of data and instructions and 128 MBytes of local memory, for a total of 8 GBytes of physical memory. The operating system running on it is the IRIX 6.5SE OS.

For sake of completeness, our application has been compiled with the MIPSpro 7.2.1 C++ compiler, with the following options: \texttt{CC -n32 -Ofast=IP27 -mips4 -r10000}.

4.2 Measures

For our data locality performance analysis, we intensively used the R10000 hardware performance counters, combined with the software tool \texttt{perfex}. In particular, we have chosen to study:

- \textit{Speed-up}. This is defined by the fraction between the best sequential time over the parallel time obtained with \textit{n} processors.

- \textit{Memory overhead}. This is the fraction of time spent in memory over the total execution time.

- \textit{L1 cache hit rate}. This is the fraction of data accesses which are satisfied from a cache line already resident in the primary data cache.

- \textit{L2 cache hit rate}. This is the fraction of data accesses which are satisfied from a cache line already resident in the secondary data cache.

The curves we present further in this paper are automatically generated, with the help of \texttt{shell scripts}, from our application and \texttt{perfex} output traces. We are currently working on a graphical interface for instrumenting, running and exploiting the parallel experiments. Such a tool would be a great help for parallel program tuning and optimization phases.

4.3 Test Scenes

Our experiments were performed on three test scenes, all coming from real world applications, but with different characteristics:

- \textit{Sta
nislas Square Opera} in Nancy. This test scene comes from an evaluation project of potential new lighting design. The geometrical model was created from architectural drawings. The direct illumination is computed using accurate light and reflectance models.

- \textit{Cloister} in Quito. It is also a lighting design project, but it was chosen because the effects of indirect illumination (inter-reflections) are more visible. It serves as a life-size test.

- \textit{Soda Hall}. The Soda Hall building has become a reference test scene. It is suitable for virtual reality environments and interactive walk-through. We both consider a single room of this building (with high precision parameters) for speed-up measures, and one complete floor with furniture as another life-size test.

Table 1 gives their numerical characteristics (number of initial surfaces, number of light sources and number of final meshes) and reference computation times.

<table>
<thead>
<tr>
<th>Scene</th>
<th>Initial</th>
<th>Lights</th>
<th>Final</th>
<th>Time [s]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Opera</td>
<td>38 258</td>
<td>98</td>
<td>513 310</td>
<td>40 982</td>
</tr>
<tr>
<td>Cloister</td>
<td>54 780</td>
<td>83</td>
<td>597 135</td>
<td>9 663</td>
</tr>
<tr>
<td>Room</td>
<td>9 189</td>
<td>3</td>
<td>232 349</td>
<td>11 001</td>
</tr>
<tr>
<td>Floor</td>
<td>144 255</td>
<td>163</td>
<td>1 721 354</td>
<td>54 627</td>
</tr>
</tbody>
</table>

Table 1: The test scenes.
5 Results and Discussion

In this Section, we analyze the performance of our parallel algorithm on the test scenes of Paragraph 4.3. Table 1 gives the complexity of each scene and the computational time needed to simulate them. We used the Opera and room models for a complete analysis and the cloister and floor models as punctual life-size tests.

We start by presenting a performance evaluation, in terms of speed-up and synchronization times. Then we focus in detail on the data locality analysis, and we complete the Section with an open question about the scalability.

5.1 Speed-up Results

Figure 4 shows the speed-up results for the Stanislas Square Opera illumination and for the Soda Hall room simulation. The speed-ups, although not linear, are quite good for both scenes.

![Speed-up Graph](image)

Figure 4: Speed-up measures.

Our application was instrumented to compute the time lost at synchronization points (i.e. locks and barriers); unfortunately, a bug in `time()` and `gettimeofday()`\(^4\) prevents us from exploiting the results. However, when it did not occur, we could notice that synchronization times increase slowly with the number of processors and only represent, in the worst cases, about 10% of the total execution time.

\(^4\)Making sometimes time go backward of 1 second!

The intermediate granularity we have chosen allows a fair load balancing with an algorithm of low complexity, at least in the case of large scenes and at a moderate scale. Indeed, reducing the amount of work (i.e. the scenes size) or increasing the number of processors might have processors either waiting on a lock, or remaining idle at the end of the computation, especially when only a few interactions, sharing common receivers, are time consuming compared to the others.

Let us now see the impact of this granularity choice on the data locality afforded by the algorithm.

5.2 Data Locality

The aim of our study was to evaluate the capabilities of the SGI Origin2000 architecture to handle the hierarchical behavior and inherent data locality of the parallel hierarchical wavelet radiosity algorithm.

Actually, the design of our parallel algorithm did not take into account the data locality problems. Indeed, we expected that the granularity we had chosen would naturally increase the data locality. First, when a given processor handles a surface-surface interaction, it computes the energy transfers at each needed level of the hierarchy, thus ensuring a good reuse of the data structures. Then the visibility computations for this interaction involve points of the two surfaces and thus lead the same parts of the BSP structure to be traversed. Finally, the ccNUMA architecture of the SGI Origin2000 [9] seems to be very promising to handle the dynamic behavior of our hierarchical algorithm.

The first point of our experiments concerns the memory overhead. Figure 5 shows the ratio of time spent in memory to total execution time for the Opera and room models. We can first notice that our application intensively uses the memory (about 70% for a single processor and about 50% with more than 8 processors). This fact is due to the many memory allocations involved by the hierarchical algorithm and confirms the necessity of an efficient non-blocking memory management.

Moreover, the communication to computation ratio decreases slightly with the number of processors. This can be explained by the fact that a large amount
of memory (more than 128 MBytes\(^5\)) is needed for the simulation of both scenes: the memory becomes better distributed as the number of processors increase, and the latest have only access to a part of the total memory.

![Figure 5: Time accessing memory/Total time.](image)

At the same time, the cache miss rates are very low: less than 5% for the \(L1\) cache and less than 10% for the \(L2\) cache (see Figure 6).

![Figure 6: Cache Miss Rates.](image)

This rates have surprisingly low values, if we consider that we do not focus our algorithm on data locality. However, this seems to prove that the working set (i.e. hierarchical data structures + parts of the BSP) of an elementary task has an adequate size for the memory caches of the Origin2000 and that its ccNUMA architecture is very efficient to handle the dynamic communications of the algorithm.

To be complete, our two punctual life-size tests, with the Quito cloister and the Soda Hall floor models, gave similar results. Naturally, we plan to confirm these experiments with a complete study of these two models.

### 5.3 Scalability Problems?

The careful reader will have noticed that the results we have shown only concern 1 to 32 processors, while our Origin2000 has 64 processors. Actually, Figure 7 shows the complete speed-up curve for the Stanislas Square Opera model (the same problem appears with the room scene): with more than 28 processors, the speed-up is limited about 20. There is clearly a scalability problem.

![Figure 7: Speed-up for the Opera model.](image)

What seems very surprising to us is that all prefix informations (\(TLB, L1, L2, \ldots\)) and synchronization times are coherent from 1 to 60 processors.

One common cause to speed-up decrease is known under the term of false sharing: when a process modifies a memory page owned by one or several other processes, this page is automatically invalidated in their memory caches, thus causing a cache miss and a page transfer for the next access. The more processes there are, the more it is likely to happen. It may be more critical within our application, because

---

\(^5\) The size of a processor's local memory.
we did absolutely nothing (for instance, data alignment) to prevent this phenomenon.

Fortunately, a specific counter of the R10000 processor (counter 31) allows to study false sharing, by counting stores or prefetches with store hints to shared blocks in secondary memory cache. Figure 8 shows the evolution of this counter with the number of processors for the same scene. Surprisingly, our application does not suffer from false sharing problems: another good point for the Origin2000.

![Figure 8: Evolution of R10000 counter 31.](image)

However, "the truth is out there": for an unknown reason, the number of cycles increases with the number of processors, thus decreasing the speed-up, as shown by Figure 9. This curve is interesting: it presents a (non dramatic) first grow from 1 to 4, due to the parallelism overhead. Then, it remains constant (the normal case) to 20, giving a quite linear speed-up. It then begins to slowly increase from 24 to 40 before abnormally growing until 60, with a strange decrease at 56. Looking at Figure 7, we can notice that the speed-up curve exactly follows these variations.

It just remains to locate the problem, that is to say, the part of the code which takes more time as the number of processors increase. We first tried the ssrun tool, but the results it gave were not coherent enough to help us. We would have liked to try the pixie tool, but, unfortunately, it seems to crash the computer under the IRIX 6.5SE OS, and so we cannot use it. The scalability problem remains open.

![Figure 9: Scalability problem.](image)

6 Conclusion

We intended in this paper to make a data locality performance analysis of the SGI Origin2000 for a modern computer graphics hierarchical application: the wavelet radiosity algorithm. The machine proves to be, as it was designed for, very efficient to handle the dynamic and unpredictable communication and synchronization needs of this algorithm. The memory management performances (memory caches, false sharing) of its ccNUMA architecture, from 1 to 60 processors, allow us to focus on load balancing problems, leaving the data locality to the computer.

However, the speed-up of our parallel application is currently bounded to 20, from 28 to 60 processors. Indeed, for an unexplainable reason, the number of cycles suddenly increases after 20 processors. Currently, the state of the available tools and of our knowledges do not allow us to solve this scalability problem: we are still working on it.

During this study, we have felt the need to work on the development environment. Indeed, the SGI Origin2000 provides some efficient, but not convivial, tools for applications development and tuning. We so have developed a framework for C/C++ applications development, synchronization times monitoring and perfex outputs automatic graphical analysis. We plan to apply it to a completely different application to be sure of its reusability.
Acknowledgments

The authors would like to thank François Cuny, Slimane Merzouk and Christophe Winkler for their work on the Candela platform, Marc Albouy, from Electricité de France which provided the Stanislas Square and the Quito models, Carlo Sequin who provided the Soda Hall. We also thank the Centre Charles Hermite, which owns the Origin2000, and Alain Filbois for his great technical support on the computer.

References


A Parallel One-dimensional FFT for Cray T3E

O. Haan

Gesellschaft für wissenschaftliche Datenverarbeitung mbH Göttingen, Am Faßberg, D-37077 Göttingen

The paper describes the parallelization of the one-dimensional FFT algorithm on the Cray T3E distributed memory parallel computer. A two-dimensional formulation for the one-dimensional FFT is employed in order to reduce the communication overhead. The implementation performs with more than 70 MFlops per processor up to 32 processors and more than 35 MFlops up to 512 processors.

1. Introduction

The fast Fourier transform (FFT) algorithm rearranges the computation of the discrete Fourier transform of a length n series into $O(\log_2 n)$ steps of $O(n)$ operations, reducing the operation count from $O(n^2)$ to $O(n \log_2 n)$. In each of the intermediate steps all elements of the series have to be accessed with varying strides. This large amount of data movement causes problems for fast implementations of the FFT on parallel computers with a hierarchy of cache, local and remote memories.

On modern RISC processors the FFT performs typically with a large fraction of the processors maximal speed as long as all elements of the series can be kept in the memory level closest to the CPU. The performance drops by an order of magnitude if the series does not fit into the cache and therefore data have to be read from and written to the main memory in each of the intermediate steps. A further reduction in speed results when elements have to be collected from different nodes of a parallel processor in order to form the intermediate steps of the FFT.

In order to avoid the access to the whole set of data during each of the $O(\log n)$ intermediate steps of the FFT, one can employ a reformulation of the one-dimensional transform in a two- or higher-dimensional setting. This algorithm concentrates the access to the higher memory levels into three separated steps of matrix transpositions. A further reduction to a single transposition step can be achieved by working with a special non-natural order of the elements to be Fourier transformed.

This reformulation has been developed by several authors ([8], [4], [1], [5]). An implementation for IBM’s SP2 parallel processor was described in [2].

The present paper analyses the performance of the two-dimensional formulation on a Cray T3E parallel computer. If covers in some detail the optimization of the transposition steps, taking into account the special structure of the memory system of the T3E.
2. The FFT algorithm

The discrete Fourier transformation (DFT) of a series $x(j)$ with $n$ elements

$$
\hat{x}(k) = \sum_{j=0}^{n-1} W(n)^{jk} x(j), \quad \text{where} \quad W(n) = e^{-\frac{2\pi i}{n}}
$$

can be determined from a sequence of partial transforms $x_{s}, s = 1, \ldots, q$, whenever the size $n$ of the series admits a factor decomposition of the form

$$
n = n_1 \cdots n_q.
$$

(the presentation of the FFT follows [8] and [7], where also a list of references to the FFT can be found).

The partial transforms are defined as transformations over a subset of indices, into which $j$ can be decomposed in a digit representation corresponding to the factorization of $n$:

$$
x_s(j_1, \ldots, j_s+1, k_s, \ldots, k_1) = \sum_{j_1=0}^{n_1-1} \cdots \sum_{j_s=0}^{n_s-1} W(n_1 \cdots n_s)^{d(j_1, \ldots, j_s)d(k_s, \ldots, k_1)} x(d(j_1, \ldots, j_s)).
$$

The index mapping $d$ generalizes the digit representation to arbitrary number systems:

$$
d(j_1, \ldots, j_s) = j_s + n_s j_{s-1} + \cdots + n_s \cdots n_2 j_1.
$$

Two successive partial transforms are connected by a modified Fourier transformation in a single digit:

$$
x_s(j_1, \ldots, j_s+1, k_s, \ldots, k_1) = \sum_{j_s=0}^{n_s-1} W(n_s)^{j_s k_s} W(n_1 \cdots n_s)^{jd(k_s, \ldots, k_1)} x_{s-1}(j_1, \ldots, j_s, k_s-1, \ldots, k_1)
$$

Starting with

$$
x_0(j_1, \ldots, j_q) = x(d(j_1, \ldots, j_q))
$$

the Fourier Transform is obtained after $q$ steps as

$$
\mathcal{F}(d(k_q, \ldots, k_1)) = x_q(k_q, \ldots, k_1).
$$

The FFT algorithm reduces the operation count for the Fourier transformation from $O(n^2)$ to $O(n \log_2 n)$. Notice also, that the transformed series $\mathcal{F}$ is obtained in a digit reversed order [6]. In order to obtain a naturally ordered FT, one has to rearrange the elements of the transformed series, either at the end of the FFT algorithm, or after every partial transformation [9].
3. FFT and memory hierarchy

The large amount of data movement during the FFT algorithm leads to performance problems on computers with a memory hierarchy. On a single processor with cache this problem will make itself felt, when the size of the data to be transformed exceed the size of the cache. Then the elements of the series have to be accessed from main memory in each of the \(O(\log_2 n)\) partial transformation steps.

As an example, fig. 1 shows the performance of the routine cfft from Cray’s Scilib, which computes the complex one-dimensional Fourier transform.

For a transform of size \(n = 2^n\), three complex arrays with \(n\) elements have to be accessed, one for the series to be transformed, one for storing intermediate partial transforms and one to hold the precalculated trigonometric function values for the twiddle factors. The total amount of data to be accessed therefore is \(2^n \cdot 48\) Bytes. The cache of a T3E processor has a size of 96 KByte, it should be able to hold all data for a FFT with \(q = 11\). As can be seen from fig. 1, for \(q = 11\) the performance has already dropped by ca. 30\%. This is due to the three way associative organization of the cache. For large transforms the slow access to main memory reduces the FFT speed to 20\% of the maximal value. The two curves in fig. 1 for cfft differ in the use of the stream feature of the T3E. With active streams, consecutive data can be loaded faster from memory into the cache. The streams-off performance is displayed in order to compare with the six-pass algorithm (cf. section 4), which uses the shmemb library. This library is not stream-save on the older T3E systems, streams are switched off for all programs using shmemb on those systems.

The parallelization of the one-dimensional FFT algorithm on a distributed memory system leads to an additional slow down due to the access to remote data. To be specific, let the transform’s size be a power of two, \(n = 2^n\) and let also the number of processors be a power of two, \(np = 2^m\). As in section 2, split any index \(i\), enumerating the elements
of an array \( x \), into its \( q \) digits \( i_1, \ldots, i_q \), which are bits in the power-of-two case:

\[
i = d(i_1, \ldots, i_q) = i_q + 2 \cdot i_{q-1} + \cdots + 2^{q-1} \cdot i_1.
\]

Typically, the elements of any \( x \) will be distributed in such a way, that the \( u \) high bits of \( i_1, i_{u}, \ldots, i_u \), designate the processor number, on which \( x(i) \) is located, whereas the \( q-u \) low bits \( i_{u+1}, \ldots, i_q \) generate the local index of \( x(i) \) on processor \( ip = d(i_1, \ldots, i_u) \). This splitting of index \( i \) into processor and local part will be symbolized as

\[
x(i) = x(i_q, \ldots, i_{u+1} | i_u, \ldots, i_1)
\]

Since the FFT algorithm in each step mixes the elements differing in exact one digit, it is clear that only during the steps \( s = 1, \ldots, u \) interprocessor communication is involved. In these steps the two processors differing in bit number \( s \) have to exchange \( n/np \) complex array elements. Every processor then has to combine the elements with the trigonometric factors resulting in \( 5n/np \) real operations. The theoretical efficiency of the parallel distributed FFT is displayed in fig. 4 in sec. 6.

4. Two-dimensional formulation of one-dimensional FFT

The one-dimensional FFT accesses all \( n \) elements to be transformed in each of the \( \log_2 n \) partial transformation steps. A reformulation of the FFT as a two-dimensional transformation reduces the access to the full series. Let \( n \) be factorizable into two numbers of equal order of magnitude, \( n = n_1 n_2 \), and partition \( x \) and \( \overline{x} \) as two-dimensional arrays:

\[
x(j) = x(j_1, j_2), \quad \overline{x}(k) = \overline{x}(k_2, k_1)
\]

where \( j = j_1 + n_1 j_2, \quad k = k_2 + n_2 k_1 \)

Using the factorization properties of the exponential,

\[
W(n_j^j) = W(n_1^{j_1}) W(n_1^{j_2}) W(n_2^{j_2}),
\]

the Fourier transformation can be realized in two steps transforming the two dimensions successively:

\[
\overline{x}(k_2, k_1) = \sum_{j_1=0}^{n_1-1} W(n_1^{j_1}) W(n_1^{j_2}) \sum_{j_2=0}^{n_2-1} W(n_2^{j_2}) x(j_1, j_2).
\]

The first step involves \( n_1 \) FFT's of size \( n_2 \), the second \( n_2 \) transformations of size \( n_1 \).

If the series of sizes \( n_1, n_2 \) fit into the fast local cache, the access to the slower memory levels is greatly reduced. In order to exploit this data locality, the index to be used in the transformation should always run fastest. This is achieved in a six-pass algorithm with three transpositions involving global data rearrangements (cf. fig. 2).

One can dispose of the transposition steps at the beginning and the end of the two-dimensional formulation, if one is willing to work with data, which are not ordered in the natural ascending order. This four-pass algorithm can be of great advantage because of the reduced data exchange with remote processors on a distributed memory parallel computer. The key to a fast implementation of the six- and four-pass algorithms of course is an efficient matrix transposition.
step 1 transposition  \[ x^{(1)}(j_2, j_1) = x(j_1, j_2) \]
step 2 fast Fourier transform  \[ x^{(2)}(k_2, j_1) = \sum_{j_2=0}^{n_2-1} W(n_2)^{j_2 k_2} x^{(1)}(j_2, j_1) \]
step 3 transposition  \[ x^{(3)}(j_1, k_2) = x^{(2)}(k_2, j_1) \]
step 4 twiddle factor  \[ x^{(4)}(j_1, k_2) = W(n_1)^{j_1 k_1} x^{(3)}(j_1, k_2) \]
step 5 fast Fourier transform  \[ x^{(5)}(k_1, k_2) = \sum_{j_1=0}^{n_1-1} W(n_1)^{j_1 k_1} x^{(4)}(j_1, k_2) \]
step 6 transposition  \[ \overline{x}(k_2, k_1) = x^{(5)}(k_1, k_2) \]

Figure 2. Six-pass algorithm

5. Local matrix transpose

The transposition of a two-dimensional array \( a \) of size \( n_1 \times n_2 \) into an array \( b \) of size \( n_2 \times n_1 \), both residing in local memory, is performed by the following piece of FORTRAN code:

```fortran
    do i1 = 1, n1
        do i2 = 1, n2
            b(i2,i1) = a(i1,i2)
        end do
    end do
```

The access to array \( a \) is strided, which results in reloading of not fully used cache lines (trashing), if the leading dimension of \( a \) is too large. This effect shows up in fig. 3, where the bandwidth of memory access for transposition drops from 150 MB/s to 75 MB/s for larger sizes of the array.

The processors of the Cray T3E have special E-registers, which communicate data directly from local or remote memory to the register files of the CPU, without using the cache. By a compiler directive

```
!dir$ cache_bypass a, b
```

inserted into the FORTRAN code, the data flow avoids the cache and reaches a bandwidth of more than 250 MB/s. Since the leading dimension of the array \( a \) is a power of two, the strided access to the local memory leads to bank conflicts. These can be resolved, using special commands of the benchlib library, which allows indexed access from E-registers to the main memory [3]. Using these features gives more than 500 MB/s bandwidth for the transpose of a complex array (cf. fig. 3).

An implementation of the six-pass algorithm, using the Scilib routine `ccfft` for the two FFT passes is compared to the direct use of `ccfft`, which implements the FFT algorithm of section 2, in fig. 1 of section 3. In both cases the cache has been flushed prior to executing the transformation.
6. Global matrix transpose

For the parallelization of the two-dimensional formulation of the one-dimensional FFT it will be assumed, that \( np \leq n_1 \) and \( np \leq n_2 \), where \( np \) is the number of processors. \( n = n_1 \cdot n_2 \) is the factorization of the series’ size into the dimensions of the two-dimensional array. Under this restriction the data distribution can be chosen in such a way, that the partial FFT’s in steps 2 and 5 of the six-pass algorithm can be calculated locally without communication between different processors. The interprocessor exchange of data is restricted to the transposition steps 1, 3 and 6. For simplicity \( n \) and \( np \) are assumed to be powers of 2.

The distribution of the input series will be chosen blockwise. Let \( n = n_1 \cdot np \) and split any global index \( i \) into a local part \( i_l \) and a global part \( i_g \), which designate the processor, on which index \( i \) resides. Then

\[
x(i) = x(i_l|i_g)
\]
denotes element \( i_l \) on processor \( i_g \).

In the two-dimensional setting, the global array \( x(i_1, i_2) \) is distributed as

\[
x(i_1, i_2) = x(i_1, i_2|i_2).
\]

It follows, that in the six-pass algorithm there are four different data distributions, which are connected by three global transposition steps:
\[
\begin{align*}
x(j_1, j_2 | j_2) & \rightarrow \text{transpose} \rightarrow x^{(1)}(j_2, j_1 | j_2) \\
x^{(1)}(j_2, j_1 | j_2) & \rightarrow \text{FFT} \rightarrow x^{(2)}(k_2, j_1 | j_2) \\
x^{(2)}(k_2, j_1 | j_2) & \rightarrow \text{transpose} \rightarrow x^{(3)}(j_1, k_2 | k_2) \\
x^{(3)}(j_1, k_2 | k_2) & \rightarrow \text{twiddle} \rightarrow x^{(4)}(j_1, k_2 | k_2) \\
x^{(4)}(j_1, k_2 | k_2) & \rightarrow \text{FFT} \rightarrow x^{(5)}(k_1, k_2 | k_2) \\
x^{(5)}(k_1, k_2 | k_2) & \rightarrow \text{transpose} \rightarrow x(k_2, k_1 | k_2)
\end{align*}
\]

The parallel transposition of a \( n \times m \) matrix \( a(i, j) \) to a \( m \times n \) matrix \( b(j, i) \) on \( np \) processors,

\[
a(i, j | j_2) \rightarrow b(j, i | j_2)
\]

is performed by splitting the first index of \( a \) and the first index of \( b \) into two parts:

\[
a(i_t, i_g, j_1 | j_2) \rightarrow b(j_1, j_2 | i_t, i_g)
\]

and exchanging and transposing in \( np \) sequential steps submatrices of size \( nl \cdot ml \), in each step \( np \) submatrices in parallel:

for \( step = 0, \ldots, np - 1 \) do
  for \( ip = 0, \ldots, np - 1 \) parallel do
    \( it = ip + step \mod np \)
    for \( i_t = 0, \ldots, nl \) do
      for \( j_1 = 0, \ldots, ml \) do
        \( b(j_1, ip, i_t | it) = a(i_t, i_g, j_1 | ip) \)

The communication load for a parallel transposition of a \( n \times m \) matrix therefore is \((np - 1) \cdot (n/np) \cdot (m/ml) \approx (n \cdot m) / np \) data elements. The communication time for the parallel six-pass algorithm consequently is

\[
T_{\text{comm}} = 3 \cdot \left[ (np - 1)t_{\text{lat}} + 2 \cdot \frac{np - 1}{np} \cdot \frac{n}{np} \cdot c^{-1} \right]
\]

where \( t_{\text{lat}} \) is the latency, \( c^{-1} \) the rate for the communication.

Fig. 4 compares the theoretical efficiency for the six-pass algorithm with that of a four-pass algorithm, which poses of the initial and final transpose, and with a parallel Cooley-Tukey algorithm, which was sketched in sec. 2. The size of the transformation is \( 2^{20} \), as performance parameters a computational speed of 100 MFlops, a bandwidth of 160 MByte/s and a latency of 2 \( \mu \) have been taken. The actual performance will of course deviate from this simple picture, because the computational speed will depend on the sizes of the local arrays, memory access to local memory will vary according to the different access patterns and so on. But the general trend is clear, the two-dimensional formulation is superior in the range of 16 to 256 processors.
The actual parallel matrix transposition of a $n \times m$ array $a(i, j)$ to a $m \times n$ array $b(j, i)$ is a three step process:

$$a(i_l, i_g, j_l | j_g) \rightarrow a(j_l, i_l, i_g | j_g) \rightarrow b(j_l, j_g, i_l | i_g) \rightarrow b(j_l, j_g, i_l | i_g).$$

The first and last steps are local rearrangements of the data, which perform with several hundreds of Mbyte/s by using the E-registers (cf. fig. 3). The global reordering consists of $np$ sequential steps, in each of which $np/2$ pairs of processors simultaneously exchange blocks of $n/np \cdot m/np$ data elements. The performance of this global rearrangement is shown in figs. 5 and 6.

![Figure 5. Per processor speed of global reordering on a 40 processor T3E](image1)

![Figure 6. Per processor speed of global reordering on a 784 processor T3E](image2)
These figures show clearly, that the multiple pairwise exchange operations lead to a drop in the communication speed, because several data streams have to share the same physical network links. Also, on the system with 748 computing nodes, this effect is larger than on the smaller system with 40 nodes. On the other hand, the start up for the communication is seen to decrease with the number of participating processors. Fig. 7 shows the latency for the communication, which has been extracted from the curves of fig. 6 by fitting the communication time to the form \( t_c = t_{lat} + n \cdot c^{-1} \). The startup time shows a pipelining behaviour, because for a sequence of communication steps, the preparation for the next step can already start while data of the previous step are still flowing through the network.

The data of figs. 5, 6 and 7 describe the communication with shmem routines. It is interesting to compare this with the performance of the MPI routine allreduce, which is displayed in fig. 8. The implementation is very good, with a communication speed for large data sizes comparable to the shmem performance.

![Figure 7. Latency of global data reordering](image1)

![Figure 8. Per processor speed of global reordering with MPI allreduce](image2)

7. ccft1d, a package for one-dimensional FFT

From the discussion in the last section it is clear that the parallel efficiency for the FFT will drop more rapidly than the theoretical curves of fig. 4 suggest. This is due to the slow-down of the communication speed as observed in figs. 5 and 6 and to the fact, that the global transpose, in addition to the global reordering, needs two local rearrangement steps, which further decrease the overall time for the data movement.

Fortunately, there are many applications of the FFT, which can use the four-pass algorithm with not naturally ordered series, which has only one third of the communication overhead of the full two-dimensional formulation of the FFT.

A package for the parallel computation of the one-dimensional complex Fourier transformation on Cray T3E system named ccft1d has been developed, which implements the four-pass version. It contains the following modules:
• `ccft1d_i`: computes the two-dimensional factorization and precalculates the trigonometric function values for the twiddle factors.

• `ccft1d_x`: calculates the four-pass algorithm with input and output series not naturally ordered.

• `ccft1d_r`: performs the reordering of the series before or after the four-pass algorithm.

• `ccft1d`: calculates the full six-pass algorithm with naturally ordered input and output series.

A typically application of the four-pass algorithm would be the iteration of a vector $a$ by the product of two operators, one of which is diagonal in the original representation, the other in the Fourier transformed representation. Let $x$ and $p$ be these operators in their diagonal forms. The program using the four-pass scheme than looks like:

c initialize start vector $a$, operators $x$ and $p$ in their natural order and distribute them blockwise across the processors
c initialize the FFT algorithm
call ccft1d_i( n, tables, ntables, nwork )
c transform $a$, $x$, and $p$ into the order needed for the four-pass algorithm
call ccft1d_r( +1, a, n, work, nwork )
call ccft1d_r( +1, x, n, work, nwork )
call ccft1d_r( -1, p, n, work, nwork )
c iterate the vector $a$
do irep = 1, lrep
c operation in direct representation
$ a = x \ast a$
c FFT of $a$ (four-pass version)
call ccft1d_x( +1, 1., n, a, tables, work )
c operation in transformed representation
$ a = p \ast a$
c inverse FFT (four-pass version)
call ccft1d_x( -1, 1./real(n), n, a, tables, work )
end do
c rearrange $a$ to natural order
call ccft1d_r( -1, a, n, work, nwork )

The performance of a real application according to this scheme is displayed in fig. 9. Performance drops for data sizes larger than $2^{20}$, because then the local FFT's can no longer be calculated in cache. For applications of size larger than $2^{20}$, one has to generalize the two-dimensional to a three- or even higher-dimensional scheme (cf. [2]).

Finally fig. 10 shows the performance of `ccft1d_x` for a single data size, $n = 2^{20}$, versus the processor number. The drop in performance follows qualitatively the behaviour
expected from fig. 4. Notice, that for 512 processors only two complex numbers are exchanged between processor pairs during the reordering steps.

The code for ccftid is freely available. Please contact the author in order to obtain the package (E-mail ohaan@gwdg.de).

REFERENCES

8. P.N. Swarztrauber, Multiprocessor FFTs, Parallel Computing 5 (1987) 197-210
Parallel Iterative Methods for Solving Sparse Linear Systems from Thermalhydraulic Codes Using HPF Programming on Cray T3E

C. Gaudart  
DRN/DTP/SMTH  
CEA - Grenoble  
gaudart@alpes.cea.fr

Ph. Michallon  
CEA/DM - CISI  
CEA - Saclay  
pmichallon@cea.fr

Abstract: the next generation of thermalhydraulic codes will be parallel codes. But before, developing new codes, programmers need to know parallel numerical algorithms, their properties, the best way to implement them and their efficiency. This paper presents parallel methods for solving sparse linear systems using HPF or MPI, and results on Cray T3E for large systems.

1. Introduction

1.1 Main objectives

Parallel programming is more difficult, programmers have to deal with a large number of choices involved by the diversity and the perpetual evolution of methods and parallel architecture.

This study proposes to compare, in an industrial context, several methods of parallelization. This work has been done at the CEA-Grenoble in the department SMTH and in collaboration with the Laboratory of Electricité de France LNH. The study is based on several scientific industrial software, which simulate a large diversity of thermalhydraulic phenomena overlaying a huge physical domain on fluid mechanics. This great variety of applications lead to an heterogeneous set of codes developed with different choices: languages, numerical methods, preferential parallel machines. Due to specifics requests of a client or induced by the applications processed, the strategy of the parallel development may be very different from a code to another. It can be involved by the necessity of simulate more and more fine applications but also for the use of tridimensional networking.

One of the difficulties when you chooses a parallelization technique is that one does not know, before the development of the application, the solutions proposed by the different methods of parallelization report to the expressed requirements. These requirements can be performed as fixed constraints that developments have to respect. They also defined the criteria for the comparison of the different parallelization methods. These criteria are: evolution possibilities of the parallel developments, respect of the physical and numerical validation fixed, portability of the application on different machines for exploitation and performances.

For different parallelization methods, it was not possible to envisage a study on each whole code. In order to solve the given problem, the linear solver part of codes has been chosen. It’s a common part of all codes in fluid mechanics, always present in this kind of simulations. Moreover, the linear solver is representative of the numerical choices (like networking, space and time discretizations...) and of important functions of a scientific software. We can find in the parallelization of this part, lot of the difficulties encountered in parallel developments. At the end, for the main applications, it is the more expensive part of the software.

The work on the linear solver part of codes allows to have a good vision of the parallelization difficulties, to extrapolate for all codes some results and to propose some first solutions for the optimizations of scientific software.

1.2 Organization of the paper

In this paper we describe the approach for parallizing sparse linear systems using data-parallelism and message passing. Although, this study is not yet finished, we present performances results obtained for solving large 3D Laplacian on Cray T3E. In the first part, the methods we have used for solving sparse linear systems are presented (preconditioners, conjugate gradient, sparse matrix storage formats). In the second one, a first approach for parallizing sparse solvers with HPF is given. The second approach based on MPI is described in the next part. The fourth part presents an optimization of the solver using message passing techniques. Finally, some experimental results on Cray T3E are presented.
2. Solve sparse linear system

A linear system of equations can be written as a matrix equation $Ax = b$ where $A$ is an $n \times n$ matrix, $b$ and $x$ are vectors of size $n$. We have to find values for $x$ that satisfy this equation when $A$ is sparse and non-singular.

Iterative methods are the most popular way to solve this problem. They use successive approximations to obtain more accurate solutions to a linear system at each step. There exists 2 classes of iterative methods: stationary and nonstationary [Bar94][Saad96]. Stationary methods like Gauss-Seidel, SOR..., can easily be implemented but usually they are not efficient. Nonstationary methods are more complicated. They are based on the principle of sequences of orthogonal vectors (Krylov methods)[Fre91] .

Performances of these methods depend on the number of iterations. A good way to reduce this number is to use a preconitioner for matrix with bad spectral properties.

2.1 Preconditioners

A precondioner allows to transform the linear system into one that is equivalent in the sense that it has the same solution but with more favorable spectral properties for the new matrix. The new matrix is $M = M b$. $M$ must be chosen, for having a new system easier to solve than the first.

Two preconditioners are used by us: Neumann because it is easy to parallelize, SSOR (Symmetric Successive Over Relaxation) because it is used in TROIVF and N3S codes.

**Neumann preconditioning**

With this preconditioning the matrix $M$ is constructed in the form $M = P(A)$ where $P$ is an approximation derived from a truncation of the Neumann series [Saad96].

$$M = (I + N + ... + N^k)D^{-1}$$

where $N = I - \omega D^{-1}A$, $D$ is a diagonal matrix with the main diagonal of $A$ and $\omega$ a weighting factor. This method is easy to implement because it consists only in a matrix-vector product, but it is not very efficient.

**SSOR preconditioning**

It is based on the fact that a symmetric matrix can be decomposed as $A = D + L - L'$ where $D$ is a diagonal matrix with the main diagonal of $A$, $L$ is the lower part of $A$. After some computations [Gol89], one can prove that $M$ can be done by:

$$M = \frac{I}{\omega(2-\omega)} \times (D - \omega L) \times D^{-1} \times (D - \omega L')$$

A SSOR preconditioning can be solved, for $My = z$ by:

- solving $(D - \omega L)y = z$ for lower triangular system (forward substitution)
- solving $\frac{I}{\omega(2-\omega)} D^{-1}y = y$ for upper triangular system (backward substitution)

Remark: for an optimal value of $\omega$ the number of iterations is minimal, but this value is prohibitive to compute.

2.2 Resolution methods

For solving the system $Ax = b$ nonstationary methods are used (Conjugate Gradient, GMRES,...), based on projections on Krylov subspace [Bar94][Gau98][Saad96].

In the following we restrict our study to the most popular method which is the Conjugate Gradient method [Hes52]. This method applies only for symmetric positive definite systems. We can find it into 2 thermalhydraulic codes (TROIVF, N3S).

Eisenstat algorithm for preconditioned conjugate Gradient method is [Eis81]:
choose \( x_0 \)

compute \( r_0 = b - Ax_0 \); \( z_0 = Mr_0 \); \( p_0 = z_0 \)

do \( k = 1, 2, 3, \ldots \) until convergence

\[
\begin{align*}
\alpha_k &= \frac{(r_k \cdot z_k)}{(Ap_k \cdot p_k)} & \text{! matrix–vector product and scalar products} \\
\alpha_k &= (\cdot) & \text{! axpy} \\
x_{k+1} &= x_k + \alpha_k p_k & \text{! axpy} \\
r_{k+1} &= r_k - \alpha_k Ap_k & \text{! axpy} \\
z_{k+1} &= Mr_{k+1} & \text{! matrix–vector product} \\
\beta_k &= \frac{(z_{k+1} \cdot r_{k+1})}{(z_k \cdot r_k)} & \text{! scalar product} \\
p_{k+1} &= z_{k+1} + \beta_k p_k & \text{! axpy}
\end{align*}
\]

end do

The basic numerical kernels, which are involved in this algorithm, are: matrix-vector product, scalar product (dot product), and axpy. Conjugate gradient performances depend on the parallelization of the kernels, and their optimization, like we can see later.

At each step, the preconditioning is made by the matrix vector product \( z = Mr \).

In this algorithm the matrix \( A \) is stored as a dense matrix which needs \( n^2 \) data. But thermalhydraulic systems are sparse and the size is about 1,000,000, so a sparse matrix storage is needed if we do not want to lose memory space and to do useless computations.

### 2.3 Matrix storage formats

This section presents the storage MSR (Modified Sparse Row) used in the following. All others storage formats can be found in [Bar94][Saa94].

MSR is an adaptation of CSR (Compressed Sparse Row) storage. For the CSR storage 3 vectors are needed:

- vector \( a \) which contains the non zero elements of the matrix;
- vector \( ja \) which contains the column indexes of the elements in the vector \( a \);
- vector \( ai \) which stores the location of the beginning of each row in the \( a \) vector. \( ai(k+1) - ai(k) \) gives the number of non-zero elements in row \( k \). \( a(n+1) \) gives the number of non-zero elements in \( A \).

The following example illustrates it:

\[
\begin{pmatrix}
1 & -1 & 2 & 0 & 0 & 3 \\
0 & 2 & 1 & 0 & -1 & 0 \\
4 & 0 & 3 & 0 & 0 & 0 \\
0 & -2 & 4 & 1 & 0 & 0 \\
3 & 0 & 0 & 5 & -1 & 0 \\
0 & 0 & 0 & 0 & -2 & 6 \\
\end{pmatrix}
\]

\[ia = (1, 5, 8, 10, 13, 16, 18)\]
\[a = (1, -1, 2, 3, 2, 1, -1, 4, 3, -2, 4, 1, 3, 5, -1, -2, 6)\]
\[ja = (1, 2, 3, 6, 2, 3, 5, 1, 3, 2, 4, 5, 1, 5, 6, 5, 6)\]

For MSR, we store the main diagonal of \( A \) in the first elements of \( a \). This allows to store the matrix with only 2 vectors, that gives an easier access to matrix elements:

- vector \( a \) which has the \( n \) elements of the main diagonal of \( A \), then the \( n+1^{\text{th}} \) element is a bad element and the following values are the other elements of \( A \).
• vector \( ja \) where the \( n+1 \) first elements are the same as in \( ia \) and the count of the others elements of \( a \) starts at the \( n+2^{\text{th}} \) element.

With the above matrix, the MSR storage is:

\[
\begin{align*}
a &= (1, 2, 3, 4, 5, 6, 0, -1, 2, 3, 1, -1, 4, -2, 1, 3, -1, -2) \\
ja &= (8, 11, 13, 14, 16, 18, 19, 2, 3, 6, 3, 5, 1, 2, 5, 1, 6, 5)
\end{align*}
\]

The main consequence of this storage on the conjugate gradient is on the matrix-vector product. This multiplication can be done by the following algorithm [Gau98]:

\[
\begin{align*}
\text{DO } i &= 1, n \\
  && Y(i) = a(i) \times x(i) \\
\text{END DO} \\
\text{DO } i &= 1, n \\
  && k = ja(i), ja(i+1)-1 \\
  && y(i) = y(i) + a(k) \times x(ja(k)) \\
\text{END DO}
\end{align*}
\]

We have presented techniques for solving sparse linear systems, now we will present to you, how to implement them using HPF and message passing programming.

3. HPF implementation

HPF [Hpf93] [Koe94] is a parallel extension to Fortran 90. It is based on the mapping of data-object and instructions onto parallel architectures with compiler directives.

Data can be distributed by BLOCK or CYCLIC way. In the following we consider only BLOCK distribution. CYCLIC distribution is not implemented, but it will be done in a near future.

A natural block distribution is to distribute \( n/p \) rows by processors, what we can do by \textsc{Distribute A(BLOCK)}. We consider \( n/p \) as an integer for the sake of simplicity, but HPF can easily manage the case where \( n/p \) is not an integer.

This distribution can be easily used with structured or dense matrix, but HPF does not allow distribution for sparse matrix storage like CRS or MSR. For example if we have a sparse matrix like this:

\[
A = \begin{bmatrix}
1 & 5 & 0 & 0 & 3 & 1 \\
0 & 2 & 1 & 4 & 0 & 0 \\
1 & 0 & 3 & 0 & 5 & 6 \\
0 & 0 & 0 & 4 & 7 & 8 \\
9 & 0 & 1 & 0 & 5 & 0 \\
0 & 3 & 0 & 2 & 0 & 6
\end{bmatrix}
\]

MSR storage is given by:

\[
a = (1, 2, 3, 4, 5, 6, 0, 5, 3, 1, 4, 1, 5, 6, 7, 8, 9, 1, 3, 2) \text{ et} \\
ja = (8, 11, 13, 16, 18, 20, 22, 25, 6, 3, 4, 1, 5, 6, 5, 6, 1, 3, 2, 4).
\]

Using the distribution (BLOCK,*), programmer would like to have in each processor \( n/p \) rows of the matrix, and the \( n/p \) corresponding elements of vector \( x \), in order to compute the local matrix-vector product efficiently. On the example, if processor Pe1 owns the two first rows, local arrays would have to be: \( a = (1, 2, 0, 5, 3, 1, 4) \) and \( ja = (4, 7, 9, 2, 5, 6, 3, 4) \). Like this, the local data access is the same that the global access. But the HPF distribution does not permit this. Arrays \( a \) and \( ja \) are distributed without keeping the storage coherency. Always with the same example, with the distribution (BLOCK,* on 3 processors: processor Pe1 owns \( a = (1, 2, 3, 4, 5, 6) \) and \( ja = (8, 11, 13, 16, 18, 20, 22) \); processor Pe2 owns \( a = (5, 3, 1, 4, 1, 5) \) and \( ja = (2, 5, 6, 3, 4, 1, 5) \); processor Pe3 \( a = (6, 7, 8, 9, 1, 3, 2) \) and \( ja = (6, 5, 6, 1, 3, 2, 4) \). Moreover, with this distribution it is not possible to align the matrix with the vector for the matrix-vector product, so there will be many communications and performances will decrease.

So, for having a real block distribution with the MSR storage, some modifications must be realized on the way to storage information. The idea is to transform \( a \) and \( ja \) in two dimensional arrays of size (number of rows, maximal number of non-zero
elements by row). In a we have always the non-zero elements, with a specific storage for diagonal elements, so in ja we have only column indices.

- \( a(i,:) \) = non zero elements on row \( i \), with in the first position the diagonal element. If the number of elements of this row is less than the maximal number of non-zero elements by rows, it is filled by 0 value.
- \( ja(i,:) \) = column indices in the matrix \( A \) of the \( a \) elements. If the \( a \) element is equal to 0, the column indice is substituted by the size of the matrix \( A \).

The new \( a \) and \( ja \) obtained are:

\[
\begin{bmatrix}
1 & 5 & 3 & 1 \\
2 & 1 & 4 & 0 \\
3 & 1 & 5 & 6 \\
4 & 7 & 8 & 0 \\
5 & 9 & 1 & 0 \\
6 & 3 & 2 & 0
\end{bmatrix}
\quad \text{et} \quad
\begin{bmatrix}
1 & 2 & 5 & 6 \\
2 & 3 & 4 & 6 \\
3 & 1 & 5 & 6 \\
4 & 5 & 6 & 6 \\
5 & 1 & 3 & 6 \\
6 & 2 & 4 & 6
\end{bmatrix}
\]

Now, with a BLOCK distribution on the first dimension on \( a \) and \( ja \), it is possible to obtain the wanted distribution. Moreover, it is now possible to align the vector distribution with this of the matrix. In other side, this new storage uses supplementary elements, which could give useless computations.

The new matrix-vector algorithm is:

\[
\text{DO } i = 1, n \\
y(i) = a(i,1)x(i) \\
\text{ENDDO}
\]

(1)

\[
\text{DO } i = 1, n \\
\text{DO } k = 2, m_n \\
y(i) = y(i) + a(i,k)x(ja(i,k)) \\
\text{ENDDO}
\]

(2)

ENDDO

Instruction (2) uses an indirection that can give bad performance with many HPF compilers. For helping the compiler, we introduce the new variable \( t \):

\[
\text{DO } i = 1, n \\
\text{DO } k = 2, m_n \\
t = ja(i,k) \\
y(i) = y(i) + a(i,k)x(t) \\
\text{ENDDO}
\]

(2)

ENDDO

This matrix-vector product is not optimized, because many useless communications and computations are made. In the goal to reduce its, we look a message passing implementation.

4. MPI implementation

With the message passing approach programmers must explicitly describe data distribution and communications between processors. The distribution choice is not easy. It depends on many parameters like computer performances (latency, bandwidth) or like application characteristics (the size of messages or their number, or the tools used for the domain decomposition). But in the following, we keep the distribution used with HPF.

In order to achieve the portability and the best performance on Cray T3E, we chose MPI [Sni95] as message passing library.

Using the block distribution each processor owns \( n/p \) consecutive rows of matrix \( A \) and \( n/p \) consecutive elements of vectors \( x \) and \( b \) (where \( n \) is the number of rows of matrix \( A \) and the size of vectors \( x \) and \( b \), and where \( p \) is the number of processors used). To simplify the presentation we will consider that \( n/p \) is an integer, but programmers have to plan the distribution when \( n/p \) is not an integer.

The aspxy kernel do not need communications and the scalar product can be done efficiently by a collective communication (MPI_Allgather), but the sparse matrix-vector product (\( z = Mr \) can not be computed so easily.
Computing each component of vector $z$ requires the knowledge of the whole vector $r$; so each processor must send its $n/p$ components of vector $x$ to all the other processors. The data movement is a total exchange (MPI_ALLGATHER).

If this parallel matrix-vector algorithm is efficient for dense matrix, it is totally inefficient for sparse matrix (too many useless communications). For minimizing communications, an inspector/executor mechanism [Das92] [Uja93] must be used. The inspector function gives the communication matrix corresponding to the matrix-vector product, and the function executor realizes only the communications.

The inspector function is called only one time just after the data distribution phase. The iterative method does not use this function. The executor function is called before every matrix-vector product, then the product can be done like in the sequential programming model.

The following example shows how the inspector function works:

We consider a 6 by 6 matrix, distributed by blocks onto 3 processors (Pe0 owns row 1, 2; Pe1 owns row 3, 4; Pe2 owns row 5, 6). Vectors $r$ and $z$ are distributed on the same way, and we compute $z = Mr$. $m[k]$ represents the $k$th non-zero element of $M$ using CSR data storage.

$$M = \begin{bmatrix}
\alpha & 0 & 0 & 0 & 0 & 0 \\
0 & \alpha & 0 & 0 & 0 & 0 \\
\alpha & 0 & \alpha & 0 & 0 & 0 \\
0 & 0 & 0 & \alpha & 0 & 0 \\
0 & 0 & \alpha & 0 & \alpha & 0 \\
0 & 0 & 0 & \alpha & 0 & \alpha

Processor Pe0 needs $r[2]$ from processor Pe1 and $r[5]$ from processor Pe2. Pe0 sends a request to processors Pe1 and Pe2 which send back only $r[2]$ and $r[5]$. Pe1 and Pe2 act in the same way. So the communication matrix obtained by the inspector function is:

<table>
<thead>
<tr>
<th>receiver</th>
<th>Pe0</th>
<th>Pe1</th>
<th>Pe2</th>
</tr>
</thead>
<tbody>
<tr>
<td>sender</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Pe0</td>
<td>$r[0]$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Pe1</td>
<td>$r[2]$</td>
<td>$r[3]$</td>
<td></td>
</tr>
</tbody>
</table>

The executor function is called before each matrix vector, so each processor sends to other only data that they need. In example below, if data change at each iteration, the communication matrix does not change. At each step of the iterative method processor Pe1 sends to Pe0 $r[2]$ and processor Pe0 sends to Pe0 $r[5]$, …

The following table illustrates the computation times obtained without and with inspector/executor. These results are obtained with a symmetric diagonal matrix. This matrix is given by a Laplacian 3D with structured meshing with finite difference.

<table>
<thead>
<tr>
<th>Processors number</th>
<th>Matrix-vector product without inspector/executor</th>
<th>Matrix-vector product with inspector/executor</th>
</tr>
</thead>
<tbody>
<tr>
<td>20</td>
<td>13.15 sec.</td>
<td>11.34 sec.</td>
</tr>
<tr>
<td>100</td>
<td>13.77 sec.</td>
<td>3.67 sec.</td>
</tr>
</tbody>
</table>

Time of 100 matrix-vector product of size 1 million on Cray T3E

As one can observe, we obtain a good decreasing of execution times. We can also see a great influence of the number of processors.

In order to reduce the number of communications and to have best performances, another optimization can be realized. In fact, SSOR preconditioner requires to solve 2 triangular systems, which can not be parallelized efficiently.
A good way to parallelize it, consists in using block preconditioning. The matrix is written as the sum of a block diagonal matrix $A$ and a matrix $R$.

\[
\begin{bmatrix}
  x & x & 0 & 0 & x & 0 \\
  x & x & 0 & 0 & x & x \\
  0 & x & x & 0 & 0 & x \\
  0 & 0 & x & x & 0 & 0 \\
  x & 0 & 0 & x & x & 0 \\
  x & x & 0 & 0 & x & x \\
  0 & 0 & 0 & 0 & 0 & x \\
  0 & 0 & 0 & 0 & 0 & x \\
  0 & 0 & 0 & 0 & 0 & x \\
  0 & 0 & 0 & 0 & 0 & x \\
\end{bmatrix}
\begin{bmatrix}
  x & x & 0 & 0 & 0 & 0 \\
  x & x & 0 & 0 & 0 & 0 \\
  0 & x & x & 0 & 0 & 0 \\
  0 & x & x & 0 & 0 & 0 \\
  x & 0 & 0 & x & 0 & 0 \\
  x & 0 & 0 & 0 & 0 & 0 \\
  0 & 0 & 0 & 0 & 0 & 0 \\
  0 & 0 & 0 & 0 & 0 & 0 \\
  0 & 0 & 0 & 0 & 0 & 0 \\
  0 & 0 & 0 & 0 & 0 & 0 \\
\end{bmatrix}
\begin{bmatrix}
  x & x & 0 & 0 & 0 & 0 \\
  0 & 0 & x & 0 & 0 & 0 \\
  0 & 0 & 0 & 0 & x & 0 \\
  0 & 0 & 0 & 0 & 0 & x \\
\end{bmatrix}
\]

The preconditioner is then applied to each block of $\tilde{A}$ that allows a parallel preconditioning without communication. This method is not robust but significantly decreases the number of iterations, as showed in the experimental part.

5. **HPF optimization**

The first optimization is to introduce inspector/executor method. Each processor must be able to take back only elements which are needed for the computation. So, a processor has to know on which processor are the data and which are their indices on it. The only way to do this it is to use an extrinsic **HPF_LOCAL** routine. In such a routine, data become private on each processor: A processor can not access data in other processors, but it can know with its local indices, the global indices of its data using the HPF routine **LOCAL_TO_GLOBAL()**. Knowing the global indices of an element, HPF specifications give the owner processor, and which are the local indices of the element on this processor. For example:

- element of global position $j$ is on the processor $\left\lceil \frac{j}{p} \right\rceil$, where $p$ is the number of processors.
- Its location is given by $\left\lceil \frac{j \mod p}{p} \right\rceil$, if $j \mod p \neq 0$.

With these informations, a processor can access in a **HPF_LOCAL** routine to the data of another processor with message passing calls. As in the message passing approach this strategy reduces the number of messages exchanged.

Mixed HPF program with **HPF_LOCAL** routing using MPI is very useful for irregular data movement, or when the compiler can not extract communications. CEA used this mixed on several applications.

The second optimization is to use block diagonal preconditioning. So, each processor must only do the computation with its local data. As each processor owns a block of the vector, the computation is done only with the corresponding components. For example, processors will compute only with elements in grey.

\[
\begin{array}{c}
\text{Pe1} \\
\text{Pe2} \\
\text{Pe3} \\
\end{array}
\begin{array}{ccc}
1 & & \\
2 & & \\
3 & & \\
\end{array}
\]

The solution for doing this is to use an extrinsic **HPF_LOCAL** routine, in which we use Fortran 90 intrinsic functions like UBOUND(), LBOUND() and a HPOF routine which is **LOCAL_TO_GLOBAL()**. With these informations, the local preconditioning can be done without communication.

6. **Experimental results on Cray T3E**

The test codes are done using a kernel that was developed at CEA/DRE. This kernel solves in parallel a 3D Laplacian equation with Dirichlet Boundary (Boundary and domain are the same as in the industrial case).

We consider a domain composed of $n x n x n$ elements and we choose a value of 1.65 for $\omega$ for the SSOR algorithm. On the Cray T3E we use the Portland Group compiler pg4f7 Ver. 2.4, the MPT Ver.1.2.0.1 and the F90 Ver. 3.0.2.0. Compiler options are -O3 -pieLine -O3 -Mautopar for pg4f7.
For 125,000 elements \((nx=ny=nz=50)\), we compare results obtained with sequential, MPI and HPF implementations of the kernel.

The following figure represents performances of the conjugate gradient with SSOR preconditioning for different numbers of processors. The 4 curves are:

- **MPI inspector/executor** is an MPI implementation with inspector/executor method and diagonal blocks preconditioning;
- **HPF** is a HPF implementation without optimization;
- **HPF local blocks** is a HPF implementation with diagonal local blocks preconditioning;
- **HPF optimized** is a HPF implementation with inspector/executor method and diagonal blocks preconditioning.

![Conjugate gradient with SSOR preconditioning](image)

As we can see, HPF code has bad performances, due to many communications. The best performance obtained with this code is for 8 processors. This result is 5 times higher than the other versions. We can remark that for 2 to 8 processors HPF local blocks is better than MPI and HPF optimized versions. This can be explained by the fact that for a small number of processors there is few communications. For this small number of processors the cost of the matrix computation can not be compensated by the computation. Beyond 8 processors, MPI and HPF optimized have the best performances.

We can remark that MPI and HPF optimized codes have the same performances. So, it is possible to have good performance with HPF replacing the expensive communications generated by the compiler by MPI communications on **HPF_LOCAL** routines.

Remark: On the same size, with the Neumann preconditioner we obtain the best execution time in 22 seconds on 24 processors, which is no best than SSOR preconditioner.

As we have mentioned earlier, the diagonal blocks preconditioning reduces the number of iterations as we can see on the following figure:

![Impact of the preconditioning](image)
The number of iterations is very reduced for a small number of processors, because in this case computation results are less disturbed by the computation order, which is not true for a full SSOR preconditioning, and a bigger number of processors.

For thermal-hydraulic code a good simulation needs several millions of elements. Today we use only 1,000,000 elements. The following table shows that results obtained on smallest systems are always true for large systems:

<table>
<thead>
<tr>
<th>Number of processors</th>
<th>MPI insp/exec</th>
<th>HPF optimized</th>
<th>HPF</th>
</tr>
</thead>
<tbody>
<tr>
<td>16</td>
<td>61</td>
<td>65</td>
<td>128</td>
</tr>
<tr>
<td>32</td>
<td>39</td>
<td>41</td>
<td>223</td>
</tr>
<tr>
<td>64</td>
<td>24</td>
<td>26</td>
<td>496</td>
</tr>
</tbody>
</table>

7. Conclusion
In this paper, we have described several ways to solve parallel sparse linear systems. We have also shown that an efficient parallelization can be realized on the main kernels of thermal-hydraulic codes.

We focused on the HPF and MPI implementation. For HPF we made modifications of the MSR storage, in order to keep a row distribution and for MPI, we used an inspector/executor method for decreasing the number of communications. Then we included this method in the HPF code, using extrinsic HPF_LOCAL routine with MPI calls. Results obtained on the Cray T3E, showed the benefit of the optimization on MPI and HPF code.

This work is the beginning of a largest study, where we have to valid other methods, others kernel of industrial codes. With this knowledge, we will be able to start the development of a new generation of thermal-hydraulic codes.

References


[Man77] T. Manteuffel – An Incomplete Factorization Technique for Positive Definite Linear System – Mathematics of Computation, number 34, 1977


A portable and easy-to-use Fortran 90
MPP Environment

Renate Dohmen, Rechenzentrum Garching
Friedrich Hertweck, Bereich Informatik
Maria-Grazia Pacco-Düchs, Bereich Informatik
Max Planck Institut für Plasmaphysik IPP, Garching
EURATOM Association

Executing Fortran 90 programs on a massively parallel computer requires quite a bit of extra work to adapt to the MPP environment. We present a Fortran 90 module \texttt{MPP\_Environment}, containing constants, data structures, and procedures that forms an intermediate layer between the standard message passing libraries usually available on MPP Systems and an application program. A small set of frequently needed procedures is built on top of the many powerful, but also complex, procedures of the message passing libraries. By making use of the modern Fortran 90 techniques, the new procedures have a very simple and easy-to-use calling syntax with only few, partially optional, parameters. As a result, the writing of parallel programs will be more straightforward, simpler, and safer. In addition to universal communication and collective functions, the module also helps to support the parallel implementation of finite-difference schemes. The methods presented apply in principle to all MPP Systems. There are two versions of the module: an MPI version with the advantage of portability, and a T3E version based on the \texttt{shm}em library, exhibiting better performance. \texttt{MPP\_Environment} has been tested on a Cray T3E and on an IBM SP2 with several applications and proven to be both useful and user-friendly.

0 Introduction

With the advent of massively parallel computer systems (MPP systems) a whole series of programming issues arose: Data decomposition and spreading of work across processors, access to data on other processors, synchronization of the processors, the use of collective operations (like a global sum over an array spread across processors), or load balancing. Like other supercomputers, the MPP systems

© 1998, R Dohmen, F Hertweck, M-G Pacco-Düchs Informatik, MPI Plasmaphysik
are also mostly programmed in Fortran, with Fortran 90 now rapidly replacing the older Fortran 77. Part of the problems of parallel programming has been addressed by the further development of Fortran 90 into Fortran 95 and HPF (High Performance Fortran), and part is being solved by parallel programming libraries and tools. While some of the libraries are vendor proprietary, some standards like PVM (Parallel Virtual Machine) and MPI (Message Passing Interface) emerged. Nevertheless, programs to run on a massively parallel computer require quite a bit of adaptation.

In this paper we present a Fortran 90 module, containing constants, data structures, and procedures that will help the user to run Fortran 90 programs in the MPP environment. As a result, the writing of parallel programs will be shown to be more straightforward, simpler, and safer. While the methods developed apply in principle to all MPP Systems, we describe them in terms of the Cray T3E environment.

In this paper the term MPP Environment refers to the generic programming model we develop for any MPP system (as explicit examples we use the Cray T3E and the IBM SP2). It is implemented in two versions: as module T3E_ENVIRONMENT for the T3E using the shmem Parallel Programming Library, and as module MPI_ENVIRONMENT for any MPP System with an MPI library.

Note that at present the Fortran 90 Compiler for the T3E has certain deficiencies regarding parallel processing which we will describe later.

1 MPP Programming Issues

In this section we discuss MPP programming issues. These include setting up the basic MPP environment, structuring the actual PE set (Processing Element set) into a 2- or 3-dimensional processor array, etc. As a general programming model we use the SPMD model (Single Program, Multiple Data), [1].

We also assume scalable programs, i.e. programs that may run on any number of processors, usually with identical results. The latter is neither trivial nor always possible to achieve. As an example we may consider a Monte Carlo Program: typically the sequence of pseudo random numbers generated are dependent on the number of processors we use - the best we can do is to ensure that the sequences are disjoint between processors.

1.1 Basic Parameters and Procedures

There are a few important items a program needs to know: \texttt{npe}, the number of processing elements the job is running on, and \texttt{pid}, the number of the processor on which a particular instance of the SPMD program is running. These values are obtained at run-time by invoking, at the beginning of the main program, the subroutine \texttt{init\_MPP\_Environment}.

Another important feature to have is an accurate timer. The way times are measured vary from system to system. A frequently used convention is to measure time in elapsed microseconds. We shall use

© 1998, R. Dohmen, F. Hertweck, M-G. Pacco-Düchs Informatik, MPI Plasmaphysik
a function `clock()` which returns (as a real number) a time measured in microseconds, starting at an arbitrary time. Therefore only differences of `clock()` are meaningful.

For the Cray T3E, `clock()` is implemented by using `irtc()`, the real time clock which returns a multiple of elapsed machine cycles which is multiplied by the clock cycle_time.

### 1.2 Synchronization

Synchronization in our MPP Environment adheres to the SPMD model: all npe processors take part in the synchronization. Therefore one may simply use

```
call barrier()
```

to do the synchronization.

### 1.3 Collective Functions

These functions serve to compute global sums, maxima, minima, etc. A global operation collects the operands from the npe individual processors of the MPP system. Using for example the Cray T3E shmem-Library to sum a variable \( x \), one has to write:

```
call shmem_real8_sum_to_all(sumx,x,n,0,0,npe,pWrk,pSync)
```

where the source \( x \) and the destination \( sumx \) are required to be in a common area. (It is implicitly assumed that there is an \( x \) and a \( sumx \) on each of the npe processors. Note also that one has to specify a host of other parameters, including the pWrk and pSync arrays.

The corresponding example in MPI would be:

```
call MPI_ALLREDUCE(x,sumx,n,MPI_REAL,MPI_SUM,0,comm,ierr)
```

Also in this example there are many arguments in the call. A programmer would always need to consult a manual because it is unlikely that all the details can be remembered.

In order to make the whole procedure simple and safe, we use the following expression

```
sumx = global_sum(x)
```

which will produce the desired result. The variable \( x \) may be real or integer, scalar or array, and need not be in COMMON. The variable \( sumx \) must of course be conformable with \( x \), i.e. be of the same `rank` and `extent` as \( x \). The `types` and `kinds` may be different, as long as automatic conversion is guaranteed by the Fortran 90 language.

We believe that the negligible amount of CPU time spent for safety and simplicity is very much worth while.
1.4 Random Numbers

In Monte Carlo programs, random numbers play an important role. When testing such programs on a parallel computer, the problem arises of reproducibility of the (pseudo) random numbers. Normally, we have to ensure that each processor by itself produces a good sequence of random numbers, and that these sequences are disjoint. Then the results of a Monte Carlo program, run on different sets of processors, are not predictable. This makes program development more difficult.

In order to alleviate this problem, the random number generator of Cray Research, \texttt{ranf()}, a simple linear congruent generator which is initialized with \texttt{raset(seed)}, has been rewritten. The user should invoke instead the function \texttt{random(seed)}, which with the same \texttt{seed} returns the same random numbers as \texttt{ranf()}. But now the variable \texttt{seed} is part of the program and not part of the Fortran run-time environment. In effect, the user may use as many random number generators as desired, by just having that many variables \texttt{seed}. The additional function \texttt{skip_seed(seed,m)} can be used to advance \texttt{seed}, i.e. the random number to be returned, by \(2^m\). The Cray random number generator can produce a sequence of \(2^{46}\) random numbers before repeating itself.

Though we have used the Cray random number generator as the basis for our implementation of \texttt{random(seed)}, also a portable generator may be used, especially one that may exploit the common 64-bit architecture of many present day RISC processors.

There is a serious pitfall for the unaware user: in the Cray Fortran 90 run-time library, the Fortran 90 random number generator, subroutine \texttt{random_number(...)}, is initialized to the same seed on each processor. In other words, all the npe processors produce exactly the same random numbers. We change this undesirable fact when calling \texttt{init_MPP_Environment} by adding on each processor \(2^\text{pi}d\) to the default random number seed, obtained by calling \texttt{random_seed(get=...)}, and then restoring the seed by calling \texttt{random_seed(put=...)}. Then each processor will generate a different sequence of random numbers.

1.5 Support for Finite Difference Schemes

A group of procedures has been designed to support the execution of partial differential equations. These procedures are used to set up an appropriate array of processors, to accomplish domain decomposition, and to exchange halo boundaries between processors assigned to adjacent subdomains.

1.5.1 Setting up the Topology of a Processor Array

Often it is necessary to adapt the processor array to a 2- or 3-dimensional problem. For this purpose the procedures \texttt{PE_d2_array} or \texttt{PE_d3_array} can be called. They construct a rectangle or cuboid of npe processors, respectively:

\[
npe = \text{np}_x \times \text{np}_y \quad \text{or} \quad npe = \text{np}_x \times \text{np}_y \times \text{np}_z
\]

© 1998, R Dohmen, F Hertweck, M-G Pacco-Düchs Informatik, MPI Plasmaphysik
Each processor \texttt{pid} can then be described by two or three indices, \texttt{px, py, pz}. All these variables are defined in the module \texttt{MPP\_Environment}.

The default values, valid when the above mentioned procedures have not been called, are

\[
\texttt{npx=npe, npy=1, npz=1, \ and \ px=pid, py=0, pz=0}
\]

The procedures \texttt{PE\_d2\_array} and \texttt{PE\_d3\_array} also allocate for the user the arrays \texttt{pp(0:npx-1,0:npy-1)} and/or \texttt{ppp(0:npx-1,0:npy-1,0:npz-1)} and fill them with the correct \texttt{pid} values for each node of the rectangle \texttt{pp(:, :, :)} or the cuboid \texttt{ppp(:, :, :)}.  

### 1.5.2 Domain Decomposition and Domain Recomposition

These operations are frequently used in the beginning and the ending phases of a parallel program solving partial differential equations on a grid. The process of partitioning and spreading an array across processors is usually referred to as \textit{domain decomposition}. The inverse operation, \textit{domain decomposition}, is to gather the partitions of an array from the npe processors and reconstruct the original array. Typically the array is then output to a file or a printer.

In the following discussion we use a three-dimensional array as an example, but note that the procedures work for two- or one-dimensional arrays as well. The subroutine

\[
\texttt{decompose(ff,f,halo)}
\]

is used to spread an array \texttt{ff(:, :, :)}, assumed to reside on processor 0, across the set of \texttt{npe} parallel processors, where each processor will have one part \texttt{f(:, :, :)}, usually with overlapped boundaries (halo points, see Fig. 1). We sometimes call the array \texttt{ff(...)} a global array, and the array \texttt{f(...)} a local array. The halo points hold the physical boundary conditions for the global field

\[
\texttt{ff(1-halo:nnx+halo, 1-halo:nny+halo, 1-halo:nnz+halo)}
\]

which is decomposed into the local fields

\[
\texttt{f(1-halo:nx+halo, 1-halo:ny+halo, 1-halo:nz+halo)}
\]

where \texttt{nx=nnx/npx, ny=nny/npy, nz=nnz/npz}. The halo points of the latter arrays (the internal halo points) serve as the "boundary conditions" for computing the subfields.
Fig. 1 A 2-dimensional field is decomposed with halo points around it

Also the reverse operation, \texttt{recompose(ff,f,halo)}, is often needed: the arrays \( f(:, :, :) \) are collected and the original array \( ff(:, :, :) \) is to be reconstructed on processor 0, where the halo points of the local arrays are deleted. Note that in many parallel programs the global array is only virtual, i.e. it does not really exist as an entity, but only the local arrays exist in reality. Note that the subroutines \texttt{decompose(ff,f,halo)} and \texttt{recompose(ff,f,halo)} are used are called in exactly the same way for up to three-dimensional fields of REAL values.

1.5.3 Exchange of Halo Boundaries

The various algorithms for solving partial differential equations all work in a similar way: they update iteratively the values of the field. On a parallel machine each processor is responsible to update his own arrays \( f(1:nx,1:ny) \), excluding the halo points. In the following example we use a two-dimensional array. The halo points serve as the boundary conditions for the subfields. After each iteration, the halo points must be updated from the values computed by the neighbouring processors of each processor. Subroutine \texttt{exchange(f,halo,dir)} will do this task. Again it is a generic procedure, to be used for one- two- or three-dimensional arrays. While \texttt{halo} is a required argument in the range \( 0<\text{halo}\leq2 \), the direction \texttt{dir} is optional with values in the range \( 1\leq\text{dir}\leq3 \). If omitted, the exchange is done in all dimensions of the array \( f(\ldots) \). Otherwise, the exchange is only done in the direction mentioned. The direction parameter corresponds to the dimensions of the array and we consider the indices as corresponding to \( x, y, z \).

1.5.4 Periodic Boundary Conditions

The subroutine \texttt{periodic_boundary(f, halo, dir)} is used to do the exchange in such a way that in the direction requested the halo points are exchanged between the processor pairs \( px=0 \) and
px=npx-1, py=0 and pz=npy-1, and pz=0 and pz=npz-1, respectively. This subroutine is used if the partial differential equation is to be solved with periodic boundary conditions.

2 Fortran 90 and the Parallel Computer

In this section we discuss various aspects of Fortran 90 as a language suitable for parallel computing. The further development of Fortran 90 into Fortran 95 is indeed stressing the application on parallel computers even more (FORALL statement, PURE procedures, ELEMENTAL procedures, etc.). Indeed, Fortran 95 is the basis for the High Performance Fortran Language (HPF). We shall however not go into any details.

Fortran 90/95 is a big step forward for doing numerical simulations on any computer, but even more so on a parallel machine. We cannot discuss in detail all the advantages of Fortran 90 and Fortran 95 over Fortran 77 (like: modules, explicit procedure interfaces, syntax for array arithmetic, assumed shape arrays, user derived types, generic procedures, operator overloading, pointers, FORALL, PURE procedures, ...).

One of the main advantages of Fortran 90 is having dynamic memory allocation built right into the language. Some Fortran 77 compilers offered the extension INTEGER POINTER: this was a concept in accordance with the memory-oriented view of the older Fortran language. In Fortran 90 memory orientation should be considered obsolete, and indeed COMMON, EQUIVALENCE, and assumed size arrays, like var(*), need not and should not be used any more. Fortran 90 is in a way object-oriented: entities are referenced by their names (use association for entities in MODULEs, host association for internal procedures, etc.) and not by their storage location.

We consider allocatable arrays to be absolutely necessary for parallel computing: Given a parallel program, it must be decidable at run-time, depending on the problem size and the number of processors available on the parallel machine, what size of arrays are needed. Otherwise one would have to do with re-compilation of the program after having changed array boundaries, etc. (a nightmare with the notoriously slow Fortran 90 compiler of the T3E), or one would have to resort to the INTEGER POINTER mechanism which we consider harmful because it is not a high level language concept but introduces explicit addresses into a program. Unfortunately, we have to use the latter method on the Cray T3E.

The MPP Environment has been implemented in two versions: The high performance T3E version T3E_Environment, based on the shmem library of Cray, and the portable MPI version MPI_Environment which may be used on different parallel machines. In the next two subsections we discuss the two versions.

2.1 Using MPP Environment

The modules MPI_Environment and T3E_Environment have been implemented in such a way that the functionality is the same. The more general version MPI_Environment can easily be
replaced by the specific version T3E Environment by just exchanging the USE statements in the user's program, to recompile it, and to load the correct module.

At present, the functions shown in Fig. 2 are available. If the need arises, the functionality may easily be extended.

Initialization, clock, random numbers:

```fortran
subroutine init_MPP_Environment()
  real function clock()
  real function random(seed)
  integer function skip_seed(seed, incr)
```

Broadcast and collective functions:

```fortran
subroutine broadcast(object)
  function global_sum(object)
  function global_min(object)
  function global_max(object)
```

Support for partial differential equations:

```fortran
subroutine PE_2d_array
subroutine PE_3d_array
subroutine decompose(ff, f, halo)
subroutine recompose(f, ff, halo)
subroutine exchange(f, halo, dir)
subroutine periodic_boundary(f, halo, dir)
```

Fig. 2 The list of functions available in MPP Environment

### 2.2 Using MPP Environment on the T3E

We have set forth the requirements for scalable parallel programs, which includes both the run-time assignment of the number of processors on which to execute, as well as the possibility to determine the size of the arrays to be used at run-time.

While MPI implements communication with intermediate buffers, the situation is different for the Cray T3E shmem library: communication may only be performed between symmetric objects which have the same memory address on each processor (see the shmem_intro manual page, references to UNICOS/mk). Symmetric objects are:

© 1998, R Dohmen, F Hertweck, M-G Pacco-Düchs  Informatik, MPI Plasmaphysik
• Fortran data objects in COMMON or with the SAVE attribute

• Fortran arrays allocated with shpalloc(...)

• Fortran stack variables declared with a !DIR$ SYMMETRIC directive  
  (NOTE: there are no stack variables defined in the Fortran standard! The user obviously is  
  supposed to know something about the implementation of the compiler which we consider  
  objectionable)

The following data items may be accessed with shmem procedures if first the addresses are exchanged in a prelude:

• Fortran arrays allocated by halloc(...)

• Fortran PE-private data objects on the stack (see the note above!)

This list is obviously incomplete since allocatable objects (arrays) are not mentioned. Indeed, a test  
program shows that they are not treated correctly: though a variable might be declared as an  
ALLOCATABLE SAVEd array, the addresses on different processors may be different. The attempt to  
use the !DIR$ SYMMETRIC directive is rejected by the compiler.

We have to conclude that the present implementation of the Cray Fortran 90 compiler on the T3E is  
not fully suited for parallel programs. What is obviously necessary is to have a directive like SHARED  
for allocatable arrays, so that the Fortran 90 ALLOCATE statement will invoke the shpalloc  
procedure underneath, i.e. the shared heap storage instead of the local heap. All processors must of course  
execute the ALLOCATE on SHARED variables synchronously. (Most desirable of course would be an  
additional attribute SHARED in the Fortran language for the declaration of shared variables.)

It seems that so far the Fortran Committee (X3J3 of WG5 of ANSI) restrained itself from  
introducing explicit parallel processing concepts into the language, in the (in our view unjustified) belief  
that parallel processing can be controlled with comment-type directives, so that a parallel program can  
also be run on a sequential computer. This restricts parallelism more or less to SIMD type applications. In  
our experience any efficient parallel program of the SPMD type contains explicit references to a Parallel  
Function Library that does not exist on sequential machines. In addition, if a parallel computer truly is  
needed, a sequential computer would be too slow anyway.
```plaintext
type (my_type), allocatable :: dummy(:)
integer :: size_of_my_type
... allocate (dummy(2))
size_of_my_type = address(dummy(2)) - address(dummy(1))
deallocate (dummy)
...
where

int address(int var)
{ return var; }
```

Fig. 3 How to obtain the memory size of a derived type object

The only way around this problem is to use the Cray procedures `shmalloc(...)` and Cray integer pointers to allocate memory to variable sized fields (note that Cray integer pointers are not Fortran 90 pointers and are a non-standard extension!). This method has a serious drawback, however: because when calling `shmalloc(p_var,n,...)` the memory for the pointee array `var` is allocated as a sequence of `n` words. Therefore the user must know the number of words needed. This may be difficult when the object is an array of a derived type. Fortran 90 being a high level language, there is no notion of the memory size of a data type: for the intrinsic types, we usually just happen to know how many bytes are used -- for derived types there may be internal padding! In most cases the size can be computed by counting the components of a data type, but there is no guarantee of a correct answer. So probably the safest way is to allocate an array of two objects and use a C-function `address(var)` which returns the address of the variable (see the example in Fig. 3). Note that Fortran always transmits addresses of arguments.

We explain now the structure of a parallel program that meets the requirements of scalability. We define a module `job_environment` which contains various global constants and variables, like the size of the arrays, and the subroutines and functions of the job. The user's main program shall be constructed as a subroutine `main_program`, where the variably sized fields are imported as arguments. To make things simpler we consider a two-dimensional problem (see Fig. 4).

The de facto main program is `T3E_job` which first calls `init_T3E_Environment` to initialize the module `T3E_Environment`. Then `T3E_job` should read in the size of the problem to be solved (the values of `nx`, `ny`, ...). These define the global array `ff(0:nx+1,0:ny+1)` which has to be distributed onto a processor array of shape `nx × ny`. So next we compute the shape of the distributed arrays `f(0:nx+1,0:ny+1)`, where `nx = nx / npx`, etc. The memory is allocated by the shared memory heap allocation procedure for integer pointers, `shmalloc(nwords)`. This sets the address for the pointee `f0(*)`. We then call the subroutine `main_program` with the argument `f0` which transfers the address of `f0`.

Because it will frequently be needed, the size of the array is declared in the module `job_environment` (their values were set by `T3E_job`). These global values are used also in the declaration of the dummy argument of `main_program`. Note that the distributed arrays `f(:, :)` are declared with the attribute `POINTER in `job_environment`, whereas the dummy argument of
main_program has the TARGET attribute. The first action of subroutine main_program is to associate the pointer variable \( f(\cdot,\cdot) \) with the dummy argument \( \text{ff}(0:nx+1,0:ny+1) \). Then the subroutines of the job can be called. Note that this pointer association cannot be done in T3E_job because \( f0 \) is a linear assumed size array there (therefore not conformable). To alternatively define it as \( f0(1,1) \), as is often done, will not do because then it is conformable but fixed size.

```
module MPP_Environment
  ... constants
  ... variables
  ... generic interfaces
  contains
    ... functions
    ... subroutines
end module

module job_environment
  use MPP_Environment
  integer :: nx, ny, nz
  real, pointer :: f(\cdot,\cdot)
  contains
    subroutine init
      ... end subroutine
    subroutine output
      ... end subroutine
end module

program T3E_job
  use MPP_Environment
  use job_environment
  real :: f0(1,1)
  pointer (pf0,f0)
  call init_MPP_Environment
  read(*,*) nnx, nny
  nx = nnx / npx
  ny = nny / npy
  n = (nx + 2) * (ny + 2)
  call shmalloc(pf0,n,err,0)
  call main_program(f0)
end program T3E_job

subroutine main_program(ff)
  use T3E_Environment
  use job_environment
  real, target :: ff(0:nx+1,0:ny+1)
  f => ff
  call init
  call output
end subroutine main_program
```

Fig. 4 The structure of a program using T3E_Environment.

In this way we have restricted the use of INTEGER POINTERs to the program T3E_job. The remainder of the job (main_program and the user's subroutines) only uses standard Fortran 90 features and is thus not contaminated with INTEGER POINTERs.

There is another possibility for constructing the Job which does not need pointers but uses the \!DIR$ SYMMETRIC directive instead. The job is shown in Fig. 5. In the main program T3E_job the field size \( nnx, nny \) is read in, \( nx, ny \) are computed and the subroutine main_program is called. In

© 1998, R Dohmen, F Hertweck, M-G Pacco-Düchs  Informatik, MPI Plasmaphysik

219
subroutine the array \( f(0:nx+1,0:ny+1) \) is declared with the directive \!DIRS\ SYMMETRIC. This will cause the field to be allocated implicitly with shmalloc. We don't like this method, because the field has to be passed to each and every subroutine explicitly. The more elegant solution is to have the global data -- here the field \( f(0:nx+1,0:ny+1) \) -- declared as a pointer in the module job environment. Note that a combination of the two methods is not possible because the compiler tells us the symmetric variable must not be a target.

There is a general problem with using shmalloc both explicitly and implicitly: there is no assurance that the address of variables allocated by this method have the same addresses on all Pes: it is possible to generate different addresses on different processors with shmalloc by using a processor dependent size. The runtime system does not check for this error.

---

**Fig. 5** A different approach setting up a job: using \!DIRS\ SYMMETRIC.
2.2 Using the MPI Version of MPP Environment

The module \texttt{MPI\_Environment} has the same functionality and the same interface structure as the module \texttt{T3E\_Environment}. Wherever \texttt{shmem} functions were used before, they were replaced by the MPI procedures. The only machine dependence arises in the following areas:

- The random number generator (it would be advisable to use a truly portable generator, like that of the Numerical Recipes)

- Construction of the processor array: In our model we assume that processors adjacent in the virtual 2d or 3d processor array also have low latency in reality. This depends on the type of parallel machine. Therefore assignment of processors to nodes of the processor array may possibly be optimized.

All other functionality is portable because we rely on the abstraction offered by Fortran 90 MPI.

3 Examples

In this section we shortly discuss a few examples to illustrate the usage of MPP Environment. The examples were run on our Cray T3E and some of them also on our IBM SP2.

3.1 The Monte Carlo Code TRIDYN

MPP Environment was used in the parallelization of the Monte-Carlo code TRIDYN which serves to simulate target sputtering processes. An incident ion striking the surface of a target typically collides with several atoms on its way through the target, and in case of sufficiently high impact energy these particles can come free, and moving through the target possibly create further recoil atoms. The paths of all these particles are followed until they have either come to rest or have left the target; for each recoil atom the final position and other characteristic quantities are recorded. From these quantities the modifications caused by the particle cascade in the target can be calculated and are taken into account before a new projectile is treated.
In the parallel program version each processor treats only a part of each particle shower and stores the corresponding particle characteristics in PE private variables. In order to do the target update the results of all PEs have to be collected and the updated target has to be broadcast to all PEs before the next shower can be treated in parallel. These steps require communication and are processed by using procedures of MPP Environment. Both the T3E and the MPI version of MPP Environment were used on the T3E, the T3E version showing only slightly better performance (cf. Fig. 6).

### 3.2 Heat Conduction Equation

As a second example to test the applicability of the MPP Environment we used a program for the iterative solution of the heat conduction equation in three dimensions:

$$\partial T/\partial t = D \Delta T$$

where $T$ is the temperature, $t$ the time, $D$ the heat conductivity, and $\Delta$ the Laplace operator. The temperature is a function of space and time,

$$T = T(x,y,z,t)$$

The computational domain is the unit cube. We used Dirichlet boundary conditions: in the plane $z=0$ a bump is preset, the other boundaries are fixed to 0. In the course of the simulation the bump
spreads into the inner of the cube by diffusion. The equation is solved iteratively with a finite-difference scheme using a spatial discretization of $64 \times 64 \times 32$ and carrying out 100 iteration steps.

![Graph showing performance per PE for the Heat Equation for the T3E and MPI versions.]

Fig. 7  Performance per PE for the Heat Equation for the T3E and MPI versions.

For the parallel implementation the processors are arranged as a three-dimensional processor array by means of the procedure `PE_3d_array` and the domain is decomposed blockwise to the processor array. The exchange of the halo boundaries after each iteration step can easily be carried out with the subroutine `exchange`. At the end the subdomains are recomposed and the resulting global array is printed.

Again both the T3E Environment and the MPI Environment have been tested. But now, for a growing number of processors, the MPI version becomes increasingly worse compared with the T3E version which uses the `shm` library. The `exchange` routine which is called very often cannot be implemented that efficiently in MPI. One reason is that there is no one-sided communication in MPI, whereas the `shm` library contains the powerful subroutines

```c
shm_put(target, source, length, pe)
shm_iput(target, source, t_stride, s_stride, length, pe)
```

the first for contiguous arrays, the second for strided array segments

Fig. 7 shows the performance of the test program on the T3E for the T3E and the MPI versions. The MPI version was in addition tested on the IBM SP2 and delivered correct results. The performance values, however, are not meaningful, because we used the machine concurrently with other users.

© 1998, R Dohmen, F Hertweck, M-G Pacco-Düchs  Informatik, MPI Plasmaphysik
4 Conclusions

We have shown that with the MPI version of MPP Environment we obtain portability between systems. In addition, because of identical interfaces between the user program and MPP Environment, the T3E version is also "portable".

Because the T3E has the advantage of "one-sided communication", which in fact is close to the idea of a shared memory machine as compared to a message passing machine, we obtain substantial benefits from exploiting this possibility for certain applications (see sect. 3.2).

Because we have extensively used the generic procedure concept of Fortan 90, the procedures can be called uniformly for all intrinsic data types, and the variables may be scalars or up to three-dimensional arrays. This makes programming very simple and we believe that our MPP Environment will cover a wide area of applications. For very sophisticated applications there is always the possibility to revert to MPI or shmep.

Acknowledgments

The discussions with and the suggestions by Jakob Pichlmeier are gratefully appreciated. Heidrun Friedrich supplied us with the parallel version of the TRIDYN code which she ported to the Cray T3E, including a few useful procedures.

References

[1] The Cray T3E Programming Model
[2] The Cray T3E Shared Memory Library
Data Management in Climate Research

by
Kerstin Kleese, CLRC - Daresbury Laboratory
Email: k.kleese@dl.ac.uk, URL: http://www.dci.clrc.ac.uk/Group/DCICSEHPA

To address the identified environmental research challenges, researchers need access to a wide range of observational data and model output, covering the human, physio-chemical and biological components of the Earth system. This data should be recognized as a highly valuable resource, but data from existing data centres are frequently under-used because of the difficulties of accessing the data and of assessing whether they contain anything of interest or relevance [Environmental Change Network, 1997]. Furthermore data exploration is still in its early stages.

The complexity of the phenomena and processes involved in climate research, make powerful super-computing essential. Today's parallel models are some of the most demanding codes we have. They push the machines available to their limit and are still in need of more resources. However the bottleneck in running these codes is not so much the code performance as the data handling strategies employed.

The seriousness of the problem is increasing rapidly, as new computer and observation technologies encourage the production of more data in shorter time spans. For an efficient working environment, mechanisms have to be put in place to support scientific work more effectively. In response, CLRC – Daresbury Laboratory has set up a new project to investigate this important issue and to develop new more satisfying strategies [Kleese K, 1998]
What is Data Management?

Data Management can be described by the software and hardware techniques used to facilitate the management and the exploration of data.

We can then group these techniques into four different categories:
- Data archival mechanisms
- Data retrieval mechanisms
- Support for program I/O
- Data exploration techniques

Further consideration has to be given to local, national and international aspects of these points as well as the question of whether we are dealing with "approved data sets" or "private data". Approved data sets are validated data sets stored for public access, e.g. in a national data centre.

LOCAL LEVEL
Data archival and retrieval techniques comprise managing the stored data, mechanisms for access and additional user interfaces (e.g. graphical interfaces/ web access) both for private data and approved data sets. Support for program I/O includes "in-time" delivery of input data at program start and during the program run, mechanisms to handle program output during and after the program run, and programming techniques to generate suitable I/O patterns in the code. These points are highly dependent on computer system type and configuration. Data visualisation is a special subsection of program I/O. It includes "in-time" and "real-time" delivery mechanisms for input data for the visualisation tool as well as mechanisms to handle data output during and after visualisation. Data formats also have to be considered, which ones are most commonly used, and whether they are available or easily compatible etc. Data exploration refers to tools that help the presentation of information in an accessible way as well as techniques that generate new information through analysing the existing data, e.g. statistics, pattern recognition or improve the data quality e.g. data fusion.
NATIONAL AND INTERNATIONAL LEVEL
On the national or international level we are more interested in organisational aspects like the similarity or rather dissimilarity of:
- Data formats
- Access mechanisms
- Computing environments
- Languages
We are also looking for mechanisms to manage and organize our personal data, as well as a more general support for locating data sets.

IN GENERAL
To determine the quality of the Data Management at a particular site or level, you do not only have to look at the quality of the involved components, but more importantly at how smoothly they work together and how user friendly the whole solution is. Only this will determine how effectively and efficiently it can be used.

Data - A valuable resource
Progress in climate research is based on two components the scientists and the scientific data holdings. Data are the lifeblood of scientific research and represent therefore one of our most valuable assets. In their day to day work researchers need efficient access to a wide range of observational data sample collections and model output covering the human physio-chemical and biological components of the earth system.

“Up to date, data from existing data centres are frequently under-used because of the difficulties of accessing the data and of assessing whether they contain anything of interest or relevance.”

[Environmental Change Network, 1997].

Data Centres
The technologies we use in Climate research have undergone a considerable change new observation, computation, and visualisation systems have been developed over the years, and more and more data is produced faster than ever before. This is putting high demands on the data centres! They have to prepare the data for further use by the academic and industrial community, often under considerable time constraint. And if they are linked to a computing centre they also have to manage the requests for private data archival and retrieval.
"National data centres, like world data centres, typically have a discipline focus, thus, which data centre you approach depends on the type of data you’re interested in."

Ann Linn, ICSU Panel on World Data Centres.

Another aspect that we need to consider is, that traditionally many data centres focus on a specific scientific area, e.g. sunspots. As a result, today’s climate data are distributed over numerous systems and sites. The UK alone has at least 37 different data centres. 37 different data centres also mean 38 different access mechanisms and even more data formats in which the data are stored. What it doesn’t mean is, that there is any support to find these data centres, assess the stored data and explore it. And this is only on a national level! Many of nowadays research projects need access to international data holdings and they are also combining a growing number of scientific disciplines. Existing arrangements are unsuitable to support these types of projects efficiently.

Climate Modelling

Computational environmental modelling has steadily progressed over the last decades. Modern super computing technology enables us to produce increasingly realistic representations of environmental processes. Research groups all over the world have developed sophisticated parallel codes, which explore the capabilities of current High Performance Computing platforms very well. So, everything should be alright. Unfortunately, that is not the case. As funny as it may sound, but the speed and the efficiency of these models is also their downfall. Today’s climate modelling codes produce significant amounts of output and they require in many cases large input data sets.

The following tables originate from a paper by Alan O’Neill and Lois Steenman-Clark, UGAMP [O’Neill A, Steenman-Clark L, 1998]:

Table 1. Typical atmospheric experiment data sizes for 10 model year run with data output four times per model day.

<table>
<thead>
<tr>
<th>Spatial Resolution</th>
<th>Data Sizes (Gbytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>climate</td>
<td>36.5</td>
</tr>
<tr>
<td>seasonal</td>
<td>74.8</td>
</tr>
<tr>
<td>climatology</td>
<td>98.1</td>
</tr>
<tr>
<td>forecast</td>
<td>324.1</td>
</tr>
</tbody>
</table>
Table 2. Typical ocean experiment data sizes for a 10 model year run with data output once per model day

<table>
<thead>
<tr>
<th>Spatial Resolution</th>
<th>Data Sizes (Gbytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>4° x 4° (global)</td>
<td>5.8</td>
</tr>
<tr>
<td>1° x 1° (Atlantic)</td>
<td>21.5</td>
</tr>
<tr>
<td>1° x 1° (global)</td>
<td>133.2</td>
</tr>
<tr>
<td>1/4° x 1/4° (Atlantic)</td>
<td>347.1</td>
</tr>
</tbody>
</table>

So, how do today’s systems cope with these requirements? The capabilities of components like: processors, networks, compilers, and scientific libraries have improved dramatically over the last years. But disk I/O and data archival and retrieval mechanisms haven’t kept pace with that. In fact, they already represent a major bottleneck for codes with high I/O requirements.

"The data volumes generated by climate models can be very large and are a problem to deal with especially when the models generating this data have been optimised to run quickly and efficiently on HPC platforms."


"A major bottleneck facing the SEA model now, is that of I/O."

Matthew I. Beare, University of East Anglia [Beare MI, 1998].

Future Tendencies

The seriousness of the situation is increasing rapidly, as new computing and observation technologies encourage the production of more data in shorter time spans. And lets be honest who isn’t interested in increased model resolutions, longer runs, more complex models and larger ensembles. So, the volume of the data we are going to produce in the future and therefore the amount of data the data and computing centres have to cater for, will increase dramatically over the next years. For an efficient scientific working environment better mechanisms have to be put in place to support scientific work on a local, national and international level!

"By the end of the century the volumes of data available will increase by several orders of magnitude."

Peter Churchill, CEO [Churchill P, 1995]
"The European Centre for Medium Range Weather Forecast (ECMWF) expects that its already big data store doubles in size every 18 months!"

Dick Dixon, ECMWF [Dixon D, 1998]

Scope of the Project

In the short term we will continue to study and assess the current situation of Data Management in Climate Research, as well as its influence on everyday scientific work. A big part of our efforts over the next year will be directed at a detailed investigation of all major hardware and software components involved in Data Management. We want to assess their quality, ease of use and how well they work together with other products. We would also like to get a better overview about Data formats. How many are there? Which ones are most commonly used? What is influencing peoples decision to use a particular format?

In the long run we would like to build on the the results of the previously described work. The study of the current situation will give us indications on, which areas of Data Management have the biggest influence on the scientific work and should therefore be the first ones to be targeted. The survey of the hardware and software components, will enable us to give advice, on preferable product choices. Often the exchange of a single product or the addition of a new one can result in remarkable improvements. We also hope to establish an information service that keeps the community up to date on the afore mentioned topics as well as new developments.

However our biggest goal will be, to be able to investigate and develop new tools:

To provide better access to the data, through standardized access interfaces and better support for multidisciplinary searches.

But more importantly we would like to develop tools to further the easier, better and more efficient exploration of data. We think that this is an area that could provide tremendous benefits for the community.

Some projects, though only covering limited areas, already show their potential, if they could be applied to a wider area.

There is for example Data fusion, where data from different sources and with different properties, e.g. Data from the Hubble space telescope and from a ground based telescope, are automatically combined on the computer to get the best of both worlds. This technique applied to other data could provide us with generally higher quality input data.

Another idea is the application of data mining techniques on large amounts of stored data, in this case to investigate air pollution tendencies, without the necessity of model runs. It is thereby freeing valuable computing time.
Summary

Data Management influences many areas of our daily working life, whether we want to store data, retrieve data, if we are concerned with program I/O optimization or if we want to explore the existing data holdings, again and again we are confronted with Data Management mechanisms.

And as Data Management has such a big influence, we should all be concerned about its quality. Today all of us have to deal with more and more tasks that have nothing to do with our real scientific work. In many cases we have to accept this development. But many of the tasks related to Data Management are just unnecessary! There are better ways, there are easier ways and there are faster ways to handle them!

So lets make sure that this will change in the future.

References

A Parallel Version of the Lagrangian-Eulerian Model of the HydroGeoChemical (LEHGC) Code on the Cray T3D.

Michael W Bowers

EPCC, University of Edinburgh,
James Clerk Maxwell Building,
Mayfield Road,
Edinburgh EH9 3JZ UK.

Email m.bowers@epcc.ed.ac.uk

Abstract

The LEHGC code was parallelised on the Cray T3D at EPCC. The mesh decomposition was accomplished using EPCC's PUL-MD utility. The message passing paradigm was used using MPI. The finite element of the original code was replaced with a parallel Conjugated-Gradient method using the Parallel Iterative Method library. Initial results show good scaling properties and there is considerable scope for future development.

1 Introduction

Over the last decade, the general public's awareness of environmental issues has greatly increased. These issues include; recycling of raw materials; the extinction of wildlife; land management; renewal of energy sources and global warming. The ability of mathematical modelling to simulate these and related problems will enable mankind to take decisions in the immediate future which will have important long term consequences.

An important problem that has to be addressed is the disposal of radioactive waste from nuclear powers stations. Suitable geographical locations and insulation methods are required to prevent radioactive products entering the environment. Mathematical models can be used to investigate such locations, but also, just importantly, help reassure the local population.

The Lagrangian-Eulerian Model of the HydroGeoChemical code, LEHGC, (Yeh, 1994), is a state of the art coupled transport code. It can be used to model the migration of radionuclides in groundwater. LEHGC is a FORTRAN code with a reasonable level of documentation and a test history in international projects. It is also undergoing continual development which makes its an appropriate choice for parallelisation.

The following sections give a brief introduction to transport modelling; an overview of the LEHGC code; an overview of the parallelisation approach; initial results; conclusions and possible future work.
2 Couple Chemical Transport

The simulation of reactive contaminant transport requires an adequate understanding of both chemical and hydrogeological processes acting in tandem. The dominant chemical factors likely to influence aqueous radionuclide migration include; speciation, both organic and inorganic, precipitation and dissolution, sorption/exchange and diffusion in reponse to chemical gradients. To these must be added colloid dispersion and flocculation, though the impact of these ultra-fine particulates is not full understood.

This section describes the basic concepts behind hydrogeochemical models and outlines some of the approaches used to couple hydrogeologic transport to chemical equilibria. Yeh and Tripathi (1994) provide useful appraisals of the relative merits of alternative coupling strategies.

Two of the most frequently mentioned terms in chemical transport modelling are components and species. Definitions are as follows. Components are set of linearly independent "basis" chemical identities through which every species can be uniquely represented as a linear combination.

Mathematical representation of the transport of solutes by advection, diffusion and kinematic dispersion is governed by the conservation of mass, energy and charge. Rubin (1983) describes alternative formulations of the basic solute transport equations and assesses their application to fast, reversible as opposes to slow and/or irreversible reactions. This clear distinction defines the limits of applicability of the 'local equilibrium assumption' (LEA) but, in practice, coupling inevitably involves a comprise. Until recently, limitations with respect to data availability and computer resources meant that reconciling the requirements of a large number of non-equilibrium reactions with a geometrically complex flow field was unrealistic. New developments in the computer sciences have now removed one of the obstacles to achieving significant enhancements in coupled chemical transport modelling.

Current tools overwhelmingly simulate chemical systems at equilibrium subjected to a constant velocity flow in one or, rarely, two dimensions. They all address the ingress of a solution of known composition at the boundary (static or transient) into a field solution whose composition must also be specified a priori. The model then simulates the displacement of the initial solution by mixing and reaction with the boundary fluid.

Two distinct protocols for coupling chemical reactions to transport equations have been followed. The direct or 'one-step' method (Jennings et al. 1982). This approach tends to be restricted to one dimension problems. By contrast, the indirect or 'two-step' method (Yeh and Tripathi, 1991) employs a discrete chemical and non-reactive transport modules, iterating between these two move the evolving solution in time and space. This approach, though less rigorous is inherently more flexible and, in principle, can be extended to highly complex problems. For these reasons, this approach offers the best prospects for advanced applications.

LEHGC is a recent example of the indirect model. It simulates two-dimensional transport through both saturated and unsaturated media and offers a number of advantages over similar codes.
3 LEHGC

3.1 General Features

The LEHGC code is a hybrid Lagrangian-Eulerian Finite-Element Model of HydroGeoChemical (LEHGC) transport through saturated and unsaturated media. It solves two-dimensional transport and geochemical equilibrium equations iteratively and is a direct descendant of the widely-used HYDROGEOCHEM (Yeh and Tripathi, 1991). LEHGC employs an Eulerian scheme which allows larger time steps to be used in the advection-dominant transport calculations. This causes less numerical dispersion and thus, alleviates the problem of negative concentration. It is also computationally efficient when compared to its predecessor.

The version to be used in this work separates the system of transport and geochemical equilibrium equations into subsystems which are solved in an iterative fashion. The transport equations include terms for advection, dispersion, diffusion, sources/sinks and mass production and removal due to chemical reactions and radioactive decay. The code has been designed to treat heterogeneous and anisotropic media; consider spatially and temporally distributed sources/sinks as well as point sources/sinks; accept the prescribed initial conditions or obtain initial conditions by simulating the steady-state version of the system under consideration and deal with prescribed transient concentrations over a Dirichlet boundary. The chemical reactions considered are aqueous complexation, adsorption/desorption, ion exchange, precipitation-dissolution, redox and acid-base reactions.

The treatment of chemical equilibrium is based on the equilibrium constant approach which produces a set of non-linear algebraic equations based on the law of mass action and the principle of mass balance. The aqueous and complexed species are subject to hydrologic transport; the adsorbed, ion-exchanged and precipitated species are not and their transport equations do not include terms for advection, dispersion and diffusion. The set of non-linear, algebraic equations that define the geochemical equilibrium reactions are solved using the Newton-Raphson iterative technique. Although hydrologic transport is addressed, flow fields are specified within an input file. This file may be produced by any flow code and may contain a steady-state solution or a transient results over any number of time planes. Flow fields for saturated or unsaturated flow in porous or fractured media can be used in LEHGC simulations.

LEHGC is designed for generic application to reactive transport problems associated with such systems as contaminant in subsurface media. Input to the program includes the geometry of the system, the spatial distribution of the finite elements and nodes; the properties of the media; the potential chemical reactions and the initial and boundary conditions. Output includes a summary of simulation progress as a function of space and time, and the detailed chemical speciation at user specified nodes.

3.2 Solution Algorithm

The key variables of the problem are chemical concentrations of various types, representing 'master species', complexes, precipitated and sorbed components. The three main (partial differential) transport equations relate gradients of these concentrations with the moisture content field $\theta$.
\[
\frac{\partial T_j}{\partial t} + \frac{\partial \theta}{\partial t} (S_j + P_j) = L(C_j) - \theta A_j^a + M_{j}^a - QC_j, \quad j \in N_a \tag{1}
\]

\[
\frac{\partial W_j}{\partial t} + \frac{\partial \theta}{\partial t} W_j = -\theta A_{j}^s + M_{j}^s, \quad j \in N_s \tag{2}
\]

\[
\frac{\partial \theta}{\partial t} N_{eq} + \frac{\partial \theta}{\partial t} N_{eq} = -\theta A_{eq} + M_{eq} \tag{3}
\]

where

\( \theta = \) moisture content field (i.e. groundwater field);
\( T_j = \) total analytic concentration of the \( j \)th aqueous chemical compound;
\( S_j = \) total sorbed concentration of ditto;
\( P_j = \) total precipitated concentration of ditto;
\( C_j = \) total dissolved concentration of ditto;
\( L(\cdot) = \) spatial advection/dispersion differential operator \( f(\nabla) \);
\( A_j^a = \) total decay rate ditto;
\( M_j^a = \) total rate of source/sink production ditto;
\( Q = \) total rate of water injection/extraction at \( \theta \) source/sinks;
\( N_a = \) number of aqueous chemical components;
\( W_j = \) total analytic concentration of the \( j \)th adsorbent chemical component;
\( A_j^s = \) total decay rate of ditto;
\( M_j^s = \) total decay rate of ditto;
\( N_s = \) number of adsorbent chemical components;
\( N_{eq} = \) equivalent concentration of ion-exchange sites;
\( A_{eq} = \) total decay rate of ditto;
\( M_{eq} = \) total rate of source/sink production of ditto;

A further ten algebraic chemical equilibrium equations provide additional reactions between these quantities.

The above equations are discretised over a two-dimensional grid composed of irregular quadrilateral and or triangular finite elements. The field variables take values at the nodes of this finite element grid.

The most important equation is equation (1), for two reasons. Firstly, the analytic concentration \( T_j \) is the key variable of interest, and secondly, equation (1) is the only equation containing spatial derivatives; the other transport equations can be solved by simple integration in time, while the chemical equations are of the non-linear algebraic form.

The general algorithm used by LEHGC v1.0, is thus:

1. Solve equations 2 and 3 for \( W_j \) and \( N_{eq} \).

2. Take an initial guess for \( T_j \).
3. Solve the ten chemical equations to give \( S_j \) and \( P_j \). Since these equations involve no spatial derivatives, solution proceeds on a gridpoint by gridpoint basis.

4. Solve equation 1 for \( T_j \) using the \( S_j \) and \( P_j \) from the previous step. Given the spatial derivative term in this equation, the solution proceeds using the full finite element method of assembling, element by element, a node-to-node stiffness matrix and load vector, then solving the resultant matrix equation.

5. Compare the new \( T_j \) with the previous value; if convergence is not obtained, go back to step 3.

4 Parallelisation of LEHGC

4.1 Comments on Parallelising LEHGC

LEHGC is fundamentally a straightforward finite element code. The addition of chemical interactions to the transport equations adds no real complication since there is no spatial dependency in the chemical reactions. The Finite Element Method (FEM) is a computationally efficient tool for deriving approximate numerical solutions to partial differential equations over a discretised domain and is currently a very active application area in massively parallel computing.

FEM derives this numerical approximation of the hydrologic transport equations by dividing the domain of interest into a collection of non-overlapping but touching elements. This set of elements is collectively known as the mesh. LEHGC only allows the use of triangular or quadrilateral elements. The value of the equations is typically expressed as a simple polynomial, that is, a linear combination of a set of functions called basis functions. The coefficients of the basis functions at each node are derived from a system of linear equations. This system arises from the minimisation of theerror between the approximate and exact solutions to the partial differential equations. Thus, the FEM transforms the hydrologic transport equations into a set of linear equations of the form \( Ax = b \). In the context of FEM, the coefficient matrix \( A \) is called the stiffness matrix and \( b \), the force vector. This system is solved for \( x \), which gives the value of the coefficients of the basis functions at each point in the discretised domain. The stiffness matrix, \( A \), can be derived by computing a set of definite integrals over the elements of the mesh. If nodes \( i \) and \( j \) in the mesh share elements, then \( A(i,j) \) is given by the summation of the integrals calculated over all the elements shared by points \( i \) and \( j \). Thus, the only non-zero entries in the \( A(i,j) \) are where grid points \( i \) and \( j \) share an element.

There are three important properties of the stiffness matrix and force vector:

(a) For most applications, the finite element mesh is not a regular structure, but is highly irregular. Thus, the stiffness matrix is usually an unstructured sparse matrix.

(b) Computing the stiffness matrix and force vector is relatively inexpensive compared to the overall solution of the linear system. Furthermore, computing individual entries of \( A \) only requires only local computations to the element. Consequently, this computation is trivial to parallelise.

(c) The resulting system of linear equations is large and sparse, and, hence solving it is the most computationally expensive phase of the FEM. It is this phase for which efficient parallel
solutions are critical.

The principal issues in efficient parallel implementations of FEM are minimising load imbalance among processors and maximising the ratio of computation to communication on the processors.

4.2 Parallel Program Structure

The parallelisation of a code using mesh decomposition usually leads to the development of two extra stages in the solution procedure. These are a preprocessor and a postprocessor in addition to the main parallelised code.

A preprocessor was developed which runs on a Cray J90, the front end to EPCC’s Cray T3D. Its main function is to decompose the original, large unstructured mesh into a number of partitions suitable for distribution among several processors. This application code used EPCC’s mesh decomposition library, PUL-MD (Trewin, et al, 1997).

The parallel version of LEHGC has been designed to run the a massively parallel computer with distributed memory such as the Cray T3D. The message passing interface, MPI (see references), has been used for communications. The equilibrium linear matrix solvers have been replaced by a parallel conjugate-gradient method implemented using the Parallel Iterative Library, PIM (da Cuhna and Hopkins). This particular library was chosen for several reasons. First, it is freely available to academics and has been used previously in industrial projects. It is also in continual development and ports are available for many machines.

4.3 Message Passing Interface (MPI)

The Message Passing Interface (MPI) is a standard for use by all those who want to write portable message passing in FORTRAN F77 and C. The attractiveness of this paradigm at least stems from its wide portability. Programs expressed this way may run on distributed-memory multiprocessors, networks of workstations, and combinations of all these. The standard includes callable routines for:

- Point-to-point communications
- Collective operations
- Process groups
- Communications
- Process topologies
- Bindings for FORTRAN F77 and C
- Environmental management and inquiry
- Profiling interface

4.4 Parallel Utilities Library - Mesh Decomposition (PUL-MD)

The intention of mesh decomposition is to produce evenly sized mesh partitions (to balance compute load) and to minimise the size of the boundaries between partitions (to minimise communications costs). This optimal partitioning is difficult to obtain but several algorithms have been devised.
EPCC, as part of its Parallel Utilities Library (PUL) has developed a mesh decomposition library, PUL-MD. It is a library of serial routines that may be called from an ANSI-C or FORTRAN 77 application code. PUL-MD provides a choice of decomposition algorithms, and a set of related functions to manipulate mesh data into a form ready for input into a parallel application. The choice of algorithm is a trade off between performance (quality of decomposition) and computational cost of obtaining it.

In a parallel mesh application, each processor works on only one mesh partition. It is usually necessary to know which nodes and sides are part of the local mesh, as well as which elements. The decomposition process provides a list of elements belonging to each partition and hence, which nodes belong to which partition. It is obvious that adjacent elements within the original mesh will share nodes. The decomposition process may have to decide which partition is responsible for carrying out certain calculations at those nodes and to which partitions those results would have to be passed.

4.5 Parallel Iterative Methods (PIM)

The Parallel Iterative Methods (PIM) is a collection of F77 routines designed to solve systems of linear equations on parallel computers using a variety of iterative methods. To allow portability of code across different multiprocessor platforms, PVM and MPI are used for interprocessor communications.

Several iterative methods are provided. In the version used there are eleven implemented. These include the Conjugate-Gradient (CG); the Bi-Conjugate-Gradient (Bi-CG); the restarted generalised minimal residual (RGMRES) and Chebyshev acceleration. The coefficient matrix may be either real or complex and there is a choice of both single and double precision.

In general, iterative methods are used in conjunction with preconditioners which can accelerate convergence. PIM is structured to allow the use of preconditioners, whether they be left-, right or symmetric preconditioners. Also, there are several differing stopping criteria available.

In order to maximise flexibility of usage, PIM requires the user to provide three linear algebra operations: a matrix-vector (and transpose-matrix-vector) product; a preconditioning step; inner product and vector norm. By hiding these operations from PIM the user has complete freedom with respect to how the matrix is stored, accessed and partitioned. It also allows portability across a variety of parallel machines.

A particular iterative method can be used simply by calling the relevant routine. For example, to use the single precision conjugate-gradient method then the following would be included in the application program;

```
call pimsfcg(x,b,wrk,ipar,spar,matvec,diagl,diagr,pssum,psnrms)
```

The routine has several arguments which can be classified into two particular types; those that define the user defined routines and those that define the parameters of the problem to be solved. The coefficient matrix is passed to the user defined routines using common blocks.

The arguments `matvec`, `diagr`, `pssum` and `psnrms` are the routines for the matrix-vector product; the left or right preconditioner; the global sum and the vector norm respectively.
The arguments \textit{ipar} and \textit{spar} are the integer and floating point arrays containing the input and output parameters. The input data would include the number of processors being used; the processor identification; the stopping criteria choice; the maximum number of iterations allowed and details on how the coefficient matrix is partitioned. The output information gives details on whether the routine has converged to a solution and if not, an indication of any problems encountered.

The argument \(x\) is the vector which contains the initial estimate of the solution at the start of the routine and the last estimate when the routine terminates. \(b\) is the right hand side of the matrix problem while \(wrk\) is a work vector used internally by PIM.

## 5 Test Problem and Results

Due to the complex problem description of the original code which was retained only a relatively small test problem has been currently used.

The mesh is rectangular with quadrilateral elements consisting of a 128x16 elements. Each element is 20dm by 20dm. The test problem considers the transport and interaction of eight chemical components. This results in twenty-three aqueous complexes and five subsurface species. The flow velocity is 3dm/day, the porosity is 0.3, and the dispersivity is 1dm. A total of three days were simulated.

The time-dependent boundary conditions are such to allow the buildup of certain compounds.

The speedup obtained using an increasing number of processors is shown in figure 1. It can be seen that even for this small test problem there is good scaling. With sixteen processors, there is a speedup of thirteen. This can be explained by the fact that most of the computation involved in solving the spatially independent chemical equations. A set of linear equations has to be solved for each component and species on each processor.

## 6 Conclusions and Future Work

The initial results indicate that the parallel LEHGC has good scaling. Although the Langrangian and transient velocity field functionality were not implemented in the present work, it is anticipated that their future implementation should not effect the performance greatly.

A typical industrial code has been parallelised within three months. The use of MPI and PIM will allow a great deal of portability across many different types of distributed memory computers. Useful experience of PIM has been gained which can be fed into future projects at EPCC.

In addition to implementing the missing functionality of the original LEGHC code, future work would include investigating the performance of the CG method with preconditioning. Also, the parallel code would benefit by replacing the solver for the chemical equations with a serial CG method, possibly using PIM in serial mode.

In the longer term, the parallelisation of LEHGC will allow realistic problems to be solved faster. However, it also gives the opportunity to modify the code to solve for other features not currently modelled, thereby, expanding its capability. The most obvious way forward is to couple LEHGC with a flow solver. Currently, the flow field is predefined before each run and
would have to be possibly have to be read in for each time step. FEMWATER (Yeg and Ward, 1980) uses the same mesh and hence, the same partitioning. Coupling them into the same parallel code should be straightforward if they share the same data structures.

Further proposed development of LEHGC by its author would also be applicable to the parallel version. For example, it has been proposed to add thermal transport. In addition it is proposed to replace the chemical equilibrium model with the updated, EQPITZ; a geochemical module that uses the Pitzer activity approach. Another model is KEMOD, a mixed kinetic/equilibrium chemical speciation model.

7 References


8 Figures
The Parallel and Portable Linear Solver Package LINSOL

Hartmut Häfner, Willi Schönauber, Rüdiger Weiss

August 26, 1998

Numerikforschung für Supercomputer, Rechenzentrum Universität Karlsruhe, Postfach 6980, 76128 Karlsruhe, Germany, e-mail: haefner@rz.uni-karlsruhe.de

Abstract

LINSOL is an iterative linear solver package (solving $Ax = b$). An integrated LU/ILU algorithm working as preconditioner is being implemented. The program package is adapted to the application of sparse matrices, but can be efficiently applied to full matrices, too. It contains presently fourteen iterative methods and ten polyalgorithms of generalized Conjugate Gradient (CG) methods. Currently eight different normalization methods are used as preconditioner.

Ten different data structures are supported by LINSOL to ease the embedding of LINSOL into an application as well as the mapping of arbitrary sparse matrices to storage patterns.

1 Basic Concepts of LINSOL

Some fundamental rules have been followed designing easy-to-use and efficient mathematical (super)computer software:

- **Flexibility or ease of use** - by the design of LINSOL that enables the user to use three types of interfaces and by the support of ten different data structures and storage patterns, respectively.

- **Portability** - by using the standardized Fortran77 and Fortran90 programming languages on single processor systems and additionally the message passing paradigm on parallel computers.

- **Robustness** - is obtained by fourteen different iterative solvers based on generalized CG methods; they are suitable for different classes of matrices $A$; additionally, adaptive polyalgorithms are provided that choose an appropriate solver from the fourteen methods. To enhance the robustness of LINSOL an (incomplete) LU algorithm with an integrated bandwidth optimizer embedded as preconditioner into the iterative methods is being integrated into the program package.

- **Efficient code** for workstations (PCs), vectorcomputers and parallel computers - by reusing data in the highest memory hierarchy as far as possible and by using vector pipelining and parallel algorithms with highest possible volume-to-surface ratio and designing them so that latency hiding is possible.

2 The Design of LINSOL

LINSOL is designed so that three types of interfaces can be used. Using the standard interface, parameters must be defined in a file controlling the program flow of LINSOL. The matrix $A$ can be stored in Harwell/Boeing- or LINSOL-format. On $p$ processors the matrix $A$ can be stored distributed in $p$ files. The matrix $A$ stored distributed in $p$ files can be read again on $p$ processors to be computed locally. The LINSOL-format has been developed with the aim of minimizing the storage requirements of a single processor on parallel computers. For the storing the matrix is read in partitions of vectors corresponding to the data structures of LINSOL; on parallel computers each vector, i.e. each basic data structure like diagonals, rows or columns, is divided into vector parts that belong to different processors; then these vector parts are sent immediately to the respective processors. As a consequence this method enables the reading of a matrix, to be distributed on many processors, on a single processor, if the main memory of one processor is too small to store the whole matrix.

An arbitrary number of Right Hand Sides must be stored in an additional file. The use of Initial Guesses is optional. On parallel systems the solution vectors are gathered on one processor. Their number corresponds to the number of Right Hand Sides and they are written into a new user-specified file.
The Fortran90 user interface LSOLP enables the user to integrate the program package into an application. To run LSOLP on more than one processor the matrix $A$ and the vectors $x$ and $b$ must be distributed onto the processors. The Fortran77 user interface LSOLPP can be used on systems lacking a Fortran90 compiler.

All commonly used storage patterns for sparse matrices can be mapped to one of the ten data structures of LINSOL; these basic data structures are: main diagonal (assumed full), full (non-main) diagonal, packed diagonal (elements and their indices, from the first to the last nonzero element), indexed column (one element per row and the corresponding column index, from the first to the last nonzero element), indexed row (similarly one element per column and its row index), starrsy (nonzero elements with row and column indices), full row, full column, packed row (nonzero elements in a row and their column indices from the first to last nonzero element) and packed column (similarly nonzero elements in a column and their row indices). They can be assembled without cutback to the matrix $A$, if $A$ is nonsymmetric. A symmetric matrix $A$ is stored in a special storage scheme using the symmetry of the matrix. Matrix $A$ can be represented as

$$A = A_s + D + A_{sT},$$

where $D$ is the main diagonal of $A$ and $A_{sT}$ denotes the transpose of $A_s$. If $A$ is symmetric, only the matrix $A_s$ and the main diagonal $D$ must be handed over to LINSOL. Assembling $A_s$ all basic data structures can be used with the cutback that main diagonal elements must be stored in $D$ - equivalent to the basic data structure "main diagonal". Thus the user interfaces LSOLP and LSOLPP can be embedded easily (in less than one day) into an application program. The storage patterns of LINSOL except "main diagonal" are shown in Fig. 2.

3 Portability

On single processor systems portability is achieved by using the standardized programming languages Fortran77 and Fortran90.

At present the only possible choice is the message passing paradigm to implement portable parallel programs. All manufacturers of MIMD computers offer some message passing libraries, but the problem is that these libraries have no common subset of basic message passing routines. There are two ways to solve this problem.

The first one is to use PVM or MPI. On nearly all MIMD computers PVM and MPI are available, so that this is now the most comfortable way to implement portable parallel programs. For performance reasons MPI should be used.

The second one - which has been chosen by us - is to implement our own Portable Message Passing Interface (P.MPI) [4]. We only defined interfaces for a few important basic message passing routines like send, receive and communication-setup routines being common to all well-known message passing interfaces; the communication-setup routine sets a few communication parameters like e.g. a table for the logical processor numbers and the
Basic data structures of the matrix A supported by LINSOL and the corresponding basic vector operation types of the matrix-vector multiplication (c=A*r):

![Diagram showing different data structures with corresponding operations](image)

Figure 2: Basic data structures of LINSOL.

according physical processor numbers. Since we provide interfaces to all well-known message passing interfaces the user only has to change the name of the message passing interface in the link step to get another message passing interface. So the user can get the highest possible performance rates (concerning the communication network) for all MIMD computers.

Thus we have implemented a simple, synchronous message passing interface for the Cray T3E basing on the shared memory put operation. On each processor 2 * p buffers, 2 buffers for each of p processors, of a predefined length are initialized within this interface. The length determines the number of outstanding sends and receives respectively. For each of these 2 * p buffers exists a pointer to the next free location in the buffer scrolling through the buffer till its end to start again at the first buffer entry.

How does the message passing now work? We want to send data from processor j to processor i. The receive routine called on processor i puts (shared memory put) the three variables mid, loc, msgtyp into the i-th send buffer on processor j from whom the matching send routine is started. The pointer to the i-th send buffer on processor j shows the next free location in the send buffer. The variable loc contains the memory address of the receiving processor where the data to be sent should be stored in. Then the receive routine sets the pointer to the next free location and returns.

The send routine scans through the i-th buffer until the variable msgtyp identifying matching communication operations is found. Then it puts the data to be sent - start address and length of the data are parameters of the interface - into the memory of processor i starting with address loc. After that it puts the variable mid into the j-th receive buffer on the processor i. The pointer to the j-th receive buffer shows the next free location in the this buffer. Then the send routine clears the entry in the send buffer for the sent data, sets the pointer to the next free location and returns.

The receive-wait routine scans through the j-th receive buffer until the variable mid identifying matching receive and receive-wait routines is found. Then it knows that the sent data have arrived and returns.

Analyzing the protocol one can see that this message passing interface is synchronous. The send routine is blocking and can not complete before the matching receive has sent necessary information data. The receive routine is nonblocking. Fig. 3 shows exemplarily the message passing protocol on 3 processes.

A synchronous interface means that there are restraints one has to pay attention to. If e.g. data are sent in a ring shift, the alternate buffer technique must be used. Omitting this feature means to run into a deadlock. As LINSOL always uses this technique, synchronous message passing protocols can be employed; beyond it this feature allows...
the hiding of the computation behind the communication (latency hiding).

4 Robustness

To get robustness fourteen different iterative solvers [8] enabling the solution of linear systems for most classes of matrices are implemented. Additionally, adaptive polyalgorithms are provided that choose an appropriate solver from the fourteen methods. The used iterative methods are Krylov subspace methods. There are five different classes of methods:

1. methods for symmetric and positive definite matrices: classical CG and ORTHOMIN. The methods require one matrix-vector multiplication per iteration step.

2. truncated and restarted methods for moderately nonsymmetric systems like GMRES, ORTHORES and PRES20. These methods require one matrix-vector multiplication per iteration step. PRES20 is a special version of smoothed ORTHORES and the iteration process is restarted every 20-th step. For sufficient diagonal dominance PRES20 is a quickly converging method.

3. biconjugate gradient-based methods for nonsymmetric but definite systems like BCG, BICO, QMR, CGS and BiCGSTAB. All above mentioned algorithms have the same origin: BCG. All of them have the common property that two matrix-vector multiplication have to be performed in each iteration step and that the exact methods can be formulated with a short recurrence, i.e. the methods do not have to be restarted during the iteration. The latter property is one of the reasons for the popularity of BCG-based methods. BCG can be considered as exact CG method applied to a double system. The residuals and errors of BCG oscillate heavily. In order to obtain a better behaviour Schönhauer applied minimal residual smoothing to BCG [5]. Then the algorithm is called BICO. QMR is implemented without the look-ahead process that is proposed in the original paper by Freund and Nachtigal [1]. The difference between QMR and BICO is the different kind of smoothing to be applied to BCG. All these methods including CGS and BiCGSTAB are closely related, because they have the same origin. Numerical tests confirm a similar behaviour of these methods. However, BiCGSTAB seems to be superior in many cases.

4. methods based on the normal equations as emergency exit like CGNE and CGNR-ATPRES. Both methods require two matrix-vector multiplications, one by $A$ and one by $A^T$. For these two methods the condition
number is squared, i.e. the convergence is slow for matrices close to symmetric matrices. Theoretically both methods converge always. Thus they are very robust, but not very efficient.

5. error-minimizing methods like GMERR. Three variations of this algorithm are implemented in LINSOL: GMERR(5), GMERR(20), GMERR(S). GMERR(S) is Fridmann's method and can only be applied to symmetric matrices. GMERR(5) means that the GMERR method is restarted every fifth step; GMERR(20) restarts GMERR every 20-th step. Both restarted versions can be applied to nonsymmetric matrices. To avoid the oscillation of the residuals, residual-minimizing smoothing is applied.

Smoothing leads to a monotonical decrease of the residuals and thus can improve the qualitative convergence, even if it cannot improve the quantitative convergence, i.e. the convergence is not essentially accelerated by smoothing; however, a better control of the iteration process results.

All these methods are explained with their properties in detail in [7].

The fourteenth algorithm solves the linear system $A^T A z = b$ with the matrix $A$ as input matrix. LINSOL internally exploits the symmetry of the matrix $A^T A$.

Furthermore, LINSOL supports different kinds of polynomials. A key for the development of these techniques is the perception that the full set of iterative methods can be represented by a quite smaller set. It seems to be reasonable to select one representative method of the truncated or restarted CG methods (class 1), one representative of BCG-based methods (class 2) and one of methods based on normal equations (class 3); the polynomials in LINSOL monitor the norms of the residuals and switch from efficient but less robust methods to more robust but less efficient methods. Polyalgorithm I and II start with PRES20. If the reduction of the residual is not sufficient after certain steps, then BICO is applied. If BICO converges too slowly polyalgorithm I switches to the emergency exit ATPRES, whilst polyalgorithm II checks the convergence behaviour and returns to PRES20, if its convergence behaviour was better than that of BICO, else it switches to ATPRES, too. Polyalgorithm I never changes from ATPRES to another algorithm; polyalgorithm II again checks the convergence behaviour of ATPRES against that of BICO and returns to BICO, if it is better. Three more polynomials call GMERR(5) or GMERR(20) before calling ATPRES and are implemented without or with checking of the convergence behaviour. Polyalgorithm VI (VII) starts with GMERR(5) (GMERR(20)) and switches to ATPRES; polyalgorithm VIII additionally checks the convergence behaviour. The polynomials IX and X start with GMERR(S) and switch to the classical CG method; they are only suited to symmetric matrices. Over all ten polynomials are implemented in LINSOL. For all methods and polynomials hold, that no information about the eigenvalues or other inner properties of the matrix is required.

Currently an LU/ILU algorithm is being implemented as preconditioner (ILU stands for Incomplete LU). The Gaussian elimination process will only operate on the skyline of the matrix $A$ to accelerate the computation and minimize the storage requirements. To get a small skyline of the matrix $A$ a bandwidth optimizer will be used in a preprocessing step.

5 Optimization Strategies

The following principles [5] have been considered for the design of LINSOL:

- Keep the floating-point units continuously busy. As this is a mostly unattainable goal, this principle can be reformulated into a practical rule: Reuse data in the highest memory hierarchy as far as possible. This rule has been realized in LINSOL by the implementation of cache reuse strategies for the matrix-vector multiplication.

- Use the principle of the separation of the selection and processing of the data. Thus processing can take place with vectors of contiguous or (and) indexed elements. This rule has been realized implicitly in LINSOL by supporting data structures for all commonly used storage patterns for sparse matrices.

The consequent use of vector operations in LINSOL leads not only to an optimal code for vectorcomputers, but also for workstations with superscalar processors; for cache-based systems like workstations there is a switch in LINSOL to enhance cache reuse.

These single processor optimizations are a prerequisite for an optimal parallel code.

LINSOL has been developed as "black box" software. Thus the parallelization of the used methods must be transparent to the users. This goal has been achieved by the automatic distribution of the matrix $A$ by a single processor onto matrix blocks on many processors. Changing the number of processors means to change the data the processor works on; it does not mean to change the algorithm. Thus the convergence rate is the same - apart from rounding errors - for all numbers of processors. Adapted to the distribution of the matrix $A$ all vectors of
LINSOL are distributed in vector parts onto the processors. This leads to scalability regarding the memory, which is an absolutely essential feature for parallel software. All iterative methods implemented in LINSOL use as basic operations

- matrix-vector multiplications,
- reduce operations like dot products and
- vector operations.

The parallel key operation is the parallel matrix-vector multiplication. The overall goal for the implementation of this operation is to achieve scalability regarding the computation time by minimizing communication, skipping unnecessary communication cycles and hiding the communication behind the calculation on parallel computers with asynchronous communication (latency hiding).

![Diagram of parallel matrix-vector multiplication](image)

**Figure 4:** The Parallel Matrix-vector Multiplication on 4 processors. Active blocks are shaded.

To minimize communication, i.e. to hold the matrix $A$ locally on the single processors, the matrix $A$ is distributed in $p$ row blocks of nearly the same size onto the $p$ processors. Additionally each row block of the matrix $A$ is subdivided into $p$ logical column blocks corresponding to the distribution in row blocks, Fig. 4.

247
Thus the matrix-vector multiplication $c = A \cdot r$ is computed concurrently on the $p$ physically distributed row blocks - all processors working on different column blocks. The mathematical formulation of the parallel matrix-vector multiplication is:

$$c^i = c^i + \sum_{j=1}^{p} A^{i,j} \cdot r^j.$$ 

In Fig. 4 the matrix-vector multiplication is depicted on 4 processors. Each processor $i(i = 1, \ldots, p)$ must multiply sequentially different column blocks $A^{i,j}(j = 1, \ldots, p)$ with the corresponding vector parts $r^j$. During these sequential computation steps the vector parts $r^j$ must be sent around in a ring shift. Diagonal matrix blocks (e.g. $A^{1,3}, A^{2,4}, A^{3,1}, A^{4,2}$) are computed concurrently, if at least one of these blocks contains non-zero matrix elements. If all diagonal matrix blocks contain only zero matrix elements (e.g. in Fig. 5 $A^{1,1}, A^{2,2}, A^{3,3}, A^{4,4}$), the computation of all these diagonal matrix blocks can be skipped, i.e. the number of communication cycles is reduced by one. Double buffer technique is used to enable overlapped processing of computation and communication on parallel computers with independent communication units.

For many algorithms like BCG, BICO, QMR, CGNE and CGNR-ATPRES the matrix-vector multiplication by the transpose of $A$ ($d = A^T \cdot r$) is additionally required. Since the matrix $A$ is not transposed physically, now the parts of the result vector have to be sent around in a ring shift. Whilst the contribution of a block is computed, the part of the result vector is transferred, then the contribution is added and the new partial result is sent again. The parts of the input vectors now don't have to be moved around. Thus on each processor $i(i = 1, \ldots, p)$ different vector parts of $d$ are computed by

$$d^i = d^i + \sum_{j=1}^{p} A^{i,j \cdot T} \cdot z^j = d^i + \sum_{j=1}^{p} A^{i,j} \cdot z^j.$$ 

Again the double buffer technique is used.

This strategy of parallelization of the matrix-vector multiplication leads to a high volume-to-surface ratio depending on the sparsity of the matrix $A$ and thus the software gets scalable regarding the computation time.

![Diagram](image)

**Figure 5:** Computation of dot products on 7 processors.

While reduce operations like dot products are processed very efficiently on vector computers, they become a critical operation on parallel computers. As all vectors are distributed onto the $p$ processors, each processor first
computes its local dot product(s); then a reduce routine is called that gathers scalars via a cascade over all processors; these scalars are newly computed as sums of the local and received scalars within the reduce routine. After this step the global dot product(s) are only available on the first processor. In the second part of the reduce routine this scalars are scattered onto all processors using the reverse cascade as for the gather operation (see Fig. 5).

In an optimized reduce routine the gather and scatter operations are performed concurrently on all processors, i.e. all processors - not only the first processor - compute the global dot products in a cascade (see Fig. 6). This optimization leads to a reduction of reduce steps by a factor of 2.

![Figure 6: Optimized computation of dot products on 8 processors.](image)

In order to reduce the communication overhead independent reduce operations can be executed locally and then all independent partial and final results are sent by one and only one message on each processor involved in the cascading operation.

Vector operations can be executed independently for the corresponding parts of the vectors that are distributed on different processors.

## 6 Measurements on an CRAY T3E-900

First we measured the simple and double Ping-Pong communication benchmark with standard nonblocking MPI-routines. For the simple (double) Ping-Pong we got a startup time of 23 (31) microseconds and a peak performance of 166 (147) MB/s. Then we performed the same measurements with the described synchronous message passing protocol basing on shared memory put routines. Now we got for the simple (double) Ping-Pong a startup time of 7 (11) microseconds and a peak performance of 300 (218) MB/s. Thus the communication time is reduced by a factor of 3 for very short messages and by a factor of 1.5 till 2 for very long messages.

Second we measured the computation of a dot product on 8 processors for different vector lengths and compared the performance rates by linking the two different reduce routines. Using MPI we got a performance enhancement for the optimized reduce routines of factor 1.3 (5) for very long (short) vectors. Using the described synchronous message passing protocol we got a performance enhancement for the optimized reduce routines of factor 1.009 (1.3) for very long (short) vectors. The measurements show that the optimized reduce routine accelerates the computation of reduce operations on parallel computers.

We tried to enhance the performance of our test example with both optimization methods.

<table>
<thead>
<tr>
<th>test cases</th>
<th>$e$</th>
<th>$n$</th>
<th>$m$</th>
<th>$op$</th>
<th>$it$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1872</td>
<td>26793</td>
<td>$4.14 \times 10^6$</td>
<td>$8.98 \times 10^5$</td>
<td>513</td>
</tr>
<tr>
<td>2</td>
<td>3825</td>
<td>52800</td>
<td>$8.38 \times 10^6$</td>
<td>$18.13 \times 10^5$</td>
<td>657</td>
</tr>
<tr>
<td>4</td>
<td>8000</td>
<td>107163</td>
<td>$17.38 \times 10^6$</td>
<td>$37.55 \times 10^5$</td>
<td>778</td>
</tr>
<tr>
<td>8</td>
<td>15625</td>
<td>204828</td>
<td>$33.73 \times 10^6$</td>
<td>$72.79 \times 10^5$</td>
<td>974</td>
</tr>
<tr>
<td>16</td>
<td>31713</td>
<td>408192</td>
<td>$68.12 \times 10^6$</td>
<td>$146.86 \times 10^5$</td>
<td>1323</td>
</tr>
<tr>
<td>32</td>
<td>64000</td>
<td>811923</td>
<td>$136.93 \times 10^6$</td>
<td>$294.97 \times 10^5$</td>
<td>1561</td>
</tr>
</tbody>
</table>

Table 1: Characteristics of the test cases.

As test example we take matrices that are generated by the FEM program package VECEM [2] applied to the system of 3-D partial differential equations of computational mechanics for the unit cube under surface load. A 3-D grid is generated so that the number of unknowns per processor approximately stays constant for all processor

249
numbers. As the resulting matrix is symmetric, only half the non-zero elements except of the main diagonal are stored and the classical CG algorithm is used in LINSOL.

The characteristics of the test cases are compiled in Table 1 (ε means global no. of elements, n means global no. of unknowns, m means no. of non-zero elements in matrix, op means no. of operations, it means no. of iterations in LINSOL).

For test case i, i = 2, 4, 8, 16, 32 we get approximately the i-fold number of unknowns in comparison to the test case 1. For this type of problem the bandwidth, relative sparsity and the condition number of the matrix increases with the problem size. With the condition number the number of iterations in LINSOL increases. Increasing n means to get more critical problems.

In Table 2 we present only two elapsed timings in seconds: tc : p means that test case tc runs on p processors, tCG is the total time for CG and tIt is the time for one CG iteration. In addition to the MFlop/s rate the MFlop/s ratio to the single processor test case is presented. In Fig. 7 the ratio of MFlop/s for the T3E-900, based on the last column of Table 2, is depicted.

<table>
<thead>
<tr>
<th>tc : p</th>
<th>tCG</th>
<th>tIt</th>
<th>MFlop/s</th>
<th>ratio_MFlop/s</th>
<th>tCG</th>
<th>tIt</th>
<th>MFlop/s</th>
<th>ratio_MFlop/s</th>
</tr>
</thead>
<tbody>
<tr>
<td>1:1</td>
<td>229.3</td>
<td>.447</td>
<td>20.1</td>
<td>1</td>
<td>145.6</td>
<td>.284</td>
<td>31.6</td>
<td>1</td>
</tr>
<tr>
<td>2:2</td>
<td>287.2</td>
<td>.437</td>
<td>41.5</td>
<td>2.06</td>
<td>208.8</td>
<td>.318</td>
<td>57.0</td>
<td>1.80</td>
</tr>
<tr>
<td>4:4</td>
<td>130.1</td>
<td>.198</td>
<td>91.6</td>
<td>4.56</td>
<td>104.9</td>
<td>.160</td>
<td>113.3</td>
<td>3.59</td>
</tr>
<tr>
<td>8:8</td>
<td>149.5</td>
<td>.192</td>
<td>195.6</td>
<td>9.73</td>
<td>137.8</td>
<td>.177</td>
<td>212.1</td>
<td>6.71</td>
</tr>
<tr>
<td>16:16</td>
<td>177.0</td>
<td>.182</td>
<td>400.0</td>
<td>19.90</td>
<td>166.0</td>
<td>.170</td>
<td>428.2</td>
<td>13.55</td>
</tr>
<tr>
<td>32:32</td>
<td>89.4</td>
<td>.092</td>
<td>791.2</td>
<td>39.36</td>
<td>88.1</td>
<td>.090</td>
<td>808.8</td>
<td>25.59</td>
</tr>
<tr>
<td>16:64</td>
<td>129.0</td>
<td>.098</td>
<td>1498.5</td>
<td>74.55</td>
<td>155.0</td>
<td>.117</td>
<td>1255.1</td>
<td>39.72</td>
</tr>
<tr>
<td>32:128</td>
<td>164.4</td>
<td>.105</td>
<td>2809.2</td>
<td>139.76</td>
<td>165.3</td>
<td>.106</td>
<td>2782.7</td>
<td>88.06</td>
</tr>
</tbody>
</table>

Table 2: Measurements on the Cray T3E-900 in Jülich with MPI for p = 1 to 128 processors (DEC Alpha EV-5, 450 MHz, 900 MFlop/s peak performance, 8 KB primary data cache and 96 KB secondary cache, 128 MB main memory). Left results without, right ones with cache reuse.

![Graph showing the ratio of MFlop/s for the Cray T3E-900 based on the measurement of test case 1 on 1 processor with switched on cache reuse.](image)

Figure 7: The ratio of MFlop/s for the Cray T3E-900 based on the measurement of test case 1 on 1 processor with switched on cache reuse.

In [6] measurements on further supercomputers like Cray T90, NEC SX-4 and Fujitsu VPP300 are presented. In [3] measurements on the parallel computer IBM SP are presented.

It should be stressed that the MFlop/s ratio is not a speedup. To get the classical speedup we would have to solve the same problem for all measurements. But such a test example is not suited as the larger matrices A do not fit into the memory of a single processor.

Because of the small memory of 128 MB on the Cray T3E-900 in Jülich we had to increase the number of processors for the test cases 2 and 8 to run the test example up to test case 32. Without cache reuse we get a
MFlop/s ratio which is greater than the theoretical speedup. This is due to a better cache reuse for matrices. A with increasing number of unknowns n. Switching on the implemented software module for cache reuse pays nearly always. Fig. 7 also shows the scalability of the program package LINSOL regarding the computation time.

Linking the own synchronous message passing interface instead of MPI only leads to a performance enhancement of 1.4 percent (on 4 processors) till 2.5 percent (on 64 processors) in comparison to the listed performance rates. Using the optimized reduce routine the performance rates stay constant. The minimal performance enhancement in comparison to the benchmarks show that the communication is optimally hidden behind the computation in LINSOL.

7 Concluding Remarks

LINSOL is a black-box iterative linear solver package. This means that firstly all used iterative solvers are parameter-free and secondly the users do not have to know anything about the internal data structures to use the LINSOL-library. The software is optimized for vectorcomputers and parallel computers, but it also runs efficiently on all UNIX- or LINUX-based workstations. The measurements on the parallel computer Cray T3E show that LINSOL can solve extremely large problems on distributed memory parallel computers with an efficiency comparable to that of vectorcomputers.

Acknowledgement: We thank the HLRZ Jülich for the possibility to use their Cray T3E-900.

The references [4] [6] [8] are available in WWW under URL http://www.uni-karlsruhe.de/LINSOL.

References


Remote Visualization over Standard Network Connections

Youngbae Kim
Lawrence Berkeley National Laboratory, Berkeley, California
A. E. Koniges
Max-Planck Institute for Plasma Physics, Garching, Germany and
Lawrence Livermore National Laboratory, Livermore, California
Shrihari Gopalakrishna
Department of Applied Science, UC Davis, Livermore, California

ABSTRACT:

One useful role of an MPP is to generate data that is to be visualized in real-time on a remotely located workstation. We consider the use and implementation of this operation on the Cray T3E MPP system over standard networking available to the average user. At the MPP end, we use CUMULVS (combined with PVM) developed by Oak Ridge National Laboratory for messaging. To visualize the data received through CUMULVS on the remote workstation, we use the AVS visualization system, a commercially available package from Advanced Visual Systems. We discuss potential PVM start-up problems and remedies for computers with various levels of security. We also consider interaction with the T3E batch mode of operation. We use a test program to generate data at various equally spaced intervals. We consider both local visualization (i.e., both computers located at the same site) and transatlantic communication (one computer located in California and the other located in Germany). This latter example of very long distance communication has advantages because it allows a user, say in Germany, to communicate with large batch jobs (which might be restricted to run at night) in California because of the 9-hour time difference.

KEYWORDS:
Remote Visualization, Production MPP Computing, CUMULVS, AVS, PVM, T3E

1. Introduction

Motivation

Along with the now practical use of massively parallel, high-performance computers the nature of scientific simulations is rapidly changing. High-performance computers that allow users to interactively visualize and diagnose problems can dramatically alter the way one studies the dynamics of complex systems. In modern simulations of nonlinear dynamical systems, for example, it is not unreasonable to represent a typical volume with $100^{*100}3$ grid points and this resolution level is expected to increase to $1000^{*100}3$ grid points within the decade. The need for managing this massive data, often in real-time if human interactivity is an objective, is very real and spawned a relatively new discipline, called Scientific Data Management (SDM). The challenge of SDM is to provide the tools and supporting infrastructure to organize, navigate and manage tera-scale data streams. An effective SDM framework should seamlessly integrate databases, mass storage systems, networking, visualization and other computing resources in a manner that hides the complexity of the underlying system from the scientist performing data analysis. Interactivity, mainly for inputting data and parameters, and for analyzing the output is an essential element in these simulations. Interactive graphical interfaces are far superior to textual information interfaces
during the analysis phase as important collective phenomena can be readily identified in a timely manner and acted upon.

If real-time visualization capability is an objective, then the scientist should have the capability to control the display of results interactively. In a traditional scenario, the output generated during a simulation would be stored in files at fixed regular intervals which would then be analyzed by the scientist using graphics tools in order to draw conclusions. If it is desired to view the results with a better temporal resolution, the simulation has to be repeated with the results being stored more often. This process can get very inefficient for time-consuming simulations, especially in the developmental phase. Often it is desirable to change the frequency with which the results are stored, based on whether the data being generated at some point in the course of the simulation is interesting or not, as decided by the scientist.

It is well known that remote steering and visualization are possible, and many demonstrations of such communication have been given. [1] In this paper, we consider the practical issues for those with ordinary connections. In particular, both the local and the remote machines may have security issues. The connection between the machines may not be fast and dedicated. Thus our goal here is to provide advice and sample programs for use by the average computational scientist. In particular, our systems consist of a T3E MPP, and a local workstation. Remote steering or visualization requires the coordination of local (workstation) and remote (T3E) processes. The following issues must be addressed

- Starting up local and remote processes, either together or separately.
- Establishing communication between these processes on the T3E and a remote steering machine (your workstation).

One software solution that can address both issues is PVM (Parallel Virtual Machine). PVM enables users to configure a collection of computers (hosts) as a virtual machine (VM) and start processes anywhere on the VM. PVM subroutines allow these processes to communicate. The "glue" that holds together a virtual machine is a set of PVM daemon processes, one per host. These daemons start and manage PVM user processes, and help them establish communication. The problems of starting processes and establishing communication therefore reduce to the problem of setting up the virtual machine. In Section 2, we discuss some common start-up problems and their solutions.

Once a virtual machine is set up (meaning that a PVM daemon has been started on each host of the VM) you need to start PVM programs on each host -- a PVM program (e.g., parallel application) on the T3E and a PVM program (i.e., for steering and visualization) on your local workstation. This prototypical situation is depicted in Figure 1.

Various T3E’s are configured to limit interactive jobs, both by number of processors and CPU time. Thus most production jobs on a real T3E environment have to be controlled while running in batch mode. This adds additional complications to steering longer-running jobs. In Section 3 we discuss how to run PVM daemons and jobs in a batch mode, (i.e., inside an NQE/NQS job) and how to interact with PVM batch jobs. For general machine configurations, there are two possibilities of starting these programs on each of the two hosts.

- User runs a PVM program as the master program on local workstation, which spawns using `pvm_spawn()` a slave program (e.g. parallel program) on the T3E.
- User starts a parallel program on the T3E and attaches a steering/visualization program to it.
In Section 4, we provide example codes to illustrate these two possibilities. PVM on the T3E is a specific implementation designed to exploit the MPP architecture. Special properties and limitations of this PVM are described in Section 5.

PVM is low-level, and doesn’t provide automatically a capability for interfacing steering and visualization software to applications. In general, a remote steering and visualization problem involves several tasks, e.g.,

1. controlling job execution
2. inspecting results, e.g. by visualization
3. changing input parameters interactively

CUMULVS, developed at Oak Ridge National Laboratory, provides a higher-level interface for transferring data to and from a parallel application. It allows a user to change input parameters, visualize simple data, or send data to a visualization/analysis tool such as AVS. PVM remains the basic means of communication. In Section 6, we describe a test implementation of remote visualization using CUMULVS and AVS. As a proof of concept, we implement this on workstations at the Max-Planck Institute for Plasma Physics in Garching, Germany, (MPG) and test communications to the T3Es at Lawrence Berkeley Laboratory’s NERSC (National Energy Research Scientific Computing Center) in California and the T3E at MPG.

2. Remote Job Startup

Before one can use PVM for communications, it must be established that the two (or more) machines can be configured as a virtual machine. In an ideal world, there perhaps would be no hackers thus no security issues on computer communication. However, this is clearly not the case in real life. Thus, major computing sites enable various forms of security to prevent damage to systems and files. Detailed information on PVM start-up and problems is given in the PVM Book. Here, we describe briefly some problems and solutions encountered in our specific environments.

2.1 Use \texttt{rsh} with the T3E as master: batch or interactive

If a remote shell (rsh) is permitted onto the workstation, the standard PVM startup mechanisms of using rsh should work. As long as you have the appropriate .rhosts file on the workstation, which identifies the T3E as an allowed host, you should be able to start up PVM and configure a virtual machine. For example,

\textbf{using the command "add host" in the PVM console}

```
pc% pvm
real  pvm> add rdfs.aug.ipp-garching.mpg.de #add a Sun workstation
1 successful
HOST DTID
rdfs.aug.ipp-garching.mpg.de 80000
pvm> conf #Check configuration
2 hosts, 2 data formats
HOST DTID ARCH SPEED
pc 40000 CRAY 1000
rdfs.aug.ipp-garching.mpg.de 80000 SUN4SOL2 1000
pvm> halt #be sure to halt pvm when you are done
pc%
```

Some problems we incurred while attempting this included (solutions given in parentheses):

254
Different user names on the different machines (see solution below)
.host-deny files based on IP address (added a .host-allow file)
Access to .rhosts file limited by AFS (changed AFS access rights)
.rsh path not available (set the environment variable PVM_RSH)
Environment variables not set properly on workstation, so pvmd cannot be found (see solution below)
PVM already running on workstation or PVM terminated abnormally in a previous execution (delete your pvmd log file /tmp/pvmd.*)

To test for this basic functionality, one can try executing commands with rsh between machines. If all is working correctly, the following dialogue from PVM Book, pg. 152 should be possible (here demonstrated between the Sun Workstation and PC).

```
ack@s4rfs 104) rsh pc /opt/ctl/mpt/mpt/pvmd3/lib/pvmd -s
[pvmd pid70323] slave_config: bad args
[pvmd pid70323] pvmbailout(0)
ack@s4rfs 105)
```

**Manual start-up using a hostfile with an entry for each host**

An alternative way to start PVM is using a hostfile, which may include various options such as a username on the remote machine, a password on that machine, and the path to pvmd on the host. Here is an example:

```
[pierre.56 ] cat hostfile
dolly.lbl.gov dx=$PVM_ROOT/lib/pvmd
louis.lbl.gov dx=$PVM_ROOT/lib/pvmd

[pierre.57 ] pvmd hostfile & # run pvm on each host in background
[1] 68001
socket address: /tmp/jtmp.008381a/aaa0000a68001

[pierre.58 ] pvm
pvmd already running.
3.3.10 (Cray PVM for UNICOS Version 3.1.x.6)
t40001
pvm> conf
3 hosts, 2 data formats

<table>
<thead>
<tr>
<th>HOST</th>
<th>DTID</th>
<th>ARCH</th>
<th>SPEED</th>
</tr>
</thead>
<tbody>
<tr>
<td>pierre</td>
<td>40000</td>
<td>CRAY</td>
<td>1000</td>
</tr>
<tr>
<td>dolly.lbl.gov</td>
<td>80000</td>
<td>SUNMP</td>
<td>1000</td>
</tr>
<tr>
<td>louis.lbl.gov</td>
<td>c0000</td>
<td>SUNMP</td>
<td>1000</td>
</tr>
</tbody>
</table>

pvm> quit
pvmd still running.
```

**Calling pvm_addhosts() in a program**

This approach seems at first like the most flexible, but it puts responsibility for virtual machine startup inside the program and generally forces the PVM master host (the one with the first PVM daemon) to be the same as the application master host (the one with the controlling process). By entangling virtual machine startup and process startup, this approach makes life more difficult when rsh doesn’t work, or when a job is run under the batch system.

**2.2 Manual startup using the so=ms option in host file: interactive only**
If rsh and related network services are disabled, PVM can’t automatically start remote daemons. However, as long as IP connectivity exists between hosts in a virtual machine, it is always possible to start remote daemons by hand, logging into each host separately and starting each daemon manually. In essence, you, the user, are acting as a surrogate rsh program for PVM. To enable manual daemon startup, you must use a PVM hostfile and supply the so=ms option for each host entry in the host file that you need to start by hand. Notice that this mechanism is incompatible with batch mode because it requires an interactive login session for each host.

For example, assuming that you are trying to configure a virtual machine with three machines, dolly (Sun workstation), pierre (T3E 600), and mcuirie (T3E 900), you create a host file as shown below. You may think of any of the hosts as master, then others will become slaves. In the following example, we assume that a master daemon is started on dolly.

```
[dolly. 11] cat hostfile
pierre.nersc.gov so=ms
mcuirie.nersc.gov so=ms
```

On the master host (dolly), you start PVM with the following command:

```
[dolly. 12] pvmd hostfile
```

Once a master daemon is started on dolly, it requests that you manually start a slave pvmd on pierre and mcuirie separately. Then you will see on the terminal of the pvmd on dolly

```
*** Manual startup ***
Login to "pierre" and type:
$PVM_ROOT/lib/pvmd -S -d0 -npierre 1 80a9ca95:0cb6 4096 2 80a95c43:0000
Type response:
```

On pierre, type the given line,

```
[pierre. 12] $PVM_ROOT/lib/pvmd -S -d0 -npierre 1 80a9ca95:0cb6 4096 2 80a95c43:0000
```

and the new daemon on pierre will print something like

```
ddpro<2312> arch ip<80a95c43:0a8e> mtu<4096>
```

which you should relay back to the master pvmd by typing (usually with cut and paste) the given line. At that point, the master pvmd prints

```
Thanks
```

and the two pvmds should be able to communicate. The process is repeated for mcuirie: the master daemon requests manual startup, you start the remote daemon by hand, and type the response into the master pvmd.

Here is an example.

### 2.3 Use of ssh
When `ssh` (secure shell) but not `rsh` is available on all hosts of a virtual machine you can start remote PVM daemons using `ssh`. To use `ssh` to start up remote PVM daemons:

- When using the T3E as a master host, set the `PVM_RSH` environment variable to the full path of `ssh` on the T3E.
- When using your local workstation as a master host, you have to re-compile the PVM from source after defining `RSHCOMMAND` in the appropriate configuration file (`$PVM_ROOT/conf/$PVM_ARCH.def`) to be the full path name of `ssh`.

The following example illustrates a modification in the `SUNMP.def` file for `dolly`.

```bash
[dolly:5:48pm:59.~] cat $PVM_ROOT/conf/SUNMP.def
ARCHFLAGS = -DSYSVFUNC -DSYSVSTR -DNOGETDTBLSIZE -DSYSV SIGNAL \ 
            -DNOWAIT3 -DNUNIXDOM -DRSHCOMMAND="/usr/local/bin/ssh" \\
ARCHLIB = -lnsl -lsocket -lthread
ARCHOBJ = 
ARCHLIB = -lnsl -lsocket -lthread
HASRANLIB = f
PVM_ARCH = SUNMP
```

### 2.4 Other mechanisms

The flexibility provided by the `PVM_RSH` environment variable is actually quite general. It is always possible to write your own command that implements the same functionality. In this document we mention using this flexibility to use `ssh` or to work around a bug with PVM running in batch mode, but you should keep in mind that it can be used to implement special purpose startup mechanisms.

### 3. Interaction with a Batch Job

In this section, we consider using the T3E as the master. The batch system on the T3E is the Network Queuing Environment (NQE), which interacts with an underlying NQS (Network Queuing System) batch system. NQE has additional layers of scheduling machinery, but from the user's point of view looks very similar to NQS.

Some of the issues that arise when trying to run distributed PVM applications under NQS are:

- Starting and stopping a virtual machine from within an NQS job
- Starting PVM processes within an NQS job
- Attaching remote processes
- Controlling batch jobs

Note: if not used carefully, the methods described in this section have the potential to wreak havoc with T3E scheduling. In particular, it is very important that your parallel application is started immediately after the PVM daemon is started, and inside the batch script, and it is important not to let any PVM job start unattended. This can cause significant idle time on the T3E.

### 3.1 Starting and stopping a virtual machine from within an NQS job

PVM applications can be run as part of an NQS job script. PVM daemons and processes started by those daemons inherit their process limits (e.g. number of PEs and CPU time) from the controlling NQS job. Therefore the PVM daemon on the T3E must be started from within the batch job. PVM
processes can also be started within the batch job, or then can be spawned from an external host using `pvm_spawn()`, since the daemon carries out the actual spawning. The requirement of starting the PVM daemon from within the batch job means that the T3E must be the master PVM host. (Technically, one can imagine writing an auxiliary program run from a batch job that listens to requests from a remote PVM master and starts a slave daemon, but this would be complicated and of dubious value).

### 3.1.1 Starting a local virtual machine.

The following example is a simple NQS job script that shows how to start a PVM daemon on a T3E.

```bash
module load mpt
pvm3 &
sleep 60
... start program...
pvm << EOF
halt
EOF
```

This example starts up a virtual machine that includes only the T3E. To start up a virtual machine with other hosts, you need to use a hostfile.

### 3.1.2 Starting a virtual machine including remote hosts

To add remote hosts to the virtual machine, you need to start PVM with a hostfile. Unfortunately you have to work around a bug in T3E PVM when running in batch mode. When a remote host is added to the current PVM configuration in batch mode under NQS, PVM passes an unnecessary internal argument to the `pvm3` started on the remote host. This causes the slave daemon to fail. PVM detects that it is running inside an NQS job by checking the environment variable `ENVIRONMENT`, which inside an NQS job is set to the string "BATCH". To work around the bug, you can fool PVM into thinking it is running interactively by setting this environment variable to an empty string.

Another workaround would be to write a shell script that removes the unnecessary argument. This shell script would replace the default remote shell, but invoke that remote shell after modifying the argument list. To have PVM use this special shell instead of the default shell, you would set the `PVM_RSH` environment variable to the full path of the shell script. For reasons of simplicity, we recommend the first solution, though considerations discussed in the next section might motivate the second solution.

The following NQS script illustrates how to work around the bug by setting `ENVIRONMENT` to an empty string.

```bash
module load mpt
setenv TMPDIR /tmp
setenv ENVIRONMENT ""
setenv PVM3_HOSTFILE &
sleep 60
```

# set $TMPDIR to '/tmp'
# set $ENVIRONMENT to null
# start the daemons
# wait for startup
... start program...

pvm << EOF
halt
EOF

# start up application in rest of script
# details discussed later
# start console to halt pvm

3.1.3 Interactive interaction with batch job.

When PVM is started interactively, the PVM log and daemon files are placed in the /tmp as /tmp/pvml.uid (log file) and /tmp/pvmd.uid (pvmd socket) respectively, where uid is the user ID. The PVM daemon file contains information telling local PVM processes how to contact the PVM daemon, which is required for them to be able to talk to each other or to remote processes.

As noted earlier, each batch job is associated with a separate virtual machine. This is accomplished as follows. PVM looks for two environment variables that NQS sets for each batch job: ENVIRONMENT and TMPDIR. NQS sets ENVIRONMENT to "BATCH" and TMPDIR to the name of a temporary directory that is private to the NQS job and that exists only while the job is running. As noted in the previous section, PVM checks whether ENVIRONMENT is set to "BATCH". If it is, PVM uses TMPDIR for its special files instead of /tmp. This allows more than one NQS job to run simultaneously without interfering, and also prevents interactive use of PVM from interfering with a running batch job.

Suppose, however, that you want to start a PVM daemon from a batch job and connect to it with an interactive job. For instance, you might want to start a PVM console on the T3E to monitor what processes are running, or possibly to add a new host. (Note that the interactive job will have normal interactive limits, while any process spawned by the daemon will have the limits of the NQS job.) To do this, you have three choices.

1. Set TMPDIR to /tmp inside the batch job.
2. Before running the interactive process, set ENVIRONMENT to "BATCH" and TMPDIR to the location of the temporary NQS directory (which is /tmp/nqs.++++????, where ????? is different for each NQS job).
3. Unset the ENVIRONMENT variable within the batch job, so that the daemon puts its files in /tmp (ignoring TMPDIR).

Note that options 1 and 3 only allow one PVM daemon per system, preventing several PVM jobs from being run per user at the same time under NQS. The discussion of the previous section forces option 1 unless you use a special script for starting jobs.

3.2 Starting processes within an NQS job

Inside an NQS job, processes are started in the same way as interactive processes.

In the following example, we show how to start two types of programs. In the first case, we start a single program called master. This program runs on a time-shared command PE but any processes it spawns run on dedicated application PEs. In the second case, we start an SPMD program with 4 processes, all of which run on application PEs.

module load mpt
setenv ENVIRONMENT "" # see previous sections
pvmd3 hostfile &     # start the daemons

259
sleep 60  # wait for startup
master  # run a master program which may
        # spawn slave programs on application PEs

or mpprun -n 4 spmd  # run a SPMD program using mpprun

pvm << EOF
halt
EOF  # start console to halt pvm

Note that the job will end as soon as the master or spmd program finishes. This means, for instance, that you can’t use a master that spawns some children and exits without waiting for them to finish.

### 3.2.1 Batch job limits

PVM processes spawned by the daemon inherit the limits of the NQS job. This allows a user to run multiple NQS jobs that use PVM, each with limits of the NQS job being run. Previous versions of PVM used the same daemon for multiple NQS jobs. A PVM job running in distributed mode under NQS has the following resource limits.

- First, since a pvm process is run on a command PE, it can be is limited by limits on command PEs. The `-l t` and `-l t` options specify the maximum CPU time on command PEs for any process of the request (`-l t`) and the cumulative CPU time of all processes of a request on command PEs (`-l t`).
- Second, a parallel pvm job running on application PEs it limited by limits on application PEs. The `-l p_mpp_t` and `mpp_t` options specify the maximum CPU time on application PEs for any process of the request (`p_mpp_t`) and the cumulative CPU time of all processes of a request on application PEs (`mpp_t`).
- The `-l mpp_p` option specifies the maximum number of application PEs required for the request (the default is one command PE).

Note that these options can be specified with the `qsub` command either within an NQS job script or from the command line.

### 3.2.2 When do you want the batch job to start?

In order to do interactive steering and visualization, you need to require a batch job to start when you are at your workstation. Most T3E’s are configured to have different queues at different times of the day and night, to handle different workloads. Depending on the time zone difference between you and the T3E, you may need to consider this queue structure and plan for your visualization accordingly. Indeed, one advantage of long-distance remote visualization, particularly as network speeds improve, is the ability to interact with night-running batch jobs from a daytime location.

NQS provides some control over job start times, allowing you to specify that a job will start no earlier than, say, 9 AM. It does not, however, allow you to specify that a job will start no later than a certain time. If you are running on a local T3E and want your job to start during the daytime, one possible way is to submit a job when you leave at night in expectation of it running the next day. As long as the queue wait time is usually less than a day, this is a workable solution. If a job doesn’t run during before, you should delete it before you leave and resubmit a new one. Otherwise there is no guarantee it will not run during your absence.

To specify that a job should start no earlier than a certain time, use the `-a` option for `qsub`. For example, the following job `foo.nge` will not start earlier than 09:00am on 03/18/98.
3.2.3 How to find out when your batch job has started.

NQS can send notification by mail when a job starts (or finishes), alerting you to attach the visualization or steering process. Simply use the `-mb` option to `qsub`. Something you may want to consider is having the mail activate a pager, so that you can be notified immediately even if you aren't at your terminal.

3.2.4 Problems related to checkpoint/restart and gang-scheduling

Some T3E's are configured with checkpoint/restart and gang-scheduling capabilities. Open sockets and pipes cannot be preserved across a checkpoint/restart on the T3E. Since PVM uses these mechanisms, PVM jobs cannot be checkpointed.

3.3 A Sample NQS Job Script and Output

The following sample NQS job script is to run pvmd, add hosts, and then run a pvm application of spawning two jobs on different partitions.

```
sample NQS job script
NQS job output
```

4. Startup of PVM Programs

For general steering, there are two possible ways to consider setting up the problem. As in Figure 2, lines show communication paths among PVM daemons and jobs running on both hosts. The following two set of codes illustrate two possible ways of running PVM programs on the T3E and your local workstation once PVM daemons are already started on both hosts.

4.1 Example I

In this example, a PVM program "master.f" (e.g. a steering/visualization program) run as the master program on your local workstation will spawn a slave program "xslave" on application PEs of the T3E, which is created by compiling t3slave.f (i.e. parallel application).

4.2 Example II

In this example, "t3eappl.f" is a parallel program, which is waiting for a connection from a steering/visualization process outside of the T3E while the program itself is doing a work. A steering/visualization program "steer.f" is started on your local workstation, a connection is actually established between two programs.

Note that in either example, by default only PE 0 of a parallel job on the T3E can communicate with a process outside the job through the daemon.

The virtual machine defined by a set of PVM daemons is "booted" in a two-step process. A master daemon is started on a master node (possibly by hand, indirectly by a program called the PVM
console or even by a batch job) and this master daemon starts the others. Note that only one daemon
is normally started on the T3E, which from the point of view of PVM is a single "host".

In order for a master daemon to start up other PVM daemons and jobs, normally PVM relies on
remote shell or execution commands, rsh and rexec.

5. T3E Implementation of PVM vs. Generic PVM

PVM is public domain software to enable a collection of heterogeneous computer systems to be
configured as a virtual parallel computer. User programs written using PVM use PVM library
routines (libpvm3.a) for starting and managing processes, message passing, and synchronization.
Users can control the execution location of specific application components. PVM transparently
handles data conversion for incompatible architectures, and other tasks that are necessary for
operation in a heterogeneous, networked environment.

Generic PVM is comprised of two main components: the PVM daemon process (pvm3d) and library
interface routines (libpvm3.a, libfpvm3.a & libgpm3.a).

Once a virtual machine is configured with a collection of computer systems (called hosts), a PVM
daemon must be started on every host and a job written using a PVM library spawned on each host.

5.1 Cray T3E (UNICOS or UNICOS/mk) Implementation of PVM

Like the generic PVM, the T3E implementation of PVM consists of two components -- pvm3d and
libpvm3.a. PVM on the T3E operates in two different modes to exploit its MPP architecture. --
stand-alone mode and distributed mode.

In stand-alone mode, the PVM library libpvm3.a is used without a daemon and performs only
message passing - no host or process management. PVM in stand-alone mode has functionality
similar to MPI. This mode allows you to use PVM to communicate among PEs within a multiple
PE process (that is, a single executable file), but it does not require a pvm3d but uses shmem calls to
communicate between PEs.

Using PVM in the distributed mode on the T3E requires running a PVM daemon so that PVM
processes communicate each other through the daemon. This mode allows you to spawn more than
one executable and allows your application to and also to configure a virtual machine that includes
the T3E and other systems. In distributed mode, PVM uses sockets for communication between the
T3E and remote machines, and between to T3E processes that were not started at the same time. A
given PVM task may have several sockets open at once: one to its local daemon and, optionally,
one or more to specific tasks with which it is communicating. For processes started in the same
pvm_spawn call, PVM communication uses relatively fast communication based on the Cray shmemb
library. While faster than sockets, there is still considerable overhead, so that this communication is
slower than PVM communication in stand-alone mode and much slower than MPI.

More fundamental issues can be found in the Cray MPT PVM manual and the PVM book.

5.2 Key Issues in the distributed mode

In this section, we describe key issues in using PVM on the T3E in distributed mode.
5.2.1 Cross-system Dynamic Groups

You can not form a dynamic group consisting of tasks from the T3E system and another system. You can not form a dynamic group consisting of tasks from more than one partition within the T3E even if you may run a pvm group server (pvmgs) on the T3E. But, you can form any group of PEs within one partition.

A predefined group (called the global group) consists of all PEs within the same partition. This can be used with communication and synchronization between the PEs. The name pvmall is used in a Fortran program for the global group.

5.2.2 Differences in functionality with stand-alone mode

When a PVM program is initiated, the program checks to determine if the PVM daemon is running. If it is not, the program assumes it is in stand-alone mode and certain PVM functions are not available and return errors if called.

5.2.3 Limits on Socket Communications

When the PVM daemon runs on the T3E, a PE communicates with the daemon and with PVM tasks outside its own partition. In theory, any PE can do so. But the T3E limits the number of open files per application and the number of open sockets in the system. So, if a parallel application running on a large number of PEs were to set up communications for each PE, it may hit either or both of these limits.

Socket communications are very slow, especially compared to the speed of communications between PEs. Because much of socket communication is single-threaded in the PVM daemon, the performance cost goes up as more PEs try to communicate at the same time.

For these reasons, by default, only PE 0 establishes communications with the daemon, and we recommend that you use PVM on the T3E in this manner. However, you can specify additional PEs by setting the PVM_PE_LIST environment variable, as follows:

```
setenv PVM_PE_LIST 0, 2, 4, 6
setenv PVM_PE_LIST all
```

Note that this setting should be done before a PVM daemon gets started.

6. Visualization with CUMULVS and AVS

CUMULVS implements messaging primitives that can be linked in as a library and called through an API (Application Programmers Interface) whenever data has to be transferred between two computers on a network. CUMULVS is designed by its developers to make adaptation to any application straightforward. Before any data can be transferred between machines, CUMULVS should be initialized by designating the structure of the data that is to be transferred which includes specifying the dimension of the data array (1-D, 2-D, 3-D), the global upper and lower bounds of the data array, the dimension of the logical processor decomposition, how each axis of the array is decomposed (block, cyclic), etc. After this is done, the program can transfer data to the remote machine by making a single call to CUMULVS in the main program loop. CUMULVS then returns with the status of the transfer. The frequency with which data is transferred (every time-step, every
other time-step, etc.) can be controlled from the remote workstation. This enables the user to initially start off with a low frequency of updating the data and increase the frequency if there is a need to monitor the process with a better time resolution at some point in the simulation. It should be mentioned that CUMULVS also offers capabilities for the user to interactively steer parameters from the remote workstation thus affecting the future course of the simulation.

The data that is generated on the remote computer and transferred to the local workstation through CUMULVS is visualized on the local workstation using AVS, a commercially available visualization package developed by Advanced Visual Systems. Of available graphics software, AVS provides some of the best possibilities for distributive processing and computational steering. AVS can run in a heterogeneous network where modules run on different hardware platforms. Communication is done using sockets. External applications can be integrated into a module to accomplish computational steering. Dynamic applications, which produce a series of data sets while they are running, are good candidates for AVS. Additionally, AVS has extensive 2-D and 3-D rendering, and animation capabilities with the ease of use of a visual programming interface - The Network Editor. Through the visual interface, all the programming in post-processing is reduced to making connections in a logical flow diagram with the individual components being dragged and dropped from an extensive menu. An example network is shown in Figure 3. With AVS, it is possible to input the data directly from another concurrently running process, without the need for input files, which is a central requirement for our scheme. We incorporate a CUMULVS-AVS interface written by the developers of CUMULVS, to input the data from our program (transferred through CUMULVS) to AVS.

6.1 Implementation and Testing of Remote Visualization

We implement the test programs for generating data in parallel on both the T3E’s at NERSC, and the T3E at MPG. Then for AVS visualization, we run in real-time on remote workstations both at NERSC and MPG. Here, we give details of the test implementation, and provide source code for those wishing to try it out.

6.1.1 Input Data

We create a scalar sample data set on a 3-D regular grid test for demonstration purposes. This data set is generated in the code fragment:

```plaintext
for(i=0; i<SIZE; i++)
for(j=0; j<SIZE; j++)
for(k=0; k<SIZE; k++)

DataArray[i][j][k] = \((\text{Count}\times5)\%255)\times(\sin(\pi\times i/\text{SIZE})\times\sin(\pi\times j/\text{SIZE})\times\sin(\pi\times k/\text{SIZE}))
```

where `DataArray` contains the scalar field on a 3-D grid whose indices are `i`, `j`, and `k`, `Count` is a variable which is incremented every iteration to ramp up the amplitude of the scalar field, `SIZE` sets the size of the grid along each dimension and `PI` is 3.1415. Figure 4 and Figure 5 show a 2-D slice and a 3-D rendering, respectively of this data set as it appears in AVS, visualized on the workstation. Figure 6 gives the AVS network, and Figure 7 shows a screen snap of the whole process.
6.1.2 The Test Program -- Remote T3E

The main program datagen.c runs on the remote T3E machine. It is responsible for generating the data and sending it to the local workstation for visualization. datagen.c can be run on any number of T3E processors. datagen.c includes the header files pvm3.h and stv.h which, contain the declarations for PVM and CUMULVS respectively. The main program performs the following:

Initializes PVM by calling pvm_init(...) 
Initializes CUMULVS by calling stv_init(...) 

Defines the data decomposition for CUMULVS by calling stv_decompDefine(...) 

Defines a field of data for CUMULVS with the previously defined data decomposition by calling stv_fieldDefine(...) 

Enters the main computational (data generating) loop in which in each iteration, it computes a new data set and sends it to the local work-station by calling stv_sendToF(...) 

It should be noted that the PVM daemon pvm3d should be running on both the local and remote machines before the programs explained here can be started-up. The status of the PVM daemon can be queried by using the PVM console pvm on the UNIX prompt.

6.1.3 Local Work-Station End

It should first be ensured that the PVM daemon pvm3d is running on the machine. The sequence of steps to be followed in starting up the visualization tool AVS is as follows:

Execute the data generator program datagen.c on the remote machine. 

Run AVS on the local machine and open up the Network Editor. 

In the Network Editor read the network stv_viewer, which is the CUMULVS-AVS interface, the source of data obtained from the remote machine. 

Build any post-processing of the data obtained in the visual programming environment of the Network Editor. Ultimately the output is piped into the Geometry Viewer or the Image Viewer. 

Specify the data field name to be viewed (datagen in our case) to stv_viewer in the appropriate list box provided. 

Finally, execute stv_viewer to start receiving the data from the remote machine and continuously displaying it as and when it arrives.

6.2 Performance

Despite the use of standard internet connections and production T3E queues, the test visualization performed remarkably well. As a rule of thumb, the local visualization update rate was about 2-5 times faster than when the data was sent across the world. (Typical ftp rates between the German workstation and the California T3E at 4pm German time were running at 5 - 28 Kbytes/s. At the same time, wholly within the German domain, ftp rates between the workstation and the T3E were
varying from 7 - 40 Kbytes/s.) These rates vary greatly with network traffic.

7. Conclusions

In this report, we discussed the use of remote visualization in real-time over standard network connections. We considered in some detail the obstacles faced in implementing remote operations in a production-computing environment that includes security issues, job scheduling, and varying communication rates. We used readily available software packages like CUMULVS for communication, and AVS for visualization, thereby cutting development time significantly. Additionally, the fault tolerance of CUMULVS was important for the implementation over the standard network.

We feel that remote visualization will become an important component of steering and interpreting large-scale simulations. It is certainly possible to transfer enough data to get a glimpse of what is going on in a simulation, and to be sure that hours of batch computer time are not wasted on improper parameters. Furthermore, through careful overlapping of communication to the workstation, with calculation on the T3E, one could further optimize the process so that the cycles on the T3E end are not wasted.

Acknowledgments

The authors gratefully acknowledge the support, humor, and software provided by the developers of CUMULVS, James Arthur Kohl and Philip M. Papadopoulos.

- Work at LLNL was sponsored by the United States Department of Energy under Contract W7405-ENG-48.
- Work at LBL was supported by the Director, Office of Computational and Technology Research, Division of Mathematical, Information, and Computational Sciences of the U.S. Department of Energy under contract number DE-AC03-76SF00098.
- Computer time provided by NERSC and MPG
- Graduate Student support for Shrihari Gopalakrishna provided by LLNL Laboratory Directed Research and Development

One of us (AEK) acknowledges the Max-Planck Institute for funding her visiting researcher position in Garching.

---

Lawrence Livermore National Laboratory

Disclaimers
UCRL-MI-

Authors: Youngbae@nersc.gov
Kooges@llnl.gov

206
Simulating and Visualizing Natural Flocking Behavior

Olaf Schnapaufl, Josef Schüle\textsuperscript{**}

Institute for Scientific Computing,TU Braunschweig, D-38009 Braunschweig

1 Introduction

The flight of a flock is a symbol for harmony and peace. How can a large number of birds coordinate their flight patterns and act in unison? There does not seem to be a lot of communication involved. Do all birds in a flock decide on a leader and how do they follow one bird in the mass of a flock? It does not seem likely. How if each single bird decides on its course and velocity only from its perceived environment? How much information is needed to create natural flocking?

Several models to create flock motion have come up over the last years. One by Susan Amkraut [AGK85] uses a force field. At each point in space a certain force is applied. Birds travel along the "phase portrait" of the force field. There are rejecting forces around each bird and around static objects. This is interesting as one obstacle avoidance scheme we implemented uses such force fields around obstacles, although our flight model is completely different (see section 3.1).

Another technique used before is "follow a leader". In this model a path for one bird is scripted and all other birds will just follow. This does not allow natural flock movement like splits on obstacles or stray birds.

Craig W. Reynolds [Rey87, CT96, LeB93] suggested a model that uses a set of simple urges like wanting to keep to the center of the flock without colliding with flock-mates. This model can simulate the behavior of flocks, herds and schools. As flocks are the most complex of these three our implementation focuses on flocks of birds. Schools of fish are essentially the same while herds can be simulated by restricting movement in the third dimension. Reynolds introduced two obstacle avoidance schemes, force fields around obstacles and active steer-to-avoid. His model is the base for the methods used in this work.

This paper describes the design and implementation of a model to simulate a flock's flight and its visualisation.

Each bird acts independently and computes its direction and velocity according to its perception of the environment, which contains other birds and a number of obstacles. There is no designated leader, the flock's behavior is the combination of all the birds' independent flight.

Another emphasis of this paper is to compare two types of obstacle avoidance schemes. One uses a set of force fields around each obstacle. The other is based on raytracing. It allows the bird to realistically look ahead and navigate its own flight path around obstacles.

We are using MPI [GLS94] or PVM [GBD94] on a Cray T3E 900 parallel computer for the computation of the flock. Using either RSH or PVM, the computer is connected to a visualization module running on an SGI O2.

The visualization module is based on VTK [SML98] (The Visualization Toolkit) and can be used to view the data computed by the T3E in real time and to render and store the frames for future use, such as creating videos. Scripted camera movement is supported by this setup. When connecting with PVM, one could also use it to influence the computation as it is performed. For example, the user could interactively alter the camera angle and position.

2 Modeling a Flock

Reynolds suggested a straightforward $O(N^2)$ algorithm:

"For each bird compute the influence every other bird has on its course."

The model takes into account a set of simple urges that each bird follows, using information about its current environment. These urges are:

---

\textsuperscript{*} O.Schnapaufl@tu-bs.de
\textsuperscript{**} J.Schule@tu-bs.de

267
- **Obstacle Avoidance**: To avoid flying into obstacles the bird has to find means to alter its course to steer clear. Two different models have been implemented and are explained in section 3.

- **Bird Avoidance**: One has to make sure the birds do not collide with birds around them. The avoidance vector computes as the weighted sum of the repelling forces asserted by each flock-mate:

  \[
  \text{avoidance} = \frac{\text{No. of Birds} \setminus \text{this bird}}{\sum_{i=0}^{\text{const}1}} \times \frac{\text{sensitivity} \times \text{delta}}{\text{distance}^3} \tag{1}
  \]

  → is used to mark vectors. \((\vec{a}, \vec{b})\) is the scalar product of two vectors.

  \text{delta} is the normalized vector from the current birds position to the other birds position.

  Perfect vision does not lead to reasonable results. As every bird could see every other bird the flock would not split and distance would not matter. \text{sensitivity} limits the view of the bird.

  For more details, see section 2.1.

  \[
  \text{sensitivity} = (\text{const}2)^{\text{distance}} \times \text{const}3 \times \left( \left( -\text{delta}, \text{direction} \right) + 1 \right) \tag{2}
  \]

  \text{direction} is the current course vector, normalized.

- **Flock Centering**: Each bird has the desire to steer towards the middle of the flock to keep the flock together. The center is determined according to the birds perception of the environment, not the absolute center of the flock when considering every bird.

  \[
  \text{centering} = \frac{\text{No. of Birds} \setminus \text{this bird}}{\sum_{i=0}^{\text{const}4}} \times \left( \text{delta}, \text{sensitivity} \right) \tag{3}
  \]

  with \text{delta} and \text{sensitivity} from above.

- **Velocity Matching**: This is the urge to fly as the others do, to match the own course and velocity with the flock. This computes as

  \[
  \text{matching} = \frac{\text{No. of Birds} \setminus \text{this bird}}{\sum_{i=0}^{\text{const}5}} \times \left( \left( \text{direction}_{\text{bird}}, -\text{direction} \right), \text{sensitivity} \right) \tag{4}
  \]

  with \text{sensitivity} from above and direction being the course vector of a bird with \(|\text{direction}| = \text{velocity}.

- **Fly to Goal**: At each time-step a vector pointing to a goal is added to keep the flock moving towards a certain point. This only gives the flock a push into the direction we want it to fly if the dynamics of flock motion permit. The actual flight path of the flock might diverge greatly from this direction, especially when encountering obstacles.

2.1 Sensitivity Function

A sensitivity function \(\text{const}_{\text{distance}}\), with \(0 < \text{const} < 1\) as suggested by Reynolds results in a decrease of the influence of other birds with distance, uniformly in all directions. This models birds with a spherical symmetric perception.

The results where inadequate. If the influence of all birds around is identical, the centering and avoidance equations lead to a stable but spherical flock. It seemed necessary to refine the simulation of natural vision. Vision does not only decrease with distance but the angle is also limited. The sensitivity function was adapted to take this into account (equation (2)). The resulting visual range is plotted in figure 1. The birds are oblivious to what is going on behind them and get more sensitive to objects in front of them.

The performance of this formula is much improved over the uniform direction formula.
2.2 Urges and the Priority Queue

The real world imposes some limits that have to be considered. As beings with limited amount of energy available, their velocity, the acceleration and the change of direction per time-step is limited.

Still, just adding up the forces and adding them to a direction is inadequate to create movement that looks natural. Each of the urges might create a force that is in itself a valid suggestion to alter our course and speed, while the sum of all these urges is not a good idea. An example might be a flock flying through Manhattan, where the course "north" and the course "west" might be a good suggestion, while "northwest" results in a crash into a skyscraper.

The solution to this dilemma is deciding on priorities. Each bird has a maximum acceleration to use at each time step. A queue is set up where the most important urges are processed first, allowing them to use up some or all of the acceleration. The queue is exit when all acceleration has been used up. This queue is called the "Priority Queue".

First avoidance, centering and matching is computed, then the Priority Queue (figure 2) is entered.

![Diagram](image)

Figure 2. Computing the acceleration vector with the Priority-Urge Queue
After each step in the queue we test if the acceleration so far is less than the allowed maximum. If it is, we add the next vector to the 

direction. With this technique we allow the most important urges to use up the maximum acceleration and disregard less important urges for the moment. If we are close to crashing into an obstacle we want to use all the force we can to steer away and forget about staying close to the flock for now.

If the requested acceleration after a step in the queue is greater than the maximum acceleration, the queue is exit and the direction and velocity are passed on to the module imposing limits on certain values, to cut back the excess acceleration produced by the last step to the allowable maximum. The effect of this procedure is that the last step in the queue is allowed to influence the direction, and to influence the acceleration to the permissible limit. If the queue would be exit and the last action that created the excess acceleration would be cut back, a situation could arise where a large amount (or all) available acceleration would not be used, for example when the first urge in the queue would request more than the allowable maximum acceleration.

The computed acceleration is used to modify the current velocity, which cannot exceed a defined maximum velocity.

It is necessary to limit the changes in direction per time step. To accomplish this the angle between the new course vector $\vec{B}$ with velocity $|\vec{B}|$ and the old course vector is limited.

2.3 Looking through Obstacles

In Reynolds algorithm (section 2) birds do see each other even through obstacles. Thus flocks that have split on an obstacle try to stay together as a single flock and stick to the two lateral faces of the obstacle as they are not aware of being separated.

To improve this, a ray from each bird to the other bird is traced. If this ray intersects an obstacle before reaching the other bird, that bird is currently invisible and its influences is ignored (see section 3.2 for the description of the raytracing routines used).

Although this adds a lot of extra computation, the check if a bird can actually see another bird before being influenced by it is vital to the quality of the model.

3 Obstacle Avoidance Methods

Different approaches can be taken to include obstacles. The model has to prevent birds from crashing into obstacles, allow a flock to alter its course and enable flocks to split when they approach an obstacle.

3.1 Force Fields

As suggested by Armkraut [AGK85], obstacles can be modeled as emitters of force fields. At every discrete time step each bird is influenced by the field, pushing the birds course away from the obstacle. Obviously, it is important that the force decreases with distance from the obstacle and that birds flying parallel or away from the obstacle are not affected.

The radial symmetric force field of a point obstacle is easily created. The force field of a box is more complicated. The first model implemented a single force field pushing straight away from each lateral face. This works in most cases, but fails if a bird approaches the face directly from above, as this bird is only slowed down. Superimposing a second force field that splits each face into four zones (figure 3) pushes the birds course to the closest edge.

The computation of the forces is costly as the relative position of each bird to the box and the appropriate zone of the face has to be found out. Two forces have to be calculated and it has to be taken into account that the force to push a bird around a big obstacle when the bird is close to the middle of the face has to be much greater than for a small obstacle.

Still the quality of this model is rather poor. The dependence on the obstacles size makes it difficult to find parameters that allow the whole flock to clear any obstacle. In addition, birds do not see a pathway through neighboring boxes with overlapping force fields.
3.2 Looking ahead: Raytracing

Reynolds [Rey87] suggested a model he calls steer-to-avoid that has birds actively looking ahead and finding out if they are on a collision course with an obstacle. If this is the case the bird has to find the closest way around it. We utilize methods from raytracing to look ahead and find tangent to an obstacle, steering the bird clear of the obstacle.

The algorithm to find an intersection of a ray and a sphere can be found in [Gla89]. The tangent to the sphere closest to the intersection point can be computed directly by slightly altering the intersection algorithm.

To calculate the tangent to a box obstacle in direction of a birds flight three basic cases have to be distinguished: (figure 4):

![Figure 4. Computing the new direction when on collision course with a cube (Look-ahead)](image)

Let $x$ and $y$ be the length of the face of the box we intersect. If $x > 2 \times y$ the face is split along the x direction, if $y > 2 \times x$ along y. If neither is the case (so the face is nearly quadratic) one splits the face into 4 zones along its diagonals. Then it is determined which zone the intersection is in and the tangent to this zone is computed. This tangent is the new desired course and the bird gradually tries to change its current course to match it. The strength of the course change will be scaled by $\text{const}_{\text{distance}}$.

This obstacle avoidance scheme, although computationally expensive, results in much improved behavior. The birds look ahead and alter their course to lead them around the obstacles. They tend to surprise their creators with things like refusing to fly through a maze of bars and deciding to simply take a detour around it. The flocks realistically handle a variety of obstacles built from these basic sphere and box shapes.

Using raytracing to avoid obstacles by computing the closest tangent to them also allows us to consider more realistic obstacles, e.g. created from bezier patches. This might significantly slow down obstacle avoidance as their intersection algorithm is much more expensive. Trying to compute force fields around bezier patches would be even more difficult.
4 Implementation

The system consists of the visualization front-end using VTK[SML98] that renders a picture of the scene from a set of data, and the compute module that creates the data. The modules can be connected in a variety of ways, one being the standard Unix way of stdin/stdout (e.g. via rsh) and another via PVM. The compute module can use multiple processors to do its work by utilizing either MPI or PVM.

4.1 The Visualization Module

VTK (The Visualization Toolkit) is a powerful library to visualize data. It is available for free as source and can be compiled on most UNIX systems. It can utilize a variety of graphics libraries, OpenGL being one of them. We used an SGI O2 running Irix 6.3 as visualization front-end.

VTK provides efficient high level routines for visualization. The whole module consists of less than 600 lines of C++ code with most of it being required for parsing the data and PVM connectivity.

The basic principle of VTK is to connect a set of data sources with a renderer through a pipeline of filters and mappers that transform the data to achieve the desired result. In this case the data sources are the primitives used to depict obstacles and birds and the filters read in position data and place the primitives on the positions described by that data. After connecting all objects of this pipeline the rendering method in the render object is invoked, reading the data from the pipeline and creating a picture. The render object also provides methods to save images that are later reused to create movies.

The data describing a frame of the flight can be fed to the visualization module from stdin or it can be asked to spawn the compute module via PVM and receive its data through a PVM connection. The advantage of this is the possibility to send information to the compute module during the computation, e.g. to interactively influence the flock during its flight.

The scenes are described by specifying some general data like the various constants used to influence flock behavior, obstacle position, shapes and goals for the flock to fly to and the camera position at certain frame numbers. The camera can also be attached to a bird to allow the viewer to fly along inside the flock.

It is possible to watch the computation as it is done or to save the rendered frames to disk for later use.

4.2 The Compute Module

The compute module implements the model described above. We provide several C++ classes for birds, flocks, point and box obstacles. The flock is moved in discrete time-steps and the resulting data is output for the visualization module.

As this algorithm is computational expensive and has an \(O(N^2)\) behavior with the number of birds, it is necessary to parallelize it to get a realistic real time visual output. Parallelization is implemented with both MPI and PVM on a Cray T3E-900 as compute server.

The nature of the algorithm allows an easy way to split the load to several PEs. The Master is connected to the visual device either via stdin/stdout and rsh (MPI) or via a PVM socket. It broadcasts the flock state to all PEs. Each PE then simply selects an equal portion of the flock irrespective of their location or sensitivity and computes the new course and velocity for each bird in its portion, using the influence of the whole flock. The master collects all information from all birds, sends the necessary information to the visualizer. These steps are repeated.

No effort was undertaken to implement more effective algorithms, like particle-in-cell, to reduce the asymptotic running time as it turns out, that the quadratic term in the asymptotic expansion has a quite small pre-factor (compare equation 5).

5 Discussion

In the following sections performance measurements are presented. A number of benchmark runs was done to determine the behavior of the system. The time to compute a single frame was measured for different numbers of PEs and birds in the flock.
One aspect is to compare the MPI and PVM performance on the T3E (section 5.1) and finding an equation to describe the behavior of the model (section 5.2) to allow a approximation of the runtime for any number of birds and PEs and to discuss the influence of the sequential and parallel parts of the algorithm.

Another interesting topic is the influence of the network connection (section 5.3), the visualizer (section 5.4), and the obstacle avoidance scheme (Force Field or Raytracing, section 5.5).

The second and may be more important class of results (section 5.6) are the visible results of flock-like movement and obstacle avoidance without the need to specify a path for the flock or each bird. This is usable for computer animation when combined with an engine to place photo-realistic birds on the positions determined by the compute module. The obstacles can be enhanced with any obstacle a raytracer can compute. This is a strong advantage as algorithms and tools to create and compute scenes with raytracers are readily available.

5.1 Comparing PVM and MPI

To compare the core PVM-Daemon and MPI performance on the T3E the computation module was run without a visualizer but connected to a dummy program on the T3E, so external network delays and/or disk I/O would not slow down computation. Figure 5 show the result for small number of birds. Larger numbers of birds result in straight parallel lines.

As expected, MPI outperformed PVM by a constant factor depending on the number of birds. The graphs also show that the computation has an inherently sequential part as \( \lim_{P E s \to \infty} = \text{const} > 0 \)

![Figure 5. Execution time of flock sizes 150, 300 and 600 on several PEs, comparing MPI and PVM on the CRAY T3E](image)

5.2 Behavior of the System

The data in figure 5 together with the results for larger number of birds were used to find a formula to describe the execution time depending on the number of PEs and birds. Graphs where
extrapolated through the measured points. Equation 5 is a very close approximation of the system behavior, the measured results differ less than $10^{-5}$ seconds.

\[
f(b, p) = 0.0645b + \frac{0.759b}{p} + \frac{9.04 \times 10^{-3}b^2}{p}
\]  

(5)

$b$ is the number of birds, $p$ the number of PEs and $f(b, p)$ the expected execution time in milliseconds per frame.

This equation allows to compute the expected time to compute a scene with an arbitrary number of PEs and birds. The influence of the visualization module will be discussed later in section 5.4.

As could be expected the computation has a dependency on $b$ (the number of birds) which is the sequential part communicating the results to the master process and to the visualization module (which would be much higher when rendering and saving frames), on $\frac{b}{p}$, the time each PE spends communicating and setting up its fraction of the data, and on $\frac{b^2}{p}$, the time used in the $O(N^2)$ part of the algorithm, doing the core flocking computation. The pre-factor to this term is small, so no algorithmic effort was undertaken to reduce the complexity of this term.

5.3 Network influence

A network connection to the visualization module could be a bottleneck, as the master process on the T3E has to send the current set of positions to it after every frame. This bottleneck would only appear if the time needed to send the data over the network is significant.

![Figure 6. Execution time of flock sizes 150, 300 and 900 on several PEs, comparing PVM on the CRAY T3E connected to a dummy program on the T3E, and connected to the visualizer on the SGI.](image)

Figure 6 shows that this is not the case. The bandwidth of 10 Mbit Ethernet, which was used to connect the SGI O2 and the T3E, is more than enough to transport the little positional data.
It does not significantly increase the time spent on sequential operations over the time needed for a local connection on the T3E. Results for 900, 2000 and 4000 birds where omitted as it did not make a measurable difference at all if the communication happened locally or over a network link.

5.4 Influence of the Visualizer

The data sent from the compute module to the visualizer is not buffered, but instantly rendered (and, when requested, the images are saved to disk). This sequential and time consuming operation could limit the achievable speedup. Rendering time depends on the number of birds, but only marginally. Image size and saving to disk are significant factors. All tests where run without saving at a resolution of 400x400 pixels, figure 7 shows the measured results.

![Graph showing time per frame in microseconds vs. 1/PES for different flock sizes with and without rendering.](image)

**Figure 7.** Execution time on several PEs with and without rendering the results with the visualizer for flock sizes 150, 300, 600 and 900 employing PVM daemon.

For up to 600 birds the maximum speedup is reached at about 10 PEs, sequential rendering becomes the limiting factor.

For 2000 and 4000 birds the time spent in the $O(N^2)$ flocking algorithm is still large enough that the visualization module finishes its work before a new frame has been computed, even with 20 PEs, the maximum number available for this benchmark. The measured times for 2000 and 4000 birds did not differ if the renderer was active or not, so the results are not plotted in figure 7.

5.5 Influence of the Obstacle Avoidance Scheme

Raytracing is a time consuming algorithm. As the Look-Ahead obstacle avoidance scheme (section 3.2) uses raytracing and the Force Field scheme (section 3.1) just some geometry and a lot of case switching it could be expected that Force Field is faster (figure 8). This is not the case. For all measured situations Look-Ahead slightly outperformed Force Field. This can be explained with the compact and fairly optimized raytracing algorithms that are well understood in the computer graphics world. Finding the new direction for a bird on collision course is easy. The tangent to
the obstacle closest to an intersection can be computed efficiently. The box and sphere obstacles used also have fast intersection algorithms, using obstacles created from beizer patches might significantly slow down obstacle avoidance.

![Graph](image)

**Figure 8.** Comparing the speed of the Look-Ahead and Force Field obstacle avoidance schemes employing MPI and rsh coupling to the visualizer.

### 5.6 Watching the Flocks fly

Still frames of the flocks are depicted below, a video is available. Mpg-Videos, more still frames of the flocks, this paper and an extended, more detailed paper on this topic are available on http://www.sc.cs.tu-bs.de/olaf/.

### References


Figure 9. Splitting on a Quadratic face of a Box Obstacle with Look-Ahead, side and frontal view

Figure 10. Finding their way through 3 obstacles, splitting the flock on the way through

Figure 11. Flocks rejoining after having split on obstacles
INVESTIGATION AND IMPLEMENTATION OF A
NUMBER OF DIFFERENT LOAD BALANCING
STRATEGIES IN THE UK MET. OFFICE’S UNIFIED
MODEL

P. Burton
UK Meteorological Office, Bracknell, Berks, UK
R. Ford
University of Manchester, UK
D. Salmond
SGI-Cray, UK

September 5, 1998

Abstract

The UK Met. Office has an 880 processor Cray T3E which is used for running
both an operational weather forecasting service and an extensive climate prediction
programme. Both of these use the Unified Model (UM), a highly integrated modelling
code, capable of atmosphere and ocean forecasting as well as data assimilation.

High resolution weather forecast configurations generally use 144 processors and
a very large input data set, while climate integrations more usually utilise a smaller
number of processors and smaller input data sets.

It is essential for the UM to be scalable, to allow an effective use of the T3E with
all the different UM configurations. One of the major obstacles to good scalability is
load imbalance.

We will present work carried out in three areas of the model to address significant
load imbalances:

- The data assimilation scheme, where load imbalance is caused by observations
  being non-uniformly spread around the globe.
- The short wave radiation parameterisation, where load imbalance is created by
  half of the globe always being in darkness.
- The convection parameterisation, where load imbalance occurs due to high amounts
  of convective activity in equatorial regions, and very little around the poles.

The different techniques used to remove the load imbalance in each of these areas
will be described and compared. Performance results will be shown.
1 INTRODUCTION

1 Introduction

The UK Met. Office's (UKMO) suite of modelling codes known as the Unified Model (UM) has been in use for both operational Numerical Weather Prediction (NWP) and climate prediction since 1991. The UM consists of over half a million lines of Fortran 77 code which include atmospheric and oceanic prediction models (which may be coupled together), and data assimilation for both these models. The models can be run in many different configurations and resolutions. Table 1 shows the most commonly used configurations at UKMO.

<table>
<thead>
<tr>
<th>Application</th>
<th>Mode</th>
<th>Grid(Km)</th>
<th>Dimensions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Global NWP</td>
<td>Atmosphere</td>
<td>60</td>
<td>432x325x30</td>
</tr>
<tr>
<td>UK Regional NWP</td>
<td>Atmosphere</td>
<td>10</td>
<td>146x182x38</td>
</tr>
<tr>
<td>Global Climate</td>
<td>Atmosphere</td>
<td>300</td>
<td>96x73x19</td>
</tr>
<tr>
<td>Ocean Climate</td>
<td>Ocean</td>
<td>40</td>
<td>290x144x20</td>
</tr>
</tbody>
</table>

Table 1: Commonly run configurations of the UM.

In contrast to many other forecasting centres, which use semi-implicit spectral models, the UM uses an explicit grid point formulation in both its global and regional configurations. The scientific formulation of the model can be split into two sections; the dynamics which keep the various fields in dynamical equilibrium, and the physics which parameterise the physical aspects of the atmosphere (for example solar heating, rainfall etc.). The various physics packages together contribute the majority of the timestep by timestep CPU cost, so it is important to ensure they are performing at maximum efficiency.

The UM code was originally developed on a vector parallel shared memory architecture, where it scaled reasonably well over a small number of processors. Over the last five years, the code has been developed to allow it to run on distributed memory architectures, and it is now run operationally at UKMO on a 880 processor Cray T3E. The parallelism is achieved by a horizontal regular domain decomposition, and table 2 shows the typical decompositions used. This form of parallelism means that each processor is responsible for forecasting a vertical column of atmosphere above a rectangular subdomain of the full model area.

<table>
<thead>
<tr>
<th>Application</th>
<th>Dimensions</th>
<th>Decomposition</th>
<th>Processors</th>
</tr>
</thead>
<tbody>
<tr>
<td>Global NWP</td>
<td>432x325x30</td>
<td>9x16</td>
<td>144</td>
</tr>
<tr>
<td>UK Regional NWP</td>
<td>146x182x38</td>
<td>12x12</td>
<td>144</td>
</tr>
<tr>
<td>Global Climate</td>
<td>96x73x19</td>
<td>6x6</td>
<td>36</td>
</tr>
<tr>
<td>Ocean Climate</td>
<td>290x144x20</td>
<td>1x24</td>
<td>24</td>
</tr>
</tbody>
</table>

Table 2: Typical decompositions used on the UM.

Using such numbers of processors, it is essential to minimize any load imbalance, which can seriously affect scalability. In practical terms, a lack of scalability reduces the amount of useful work that can be done on the machine within a given time and delays the production of essential forecast products.
2 DATA ASSIMILATION

1.1 Load Imbalance in the UM

This paper will concentrate on three of the potentially most damaging areas of load imbalance in the UM – where computationally expensive code is badly balanced across processors. Each of these three areas produce load imbalance for different reasons, and it has been found necessary to use a different solution for each case. These three solutions together can cover a wide range of load imbalance problems, and certainly within the UM, result in a significant improvement of scalability and absolute performance.

These three areas are:

1. Data Assimilation
   The data assimilation takes raw observations from many sources; from human observers to automated aircraft observations to satellite data, and integrates this information, together with a “first guess” field from the previous forecast, to produce a set of analysis fields from which to start a forecast integration. The processing of these observations can have a significant load imbalance.

2. Short Wave Radiation
   Shortwave radiation from the Sun is absorbed by the atmosphere and at the earth’s surface, and the parameterisation of these processes is very computationally expensive. These calculations are only carried out at daylight points - this means that for a global model, half the points have no work to do, so this is a serious load imbalance.

3. Convection
   The convection scheme parameterises the transport of heat, moisture and momentum due to cumulus convection. On a global scale, there is vigorous convection high into the atmosphere around the equator, which reduces towards the pole where there is typically very little convection at all. On a local scale, the amount of convection can depend on the passage of weather systems or the relative temperature of land and sea to the atmosphere above. So, for a number of reasons, there can be a bad load imbalance in the convection scheme.

2 Data Assimilation

2.1 Background

The data assimilation scheme is a subroutine of the UM, and is called using the data structures already existing within the main model. For this reason, it was natural to parallelise it using the same two dimensional domain decomposition as is used throughout the rest of the model. This means that each processor is responsible for processing all the observations in its subdomain of the atmosphere.
2 DATA ASSIMILATION

2.2 Description of Load Imbalance

2.2.1 Causes

The most expensive part of the data assimilation is the “horizontal analysis” (HORINF). This takes observations from various sources, and incorporates them into the meteorological fields. HORINF exhibits a serious load imbalance, with some processors having ten times as much work as others do. This is for three main reasons:

1. The observations are spread inhomogeneously around the globe, for example:
   - Ground observations tend to be concentrated in populated land masses, such as Western Europe. On a global scale there is a big load imbalance between the Northern hemisphere with relatively large land masses, and the Southern hemisphere which is mostly ocean.
   - Many airliners take automatic observations as they are flying. However, airliners tend to follow well defined aviation routes between population centres, leading to narrow “tracks”, radiating from major airports.
   - Satellite observations are highly concentrated in certain regions, depending on the path of the satellite relative to the surface. Some satellites are geostationary - meaning they are stationary with respect to the earth’s surface, while others are polar orbiting, with a typical orbit frequency of a 100 minutes, constantly moving over different parts of the earth’s surface.

2. The area of influence can be different for each observation. The larger the area of influence, the more computation is required to assimilate the observation into the meteorological fields, as more points must be updated.

3. The UM uses a regularly spaced latitude-longitude grid, which results in a convergence of points towards the poles. This means that a given area of influence will contain a much larger number of points if it is at high latitudes than if it is an equatorial region. As it contains more points, it will require more computation.

2.2.2 Demonstration

Figure 1 shows the amount of time spent performing the HORINF calculations on each processor before any load balancing is applied to the code. From this it can be found that there is a 1 : 11.85 ratio between the mean computation time per processor and the maximum time on any processor - a very serious load imbalance.
2.3 Load Balancing Strategy

2.3.1 Background

As was described in section 2.2.1, there are many contributing factors to the load imbalance in HORINF, so it would be difficult to write an “intelligent” load balancing algorithm that could predict where the load imbalance would occur and move the data around using this prediction.

2.3.2 Description of algorithm

What is used instead is a pseudo-random distribution of observations to processors, which results in the workload being distributed in a close-to-optimal manner.
3 SHORT WAVE RADIATION

Observations are read in from disk, each processor receiving the observations lying within its horizontal subdomain. At this stage some processors will contain many more observations than others do. A first round of redistribution takes place, which results in each processor containing the same number of observations. Some observations have a larger area of influence than others (meaning more points must have an increment calculated). To ensure that these observations are well spread across the processors a second stage of redistribution is applied, which uses a round-robin scatter to send each observation to a successive processor.

Each processor then assimilates the observations that it has been allocated, calculating an increment field to the meteorological fields. When each processor has a complete increment field calculated, a global sum is performed to sum all the increment fields together to form a final increment field that contains the effects of all the observations. This increment field is then scattered back to the original horizontal decomposition, so each processor can add the increment to its subdomain of the meteorological fields.

2.3.3 Demonstration

Figure 2 shows the amount of time spent performing the HORINF calculations on each processor after the load balancing has been applied to the code. From this it can be found that the ratio between mean and maximum times per processor is now 1:1.1, showing only a very low level of load imbalance. The mean time has increased by 10%, indicating the overhead incurred by the load balancing algorithm itself, but the maximum time (which is the observed overall time) has decreased to just 10% of the original time.

3 Short Wave Radiation

3.1 Background

Short wave radiation is a physics routine in the UM which calculates the effects of radiation from the sun. It uses the same two dimensional regular domain decomposition as the rest of the model. Each processor calculates which of its points are in daylight (a relatively trivial computation) and performs computation at these points. For efficiency, short wave radiation gathers a number of the relevant fields (at daylight points) into local work arrays and scatters the results.

3.2 Description of Load Imbalance

3.2.1 Causes

As short wave radiation calculates the effects of radiation from the sun, points that are in darkness do not need to participate in this computation.
HORINF : Load Balanced

![Graph showing load balance](image)

Figure 2: Load Balanced version of the Data Assimilation (64 Processors)

In global modelling half of the total points will be in darkness (as half of the earth is always in darkness) and for a reasonable number of processors there will be some processors in the standard 2 dimensional partition with all of their local points in daylight and a number of processors with all of their local points in darkness. As half of the total points are in darkness and each processor has approximately the same number of local points, the average number of sunlit points per processor is half of its local points. Therefore, processors with all of their local points sunlit perform twice the average amount of work giving a 50% load-imbalance.

For regional modelling, the load balance is not so serious, as for most of the time either all the processors are in darkness or all the processors are in daylight. It is only at dawn and dusk that a load imbalance occurs. This situation worsens for regional areas closer to the poles during mid winter and mid summer, where there may be a large difference in the time spent in sunlight between the Northern and Southern edges of the region.
3 SHORT WAVE RADIATION

3.2.2 Demonstration

Figure 3 shows the time spent in the short wave radiation routine through a 3 day period of a global model. The top line shows the maximum time spent by any processor, the lower line shows the minimum time, and the middle line shows the mean time over all processors. This is the ideal time the code should take if it were perfectly load balanced. The graph shows a load imbalance of approximately 50%, as one would expect (from the discussion in the previous section).

![Short Wave Radiation Load Balance: 144 PEs](image)

Figure 3: Load Imbalance in the Short Wave Radiation (144 Processors)

3.3 Load Balancing Strategy

3.3.1 Background

Unlike the data assimilation, where it was virtually impossible to predict the load distribution beforehand, the short wave radiation load can be predicted by counting the number of sunlit points. As each sunlit point requires the same amount of computation, redistributing these points across the processors so that each processor has an equal number of sunlit points should load balance this routine. It is, of course, very important to try to minimise the extra computation and communication costs as they will affect the load balance.
3 SHORT WAVE RADIATION

3.3.2 Description of the algorithm

Re-distributing the sunlit points so that each processor has an equal number of sunlit points is a relatively simple task, however doing so with the minimum number of messages is a much more difficult task.

A number of algorithms have been devised and tested which attempt to minimise the number of messages required to re-distribute the sunlit points, the best of these has been implemented in the UM and is described below.

1. The total number of daylit points over all processors is summed, and the mean value per processor is calculated. From this, each processor can find the number of points by which it is in excess or deficit from this mean value. The example in figure 4 shows this for a four processor run.

<table>
<thead>
<tr>
<th>Number of sunlit points</th>
<th>P0(1103)</th>
<th>P1(1795)</th>
<th>P2(217)</th>
<th>P3(1133)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total:</td>
<td>4248</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Mean:</td>
<td>1062</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Excess/deficit</td>
<td>P0(+41)</td>
<td>P1(+733)</td>
<td>P2(-845)</td>
<td>P3(+71)</td>
</tr>
</tbody>
</table>

Figure 4: Step 1 - Counting points and calculating mean

2. The list is sorted so that the processor with the largest excess of points is at the head of the list, and the processor with the greatest deficit of points is at the tail of the list, as shown in figure 5

<table>
<thead>
<tr>
<th>Ranked list</th>
<th>P1(733)</th>
<th>P3(71)</th>
<th>P0(41)</th>
<th>P2(-845)</th>
</tr>
</thead>
</table>

Figure 5: Step 2 - Sort list

3. The following process is repeated until all processors have been removed from the list or all the remaining processors are under-loaded. The process is illustrated in figure 6.

- Points will be transferred from the processor at the head of the list to the processor at the tail of the list.
  The number of points transferred is defined as \( \text{MIN}(|\text{ABS}(\text{head})|, |\text{ABS}(\text{tail})|) \).
- The number of points remaining for the under or over-loaded processor is checked to see if there is an exact pair to match it, in which case points are transferred.
- Any members of the list having no excess or deficit points are removed from the list.
- The list is resorted.

There is now a communication map defining which processor(s) each processor will send or receive from. The work arrays mentioned in section 3.1 are dimensioned to be large enough to contain the mean number of sunlit points. Any excess sunlit points are sent to their
destination processor(s), and the remaining points are packed into the work arrays. If a processor has a deficit of sumit points, then it will receive some excess points from other processor(s) and pack these into the work arrays. When the communication is completed, each processor passes the work arrays to the short wave radiation routine. When the routine has completed results are passed back to their processor of origin.

### 3.3.3 Demonstration

Figure 7 shows the result of applying this load balance code. The three lines are now very close together, indicating very little load imbalance, and virtually no increase in the overall mean time to perform short wave radiation - showing that the extra computation and communication overhead is of this scheme is negligible. The load balance code therefore enables the short wave radiation computation to complete in up to half the time of the previous version, a significant saving in one of the UM’s most expensive physics routines.

### 4 Convection

#### 4.1 Background

The convection, like the short wave radiation is one of the more expensive UM physics packages, and also uses the standard two dimensional regular domain decomposition. Unlike the short wave radiation, each processor has no way of knowing which points will require convection calculations to be carried out, so passes its entire fields to the convection science code to process.

The convection science code allocates a large number of dynamic memory arrays during its computations. To reduce the amount of memory required at any one time, the calling code allows the input data to be split into a number of segments, each one of which is passed to consecutive calls of the science code.
4.2 Description of Load Imbalance

4.2.1 Causes

Convection is essentially caused by solar heating of the ground or sea surface, causing warm air to rise. The amount of heat and moisture in the atmosphere also has a large bearing on how vigorous the convection is. This means that equatorial regions, where there is very significant amounts of solar heating, and often high humidities, tend to exhibit very vigorous convection which extends high into the atmosphere. At mid-latitudes, where this is less solar heating, and less available moisture, the convection is less vigorous, and is mostly driven by the passage of weather systems. Towards the poles, convection dies out completely.

Some processors, especially around the equator will have a high computational workload, those at midlatitudes will have a variable workload, depending on the prevailing meteorological conditions in their subdomain, and processors around the poles will have virtually no work to do.

4.2.2 Demonstration

Figure 8 shows the time spent in the convection routine through a 3 day period of a global model, and is of the same format as the graph for the short wave radiation (see section 3.2.2). Again, the graph shows a clear load imbalance, with some processors taking eight time longer than others to perform their computations. The most expensive processors take...
about twice as long as the mean time, indicating that a good load balance could halve the time spent in this routine.

![Convection Load Balance: 144 PEs](image)

Figure 8: Load Imbalance in the Convection (144 Processors)

### 4.3 Load Balancing Strategy

#### 4.3.1 Background

A load balancing solution such as used in the short wave radiation is not directly applicable, since the workload at each point is not known before the convection is called\(^1\).

Instead, we have chosen to make use of the segmenting ability (see section 4.1) already available in the code. This breaks the data down into chunks that can be passed around between processors to achieve better load balance. However, there is a drawback to this technique. The large amount of dynamic allocation that takes place inside the convection science code produces a significant overhead (the allocation and deallocation of a large number of arrays takes a considerable amount of time), and calling the code multiple times for the different segments magnifies this overhead, and can easily dominate the cost of the computations. This limits the number of segments that can be usefully utilised to around five.

---

\(^1\)A possibility may be to use the time taken to perform the convection computation at the previous timestep as a guide to the cost per point, and use this to implement a similar solution to the short wave radiation load balancer, but this has not yet been attempted.
4 CONVECTION

4.3.2 Description of the algorithm

The first attempt at load balancing the convection used an approach similar to that used in the data assimilation code; each processor scattered its segments of data onto different processors. However, because only a small number of segments can be used, this was found not to produce a very optimal load balance, and it was not obvious what was a good algorithm to use to decide where to send each segment of data from a particular processor.

The solution finally adopted was developed from this, and can be thought of as a modified "task farm" approach. In the typical task farm, a master processor is responsible for handing out data to subprocessors as they complete their previous tasks. In this modified version, each processor is responsible for looking for work to do, either using its own local data, or if all that has been computed, then using remote data on another processor. For this reason, I prefer to describe this solution as the "work snoopers", each processor snooping other processors to see if they have any available work to do.

The "work snoopers" makes full use of the Cray SHMEM one-sided communications library, which allows a processor to "snoop" into another processors data, to see how far it has got, and if necessary shmem.get data back to itself, and then shmem.put the results back to the remote processor's memory, without the remote processor ever having to participate.

The algorithm is as follows:

1. Each processor sets up a counter, next_seg which contains the next segment of data on that processor which is to be processed.

2. Each processor loops over all processors, starting with itself, and looks at next_seg on that processor. If next_seg is not greater than the total number of segments, then next_seg on that processor is incremented. If the processor is the local processor, then the convection science is called with the local data. If it is a remote processor, then the data must be shmem.getted into work arrays, which are passed to the convection science, and the results then shmem.putted back to the remote processor.

3. If a processor has looped over all processors, and not found any with any segments remaining to be computed, it exits to a barrier, where it will wait until all the other processors have also finished and reached the barrier.

4.3.3 Demonstration

Figure 9 shows the result of applying this load balance code. It can be seen that the load balance is not as good as that obtained by the short wave radiation - this is because the number of segments used is relatively small (five). If a larger number of segments is used, the load imbalance can be almost completely removed, but at the expense of the maximum, mean and minimum times all increasing due to the overhead incurred by the dynamic memory allocation. It has been found by experimentation that setting the number of segments to five gives the best compromise between load imbalance and cost of dynamic allocation.
Despite these problems, a significant performance improvement has been gained by load balancing the convection. The time taken has reduced to almost half of its original value, with the mean time showing only a very small increase, almost entirely due to the dynamic memory allocation overhead.
5 SUMMARY

5 Summary

This paper has presented three different methods to achieve better load balance in UM code, and these methods are also applicable to other problems with similar load imbalance characteristics. Different techniques are required for different underlying properties of the load imbalance:

1. Where the load is randomly scattered over processors, or difficult to predict the imbalance can be removed by:

   (a) **Round-robin** distribution of work over processors. This is suitable where the work can be broken down into a large number of independent chunks.

   (b) **Work-snooper** algorithm, allowing processors to search for work to do on other processors when they have completed their own work. This is more suitable where the work can only be broken down into a small number of independent chunks.

2. Where the pattern of load over processors is easily predictable, the imbalance can be removed by moving the data between processors before the work is started. This should be done in such a way to minimize the amount of communication and the number of messages sent, by using an algorithm such as that described in section 3.3.2.

The UM has successfully utilised all three of these load imbalance minimisation techniques, almost completely removing load imbalance from sections of the code, with little or no additional performance overhead.