Comparison of Different Methods for Performing Parallel I/O

Matthieu Haefele, HLST

7^{th} June 2010

1 Introduction

Transistor numbers within a single chip have increased and still increase according to Moore's law since the seventies, so it has doubled every 18 months since that time. The dataset size generated by computer simulations which use such computational power has increased accordingly. The capacity of storage systems has followed the same kind of growth but not their bandwidth even if large improvements have been made. So, the cost for exporting large datasets on storage system is high as well as their post processing. When the price for such export is paid, then the interactive visualization and interpretation of such large dataset becomes an issue. As a consequence, scientists perform that kind of exports nowadays as seldom as possible or never in some cases, exporting only already post-processed quantities. During the last decade, the computational power available within a single high performance computer has increased thanks to an increased number of computational cores. So computations as well as data computed are distributed among these cores making the data export even more difficult. The trend for an increasing number of cores for the next generation of computers is still going on with an exponential growth, raising accordingly the I/O problem complexity.

However, efforts have been made in that field to design software that improve I/O scaling. On the one hand, parallel file systems give to the user a unified view of several I/O devices. So the user reads and writes files as if a single I/O device is connected to the machine but the parallel file system stripes the files among the different real I/O devices. As a consequence, the user sees the aggregated bandwidth of the different I/O devices instead of seeing the bandwidth of a single one which partially solves the large sized dataset issue. On the other hand, the MPI-2 [1] standard contains functionalities to export a single logical file from data distributed among the different MPI tasks of an MPI application. This MPI-IO [2, 3] set of functions gives an answer to the export of highly distributed data in current and future MPI applications.

The purpose of this report is to evaluate the different possible solutions to export a large AND distributed dataset. Indeed, the behavior of a parallel application in interaction with a parallel file system is difficult to foresee. It is even more difficult as several methods exist to solve the same problem. Contrarily to [4] whose purpose is to benchmark storage systems and file systems, our focus is really to evaluate what a user can expect of the different methods in a production environment. So all the measurements in this study have been performed within standard queues on heavily loaded machines on which the storage system is shared. Section 2 describes the test case used for our evaluation and the different I/O methods evaluated. Section 3 shows the performance of the different methods and gives some interpretations. Finally, conclusions and future work are given in section 4.

2 Different methods for parallel I/O

2.1 Test case

The test case consists of the export of a simple 2D structured array distributed in a block-block manner among the MPI tasks as depicted on figure 1. The array always contains S entities in both direction. However, the number of cores affected in each dimension (p_x, p_y) can be different, leading to different sizes in each dimension of the array local to each MPI task. The total number of used cores is $P = p_x p_y$. The array ordering is shown with a Fortran style major column ordering.

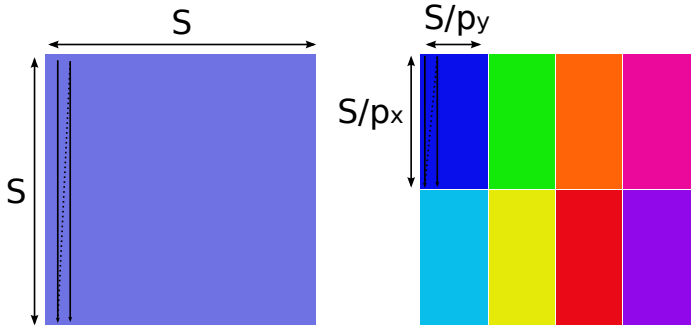
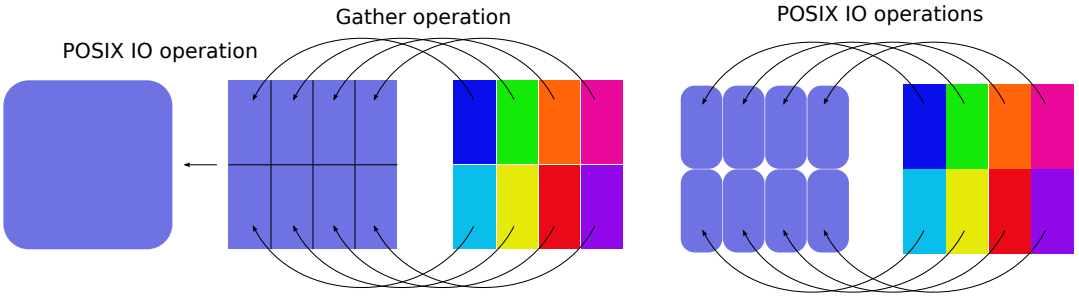


Figure 1: Data distribution among MPI Tasks with $p_x = 2$ and $p_y = 4$.

2.2 POSIX file

Portable Operating System Interface (POSIX) is the name of a family of related standards specified by the IEEE to define the application programming interface (API), along with shell and utilities interfaces for software compatible with variants of the Unix operating system, although the standard can apply to any operating system. To provide an application access to a file for reading or writing is part of this standard. In short words, this is the way I/O is performed on UNIX like system outside the scope of high performance computing.

Files written through POSIX interface represent the simplest mechanism to export data on disk. Considering a parallel application, two strategies can be used to perform this export.



(a) MPI_gather + single POSIX file

(b) Multiple POSIX files

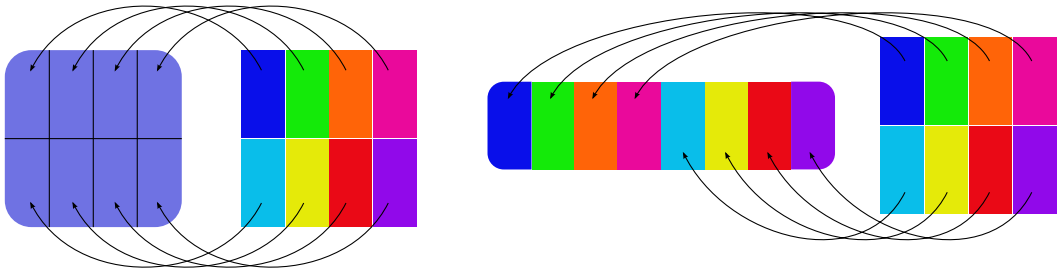
Figure 2: POSIX files

MPI_gather + single POSIX file method consists in gathering all the data within a single MPI task and then writing it in a file on disk through a POSIX call. The main

implementation difficulty resides in the MPI gather call. The file write is obvious. But this method is restricted by the memory limitation of the receiving MPI task. Indeed, as the whole dataset is gathered on one particular MPI task, this task must be able to allocate enough memory to store the entire dataset. It is typically bounded by the memory of a single node which can be a limiting factor for large datasets. I/O performance is only determined by the performance of the underlying storage system. Figure 2(a) presents a scheme of this method.

Multiple POSIX files method consists in exporting the data as it is distributed. Each MPI task writes its own local data in a separate file. The implementation is even easier than the previous one as only the file write has to be implemented. On the other hand, the data on disk is not organized like previously as a single logical file. Instead, the dataset is spread among several files with data organized as it was distributed within the parallel application. The number of files as well as the data organization changes as the number of MPI task changes. This represents a drawback for post processing tools development and limits the flexibility for restart files. I/O performance is also mainly determined by the performance of the underlying storage system. However, when the number of MPI processes becomes large, the file system has to handle a large number of files which can become a bottleneck that reduces the overall performance. Figure 2(b) presents a scheme of this method. We have tested two different implementations. The first one writes all the files in the same directory (multi_file inf), the second one gathers N files in separate directories (multi_file N). It turned out, that writing a large number of files in the same directory can cause some parallel file system meta-data server to struggle, in particular IBM GPFS.

2.3 MPI-IO file



(a) Single contiguous array

(b) Multiple chunks

Figure 3: MPI/IO files

The purpose of MPI/IO is to provide a high performance, portable, parallel I/O interface for high performance, portable, parallel MPI programs. MPI/IO, which was contributed to the MPI-2 standard, builds on MPI derived data types and collective communications and so, has very similar resulting semantics. A good introduction can be found in [5]. MPI/IO files are shared, i.e., multiple processes running on multiple cores can operate on a single MPI/IO file all at the same time. So, a single file is created on disk holding the data as if it would have been written through a POSIX call from a single process holding the whole dataset. From a functional point of view, this method is quite close to the MPI_gather + POSIX file solution. The difference is, that in this case, the gather is performed by the MPI implementation within MPI buffers and not by the user. On the other hand, performance strongly depends on the MPI implementation performance in

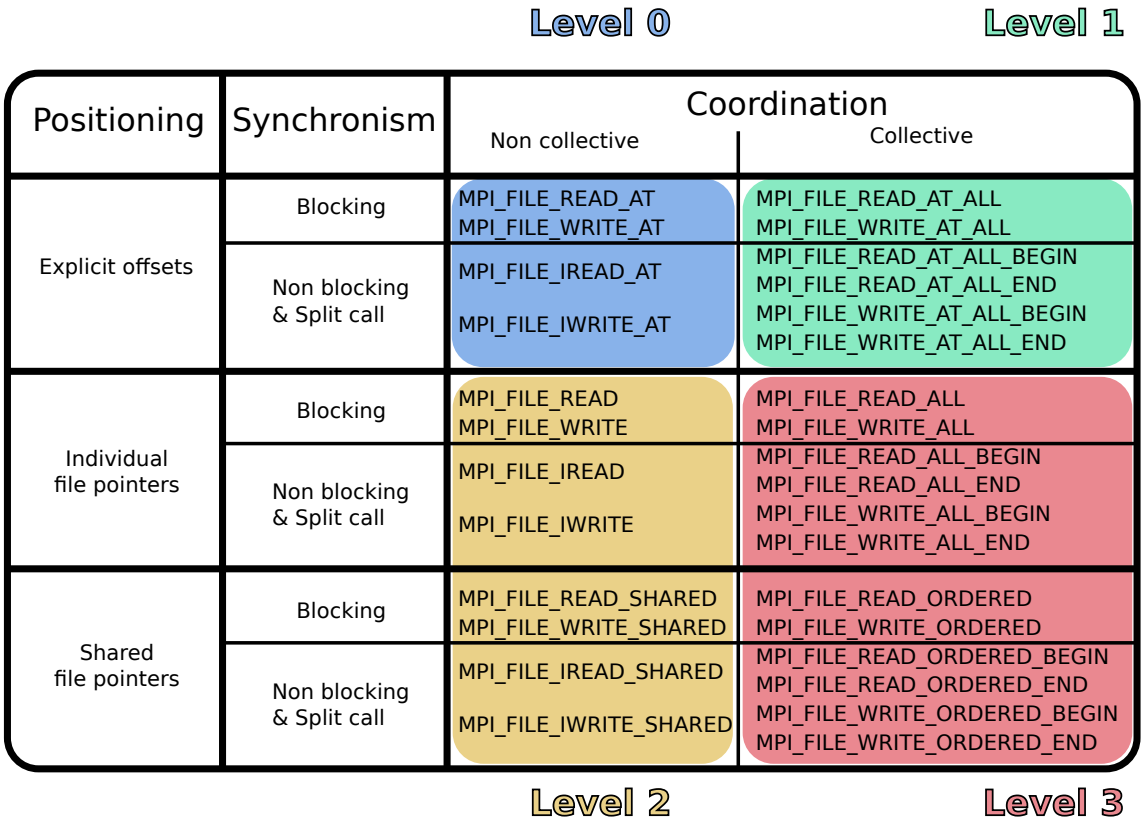


Table 1: MPI/IO Application Programing Interface

particular the way MPI/IO interacts with parallel file system. Two implementations have been performed. Figure 3(a) presents *mpi_file* implementation where the distributed array is gathered in the file to build a single contiguous array. Whereas Figure 3(b) presents the *mpi_file_chunk* implementation where the distributed array is written in chunks within the file so each MPI process writes its array portion contiguously in the file. Better performance are expected from the chunked version but it is more difficult to handle the post-processing of such files.

Table 1 shows all the data access functions provided by MPI/IO. They can be gathered in three different categories according to their positioning method. The first group uses explicit offsets positioning. It enables an MPI process to access contiguous data at a position in the file specified as an offset expressed in bytes. The second group uses individual file pointers. In that case, each MPI process that opens a file owns its own file pointer. Thanks to MPI data types, it is possible to describe how each process accesses the file. This access pattern is called "view" within the MPI terminology. This functionality is quite powerful as it is possible to define non contiguous patterns. So several discontinuous chunks of data from the same file can be accessed within a single access call. The last group uses a single shared file pointer. It enables the same view mechanism as with individual file pointers. The only difference is that this view is shared among all the MPI processes instead of being local to a single one.

For each group described above, the access functions can be classified according to three criteria: access type, coordination and synchronism.

- **Access type** is obvious, a method is either a *read* method or a *write*.

- **Coordination** is either *collective* or *non-collective*. A collective call involves all the MPI processes within a MPI communicator, whereas a non-collective call involves a single process. In general, it is always better to use a collective call than a non-collective one when it is possible. Indeed, more information is given to the MPI implementation which enables potential optimization that would not be possible otherwise. This is even more crucial for file access. A non-collective call is equivalent to a concurrent file access performed by every MPI processes. MPI manages these concurrent accesses at the expense of performance.

- **Synchronism** is either *blocking* or *non-blocking*. A blocking function will return only when the data access is finished, whereas a non-blocking one returns before the end of the data access. Typically, non-blocking calls allow to overlap data access with computations. Collective-non-blocking functions are always a pair of *begin* and *end* functions between which computations can be performed. Non-collective non-blocking functions do not have an explicit *end* function, the MPI_WAIT function is used instead.

From the performance point of view, according to [6], these functions can be classified in four different levels illustrated in figure 4. We have decided to evaluate the four levels of MPI/IO performance for our simple problem. We have used the blocking calls in order to be able to measure the I/O time elapsed within the application and individual file pointers to define the different views.

- **Level 0: Explicit offsets - non-collective calls** It means that each MPI process writes concurrently a single contiguous chunk of data within one call (mpi_file_0).

- **Level 1: Explicit offsets - collective calls** It means that every MPI process writes concurrently a single contiguous chunk of data within one call. Data coming from different processes must not overlap (mpi_file_1).

- **Level 2: File pointer - non-collective calls** It means that each MPI process has a view of the file enabling possible discontinuous accesses within a single call, but accesses are performed concurrently (mpi_file_2).

- **Level 3: File pointer - collective calls** It means that every MPI process has a view of the file enabling possible discontinuous accesses within a single call and accesses are managed collectively by the MPI implementation (mpi_file_3).

2.4 HDF5 file

HDF5 [7] is a data model, library, and file format for storing and managing data. It supports a vast variety of data types, and is designed for flexible and efficient I/O and for large and complex data. HDF5 is portable and is extensible, allowing applications to evolve in their use of HDF5. A parallel implementation [8] exists and has been built on top of MPI/IO. So from a performance point of view, we can expect similar performance as MPI/IO. We have implemented two different tests. The first one, called *phdf5_file* being represented in Figure 5(a), is the HDF5 version of the *mpi_file* implementation. The second one, called *phdf5_file_chunk* being represented in Figure 5(b) is the HDF5 version of the *mpi_file_chunk* implementation. As with MPI, increased performance can be expected from the chunked version because one process writes its entire local data contiguously into a single HDF5 chunk. But contrary to MPI, the post-processing in this case is not changed because chunks are handled by the HDF5 library and not directly by the user.

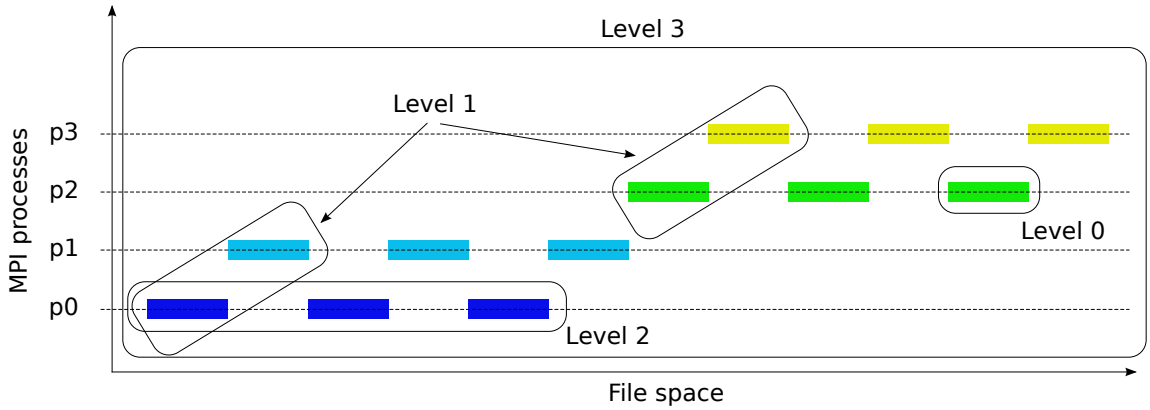
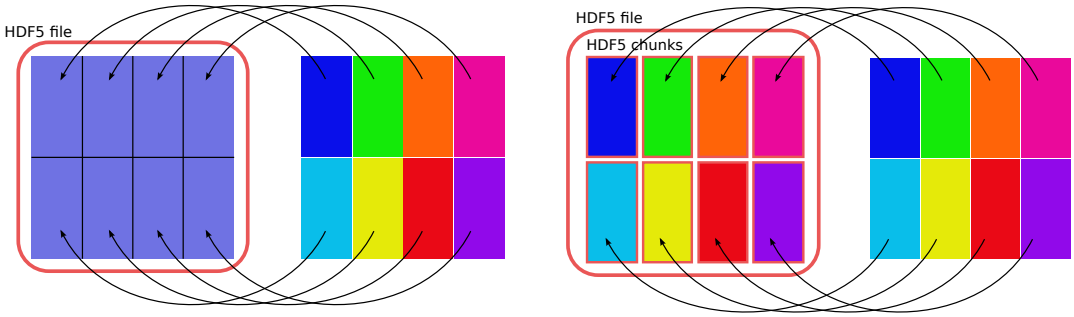


Figure 4: MPI/IO performance levels



(a) Single contiguous array

(b) Multiple chunks

Figure 5: HDF5 files

3 Performance

We have decided to evaluate the POSIX approach, the four levels of MPI/IO performance and the Parallel HDF5 library by using our simple problem. Figures 6(a), 6(b), 7(a) and 7(b) show the achieved bandwidth of the different methods as a function of the number of cores used in different scaling configuration and on two different machines. Figures 6(a) and 6(b) show respectively the result for a strong and a weak scaling on the VIP machine at RZG, an IBM machine with 6560 POWER6 cores connected to a GPFS file system [9] (/ptmp used for these benchmarks). Figures 7(a) and 7(b) show respectively the result for a strong and a weak scaling on the HPC For Fusion (HPC-FF at JSC), a Bull machine with 8640 INTEL Xeon Nehalem-EP cores connected to a Lustre file system [10] (/lustre/jwork1 used for these benchmarks with stripe count=-1). The name of the methods are suffixed with the level of the MPI function used for performing the I/O. For HDF5, only the collective/non-collective coordinations is available as an option and we assume HDF5 uses MPI view to perform the I/O. The strong scaling study consists of exporting a 8 GB dataset distributed among an increasing number of cores. The weak scaling study consists of exporting 4 MB per MPI task, leading to a total file size increasing with the number of cores used, to an 8GB file when 2048 cores are used.

In the ideal case, *i.e.* if the overall performance is only driven by the hardware performance, thus neglecting every communication and software layer overhead, we should observe a single horizontal line showing the bandwidth of the storage system. At a first glance, we can notice the very large performance difference between the different methods

despite they achieved the same functionality. Indeed, this difference is between two and three orders of magnitude. Additionally, the measure's noise is obvious. These performance benchmarks have been performed on the two machines as they were in production. As the storage system is shared among the whole machine, the available bandwidth is reduced if a measure is done while another application is using intensively the storage system, thus reducing the achieved bandwidth. It would have been possible to run these benchmarks several times in order to average the results and to reduce the noise. As our purpose is not to benchmark the storage system but rather to determine best programming practices for I/O, we will see that the current results are precise enough to make some principle conclusions.

In every figures, we can cluster the methods in three different groups. Efficient implementations:

- MPI gather + single POSIX file
- Both Multiple POSIX files
- MPI/IO and Parallel HDF5 level 3

Moderate implementations:

- MPI/IO and Parallel HDF5 level 2

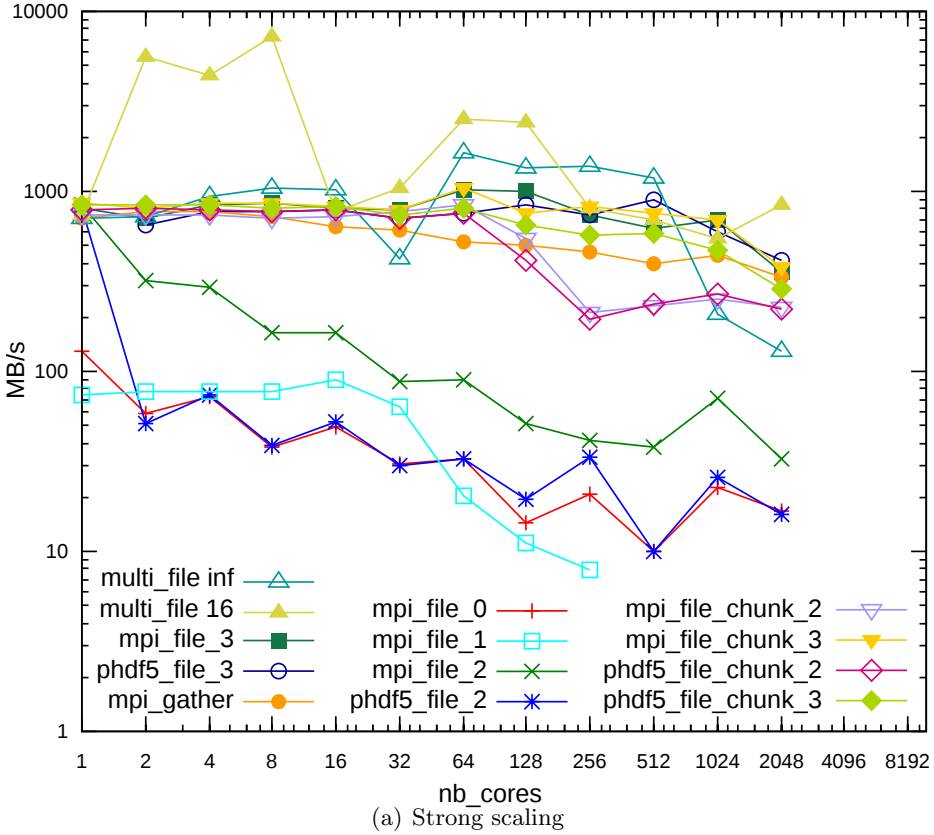
And non efficient implementations:

- MPI/IO level 0 and 1

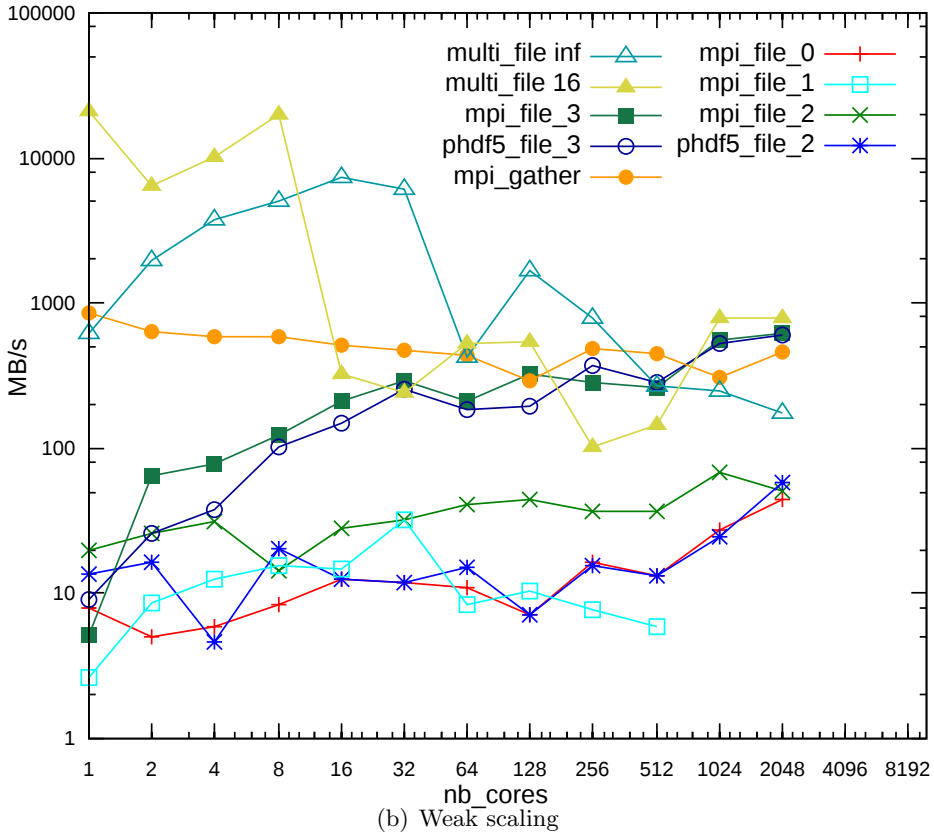
Conclusion 1: Parallel I/O without the view mechanism coordination (explicit offsets) should not be used when good performance is needed. With the view mechanism and with non collective coordination, performance strongly depends on the type of treated problem, so they should not be used without knowing exactly what is going on. Indeed, if every process accesses a different part of the files, good performance can be achieved, even better than in collective coordination in some cases. But that is not the case for our problem, so performance drops.

Conclusion 2: For small file size (typically less than 100MB on nowadays systems) the POSIX interface should be preferred rather than the MPI/IO one. Among the efficient implementations, still several factors separate them. The weak scaling experiments show that the MPI/IO solutions are far behind the POSIX files for small number of cores, *i.e.* small file size. Indeed, POSIX files seems to take advantage of the file system's cache memory, whereas MPI/IO machinery introduces a large overhead compared to the file size. This overhead is less and less visible as the file size increases, leading to comparable POSIX and MPI/IO performance.

Conclusion 3: The usage of HDF5 without chunking in a context of high performance I/O makes sense. Parallel HDF5 performance follow MPI/IO one. As parallel HDF5 is built on top of MPI/IO, it is not surprising, but it is already good to see that this additional software layer does not degrade performance. It might introduce a little overhead, but the high observed noise makes it difficult to quantify.

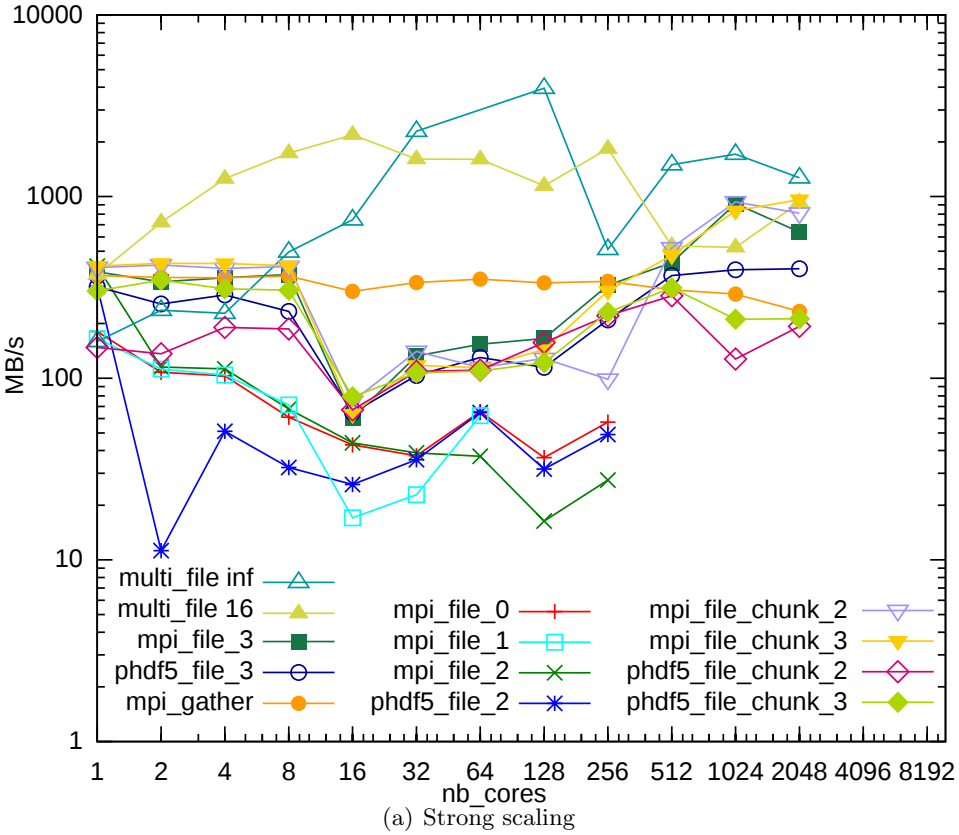


(a) Strong scaling

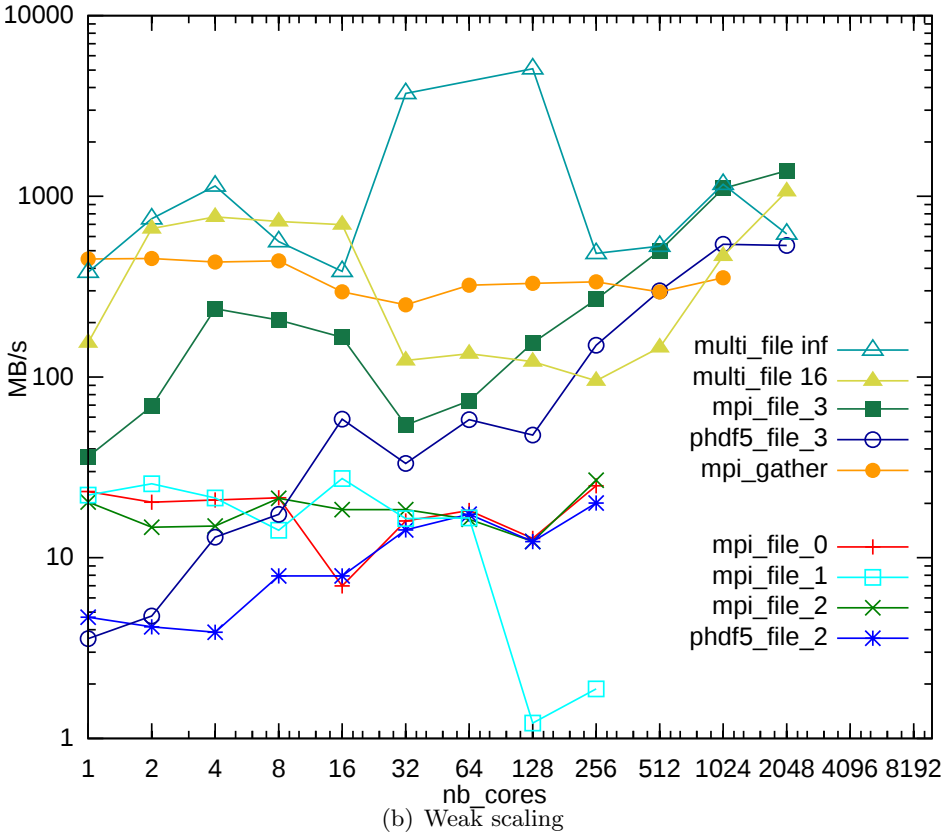


(b) Weak scaling

Figure 6: Performance on RZG VIP machine



(a) Strong scaling



(b) Weak scaling

Figure 7: Performance on JSC HPC-FF machine

Conclusion 4: Additional implementation work to export chunked dataset is not worth. The MPI/IO performance with chunks is similar to the one without chunks. With HDF5, it is even worse: performance with chunks is eight times slower than the one without chunks for large numbers of cores. This overhead is not understood yet. Additional information can be given to HDF5 in order to reduce it but this study has been left as future work. However, it is still interesting to note that the non-collective coordination gives almost the same result as the collective one in the case of chunked dataset on the HPC-FF machine. Indeed, each MPI process writes its local data in a dedicated area within the file. All these areas do not overlap contrarily to the non chunked implementations. As a result the concurrent access issue is easier to handle for the MPI implementation. But this is not true on the VIP machine for unknown reason.

Conclusion 5: Both POSIX multi files and MPI/IO with view mechanism and collective coordination achieve good performance in the context of large data distributed among a large number of cores. The little decay of the MPI_gather + single POSIX file solution is presumably due to the multiple gather operations and not the actual file writing. But this solution is limited by the memory available within a single node which prevents this method to scale with data size. So the MPI_gather + single POSIX file solution should be only used when scaling in data size is not needed. It seems that the POSIX interface with multiple file seems to be the most efficient. A decay of the POSIX multi files performance was expected for large numbers of cores due to the large number of files the file system meta data server has to handle at the same time. But that is not the case, at least when files are written in separate directories, meta data servers (Lustre and GPFS) support the load of a large amount of files (2048 files) simultaneously opened. If this will hold for more than 20K files is an open question here. But once this bottleneck is passed, it is easier to stripe a large amount of independent files than to manage the concurrent access to a single file of a large amount of processes using parallel I/O. However post-processing results coming out of the POSIX multi files approach is more costly from a development point of view.

4 Summary

In this report we have described and tested several methods which enable a parallel application to write data to a storage system. It comes out that many methods are available to perform such accesses, some of them must be avoided and a choice must be taken among the remaining ones. Typically, MPI/IO functions using explicit offsets should be avoided, they never achieve good performance, MPI file pointers should be preferred instead with appropriate MPI data types. Additional implementation work to export chunked dataset is not worth, it does not lead systematically to better performance. Collective calls are also mandatory to achieve good performance in most cases. Parallel HDF5 performance reflects the ones achieved by MPI/IO. So it makes sense to add this software layer in order to get all the additional properties of standardization and portability of an HDF5 file. However, an MPI_gather followed by the export of a POSIX file is still a efficient method and easy to develop when the memory of a single node is not a limitation. It is even far more efficient than parallel I/O for small file sizes. When the data export in separate files is not a problem from a post-processing point of view, then exporting one POSIX file per MPI process is still very efficient on nowadays storage system with up to 2048 cores.

As future work, it would be interesting to test further the scaling of these methods, in particular the multi files approach. A better understanding of the performance issue with HDF5 chunked dataset could also be interesting. Finally, our benchmark tests the parallel

I/O performance in the context of a structured array. It would also be interesting to test also non structured datasets.

References

[1] MPI2 documentation, 1997. http://www.mpi-forum.org/docs/docs.html.

[2] ROMIO MPI/IO implementation, 2003. http://www.mcs.anl.gov/research/projects/romio.

[3] Parallel I/O materials. http://www.cs.dartmouth.edu/pario/.

[4] Existing parallel IO benchmarks. http://www.mcs.anl.gov/~thakur/pio-benchmarks.html.

[5] Rajeev Thakur. Introduction to MPI/IO and MPI2. http://www.classes.cs.uchicago.edu/archive/2000/fall/CS103-01/Lectures/mpi-io/.

[6] Rajeev Thakur, William Gropp, and Ewing Lusk. A case for using MPI's derived datatypes to improve I/O performance. In *Supercomputing '98: Proceedings of the 1998 ACM/IEEE conference on Supercomputing (CDROM)*, pages 1–10, Washington, DC, USA, 1998. IEEE Computer Society.

[7] HDF5 library, 1990. http://www.hdfgroup.org.

[8] Parallel HDF5 library, 2001. www.hdfgroup.org/HDF5/PHDF5.

[9] VIP machine detailed description. http://www.rzg.mpg.de/computing/hardware/Power6/configuration.

[10] HPC for fusion (HPCFF) machine detailed description. http://www.fz-juelich.de/jsc/juropa/configuration.