Optimized Process Placement for Collective I/O Operations

Vishwanath Venkatesan
Department of Computer Science
University of Houston
venkates@cs.uh.edu

Rakhi Anand
Department of Computer Science
University of Houston
rakhi@cs.uh.edu

Edgar Gabriel
Department of Computer Science
University of Houston
gabriel@cs.uh.edu

Jaspal Subhlok
Department of Computer Science
University of Houston
jaspal@uh.edu

ABSTRACT

Mapping of MPI processes to the available resources is an increasingly complex but important task on modern parallel systems. This paper presents a new approach to optimize the process placement of a parallel application based on its I/O access pattern. The paper introduces the *SetMatch* process mapping algorithm, which significantly reduces the cost of the communication occurring in collective I/O operations. The effectiveness of the approach has been evaluated for multiple scenarios on a PVFS2 file system. Our results demonstrate significant improvements in the communication time of collective I/O operations as well as improvements in the overall application execution time with our mapping strategy. The generalized *SetMatch* algorithm was the only mapping strategy that was able to provide adequate performance for all scenarios used in this paper.

1. INTRODUCTION

The complexity of modern micro-processors and the utilization of hierarchical networks makes process placement, i.e., the decision on where to place each MPI process, an increasingly difficult but important task. Various projects aim to map the communication pattern of the application to the underlying hardware such that process pairs with high communication volume are close to each other from the hardware perspective [4, 8], often focusing on specific network topologies such as torus or mesh networks [1, 13], hierarchical networks [11] or by minimizng network congestions [7].

An often overlooked component in the process placement research is the I/O occurring in the application. The time spent in I/O operations dominates the overall execution time for an increasing number of data intensive applications since the communication and computational components of high-end systems evolve at a faster rate than the storage components. MPI I/O has been shown to be beneficial for the I/O performance of many application due to features such as the fileview – which allows registering the I/O access pattern of processes in advance – and collective I/O operations, which represent the counterpart of group communication operations for file I/O. In most applications the logical organization of the processes within the fileview, i.e., the order in which processes access the file based on their offset into the file, is unique, since it is tightly coupled to the data distribution strategy used by the application.

This paper presents an approach to optimize the process placement of a parallel application based on their I/O access pattern. Specifically, the paper focuses on optimizing the communication occurring in collective I/O operations. We present the *SetMatch* algorithm which calculates a near-optimal process placement at minimal cost that minimizes communication time in collective I/O operations. This work makes a significant contribution towards the solution of the general problem, and also presents a simplified approach for commonly occurring data access patterns such as 2-D and 3-D data distributions and process topologies.

The organization of the document is as follows: section 2 discusses the collective I/O algorithms used, while section 3 presents the details of the approach that is at the core of this paper. Section 4 discusses performance results obtained with various process placement algorithms for multiple benchmarks. Section 5 summarizes the paper and presents the ongoing work in this domain.

2. BACKGROUND

The goal of this work is to optimize the communication occurring in collective I/O operations based on the access pattern of the application, and optimize the mapping of processes to the hardware correspondingly. The two collective I/O algorithms considered in this work are the two-phase I/O algorithm [5] and the dynamic segmentation algorithm [2].

As the name implies, the two-phase I/O algorithm consists of two steps. For a write operation, the first phase – the shuffle step – redistributes data among the processes to match the layout of the data in the file, while the second phase executes the actual write operation. In addition, the two-phase I/O algorithm introduces two further optimizations. First, only a subset of the MPI application processes

EuroMPI 2013, Madrid, Spain

have to touch the file, i.e., perform read or write operations. The processes executing actual I/O operations are also referred to as the *aggregators*. Second, for very large collective read and write operations, the two-phase I/O algorithm is split into multiple cycles internally. This keeps memory requirements on the aggregator processes within reasonable limits, and allows for potential overlap of the shuffle step and the write operation of subsequent cycles.

The dynamic segmentation algorithm [2] is an extension of the two-phase collective I/O algorithm, which subdivides processes into independent groups. In contrast to two-phase I/O, this algorithm does not create a globally sorted data array based on the offsets in the file. Instead, each aggregator performs the data gathering/scattering and sorting only within its group. This replaces the Alltoall(v) type communication in the two-phase I/O algorithm by a number of independent Allgather(v) operations in the dynamic segmentation algorithm.

3. PROCESS PLACEMENT FOR COLLECTIVE I/O

For the subsequent discussion, the process placement problem can be formally described as follows. Given an architecture matrix P, where each element of the matrix P_{ij} is the bandwidth capacity between processors i and j; and an application matrix R, where each element of the matrix R_{ij} is the amount of data communicated between processes i and j. The goal is to find a mapping of processes onto processors which optimizes the total communication costs, where the costs between a pair of processes i, j is R_{ij}/P_{ij}.

From the technical perspective, the problem can be broken down into three components: i) generating the architecture matrix; ii) generating a description of the communication pattern to create the application matrix; and map application processes to underlying node architecture such that communication cost is minimized. In the following, we discuss the most relevant aspects of our work in more details.

The architecture matrix is generated by providing a description of the hardware used for the application and is based, within the context of this work, on bandwidth values between pairs of processors. The bandwidth values are determined by using a ping-pong benchmark between cores on the same node, and cores on different nodes.

3.1 Application Matrix

The application matrix contains the amount of data communicated between each pair of processes. The fundamental assumption in the work is, that MPI processes reading/writing neighboring portions of a file communicate with the same aggregator processes during the collective I/O operations.

To support arbitrary access patterns, the order in which processes access the file based on the offset into the file has to be recorded. This can be done during `MPI_File_set_view` and written into a separate file, that can be used when re-executing the same problem/application. For applications not setting the file view, the offsets of each file access can be recorded during the `MPI_File_read/write` operations, although this is not supported in the current implementation. To minimize the size of the output file during the record operation, a compressed row storage (CRS) format is used to record the matrix. In the following, we illustrate how the application matrix is constructed based on the file I/O access pattern.

Consider an application in which the file view is set such that processes access the file in the following order:

$$0; 4; 1; 0; 4; 1; 5; 2; 3; 4; 1; 3; 2; 5; 4; 2; 5; 4; 0; 4; 1$$

with each number representing the rank of the process accessing the next portion of the file. Every time two processes have a neighboring region in the file, i.e. they are adjacent in the list shown above, the value representing the amount of communication between those processes is increased by one in the application matrix. For example, processes 0 and 4 have four neighboring file regions in the sequence shown above and consequently have a value of 4 in the matrix, while the processes 1 and 5 only have one common boundary. This results in the following matrix:

$$\begin{pmatrix} 0 & 1 & 0 & 0 & 4 & 0 \\ 1 & 0 & 0 & 1 & 4 & 1 \\ 0 & 0 & 0 & 2 & 1 & 3 \\ 0 & 1 & 2 & 0 & 1 & 0 \\ 4 & 4 & 1 & 1 & 0 & 2 \\ 0 & 1 & 3 & 0 & 2 & 0 \end{pmatrix}$$

While creating the application matrix by recording the file view works for arbitrary access patterns, it requires running the application to build the application matrix. Note, that the same problem occurs in many other projects working on process placement problems [4, 8], and is therefore not specific to our approach. However, we also developed a significantly simplified methodology to generate the application matrix for certain common data distributions. The most important scenario covered by this simplified approach are applications using a 2-D data distribution and a 2-D cartesian process topology. In this particular scenario, the vast majority of the communication in the collective I/O operation will occur between processes having the same coordinate in the outermost dimension of the cartesian process topology, assuming that one process per row will act as an aggregator in the collective I/O operation. The application matrix can be created in this case by assigning process pairs with the same coordinate in the outermost dimension of the cartesian grid a larger value than between process pairs in different rows of the cartesian process topology.

3.2 Mapping Algorithm

Generally speaking, once the architecture and the application matrix are available, any graph mapping algorithm from literature could be used for the mapping step. Our initial focus was on the algorithm used by the MPIPP toolset [4]. This algorithm uses a random mapping of processes to available cores as a starting point, swaps a pair of processes and recalculates an objective function for each new mapping. In case the new configuration processes shows benefits, i.e. a lower value of the objective function, the modified configuration is kept, otherwise it is undone. The algorithm stops if no improvement can be made over multiple iterations.

The main drawback of this algorithm is the time it takes to run to completion for large process counts. In our study we found that for the 256 processes test case the algorithm can take up to 90 minutes on a typical desktop system to compute the mapping with two passes, and still produces suboptimal mapping due to the low number of passes.

Therefore, we developed a much simpler algorithm called

SetMatch which is a simplified version of the *Treematch* [8] algorithm for these type of communication and architecture matrices. Similarly to Treematch, this algorithm first partitions the application and architecture matrix into smaller, independent sets. For the architecture matrix this can be achieved by grouping all processors/cores on the same node. For the application matrix, this is typically achieved by grouping all processes with high communication volumes. In the second step of the algorithm, each subgroup of processes that resulted from subdividing the application matrix is mapped to subgroups of the architecture matrix.

There are two main differences between the *SetMatch* and the Treematch algorithm: first, *SetMatch* only uses one level of hierarchy; second, we choose to ignore insignificant values in the application matrices, and replace them by a value of 0. Specifically, any value that falls within a certain percentage of the maximum value found (e.g. 10%) in the application matrix is used for creating independent application sets while other values are ignored. This reduces the cost involved in creating the independent groups of the application matrix, which is the main cost of the *Treematch* [8] algorithm. Pseudo-code for the *SetMatch* algorithm is provided in Algorithm 1.

Once the initial sets are created, they are checked for interleaving, to ensure that they are completely independent. In case of the specialized application matrix this approach will result in independent sets. This cannot be guaranteed however for the generalized application matrix scenario, and necessitates a merging of interleaved sets. This is shown in Phase II of the algorithm. Once all sets are independent, each application set has to be matched to an architecture set. The goal is always to fit an entire application set into an architecture set.To accomplish this, the algorithm locates the largest possible architecture set to match the application set. Once the architecture processes are matched, the existing architecture set is fragmented to be used for another application set. If there are no architecture sets found that could match the application sets, the algorithm fragments the application matrix and maps it to the next biggest architecture set available. This is shown in Phase III from line number (36) of the algorithm.

3.3 Implementation

The work has been implemented in OMPIO [3], a parallel I/O framework in Open MPI [6]. While the implementation is specific to OMPIO, the conceptual aspects of this work can easily be extended and transferred to other implementations. A new component of the rank mapping framework (rmaps) has been created, the Data Locality Aware Mapping (dla) component. Generally speaking, an rmaps component retrieves information about the allocated resources, the number of processes requested, and creates a mapping for them. The output is an array of ranks which is used to associate application processes to actual nodes. The new dla rmaps component takes an additional input file, which allows to specify either the cartesian topology or the actual application matrix generated from the fileview as an input. The module is available for free download on the authors' webpage, and will be contributed to the Open MPI source code in the near future.

4. EVALUATION

The efficiency of the approach discussed in section 3 is

Algorithm 1 ***SetMatch***

Input: num_procs, comm_matrix, arch_matrix, min_value
Output: ranklist

```
 1: procedure MAP_SETMATCH
 2:     make_sets                                  ▷ app_sets
 3:     merge_sets                                 ▷ remove interleaving
 4:     make_sets                                  ▷ Independent arch_sets
 5:     match_sets                                 ▷ generate ranklist
```

Phase - I Creating Initial Sets

Input: num_procs, matrix, matrix_type, min_value
Output: num_sets, sets

```
 6: function MAKE_SETS
 7:     availList[1..|N];Boolean, numsets, sets
 8:     Initialize availList
 9:     for i=0 to N do
10:         if availList then
11:             continue
12:         numsets++
13:         Allocate new set of size N and initialize
14:         for j=0 to N do
15:             if matrix_type = APP_MATRIX then
16:                 if matrix[i][j] ≤ min_value then
17:                     Add process to the current app_set
18:                     Update Availability List
19:             if matrix_type = ARCH_MATRIX then
20:                 if matrix[i][j] ≤ min_value then
21:                     Add this process to the current arch_set
22:                     Update Availability List
```

Phase - II Merging Interleaved Sets

Input: set, numSets, numProcs
Output: finalSets, final_set

```
23: function MERGE_SETS
24:     Allocate and initialize final_set and finalSets
25:     copy (final_set[0], set[0])
26:     num_final_sets++
27:     for i ← 1 to numSets do
28:         setFound ← false
29:         for j ← 0 to finalSets do
30:             if groups_interleave(final_set[j], set[i]) then
31:                 copy (final_set[j], set[i])
32:                 setFound ← true
33:                 break
34:         if ¬setFound then
35:             finalSets++
36:             copy (final_set[finalSets - 1], set[i])
```

Phase - III Match application and architecture sets

Input: app_set, arch_set, archSets, appSets, numProcs
Output: ranklist

```
37: function MATCH_SETS
38:     while mappedProcs ≠ numProcs do
39:         for i ← 0 to appSets do
40:             archFragment ← false
41:             for j ← 0 to archSets do
42:                 if (arch_set[j].nprocs is higher) then
43:                     archFragment ← true
44:                     map(ranklist, archSet[i], appSet[j])
45:             if ¬archFragment then
46:                 map(ranklist, archSet[j], appSet[i])
47:             sort(appSet)
48:             sort(archSet)
```

evaluated for several scenarios. The platform used is the *crill* cluster at the University of Houston which consists of 16 nodes with four 12-core AMD Opteron (Magny Cours) processors each (48 cores per node, 768 cores total), 64 GB of main memory and two dual-port InfiniBand HCAs per node. The cluster has a PVFS2 (v2.8.2) parallel file system with 15 I/O servers and a stripe size of 1 MB. The file system is mounted onto the compute nodes over the second InfiniBand network interconnect of the cluster.

Six different mapping approaches were taken into consideration for evaluation are i) *Byslot*: Linear mapping on available slots; ii) *Bynode*: Round robin based mapping; iii) *MPIPP*: MPIPP with cartesian topology; iv) *MPIPPG*: Generalized MPIPP using the application matrix generated by recording the fileview; v) *SetMatch*: *SetMatch* algorithm with cartesian topology; vi) *SetMatchG*: *SetMatch* algorithm using the application matrix generated by recording the fileview.

A pre-release version of OpenMPI v1.7 was used for the measurements.A simple point-to-point benchmark from the OSU microbenchmark suite [9] was used to determine the intra-node and inter-node bandwidth for the architecture matrix. The intra-node bandwidth obtained with these tests were peaking at around 5800MB/s – although the bandwidth dropped for larger message lengths to around 3500MB/s. The inter-node bandwidth using the DDR InfiniBand network interconnect is up to 2100 MB/s.

4.1 Tile I/O

The MPI Tile I/O benchmark is widely used to study performance of MPI-IO for tiled accesses to a two-dimensional dense dataset [10]. The benchmark uses a cartesian communicator to access the file which makes it an ideal benchmark to study the effectiveness of our mapping strategies. In our measurements we select the number of tiles in the x and y dimension based on the number of processes. We use two different tile sizes (1KB, 1MB) with (768x800, 40x15) elements respectively. All tests were executed three times and the maximum achieved bandwidth across those runs is selected.

Tests were executed using 128 processes on four nodes and 256 processes on eight nodes. Note, that 128 processes could be executed on three nodes of our cluster, and 256 processes on six nodes. However, we choose to allocate a slightly larger number of nodes due to performance considerations of our parallel file system. This leads in most scenarios to a higher number of cores being available for the process placement algorithm than actually requested by mpirun. In addition, the number of aggregators used in the collective I/O algorithms was kept constant and same as the number of nodes used.

Figures 1 and 3 show the average communication time in the communication step of the dynamic segmentation and the two-phase I/O algorithm with the different mapping strategies, since the variance in these measurements was negligible. Our mapping methods show significant reduction in communication times compared to the standard bynode and byslot mapping strategies of Open MPI with the dynamic segmentation algorithm, except for the 256 processes case with the MPIPP algorithm. Our analysis of the degraded performance for this scenario revealed a suboptimal mapping by the MPIPP algorithm, since the number of internal iterations had to be limited to two due to the exceeding amount of time spent in the mapping algorithm. The *SetMatch* algorithm provides the best performance in comparison to all other methods at significantly lower costs for the mapping step: calculating the process placement for the 256 process scenario took around 100 ms on this platform.

Fig 2 shows the resulting I/O bandwidth for Tile I/O benchmark using the dynamic segmentation algorithm. The results indicate a significant improvement in the write bandwidth along the lines of what is expected based on the savings in the communication time.

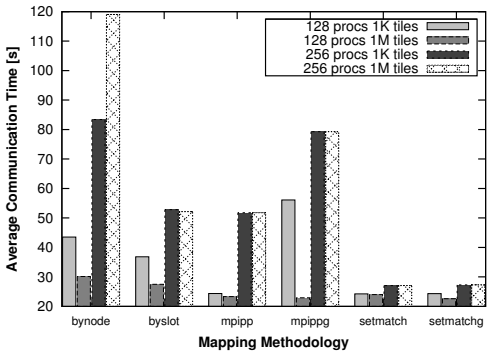

Figure 1: Communication time for different mapping strategies in the dynamic segmentation algorithm on crill.

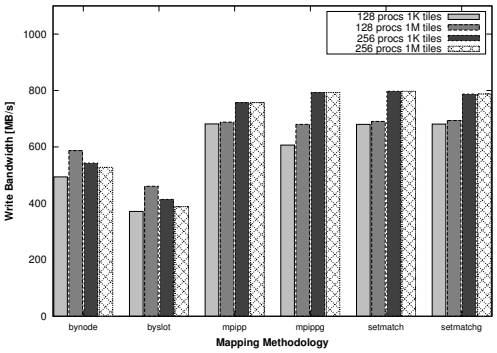

Figure 2: Bandwidth comparison for different mapping strategies using the dynamic segmentation algorithm on crill.

Trends are mostly similar with the two-phase I/O algorithm, with some interesting deviations as shown in figure 3. The first difference is evident when comparing the communication times obtained with the default bynode and byslot mappings. Based on the analysis of the data access and communication pattern of the benchmark, the byslot mapping should lead to lower communication times compared to the bynode mapping, similar to the results obtained using dynamic segmentation algorithm. This is however not the case in this test. The reason for this behavior is that the byslot mapping in Open MPI will fill up all the 48 cores on an individual node before assigning processes to another node, which increases the memory pressure on each node dramatically and negates the performance benefits of the

intra-node communication. Fig. 4 shows a slightly modified version of the same test, in which we compare the default byslot mapping of Open MPI to a byslot mapping restricting the maximum number of processes per node to 32, which is the number of MPI processes per node in the bynode scenario. In this case, byslot clearly outperforms bynode for the two-phase I/O algorithm as expected.

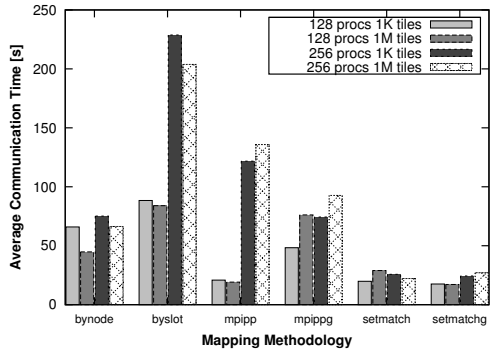

Figure 3: Communication time for different mapping strategies in the two-phase I/O algorithm on crill.

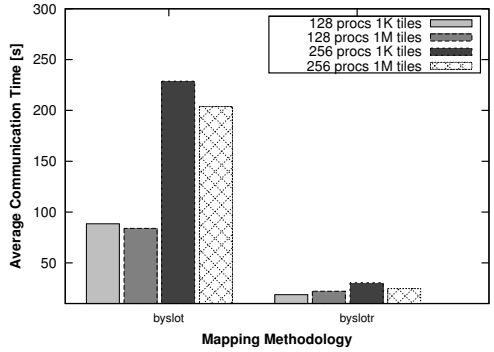

Figure 4: Average communication time for 48 and 32 processes per node using the byslot mapping with the two-phase I/O algorithm.

Second, there is no significant difference in the bandwidth observed using the two-phase I/O algorithm for any mapping approach (graph not shown due to space limitations). Two reasons have been identified for contributing to this behavior. First, the overall bandwidth achieved was in the range of 800 MB/s, which is close to the peak bandwidth supported by this parallel file system. Second, we added a file synchronization call using `MPI_File_sync` to the benchmark to avoid caching effects, which was originally not part of the Tile I/O benchmark. Removing this call led to bandwidth improvements up to 45% in the write bandwidth, although the improvement can be attributed mostly to caching effects on the server side. Ultimately, we are confident that the results observed in this test are an artifact of our parallel file system, since we did observe the expected improvements in the communication time using the two-phase I/O algorithm as well.

4.2 Modified Tile I/O

In addition to the normal Tile-IO benchmark we also created a new version where the communicator used in the benchmark is modified to reorder ranks internally and thus create a more irregular scenario. In the previous tests, the byslot mapping would have provided 'accidentally' a correct mapping, except for the fact that it overloads each node with processes. For the modified Tile I/O test, neither byslot nor bynode provides the correct mapping. The goal therefore is to demonstrate that the *SetMatch* algorithm provides the optimal mapping also for this irregular scenario. Fig. 5 shows that only the *SetMatch* algorithm with the generalized application matrix is able to provide consistently low communication times across all scenarios. The version of the MPIPP and *SetMatch* using the cartesian topology as an input can not correctly map in this scenario, since the cartesian topology is not created from `MPI_COMM_WORLD`, but from a different communicator. The generalized MPIPP algorithm can correctly describe the application matrix, but still produces a suboptimal mapping, especially for 256 processes. Also shown in this graph are numbers for a restricted byslot mapping, i.e. the mapping which limits the number of processes per node to 32. Neither bynode nor any version of the byslot produces a configuration leading to similar improvements in the communication time as the generalized *SetMatch* algorithm. Similar performance has been observed in our experiments with even the dynamic segmentation algorithm, which we are omitting here due to space constraints.

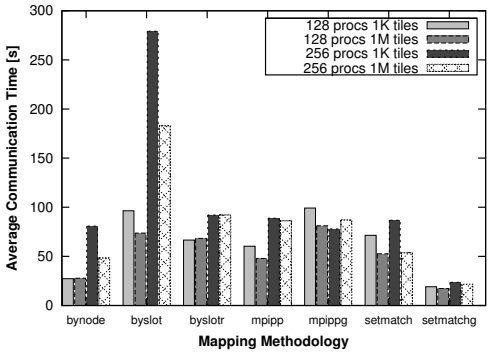

Figure 5: Communication time for different mapping strategies using the two-phase algorithm for the modified Tile I/O test.

4.3 BT-I/O

BT-I/O is a part of the NAS parallel benchmarks (NPB) suite [12]. It has been developed based on one of the kernels of the BT computational kernels. The class D of the BT-I/O benchmark used in this study writes 135 GB data over 250 iterations, i.e. around 600MB of data per-iteration. Due to the smaller amount of data written per function call, the communication times per iteration will also be low, which fundamentally limits the improvement per-iteration that can be achieved. Nevertheless, the results shown in Table 1 for the two-phase I/O algorithm indicate that using the setmatchg algorithm leads up to 30% reduction in I/O time in the benchmark and 17% reduction in total application

Table 1: I/O time and total execution time of BT I/O using two-phase I/O algorithm.

Mapping	No.of Processes	Avg Total Time (s)	Avg I/O Time (s)
Bynode	144	838.91	264.09
	256	549.27	193.58
Byslot	144	1100.88	407.3
	256	1020.7	637.75
mpipp	144	769.49	223.46
	256	461.28	134.38
mpippg	144	819.25	228.01
	256	493.84	151.96
setmatch	144	805.25	243.67
	256	477.36	135.13
setmatchg	144	792.25	232.96
	256	458.42	134.98

time. Similar performance was obtained using the dynamic segmentation algorithm, which we skip here due to space constraints.

5. CONCLUSIONS

This paper presents an approach to optimize the process placement of a parallel application based on its I/O access pattern. We extended the OMPIO library to record the access pattern of an application using the file view, and used the resulting data to map processes to the underlying hardware such that the communication occurring in collective I/O operations is minimized. A simplified approach which does not require to record the file view has been demonstrated for a regular 2-D data distribution and process topology. Furthermore, we present the SetMatch algorithm as a simple method to calculate a near-optimal process placement at a low cost. Our results demonstrate significant improvements in the communication time of collective I/O operations and the application scenarios overall due to our mapping strategy. When applicable, the simplified approach for 2-D and 3-D data distributions shows significant benefits without requiring to record the file view of an application first. For generic and irregular scenarios, the generalized *SetMatch* algorithm that we presented in this paper was able to provide adequate performance for all scenarios used in this paper. This work can be expanded in multiple directions, including more and larger application scenarios and platforms. We can also support process placement approaches for cases without fileview information. In addition, the special case that we demonstrated for the 2-D pattern can be extended to various other regular process distributions.

Acknowledgments. Partial support for this work was provided by the National Science Foundation's Computer Systems Research program under Award No. CNS-0834750. and No. CRI-0958464. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

6. REFERENCES

[1] A. Bhatele, L. V. Kale, and S. Kumar. Dynamic topology aware load balancing algorithms for molecular dynamics applications. In *Proceedings of the 23rd international conference on Supercomputing*, ICS '09, pages 110–116, New York, NY, USA, 2009. ACM.

[2] M. Chaarawi, S. Chandok, and E. Gabriel. Performance Evaluation of Collective Write Algorithms in MPI I/O. In *Proceedings of the International Conference on Computational Science (ICCS)*, volume 5544, pages 185–194, Baton Rouge, USA, 2009.

[3] M. Chaarawi, E. Gabriel, R. Keller, R. L. Graham, G. Bosilca, and J. J. Dongarra. OMPIO: A Modular Software Architecture for MPI I/O. In Y. Cotronis, A. Danalis, D. Nikolopoulos, and J. Dongarra, editors, *Recent Advances in Message Passing Interface*, pages 90–98, Santorini, Greece, September 2011. Springer.

[4] H. Chen, W. Chen, J. Huang, B. Robert, and H. Kuhn. Mpipp: an automatic profile-guided parallel process placement toolset for smp clusters and multiclusters. In *Proceedings of the 20th annual international conference on Supercomputing*, ICS '06, pages 353–360, New York, NY, USA, 2006. ACM.

[5] J. M. del Rosario, R. Bordawekar, and A. Choudhary. Improved parallel I/O via a two-phase run-time access strategy. *SIGARCH Comput. Archit. News*, 21(5):31–38, 1993.

[6] E. Gabriel, G. E. Fagg, G. Bosilca, T. Angskun, J. J. Dongarra, J. M. Squyres, V. Sahay, P. Kambadur, B. Barrett, A. Lumsdaine, R. H. Castain, D. J. Daniel, R. L. Graham, and T. S. Woodall. Open MPI: Goals, Concept, and Design of a Next Generation MPI Implementation. In *Proceedings of the 11th European PVM/MPI Users' Group Meeting*, pages 97–104, Budapest, Hungary, 2004.

[7] T. Hoefler and M. Snir. Generic Topology Mapping Strategies for Large-scale Parallel Architectures. In *Proceedings of the 2011 ACM International Conference on Supercomputing (ICS'11)*, pages 75–85. ACM, Jun. 2011.

[8] E. Jeannot and G. Mercier. Near-optimal placement of MPI processes on hierarchical NUMA architectures. In *Proceedings of the 16th international Euro-Par conference on Parallel processing: Part II*, pages 199–210. Springer-Verlag, 2010.

[9] OSU-micro benchmark homepage. http://mvapich.cse.ohio-state.edu/benchmarks/, 2002.

[10] R. Ross. *Parallel I/O Benchmarking Consortium.* http://www-unix.mcs.anl.gov/ross/pio-benchmark.html.

[11] J. Traff. Implementing the MPI process topology mechanism. In *Supercomputing, ACM/IEEE 2002 Conference*, pages 28–28, 2002.

[12] P. Wong and R. F. V. der Wijngaart. *NAS Parallel Benchmarks I/O Version 2.4. Technical Report.* NAS-03-002, Computer Sciences Corporation, NASA Advanced Supercomputing (NAS) Division.

[13] H. Yu, I.-H. Chung, and J. Moreira. Topology mapping for blue gene/l supercomputer. In *Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, SC '06, New York, NY, USA, 2006. ACM.