A collective I/O implementation based on Inspector-Executor paradigm [*]

David E. Singh, Florin Isaila, Juan Carlos Pichel and Jesús Carretero
Computer Science Department
Universidad Carlos III de Madrid
Spain

Abstract

In this paper we present a multiple phase I/O collective operation for generic block cyclic distributions. The communication pattern is automatically generated by an inspector phase and the communication and file access phase are performed by an executor phase. The inspector phase can be amortized over several accesses. We show that our method outperforms other techniques used for parallel I/O optimizations for small access granularities.

1 Introduction

Nowadays, one important bottleneck in parallel architectures is the I/O subsystem. Usually I/O is considerably slower than CPU computations. In parallel systems, when the number of processors increases, the I/O bottleneck becomes more and more important. Some parallel file systems try to solve this problem by exploiting parallelism at file-level. However, there is a need of efficient and scalable parallel I/O techniques that exploit both hardware and O.S. infrastructures. In this work a new parallel I/O technique for distributed systems is presented.

A major contribution of this paper is the development of a novel I/O technique for distributed systems called Inspector-Executor Collective I/O (IEC I/O). Our method takes advantage of fast communication networks for exchanging the data, so that we can improve the I/O locality and reduce the cost of the global I/O operation. Additionally, we present an experimental evaluation of the effectiveness of our method for a broad range of input data and architectural configurations. Experimental results prove that our method obtains the best performance in a broad number of scenarios, when compared with other state-of-the-art techniques.

[*]This work was supported by the Madrid State Government, under project CP06 Técnicas de optimización de la E/S en aplicaciones para entornos de computación de altas prestaciones.

This paper is structured as follows. The next section contains a description of the existing I/O techniques for distributed systems. Section 3 describes the IEC I/O method. Section 4 shows the experimental comparative study. Finally, Section 5 presents the conclusions.

2 I/O techniques for distributed systems

It has been shown [6] that the processes of a parallel application frequently access a common data set by issuing a large number of small non-contiguous I/O requests. 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* [3, 7]. If the merging occurs at intermediary nodes or at compute nodes the method is called *two-phase I/O* [2, 1]. Two-phase I/O is in ROMIO [10], an implementation of MPI-IO interface.

Another parallel I/O optimization technique is List I/O [11]. In List I/O, the non-contiguous accesses are specified through a list of offsets of contiguous memory or file regions and a list of lengths of contiguous regions. MPI-IO [5] is a standard interface for MPI-based parallel I/O. MPI data types are used by MPI-IO for declaring views and for performing non-contiguous accesses. A *view* is a contiguous window to potentially non-contiguous regions of a file. After declaring a view on a file, a process may see and access non-contiguous regions of the file in a contiguous manner.

In an earlier work [8], we have presented the optimization of the I/O stage of STEM-II scientific application. We implemented a collective I/O technique, targeting the particular data distribution of STEM-II. The work presented in the current paper generalizes the work from the previous one, by including an inspector phase, which automatically generates the data types used for selecting the data to be exchanged, remote data placement and data transfer to disk. This is a complete new design that can han-

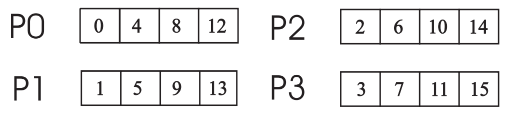

Figure 1. Data distribution for $N_x = 16$, $B_x = 1$ and $N_p = 4$.

dle generic block-cyclic distribution and does not require a specific predefined memory layout. Additionally, unlike in two-phase I/O, the inspector phase is decoupled from the executor phase (data exchange and disk transfer), which allows the amortization of the inspector phase over several execution with a similar pattern.

3 Inspector-Executor Collective I/O Algorithm

In this section we present our algorithm, for the case of storing to disk a vector x with N_x entries distributed among N_p processes using a block-cyclic scheme. Each block consists of B_x entries and each process has N_b blocks assigned. Subsequently, we have that: $B_x * N_b = N_x$. Figure 1 shows the resulting distributed values for $N_x = 16$, $N_p = 4$, $B_x = 1$ and $N_B = 4$.

There are many real applications fitting this scenario. For instance, parallel simulations of discretized environments where a particular problem is discretized into volume elements distributed among a given number of processes. In these situations, block-cyclic distributions are commonly used, given that they achieve a good load-balance. Periodically, parts of these data are transferred to disk (for instance, during check-pointing operations). These data are subsequently read for visualizing and monitoring the simulated environment. Usually, the simulation and visualization programs are developed separately, being necessary a predefined data disk format for allowing the proper environment reconstruction. One standard format consists of storing the data in their original order.

Once data are distributed, the scenario that we are considering consists of storing the x vector on disk in the proper order. We understand under *proper order* storing all the x entries in their original order, that is $x = \{1, 2, \ldots 16\}$ for our example. Note that preserving the data structure avoids further off-line sorting operations.

The basic idea of IEC I/O method is increasing data locality of the I/O operations by means of data exchange among the computing nodes. Figure 2 shows the algorithm pseudocode. It is divided into four stages: memory allocation, datatype generation, data exchange and disk transfer.

Initially, we will describe the communication scheme and the memory layout for the particular distribution shown in Figure 1. Later, we will extend our technique to other kind of distributions.

```
   INSPECTOR: DATA MEMORY ALLOCATION
L1 $x = allocate(\frac{N_x}{N_p} + N_{ph} * \frac{N_x}{2*N_p})$
L2 $bin\_rank = integer2binary(rank)$
L2 $count = count\_ones(bin\_rank)$
L2 $alloc\_offset = count * N_x/(2 * N_p)$
L2 $receive(x, alloc\_offset)$

   INSPECTOR: DATATYPE GENERATION
L3 $Datatype\_send = generate\_send\_datatype()$
L4 $Datatype\_pack = generate\_pack\_datatype()$

   EXECUTOR: DATA EXCHANGING
   DO $ph = 0, N_{ph} - 1$
     IF $r\%(2^{ph+1}) < 2^{ph}$
L5        $r'_{ph} = r + 2^{ph}$
     ELSE
L5        $r'_{ph} = r - 2^{ph}$
     END IF
     $recv\_offset = compute\_offset(r'_1, r'_2, \ldots r'_{N_{ph}-1})$
L6   $Exchange(r'_{ph}, x, Datatype\_send, recv\_offset)$
   END DO

   EXECUTOR: DISK WRITTING
L7 $Pack(output\_buffer, x, Datatype\_pack)$
L8 $bin\_rank = integer2binary(rank)$
L8 $perm\_bin\_rank = permute(bin\_rank)$
L8 $offset = binary2integer(perm\_bin\_rank) * N_x/N_p$
L9 $Disk\_write(output\_buffer, file\_name, offset)$
```

Figure 2. Pseudo code of IEC I/O algorithm.

For a given architecture with N_p computing nodes, our method requires N_{ph} communication phases. With:

$$N_{ph} = \lceil log_2(N_p) \rceil \qquad (1)$$

Processes are grouped in couples, called *pairs*. During a given phase, each pair of processes exchanges a part of their data. We need to allocate an extra memory space for the exchanged data. In our method, both the communication pattern and amount of transferred data are predefined. More specifically, in each phase each processor sends and receives $N_x/(2 * N_p)$ data entries according to a fixed communication scheme. Subsequently, each process requires a total of Δx memory entries for storing all the communicated data:

$$\Delta x = N_{ph} * \frac{N_x}{2 * N_p} \qquad (2)$$

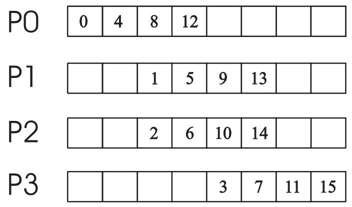

Figure 3. Data distribution of IEC I/O algorithm of x ($N_x = 16$ and $B_x = 1$) for four processes.

The distributed vector x and the incoming data are stored in the same memory region. That is, a region of $N_x/N_p + \Delta x$ array entries has to be allocated (Label L1 in Figure 2).

During the initial distribution of x we use an offset value to start storing entries. This offset is shown in lines labeled as L2 in Figure 2. Function $integer2binary$ converts an integer number (the process rank) into a binary number; $count\_ones$ returns the count of all the bits equal to one. This value is used to compute the offset used to store the distributed entries of x.

A graphical example of this distribution is shown in Figure 3. Given that $N_p = 4$ and $N_x = 16$, the memory overhead is 12 entries. The bin_rank, num_ones and $alloc\_offset$ values for each process are shown in Table 1.

During the communication phase, several entries of x are selected for being exchanged between couples of processes. Again, during the writing phase, different memory entries have to be gathered and written to disk. We use automatic generated datatypes (Figure 2, lines L3 and L4) for selecting these memory positions. In the case of regular distributions, datatype structures can be parameterized. That is, datatypes can be automatically computed by means of a set of linear equations with parameters N_p, N_x and B_x.

For non-regular distributions, datatype structures are automatically generated by means of an *inspector routine*. This routine analyzes both the data distribution and memory layout, and selects the memory entries corresponding to data exchanges (or disk writes).

$Rank$	0	1	2	3
bin_rank	00	01	10	11
num_ones	0	1	1	2
$alloc\_offset$	0	2	2	4

Table 1. Example of bin_rank, num_ones and $alloc\_offset$ values for $N_x = 16$, $N_p = 4$ and $B_x = 1$.

In this work we have decoupled the datatype generation algorithm (inspector routine) from the parallel I/O stage. By *parallel I/O stage* is meant the data exchanging and disk writing phases. This algorithm structure will allow increasing performance, given that when the same I/O operation is performed multiple times, the inspector routine is executed once and its results are reused multiple times by the parallel I/O stage. In [9] we present a detailed description of the inspector routine for datatype generation and examples of how some regular distributions can be efficiently parameterized.

Once memory space allocation is completed and datatypes are generated, the compute nodes perform data exchange. This operation consists of sending and receiving x entries between pairs of processes.

We denote the phase number ph, with $0 \le ph < N_{ph}$. Given a process with rank $r \in [0, N_p)$, the line L5 in Figure 2 determines the rank r'_{ph} of the target process used to exchange data during ph phase.

Table 2 shows the process pairs for a configuration with eight compute nodes. We also have printed in bold fonts the pairs for a four process configuration used in our example. Note that this scheme corresponds to a tree-based communication pattern.

Note that different datatypes are used for each communication phase. During the receive operation, all the received entries are stored in consecutive memory positions. The $compute\_offset$ function returns the offset value for storing the incoming data. This function is summarized in Figure 4. For each communication phase i, we check if the current destination rank (r'_{ph}) is greater than the destination rank of each communication phase (called r'_i). If it is true, we increase the offset in half of the assigned entries. In addition, when r'_{ph} is greater than the home process rank, the offset is increased by N_x/N_p. Table 3 summarizes the offset values for each process and communication phase.

Finally, Line L6 of Figure 2 shows the exchange function. This function sends $N_x/(2 * N_p)$ elements of x to process r'_{ph} and receives the same amount of elements from the same process. Datatypes are used for gathering the data to be sent, whereas the received entries are stored consec-

Ph	1st pair	2nd pair	3rd pair	4rd pair
0	**0-1**	**2-3**	4-5	6-7
1	**0-2**	**1-3**	4-6	5-7
2	0-4	1-5	2-6	3-7

Table 2. Process pairs for $N_p = 8$ ($N_{ph} = 3$). Each pair corresponds to the rank of the processes that exchange data.

```
$recv\_offset = 0$
DO  $i = 0, N_{ph} - 1$
    IF $r'_{ph} > r'_i$ THEN $recv\_offset += N_x/(2 * N_p)$
END DO
IF$r'_{ph} > rank$ THEN $recv\_offset += N_x/N_p$
return($recv\_offset$)
```

Figure 4. Pseudo code of $compute\_offset$ function.

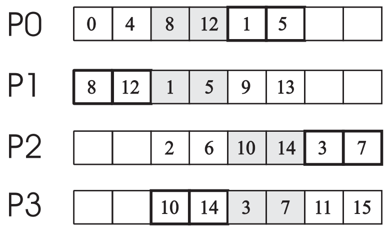

Figure 5. Data distribution of IEC I/O algorithm after phase 0 ($N_x = 16$, $B_x = 1$ and $N_p = 4$).

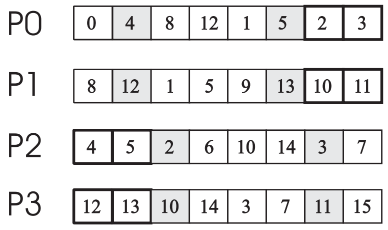

Figure 6. Data distribution of IEC I/O algorithm after phase 1 ($N_x = 16$, $B_x = 1$ and $N_p = 4$).

utively, starting at the offset value. This communication scheme is non-blocking. First, the process with lower rank sends the data to the higher rank processor. Then, the roles are swapped.

Figures 5 and 6 show an example of the sent/received values for each phase of the case of study. Sent entries are marked with a gray background and received values are marked with bolded borders.

$Rank$	0	1	2	3
$Recv\_offset$ ($ph = 0$)	4	0	6	2
$Recv\_offset$ ($ph = 1$)	6	6	0	0

Table 3. Example of $Recv\_offset$ values for each process and communication phase ($N_x = 16$, $N_p = 4$ and $B_x = 1$).

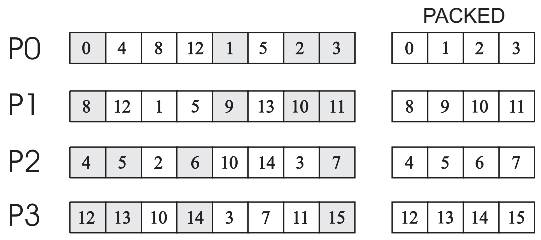

Figure 7. Final data distribution of IEC I/O algorithm after packing ($N_x = 16$, $B_x = 1$ and $N_p = 4$).

Once the communication phases are completed, I/O operations can be performed. Before that, the outgoing data have to be packed for increasing the network transfer performance. We use datatype `Datatype_pack` and `MPI_Pack` function for copying the desired values to the output buffer. Figure 7 shows the content of this buffer for each one of the processing nodes. Note that the output buffer contains a chunk of consecutive entries of x.

The last step of our method is the I/O operation. It is necessary to determine the file offset for each output buffer. This offset is computed in the lines labeled `L8` in Figure 2. The function $permute$ permutes each bit of the bin_rank sequence: for a sequence of n bits, the most significative bit ($n - 1$) is swapped with the less significative (0), the following one, $n - 2$, is swapped with bit 1 and so on; $binary2integer$ converts a binary number into integer. Table 4 summarizes the values for our example.

The $disk\_write$ function (label `L9`, Figure 2) writes the content of this buffer into the file $file\_name$ at $file\_offset$ words (in our application a word has 4 bytes). Note that this is a parallel I/O operation over non-overlapping file entries. Thus, there are not write conflicts.

$Rank$	0	1	2	3
bin_rank	00	01	10	11
$perm\_bin\_rank$	00	10	01	11
$file\_offset$	0	8	4	12

Table 4. Example of bin_rank, $perm\_bin\_rank$ and $file\_offset$ values for $N_x = 16$, $N_p = 4$ and $B_x = 1$.

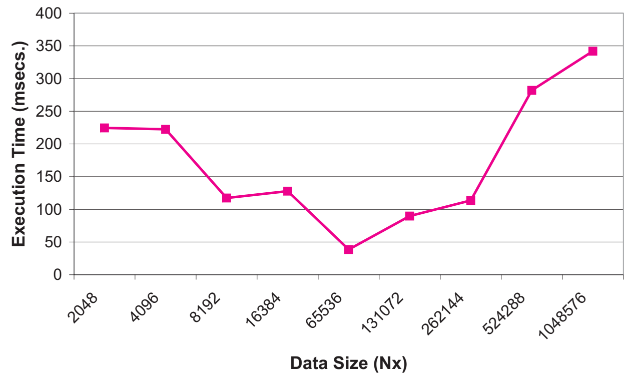

Figure 8. IEC I/O Inspector computation time (msecs.) for 16 processes with different N_x values and $N_b = 8$.

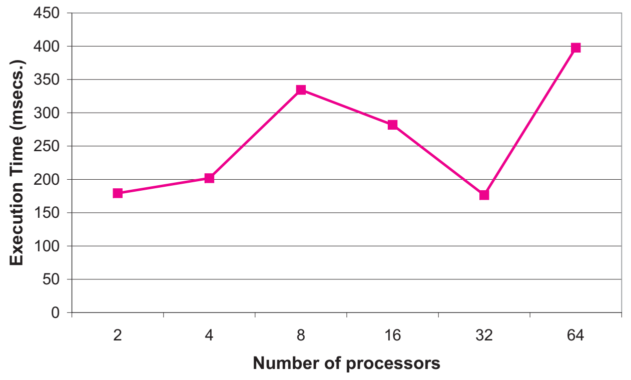

Figure 9. IEC I/O Inspector computation time (msecs.) for different processes with $N_x = 524288$ and $N_b = 8$.

4 Experimental Results

We have evaluated each component of our method under different execution environments. The platform used has the following specifications: 16 dual nodes (Intel Pentium III at 800 MHz, 256KB L2 cache, 1GB memory), Myrinet and FastEthernet interconnection networks, 2.6.13-15.8-bigsmp O.S. For the Myrinet network we have used the MpichGM 2.7.15 distribution. For FastEthernet, Mpich 1.2.6 was used. The parallel filesystem was PVFS 1.6.3 [4] with one metadata server and a striping factor of 64KB. The local filesystem corresponds to an $ext3$ partition of Linux. Each data element (given by N_x) and stride element represents a float number of 4 bytes.

This section is divided into two parts. First, the performance of the parallel I/O technique is analyzed, taking into account the datatype generation algorithm as well as the communication phases and disk access. Then, in the second part the performance of our method is compared with other state-of-the-art approaches.

4.1 Performance Analysis

A contribution of our method consists in splitting the global algorithm into two components: the inspector and the executor. The first one analyzes the data distribution and computes the required datatypes. The second one performs the data exchange and the disk access. This strategy allows amortizing the inspector overhead by means of datatype reusing. We have taken into account this division for measuring the algorithm performance: instead of making one measure of the whole method, we have evaluated the performance of each element. In following section we show and discuss the measured performance of each one of the stages.

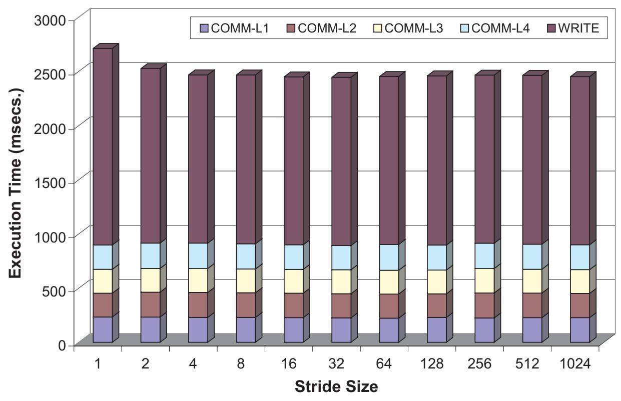

Figure 10. IEC I/O Executor time for a 300MB file, $N_p = 16$ and Myrinet network.

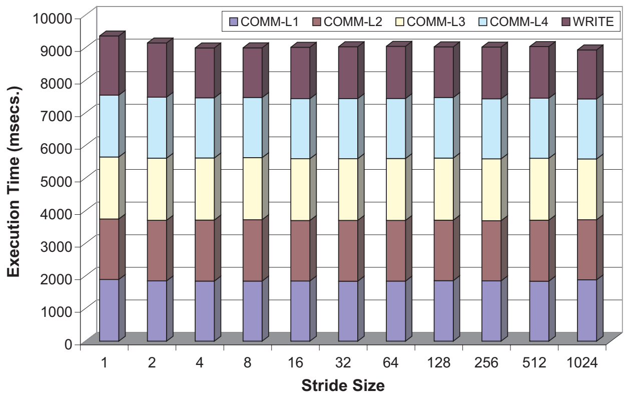

Figure 11. IEC I/O Executor time for a 300MB file, $N_p = 16$ and FastEthernet network.

The number of datatype tuples (offset and length pairs) is an important factor in the overall algorithm performance. They consume system resources like memory space, CPU

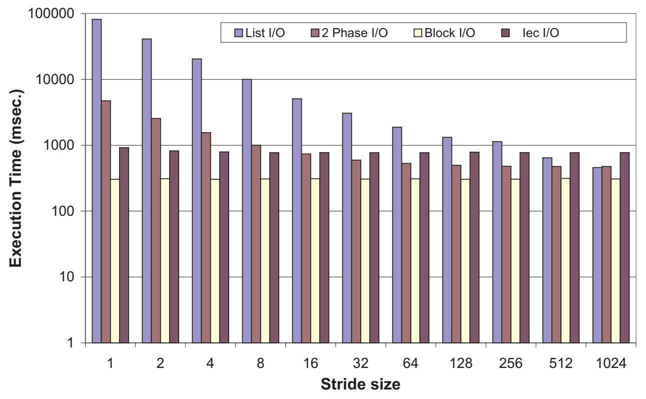

Figure 12. Comparative study for a 100MB file, $N_p = 16$ and Myrinet network.

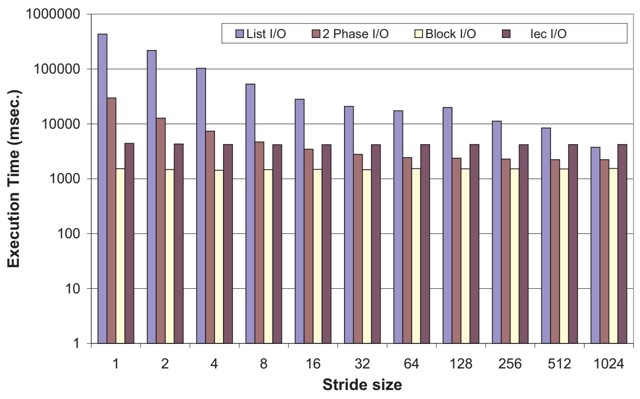

Figure 13. Comparative study for a 500MB file, $N_p = 16$ and Myrinet network.

and network bandwidth. In the case of the IEC I/O algorithm, we have two datatype structures: sending and packing data. The number of tuples does not depend on the problem size(N_x) in case of regular distributions. In our experiments, we obtained a constant value of 16 tuples in both datatypes. Another factor that we have to consider is the inspector overhead. Figure 8 shows the datatype computation time for different data sizes. IEC I/O algorithm exhibits a good behavior with respect to N_x.

The relationship between inspector overhead and the number of processors is shown in Figure 9. Note that the cost of the IEC I/O inspector increases sublinearly with N_p. This is due to the fact that that inspector algorithm is fully parallel and scales relatively well with N_p.

Regarding the performance of the rest of the stages (data exchange and disk write operation), figures 10 and 11 show the execution time for Myrinet and FastEthernet networks, respectively. Different B_x values were used for fixed values of $N_x = 78643200$ (300MB) and $N_p = 16$. PVFS filesystem employed 8 I/O nodes. The execution time is divided into the three communication stages and the disk write access. Note that the amount of data exchanged does not depend on B_x. For this reason, the communication times are almost constant for all the phases and B_x values. When FastEthernet communication network is used (PVFS filesystem keeps using Myrinet) there is a significant increment of the communication cost, but the algorithm performance does not degrades for different N_b values. Based on this figures we can conclude that the performance of the executor does not depends on N_b. Note that our method shows similar performance in all the distribution scenarios (for different strides). As we will see in the following section, this does not occur with other techniques.

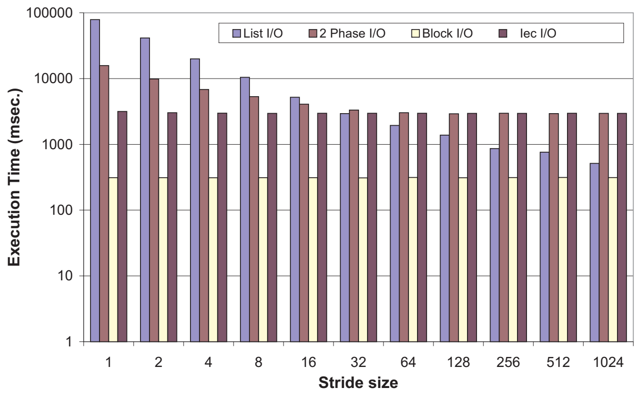

Figure 14. Comparative study for a 100MB file, $N_p = 16$ and FastEthernet network.

4.2 Performance comparison

We have compared the performance of our method with three parallel disk techniques: List I/O, Two Phase I/O and Block I/O. Block I/O technique consists of writing the distributed x entries in consecutive disk positions. Taking into account the initial requirements, it is not a valid I/O technique because disk entries are not properly sorted. In fact, disk entry order depends on how the array was initially distributed and, as we commented in the introduction, we understand that it is not a valid disk distribution. We have chosen this technique as a *reference* approach: It performs the most efficient I/O operation given that each processor writes different chunks of data with the maximum locality and without communication.

Figures 12 and 13 shows the comparative study using Myrinet network for $N_x = 26214400$ and $N_x = 131072000$, respectively. 16 processing nodes were used in all the cases. IEC I/O includes executor performance (containing both the communication phases cost plus the

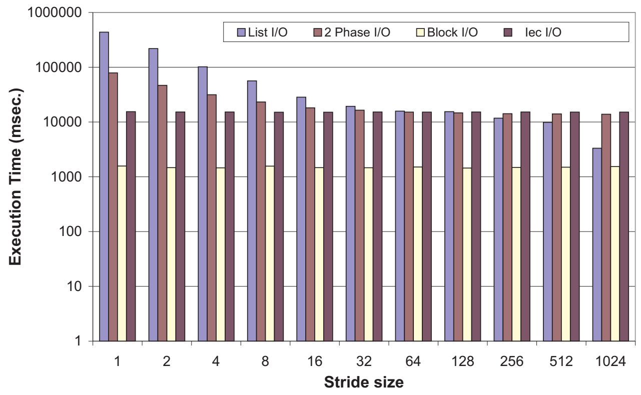

Figure 15. Comparative study for a 500MB file, $N_p = 16$ and FastEthernet network.

I/O cost). We can observe that (as expected) Block I/O obtains the best performance. For small B_x values, both List I/O and Two Phase I/O exhibit poor performance. In contrast, the performance of IEC I/O executor does not depend on neither B_x or N_x, reaching values close to the reference technique (Block I/O). When B_x increases, the performance of Two Phase I/O improves, reaching the IEC I/O performance for $B_x = 16$. List I/O requires of $B_x = 512$ or greater for reaching IEC I/O performance.

The reason of the low efficiently of List I/O is the poor management of the offset-length structures for small data grains. These structures were designed for facilitating the management of blocks of data. When the block sizes are small, the overhead of handling these structures is too large for a feasible option. In the case of Two Phase I/O, for small B_x the communication cost increases. For this technique it is necessary to determine the communication pattern among the processors and which entries need to be exchanged. The smaller B_x is, the greater is the overhead of disk access.

Figures 14 and 15, compare the performance for FastEthernet network. Now we can see that the overall execution time of methods that require inter-processor communication (Two Phase I/O and IEC I/O) increases. In contrast, the performance of both List I/O and Block I/O is similar to the one measured for the Myrinet Network. This is due to a slower inter-processor network. Note also that PVFS filesystem keeps using Myrinet network. For larger B_x, List I/O performance is better than Two Phase I/O. We can also note that for larger B_x, there is not an important difference between the performance of IEC I/O and Two Phase I/O.

5 Conclussions

Comparing the IEC I/O algorithm with Two Phase I/O, our method presents several advantages. It allows communication parallelism: during each communication phase, processors are organized in couples and perform send/receive private point to point communications. This provides a high parallelism degree because several communication operations can be performed at the same time. Additionally, because all processors send and receive the same amount of data, the communication is well balanced.

Based on the results from this paper, we conclude that our technique performs best for distributions with low granularity (small B_x). In these situations, IEC I/O algorithm outperforms the List I/O and Two Phase I/O techniques. The performance of IEC I/O depends few on B_x, allowing a good average performance for a broad number of distributions (B_x values). Another important contribution consists in splitting the inspector stage from the executor. Using this technique we can strongly reduce the overall algorithm cost in cases of repetitive I/O operations with the same access pattern.

References

[1] R. Bordawekar. Implementation of Collective I/O in the Intel Paragon Parallel File System: Initial Experiences. In *Proc. 11th Int. onference on Supercomputing*, 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] D. Kotz. Disk-directed I/O for MIMD Multiprocessors. In *Proc. of the First USENIX Symp. on Operating Systems Design and Implementation*, 1994.

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

[5] Message Passing Interface Forum. *MPI2: Extensions to the Message Passing Interface*, 1997.

[6] 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.

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

[8] D. Singh, F. Isaila, A. Calderón, F. Garcia, and J. Carretero. Multiple-phase I/O technique for improving data access locality. In *PDP 2007*, 2007.

[9] D. E. Singh, F. Isaila, J. C. Pichel, and J. Carretero. Inspector executor collective I/O internal structure. Technical report, Computer Architecture Group,Inspector-Executor collective I/O (IEC I/O) University Carlos III of Madrid., 2007. www.arcos.inf.uc3m.es/∼desingh/reports.html.

[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] R. Thakur, W. Gropp, and E. Lusk. On Implementing MPI-IO Portably and with High Performance. In *Proc. of the Sixth Workshop on I/O in Parallel and Distributed Systems*, pages 23–32, May 1999.