PIOM-PX: A Framework for Modeling the I/O Behavior of Parallel Scientific Applications

Pilar Gomez-Sanchez[1](✉), Sandra Mendez[2], Dolores Rexachs[1], and Emilio Luque[1]

[1] Computer Architecture and Operating Systems Department, Universitat Autónoma de Barcelona, Campus UAB, Edifici Q, 08193 Bellaterra, Barcelona, Spain
{pilar.gomez,dolores.rexachs,emilio.luque}@uab.es

[2] High Performance Systems Division, Leibniz Supercomputing Centre (LRZ), 85748 Garching bei München, Germany
sandra.mendez@lrz.de

Abstract. Current parallel scientific applications generate a huge amount of data that must be managed efficiently for the HPC storage systems. However, the I/O performance depends on the application I/O behavior and the configuration of the underlying I/O system. To understand the I/O behavior in the software stack and its impact on the I/O operations defined in the application logic, we propose a design framework named PIOM-PX, which allows to define an I/O behavior model based on the I/O phases of HPC applications at POSIX-IO level. We validate our framework using the IOR benchmark for four I/O patterns and we analyze the I/O behavior of NAS BT-IO.

1 Introduction

Nowadays, parallel applications produce a huge amount of data that represents a challenge for modern I/O systems. The variability of the I/O patterns and diversity of storage architectures are other issues that make it difficult to take advantage of the I/O performance capacity of the HPC-IO systems. Depending on the I/O behavior of parallel applications and the processing performed in each layer of the I/O software stack, the performance obtained can differ significantly from the maximum performance expected.

Understanding I/O behavior is fundamental to evaluate the I/O performance of the HPC applications. Several works [1–5] have focused on the extraction of the I/O patterns to understand I/O behavior and to propose techniques to optimize I/O performance in different layers of the I/O software stack [6,7]. Several tools exist to analyze the application's I/O behavior both for performance analysis and for I/O profiling such as Darshan [8] I/O profiling tool, SIOX [9] and Vampir [10] tool.

Due to the fact that most parallel applications are repetitive, and this repetitive behavior for I/O operations is observed as I/O bursts or I/O phases, we use the phase concept as the representation unit of the behavior of parallel applications. In this paper, we present a design framework named PIOM-PX, which allows us to obtain the main parameters at POSIX-IO to define an I/O behavior model.

We use PIOM-PX in order to evaluate the impact of the I/O phases on the I/O system and to replicate the application's I/O behavior in different HPC systems. The I/O phases are determined by identifying the global spatial and temporal pattern for each file opened during the execution of the parallel application. Our approach allows us to determine the I/O requirements of the application and to evaluate their impact on different I/O configurations.

This paper is organized as follows: Sect. 2 describes the proposed framework, Sect. 3 presents the validation of PIOM-PX and Sect. 4 explains the experimental results. Finally, in Sect. 5, we explain our conclusions and future work.

2 Proposed Framework

The I/O model of application is defined based on the I/O phase concept and the key characteristics, which are independent of the I/O system. We classify the application features as parameters for PIOM-PX into three levels: application, file, and phase. Table 1 summarizes the parameters for each level.

We define a design framework to obtain an I/O behavior model at POSIX-IO level named PIOM-PX. Figure 1 presents the steps of PIOM-PX structured in two main stages: tracing and post-processing.

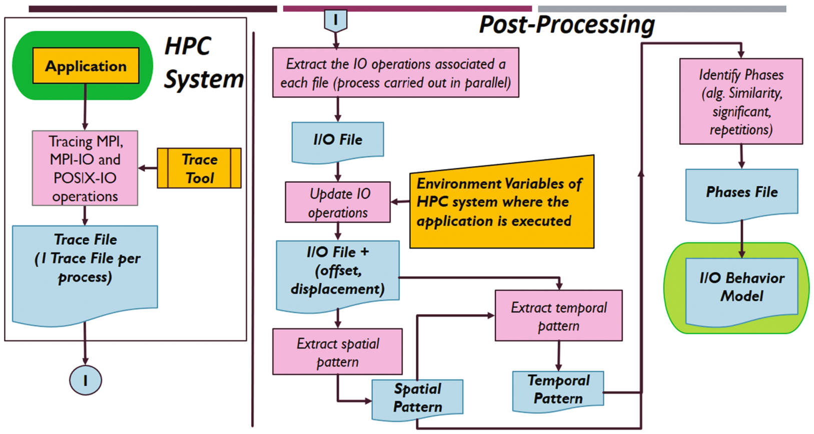

HPC System
Application
Tracing MPI, MPI-IO and POSIX-IO operations
Trace Tool
Trace File (I Trace File per process)
I
Post-Processing
I
Extract the IO operations associated a each file (process carried out in parallel)
I/O File
Update IO operations
Environment Variables of HPC system where the application is executed
I/O File + (offset, displacement)
Extract spatial pattern
Spatial Pattern
Extract temporal pattern
Temporal Pattern
Identify Phases (alg. Similarity, significant, repetitions)
Phases File
I/O Behavior Model

Fig. 1. Framework to extract the application's I/O behavior model based on identifying the I/O phases.

Table 1. PIOM-PX model parameters

Identifier	Application
app_np	Number of processes that the application needs to be executed
app_nfiles	Number of files used by the application
app_st	Storage capacity required by the application for the input files, temporal files and input/output files.
	File
file_id	File Identifier
file_name	File Name
file_size	File Size
file_np	Count of MPI processes that open the file *file_id*
file_accessmode	This can be sequential, strided or random
file_fileaccesstype	Read only(R), write only (W) or write and read (W/R)
file_accesstype	*file_np* processes can access to shared Files or 1 File per Process
file_nphase	Count of phases of the file.
	I/O Phase (PhIO)
Ph_id	Identifier of an I/O Phase
Ph_processid	Identifier of Process implied in the phase
Ph_np	Number of processes implied in the phase
Ph_weight	Transferred data volume during the phase. It is expressed in bytes
Ph_nrep	Number of repetitions per phase
Ph_niop	Number of I/O operations
IOP	Data access operation type, which can be write, read, or write/read
rs	Request size or size of an I/O operation
offset	Operation offset, which is a position in the file's logical view
disp	Displacement into file, which is the difference between the offset of two consecutive I/O operations
dist	Distance between two I/O operations, which is the difference between

2.1 Tracing I/O Operations

To obtain the information defined in Table 1, we have implemented a tracer to extract POSIX-IO events and to assign additional fields to detect I/O phases. This tracer was integrated with PIOM-MP (former PAS2P-IO), which allows us to trace I/O activities at MPI and POSIX-IO level.

Table 2 describes the fields included in a trace line (TL) of PIOM-PX. The events are traced between the `MPI_Init` and the `MPI_Finalize` operations and a trace file is generated for each MPI process. We trace the following operations:

POSIX-IO
```
open, open64, fopen64, close, fclose, write, fwrite, read, pread
pread64, pwrite, pwrite64,fread, fwrite, lseek, lseek64, fsync
creat, creat64, readv, writev, fseek, xstat, xstat64
```

Communication and MPI-IO
```
MPI_Send, MPI_Isend, MPI_Recv, MPI_Irecv, MPI_Wait, MPI_Allgather
MPI_Allreduce, MPI_Barrier, MPI_Bcast, MPI_Reduce, MPI_Sendrecv
MPI_Waitall, MPI_File_* // 51 I/O operations.
```

We define the tick concept to register the order of the MPI events and the subtick concept for POSIX-IO events. The tick is increased for each MPI event detected and the subtick is initialized after each MPI event and incremented for consecutive POSIX-IO events.

Table 2. PIOM-PX trace line

Identifier	Description
IdProcess	Identifier of Process
file_id	Identifier of File
TypeOperation	"MPI" or "POSIX"
NameOperation	Name of POSIX-IO event
offset	Operation offset, which is a position in the file's logical view
rs	Request size of for data access operations
	Metadata-line
file_name	File Name
FileAccessType	Open mode
	Added fields
Time	Logical time of the occurrence of a MPI or POSIX-IO event
Final_compute	Duration of the call of an MPI or POSIX-IO event
tick	Order of occurrence of the MPI events
subtick	Order of occurrence of the POSIX-IO events

In the *Extracting I/O operations* step, we extract the I/O operations per file opened by the application of each trace file into a new file. Therefore, from this step we obtain as many files as the application opens during its execution.

2.2 Updating I/O Operations

In this point, every file of I/O operations is reviewed to determine whether the offset and request size (rs) informed require evaluating another operation to obtain the real request or offset. For example, the case of the `write` and `read` operations, where the offset depends on `lseek` operation.

To modify the offset, we have to take into account the `whence` parameter of `lseek` operation: `SEEK_SET` (the file offset is set to `offset` bytes), `SEEK_CUR` (the file offset is set to its current location plus `offset` bytes) and `SEEK_END` (the file offset is set to the size of the file plus `offset` bytes).

We calculate the field displacement (*disp*), added in the *TL* structure, to identify the request size (*rs*) and how the displacement moves.

For Fortran program, the environment variable `FORT_BLOCKSIZE` is evaluated to determine the request size of a POSIX-IO event that the user actually wants to work with.

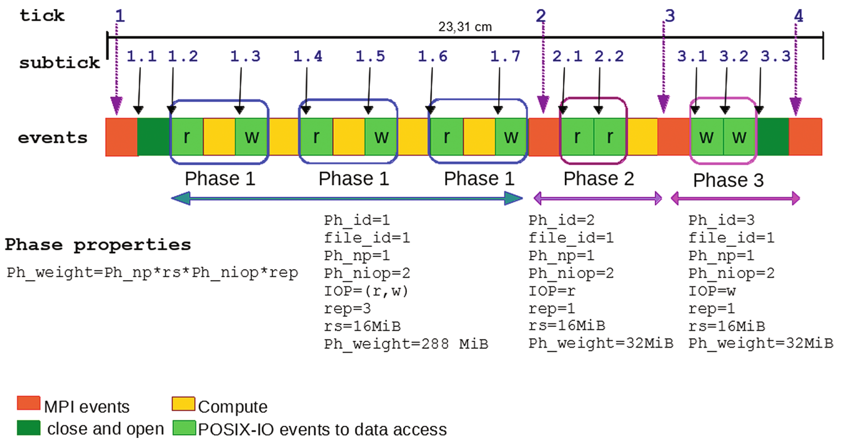
2.3 Extracting Spatial and Temporal Pattern

To extract the spatial pattern, for each I/O file, both for the write and read operation, we save the following fields: `NameOperation, file_id, file_name, offset and rs`.

Besides, we calculate the offset difference for all operations that have the same *file_id*, *file_name* and *rs*. The offset difference is calculated between two consecutive operations (read-read or write-write) and the displacement (*disp*) is calculated between two write operations and read operations.

If the application uses a shared file, we identify the global spatial pattern based on I/O operations traced for each MPI process that opens the shared file.

To detect the Temporal pattern we establish the *tick* and *subtick* (See Fig. 2). The tick identifies the MPI and MPI-IO operations and the subtick identifies the POSIX-IO operations. `MPI_Sends` are interchanged between processes and



Ph_weight=Ph_np*rs*Ph_niop*rep

Phase 1	Phase 2	Phase 3
Ph_id=1	Ph_id=2	Ph_id=3
file_id=1	file_id=1	file_id=1
Ph_np=1	Ph_np=1	Ph_np=1
Ph_niop=2	Ph_niop=2	Ph_niop=2
IOP=(r,w)	IOP=r	IOP=w
rep=3	rep=1	rep=1
rs=16MiB	rs=16MiB	rs=16MiB
Ph_weight=288 MiB	Ph_weight=32MiB	Ph_weight=32MiB

Fig. 2. Representation of the I/O phases of a parallel application. The view corresponds to an I/O process for an access type 1 file per process. Tick and subtick are used to obtain the order of occurrence of the application's events. An I/O phase is a consecutive sequence of similar I/O operations. Phase properties represent the transferred data volume during a phase and the I/O pattern.

the tick of `MPI_Send`s affects the operations of the processes that received this `MPI_Send`. If all processes write to one or more shared files, we must detect the relationship between all the operations carried out by all the processes to determine the actual logical order. This order helps us to detect dependencies among all processes and it is necessary to redefine the ticks and the subticks.

3 Experimental Validation

In this section, we validate the PIOM-PX functionality and its integration with PIOM-MP (former PAS2P-IO [4]). To do this, we define four experiments based on IOR [11] benchmark, which allows us to generate different I/O patterns for distinct I/O interfaces. We have executed IOR using intel and GNU compilers to analyze their influence on the operations detected. The MPI distribution utilized was Intel MPI 2017. Furthermore, to identify the impact of the parallel file system at the POSIX-IO level, PIOM-PX is evaluated in two HPC systems with IBM Spectrum Scale (former GPFS) and Lustre. Experiments were executed in SuperMUC (LRZ) and Finisterrae2 (CESGA) supercomputers, which are described in Table 3.

Table 3. HPC systems

Components	Finisterrae2	SuperMUC
Compute nodes	306	9216
CPU cores (per node)	24	16
RAM memory	128 GB	32 GB
Local Filesystem	ext4	ext3
Global Filesystem (GFS)	NFS	NFS
Capacity of GFS	1.1 TB	10×564×10TB
Global Filesystem (PFS)	Lustre	GPFS
Capacity of PFS	695 TB	12 PB
Data servers	4 OSS and 12 OSTs	80 NSD
Metadata servers	1	
Stripe size	1 MiB	8 MiB
Interconnection	IB FDR@56 Gbps	IB FDR10

3.1 Experimentation

Four experiments were designed to evaluate the I/O strategies `1 File per Process` and `1 Single Shared File`. Furthermore, we assess a nested strided pattern by using the collective buffering technique in "enable" and "automatic" mode. We executed the experiments for 16 MPI processes per compute node. Each experiment is described as follows:

(a) 1 File per Process using POSIX interface:
   - Objective: Detect the POSIX-IO operations for an application that only uses POSIX as I/O library.
   - Command Line:

```
IOR -a POSIX -s 1 -b 8m -t 1m -F
```

(b) 1 File per Process using MPI-IO interface:
   - Objective: Detect the POSIX-IO operations generated by an application that uses independent MPI-IO operations.
   - Command Line:

```
IOR -a MPIIO -s 1 -b 8m -t 1m -F
```

(c) A single shared file using collective buffering technique in `automatic` mode for a strided pattern:
   - Objective: Detect the POSIX-IO operations generated by an application that uses collective MPI-IO operations.
   - Command Line:

```
IOR -c -a MPIIO -s 16 -b 512k -t 512k
```

(d) A single shared file using collective buffering technique in `enable` mode for a strided pattern:
   - Objective: Detect the POSIX-IO operations generated by an application that uses collective operations with the collective buffering technique enabled.
   - Command Line:

```
romio_cb_read = enable
romio_cb_write = enable
IOR -c -a MPIIO -s 16 -b 512k -t 512k
```

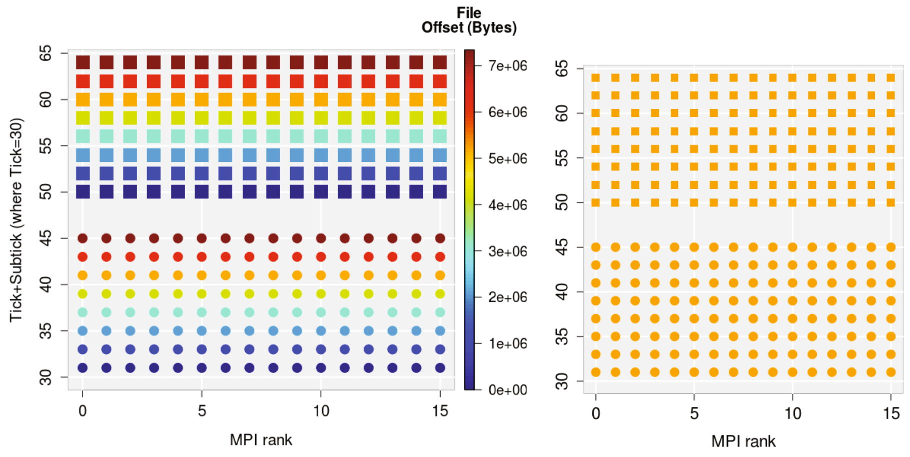
Table 4 presents PIOM-PX model parameters for the four IOR experiments at application and file level.

To explain the I/O phases detected, we present snippets of trace files for each experiment, where lines with "##" present the selected field of a trace line. Furthermore, we detected two I/O phases for the four experiments because this depends on the IOR logic. For this reason, we only show a detailed figure of the I/O behavior for experiment (a). Each experiment is explained as follows:

(a) 1 File per Process using POSIX interface: IOR is configured to write 8 MiB per MPI process by using the POSIX interface for the I/O strategy 1 File per process. Each MPI process writes and reads in request size of 1 MiB ($rs = 1$ MiB). Figure 3 shows the I/O behavior for experiment (a). We detect two I/O phases per file composed of eight operations ($Ph\_niop = 8$) each. For each file, the first phase corresponds to 8 write operations and the second phase to 8 read operations. Sixteen files are accessed in parallel. IOR starts with a communication burst of 30 events between process 0 and the rest of the processes (See Fig. 3). Later, a write phase begins in tick 30. In the tick+subtick 50, an I/O Phase of read operations is generated.

Table 4. PIOM-PX Parameters for the IOR Benchmark

Identifier	(a)	(b)	(c)	(d)
app_np	16	16	16	16
app_nfiles	16	16	1	1
app_st	128 MiB	128 MiB	128 MiB	128 MiB
File				
file_name	testFile<IdProcess>	testFile<IdProcess>	testFile	testFile
file_size	8 MiB	8 MiB	128 MiB	128 MiB
file_accessmode	Seq	Seq	Strided	Strided
file_fileaccesstype	W/R	W/R	W/R	W/R
file_accesstype	1Fx1Proc	1Fx1Proc	Shared	Shared
file_nphase	2	2	2	2
file_np	1	1	16	16



(a) File offset (b) Phase Weight

Fig. 3. IOR for POSIX interface configured for 16 MPI processes, 1 File per Process for a sequential pattern. Bullet (smaller circle) corresponds to write operations and the filled squares to read operations. Two I/O phases can be observed for each file. The Phase 1 is composed of 8 write operations and Phase 2 of 8 read operations. The color scale in (b) shows the weight, which is 1 MiB × *file_np* per subtick. The weight for both Phase 1 and Phase 2 is 8 MiB for each file per process. (Color figure online)

Snippet 1 presents part of the trace file of *IdProcess* 2, which shows part of the operations of Phase 1. We can observe that a `lseek64` operation is called before each `write` and this also occurs for `read` operations.

Snippet 1: Trace file of *IdProcess* 2
```
## IdProcess file_id file_name NameOperation  tick subtick
2 6 testFile.00000002 open64 30 0
## IdProcess file_id NameOperation offset tick subtick
2 6 lseek64 0 30 1
## IdProcess file_id NameOperation offset rs tick subtick
2 6 write 0 1048576 30 2
2 6 lseek64 1048576  30 3
2 6 write 0 1048576 30 4
...
```

(b) 1 File per Process using MPI-IO interface: the I/O phases for this case is similar to experiment (a) (See Fig. 3). In Snippet 2, a part of the *IdProcess* 2 trace file for this experiment can be seen. The number of I/O operations at POSIX-IO level changes, a `write` operation is called for each `MPI_File_write_at`. For read case, each `MPI_File_read_at` calls a `read` operation.

Snippet 2: Trace file of *IdProcess* 2
```
## IdProcess file_id NameOperation file_name tick
2 0x6e38b8 MPI_File_open testFile.00000002 31
## IdProcess file_id file_name NameOperation tick subtick
2 22 testFile.00000002 open64 31 0
## IdProcess file_id NameOperation offset rs tick
2 0x6e38b8 MPI_File_write_at 0 1048576 32
## IdProcess file_id NameOperation offset rs tick subtick
2 22 write 0 1048576 32 0
2 0x6e38b8 MPI_File_write_at 1048576 1048576 33
2 22 write 0 1048576 33 0
 ...
```

(c) A single shared file using collective buffering technique in `automatic` for a strided pattern:

Snippet 3: Trace file of *IdProcess* 0
```
## IdProcess file_id NameOperation file_name tick
0 0x2124fc8 MPI_File_open testFile3 31
## IdProcess file_id file_name NameOperation tick subtick
0 6 testFile3 open64 0 66 31 0
0 0x2124fc8 MPI_File_get_info 32
## IdProcess file_id NameOperation offset rs tick
0 0x2124fc8 MPI_File_write_at_all 0 524288 33
## IdProcess file_id NameOperation offset rs tick subtick
0 6 write 0 524288 33 0
0 0x2124fc8 MPI_File_write_at_all 8388608 524288 34
## IdProcess file_id NameOperation offset tick subtick
0 6 lseek64 8388608 34 0
0 6 write 0 524288 34 1
...
```

We define the strided pattern by setting the parameters blocksize (-b) and transfer size (-t) with the same value. Furthermore, to obtain a total I/O equal to experiment (a) and (b), the segment count (-s) is set to 16. In total, each process writes and reads 8 MiB using a request size of 512 KiB.

In Snippet 3, we can observe a `lseek64` and `write` operation for each `MPI_File_write_at_all`, except for the first collective write. The displacement is equal to $file\_np \times rs = 8388608$ Bytes, where $file\_np = 16$ and $rs = t = 524288$ Bytes. In this case, each MPI process produces a similar trace file, with the exception of the offset, where the initial offset is equal to $IdProcess \times rs$ and $offset(i+1) = offset(i) + rs \times file\_np \times (i-1) + IdProcess \times rs$ with $i \in \{1..s\}$, where `s` is the number of segments set up for IOR benchmark.

(d) A single shared file using collective buffering technique enabled for a strided pattern:

Snippet 4: Trace file of IdProcess 0
```
## IdProcess file_id NameOperation file_name tick
0 0xac80f0 MPI_File_open testFile3 31
## IdProcess file_id file_name NameOperation  tick subtick
0 6 testFile3 open64 31 0
0 0xab9338 MPI_File_get_info 32
## IdProcess file_id NameOperation offset rs tick
0 0xab9338 MPI_File_write_at_all 0 524288 33
### IdProcess file_id NameOperation offset rs tick subtick
0 6 write 0 8388608 33
0 0xab9338 MPI_File_write_at_all 8388608 524288 34
## IdProcess file_id NameOperation offset tick subtick
0 6 lseek64 8388608 34 0
0 6 write 0 8388608 34 1
...
```

To trace the I/O operations for applications that use collective buffering enabled, we set up the ROMIO hints for this technique. The strided pattern discussed in experiment (c) is the same than we employed in this experiment. In Snippet 4, we can observe similar I/O operations to experiment (c), but the request size at POSIX level is different, in this case, it corresponds to $file\_np \times rs(MPI) = 8388608$ Bytes, where $rs(MPI)$ is the request size of the MPI-IO operations. This behavior is to be expected, because we select the IOR parameters to observe the collective behavior at POSIX-IO level.

3.2 Discussion

IOR benchmark allows us to reproduce more common I/O patterns of HPC applications. PIOM-PX detected the spatial and temporal pattern for POSIX-IO level and it represented them through I/O phases.

The results showed that the spatial pattern depends on the type of I/O operation and I/O method. We have selected the same amount of data, the number of MPI processes and similar I/O strategy, but the behavior is influenced by the I/O techniques and the I/O interface. We have detected that collective buffering technique was not working in `automatic` mode because all MPI processes were carrying out I/O and we only expected an I/O aggregator per compute node. As has been observed in experiments (c) and (d), MPI-IO operations are the same, but at POSIX level they depend on the hint values of the ROMIO library and the hints explicitly defined in the application. PIOM-PX considers this property to provide information to the user in order to help them understand what the I/O library is doing with the operations defined in the application logic.

4 Experimental Results

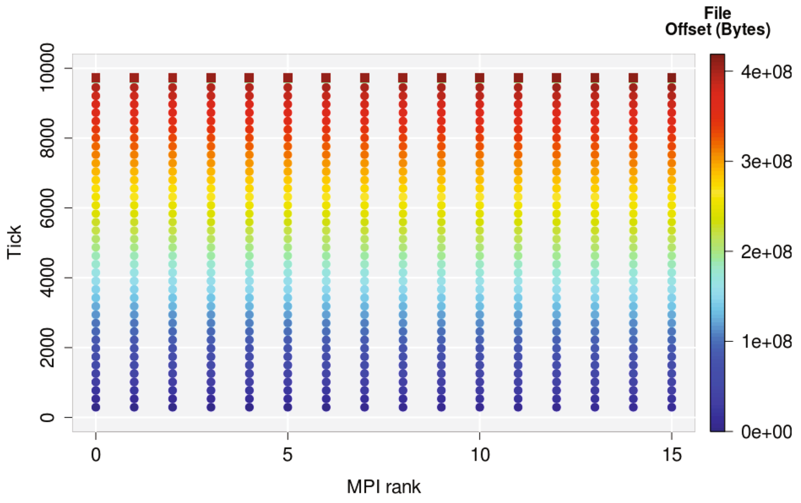
In this section, we analyze the BT-IO benchmark [12], which is part of the parallel benchmark suite NPB-MPI developed by the NASA Advanced Supercomputing Division. BT-IO presents a block-tridiagonal partitioning pattern on a three-dimensional array across a square number of processes.

Table 5. PIOM-PX parameters for the BT-IO benchmark subtype FULL

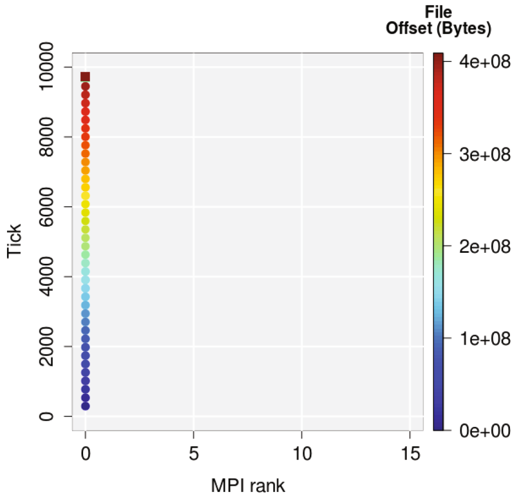
Identifier	Class A	Class B	Class C
app_np	16	16	36
app_nfiles	1	1	1
app_st	400 MiB	1.6 GiB	6.4 GiB
File			
file_name	btio.full.out	btio.full.out	btio.full.out
file_size	400 MiB	1.6 GiB	6.4 GiB
file_accessmode	Strided	Strided	Strided
file_fileaccesstype	W/R	W/R	W/R
file_accesstype	Shared	Shared	Shared
file_nphase	41	41	41
file_np at MPI-IO level	16	16	36
file_np at POSIX-IO level	1	1	3

We selected BT-IO to show the temporal pattern considering the tick and subtick concepts. Due to the fact that an I/O phase is identified depending on the communication events and compute part, BT-IO allows us to analyze a case with compute and communication events. BT-IO is implemented in Fortran, therefore we can evaluate the influence of Fortran I/O library in the request size at POSIX-IO level. We have selected the subtype FULL, which implements the I/O part with collective operations, derived data type, `MPI_File_view` and `MPI_Info` for enabled collective buffering in the application logic. We have executed BT-IO in SuperMUC and Finisterra2 supercomputers (See Table 3).

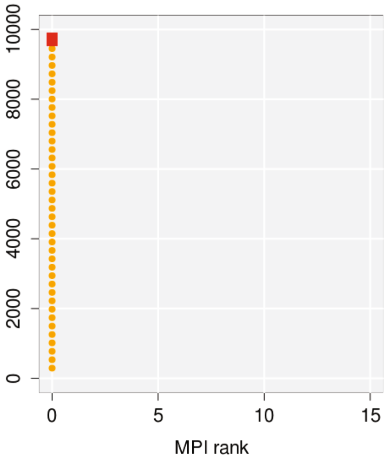
Figure 4 depicts the I/O phases at MPI-IO (Fig. 4(a)) and POSIX-IO (Fig. 4(b)) level. PIOM-PX parameters are described in Table 5 for Classes A, B



(a) File offset at MPI level by using PIOM-MP



(b) File offset at POSIX-IO level



(c) Phase Weight at POSIX-IO level

Fig. 4. BT-IO subtype FULL, Class A using 16 MPI processes for a strided pattern. The bullets (smaller circles) correspond to write operations and the shaded squares to read operations. Each first forty I/O phases (circles) are composed of 1 MPI write operation and Phase 41 of 40 read operations. At MPI-IO level, the weight of the Phase 1 to Phase 40 is 655360 Bytes × $file\_np$ for each one and for Phase 41 it is 10 MiB × np × 40. At POSIX-IO level, the colored scale in Fig. 4(c) shows the weight for Phase 1 to Phase 40, which is 10 MiB and the Phase 41 weight is 10 MiB ×40.

and C at application and file level. As can be observed in Table 5, 41 I/O phases are identified in a single shared file for a strided access mode.

In Fig. 4(a) each bullet line (y-axis) represents an I/O phase composed of *file_np* write operations. The red square represents Phase 41, which is composed of 40 × *file_np* read operations. The operation size is similar for read and write operations. At MPI-IO level, the number of MPI processes per I/O phase correspond to the *file_np*. In Fig. 4(b), we can observe the effect of the collective buffering techniques at POSIX level, where only process 0 performs I/O operations. In this layer, the number of processes per I/O phase is equal to the number of compute nodes utilized for running the application.

5 Conclusions

We have validated PIOM-PX with the IOR benchmark for four cases. Our approach allows us to obtain the application's I/O behavior at phase level. The I/O behavior helps to understand the relationship between the application and the I/O system. PIOM-PX is modular to facilitate the integration of more functionality or steps. Our framework makes it possible to have accurate information over the I/O phases. Despite the fact that number of MPI processes evaluated in validation and experimentation is small, as the I/O behavior model depends on the application logic, our approach is applicable for a larger number of MPI processes.

The next step, it is to execute the real applications and to acquire their I/O behavior at I/O phase level.

Acknowledgments. This research has been supported by the MINECO Spain under contract TIN2014-53172-P. The research position of the PhD student P. Gomez has been funded by a research collaboration agreement, with the "Fundación Escuelas Universitarias Gimbernat". P. Gomez awarded with the SEBAP Research Mobility Grant to fund her three-month research stay at Leibniz Supercomputing Centre (LRZ, Germany).

The authors thankfully acknowledge the resources provided by the Centre of Supercomputing of Galicia (CESGA, Spain) and the Leibniz Supercomputing Centre (LRZ, Germany).

References

1. Byna, S., Chen, Y., Sun, X.-H., Thakur, R., Gropp, W.: Parallel I/O prefetching using MPI file caching and I/O signatures. In: Proceedings of the 2008 ACM/IEEE Conference on Supercomputing, SC 2008, Piscataway, NJ, USA, pp. 44:1–44:12. IEEE Press, 2008. http://dl.acm.org/citation.cfm?id=1413370.1413415
2. He, J., Bent, J., Torres, A., Grider, G., Gibson, G., Maltzahn, C., Sun, X.-H.: I/O acceleration with pattern detection. In: Proceedings of the 22nd International Symposium on High-Performance Parallel and Distributed Computing, pp. 25–36. ACM (2013)

3. Kluge, M., Knüpfer, A., Müller, M., Nagel, W.E.: Pattern matching and I/O replay for POSIX I/O in parallel programs. In: Sips, H., Epema, D., Lin, H.-X. (eds.) Euro-Par 2009. LNCS, vol. 5704, pp. 45–56. Springer, Heidelberg (2009). doi:10.1007/978-3-642-03869-3_8
4. Méndez, S., Rexachs, D., Luque, E.: Modeling parallel scientific applications through their Input/Output phases. In: CLUSTER Workshops, vol. 12, pp. 7–15 (2012)
5. Carns, P., Harms, K., Allcock, W., Bacon, C., Lang, S., Latham, R., Ross, R.: Understanding and improving computational science storage access through continuous characterization. Trans. Storage **7**(3), 8:1–8:26 (2011). doi:10.1145/2027066.2027068
6. Behzad, B., Luu, H.V.T., Huchette, J., Byna, S., Prabhat, Aydt, R., Koziol, Q., Snir, M.: Taming parallel I/O complexity with auto-tuning. In: 2013 SC - International Conference for High Performance Computing, Networking, Storage and Analysis (SC), pp. 1–12, November 2013
7. Behzad, B., Byna, S., Prabhat, Snir, M.: Pattern-driven parallel I/O tuning. In: Proceedings of the 10th Parallel Data Storage Workshop, PDSW 2015, pp. 43–48. ACM, New York (2015). doi:10.1145/2834976.2834977
8. Carns, P., Latham, R., Ross, R., Iskra, K., Lang, S., Riley, K.: 24/7 Characterization of petascale I/O workloads. In: 2009 IEEE International Conference on Cluster Computing and Workshops, pp. 1–10. IEEE (2009)
9. Kunkel, J.M., Zimmer, M., Hübbe, N., Aguilera, A., Mickler, H., Wang, X., Chut, A., Bönisch, T., Lüttgau, J., Michel, R., Weging, J.: The SIOX architecture – coupling automatic monitoring and optimization of parallel I/O. In: Kunkel, J.M., Ludwig, T., Meuer, H.W. (eds.) ISC 2014. LNCS, vol. 8488, pp. 245–260. Springer, Cham (2014). doi:10.1007/978-3-319-07518-1_16
10. Knüpfer, A., et al.: The Vampir performance analysis tool-set. In: Resch, M., Keller, R., Himmler, V., Krammer, B., Schulz, A. (eds.) Tools for High Performance Computing, pp. 139–155. Springer, Heidelberg (2008). doi:10.1007/978-3-540-68564-7_9
11. Loewe, W., MacLarty, T., Morrone, C.: IOR Benchmark (2012). https://github.com/chaos/ior/blob/master/doc/USER_GUIDE. Accessed 14 May 2016
12. Wong, P., Wijngaart, R.F.V.D.: NAS parallel benchmarks i/o version 2.4, Computer Sciences Corporation, NASA Advanced Supercomputing (NAS) Division, Technical report (2003)