MPI-IO/GPFS, an Optimized Implementation of MPI-IO on top of GPFS

Jean-Pierre Prost[†], **Richard Treumann**[‡]
Richard Hedges[§], **Bin Jia**[‡], **Alice Koniges**[§]
[†] IBM T.J. Watson Research Center, Route 134,
Yorktown Heights, NY 10598

[‡] IBM Poughkeepsie, 2455 South Road,
Poughkeepsie, NY 12601

[§] Lawrence Livermore National Laboratory, 7000 East Avenue,
Livermore, CA 94550

Abstract

MPI-IO/GPFS is an optimized prototype implementation of the I/O chapter of the Message Passing Interface (MPI) 2 standard. It uses the IBM General Parallel File System (GPFS) Release 3 as the underlying file system. This paper describes optimization features of the prototype that take advantage of new GPFS programming interfaces. It also details how collective data access operations have been optimized by minimizing the number of messages exchanged in sparse accesses and by increasing the overlap of communication with file access. Experimental results show a performance gain. A study of the impact of varying the number of tasks running on the same node is also presented.

Keywords: MPI-IO, GPFS, file hints, prefetching, data shipping, double buffering, performance, optimization, benchmark, SMP node.

SC2001 November 2001, Denver (c) 2001 ACM 1-58113-293-X/01/0011 $5.00

1 Introduction

The Message Passing Interface 2 standard [Mpif-97] defines an application programming interface for portable and efficient parallel I/O. IBM has been working on both prototype and product implementations of MPI-IO for the IBM SP system, using the IBM General Parallel File System [GPFS-00] as the underlying file system.

The use of GPFS as the underlying file system provides maximum performance by tight interaction between MPI-IO and GPFS. GPFS is a high performance file system that presents users a global view of files from any client node. It provides parallel access to the data residing on server nodes through the Virtual Shared Disk interface. New hints and directives for improving GPFS performance were introduced in GPFS Release 1.3 and are now exploited by MPI-IO/GPFS.

This paper describes new optimizations which have been implemented in the IBM MPI-IO prototype, referred to as MPI-IO/GPFS[1].

MPI-IO/GPFS data shipping mode, introduced in [Pros-00], takes advantage of GPFS data shipping mode by defining at file open time a GPFS data shipping map that matches MPI-IO file partitioning onto I/O agents. A heuristic allows MPI-IO/GPFS to optimize GPFS block prefetching through the use of GPFS multiple access range (MAR) hints.

MPI-IO/GPFS has special emphasis on optimizing collective data access operations. A new file hint allows the user to specify whether file access is sparse. For sparse access, a new algorithm has been developed to minimize the number of messages exchanged between MPI tasks and I/O agents. Double buffering at the I/O agents allows an increase in the overlap of data shipping with file access operations.

Measurements, based on synthetic benchmarks, were performed on an ASCI testbed system at Lawrence Livermore National Laboratory (LLNL) in order to study the performance gain brought by these optimizations. Through this experimentation, we also studied the performance degradation induced by running several tasks versus only one task per node. While the benchmarks were designed to illustrate the benefit of each optimization, their general characteristics do allow us to draw conclusions for real applications exhibiting similar I/O patterns.

This paper differs from other performance studies performed by Thakur et al. [Tha1-99, Tha2-99] in that these performance gains are achieved through user-defined hints and implementation-defined directives, to communicate the anticipated application I/O pattern(s) to the underlying file system which has the capability to exploit this additional knowledge.

The paper is organized as follows. Section 2 details the new optimizations implemented in the MPI-IO/GPFS prototype. Section 3 presents our experimentation. Section 4 interprets the results. And Section 5 concludes the paper and presents future research directions.

[1] IBM product implementation work draws on the knowledge gained in this prototype project but features of the prototype discussed in this paper and features of eventual IBM products may differ.

2 MPI-IO/GPFS Optimization Features

This section introduces the design foundation of MPI-IO/GPFS, referred to as data shipping, and describes the optimizations recently implemented in the MPI-IO/GPFS prototype.

2.1 Data Shipping

To prevent concurrent access of GPFS file blocks by multiple tasks, possibly residing on separate nodes, MPI-IO/GPFS uses data shipping. This technique, introduced in [Pros-00], binds each GPFS file block to a single I/O agent, which will be responsible for all accesses to this block. For write operations, each task distributes the data to be written to the I/O agents according to the binding scheme of GPFS blocks to I/O agents. I/O agents in turn issue the write calls to GPFS. For reads, the I/O agents read the file first, and ship the data read to the appropriate tasks. The binding scheme implemented by MPI-IO/GPFS consists in assigning the GPFS blocks to a set of I/O agents according to a round-robin striping scheme, illustrated in Figure 1.

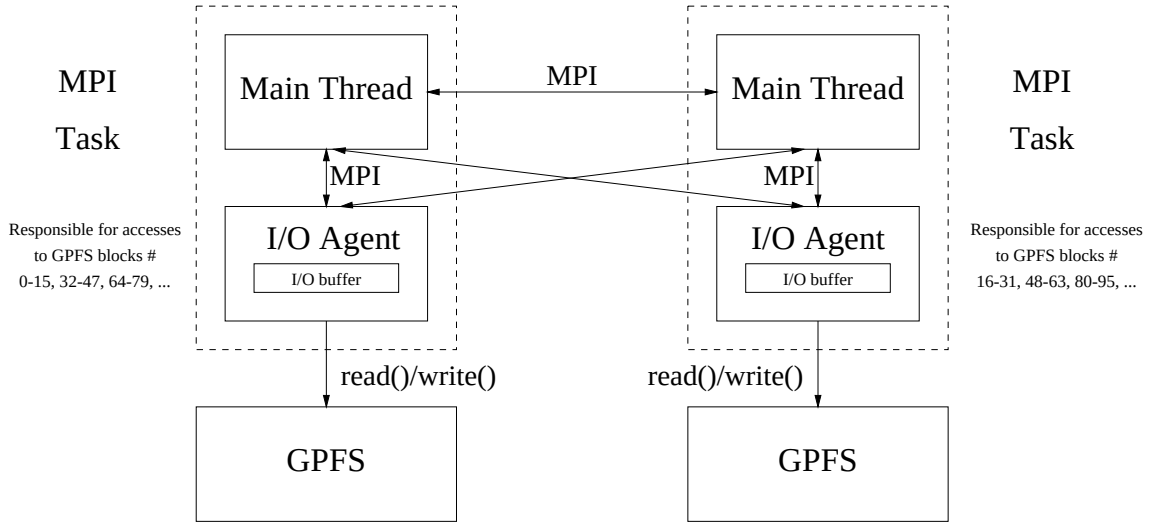

Figure 1: GPFS block allocation used by MPI-IO/GPFS in data shipping mode (using the default stripe size)

I/O agents are multi-threaded and are also responsible for combining data access requests issued by all participating tasks in collective data access operations. By default, there is one I/O agent per computing task. However, the user may restrict the allocation of I/O agents to specific nodes if desired.

2.2 GPFS Data Shipping

GPFS data shipping (DS), introduced in GPFS Release 1.3 [GPFS-00], is well matched to MPI-IO data shipping. When the user application opens the file with MPI-IO data shipping mode enabled, MPI-IO/GPFS sets up GPFS DS parameters so they match the partitioning

of the file onto the I/O agents. When MPI-IO data shipping and GPFS DS work together, the application benefits from the strengths of both features. MPI-IO can most efficiently ship the data from the requesting nodes to the GPFS nodes responsible for the data accesses and GPFS can run more efficiently by always issuing larger I/O requests and using shared locking.

When starting GPFS DS, MPI-IO/GPFS sets the GPFS DS map which matches the MPI-IO binding scheme of GPFS file blocks to I/O agents. There are two relevant aspects to GPFS DS. The benefit is that GPFS can forgo token management and instead grant a single shared read/write lock to all tasks that have participated in starting GPFS data shipping. The potential cost of GPFS DS mode is that if any node other than the one granted rights by the GPFS DS map tries to access a block, GPFS must ship the data. By combining MPI-IO data shipping with GPFS DS, we avoid token management costs and never ask GPFS to do any data shipping.

2.3 Controlled Prefetching of GPFS Blocks

The use of MPI_Datatypes in defining a file view means that a single MPI-IO read or write call may implicitly represent a series of file I/O operations. The full series is known to the MPI library before the first file I/O operation. MPI-IO/GPFS can exploit this "advance" knowledge by using the GPFS multiple access range hint to guide the prefetching of GPFS blocks. Each time a data access operation takes place, MPI-IO/GPFS analyzes the predefined pattern of I/O requests at each I/O agent. MPI-IO/GPFS computes for each GPFS block the lower and upper bounds of the byte range which contains all the byte ranges to be accessed within that GPFS block. This way, it builds a list of byte ranges (one per GPFS block) to be accessed.

Then, it applies the following prefetching policy:

1. At the beginning of the data access operation, MPI-IO/GPFS tries to prefetch the lesser of $2 * max\_acc$ byte ranges (where max_acc represents the maximum number of byte ranges that can be accessed at once – this number is controlled by an environment variable and allows to disable any prefetching when its value is zero) or all byte ranges the current I/O operation calls will involve.

2. GPFS informs MPI-IO/GPFS of how many byte ranges *actual_pre* it will schedule for prefetch. MPI-IO/GPFS accesses the minimum of *actual_pre* and max_acc byte ranges, or one byte range if *actual_pre* is zero.

3. Then, MPI-IO/GPFS releases the byte ranges it just accessed and tries to prefetch the next $(2 * max\_acc - actual\_pre)$ byte ranges which have not been either accessed or prefetched, or all the remaining byte ranges if their number is less than $(2 * max\_acc - actual\_pre)$.

4. MPI-IO/GPFS iterates steps 2. and 3. until all byte ranges have been accessed.

Through the max_acc value, the user can throttle the prefetching of GPFS blocks at each I/O agent so that fairness of prefetching can be increased by maintaining similar numbers of blocks prefetched on behalf of the various requesting tasks at each I/O agent.

For larger values of *max_acc* and large number of requesting tasks, an I/O agent may be overwhelmed with prefetching requests and may favor the first requesting task coming in over the other tasks.

2.4 Collective Data Access Operation Enhancements

The MPI-IO/GPFS prototype has emphasized improving collective data access operations and this led to two additional optimizations, one for sparse accesses, the other targeted at increasing the overlap of the shipping of the data between the tasks and the I/O agents with the file access operations performed by the I/O agents.

2.4.1 Sparse Access

A new file hint, named IBM_sparse_access, allows the user to declare that the I/O pattern of the accesses performed on a file is sparse. When this mode is enabled, MPI-IO/GPFS optimizes the communication and the synchronization between requesting tasks and I/O agents in order to minimize number of messages exchanged and points of synchronization. By default, the access is assumed to be dense, a complete communication graph between requesting tasks and I/O agents is used and global synchronization between all requesting tasks is performed. This may result in useless message exchange and superfluous synchronization overhead when a requesting task only needs the involvement of a subset of the I/O agents (typical of sparse access to the file). On the other hand, when the access is specified to be sparse, a first pass analyzes the requests of the tasks globally and computes the minimal communication graph between the requesting tasks and the I/O agents. During the second pass, the processing of the I/O agents to satisfy the task requests is identical to the dense case, except that the minimal communication graph is used, optimizing the synchronization between the requesting tasks and the I/O agents. Clearly, the additional computation of the minimal graph can only benefit sparse access and should be avoided for dense access.

2.4.2 Double Buffering

In a collective data access operation, the amount of data to be read/written is quite often far more than the size of the data buffers used by the I/O agents. Therefore, in order to process an entire data access operation, multiple steps are needed, each consisting of a phase of communication between requesting tasks and I/O agents and a phase of actual access to GPFS. The two phases of consecutive steps can be overlapped to reduce the overhead for combining individual requests and hide data shipping latency. Two data buffers are used for this purpose, one is for the communication, the other is for GPFS access.

Double buffering is always active. Comparing application benchmark results obtained with a library omitting this optimization with results from the standard library allows us to characterize the performance benefit of double buffering.

3 Performance Benchmarks

We used four benchmarks, each designed to evaluate the benefit from a particular optimization introduced in the MPI-IO/GPFS prototype. These benchmarks exhibit I/O patterns suited to illustrate each optimization benefit. The experimentation results are presented in Section 4.

For all benchmarks, the metric is read or write bandwidth expressed as MB/second for the job as a whole. To provide a consistent approach to measurement, in each benchmark, MPI_Barrier is called before beginning the timing, making each task's first call to the MPI read or write routine well synchronized. Similarly, to ensure that the entire read/write time is counted, each test calls MPI_Barrier before taking the end time stamp. We believe this method provides the most meaningful measure of operation time for both collective and noncollective tests.

3.1 Discontiguous Benchmark

In the discontiguous benchmark, each task reads or writes an equal number of 1024-byte blocks, where the set of blocks that any one task reads or writes is scattered randomly across the entire file. Working together, the tasks tile the entire file without gaps or overlap. Parameters control file size and type of file access (read or write).

The benchmark program reads or writes the file region by region, using a one gigabyte region size. At program start, each task is assigned a list of 1024-byte blocks to read or write to the file. At each step, the task creates an MPI_Datatype to map the blocks that belong to the current region. The MPI_Datatype is used as a filetype argument to MPI_File_set_view.

3.2 GPFS Data Shipping Benchmark

This benchmark is designed to measure the benefit of using GPFS DS mode. The benchmark loops on the data access operation. At each iteration, each task reads or writes the same small number of nonoverlapping noncontiguous data blocks from/to the file. Starting addresses of all blocks are generated randomly across the file and scattered among tasks. Only noncollective data access operations are used in this benchmark.

The size of each block and the total number of blocks are specified by two benchmark parameters, so that scope and amount of GPFS DS can be controlled precisely. Another parameter specifies whether to enable GPFS DS.

3.3 Multiple Access Range Benchmark

This benchmark is designed to measure the benefit of applying the controlled prefetching policy described in Section 2.3. GPFS blocks are randomly assigned to tasks. Only subblocks scattered throughout each GPFS block are read/written. The number of GPFS blocks to be accessed, the number of subblocks to be read/written within each GPFS block, as well as the size of a subblock are adjustable. An environment variable allows control of

the value of *max_acc*. If its value is zero, any prefetching is disabled. MPI derived datatypes are used to describe the layout of the subblocks to be accessed within each GPFS block.

3.4 Sparse Access Benchmark

This benchmark is designed to measure the benefit of using the IBM_sparse_access file hint. Each task reads or writes the same number of nonoverlapping blocks of data from/to the file. One benchmark parameter specifies whether to use the IBM_sparse_access file hint. Another parameter specifies a sparsity factor which, according to the number of participating tasks, determines how many data blocks are to be read/written. These data blocks are randomly distributed throughout the file, with the limitation that at most one data block per task be in each data shipping stripe. The data block size and the size of the I/O agent data buffer are adjustable through other benchmark parameters.

4 Performance Measurements

This section first describes the testbed system we used in our experimentation. Then, we present the experiments we carried out and detail the measurement results we obtained. Performance graphs are gathered at the end of the paper.

4.1 Testbed System

The IBM SP system used at LLNL for these tests is composed of 67 Nighthawk2 SMP nodes each consisting of 16 375 Mhz Power3 processors. There are 64 compute nodes, 2 dedicated GPFS I/O server nodes, and 1 login node.

The system is configured with double/single switch adapters and PSSP 3.3 with GPFS 1.4. The configuration has the two server nodes each serving 12 logical disks. Thus the filesystem is comprised of 24 logical disks of about 860GB each. The transfer rate of the disks as one integrated system is approximately 700 MB/sec.

4.2 Experiments

All benchmarks were run at 2, 4, 8, 16, and 32 tasks, with one task per node, except for the GPFS DS benchmark for which only up to 12 tasks were used, and for the controlled prefetching benchmark for which up to 16 tasks were used. Data is presented in graphs to facilitate comparison of the aggregated bandwidths obtained with or without each optimization described in Section 2. Three or more runs of each experiment were conducted and the averaged bandwidths are presented in the performance graphs. Because it was not possible to obtain dedicated use of the testbed or to make as many runs of the same test as we would have liked, the results do show some variability due to contention from other jobs on the system.

Finally, a study was made of the impact of using several tasks per node vs one task per node to characterize the compromise between resource utilization and file I/O performance.

For this study, a benchmark was run at 1, 2, 4, and 8 tasks per node with from 4 to 128 tasks.

4.3 Measurement Results

4.3.1 GPFS Data Shipping

The GPFS DS benchmark was run with GPFS DS mode enabled and disabled. A total of 10000 4KB blocks were written/read by each task to/from the file.

The results for write and read operations are shown in Figures 2 and 3, respectively.

For reads, it appears that there is an advantage in enabling GPFS data shipping in addition to MPI-IO data shipping. However, for writes, it seems that there is a degradation when GPFS data shipping is turned on, particularly as task counts increase. When GPFS data shipping is enabled, the adaptive buffer cache management of GPFS is modified. This modification may impact GPFS write behind policy for larger number of tasks, resulting in blocks to be written sooner to storage and limiting cache reuse. This assumption will need to be verified through low-level tracing of GPFS activity.

4.3.2 Controlled Prefetching of GPFS Blocks

The multiple access range benchmark was run with 32 4KB subblocks being accessed by each task out of 100 GPFS blocks. Various values for *max_acc* (0, 1, 2, 4, 6, 8) were measured. A value of zero for *max_acc* means that the GPFS multiple access range hint is not used and therefore no prefetching and releasing of byte ranges is performed.

The results for write and read operations are presented in Figures 4 and 5, respectively.

The first observation for writes is that there is a lot of variability in the results, and there is no clear conclusion which can be derived from the graph, except maybe that there is a very limited benefit of using controlled prefetching. On the other hand, for reads, it clearly appears that controlled prefetching is a winner, especially when *max_acc* is equal to one. We do not explain however why larger values of *max_acc* do not lead to better performance. Again, a low level tracing of GPFS I/O requests will be necessary to really understand this behavior.

4.3.3 Sparse Access in Collective Data Access Operations

The sparse access benchmark was run with and without the IBM_sparse_access hint and for various sparsity factors (100=dense, 50, 25, 10, 1). If n represents the number of tasks running the program and sf the sparsity factor, a total of $100 * sf * n$ blocks of 16 KB each were written/read by each task to/from the file.

The results for write and read operations are shown in Figures 6 and 7, respectively.

For both reads and writes, the benefit of using the sparse access hint becomes significant at 16 or more tasks and with truly sparse patterns. We expect this hint to become even more effective with a larger number of tasks. Note that as the I/O pattern becomes more

dense the potential communication savings is reduced and the overhead of calculating the minimal communication graph remains so performance is better without the hint.

4.3.4 Double Buffering in Collective Data Access Operations

The discontiguous benchmark was run with and without double buffering at the I/O agents.

The results for write and read operations are shown in Figures 8 and 9, respectively.

It is clear that double buffering brings noticeable benefit for writes and substantial benefit for reads. We suspect the performance drops seen for the 32 task reads and the 32 task double buffer write are due to other jobs load on the shared system. The overlapping of the communication between computing tasks and I/O agents with file transfer operations performed by the I/O agents brings the performance gain.

4.3.5 Varying the Number of Tasks Per Node

The discontiguous benchmark was run on 4 to 128 tasks, distributed 1, 2, 4 and 8 tasks per node.

Figures 10 and 11 represent the results for write and read operations, respectively.

For the discontiguous benchmark, allocating more tasks per node, for both reads and writes, decreases the aggregated bandwidth. The reason is simple. If we allocate more tasks per node, we end up having more I/O agents per GPFS client and the overall number of GPFS clients participating in the I/O operation decreases. Therefore, the degree of parallelism of the file transfer operations is reduced. We do observe that as task count grows and aggretate bandwidth begins to approach the capacity of the filesystem, the performance loss from running more tasks per node becomes less noticeable. A simple rule of thumb is to distribute a smaller job on as many nodes as are available when I/O performance is a major consideration. Larger jobs which can saturate the filesystem even when run at many tasks per node have little chance to gain I/O performance by being spread across more nodes.

5 Conclusion

We presented optimizations designed to improve the performance of MPI-IO running over GPFS and evaluated their impact by running I/O intensive benchmarks on a testbed system of 64 compute nodes, each having 8 processors.

More measurements and a low-level tracing of GPFS activity when GPFS data shipping is enabled are necessary in order to conclude about the benefit of using GPFS data shipping. Using controlled GPFS prefetching rather than default GPFS prefetching can be beneficial for read operations, but remains questionable for write operations. We found that for truly sparse I/O patterns and larger task counts there is significant benefit in using the sparse access hint. We also found that for relatively dense I/O patterns, the cost of computing the minimal communication graph is greater than the potential savings from reduced synchronization among tasks. Use of the sparse access hint is often not appropriate. Using double buffering at the I/O agent is a clear winner for both reads and writes.

We also showed there is a benefit in distributing computing tasks onto a larger number of nodes since this allows to increase the degree of parallelism of the file transfer operations performed by the I/O agents. However, for larger jobs which tend to saturate the file system, running several tasks per node could lead to better performance since shared memory communication among tasks on the same node could decrease the communication time of the job.

We are also working on further optimizations of the MPI-IO code to have fewer and shorter locked critical sections. Each I/O agent is multi-threaded and uses MPI communication so must share MPI state information with the user threads of the task. Reducing the length and frequency of the critical sections in the agent code is expected to provide better concurrency among the agent threads as well as reduce agent impact on user threads doing MPI operations.

6 Acknowledgements

This work was performed under the auspices of the US Department of Energy by the University of California, Lawrence Livermore National Laboratory under Contract No. W-7405-Eng-48.

References

[Mpif-97] Message Passing Interface Forum, *MPI-2: Extensions to the Message-Passing Interface*, July 18, 1997.
http://www.mpi-forum.org/docs/docs.html.

[GPFS-00] *IBM General Parallel File System for AIX: Installation and Administration Guide*, IBM Document SA22-7278-03, July 2000.
http://www.rs6000.ibm.com/resource/aix_resource/sp_books/gpfs/insta ll_admin/install_admin_v1r3/gpfs1mst.html.

[Pros-00] J.-P. Prost, R. Treumann, R. Hedges, A. Koniges, and A. White, *Towards a High-Performance Implementation of MPI-IO on top of GPFS*, Sixth International Euro-Par Conference, Springer-Verlag, Aug.–Sep. 2000, pp. 1253–1262.

[Tha1-99] R. Thakur, W. Gropp, and E. Lusk, *Data Sieving and Collective I/O in ROMIO*, Seventh Symposium on the Frontiers of Massively Parallel Computation, IEEE Computer Society Press, Feb. 1999, pp. 182–189.

[Tha2-99] R. Thakur, W. Gropp, and E. Lusk, *On Implementing MPI-IO Portably and with High Performance*, Sixth Workshop on Input/Output in Parallel and Distributed Systems, May 1999, pp. 23–32.

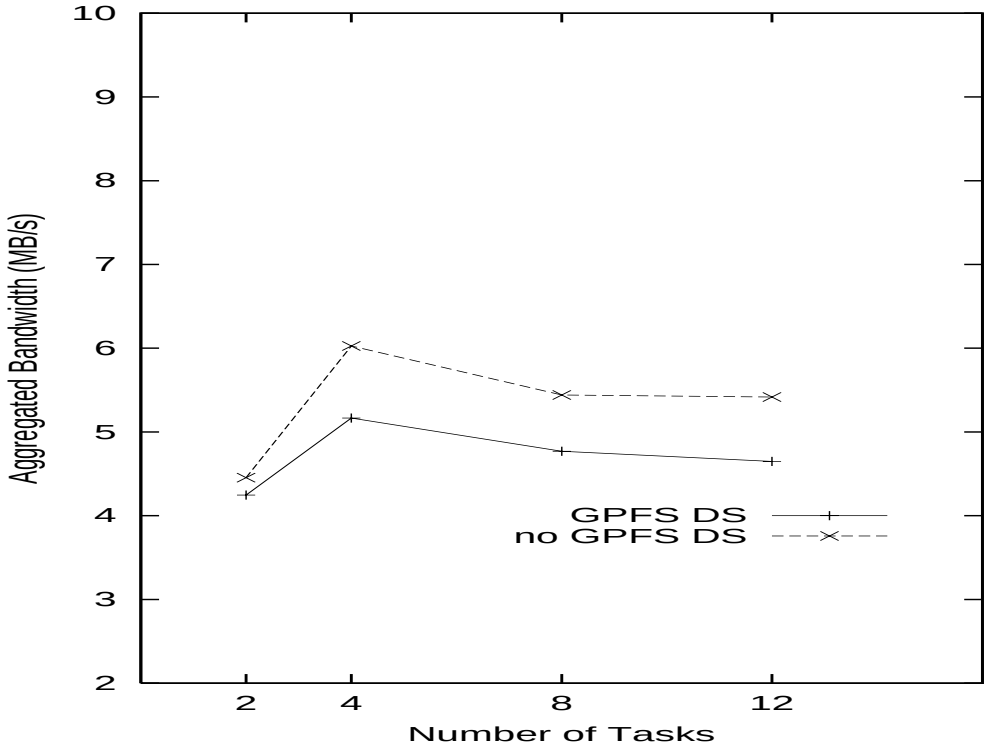

Figure 2: Averaged bandwidths measured for write operations in the GPFS DS benchmark

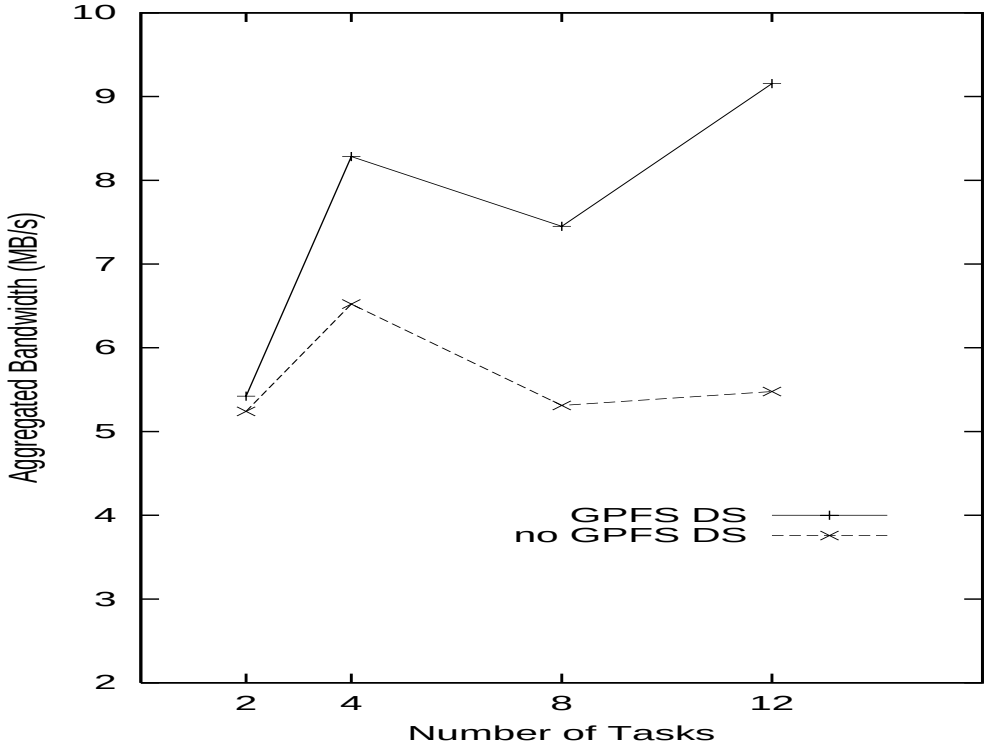

Figure 3: Averaged bandwidths measured for read operations in the GPFS DS benchmark

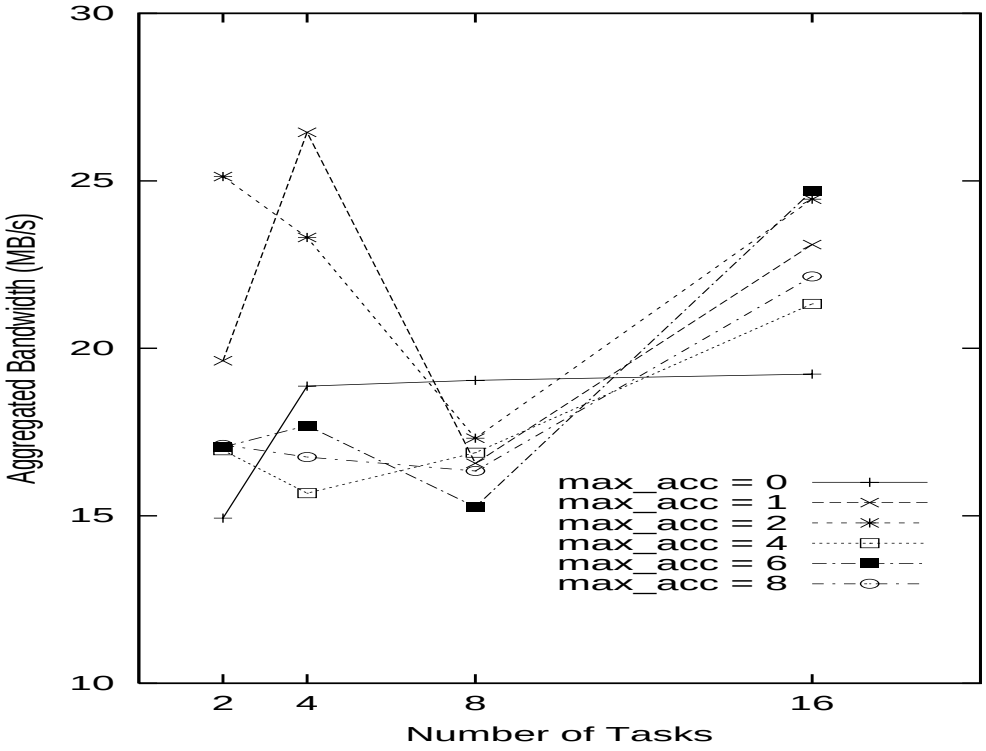

Figure 4: Averaged bandwidths measured for various values of *max_acc* when each task writes 32 4KB subblocks out of each of 100 GPFS file blocks

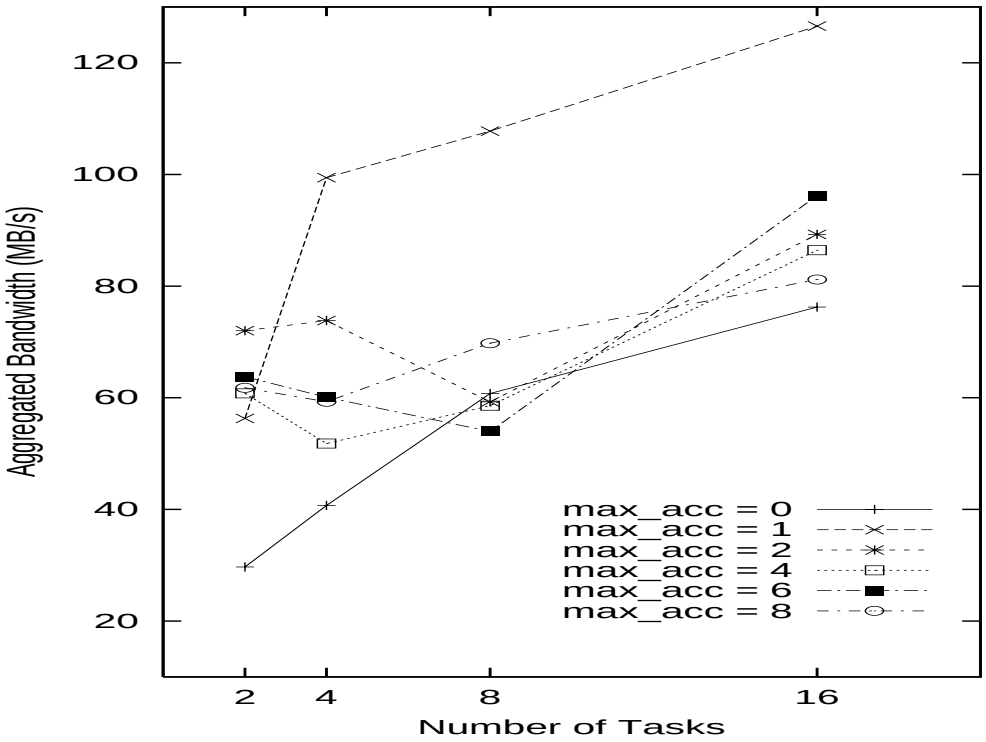

Figure 5: Averaged bandwidths measured for various values of *max_acc* when each task reads 32 4KB subblocks out of each of 100 GPFS file blocks

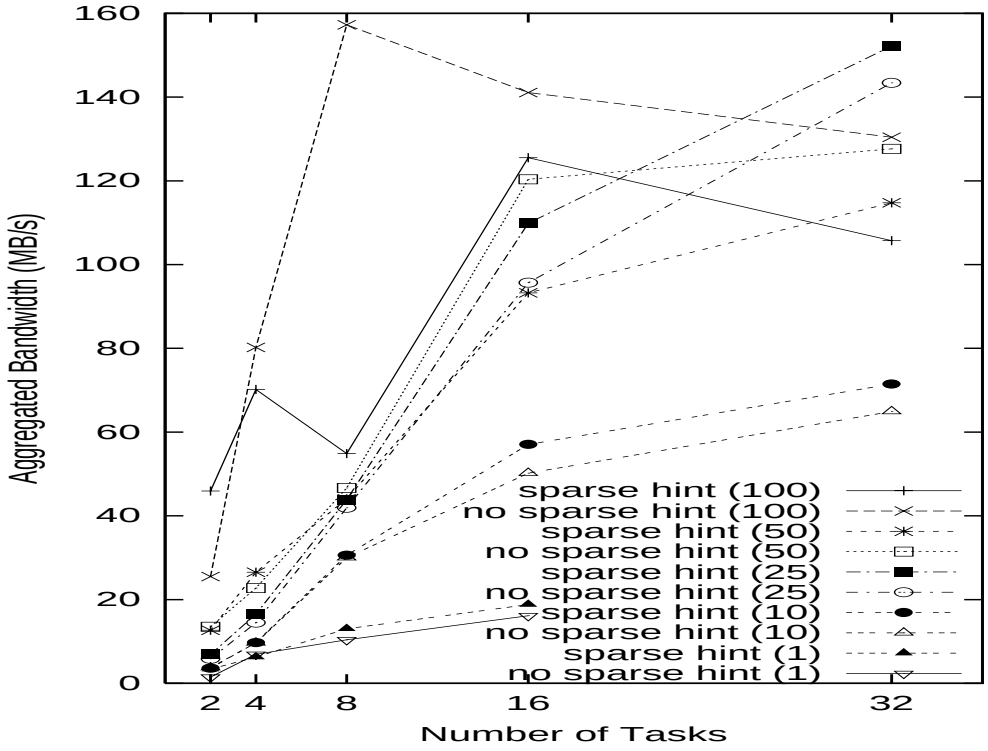

Figure 6: Averaged bandwidths measured for write operations in the sparse access benchmark

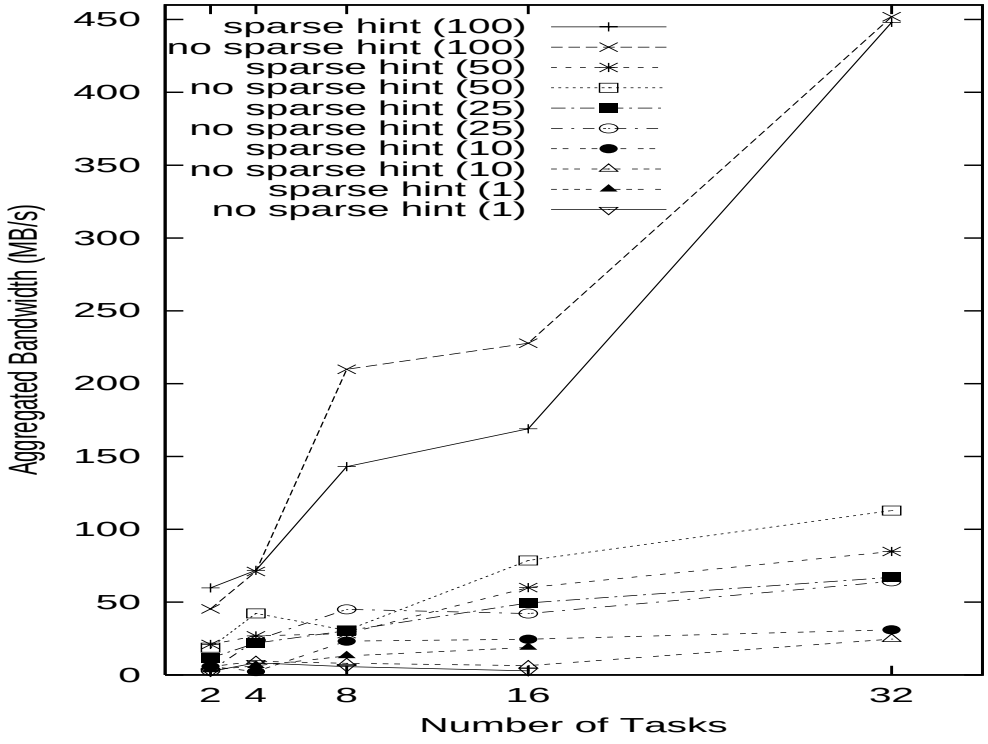

Figure 7: Averaged bandwidths measured for read operations in the sparse access benchmark

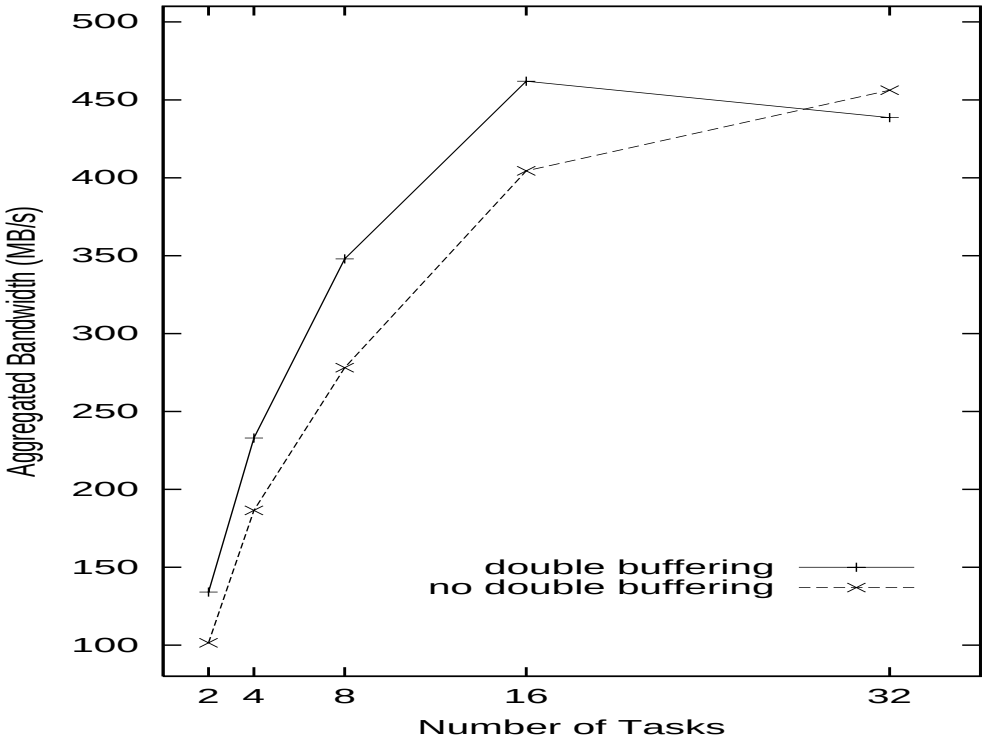

Figure 8: Averaged bandwidths measured for write operations in the discontiguous benchmark

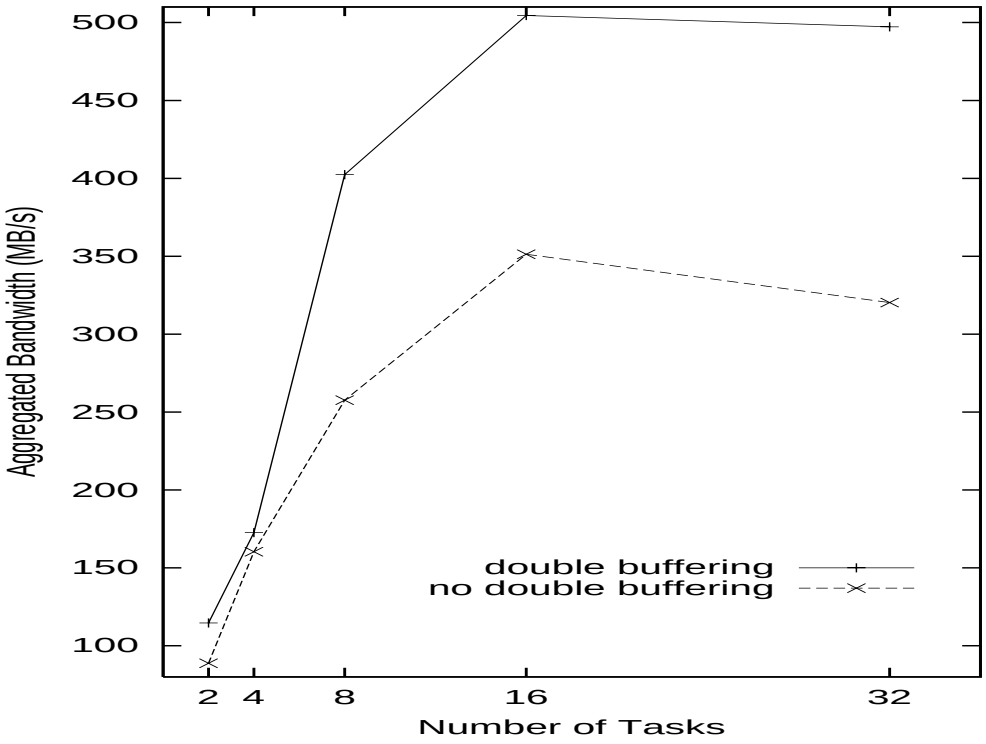

Figure 9: Averaged bandwidths measured for read operations in the discontiguous benchmark

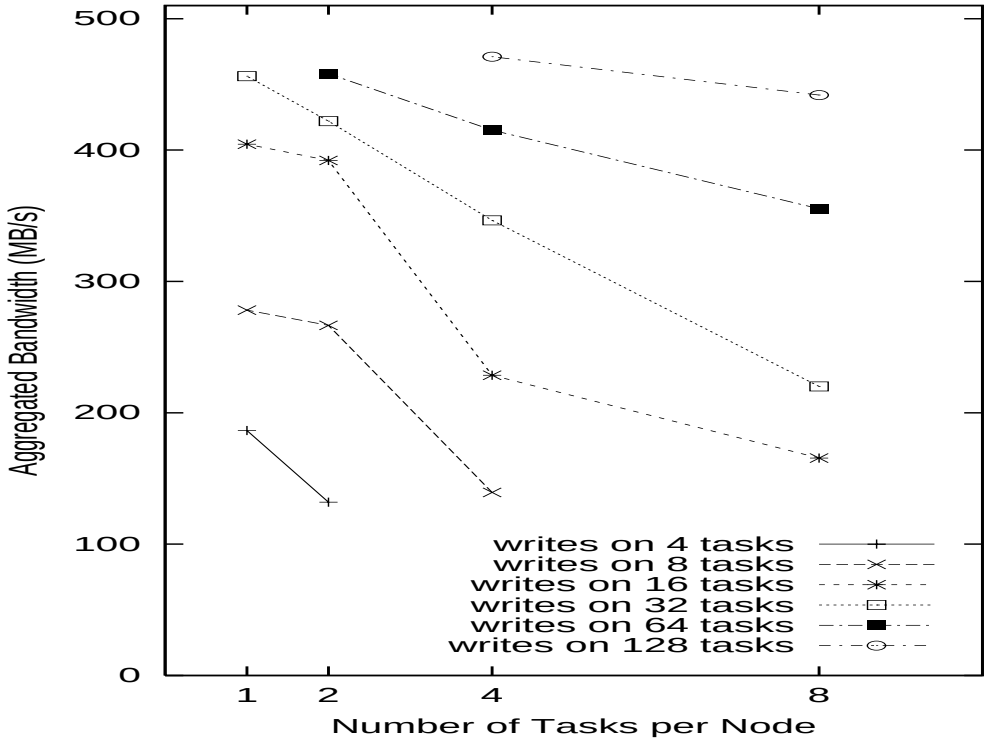

Figure 10: Averaged bandwidths measured for write operations in the discontiguous benchmark running on various numbers of tasks per node

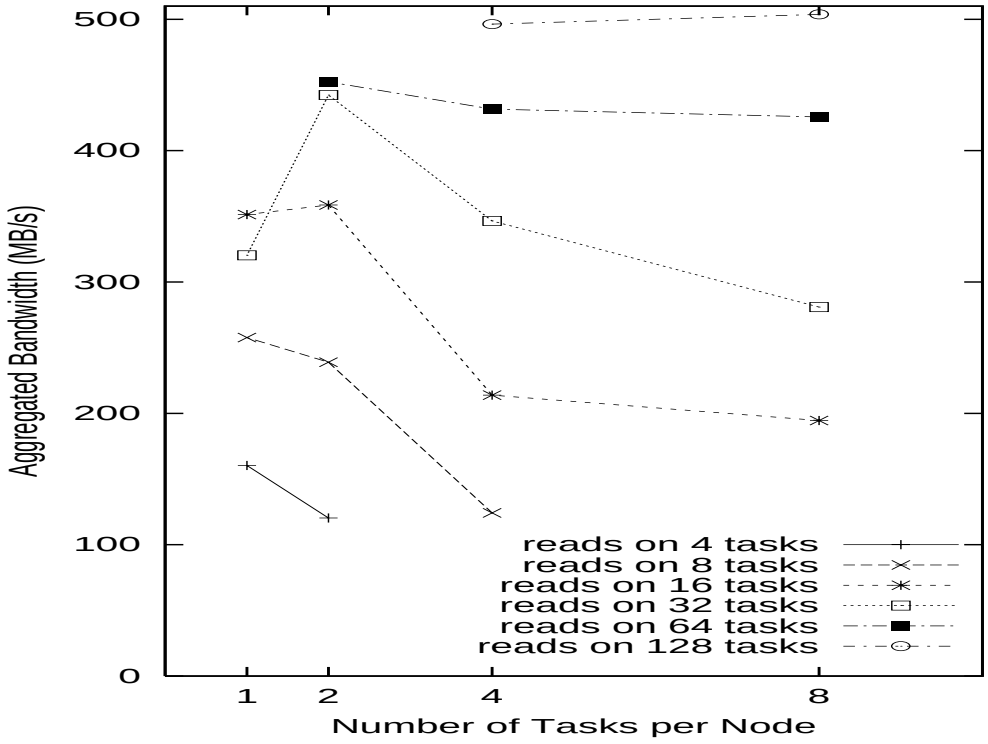

Figure 11: Averaged bandwidths measured for read operations in the discontiguous benchmark running on various numbers of tasks per node