Data Locality Aware Strategy for Two-Phase Collective I/O

Rosa Filgueira[⋆], David E.Singh, Juan C. Pichel, Florin Isaila, and Jesús Carretero

Universidad Carlos III de Madrid
Department of Computer Science
{rosaf,desingh,jcpichel,florin,jcarrete}@arcos.inf.uc3m.es

Abstract. This paper presents Locality-Aware Two-Phase (LATP) I/O, an optimization of the Two-Phase collective I/O technique from ROMIO, the most popular MPI-IO implementation. 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 has as main purpose the reduction of the number of communication involved in the I/O collective operation and, therefore, the improvement of the global execution time. Compared with Two-Phase I/O, LATP I/O obtains important improvements in most 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 [15], PVFS [11] and Lustre [12] offer scalable solutions for concurrent and efficient access to storage. These parallel file systems are accessed by the parallel applications through interfaces such as *POSIX* or *MPI-IO*. This paper targets the optimization of the MPI-IO interface inside ROMIO, the most popular MPI-IO distribution.

Many parallel applications consist of alternating compute and I/O phases. 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 [19, 20]. 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* [7, 21]. If the merging occurs at intermediary nodes or at compute nodes the method is called *two-phase I/O* [3, 2].

In this work we focus on the *Two-Phase I/O* technique, extended by Thakur and Choudhary in *ROMIO*[10]. Based on it we have developed and evaluated the *Locality-Aware Two-Phase I/O* (*LATP I/O*) technique in which file data access is dependent on the specific data distribution of each process. The comparison with the original version of Two-Phase I/O shows that our technique obtains an important run time reduction .

[⋆] Candidate to the Best Student Paper Award

This is achieved by increasing the locality of the first phase and, therefore, reducing the number of communication operations.

This paper is structured as follows. Section 2 contains the related work. Section 3 explains in detail the internal structure of *Two-Phase I/O*. Section 4 contains the description of the *Locality-Aware Two-Phase I/O*. Section 5 is dedicated to performance evaluations. 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 a file (a common case for parallel scientific applications). In disk-directed I/O [7], the compute nodes fordward file requests to the I/O nodes. The I/O nodes merge and sort the requests before sending them to disk. In server-directed I/O of Panda [21], the I/O nodes sort the requests on file offsets instead of disk addresses. *Two-Phase I/O* [3, 2] 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. We present this implementation of theses technique in ROMIO in the next section. Using Lustre file joining (merging two files into one) for improving collective I/O is presented in [22].

Several researchers have contributed with optimizations of MPI-IO data operations: data sieving [10], non-contiguous access [16], collective caching [17], cooperating write-behind buffering [18], integrated collective I/O and cooperative caching [14].

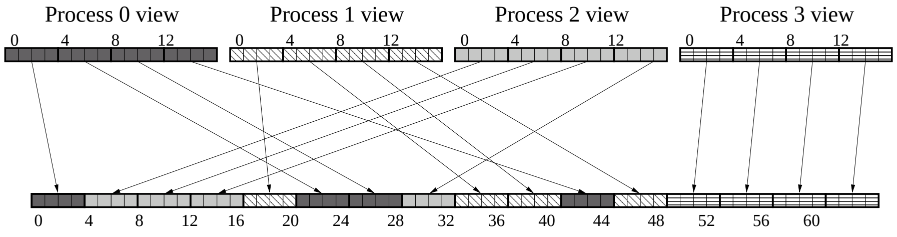
3 Internal structure of Two-Phase I/O

As its name suggests, *Two-Phase collective I/O* consists of two phases: a shuffle phase and an I/O phase. In the shuffle phase, small file requests merged into larger ones. In the second phase, contiguous transfers are performed to or from the file system.

Before these two phases, *Two-Phase I/O* divides the file region between the minimum and maximum file offsets of accesses of in equal contiguous regions,called *File Domains* (FD) and assigns each FD to a configurable subset of compute nodes, called aggregators. Each aggregator is responsible for aggregating all the data inside its assigned FD and for transferring the FD to or from the file system.

In the implementation of *Two-Phase I/O* the assignment of FD to aggregators is fixed, independent of distribution of data over the compute nodes. In contrast, based the processor data distribution, LATP minimises the total volume of communications. By means of this strategy it is possible to reduce the communication and, therefore, the overall I/O time.

We ilustrate the *Two-Phase I/O* technique through an example of a vector of 16 elements that is written in parallel by 4 processes (see Figure 1) to a file. The size of one element is 4. Each process has previously declared a view on the file, i.e. non-contiguous regions are "seen" as if they were contiguous. For instance, process 0 "sees" the data at file offsets 0-3 and 20-23 contiguously, as view offsets 0-7.



File containing the vector

Fig. 1. File access example.

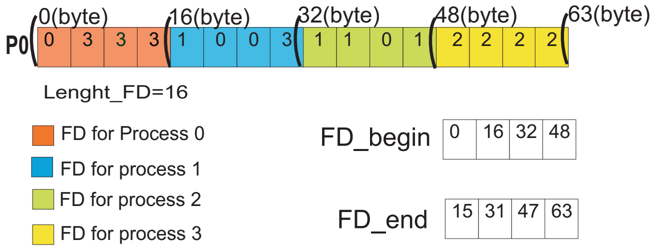


Fig. 2. Assignment of file domain

Before performing the two mentioned phases, each process analyzes, which parts of the file are stored locally by creating a list of offsets and list of lengths. According to the example, process 0 is assigned three intervals: (*offset*=0, *length*=4), (*offset*=20, *length*=8), (*offset*=40, *length*=4). The list of offsets for this process is: {0, 20, 40} and the list of lengths is: {4, 8, 4}.

In addition, each process calculates the first and last byte of the accessed intervl. In our example, the first byte that process 0 has stored is 0 and the last byte is 43. Next, all processes exchange these values and compute the maximum and minimum of file access range, in this case 0 and 63, respectively. The interval is then divided in to equal-sized FDs. If all 4 compute nodes are aggregators, it will be divided in 4 chunks of 16 bytes, one for each aggregator. Each chunk is assigned to each aggregator according to its rank value. That is, block 0 is assigned to process with rank 0, block 1 to rank 1, etc. Each chunk (FD) is written to file by the assigned process. For performing this operation, each process needs all the data of its FD. If these data are stored in other processes, they are exchanged during the communication phase.

Once the size of each FD is obtained, two lists with so many positions as number of processes are created. The first list indicates the beginning of the FD of each process. The second one indicates the final of the FD of each process. Figure 2 shows how the vector is divided into different FDs. Each FD has been assigned a different color. Also, it can be observed that the assignment of FD is independent of the local data of each process. This scheme is inefficient in many situations. For example, the FD for process 3, begins at byte 48 and finalizes at byte 63. All these data are stored in the process 2, so this implies unnecessary communications between process 2 and 3, because the process 2 has to send all this data to process 3, insted of writing it to disk.

Once each process knows all the referring data, it analyzes, which data from its FD is not locally stored and what communication has be to be established in order to gather the data. This stage is reflected in Figure 3. This figure shows the data of the P vector that each process has locally stored. For any process, the vector elements labeled 'R' are received and the ones labled 'S' are sent. The arrows represent communication operations.

For example, in Figure 3, process 0 needs three elements that are stored in the process 3, and has stored three elements that processes 1 and 2 need. In the following step of *Two-Phase I/O* technique, the processes exchange the previously calculated data. Once all the processes have the data of their FD, they write to file a chunk of consecutive entries as shown in Figure 4. Each process transfers only one contiguous region to file (its FD), thus, reducing the number of I/O requests and improving the overall I/O performance.

4 Locality aware strategy for Two-Phase I/O

As explained in Section 3, *Two-Phase I/O* makes a fixed assignment of the FDs to processes. With the *LA-Two-Phase I/O* replaces the rigid assignment of the FDs by an assignment dependent of the initial distribution of the data over the processes. Our approach is based on the Linear Assignment Problem.

4.1 Linear Assignment Problem

The *Linear Assignment Problem* (LAP) is a well studied problem in linear programming and combinatorial optimization. LAP computes the optimal assigment of n items to n elements given an $n \times n$ cost matrix. In other words, LAP selects n elements of the matrix (for instance, the matrix from Table 1), so that there is exactly one element in each row and one in each column, and the sum of the corresponding costs is maximum.

The problem of finding the best interval assignment to the existing processes can be efficiently solved by applying the existing solutions of this problem. In our case, the LAP tries to assign FDs to processes, by maximizing the cost, given that we want to assign to the process the interval, for which it has more local data.

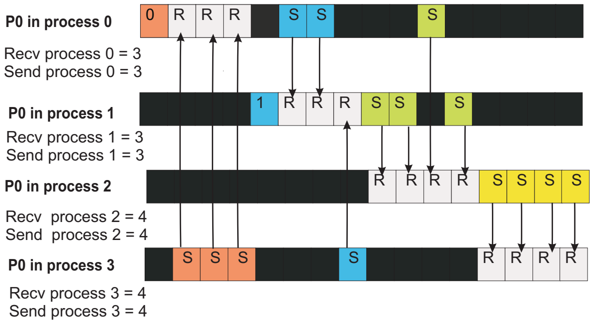


Fig. 3. Data transfers between processes

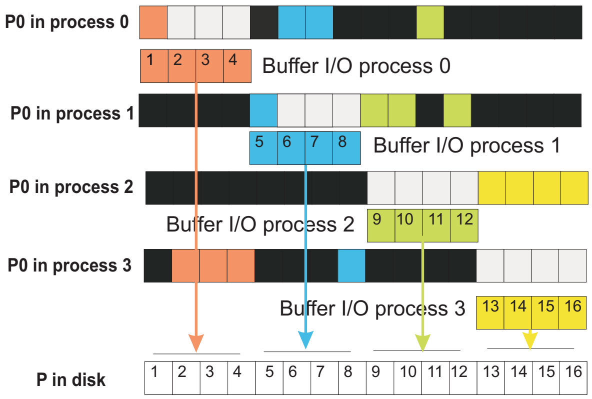


Fig. 4. Write of data in disk.

A large number of algorithms, sequential and parallel, have been developed for LAP. We have selected for our work the following algorithms, considered to be the most representative ones:

- *Hungarian algorithm* [1]: This is the first polynomial-time primal-dual algorithm that solves the assignment problem. The first version was invented and published by Harold Kuhn in 1955 and has a $O(n^4)$ complexity. This was revised by James Munkres in 1957, and has been known since as the Hungarian algorithm, the Munkres assignment algorithm, or the Kuhn-Munkres algorithm.
- *Jonker and Volgenant algorithm*[5]: They develop a shortest augmenting path algorithm for the linear assignment problem. It contains new initialization routines and a special implementation of Dijkstra's shortest path method. For both dense and sparse problems computational experiments they show this algorithm to be uniformly faster than the best algorithms from the literature. It has a $O(n^3)$ complexity.
- *APC and APS Algorithms*[4]: These codes implement the Lawler $O(n^3)$) version of the Hungarian algorithm by Carpenato, Martello and Toth. APC works on a complete cost matrix, while APS works on a sparse one.

4.2 LA-Two-Phase I/O

In order to explain *LA-Two-Phase I/O*, we use the example for Section 2 with the same data and distribution as in Figure 1.

The proposed technique differs from the original version in the assignment of the FDs to processes. Each FD is assigned based on the distribution of the local data of processes. In order to compute this initial distribution, the number of intervals in which we can divide the file is computed. This is made by dividing the size of the access interval by the sizes of the FD. In our example, the number of intervals is equal to four.

The next step consists in assigning each interval to each process efficiently. First, a matrix is constructed, with as many rows as processes, and so many columns as intervals. Each matrix entry contains the number of elements that each process has stored. The matrix from our example is shown in Table 1.

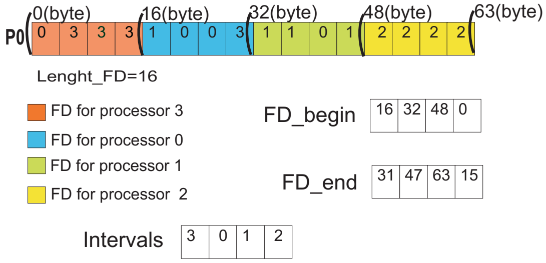


Fig. 5. Assignment of file domain for LA-Two-Phase I/O.

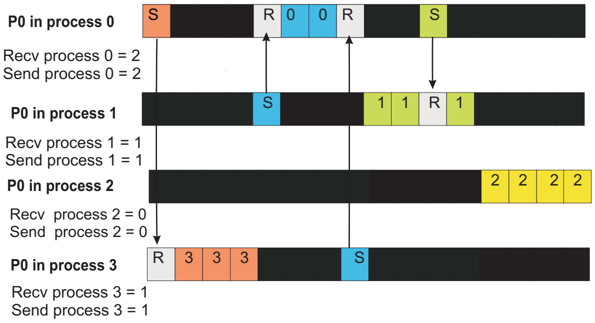


Fig. 6. Transfers of data between processes in LA-Two-Phase I/O.

Our technique is based on maximizing the aggregator locality by applying a LAP algorithm and obtaining a list with the assignment of intervals to processes. For our example, the assignment list is {3, 0, 1, 2} as indicated Figure 5. This list represents the interval that has been assigned to each process.

This strategy reduces the number of communication operations, due to the fact that each process increases the amount of locally assigned data. The following phases of the *LA-Two-Phase I/O* are the same as those of the original version. Figure 6 shows the communication operations between processes. In the corresponding step of original *Two-phase I/O*, shown in Figure 3, process 2 sends four elements and receives four elements. With our technique, the number of transfers of process 2 has been reduced to none (see Figure 6). In this example the LA-Two-Phase I/O reduces the overall transfers from 28 exchanged elements to 8.

Interval/Process	0	1	2	3
0	1	2	1	0
1	0	1	3	0
2	0	0	0	4
3	3	1	0	0

Table 1. Number of elements of each interval.

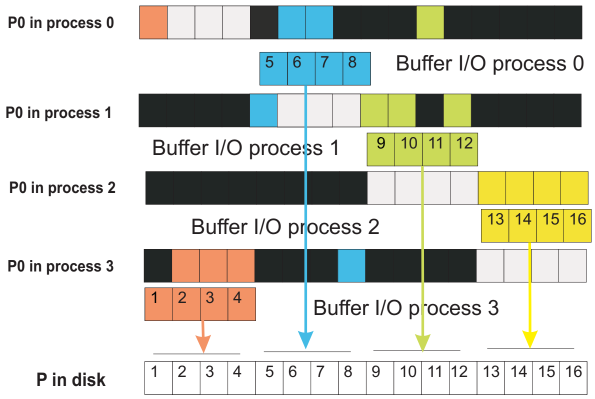


Fig. 7. Write of data in disk in LA-Two-Phase I/O.

Figure 7 shows the I/O phase of *LA-Two-Phase I/O*. In the same way that as in the original technique, each process writes to file a contiguous data region.

5 Performance Evaluation

The evaluations in this paper were performed by using the BIPS3D application with different input meshes related to different semiconductor devices. We have compared LATP I/O with the original version of the technique *Two-Phase I/O* implemented in MPICH.

The tests have been made in Magerit cluster, installed in the CESVIMA supercomputing center. Magerit has 1200eServer BladeCenter JS202400 nodes, and each node has two processors IBM 64 bits PowerPC single-core 970FX running at 2.2 GHz and having 4GB RAM and 40GB HD. The interconnection network is Myrinet.

We have used the MPICHGM 2.7.15NOGM distribution for the basic implementation of *Two-Phase I/O*. We have developed our technique by modifying this code. 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.

The remainder of this section is divided as follows. Subsection 5.1 briefly overviews the BIPS3D application. Subsection 5.2 contains the evaluation of the linear assignment technique. Finally, the evaluation of *LA-Two-Phase I/O* is presented in subsection 5.3.

5.1 BIPS3D Simulator

BIPS3D is a 3-dimensional simulator of BJT and HBT bipolar devices [8]. 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. In this work, we have used four different meshes, as described later.

Mesh/Load	$mesh1$	$mesh2$	$mesh3$	$mesh4$
100	18	12	28	110
200	36	25	56	221
500	90	63	140	552

Table 2. Size in MB of each file based on the mesh and loads.

Using the METIS library, this mesh is divided into sub-domains, in such a manner that one sub-domain corresponds to one process. The next step is decoupling the Poisson equation from the hole and electron continuity equations. They are linearized by Newton method. Then we construct, for each sub-domain, in a parallel manner, the part corresponding to the associated linear system. Each system is solved using domain decomposition methods. Finally, the results are written to a file. For our evaluation BIPS3D has been executed using four different meshes: $mesh1$ (47200 nodes), $mesh2$ (32888 nodes), $mesh3$ (732563 nodes) and $mesh4$ (289648 nodes), with different number of processes: 8, 16, 32 and 64. The BIPS3D associates a data stucture 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 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 this work we have evaluated different loads, concretely, 100, 200 and 500. Table 2 lists the different sizes (in MB) of each file based on load and mesh characteristics.

5.2 Performance of the Linear Assignment Problem

We have applied all the LAP algorithms described in Section 4 to our problem. We have noticed that in all cases all algorithms produce the same assignment (of FDs to processes). The only difference between them is the time to compute the optimal allocation. Figure 8 shows the normalized execution time (taking the APC algorithm as the reference technique) for solving the interval distribution using different number of processes and $mesh1$ data distribution. Note that the fastest algorithm is the Jonker and Volgenant, and for this reason we have chosen it to apply in *LA-Two-Phase I/O*.

5.3 Performance evaluation of LA-Two-Phase I/O

Figure 9 shows the percentage of reduction of communications for *LA-Two-Phase I/O* over *Two-Phase I/O* for $mesh1$, $mesh2$, $mesh3$ and $mesh4$ and different numbers of processes. We can see that, when LATP is applied, the volume of transferred data is considerably reduced.

In the first step of our study we have analyzed the *Two-Phase I/O*, identifying the stages of the technique that are more time-consuming. The stages of *Two-Phase I/O* are:

- *Offsets and lengths calculation (st1)*: In this stage the list of offsets and lengths of the file is calculated.

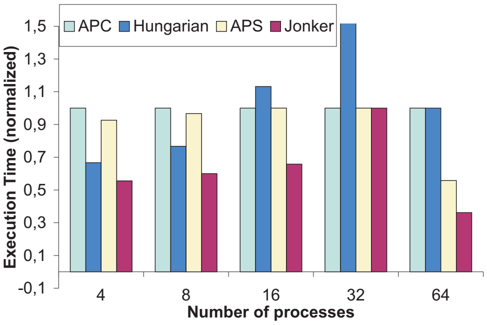


Fig. 8. Time for computing the optimal allocation for $mesh1$.

- *Offsets and lengths communication (st2)*: Each process communicates its start and end offsets to the other processes. In this way all processes have global information about the involved file interval.
- *Interval assignment (st3)*: This stage only exists in *LA-Two-Phase I/O*. First, it calculates the number of intervals into which we can divide the file, and then, it assigns intervals to processes by applying *Linear Assignment Problem* (see Table 1).
- *File domain calculation (st4)*: The I/O workload is divided among processes (see Figure 2). 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 (st5)*: It calculates the access requests for the file domains of remote processes.
- *Metadata transfer (st6)*: Transfer the lists of offsets and lengths.
- *Buffer writing (st7)*: The data are sent to appropriate processes (see Figure 3).
- *File writing (st8)*: The data are written to file (see Figure 4).

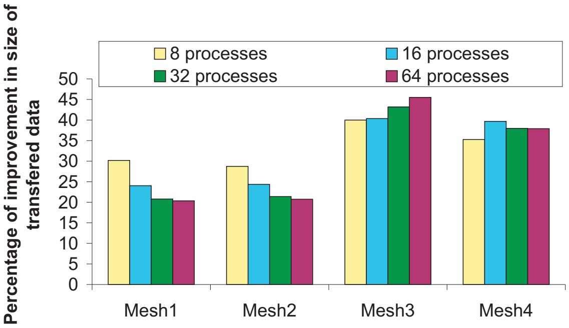
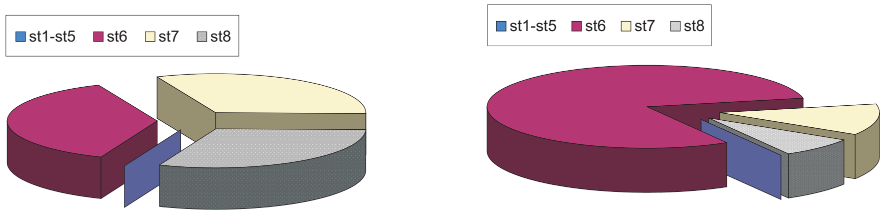


Fig. 9. Percentage reduction of transferred data volume for $mesh1$, $mesh2$, $mesh3$ and $mesh4$.



(a) (b)

Fig. 10. Stages of Two-Phase I/O for $mesh1$: (a) with load 100 and 16 processes and (b) with load 100 and 64 processes.

The buffer and file writing stages (st7 and st8), are repeated as many times as the following calculus indicates: the size of the file portion of each process is divided by the size of the *Two-Phase I/O* buffer (4 MB in our experiments). First, the write size of each process is obtained by dividing the size of the file by the number of processes. For example, for $mesh4$ with load 500 and using 8 processes the size of the file is 552 MB (see table 2). Therefore, the write size of each process is 69 MB. Then, the file size related to each process is divided to the buffer size of *Two-Phase I/O*. Consequently, the number of times is given by this value divided by 4MB, for this example is 18.

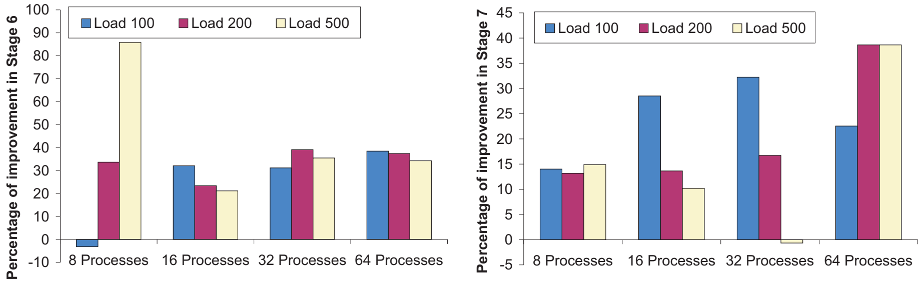
Figure 10 represents the percentage of time of *Two-Phase I/O* for $mesh1$ with load 100, with 16 and 64 processes, respectively. The costs of stages st1, st2, st3, st4 and st5 have been added up, and we have represented this value in the figures as st1-st5.

As we can see in the Figures 10(a) and 10(b), the slowest stages are st6 and st7. Note that the cost of the st6 stage increases with the number of processes. These figures show the weight of the communication stages in the *Two-Phase I/O* technique. Moreover, we can see that the cost of these stages increases with the number of processors. Based on this, we conclude that this represents an important bottleneck in this technique. For this reason we have developed the *LA-Two-Phase I/O* technique with the aim of reducing the amount of communication. This technique reduces the global time of *Two-Phase I/O*.

LA-Two-Phase I/O technique improves the communication performance of st6 and st7 stages. Figure 11(a) shows the percentage of improvement in st6 stage for $mesh1$, for different loads and different number of processes.

In this figure we can see that the time of st6 stage is significantly reduced in most cases. In this stage each process calculates what requests of other processes lie in its file domain and creates a list of offset and lengths for each process, which has data stored in its FD. Besides, in this stage, each process sends the offset and length lists to the rest of the processes. In *LA-Two-Phase I/O*, many of the data that each process has stored belong to its FD (given that data locality is increased) and therefore less offsets and lengths are communicated.

Figure 11(b) depicts the time of stage st7. Note that, again, this time is reduced in most of cases. This is because in this stage, each process sends the data that has

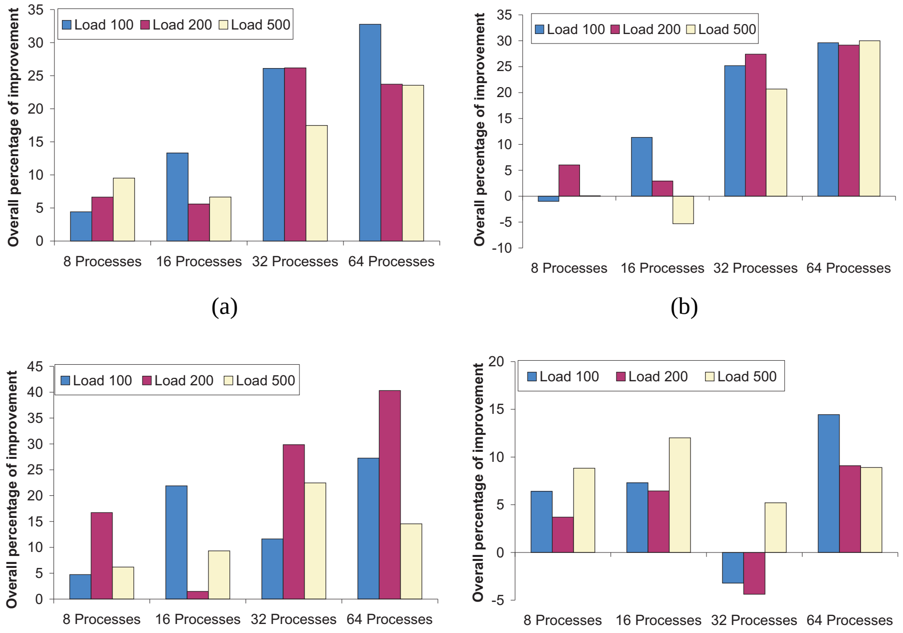


(a) (b)

Fig. 11. Percentage of improvement for $mesh1$: (a) in Stage 6 and (b) in Stage 7.

calculated in st6 stage to the appropriate processes. In *LA-Two-Phase I/O*, many of the data that each process has stored belong to its FD, therefore, they send less data to the other processes, reducing the number of transfers and the volume of data.

Figure 12 shows the overall percentage of improvement of our technique for $mesh1$, $mesh2$, $mesh3$ and $mesh4$ with 64 processes. In this figure, we included the time of all stages. For this reason the percentage of improvement is smaller than in previous stages.



(a) (b)

(c) (d)

Fig. 12. Overall improvement for: (a) $mesh1$ (b) $mesh2$ (c) $mesh3$ and (d) $mesh4$.

Nevertheless, we can notice that in the majority cases a significant improvement in the execution time for *LA-Two-Phase I/O* technique. The original technique performed better in 4 of the 48 cases, but the loss was under 5% in all the cases. It appears that, for these cases (which represent less than 10% of the total), the data locality happened to be good enough in the original distribution and the additional cost to find a better distribution did not pay off.

It is important to emphasize that the additional cost of the new stage (st3) is very small compared with the total time. The fraction of this stage in the overall execution time is small: in the best case it is 0.07% of the time ($mesh2$, 8 processes and load 500) and in the worst case the 7% ($mesh3$, 64 processes and load 100).

6 Conclusions

In this paper a new technique called *LA-Two-Phase I/O* based on the local data that each process stores is presented. First of all, we have showed that the *LA-Two-Phase I/O* improves the overall performance, when compared to the original *Two-Phase I/O*. The new stage (st3), which we have added to the technique *LA-Two-Phase I/O* has an insignificant overhead in comparison to the total execution time.

In the evaluation section we have shown that the greater number of processes, the larger the improvement brought by our technique. Finally, it is important to emphasize, that *LA-Two-Phase I/O* can be applied to every kind of data distribution.

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. S. S. Blackman. Multiple-Target Tracking with Radar Applications. In *Dedham,MA: Artech House*, 1986.
2. 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. To appear.
3. 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.
4. S. M. Giorgio Carpaneto and P. Toth. Algorithms and codes for the assignment problem. *Annals of Operations Research*, 13(1):191–223, 1988.
5. R. Jonker and A. Volgenant. A Shortest Augmenting Path Algorithm for Dense and Sparse Linear Assignment Problems. *Computing*, 38(4):325–340, 1987.
6. 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.

7. D. Kotz. Disk-directed I/O for MIMD Multiprocesses. In *Proc. of the First USENIX Symp. on Operating Systems Design and Implementation*, 1994.
8. 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.
9. K. Seamons, Y. Chen, P. Jones, J. Jozwiak, and M. Winslett. Server-directed collective I/O in Panda. In *Proceedings of Supercomputing '95*.
10. 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.
11. W. Ligon and R. Ross. An Overview of the Parallel Virtual File System. In *Proceedings of the Extreme Linux Workshop*, June 1999.
12. 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.
13. Indiana University, http://www.lam-mpi.org/. *LAM website*.
14. 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.
15. F. Schmuck and R. Haskin. GPFS: A Shared-Disk File System for Large Computing Clusters. In *Proceedings of FAST*, 2002.
16. R. Thakur, W. Gropp, and E. Lusk. Optimizing Noncontiguous Accesses in MPI-IO. *Parallel Computing*, 28(1):83–105, Jan. 2002.
17. 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.
18. 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.
19. 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.
20. 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.
21. K. Seamons, Y. Chen, P. Jones, J. Jozwiak, and M. Winslett. Server-directed collective I/O in Panda. In *Proceedings of Supercomputing '95*.
22. 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.