Automatically Selecting the Number of Aggregators for Collective I/O Operations

Mohamad Chaarawi and Edgar Gabriel

Parallel Software Technologies Laboratory, Department of Computer Science,
University of Houston, Houston, TX 77204-3010, USA.
Email: {mschaara, gabriel}@cs.uh.edu

***Abstract*— Optimizing collective I/O operations is of paramount importance for many data intensive high performance computing applications. Despite the large number of algorithms published in the field, most current approaches focus on tuning every single application scenario separately and do not offer a consistent and automatic method of identifying internal parameters for collective I/O algorithms. Most notably, published work exists to optimize the number of processes actually touching a file, the so-called aggregators. This paper introduces a novel runtime approach to determine the number of aggregator processes to be used in a collective I/O operation depending on the file view, process topology, the per-process write saturation point, and the actual amount of data written in a collective write operation. The algorithm is evaluated on two different file systems with multiple benchmarks. In more than 80% of the test cases, our algorithm delivered a performance close to the best performance obtained by hand-tuning the number of aggregators for each scenario.**

I. Introduction

Parallel computing is typically associated with problems that are considered too large to be solved on a single, standard desktop system in an acceptable amount of time. Due to the size of the problems solved, many parallel applications also have to operate with large amounts of data. Scientific applications consuming and producing tens or even hundreds of gigabytes of data are not uncommon on high-end systems. These data intensive applications spend a significant amount of their overall execution time in actual I/O operations, i.e., reading and writing the data.

Often, data access operations do not scale in a similar manner as compute operations. The most important factor limiting the performance of I/O operations is the performance of an individual magnetic hard drive, which is multiple orders of magnitude slower than any other component in a computer. In an effort to overcome this problem, storage systems attached to high-performance computing systems comprise thousands of magnetic hard drives. Exploiting the full potential of such a storage system is however not trivial, leaving parallel applications with two options: either each process operates on a separate file, or processes have to jointly access (different parts of) a single file. The first approach has the drawback of overwhelming the metadata server of parallel file systems in case tens of thousands of files are being created simultaneously. Furthermore, it often requires time consuming pre- and post-processing steps in order to create/merge the input and output files for the required number of processes used. The second approach requires coordination between the processes accessing the same file to ensure consistency, a feature that is not supported by the default POSIX I/O interfaces supported by today's operating systems.

Version Two of the Message Passing Interface (MPI) specification [12] added interfaces for handling files in a parallel application, leveraging concepts from the previous MPI document. Among the features introduced in MPI I/O is the notion of collective I/O operations, i.e., file access operations executed simultaneously by a group of processes. While collective I/O interfaces offer the possibility for collaboration among processes and therefore significant performance improvements compared to the sequential POSIX I/O operations, today's implementations suffer in many cases from suboptimal performance.

A number of algorithms have been presented over the last few years optimizing collective I/O operations, many of them taking advantage of features of parallel file systems, parameters of the message passing and I/O network, application access patterns, and data volumes written/read. One of the most influential parameters in collective I/O operations is the number of aggregator processes being used. This parameter influences the size of contiguous data chunks presented to the file system and the number of processes actually accessing the file. Having too few aggregators will not exploit the full bandwidth of the storage system. Having too many aggregators will create a large number of smaller chunks, which might be suboptimal from the file system perspective and increases the costs of the shuffle operations. To date, no consistent approach has been presented on how to determine the ideal number of aggregator processes for collective I/O operations.

This paper presents an algorithm to determine dynamically the number of aggregator processes used in a collective I/O operation, such that the data throughput for collective I/O operations is maximized. For the sake of clarity, the analysis focuses on write operations, although the approach presented here can easily be applied to read operations as well. We evaluate our algorithm for a number of different application scenarios and process counts on two different storage systems, one using a PVFS2 file system and one using a Lustre file system.

The remainder of the paper is organized as follows: Sec-

tion II discusses the most relevant related work in collective I/O algorithms. Section III introduces the components and algorithms upon which our work is based on. Section IV presents the algorithm used to determine the optimal number of I/O aggregator processes for collective I/O operations. We evaluate our algorithm in Section V. Finally, in Section VI we summarize this work and outline the future work in this area.

II. Related Work

Collaboration among processes is widely considered the most relevant aspect for achieving good performance for I/O operations in parallel applications. The most widely used optimization for collective I/O operations is the two-phase I/O [4] algorithm. This algorithm shuffles the data among the MPI processes to match the data layout in the file and performs the actual read/write operation on a subset of processes called the aggregators. The algorithm is discussed in more detail in section III-A. Two phase I/O has been the basis for a number of other approaches including datatype I/O [15], view-based collective I/O [1], resonance I/O [20], dynamic segmentation and static segmentation algorithms [2]. Liao et.al. [10] analyzed various file domain partitioning methods for improving the performance of collective I/O operations. Most of these algorithms tweak on how the shuffle and read/write operations are performed, but use the same fundamental approach as two-phase I/O uses. The most important aspect within the context of this paper for all algorithms is that they rely on a number of internal parameters, such as the number of aggregator processes to be used, but do not necessarily present a systematic approach on how to determine the optimal values for these parameters.

The work most closely related to our project has been performed by Worringen [18]. The author presents an adaptive approach for determining parameters of parallel I/O operations. His approach to tune the number of aggregator processes is similar to our algorithm presented here in that he relies on a per-aggregator threshold value. However, his work does not present how to determine the threshold value, nor does he take file-view specific options into account. Furthermore, he restricts the number of aggregators to particular ranges, which we will argue in this paper is not practical and portable.

Another relevant aspect in tuning collective I/O operations has been identified to be the locking protocol of some parallel file systems such as Lustre [11] and GPFS [14]. Adjusting the domain partitions assigned to an aggregator to the locking boundaries of the file system allows us to reduce contention for collective write operations [10], [5], [19] by minimizing the number of operations waiting for the lock. As will be discussed in section IV, we integrate the findings of these papers in our algorithm to reduce locking contention.

III. Background

The Open MPI Project [6] is an open source implementation of the MPI specification that is jointly developed and maintained by a consortium of academic, research, and industry partners. The library is built around the Modular Component Architecture (MCA), which allows a compile time or runtime selection of the components used by Open MPI. Each major functional area in Open MPI has a corresponding back-end component framework, which manages its own modules. This allows, for example, the *btl* framework to seamlessly support multiple network interconnects, since the required module for each network is loaded at runtime.

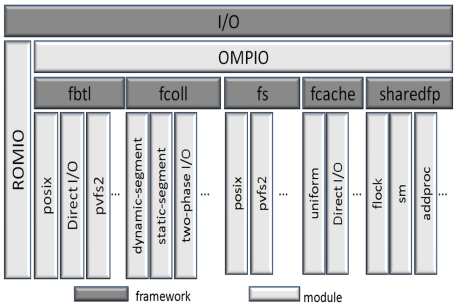

Fig. 1. Overview of the OMPIO component and its frameworks.

The same concept is exploited by the Open MPI I/O project (OMPIO) [3] to provide a highly flexible parallel I/O infrastructure. OMPIO is implemented as a module of the `io` framework of Open MPI. Upon opening a file using `MPI_File_open`, the selection logic of the *io* framework will query both the OMPIO module and the ROMIO [16] module, a widely used MPI I/O implementation developed by Argonne National Laboratory, then choose one for executing the actual I/O operations. In case the OMPIO module has been chosen, which as of today has to be explicitly requested by the end-user, a number of sub-frameworks are initialized. These are:

- The *file system framework (fs)*: abstracts general file manipulation operations, such as opening, closing, and deleting a file. The semantics of most of the operations are collective.
- The *file cache framework (fcache)*: provides the ability to cache information, such as the number of storage servers used, list of storage servers, and stripe depth. For some file systems, caching this information will allow a reduction in the traffic to the Metadata server and enable certain optimizations, such as sorting I/O requests on a per storage server basis.
- The *file byte-transfer layer framework (fbtl)*: provides the abstraction for all individual read and write operations.
- The *shared file pointer framework (sharedfp)* : abstracts functions for managing the shared file pointer.
- The *file collective I/O framework (fcoll)* : provides interfaces for the collective file I/O operations.

The frameworks and their architectural organization are displayed in fig. 1. Each framework typically has multiple modules implementing the required functionality, as well as a runtime selection algorithm that determines which component to use for a particular scenario. Note, that the rules for the runtime selection logic are strongly dependant on the functionality provided by the module and could be based,

for example, on the performance estimates of particular algorithms, file system utilized, process placements, or user-level hints. Within the context of this paper we will focus on the functionality of the collective I/O framework. In the following, we discuss the three most prominent collective I/O algorithms used by OMPIO, each of which is encapsulated into a separate module.

A. The collective I/O framework

The collective I/O framework provides the abstraction required to have different implementations of collective I/O operations. In contrast to most other frameworks, the decision of which collective I/O module to use is not determined when a file is being opened, but upon setting the file view [1].

Since different algorithms are deployed in separate modules, the user can switch between the modules at runtime, similarly to all other frameworks of Open MPI. For example, by adding the flags " `--mca fcoll dynamic` " to the mpirun command, the user enforces the utilization of the dynamic segmentation algorithm for collective I/O operations. Other algorithmic parameters can also be set at runtime such as the number of aggregators, stripe count/size of a newly created file, or number of bytes to write in one cycle without requiring recompilation of the MPI library, of the module, or of the application.

As of today, five collective I/O modules based on different algorithms are available in OMPIO. We will discuss three of them, focusing on their behavior for the write operation. Since the last two modules are irrelevant within the scope of this paper, they are omitted for the sake of brevity.

The first module implements the widely used *two-phase I/O* [4] strategy, which is the basis of many collective I/O optimizations utilized today. The algorithm divides the collective I/O operations into two phases. For write operations, these are:

1) Phase one redistributes data among the processes to match the layout of the data in the file. Since the data distribution of a parallel application might not match the layout of data in the file, re-arranging the data has often proved worthwhile. Specifically, on many file systems it is suboptimal to pose a large number of small, non-contiguous I/O requests. Re-arranging the data helps to create fewer and larger I/O requests and allows to combine data from different processes.
2) Phase two executes the actual write operation. The two-phase I/O algorithm also introduces the notion that only a subset of the application processes actually perform write operations on the file, the aggregators.

For very large collective write operations, the two-phase I/O algorithm is split into multiple cycles. This allows the amount of temporary data required on the aggregator processes to be kept within reasonable limits.

[1]Implicitly, a collective I/O module is also selected upon opening a file, due to the fact that the default file view is being set at this point in time. However, subsequent changes to the file view will trigger a re-evaluation of which collective I/O module to use.

The *dynamic segmentation* [2] module is an extension of the classical two-phase collective I/O algorithm. Similarly to two-phase I/O, the main goal of this algorithm is to combine data from multiple processes in order to minimize the number of I/O operations presented to the file system. In contrast to the two-phase I/O algorithm, the dynamic segmentation algorithm does not create a globally sorted data array based on the offsets in the file. Instead, each aggregator is assigned a group of processes and performs the sorting and data gathering/scattering only within its group. This allows executing the shuffle step including the sorting and data gathering/scattering more efficiently, since the all-to-all type communication in the two-phase I/O algorithm is replaced by a number of independent gather(v) operations in the dynamic segmentation algorithm.

Of major influence for the performance of the dynamic segmentation algorithm is how processes are grouped. In an `n` process scenario with `p` aggregators, each aggregator is currently assigned `n/p` consecutive processes. However, new algorithms that take the file view, process topology or process location into account have been developed and will be discussed in section IV.

The *static segmentation* module and the according algorithm extend the dynamic segmentation algorithm focusing on optimizing the shuffle operation itself and relaxing the requirement of creating consecutive chunks of data to be written. Contrary to the previous algorithm, an aggregator gathers a fixed number of bytes from each of the processes assigned to him in each cycle. This keeps the communication channels constantly busy and avoids the bulk-communication often occurring in two-phase I/O or dynamic segmentation. Note, that the algorithm does not necessarily reduce the overall number of I/O requests presented to the file system, but reduces the number of processes executing these I/O requests compared to overall number of application processes posting the collective write request.

Figure 2 demonstrates the differences among the two-phase I/O, dynamic segmentation and static segmentation algorithm. In this example, four processes collectively write 16 data items into a file. The write operations are further assumed to be executed in four cycles with two aggregator processes (rank 0 and 2), and a cycle buffer size of two items. For the dynamic and static segmentation, we assume that processes 0 and 1, and processes 2 and 3 form the according teams.

IV. Automatically determining the number of aggregators

Collective I/O algorithms have different parameters associated with them such as the number of aggregators that perform the actual low level I/O operations and the cycle buffer size which indicates the total size of data that will be written in one cycle by an aggregator process. The latter also determines the size of the temporary buffer to hold the data at the aggregator.

The choice of the best algorithm and the values for its corresponding parameters is not an easy task, if not impossible, for a regular user who is not aware and needs not to be aware

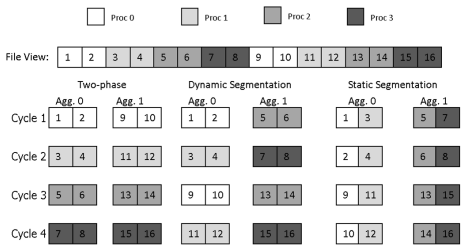

Fig. 2. Comparison of the data items written by the two-phase I/O, dynamic segmentation and static segmentation algorithm for four processes and two aggregators.

of the underlying algorithms. Having too few aggregators will create large, contiguous data streams, but the costs for the shuffle step might offset benefits of the I/O operation itself. Furthermore, depending on the parallel file system utilized, too few aggregators and therefore a small number of data streams might not be able to saturate the file system. On the other hand, having too many aggregators will create a large number of smaller, often not contiguous data streams, which might be suboptimal from the file system perspective. Ideally, one would like to determine the number of aggregator processes such that each aggregator is able to saturate its own resources.

Using default parameter values will inevitably lead to suboptimal performance. Results of a large number of analyses have shown that different types of access patterns, problem sizes, and platforms all play a major role in the performance and influence the parameter values. Pre-tuning those parameters using a benchmark that represents the access pattern of the application on the desired file system before execution is a viable option. However, this takes significant amount of time that some users may not be willing to invest. An alternative solution is to use an algorithm which is able to determine optimal (or close to optimal) values for the corresponding parameters.

The main contribution of this paper is to present and evaluate a new algorithm to select the number of aggregators and the cycle buffer size for collective I/O operations, specifically for the two-phase I/O algorithm and its derivatives, e.g. dynamic segmentation and static segmentation. Throughout the remainder of this paper, we refer to this algorithm as the auto selection algorithm.

The goals of the algorithm is two fold: first, determine the optimal number of aggregators such that each aggregator is saturating the file system from the processes perspective, but not overloading the aggregator. Second, use the process topology and/or file view to determine optimal grouping of processes for the collective write operation. The main steps of the algorithm are as follows:

1) Determine the minimum data size k that a process can write to the underlying file system that saturates the write bandwidth achieved.
2) Determine initial number of aggregators taking file view and/or process topology into account.
3) Refine the number of aggregators required by a collective write operation as a function of the overall amount of data written in the call and the data size k determined in step one.

In the following, we discuss each of the three steps separately.

A. Per process write saturation point

The first step of the algorithm is to determine the minimum data size k that leads to write saturation of an individual process. Further increasing the size of the data over k will typically not further increase the bandwidth obtained by a single process. Having less data will however underutilize the resources available to a process. Two questions are relevant within this context: (i) what causes a process to reach a write saturation point, and (ii) how can the saturation point be determined?

The answer to the first question in the most generic sense is, that the most limiting factor in the combination of compute nodes, I/O network, and parallel file system will determine the value of k. A detailed analysis of the different components contributing towards the performance of a parallel file system has been presented in [9]. A likely, but not the only source for bandwidth saturation is the network interconnect used to connect compute nodes and I/O nodes. A process will not be able to move more data out of a compute node than the maximum capability of the network. For Gigabit Ethernet for example - a common choice for I/O networks on a large number of small and medium size clusters - this would limit the maximum achievable per-process bandwidth to 1GBit/s. Various other characteristics of the hardware and/or software can also contribute however, e.g. the amount of data that can be cached by the OS, or an upper limit in the amount of internal buffer that a user/application can provide for collective I/O operations.

To determine the minimum data length k that leads to write saturation, we use a benchmark which uses sequential write operations with increasing data length. Each data length is written a large number of times in order to avoid caching effects, and the average value over all executions for the same message length is determined. Plotting the bandwidth values obtained using these data points over the message length shows the minimum data length k where write saturation can be observed. To minimize locking contention we round the obtained value to the next multiple of the stripe size or locking boundary of the file system.

This is the only step that is not actually automatic. However this needs to be done once on every platform, since it is not application specific. The value of k can be specified by the system administrator when configuring Open MPI or at runtime using an Open MPI runtime MCA parameter in a system wide configuration file.

B. Initial Number of Aggregators

Determining the number of aggregators to be used is a two-step process in this algorithm. The first step identifies characteristics of the application that can be exploited for parallel I/O. Most typically, this involves analyzing the file view registered by the application. Analyzing the file view across all processes is in the most generic case very complex. However, a number of common patterns that occur in real world applications such as 2-D or 3-D data distributions can be identified with reasonable efforts. Our approach currently handles only 2-D distributions in file views, by flattening the *filetype* and checking the starting offsets of all the processes. If a pattern is detected for a certain number of processes, they are grouped together initially.

Consider for example a 2-D distribution of data across 16 processes, as shown in figure 3 part (a). Assuming that each process works on an according portion of a two dimensional matrix, always four processes organized in a row of the virtual topology have data which leads to contiguous chunks that can be written/read from the file. We assign initially one aggregator process per row in the virtual 2-D process topology.

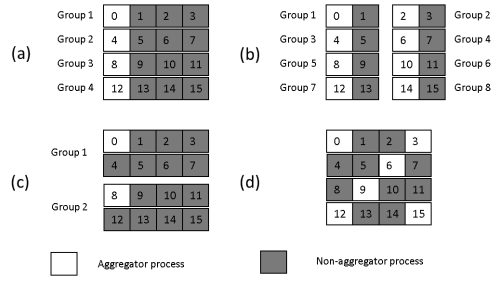

Fig. 3. (a) Initial grouping of processes in a 2-D process topology; (b) splitting the process groups based on the optimal buffer size k ; (c) merging multiple rows based on the optimal buffer size k; (d)Potential assignment of aggregators if ignoring process topology or file view

While the approach discussed so far takes advantage of the fact that the application has registered its access plan through the file view, not all applications use a file view. In fact, a large number of MPI I/O applications avoid setting a file view due to the fact that dealing with the according derived datatype(s) is often considered difficult and error prone. An alternative approach used by a number of applications is to stick with the default file view – which gives every process access to the entire file – and use explicit offset operations in the subsequent I/O calls. Since we consider extracting the application level topology information however essential for the performance of collective I/O operations, we investigate two extensions currently not covered by the MPI standard.

The first approach requires the usage of topology communicators when opening a file. Our work focuses currently on cartesian topologies. A cartesian process topology describes an n-dimensional topology with the corresponding neighborhood relations between the processes. In many scenarios, this is an equivalence to the access pattern of the processes in the file and can be often used as a lightweight version of the file view. In this model, the individual regions of access by a process are not known, however the logical organization of how data will be accessed by the process can be deducted from the process topology. Our first extension allows determining the initial number of aggregators based on the process topology provided by a cartesian communicator.

Based on the same idea, the user could also provide the same topology information using newly defined MPI hints. The advantage of this second approach is that setting hints through the `MPI_Info` object is cheaper than creating cartesian communicators, due to the collective communication operations required to create a new communicator. Note, that this second approach is currently not implemented in OMPIO.

C. Refining the Number of Aggregators

After determining the initial number of aggregators and the initial grouping of processes by analyzing the file view and/or the process topology, we check how much data each group is writing in total and compare it against the value k determined in step 1 of the algorithm. If the total bytes written in the group is greater than k, the group is split depending on the ratio of k to the total data written by one process, see fig 3 part (b) for an example. Otherwise, two or more groups can be combined together to form a single group such that the amount of data written by an aggregator is as close to k as possible. This is also shown in part (c) of fig. 3. Finally, we calculate for each process the list of the ranks of the processes that belong to its group using the new grouping strategy that was formed. Algorithm 1 gives the details on how this is achieved. Note, in the algorithm, `RootOffset` is the offset or stride where each aggregator is located in the communicator and `i` is the ratio of k to the total bytes written in a group.

In the case where a topology is not detected, we set initially the number of aggregators to be the total number of processes, which means that the topology would be one process per group. Then we refine the number of aggregators using the algorithm explained above.

As a result of the algorithm discussed in this section, a single application will use different number of aggregators depending on the amount of data written in a single collective write operation. Note, that the current implementation assumes a mostly uniform distribution of data across processes. However, extending this algorithm to non-uniform distributions is straight forward and is omitted here for the sake of clarity.

V. Evaluation

For testing the efficiency and validity of the new algorithm presented in the previous section, we used several benchmarks over two platforms. The *Shark* cluster at The University of Houston consists all-in-all of 29 nodes, 24 of them having a 2.2 GHz dual core AMD Opteron processor and 2 GB main memory, and 5 nodes having two 2.2 GHz quad core AMD Opteron processors (8 cores total) with 8 GB main memory. The nodes are connected through a 96 port 4x SDR InfiniBand switch and a 48 port Gigabit Ethernet (GE) switch. The parallel file system used is PVFS2 (v2.8.2) consisting of

Algorithm 1 Refine Number of Aggregators

Require: Initial Topology including the coordinates for the current process.

```
if K < TotalBytesWrittenInGroup then
    {need to break current group into smaller groups}
    i ← 1
    RootOffset ← ceil(K/TotalBytesWrittenByProcess)
    if NoProcsInOldGroup/RootOffset ≠ coords[1]/RootOffset then
        NoProcsInNewGroup ← RootOffset
    else
        NoProcsInNewGroup ← NoProcsInOldGroup%RootOffset
    end if
else
    if K > TotalBytesWrittenInGroup then
        {need to expand and merge with other groups}
        i ← ceil(K/TotalBytesWrittenInGroup)
        RootOffset ← NoProcsInOldGroup * i;
        i ← RootOffset;
        if TotalNumberOfProcesses/RootOffset ≠ rank/RootOffset then
            NoProcsInNewGroup ← RootOffset
        else
            NoProcsInNewGroup ← TotalNoProcs%RootOffset
        end if
    end if
end if
{Calculate for each process the list of ranks of the processes that belong in its group}
for j=0 to TotalNumberOfProcesses do
    {Determine coordinates for process j}
    {check if j and the current process belong to the same group in the initial grouping}
    if Coords_J[0]/i = Coords[0]/i then
        if Coords_J[1]/RootOffset * RootOffset = Coords[1]/RootOffset * RootOffset then
            {add j to the processes in this group}
        end if
    end if
end for
```

22 server nodes where each server uses its local disk space as the back-end storage. The stripe size of the file system is 64 KB. The file system uses GE as the network interconnect.

The Deimos PC Farm at TU Dresden has 724 compute nodes with 1, 2, and 4 dual-core AMD Opteron processors and over 2576 total cores. Each node has 2 GB of main memory. A Lustre file system (v1.6.7.2) with 38 TByte of storage is exported by 11 I/O servers via a separate 4x SDR Infiniband network. The file system is organized in 48 Object Storage Targets (OSTs) with a stripe size of 1 MB. The MPI Library used on both clusters is Open MPI trunk (r24428).

The algorithm introduced in section IV is evaluated for three collective I/O algorithms, namely two-phase I/O, dynamic segmentation, and static segmentation. The latter two algorithms utilized the information provided by the file view and/or process topology also for grouping of processes, while the two-phase I/O algorithm relies on the number of aggregators provided by that approach. In order to evaluate the efficiency of our algorithm using the automated selection of number of aggregators, we ran the exact same benchmarks also with a number of fixed aggregators for the according algorithms. However, due to the number of processes used for the test cases it is not possible to compare the performance of the automated aggregator selection algorithm vs. the full parameter space w.r.t. the number of aggregators.

A. Determining minimal saturation points

The first step consists of determining the minimum amount of data to saturate the write bandwidth of an individual process. As discussed in section IV, the value of k is determined by running a simple POSIX I/O benchmark that writes data to disk with an increasing data size. The results obtained show as expected a great difference in the performance that can be achieved over different platforms. Fig. 4 and fig. 5 show the bandwidth over the data length achieved for the Shark cluster and for Deimos PC Farm respectively. The value for k is determined as the first message length which does not yield a (significant) bandwidth increase. Based on fig. 4 the value for k is set to 32 MBs for Shark. At this point, the maximum bandwidth achieved reaches over 100MB/s, which is close to the maximum sustainable bandwidth of a Gigabit Ethernet link, which is used to mount the PVFS2 file system. It is therefore safe to assume that on Shark cluster the individual write bandwidth of a process is limited by the network link.

On the Deimos cluster, the maximum write bandwidth is achieved starting from a message length of 128 MBs. The according bandwidth value of around 285MB/s is not yet saturating the 4xInfiniBand link that is used to mount the Lustre file system onto the compute nodes. However, further increasing the message length in the test did not yield a better write bandwidth. This is mostly due to the fact that the sequential write tests require allocating a buffer of the size of a write operation. Allocating a buffer of more than 1 GB for the test introduced caused problems on a node having only 2 GBs of main memory and led to performance degradation due to excessive page swapping. Thus, the write bandwidth on this cluster is limited by the amount of memory available on the node, which is a valid concern also for parallel applications, limiting the ability of an aggregator to service other processes.

B. Tile I/O

MPI-TILE-IO [13] is a test application that implements tile access to a two dimensional dense dataset. This type of workload is seen in tiled displays (for small numbers of tiles) and in some numerical applications. The benchmark creates a cartesian communicator to access the file in a two dimensional manner, which makes it a good benchmark to test our topology

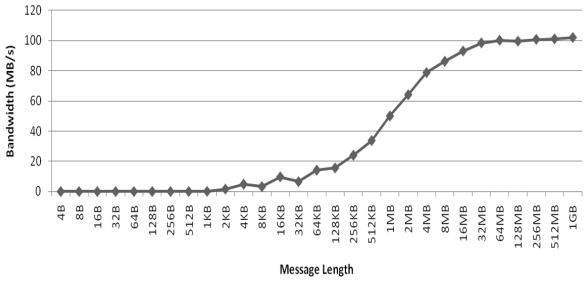

Fig. 4. Tuning for the minimum saturation point k on Shark.

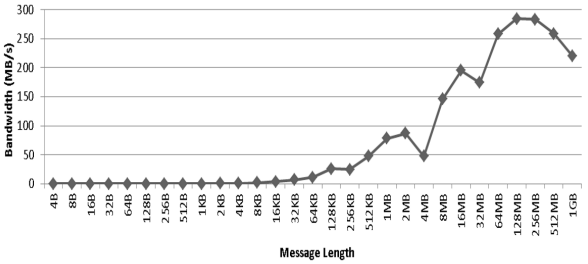

Fig. 5. Tuning for the minimum saturation point k on Deimos.

aware algorithm. The parameters that control the file access and 2D distribution of the processes are:

- *nr_tiles_x*: number of tiles in the X dimension (rows)
- *nr_tiles_y*: number of tiles in the Y dimension (columns)
- *sz_tile_x*: number of elements in the X dimension of a tile
- *sz_tile_y*: number of elements in the Y dimension of a tile
- *sz_element*: size of an element in bytes

In our measurements, the number of tiles is set according to the process count. We use three different tile sizes (64 B, 1 KB, 1 MB) and respectively three different number of elements in the X and Y dimension (2048x1600, 512x400, 20x15) to represent different access patterns in the file. For all our measurements for this test application and the other benchmarks, each test case is executed five times taking the maximum achievable bandwidth across those runs. The results obtained on Shark were very consistent, taking into consideration it is a dedicated cluster and dedicated file system. On the other hand, the result on Deimos were not always identical across different runs, considering the different allocation of nodes for different jobs and the fact that the file system underneath is a shared resource between all users. Note however, that in the vast majority of the test cases the tendencies across different executions were the same, with differences being only in the absolute values obtained. We chose to display the maximum achievable performance across the runs, as it is the execution which most likely had minimal disturbance from the outside, which in the end would be required to fairly compare the algorithms.

On the Shark cluster, tests have been executed using 20, 40 and 81 processes using Tile I/O. The results obtained for 81 processes are shown in fig. 6 for all three collective I/O algorithms used namely dynamic segmentation (left), static segmentation (center) and two-phase I/O (right). The graph shows the bandwidth achieved with each algorithm. The x-axis shows the data points obtained with the according number of aggregators. The data point labeled as *auto* is the performance obtained with our auto selection algorithm. The results indicate that the auto selection algorithm provide the best or close to the best performance for all tile sizes and all three collective I/O algorithms on this platform. The number of aggregators chosen is equal to the total number of processes, because each process is requesting to write data larger than the value of k per function call.

Similarly, on Deimos with the Lustre file system, using all processes as aggregators for this application turns out to provide the best choice for the number of aggregators as shown in fig. 7 for 144 processes for large tiles. That is also what the auto selection algorithm chooses. In the case of small tile sizes the auto selection algorithm was not giving the best result using dynamic and static segmentation, however it was giving the optimal result using the two-phase algorithm, so we argue here that in this scenario the decision to use all processes as aggregators is correct. The static and dynamic segmentation algorithms are just a poor match to the test case due to the fact that they will not re-organize data across processes at all: each process is just member of its own group.

C. Latency I/O

Latency-IO is a micro-benchmark developed as part of the latency test suite [7]. Each process has a given amount of data that it writes to the file in several segments. The size of the segment is a configurable parameter. All processes write the segment size to the first portion of the file and then on the second portion till the entire data is written. All this is within one collective I/O call per process. The test executes the collective I/O operations several times to non-overlapping regions in the file. For our measurements, we used two segment sizes, namely 64 KBs and 2 MBs. The total data written to disk per function call is 20 MBs, resulting in overall files of 3-5 GB depending on the test case. This benchmark represents the scenario where file view and process topology could not be used to optimize the grouping of processes for dynamic and static segmentation.

Tests on the Shark cluster with latency I/O have been executed for 20, 40 and 64 processes. The results show a similar trend as the tile-io results. Fig. 8 shows the results achieved for a 64 process test case. The auto selection algorithm chooses half the process count (32 processes) as the number of aggregators, which achieved in most cases close to the best performance.

We executed the latency benchmark with 64 processes on Deimos and the results are shown in fig. 9. The auto selection algorithm chooses eight aggregators. This choice was good for the small segment size. For the larger segment size, choosing only eight aggregators was not efficient and the performance penalty is highlighted mostly when using the dynamic and

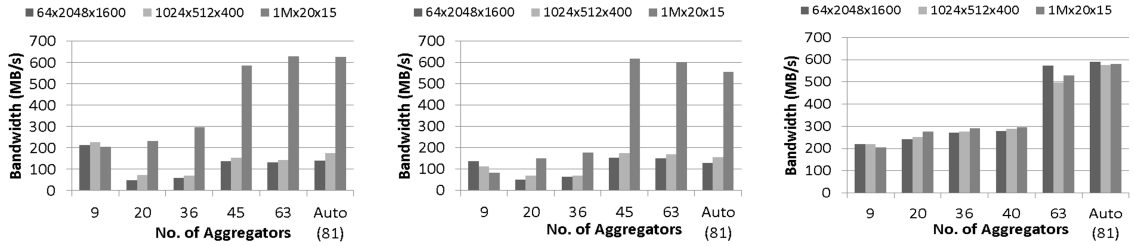

Fig. 6. mpi-tile-io with 81(9x9) processes on Shark using dynamic segmentation (left), static segmentation (middle) and two-phase I/O (right).

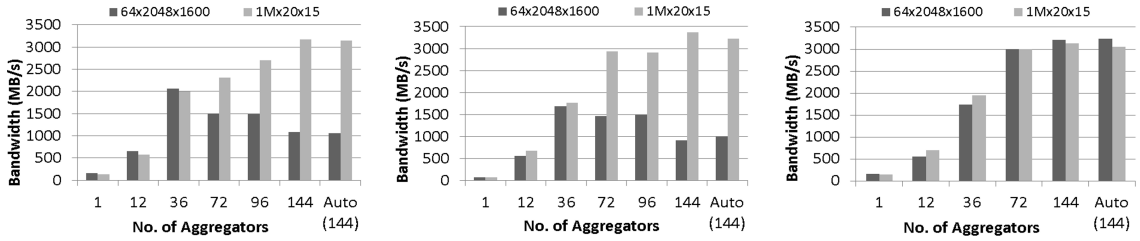

Fig. 7. mpi-tile-io with 144(12x12) processes on Deimos using dynamic segmentation (left), static segmentation (middle) and two-phase I/O (right).

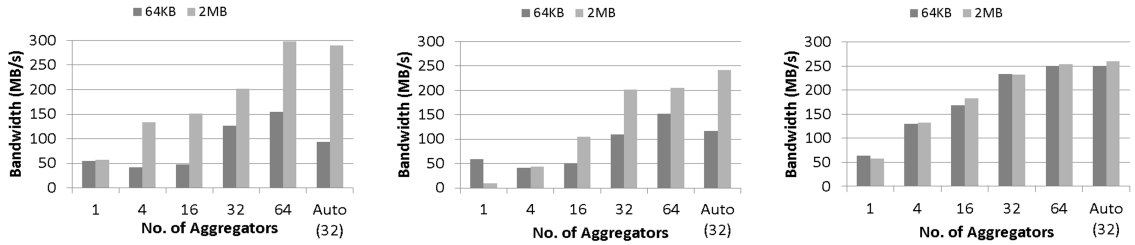

Fig. 8. latency-io with 64 processes on Shark using dynamic segmentation (left), static segmentation (middle) and two-phase I/O (right)

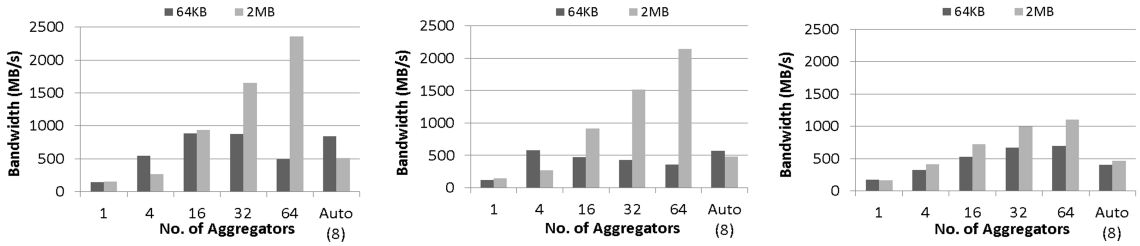

Fig. 9. latency-io with 64 processes on Deimos using dynamic segmentation (left), static segmentation (middle) and two-phase I/O (right)

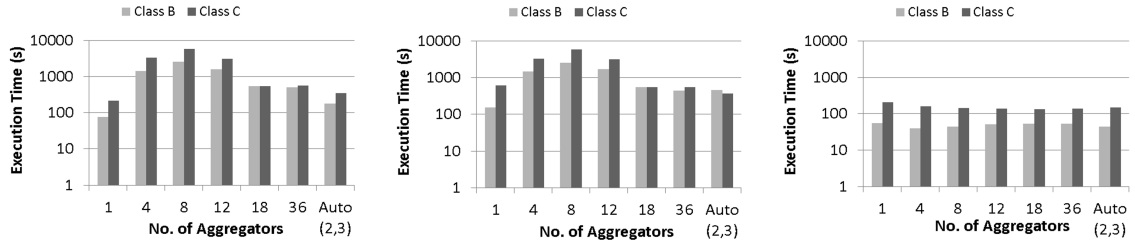

Fig. 10. BT I/O benchmark with 36 processes on Shark using dynamic segmentation (left), static segmentation (middle) and two-phase I/O (right).

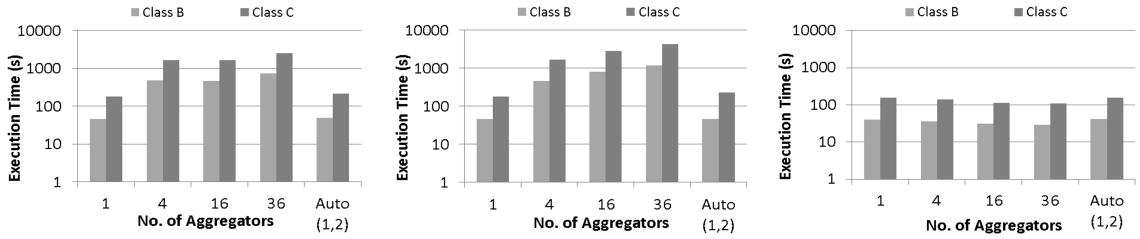

Fig. 11. BT I/O benchmark with 36 processes on Deimos using dynamic segmentation (left), static segmentation (middle) and two-phase I/O (right).

static segmentation algorithms, where the results achieved with 64 aggregators were much better. The reason for this can be attributed to the communication cost between the processes that did not hide the cost of writing large segments to disk by each process. In addition to that, since the stripe size on the file system is 1 MB, writing contiguous data portions in a multiple of two of the stripe size (2 MB) by all processes did not incur a performance penalty since the write requests were aligned with the stripe boundaries.

D. BT I/O

The Block-Tridiagonal (BT) NPB [17] benchmark has an MPI implementation that employs a multi-partitioning domain decomposition. Each process is responsible for multiple cartesian subsets of the entire data set, whose number increases as the square root of the number of processors participating in the computation. The benchmark does computation as well as I/O operations. Each measurement is executed five times, taking the best execution time between those runs. Since the execution time reported by the benchmark contains compute, read, and write operations, it is not possible to conclusively determine the write bandwidth obtained. Instead, we report the execution time of the entire benchmark.

The BT I/O measurements on Shark with 36 processes show almost identical behavior for all three classes in terms of how each algorithm performs and how the auto selection algorithm performs in comparison. We show the results for the classes B and C. The auto selection algorithm chooses two aggregators for the class B and three aggregators for class C. The choice turns out to be very suited for this benchmark, and is different from the previous benchmarks where a larger number of aggregators was giving better performance. As shown in the left part of fig. 10, using static or dynamic segmentation with a large number of aggregators would have been very inefficient. However, with the topology detection and automatic aggregator selection, the execution time obtained was acceptable.

The results for the same test case with 36 processes on Deimos is shown in the right part of fig. 11. The performance obtained was very similar to the results on Shark. One aggregator was chosen for the class B and two aggregators for class C with the auto selection algorithm. The 144 process test case on the same cluster (not shown here due to space limitations) used the same number of aggregators for the according test cases as in the 36 process test cases, which highlights a property of the algorithm, namely that the number of processes used by the application does not influence the number of aggregators for a constant amount of data to be written. In this case, using only two aggregators was a good choice for two-phase since the performance was close to optimal, but not very efficient for static and dynamic segmentation, mainly because one group for the aggregators was very large and the other one very small. The generation of highly uneven groups as a result of the algorithm is an aspect of further research.

E. Discussion

The results provided in this section show a small portion of the total number of tests performed. Out of a total of 134 test cases, 88 test cases using the auto selection algorithm achieved either the best performance that we could observe with any number of aggregators, or were within 10% of the best performance. 110 test cases were within the 25% of the best result. Note that the range between best and worst performance is often extremely wide, which makes a performance within 25% of the optimum a reasonable result considering that no user interaction is required to achieve that.

For each benchmark, we also measured the performance of the two-phase algorithm using ROMIO, which uses a different strategy to determine the number of aggregator processes, namely one aggregator per node. We compare those results to the ones obtained by the two-phase algorithm within OMPIO using the auto selection algorithm. The comparison show that in 29 out of 45 test cases which used the two-phase I/O algorithm, our approach leads to a better performance with an average performance advantage of 41% over the one aggregator per node strategy.

In most test cases in which our algorithm detected a regular access pattern and/or topology in the application, this translated to a significant performance improvement. In the tile I/O test cases, where the 2-D pattern was recognized, **all** test cases improved the performance compared to the one aggregator per node strategy; not a single test case showed any problems. Similarly good results were obtained for BT I/O, with few exceptions. However, as BT I/O resembles an application benchmark, it is difficult to separate the write performance from the performance of read operations (since the test uses both), with our read algorithm using a non-optimized version.

To demonstrate the correctness of the automatic selection algorithm, further tests have been executed with a parallel image processing application which is used to analyze smear sample from fine needle aspiration cytology, with the overall goal being to assist medical doctors in identifying cancer cells [8]. The MPI version of the code has the option to write the texture data into output files to facilitate future processing steps in realizing a complete computer aided diagnosis (CAD) solution. This makes the application compute and I/O intensive. We executed the application with the three algorithms discussed earlier (dynamic, static, and two-phase) on the Shark cluster with the PVFS2 filesystem for 64 processes. The results show that the I/O performance obtained using the auto selection algorithm was within 10% of the optimal result.

VI. Conclusions

In this paper we introduced an algorithm to dynamically adjust the number of aggregators used in collective I/O operations. Our algorithm takes the file view, process topology, individual characteristic of file system, as well as the amount of data written in a collective I/O operation into account. Preliminary results with three different benchmarks on two different platforms demonstrated the effectiveness of our algorithm, and documented that our approach improves the performance for applications using collective I/O operations 'out-of-the-box' compared to a default setting for the number of aggregators, eliminating the necessity to pre-tune the collective I/O operation for each test case.

Although the initial results are very promising, more work is required to evaluate the algorithm. Specifically, we would like to extend our evaluation to larger number of processors, real world applications and to include read-operations as well. In addition, we also plan on further improving this algorithm to avoid uneven process distributions in some scenarios. A complimentary aspect of this research furthermore deals with algorithm selection, i.e. when to use two-phase I/O vs. dynamic or static segmentation.

Acknowledgements

We would like to thank the Center for Information Services and High Performance Computing at the Technical University of Dresden, Germany for giving us access to the Deimos cluster and their file system to perform our tests and measurements.

References

[1] Javier García Blas, Florin Isaila, David E. Singh, and J. Carretero. View-Based Collective I/O for MPI-IO. In *CCGRID '08: Proceedings of the 2008 Eighth IEEE International Symposium on Cluster Computing and the Grid (CCGRID)*, pages 409–416, Washington, DC, USA, 2008. IEEE Computer Society.

[2] Mohamad Chaarawi, Suneet Chandok, and Edgar 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] Mohamad Chaarawi, Edgar Gabriel, Rainer Keller, Richard L. Graham, George Bosilca, and Jack J. Dongarra. OMPIO: A Modular Software Architecture for MPI I/O. In *accepted for publication at the EuroMPI Conference*, Santorini, Greece, September 2011.

[4] Juan Miguel del Rosario, Rajesh Bordawekar, and Alok Choudhary. Improved parallel i/o via a two-phase run-time access strategy. *SIGARCH Comput. Archit. News*, 21(5):31–38, 1993.

[5] Phillip M. Dickens and Jeremy Logan. Y-lib: a user level library to increase the performance of mpi-io in a lustre file system environment. In *HPDC '09: Proceedings of the 18th ACM international symposium on High performance distributed computing*, pages 31–38, New York, NY, USA, 2009. ACM.

[6] Edgar Gabriel, Graham E. Fagg, George Bosilca, Thara Angskun, Jack J. Dongarra, Jeffrey M. Squyres, Vishal Sahay, Prabhanjan Kambadur, Brian Barrett, Andrew Lumsdaine, Ralph H. Castain, David J. Daniel, Richard L. Graham, and Timothy 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, September 2004.

[7] Edgar Gabriel, Graham E. Fagg, and Jack J. Dongarra. Evaluating dynamic communicators and one-sided operations for current MPI libraries. *International Journal of High Performance Computing Applications*, 19(1):67–79, 2005.

[8] Edgar Gabriel, Vishwanath Venkatesan, and Shishir Shah. Towards High Performance Cell Segmentation in Multispectral Fine Needle Aspiration Cytology of Thyroid Lesions (accepted for publication). *Computational Methods and Programs in Biomedicine*, page t.b.d., 2009.

[9] Samuel Lang, Philip Carns, Robert Latham, Robert Ross, Kevin Harms, and William Allcock. I/O Performance Challenges at Leadership Scale. In *Proceedings of the Supercomputing Conference*, 2009.

[10] Wei-keng Liao and Alok Choudhary. Dynamically Adapting File Domain Partioning Methods for Collective I/O Based on Underlying Parallel File System Locking Protocols. In *Proceedings of the Supercomputing Conference*, 2008.

[11] Lustre webpage. *http://www.lustre.org*.

[12] Message Passing Interface Forum. *MPI-2.2: Extensions to the Message Passing Interface*, September 2009. `http://www.mpi-forum.org`.

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

[14] Frank Schmuck and Roger Haskin. GPFS: A shared-disk file system for large computing clusters. In *In Proceedings of the 2002 Conference on File and Storage Technologies (FAST)*, pages 231–244, 2002.

[15] Rajeev Thakur, William Gropp, and Ewing Lusk. A case for using mpi's derived datatypes to improve i/o performance. In *Supercomputing '98: Proceedings of the 1998 ACM/IEEE conference on Supercomputing (CDROM)*, pages 1–10, Washington, DC, USA, 1998. IEEE Computer Society.

[16] Rajeev Thakur, William Gropp, and Ewing Lusk. On implementing MPI-IO portably and with high performance. In *Proceedings of the sixth workshop on I/O in parallel and distributed systems*, pages 23–32, 1999.

[17] P. Wong and R. F. Van der Wijngaart. *NAS Parallel Benchmarks I/O Version 3.0*. Technical Report NAS-03-002, Computer Sciences Corporation, NASA Advanced Supercomputing (NAS) Division.

[18] Joachim Worringen. Self-adaptive Hints for Collective I/O. In Bernd Mohr, Jesper Trff, Joachim Worringen, and Jack Dongarra, editors, *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, Lecture Notes in Computer Science, pages 202–211. Springer Berlin / Heidelberg, 2006.

[19] Weikuan Yu, Jeffrey Vetter, R. Shane Canon, and Song Jiang. Exploiting Lustre File Joining for Effective Collective IO. 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.

[20] Xuechen Zhang, Song Jiang, and Kei Davis. Making resonance a common case: A high-performance implementation of collective i/o on parallel file systems. In *IPDPS '09: Proceedings of the 2009 IEEE International Symposium on Parallel&Distributed Processing*, pages 1–12, Washington, DC, USA, 2009. IEEE Computer Society.