V2.

Scalable implementation of the parallel multigrid method on massively parallel computers

K. S. Kang

High Level Support Team (HLST)
Max-Planck-Institut für Plasmaphysik
Boltzmannstraße 2
D-85748 Garching bei München
Germany
kskang@ipp.mpg.de

Abstract

We consider a scalable implementation of multigrid methods for elliptic problems for fusion simulations on current and future high performance computer (HPC) architectures. Multigrid methods are the most efficient available solvers for elliptic problems. But, these methods requires to handle the operations on several coarser levels where the communication costs are higher than the computation costs. We use only one core from a certain coarser level and get performance improvements on a large number of cores. Also, we consider the OpenMP/MPI hybridization implementation which is fitted on multi-core CPU architectures. The hybridization can use a small number of MPI tasks on the same number of cores and achieve performance improvements on a large number of cores. We investigate the scaling properties of the parallel multigrid solver with structured discretization on a regular hexagonal domain on a massively parallel computer (IFERC-CSC).

Keywords: structured grid, parallel multigrid method, OpenMP/MPI hybridization

1. Introduction

Fast elliptic solvers are a key ingredient of massively parallel Particle-in-Cell (PIC) and Vlasov simulation codes for fusion. This applies for both gyrokinetic and fully kinetic models. The currently available most efficient solvers for large elliptic problems are multigrid methods, especially geometric multigrid methods which require detailed information about the geometry for constructing the hierarchy needed.

Multigrid methods are a well-known, fast and efficient algorithms to solve many classes of problems [1, 3, 5, 6, 14]. In general, the ratio of the communication costs to computation costs increases on the coarser level, i.e., the communication costs are high on the coarser levels in comparison to the computation costs. Since, multiplicative multigrid algorithms are applied on each level, the bottleneck of parallel multigrids lies on the coarser levels, including the exact solver at the coarsest level. Additive multigrid methods could combine all the data communication for the different levels in one single step. Unfortunately, the convergence of these versions are not guaranteed if used as solvers, they can be used only for the preconditioner [13] and usually need almost twice as many iterations instead. The multiplicative versions can be used both as solvers and as preconditioners, so we consider the multiplicative versions only.

The feasible coarsest level of operation of parallel multigrid methods depend on the number of cores since there must be at least one degree of freedom (DoF) per core (the coarsest level limitation). Thus, the total number of DoF of the coarsest level problem increases with increasing number of cores. To improve the performance of parallel multigrid methods, we consider reducing the number of executing cores to one (the simplest case) after gathering data from all cores on a certain level (gathering the data) [8]. This algorithm avoids the coarsest level limitation. Numerical experiments on large numbers of cores show a very good performance improvement. However, this implementation may still be further improved, if we manage to reduce the number of MPI tasks to yield better scaling properties.

Modern computer architectures have highly hierarchical system design, i.e., multi-socket multi-core shared-memory computer nodes which are connected via high-speed interconnects. This trend will continue into the foreseeable future, broadening the available range of hardware designs even when looking at high-end systems. Consequently, it seems natural to employ a hybrid programming model which uses OpenMP for parallelization inside the node and MPI for message passing between nodes.

Expected benefits with OpenMP/MPI hybridization are a good usage of shared memory system resources (memory, cache, latency, and bandwidth), and a reduced memory footprint [7]. OpenMP coarsens the granularity at the MPI level (larger message sizes) and allows increased and/or dynamic load balancing. This is preferential for some problems which have naturally two-level parallelism or only use a restricted number of MPI tasks. Consequently, such a programming model can have better scalability than both pure MPI and pure OpenMP. The most important benefit of applying the hybrid OpenMP/MPI programming model to parallel multigrid methods is that it can reduce the number of MPI tasks and thus decrease the communication cost of the coarser level. This simple fact leads to better scalability on the same number of cores.

In this paper, we consider structured triangulations of a hexagonal domain for an elliptic partial differential equation as a test problem [9]. This test problem is a simple problem and well solved by multigrid solvers with the lowest level space which has very small number of DoF (nearly one). The parallel implementations in this paper improve the performances on massively parallel computers, not solvability of the problems, and may apply to more complex problems which are solved by multigrid solvers. To investigate the numerical performance, we use the HELIOS machine in the International Fusion Energy Research Centre (IFERC) at Aomori, Japan. We achieve performance improvements on large number of cores, especially, when the number of the degrees of freedom per core is small.

2. Parallel multigrid method with gathering data

Multigrid methods are well-known, fast and efficient algorithms to solve many classes of problems [2, 3, 6, 14]. The motivation for the multigrid method is the fact that basic iterative methods, such as damped Jacobi and Gauss-Seidel methods, reduce well the high-frequency error but have difficulties to reduce the low-frequency error, which can be well approximated after projection to the coarser level problem. Multigrid methods consist of two main steps, one is the smoothing step with smoothing iterations and the other is the intergrid transfer step with projection and prolongation operators. The former has to be easy to be implemented and be able to reduce effectively the high frequency error.

Multigrid methods are summarized in Algorithm 1 to solve

$$A_h u_h = f_h$$

with stiffness matrix $A_h \in \mathbb{R}^{N\times N}$, vector of unknowns $u_h \in \mathbb{R}^N$, and right hand side (RHS) vector $f_h \in \mathbb{R}^N$ on a discrret grid Ω_h. Two successive grid levels are denoted by Ω_h and Ω_H.

Algorithm 1 Recursive multigrid: $u_h^{(k+1)} = V_h\left(u_h^{(k)}, A_h, f_h, \nu_h^1, \nu_h^2, \mu\right)$

```
1: if coarse level then
2:     solve $A_h u_h = f_h$ by a parallel direct solver or Krylov iteration solver
3: else
4:     $\bar{u}_h^{(k)} = \mathcal{S}^{\nu_h^1}\left(u_h^{(k)}, A_h, f_h\right)$ {presmoothing}
5:     $r_H = Rr_h = R(f_h - A_h\bar{u}_h^{(k)})$ {restrict computed residual}
6:     $e_H^i = e_H^{i-1} + V_H\left(0, A_H, r_H - A_H e_H^{i-1}, \nu_H^1, \nu_H^2, \mu\right)$ for $i = 1, \ldots, \mu$ {recursion}
7:     $\tilde{u}_h^{(k)} = \bar{u}_h^{(k)} + Pe_H^{\mu}$ {prologate error and do coarse grid correction }
8:     $u_h^{(k+1)} = \mathcal{S}^{\nu_h^2}\left(\tilde{u}_h^{(k)}, A_h, f_h\right)$ {postsmoothing}
9: end if
```

In Algorithm 1, we call V-cycle if $\mu = 1$ and W-cycle if $\mu = 2$. Among V-cycle, we call variable V-cycle if the number of smoothing iterations ν_h^1 and ν_h^2 are increased on the coarser level, i.e., $\beta_0^i \nu_h^i \le \nu_H^i \le \beta_1^i \nu_h^i$ with $\beta_0^i \ge 2$. Usually, we set $\nu_h^1 = \nu_h^2$ and use forward and backward Gauss-Seidel iteration alternatively to make symmetric multigrid algorithms [2].

The main issue with the parallelization of multigrid methods is execution time on the coarser level iterations. In general, the ratio of the communication costs to computation costs increases when the grid level is decreased, i.e., the communication costs are high on the coarser levels in comparison to the computation costs. Since, multiplicative multigrid algorithms are applied on each level, the bottleneck of parallel multigrids lies on the coarser levels, including the exact solver at the coarsest level.

Because multigrid methods work on both the coarse and fine grid levels, to get good scaling performance, we might need to avoid operating on the coarser level if possible. Usually, the W-cycle and the variable V-cycle multigrid methods require more work on the coarse level problems [2, 4], so we consider for parallelization only V-cycle multigrid methods.

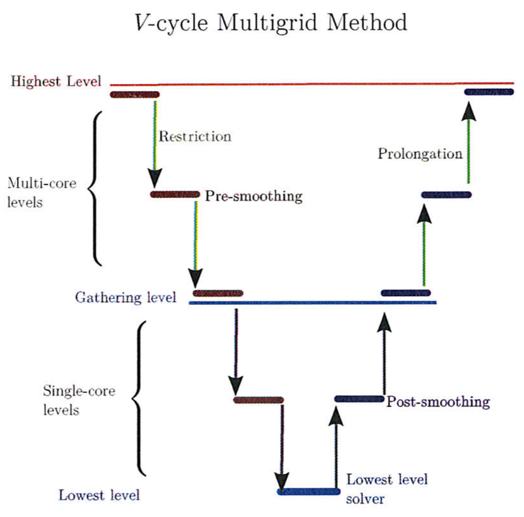

Figure 1: The V-cycle multigrid algorithm including single core levels.

In addition to the execution time on the coarser level, we have to consider the solving time on the coarsest level. As a coarsest level solver, we can use either a Krylov subspace method or a direct method. The solving time of both methods increases significantly with the problem size. So in considering the solution time of the coarsest level, we need to find the optimal coarsening level, as well as the ratio of the communication to computation on each level.

The feasible coarsest level of operation of parallel multigrid methods depends on the number of cores (the coarsest level limitation). The number of degrees of freedom (DoF) of the coarsest level problem will be increased as the number of cores is increased. To improve the performance of parallel multigrid methods, we consider reducing the number of executing cores to one (the simplest case) after gathering data from all cores on a certain level (gathering level) as shown in Fig. 1.

Such a multigrid algorithm variation can solve the coarsest level problem on one core only, independently of the total number of cores. Instead of having only one core solving the coarser level problem and the other cores idling, we choose to replicate the same computation on the coarser levels on each core; then we use these results for computations on the finer level. We use `MPI_Allreduce` which may yield a better performance than using combinations of `MPI_Reduce` and `MPI_Bcast`, depending on the `MPI` implementation on the given machine.

Among the known smoothing operators, the most efficient one is the incomplete LU factorization. But, this smoothing operator is complex to implement and is very hard to parallelize. The damped Jacobi iteration is easy to implement and easy to parallelize, but it is not efficient. The Gauss-Seidel iteration is the most popular smoothing operator because it is simple to implementation and efficient. Although it is apparently sequential in nature, multicoloring approach can be used for parallel implementation. But, this approach needs reordering and more communication steps in one ierataions and can be problematic for unstructured grids [4]. So, we consider the localized Gauss-Seidel iteration which perform the

Gauss-Seidel iteration without data communication during the iteration (known as hybrid Gauss-Seidel in [4]). After one iteration, the data are exchanged via MPI communication. The multigrid operator with the Gauss-Seidel smoother can be symmetric when the forward- and backward-iterations is performed alternatingly. This property allows to be used as a preconditioner of the conjugated gradient method (CGM). This symmetry also holds with the localized Gauss-Seidel iteration because the updated values in one iteration are always updated.

3. OpenMP/MPI hybridization

Modern computer architectures have a highly hierarchical system design, i.e., multi-socket multi-core shared-memory computer nodes which are connected via high-speed interconnects. This trend will continue into the foreseeable future, broadening the available range of hardware designs even when looking at high-end systems.

OpenMP (Open Multi-Processing) is an API that supports multi-platform shared memory multiprocessing programming in C, C++, and FORTRAN, on most processor architectures and operating systems. It uses a portable, scalable model that gives programmers a simple and flexible interface for developing parallel applications for platforms ranging from the standard desktop computer to the supercomputer. It is an implementation of multi-threading, a method of parallelization whereby a master thread (a series of instructions executed consecutively) forks a specified number of slave threads and a task is divided among master and slave threads. This parallelisation is achived by adding OpenMP pragmas with complier option which allows to keep serial mode. The threads then run concurrently, with the run time environment allocating threads to different processors.

The section of code that is meant to run in parallel is marked accordingly, with a preprocessor directive that will cause the threads to form before the section is executed. Each thread has an *id* attached to it which can be obtained using a function call. The thread *id* is an integer, and the master thread has an *id* of 0. After the execution of the parallel code, the threads join back into the master thread, which continues onward to the end of the program. By default, each thread executes the parallel section of code independently. Work-sharing constructs can be used to divide a task among the threads so that each thread executes its allocated part of the code. Both task parallelism and data parallelism can be achieved using OpenMP in this way.

In the OpenMP programming model, multiple threads share the computations which can be parallelized. So, we use the multi-threading on the computation intensive parts of the program. The computation intensive routines in the iterative solver, such as the conjugated gradient method (CGM), GMRES, and multigrid methods, are the matrix-vector multiplication and computation of the residual norm. Also, the smoothing iterations and the intergrid transfer operators including restrictions and prolongations can be considered as a matrix-vector multiplication. We modify these routines by adding OpenMP pragmas, i.e., share the work by master and slave threads.

Remark 3.1. *The property of the localised Gauss-Seidel iteration that updated values in one iteration are always updated does not hold in OpenMP parallelization due to race condition,*

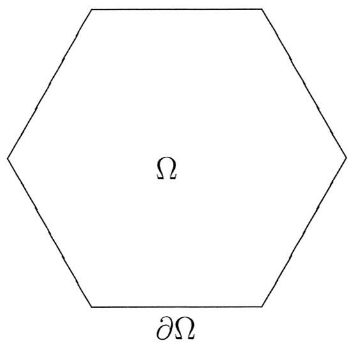

Figure 2: Domains of the model problem.

so the parallel multigrid operator with the localized Gauss-Seidel iteration with OpenMP might not be symmetric. However, this phenomenon does not happen in multi-coloring Gauss-Seidel itereation.

Expected benefits with OpenMP/MPI hybridization are to have a good usage of shared memory system resources (memory, latency, and bandwidth). It avoids the extra communication overhead with MPI within each node (reduce memory footprint). OpenMP adds coarse granularity (larger message sizes) and allows increased and/or dynamic load balancing. It is good for some problems which have naturally two-level parallelism or only use a restricted number of MPI tasks. So this programming model might have better scalability than both pure MPI and pure OpenMP. The most important expected benefit of the hybridization programming for multigrid methods is that it can reduce the number of MPI tasks, i.e., better scalability for the same number of cores can be expected.

4. Model problem

We consider the Poisson type second order elliptic partial differential equations on domain. We consider a regular hexagonal domain with Dirichlet boundary condition on the outer boundary in Fig. 2. This model problem is considered in the underlying physics code project GEMT by B. Scott [10]. It is intended to generate a new code by combining two existing codes: GEMZ and GKMHD. GEMZ, is a simulation code for turbulent tokamak dynamics using gyrofluid equations, based on a conformal, logically Cartesian grid [11, 12]. GKMHD is an MHD equilibrium solver which is intended to evolve the Grad-Shafranov MHD equilibrium with a reduced MHD description which is derived self consistently from the gyrokinetic theory. GKMHD already uses a triangular grid, which has a logically spiral form with the first point on the magnetic axis and the last point on the X-point of the flux surface geometry. The solving method is to relax this coordinate grid onto the actual contours describing the flux surface of the equilibrium solution. Such a structure is logically the same as in a generic tokamak turbulence code.

For the regular hexagonal domain, we consider the following partial differential equation

$$\begin{aligned} -\nabla \cdot a(x,y)\nabla u &= f, \quad \text{in } \Omega, \\ u &= 0, \quad \text{on } \partial\Omega\ , \end{aligned} \tag{1}$$

where $f \in L^2(\Omega)$ $a(x,y)$ is a uniformly positive and bounded function. It is well known that the Eq. (1) has a unique solution.

The second-order elliptic problem (1) is equivalent to finding $u \in H_0^1(\Omega)$ such that

$$a_E(u,v) = \int_\Omega a(x,y)\nabla u \cdot \nabla v \,\mathrm{dx} = \int_\Omega fv \,\mathrm{dx} \tag{2}$$

for any test function $v \in H_0^1(\Omega)$ where $H_0^1(\Omega)$ is the space of functions which have weak first differentives that have a finite L^2-norm in Ω with zero values on the boundary $\partial\Omega$.

We consider a piecewise linear finite element space defined on a triangulation with regular triangles for the hexagonal domain. This triangulation generates a structured grid and can be applied to a D-shape tokamak interior region with a conformal mapping. Let h_1 and $\mathcal{T}_{h_1} \equiv \mathcal{T}_1$ be given, where $\mathcal{T}_1$ is a partition of Ω into triangles and h_1 is the maximum diameter of the elements of $\mathcal{T}_1$. For each integer $1 < k \leq J$, let $h_k = 2^{-(k-1)}h_1$ and the sequence of triangulations $\mathcal{T}_{h_k} \equiv \mathcal{T}_k$ be constructed by the nested-mesh subdivision method, i.e., let $\mathcal{T}_k$ be constructed by connecting the midpoints of the edges of the triangles in $\mathcal{T}_{k-1}$, and let $\mathcal{T}_{h_J} \equiv \mathcal{T}_J$ be the finest grid.

Let us define the piecewise linear finite element spaces

$$V_k = \{v \in C^0(\Omega) : v|_K \text{ is linear for all } K \in \mathcal{T}_k\}.$$

Then, the finite element discretization problem can be written as follows: find $u_J \in V_J$ such that

$$a_E(u_J, v) = \int_\Omega fv \,\mathrm{dx} \tag{3}$$

for any test function $v \in V_J$, i.e., solve the linear system

$$A_J u_J = f_J. \tag{4}$$

Let us now consider the parallelization of the above problem. We consider the way to divide the hexagonal domain into sub-domains with the same number of cores. Except for the single core case, we divide the hexagonal domain in regular triangular sub-domains and each core handles one sub-domain. Hence, feasible numbers of cores are limited to the numbers 6×4^n for $n = 0, 1, 2, \ldots$ We use real and ghost nodes on each core. The values on the real nodes are handled and updated locally. The ghost nodes are the part of the distributed sub-domains located on other cores whose values are needed for the local calculations. Hence, the values of the ghost nodes are first updated by the cores to which they belong to as real nodes and then transferred to the cores that need them. To reduce data communication during matrix element computation, the computation of matrix elements on some cells can be executed on several cores which have a node of the cell as a real node.

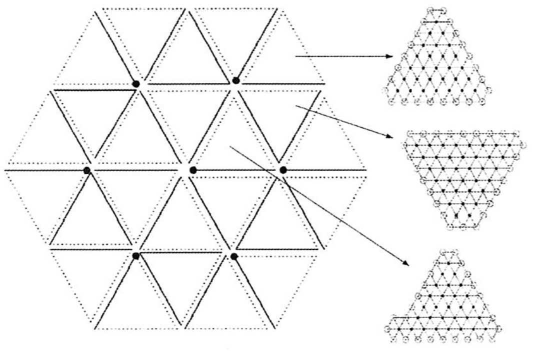

Figure 3: The sub-domains on 24 cores and real (–,•) and ghost (⋯, ∘) nodes on sub-domains according to the three different domain types

For each core we have to define what are real and ghost nodes on the common boundary regions of the sub-domains. We determine the nodes on the common boundary of the sub-domains as the real nodes of the sub-domain which are located in the counterclockwise direction or in the outer direction from the center of the domain as shown in Fig. 3. For our problem with a Dirichlet boundary condition on the outer boundary, we can handle the boundary nodes as ghost ones. The values of these boundary nodes are determined by the boundary condition and thus do not have to be transferred between cores.

We number the sub-domains beginning at the center and going outwards following the counterclockwise direction. Each sub-domain can be further divided into triangles; this process is called triangulation. In this process each line segment of the sub-domain is divided into 2^n parts. It can be shown that, independently of the total number of sub-domains and triangulation chosen, there are just three domain types (see Fig. 3). These give detailed information on the real and ghost nodes being connected to other sub-domains and cells which are needed to compute the matrix elements for the real nodes. To see how good the load balancing is, we measure the ratio of the largest number of real nodes to the smallest number of real nodes which is $\{2^n(2^n+3)\}/\{2^n(2^n+1)\}$ which tends to '1' as n is increased.

To get the values on the ghost nodes from the other cores for all sub-domains, we implement certain communication steps. The communication steps are the dominating part of the parallelization process and thus a key issue for the performance of the parallel code. The easiest way to implement the data communication would be that every ghost node value is received from the core which handles it as a real node value. However, such implementation would need several steps and the required number would then vary among the different cores. So this approach could be used for unstructured grids, but it would be too slow in our case. However, we solved the problem by using a sophisticated data communication routine which needs a fixed number of steps for each core (that is, five).

Our dedicated data communication steps are as follows :

S1: Radial direction (Fig. 4 a.)

S2: Counterclockwise rotational direction (Fig. 4 b.)

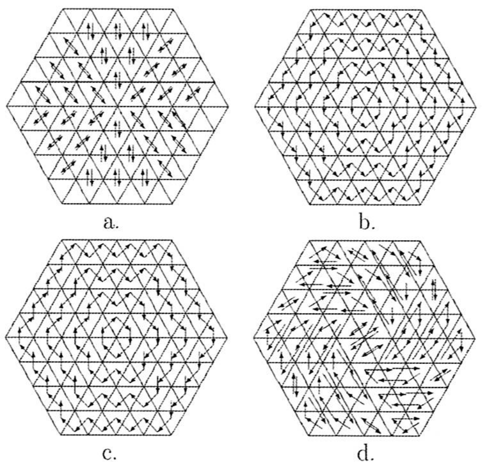
Figure 4: The data communication steps. The arrows represent data movements one sub-domain to other sub-domain in corresponding cores.

S3: Clockwise rotational direction (Fig. 4 c.)
S4: Radial direction (same as in S1) (Fig. 4 a.)
S5: Mixed communications (Fig. 4 d.)

5. Numerical Experiments

To get performance results we run the program on the HELIOS machine which was built in the framework for the EU(F4E)-Japan broader approach collaboration. The machine is made by 4410 Bullx B510 Blades nodes of two 8-core Intel Sandy-Bridge EP 2.7 GHz processors with 64 GB memory and connected by Infiniband QDR. So it has a total of 70 560 cores total and 1.23 Peta-flops Linpack performance.

First, we consider the matrix-vector multiplication which is the key component of iterative methods such as CGM, GMRES, and multigrid methods. We consider the execution time of the one matrix-vector multiplication (denoted by 'execution time') and the solution time of the preconditioned CGM with the parallel multigrid preconditioner with Gauss-Seidel smoothing (without any modification) (denoted by 'solution time'). In the following figures, we use log scales in time and the number of cores because this scaling is relevant and easy to understand the numerical results. We depicted execution time and solution timeas a function of the number of cores with the same number of DoF per core on the finest level (semi-weak scaling) in Fig. 5. We use two pre- and post-smoothings in the parallel multigrid method as a solver and as a preconditioner and the relative residual error in L^2-norm is lees than 10^{-10} as a stop criterion of the iterative solvers. For the multigrid method, a pure weak scaling seems to be hard to achieve. The number of operations per core has to be fixed. However, increasing the problem size according to the number of cores automatically leads to introducing additional multigrid levels to keep the size of the coarsest level problem constant. Therefore, the number of operations per core slightly increases in our semi-weak scaling due to additional multigrid levels. In Fig. 5, we can see that the matrix-vector multiplication has very good weak scaling properties, whereas the solution time is degraded as the number of cores is increased, especially when the number of the DoF per core is small.

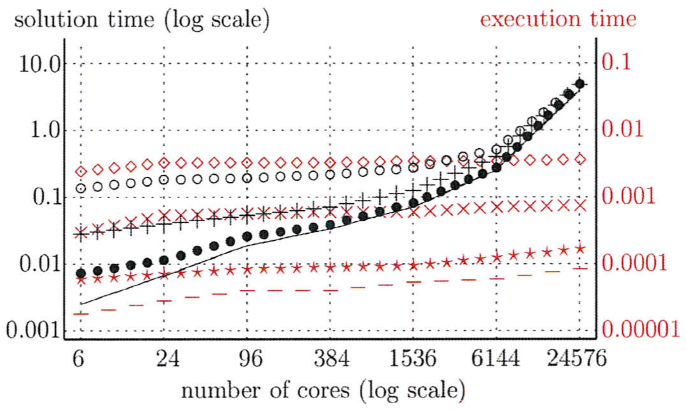

Figure 5: The solution time in seconds of the multigrid method with a Gauss-Seidel smoother as a preconditioner for the PCGM (in black) and the execution time of the matrix-vector multiplication (in red) as a function of the number of cores for domains with 2K DoF (solid line, $-$), 8K DoF ($\bullet$, $\star$), 32K DoF($+$, $\times$), and 132K DoF($\circ$, $\diamond$) per core.

Next, we consider the parallel multigrid method with the gathering level that avoids the coarsest level limitation. As a gathering level, we chose the lowest gathering level on which each MPI task has at least one DoF [9]. So the number of total DoF of the gathering level is increased as the number of MPI tasks is increased. The numerical experimental results on large numbers of cores show a very good performance improvement as can be seen in Fig. 6. They show also that this implementation still needs improvements for large number of MPI tasks and small numbers of DoF per core.

To look at the effects of the hybridization, we compare the execution time of the matrix-vector multiplication on a fixed number of MPI tasks with the same number of threads on each MPI tasks for a certain problem size. We tested varying numbers of threads per MPI task, i.e., 2, 4, 8, and 16 threads per MPI task. The total number of threads times the number of MPI tasks are matched with the number of cores being allocated for the job. We depicted the execution time of the matrix-vector multiplication as a function of the number of threads on the two different MPI tasks, on 96 MPI tasks in Fig. 7 (a) and on 384 MPI tasks in Fig. 7 (b). As a comparison, we added the ideal cases with dotted line.

The results in Fig. 7 clearly show that the OpenMP parallelization has a good strong scaling property up to eight threads. This is the number of cores per socket on a HELIOS node, which comprises by two sockets. The scaling is more efficient the larger the size of the problem is. However, for more than eight treads, the performance seems to degrade which is probably due to the architecture of the node. Due to cache memory effects, we can see super-linear speedup with eight threads for the problem size of 12.5M DoF on 96 MPI tasks

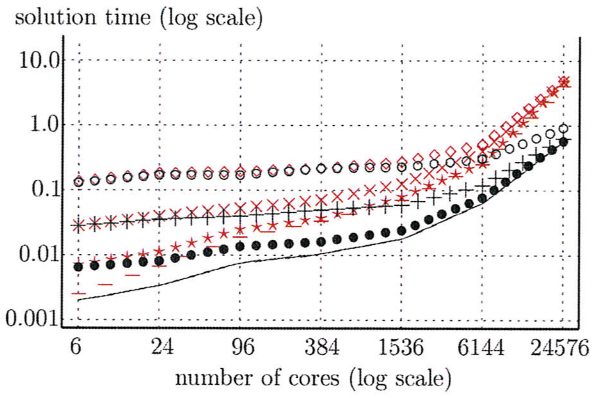

Figure 6: Semi-weak scaling. The solution time in seconds of the multigrid method with a Gauss-Seidel smoother as a preconditioner for the PCGM with (in black) and without (in red) gathering the data as a function of the number of cores for domains with 2K DoF (solid line,−), 8K DoF (•, ⋆), 32K DoF(+, ×), and 132K DoF(∘, ⋄) per core.

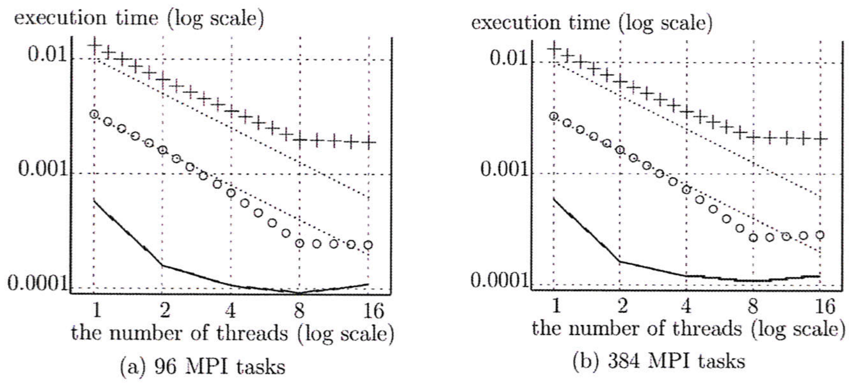

Figure 7: Hybridization. The execution time in seconds of the matrix-vector multiplication as a function of the number of threads for domain with 3.1M DoF (solid line), 12.5M DoF (∘), and 50M DoF(+) on 96 MPI tasks in (a) and with 12.5M DoF (solid line), 50M DoF (∘), and 200M DoF(+) on 384 MPI tasks in (b). The dotted lines are the ideal cases.

and of 50M DoF on 384 MPI tasks.

Finally, we consider the OpenMP/MPI hybridization cases. As we discussed in Remark 3.1, the multigrid preconditioner with the Gauss-Seidel smoother cannot be used as a preconditioner for PCGM. So, we show the solution times of the multigrid method with the Gauss-Seidel smoother as a solver with hybridization and pure MPI in Fig. 8. We select five different cases: 2k, 8k, 32k, 130k, and 500k DoF per core. There we compare the pure MPI case occupying all 16 cores on each node (in red) with the best performing hybrid cases of 1, 4, or 16 OpenMP threads per MPI task, i.e., 16, 4, or 1 MPI task per node (in black). We did not consider the hybrid cases of 2 and 8 OpenMP threads in weak scaling because the total number of DoF in one higher level is four times of them. For the pure MPI case the performance significantly degrades when the number of cores becomes large for cases with a small number of DoF per core. This situation improves with the hybridization, i.e., problems with small number of DoF can be solved with a larger number of threads on a larger number of cores. The threads work on the shared memory of a node which makes this method more efficient by avoiding part of the intra-node communication of the pure MPI method.

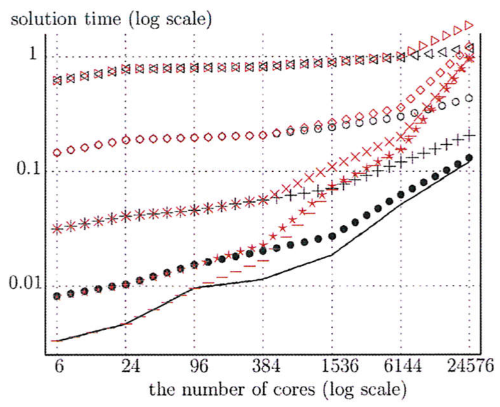

Figure 8: Semi-weak scaling OpenMP/MPI. The solution time in seconds of the multigrid method with a Gauss-Seidel smoother as a solver as a function of the number of cores for a fixed number of DoF per core. (The best results of the hybrid model in black, pure MPI in red. 2K (solid line, –), 8K (•,⋆), 32K (+, ×), 130K (○, ◇) and 500K (◁, ▷) per core).

6. Conclusions

We considered the gathering of the data on each core for a certain coarser level and OpenMP/MPI hybridization to get better performance of the parallel multigrid method. We assessed the performance improvement of these implementations on a structured triangular grid on a regular hexagonal domain. The improvement was especially significant on large number of cores with relatively small number of DoF per core cases.

Acknowledgments

This work was carried out using the HELIOS supercomputer system at the Computational Simulation Centre of the International Fusion Energy Research Centre (IFERC-CSC), Aomori, Japan, under the Broader Approach collaboration between Euratom and Japan, implemented by Fusion for Energy and JAEA. This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement number 633053. The views and opinions expressed herein do not necessarily reflect those of the European Commission. I would like to thank R. Hatzky, B. Scott, and other HLST team members for helpful discussions.

References

[1] S. F. Ashby and R. D. Falgout, *A parallel multigrid preconditioned conjugated gradient algorithm for groundwater flow simulations*, Nuclear Science and Engineering, **124**(1), 1996, pp. 145–159.

[2] J. Bramble, Multigrid Methods, Pitman, London, 1993.

[3] A. Brandt, *Multigrid techniques with applications to fluid dynamics: 1984 guide*, in VKI Lecture Series, Mar. 1984, 176 pp.

[4] W. Chow, R. D. Falgout, J. J. Hu, R. S. Tuminaro, and U. M. Yang, *A Survey of Parallelization Techniques for Multigrid Solvers*, Parallel Processing For Scientific Computing, by Heroux, Raghavan, and Simon, editors, SAIM, Series on Software, Environments, and Tools, SIAM Publications, Philadelphia, PA, 2005.

[5] B. Gmeiner, H. Köstler, M. Stürmwer, and U. Rüde, *Parallel multigrid on hierarchical hybrid grids: a performance study on current high performance computing clusters*, Concurrency and Computation: Practice and Experience, **26**, 2014, pp. 217–240.

[6] W. Hackbusch, Multigrid Methods and Applications, Springer-Verlag, Berlin, Germany, 1985.

[7] G. Hager, G. Jost, and R. Rabenseifner, Communication Characteristics and Hybrid MPI/OpenMP Parallel, Programming on Clusters of Multi-core SMP Nodes *Cray User Group 2009 Proceedings.*

[8] K. S. Kang, Parallelization of the Multigrid Method on High Performance Computers, IPP-Report 5/123, 2010.

[9] K. S. Kang, *A Parallel Multigrid Solver on a Structured Triangulation of a Hexagonal Domain*, 21st International Conference on Domain Decomposition Methods, INRIA Rennes-Bretange Atlantique, Campus de Beaulieu, 35042 Rennes Cedex, France, June 25–19, 2012, pp. 789–797.

[10] E. Poli, A. Bottino, W. A. Hornsby, A. G. Peeters, T. Ribeiro, B. D. Scott, and M. Siccino, *Gyrokinetic and gyrofluid investigation of magnetic islands in tokamaks*, Plasma Physics and Controlled Fusion, **52** (2010), Art. No. 124021.

[11] T. Ribeiro and B. Scott, *Conformal Tokamak Geometry for Turbulence Computation*, IEEE Trans. Plasma. Sci. 38 (2010) 2159.

[12] B. D. Scott, *Free Energy Conservation in Local Gyrofluid Models*, Physics of Plasmas, **12** (2005), Art. No. 102307.

[13] U. Trottenberg, C. W. Oosterlee, and A. Schüller, Multigrid, Academic Press, 2001.

[14] P. Wesseling, An Introduction to Multigrid Methods, Wiley, Chichester, 2003.