Exploiting data compression in collective I/O techniques

Rosa Filgueira, David E. Singh, Juan C. Pichel,
and Jesús Carretero

Department of Computer Science
University Carlos III of Madrid - Spain
{rosaf, desingh, jcpichel, jcarrete}@arcos.inf.uc3m.es

Abstract

This paper presents Two-Phase Compressed I/O (TPC I/O,) an optimization of the Two-Phase collective I/O technique from ROMIO, the most popular MPI-IO implementation. In order to reduce network traffic, TPC I/O employs LZO algorithm to compress and decompress exchanged data in the inter-node communication operations. The compression algorithm has been fully implemented in the MPI collective technique, allowing to dynamically use (or not) compression. Compared with Two-Phase I/O, Two-Phase Compressed I/O obtains important improvements in the overall execution time for many of the considered scenarios.

1 Introduction

A large class of scientific applications operates on a high volume of data that needs to be persistently stored. Parallel file systems such as GPFS [10], PVFS [7] and Lustre [8] offer scalable solutions for concurrent and efficient access to storage disk space. These parallel file systems are accessed by the parallel applications through interfaces such as *POSIX* [21] or MPI-IO [20]. This paper targets the optimization of the MPI-IO interface inside ROMIO [6], the most popular MPI-IO distribution.

Many parallel applications (especially related to simulations) consists of alternating compute and I/O phases. During the compute phase, the simulated process evolves to new states. These states have to be periodically stored to disk in order to recover the system in case of fault (checkpointing). During the I/O phase, the processes frequently access a common data set by issuing a large number of small non-contiguous I/O requests [14, 15]. Usually, these requests originate an important performance slowdown of the I/O subsystem. Collective I/O addresses this problem by merging small individual requests into larger global requests in order to optimize the network and disk performance. Depending on the place where the request merging occurs, one can identify two collective I/O methods. If the requests are merged at the I/O nodes the method is called *Disk-Directed I/O* [4, 16]. If the merging occurs at intermediary nodes or at compute nodes the method is called *Two-Phase I/O* (*TP I/O*) [2, 1].

In this work we focus on the *TP I/O* technique, extended by Thakur and Choudhary in ROMIO. Based on it we have developed and evaluated the *Two-Phase Compressed I/O* for different scenarios, a technique in which the data are compressed during the communications of the merging stage. The main goal of our research is to use compresion to enhance overall execution time without affecting the application or the original programm. We dont pursue reducing the size of the data stored, as this would require to change data format and to modify the file system. To achieve that, we have used the LZO [18] algorithm, a portable lossless data compression technique. It offers pretty fast compression and extremely fast decompression in real time. The comparison with the original *TP I/O* technique shows that our approach obtains for many of the considered scenarios, important reductions in the execution time when compression is used. This is achieved by reducing the size of the messages, and therefore, reducing the total communication time. Additionally, *TPC I/O* allows to dynamically select to use (or not) compression.

This paper is structured as follows: Section 2 contains the related work. Section 3 explains the internal structure of *Two-Phase I/O*. Section 4 presents the *Two-Phase Compressed I/O*. Section 5 is dedicated to the performance evaluation of both techniques. Finally, in Section 6 we present the main conclusions derived from this work.

2 Related work

There are several collective I/O implementations, based on the assumption that several processes access concurrently, interleaved and non-overlapping to a file (a common case for parallel scientific applications). In disk-directed I/O [4], the compute nodes fordward file requests to the I/O nodes. Then, the I/O nodes merge and sort the requests before sending them to disk. In server-directed I/O of Panda [16], the I/O nodes sort the requests on file offsets instead of disk addresses. *Two-Phase I/O* [2, 1] consists of an access phase, in which compute nodes exchange data with the file system according to the file layout, and a shuffle phase, in which compute nodes redistribute the data among each other according to the access pattern. Lustre file joining (merging two files into one) for improving collective I/O is presented in [17].

Several researchers have contributed with optimizations of MPI-IO data operations: data sieving [6], non-contiguous access [11], collective caching [12], cooperating write-behind buffering [13], integrated collective I/O and cooperative caching [9].

In a previous work, we presented the *Locality-Aware Two-Phase (LATP)* I/O [19] technique (*LATP*), an optimization of the *Two-Phase collective I/O*. Both techniques can perform (in combination with MPI file views) generic file access. That is, they can be used to access in parallel to contiguous and non-contiguous portions of a given file. In order to increase the locality of the file accesses, *LATP* employs the *Linear Assignment Problem (LAP)* for finding an optimal distribution of data to processes, an aspect that is not considered in the original technique. This assignment is based on the local data that each process stores, and its main purpose is the reduction of the number of communications involved in the I/O collective operation and, therefore, the improvement of the global execution time. We also used compression to enhance execution time in COMPASSION library [22], a version of the out of core library PASSION.

Collective I/O (IEC I/O) is presented. This technique takes advantage of fast communication networks for exchanging the data, so that the I/O locality is improved. This increases the communication efficiency, reducing the cost of the global I/O operation.

3 Internal structure of Two-Phase I/O

Two-Phase I/O is a collective I/O technique implemented in ROMIO. In this paper we focus in *Two-Phase I/O* for the collective writing of data. The algorithm has two phases: the first phase consists of the data exchange among the processes that take part in the writing operation, whereas the second phase is the collective writing of data to disk.

Regarding the internal structure of *TP I/O*, it is divided into several stages which can be summarized as follows:

- *Offsets and lengths calculation (st1)*: In this stage the lists of offsets and lengths of the file are calculated.
- *Offsets and lengths communication (st2)*: Each process communicates its start and end offsets to the other processes, so that all processes have global information about the involved file interval that is going to be accessed.
- *File domain calculation (st3)*: The I/O workload is divided among processes. This is done by dividing the file into file domains (FDs). In this way, in the following stages, each aggregator collects and transfers to the file the data associated to its FD.
- *Access request calculation (st4)*: It calculates the access requests for the file domains of each remote process.
- *Metadata transfer (st5)*: Transfer the lists of offsets and lengths.
- *Buffer writing (st6)*: Data are sent to appropriate processes. First, each process posts all of its receives (using MPI_Irecv), and then posts all of its sends (using MPI_Isend). The process then waits for all messages it needs to receive, and copies the data into the write buffer. In this way, the write buffer is filled with all the gathered data assigned to this process.
- *File writing (st7)*: The data placed in the write buffer. These data correspond to the contiguous region (the FD) of each process. Thus, the number of I/O requests is reduced and the overall I/O performance is improved. Note that by using the MPI view operation, a given FD can be mapped to generic file elements (non-contiguous writes).

The buffer and file writing stages (*st6* and *st7*), are repeated as many times as the result of dividing the size of the file portion of each process by the size of the *TP I/O* buffer (4 MB per process, by default in our current implementation). First, the write size of each process is obtained by dividing the size of the file by the number of processes. For example, if the size of the file is 552MB[1] and the number of processes is 8, the write size of each process is 69 MB. This is the file domain of each process. Then, the file size related to each process is divided in as many chunks as buffers fit in each file domain. For example, a 69MB file domain requires of 69/4 buffers to cover it.

[1] As we comment in Section 5.1, this file is generated by our benchmark when mesh 4 is stored using a *load* of 500.

Note that only four of all these stages have associated communication operations: st2, for exchanging the lists offsets and length of the operations; st5, for exchanging the offset-length data structures; st6, for transffering the data; and st7 for transferring the data to the filesystem (disk access).

4 Two-Phase I/O combined with compression

In general, a parallel system allows using user-customized MPI libraries (installed in the local account). However, the filesystem library is usually read-only and its modification are usually restricted. In this work, we have focused on the communication operations internally performed in *TP I/O*. More specifically, we have applied compression techniques to the offset-length data exchange (st5) and to the data suffering (st6). Both stages are internally performed in *TP I/O*, and thus, we only need to apply changes to this routine (belonging to ROMIO). In contrast, we don't apply these techniques to the disk access phase (st7) given that it would imply changes in the parallel filesystem library (which usually cannot be modified). Besides, we have decided, not to apply the compression in the stage st2, because it has very small communications.

As we can see in the Figures 1(a) and 1(b), the cost of these phases increase with the number of processes, due to increasing number of messages to transfer among processes.

In this paper we have developed a novel technique called *TwoPhase Compressed I/O* (*TPC I/O*) for reducing the volume of communications, thus enhancing exceution time.

We have used the LZO library for compressing the data. This library offers pretty fast compression and decompression in real time, favouring speed over compression ratio. It has been chosen because it provides good compression ratios with reduced compression times.

Therefore, as shown in Figure 2, before any MPI_Isend message, the set of data to sent is compressed. In the same way, after receiving the exchanged data, a decompression phase is performed. In this way, on-line data compression allows reducing the size of the messages sent, and the cost of the total communication phases. In this example, the original algorithm (*TP I/O*) sends 250MB of data from node 0 to node 1. By means of our technique, 250 MB are compressed to 10 MB, then sent by P0, then received by P1, and afterwards decompressed back to 250 MB. Note that only 10 MB of data are transferred.

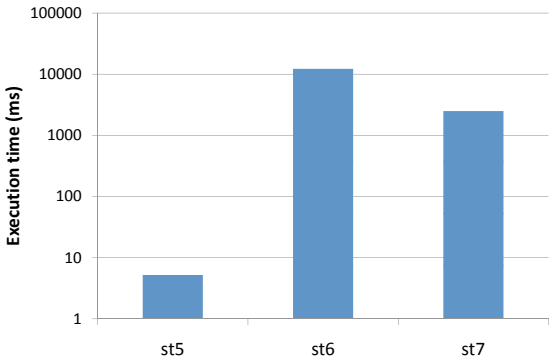
5 Performance Evaluation

We have evaluated our approach using the *BIPS3D* application with different input meshes related to different semiconductor devices. We have compared *TPC I/O* with the original version of the technique (*TP I/O*) implemented in MPICH.

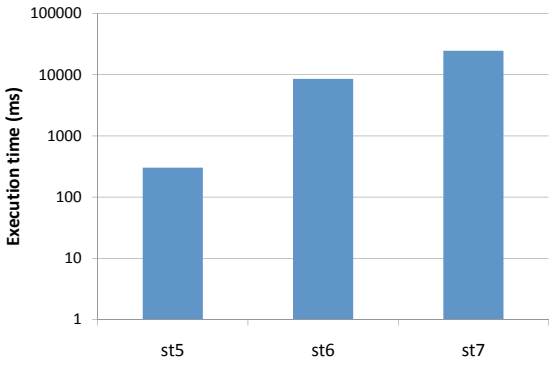
We have used a cluster with 40 nodes for our tests. Each one is Dual-Core AMD with 512 MB of RAM. The interconnection network is FastEthernet.

We have used the MPICHGM 2.7.15NOGM distribution and developed our technique by modifying the *TP I/O* implementation of ROMIO. The parallel file system used is PVFS 1.6.3 with one metadata server and 8 I/O nodes with a striping factor of 64KB.

We have evaluated both techniques when applied to the I/O stage of the *BIPS3D* Simulator. This application is described next.



(a)



(b)

Figure 1. Stages of Two-Phase I/O for mesh 1: (a) with $load$ 200 and 8 processes and (b) with $load$ 200 and 32 processes.

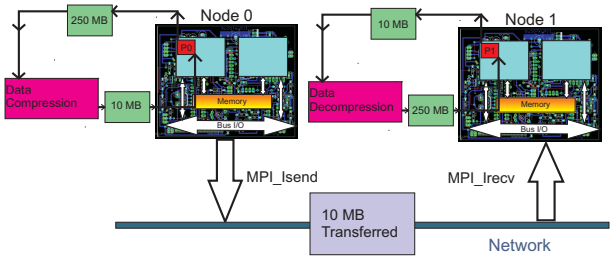


Figure 2. Example of data Compression

5.1 BIPS3D Simulator

BIPS3D is a 3-dimensional simulator of BJT and HBT bipolar devices [5]. The goal of the 3D simulation is to relate electrical characteristics of the device with its physical and geometrical parameters. The basic equations to be solved are Poisson's equation and electron and hole continuity in a stationary state.

Finite element methods are applied in order to discretize the Poisson equation, hole and electron continuity equations by using tetrahedral elements. The result is an unstructured mesh which is distributed among the compute nodes. In this work, we have used four different meshes, as described later.

Using the METIS [3] library, this mesh is divided into sub-domains, in such a manner that each sub-domain corresponds to one process. Then we construct, for each sub-domain, in a parallel manner, the part corresponding to each part of the distributed mesh. Each sub-domain is solved using domain decomposition methods, and finally the results are written to a file. For our evaluation BIPS3D has been executed using four different meshes: mesh 1 (47200 nodes), mesh 2 (32888 nodes), mesh 3 (73257 nodes) and mesh 4 (289648 nodes), with different number of processes: 4,8, 16, and 32. The BIPS3D associates a data structure to each node of a mesh. The contents of these data structures are the data written to disk during the I/O phase. The number of elements that this structure has per each mesh entry is variable (depends of each simulation configuration) and is given by the *load* parameter. This means that, given a mesh and a $load$, the number of data written is the product of the number of mesh elements and the $load$. In other words, for each element of the mesh written to disk, a structure of $load$ elements in actually written. In this work we have evaluated different $loads$, specifically, 100, 200, 500 and 1000 words (with 4 bytes). Table 1 lists the different sizes (in MB) of the file associated to each mesh and $load$ when using integer data. Results are shown based on $load$ and mesh characteristics. If we use float data, the sizes will be double.

Load per node	$mesh1$	$mesh2$	$mesh3$	$mesh4$
100	18	12	28	110
200	36	25	56	221
500	90	63	140	552
1000	180	126	280	1104

Table 1. Size in MB of each file when using integer data.

5.2 Performance evaluation of TPC I/O

We have analyzed the I/O stage performance of the BIPS3D under different scenarios. More specifically, we have evaluated:

- Different input data meshes with diverse $loads$.
- Various compression levels.
- Different number of processors.

The performance of *TP I/O* and *TPC I/O* was compared for all these scenarios. For each one of them, the execution time of each stage (introduced in Section 3) was evaluated. Note that both techniques differs only in $st5$ and $st6$ stages, which respectively represent the 2% and 83% of the overall execution time[2].

Figures 3 and 4 show the $st6$ execution time improvement for *TPC I/O* employing integers and floats. The *execution time improvement* is defined as the reduction the execution time (*TP I/O -TPC I/O*) as percentage of the *TP I/O* time. A positive value indicates that compression reduces the execution time of this stage. In contrast, a negative value implies an increase of the $st6$ execution time when compression is used. In general, for the same dataset, an increase in the number of processors reduces the compression efficiency. However, this change is different for each input data. For example, good improvements are obtained for meshes 3 and 4 in all the cases. In contrast, the performance severely degrades for meshes 1 and 2 when the processor number increases. The reasons of these behaviours are analysed in the next section. Note also that different percentages of improvement are obtained for floats and integers. This is due to the fact that an integer is compressed more efficently than a float. For example, compressing integer values for mesh 1 ($load$=200, 4 processes) obtains a 49% of space reduction and takes 39.0 msecs (42.0 msecs for decompressing). In the case of compressing float values, obtains a 23% of space reduction and takes 67.7 msecs (78.6 msecs for decompressing).

[2]This percentage corresponds for mesh 4 with $load = 200$ using 8 processors. Note that the cost of st5 stage significatively increases with the number of processors.

Figures 6 and 7 show, respectively, the impact of the data volume in the integer and float execution time improvement for $st6$. We can see that in general, the higher $load$ parameter (number of elements associated to each node) the better efficiency of the *TPC I/O*. Performance improvement highly depends on the mesh characteristics. Besides, the *TPC I/O* efficiency is better for intergers than floats.

The impact of the compression level is showed in Figure 8. We have synthetically created new data meshes filled with variable percentages of zero elements. In practice, zero values appear in the mesh boundary (where no electrical fields are applied). We have covered a complete range of zero values starting from 0% (original mesh) until 100% (full zero mesh). With this test we evaluate the efficiency of the compression technique under different data contents. Note that raising the percentage of zero elements dramatically increases the LZO performance. This behavior appears for all the considered scenarios.

integer data, $load$ 1000 and using 32 processes (74%). And the maximum improvement for 100% of zero values is obtained with mesh 3, float data, $load$ 500 and using 32 processes (98,8%).

Regarding $st5$ stage, as Figure 1 shows, when the number of processors is reduced, the cost of this stage is small (compared with $st6$ and $st7$). Thus, it has a minor impact in the *TP I/O* performance. However, when the number of processors incresases, this stage becomes more significant in terms of overall execution time. Figure 5 shows the execution time improvement for *TPC I/O* of $st5$ using integers. Note that in general good improvements are obtained using compression for this stage.

Finally, note that, in this study we are not considering the disk transfer time ($st7$ stage) given that it is the same for both techniques. This time represents the 27% of the total Two-Phase I/O execution time[3].

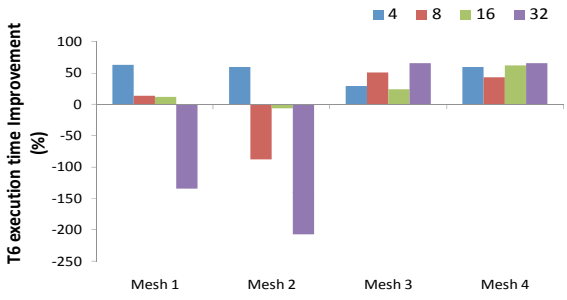


Figure 3. Percentage of improvement of compression technique for integers in stage st6 with $load$ 200.

[3]This percentage corresponds for a mesh 4 using 8 processors.

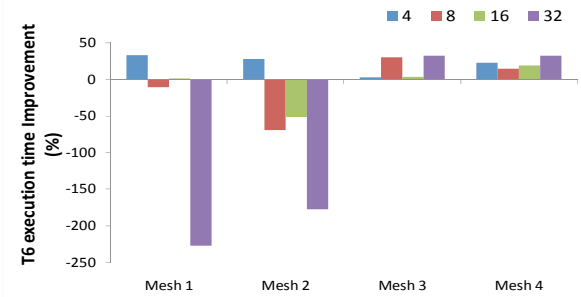


Figure 4. Percentage of improvement of compression technique for float in stage st6 with $load$ 200.

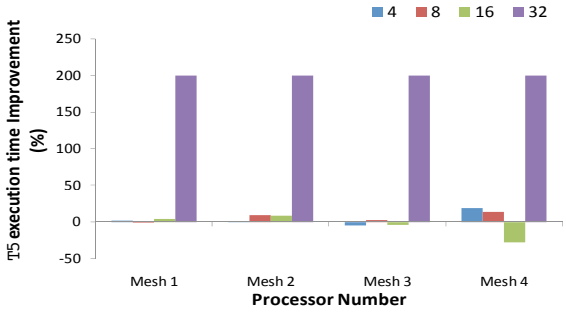


Figure 5. Percentage of improvement of compression technique for integers with $load$ 200.

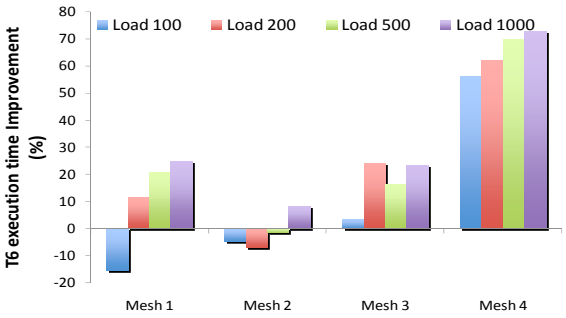


Figure 6. Impact of the data volume in the integer compression efficiency for an 16 processor execution in stage st6.

5.3 Performance analysis of TPC I/O

In order to identify the reasons for different performance improvements of *TPC I/O* technique, a more detailed study was performed. More specifically, we have focused on the $st6$ stage (given that it represents a significant fraction of the *TPC I/O* execution time), and we have evaluated the relationship between mesh characteristics and *TPC I/O* per-

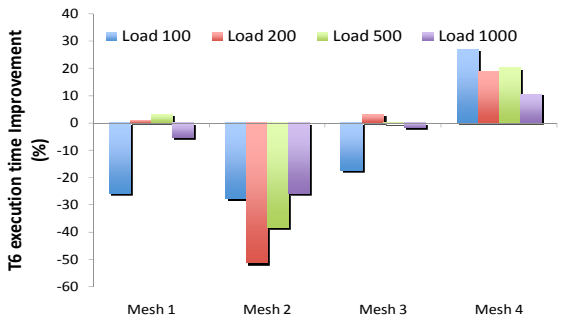


Figure 7. Impact of the data volume in the float compression efficiency for an 16 processor execution in stage st6.

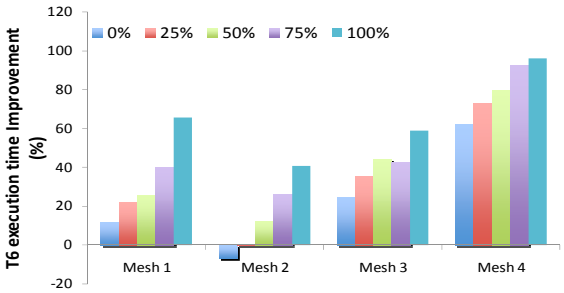


Figure 8. Impact of compression level in the integer compression efficiency for an 16 processor execution with $load$ 200 in stage st6.

formance. All the measures shown in this section were obtained for 0% of zero values and integers. Similar results can be derived for the rest of scenarios.

Tables 2 and 3 show, respectively, the total amount of exchanged messages and exchanged data for the $st6$ stage and mesh 1. The first one is the overall number of communications of $st6$ stage. The second one is the overall volume of exchanged data of this stage. These values were obtained instrumenting the TP I/O algorithm. In this table we can observe two behaviours:

1. When the $load$ increases the number of communications dimishes. This is due to the fact that larger $loads$ produce larger chunks of data associated to each mesh node. Given that the exchange buffer size is limited to 4MB, when the node size increases (bigger $loads$) the buffer is filled with less different nodes. Note that the number of communications is related to different node owners in the same exchange buffer. Consequently, increasing the $load$ tends to reduce the number of communications (Table 2) and to increase the ammount of exchanged data (Table 3). In terms of performance, this behaviour implies that when the $load$ incresases, bigger data structures are compressed, thus, the LZO performance increases. Table 4 shows the efficiency of LZO algorithm for different communications using just one exchange buffer[4]. We can see that as more processes share the buffer, the compressing/decompressing time increases.

2. When the number of processes increases, the nodes of the exchange buffer are distributed among more processes. On one hand, this causes data to be gathered from more processes, producing more communications (Table 2). On the other hand, the ammount of exchanged data also increases (Table 3). However, the amount of data of each communication decreases, producing worse LZO performance. For example, for mesh 1 and 4 processes, the average message length is: $7,199,600/10 = 719960$ whereas for 32 processess is: $9,274,400/271 = 34,223$.

Proc/Load	100	200	500	1000
4	12	10	9	8
8	38	33	26	22
16	114	105	80	64
32	408	271	240	181

Table 2. Number of different exchanged messages for mesh 1.

Proc/Load	100	200	500	1000
4	3,599,800	7,199,600	17,999,000	35,998,000
8	4,456,100	8,912,200	22,280,500	44,561,000
16	4,586,600	9,173,200	22,933,000	45,866,000
32	4,637,200	9,274,400	23,186,000	46,372,000

Table 3. Number of exchanged messages for mesh 1.

In Figures 5 and 6 the effects of the first behaviour can be observed: when the $load$ increases, the execution time improvement also tends, in general, to increase. Figures 3 and 4 reflect the second behaviour: the execution time improvement decreases with the number of processors.

Tables 5 and 6 evaluates the mesh characteristics influence on the algorithm performance. These tables show, respectively, the total amount of exchanged messages and data for $load = 200$. Each mesh has a specific data distribution pattern. However, mesh 3 and 4 produce lesser number of

[4]In this case the buffer contents are scattered among all the processors, therefore the number of communications is the total number of processor minus one.

Time(msec.)/Number of Comm.	3	7	15	31
Compress	44,1	51,1	55,3	58,5
Decompess	28,7	32	32,1	32,6
Total	72,8	83,1	87,4	91,1

Table 4. Compression and decompression time for one exchange buffer and different number of communications in msec.

Proc	$Mesh1$	$Mesh2$	$Mesh3$	$Mesh4$
4	10	10	8	7
8	33	32	22	20
16	105	112	56	49
32	271	387	159	130

Table 5. Number of different exchanged messages for all meshes with $load$ 200.

Proc	$Mesh1$	$Mesh2$	$Mesh3$	$Mesh4$
4	7,199,600	5,562,600	12,894,400	44,184,800
8	8,912,200	6,181,200	14,447,400	47,644,800
16	9,173,200	6,402,400	14,425,600	51,706,600
32	9,274,400	6,468,400	14,578,400	53,831,000

Table 6. Number of data exchanged for all meshes with $load$ 200.

communications than mesh 1 and 2 (Table 5). Given that the exchanged ammount of data is mostly the same for each mesh (Table 6), the average communication size is bigger for meshes 3 and 4 than for meshes 1 and 2. This fact produces a better TPC I/O performance for meshes 3 and 4 for 32 processes (see Figures 3 and 4).

6 Conclusions

In this paper a new technique called *Two-Phase Compressed I/O* based on the compression and decompression of communication is presented. In the evaluation section we have shown that the performance of *Two-Phase Compressed I/O* is, in many scenarios, better than the performance of the original *Two-Phase I/O technique*. However, this performance on the following factors:

- Size of message. In general, the LZO algorithm is efficient for large message sizes. For example, when large data structures are employed.
- Number of processors used in collective communications. For large numbers, the *TPC I/O* performance decreases.
- The mesh distribution. Both previous topics are strongly related with the way in which the mesh is distributed. In fact, we obtained different perfomance behaviours for each mesh.

As future work we plan to design a heuristic for deciding to use (or not) compression. This heuristic is based on network features, data characteristics, compression level, and number of processes used in the collective I/O operation.

Acknowledgements

This work has been partially funded by project TIN2007-63092 of Spanish Ministry of Education and project CCG07-UC3M/TIC-3277 of Madrid State Governement.

References

[1] R. Bordawekar. Implementation of Collective I/O in the Intel Paragon Parallel File System: Initial Experiences. In *Proc. 11th International Conference on Supercomputing*, July 1997.

[2] J. del Rosario, R. Bordawekar, and A. Choudhary. Improved parallel I/O via a two-phase run-time access strategy. In *Proc. of IPPS Workshop on Input/Output in Parallel Computer Systems*, 1993.

[3] G. Karypis and V. Kumar. METIS — A software package for partitioning unstructured graphs, partitioning meshes, and computing fill-reducing orderings of sparse matrices. Technical report, Department of Computer Science/Army HPC Research Center, University of Minnesota, Minneapolis, 1998.

[4] D. Kotz. Disk-directed I/O for MIMD Multiprocesses. In *Proc. of the First USENIX Symp. on Operating Systems Design and Implementation*, 1994.

[5] A. Loureiro, J. González, and T.F.Pena. A parallel 3d semiconductor device simulator for gradual heterojunction bipolar transistors. *Journal of Numerical Modelling: electronic networks, devices and fields*, 16:53–66, 2003.

[6] R. Thakur, W. Gropp, and E. Lusk. Data Sieving and Collective I/O in ROMIO. In *Proc. of the 7th Symposium on the Frontiers of Massively Parallel Computation*, pages 182–189, February 1999.

[7] W. Ligon and R. Ross. An Overview of the Parallel Virtual File System. In *Proceedings of the Extreme Linux Workshop*, June 1999.

[8] C. F. S. Inc. Lustre: A scalable, high-performance file system. Cluster File Systems Inc. white paper, version 1.0, November 2002. http://www.lustre.org/docs/whitepaper.pdf.

[9] F. Isaila, G. Malpohl, V. Olaru, G. Szeder, and W. Tichy. Integrating Collective I/O and Cooperative Caching into the "Clusterfile" Parallel File System. In *Proceedings of ACM International Conference on Supercomputing (ICS)*, pages 315–324. ACM Press, 2004.

[10] F. Schmuck and R. Haskin. GPFS: A Shared-Disk File System for Large Computing Clusters. In *Proceedings of FAST*, 2002.

[11] R. Thakur, W. Gropp, and E. Lusk. Optimizing Noncontiguous Accesses in MPI-IO. *Parallel Computing*, 28(1):83–105, Jan. 2002.

[12] W. keng Liao, K. Coloma, A. Choudhary, L. Ward, E. Russel, and S. Tideman. Collective Caching: Application-Aware Client-Side File Caching. In *Proceedings of the 14th International Symposium on High Performance Distributed Computing (HPDC)*, July 2005.

[13] W. keng Liao, K. Coloma, A. N. Choudhary, and L. Ward. Cooperative Write-Behind Data Buffering for MPI I/O In *PVM/MPI*, pages 102–109, 2005.

[14] N. Nieuwejaar, D. Kotz, A. Purakayastha, C. Ellis, and M. Best. File Access Characteristics of Parallel Scientific Workloads. In *IEEE Transactions on Parallel and Distributed Systems, 7(10)*, Oct. 1996.

[15] H. Simitici and D. Reed. A Comparison of Logical and Physical Parallel I/O Patterns. In *International Journal of High Performance Computing Applications, special issue (I/O in Parallel Applications), 12(3)*, 1998.

[16] K. Seamons, Y. Chen, P. Jones, J. Jozwiak, and M. Winslett. Server-directed collective I/O in Panda. In *Proceedings of Supercomputing '95*.

[17] W. Yu and J. Vetter and R. S. Canon and S. Jiang. Exploiting Lustre File Joining for Effective Collective I/O. In *CCGRID '07: Proceedings of the Seventh IEEE International Symposium on Cluster Computing and the Grid*, pages 267–274, Washington, DC, USA, 2007. IEEE Computer Society.

[18] LZO algirthm internal structure http://wildsau.idv.uni-linz.ac.at/mfx/lzo.html

[19] Rosa Filgueira, David E. Singh, Juan Carlos Pichel Florin Isaila, and Jesus Carretero. Data Locality Aware Strategy for Two-Phase Collective I/O. In *Int. Meeting High Performance Computing for Computational Science (VECPAR). Toulouse, France. June 2008.*.

[20] P. Corbett and D. Feitelson and Y. Hsu and J.-P. Prost and M. Snir and S. Fineberg and B. Nitzberg and B. Traversat and P. Wong. MPI-IO: A parallel file I/O interface for MPI. In *NAS-95-002, June 1995*.

[21] http://www.unix-systems.org/ The Portable Operating System Interface 1995.

[22] Jesús Carretero and Jaechun No and Sun-soon Park and Alok N. Choudhary and Pang Chen. COMPASSION: A Parallel I/O Runtime System Including Chunking and Compression for Irregular Applications. HPCN Europe 1998: Proceedings of the International Conference and Exhibition on High-Performance Computing and Networking. 1998.