Preprint
UCRL-JC-137128

Towards a High-Performance and Robust Implementation of MPI-IO on top of GPFS

J.-P. Prost, R. Treumann, R. Blackmore, C. Hartman, R. Hedges, B. Jia, A. Koniges and A. White

This article was submitted to
Europar 2000, Munich, Germany, August 29 – September 1, 2000

January 11, 2000

U.S. Department of Energy

Lawrence
Livermore
National
Laboratory

Towards a High-Performance and Robust Implementation of MPI-IO on top of GPFS

Jean-Pierre Prost[†], **Richard Treumann**[‡]
Robert Blackmore[‡], **Carol Hartman**[*], **Richard Hedges**[§],
Bin Jia[‡], **Alice Koniges**[§], **Alison White**[‡]

[†] IBM T.J. Watson Research Center, Route 134,
Yorktown Heights, NY 10598

[‡] IBM Enterprise Systems Group, 2455 South Road,
Poughkeepsie, NY 12601

[*] IBM Almaden Research Center, 650 Harry Road,
San Jose, CA 95120

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

January 11, 2000

Abstract

MPI-IO/GPFS is a prototype implementation of the I/O chapter of the Message Passing Interface (MPI) 2 standard. It uses the IBM General Parallel File System (GPFS), with prototyped extensions, as the underlying file system. This paper describes the features of this prototype which support its high performance and robustness. The use of hints at the file system level and at the MPI-IO level allows tailoring the use of the file system to the application needs. Error handling in collective operations provides robust error reporting and deadlock prevention in case of returning errors.

Keywords: MPI-IO, GPFS, file hint, prefetching, error handling, performance, data shipping.

1 Introduction

To provide users with a portable and efficient interface for parallel I/O, an I/O chapter was introduced in the Message Passing Interface 2 standard [For97], based upon earlier collaborative work between researchers at the IBM T.J. Watson Research Center and the Nasa Ames National Laboratory [Cor96].

Since approval of the MPI-2 standard, 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 [GPFS98] as the underlying file system. Some features of this MPI-IO prototype depend upon complementary GPFS prototype work. This paper describes the main features of the prototype, referred to as MPI-IO/GPFS. 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.

The use of GPFS as the underlying file system engenders maximum performance through a tight interaction between MPI-IO and GPFS. GPFS is a high performance file system which presents a global view of files to any client node. It allows parallel access to the data residing on server nodes through the IBM Virtual Shared Disk interface. GPFS provides coherent caching at the client, optimized prefetching techniques, and guarantees recoverability from any single point of failure. New hints and directives aimed at improving performance have been prototyped in the GPFS user interface as part of this project. These hints and directives are exploited by MPI-IO either transparently or through MPI_Info hints the user can specify.

To avoid file block contention among tasks, MPI-IO uses data shipping. This technique binds each GPFS block to a single I/O agent, which is responsible for all accesses to this block. For write operations, the tasks ship to each I/O agent, a command with that agent's write assignment and the data to be written. The agents then perform their assigned file writes. For reads, the tasks ship a command with the I/O agent's read assignment. The agents read the file as instructed and ship the data to the appropriate tasks. The GPFS blocks are bound to a set of I/O agents according to a round-robin striping scheme. MPI-IO/GPFS allows the user to define the stripe size. The stripe size also controls the amount of buffer space each I/O agent uses in each data access operation.

MPI-IO/GPFS is a robust and user-friendly implementation. It prevents deadlocks when an error occurs only on a subset of the tasks participating in a collective I/O operation. The scheme consists in having each task return either an error code corresponding to the error which occurred locally or a code stipulating that an error occurred on another participating task. The MPI standard allows a user to inquire in a portable way about the error class of the error returned and take the appropriate action on each task. In addition, errors which occur at the file system level can be traced on a per I/O agent basis through an optional error reporting feature that the user enables with an environment variable.

In this paper, we illustrate how hints allow tailoring the use of the file system to the application needs, thereby improving performance. We also show that an inappropriate use of the hints may degrade performance and address the issue of how to describe hints to the users in a language they can understand. Error reporting in collective I/O operations is detailed in order to illustrate the robustness and user-friendliness of MPI-IO/GPFS.

The paper is organized as follows. In Section 2, we present the hints and directives which have been prototyped in GPFS to improve performance of parallel access to GPFS files. Section 3 details how data shipping is implemented in MPI-IO/GPFS. It also describes how GPFS hints and directives are used by MPI-IO/GPFS and how error handling in collective I/O operations is implemented. Section 4 presents performance measurements to demonstrate the benefit of using hints appropriately and the penalty incurred when hints are used inappropriately. Section 5 presents some conclusions and suggests possible future research directions for the MPI-IO/GPFS prototype. All figures referenced in the text are gathered in the Appendix.

2 GPFS Hints and Directives

In the early stages of developing MPI-IO/GPFS, the benefit of having the underlying file system work as closely as possible with the MPI-IO layer became clear. To this end, some experimental options within GPFS, file partitioning and multiple access range hints, were exported for use by the MPI-IO/GPFS implementation. This section describes these prototype features of GPFS.

2.1 GPFS File Partitioning

Although file partitioning [Vesta] was initially an experiment to improve the performance of applications that show fine grained block sharing access patterns, MPI-IO/GPFS also benefits from reduced overhead in the file partitioning mode.

File partitioning mode partitions the file into a number of large pieces and allocates the responsibility for reads and writes of each piece to a specific node. For example, blocks might be assigned across a set of nodes in a round-robin fashion. To partition a file among a set of n nodes numbered $0, 1, ..., n-1$, block i of a file is assigned to node number $i \ mod \ n$. Each node caches only those file blocks that are assigned to it. Therefore, in this mode there is never any data shared between nodes so there is no need for distributed locking, the default locking mode of GPFS [Patent]. Instead, a single shared lock on the file is issued to each node responsible for data to facilitate recovery for failure cases. Shared partition locking eliminates lock conflicts and greatly reduces the complexity and size of the state the lock manager maintains. When in file partitioning mode, GPFS can also perform I/O operations more efficiently by issuing operations of at least one file block, usually 256K, the default block size for GPFS.

To maintain the consistency of the file system data, all nodes must access a file using either distributed locking or file partitioning, not both at the same time. In our prototype, the two modes, shared partitioning and distributed locking are mutually exclusive. However, since distributed locking is the default, our prototype has an external interface so that applications can request and set up file partitioning mode.

GPFS file partitioning mode fits well with MPI-IO/GPFS data shipping mode (see Section 3.1.1). The MPI-IO/GPFS layer knows when file partitioning would be most beneficial and how to best map the partitioning. The MPI-IO/GPFS layer requests GPFS file partitioning mode and sets up the partitioning parameters taking the burden off the user application. When MPI-IO/GPFS data shipping and GPFS file partitioning work together, the application benefits from the strengths of both services. MPI-IO/GPFS can most efficiently ship the data from the requesting nodes to the GPFS nodes responsible for the data and GPFS can run more efficiently by always issuing large I/O requests and using shared partition locking.

File partitioning mode is controlled through external interfaces to GPFS. In the prototype, the interfaces are called directives. This term is used instead of hints because hints by definition can be ignored, whereas directives cannot. Once an application issues a file partitioning directive, GPFS executes the directive and returns. If an error occurs or if GPFS is unable to complete the directive, an error code is returned.

There are three directives for file partitioning: start, declare a server mapping, and stop. The start directive is a collective operation that specifies an open file and the number of tasks participating in the file partitioning mode. Each task that issues a start directive is called a client task. By default, GPFS will use a round robin block partitioning, assigning one file block in turn to each node from which a start directive is issued. Using the map directive overrides this behavior. The map directive specifies a partition size, and the list of server nodes. This allows users, or MPI-IO/GPFS, to tailor the file partitioning to their application. Server nodes can be responsible for any number of contiguous blocks and can also appear any number of times in the partitioning. Figure 1 shows two examples of file partitioning with a varying number of servers and varying partition sizes.

To terminate file partitioning mode, all client tasks that issued a start directive must issue a stop directive. Once all the stop directives have been received by GPFS, the file is taken out of file partitioning mode. If a client task issues the close() system call before the stop directive, close() will issue a non-blocking stop directive on the task's behalf, however this is not recommended.

The greatest improvement in performance was found when doing small closely spaced write operations to one file from many nodes. In this case, we saw up to an 800% improvement in performance when using the file partitioning mode. There were gains seen in other patterns as well, though not as dramatic. The following is a brief synopsis of tests performed with our prototype. For all of the tests, an 8 node SP2 system was used. Each node had 1GB of main memory of which GPFS was given 40MB as a buffer cache. Each node was a VSD server as well which served four 4GB disks to the one GPFS file system. Thus, the GPFS file system which was mounted across all 8 nodes was 128GB. The file sizes used were all 2GB.

Two strided tests were performed for both read and write. Strided is defined as an operation of a given size repeated at a regular interval in the file.

The first test was a tiled strided pattern. With n nodes, numbered *0 to n-1* and an operation size of m bytes, this pattern is; node i starts at offset $i * m$ bytes, reads or writes m bytes, skips *(n-1)*m* bytes and repeats the operation. This pattern, with small records and a small number of nodes, yields several requests per GPFS file block. Multiple operations per block means data can be found in the cache for reads, but results in fine grained block sharing for writes. Note that even tiled strided reads with small records can be improved by using file partitioning. The benefit is that each block of the file needs to be read only once, instead of once by each node. On the down side, the overhead of additional messages and data copies will take back some of the performance gain.

The second test was for a sparse strided pattern. This time the strides were deliberately chosen so that no two consecutive operations at a node were to the same GPFS file block. For example, with a record size of 12K, the stride was defined as $24 * 12K$. Note that the stride grows with the number of nodes in the first test. For larger node counts, nodes no longer do multiple operations per block, reads do not find data already in cache and I/O patterns begin to look like those in the second test.

Not surprisingly, for access patterns that exhibit a high degree of write/write block sharing, file partitioning avoids all token conflicts and shows the greatest performance improvement. Figures 2 and 3 show the results of these tests.

2.2 GPFS Multiple Access Range Hint

GPFS recognizes sequential file access patterns and issues prefetchs and write behinds to maximize throughput [Intro]. However, some applications have access patterns that are not sequential or even regular. Yet the application does know the pattern. This is often the case for MPI-IO/GPFS. In the prototyped GPFS, we added the multiple access range hint allowing an application to communicate to GPFS its intended access pattern, both which blocks will be accessed soon and which are no longer needed.

With both sets of information, GPFS can best manage its buffer cache and provide progress on a maximum of prefetches. Successful prefetches enable GPFS to satisfy upcoming requests directly from its cache. When running at its best, the multiple access range hint can change the behavior of the application from having to do totally synchronous I/O operations to totally asynchronous I/O operations. This changes the performance from the speed of the disk subsystem to the speed of the memory subsystem. Figures 4 and 5 show the effects on GPFS's ability to do prefetch with and without the multiple access range hint.Notice in figure 4 all I/O is synchronous and in figure 5 all I/O becomes asynchronous and the job is much faster.

The multiple access range hint takes two parameters, an array of blocks that will soon be accessed and an array of blocks that the application is finished accessing. There is a maximum number of blocks that may be given in one hint. Depending on the current load, GPFS may initiate prefetching of some or all of these blocks. Each block specified in the prefetch list that is accepted for prefetching should eventually be released via an identical entry in the finished list, otherwise GPFS will reach a maximum number of cached blocks for this file and refuse any more prefetch requests for it.

Two tests were done to show the benefits of the multiple access range hint, random reads and random writes. Using record sizes ranging from 4096 to 512Kbytes, the multiple access range hint allowed the test application to specify its future accesses so that GPFS did prefetching even for this irregular access pattern. Using this hint raised the read rate by 67%. Running the random read test on only a single node improved

the performance by more than 500%. Random writes with the multiple access range hint improved the rate by 130%. The performance is not as high when using many nodes because the test program creates similar random patterns which often cause conflicts for prefetched blocks. There will be no such conflicts when MPI-IO/GPFS uses multiple access range hints.

A final test performed for random writes used both multiple access range hints and file partitioning which alleviated any write/write sharing. Combining the two options improved performance by more than 200%. Figure 6 shows the results of these tests.

3 MPI-IO/GPFS Features

This section first presents the features which enable MPI-IO/GPFS to achieve high performance with user applications which can exhibit very distinct I/O patterns. Then, features of MPI-IO/GPFS which provide robustness and user-friendliness are detailed.

3.1 High Performance Features

This section describes the various high performance features of MPI-IO/GPFS. The foundation of the design of MPI-IO/GPFS is the technique we call data shipping. The user may enable or disable this feature on a per open file basis. Additional features allow the user to instruct MPI-IO/GPFS, on a per open file basis, to use GPFS file partitioning or to guide, on behalf of the application, the prefetching of GPFS blocks. These features are implemented by MPI-IO/GPFS through the use of the GPFS directives and hints introduced in Section 2. Finally, we raise the general issue of describing the purpose of a file hint in terms which are meaningful to the user, given that the relationship between MPI calls to do I/O and the activity at the file system level is complex and largely opaque to the MPI programmer. The choice of a meaningful name and value format for a file hint is challenging. The name should be mnemonic and the set or range of values the user selects from should have a recognizable relationship to the user's understanding of her program's I/O behavior.

3.1.1 Data Shipping

To prevent concurrent access of GPFS file blocks by multiple tasks, normally residing on separate nodes, MPI-IO/GPFS uses data shipping. This technique 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 a 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, illustrated in Figure 7. 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.

On a per file basis, the user can define the stripe size used in the allocation of GPFS blocks to I/O agents. The stripe size is the value of file hint IBM_io_buffer_size, which can be specified when the file is opened, or when the MPI_FILE_SET_INFO and MPI_FILE_SET_VIEW functions are called. It is possible to change the stripe size of an opened file as long as no I/O operation is pending on the file. The stripe size also controls the amount of buffer space used by each I/O agent in data access operations, justifying its name. The stripe size given by the user is rounded up by MPI-IO/GPFS to an integral number of GPFS blocks, and its default size is the number of bytes contained in 16 GPFS blocks. GPFS block size defaults to 256K unless set by a system administrator when GPFS is configured.

Finally, on a per file basis, the user can enable or disable MPI-IO data shipping by setting file hint IBM_largeblock_io to "false" or "true", respectively. MPI data shipping is enabled by default. When it is

disabled, tasks issue read/write calls to GPFS directly. This saves the cost of MPI-IO data shipping but risks GPFS block ping-ponging among nodes if tasks located on distinct nodes contend for GPFS blocks in read-write or write-write sharing mode. In addition, collective data access operations are done independently. Therefore, we recommend to disable data shipping on a file only when accesses to the file are performed in large chunks or when tasks access disjoint large regions of the file. In such cases, MPI-IO coalescing of the I/O requests of a collective data access operation cannot provide benefit and GPFS block contention is not a concern.

3.1.2 GPFS File Partitioning

GPFS file partitioning is activated in MPI-IO/GPFS on a per file basis when a file is opened in MPI_MODE_UNIQUE_OPEN mode and file hint IBM_largeblock_io is not set or is set to "false". The IBM_largeblock_io value suggests that MPI-IO data shipping mode, as described in Section 3.1.1, is to be enabled on the file. MPI_MODE_UNIQUE_OPEN promises there will be no other concurrent accesses to the file so GPFS distributed locking is not required. There are two relevant aspects to GPFS file partitioning. The benefit is that tokens only need to be acquired once (at the time GPFS file partitioning is started by MPI-IO/GPFS) by each I/O agent for the GPFS blocks it is responsible for. Tokens will not be revoked until the file is closed or until the user changes the I/O buffer size by specifying a new value for file hint IBM_io_buffer_size. When starting or restarting GPFS file partitioning, MPI-IO/GPFS sets a GPFS file partitioning map to match the MPI-IO/GPFS striping of GPFS blocks across the I/O agents. The potential cost of GPFS file partitioning mode is that when any node other than the one granted rights by the GPFS file partitioning map tries to access a block, GPFS must ship the data. By combining MPI-IO data shipping with this GPFS mode, we gain the token management benefit and never require GPFS to do any data shipping. GPFS file partitioning will most benefit applications which write several small data records out of each GPFS block they access.

3.1.3 Guided Prefetching of GPFS Blocks

The use of MPI_Datatypes to define a file view means that a single MPI-IO call to read or write data may implicitly represent a series of file I/O operations. The full series is known to the MPI library before the first file operation. MPI-IO/GPFS can exploit this "advance" knowledge by using the GPFS multiple access range hint. MPI-IO/GPFS can guide the prefetching of GPFS blocks when data shipping mode is enabled on an opened file. Each time an I/O agent data access is to occur, MPI-IO/GPFS analyzes the pattern of I/O requests the agent has been given. If the pattern contains a single byte range or a succession of byte ranges which appear to represent a regular strided pattern, MPI will allow GPFS to make its own prefetching decisions because sequential and strided access patterns are quickly recognized and efficiently handled by GPFS. In other cases, MPI-IO/GPFS will compute for each GPFS block the lower and upper bounds of a byte range which covers all the byte ranges to be accessed within that GPFS block. This builds a list, one byte range per GPFS block, of ranges to be accessed. Then, it will apply the following prefetching policy, assuming *max_pre* represents the maximum number of GPFS blocks that can be prefetched at once (a limit set by GPFS):

1. At the beginning of the data access operation, GPFS tries to prefetch *max_pre* byte ranges or all byte ranges MPI-IO/GPFS has to access (if the total number of byte ranges it has to access is less than *max_pre*).

2. GPFS informs MPI-IO/GPFS of how many GPFS blocks, *actual_pre*, were successfully prefetched. If the total number of prefetched blocks is zero, MPI-IO/GPFS accesses one byte range. Otherwise, it accesses the minimum of *actual_pre* and *max_acc* byte ranges, where *max_acc* represents the maximum number of byte ranges which can be accessed at once (for experimentation with the prototype, this number is controlled by an environment variable and can have any value between one and *max_pre*).

3. Then, MPI-IO/GPFS releases the byte ranges it just accessed and tries to prefetch as many GPFS

blocks which have not been either accessed or prefetched in order to get *max_pre* prefetched blocks, unless there are less than *max_pre* byte ranges remaining to be accessed.

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

3.1.4 File Hint Expressiveness

In Sections 3.1.1, 3.1.2, and 3.1.3, the reader can observe how MPI-IO/GPFS exposes its high performance features to the user. To control data shipping, the user can use two file hints which are called IBM_largeblock_io and IBM_io_buffer_size. These hints do not refer directly to MPI-IO/GPFS's data shipping mode. Instead the user expresses the general I/O pattern of her application on a per file basis and how much buffer space should be made available to MPI-IO/GPFS to process each data access operation. To control GPFS file partitioning, the user does not explicitly enable or disable it by calling the GPFS programming interface which allows sending to GPFS the appropriate file partitioning directives. Instead, MPI-IO/GPFS does it transparently to the user if the user has set the MPI_MODE_UNIQUE_OPEN mode when opening the file. The MPI standard provides MPI_MODE_UNIQUE_OPEN so the user can declare a case in which the MPI-IO implementation may be able to perform additional optimizations. Because file partitioning is valuable but cannot be used on shared files, the fit is quite neat. Finally, MPI-IO/GPFS controls, transparently to the user, the prefetching of GPFS blocks when data shipping is enabled on a file. The user is not aware of the fact that MPI-IO/GPFS sometimes uses the GPFS multiple access range hint under the cover to optimize data access operations.

To summarize, file hints can either be explicitly provided by the user or deduced by the MPI-IO implementation on behalf of the user. In the former case, hint names should have meaning which applies to the user's understanding of her application rather than to implementation details of high performance feature the hint triggers. In the latter case, the user is not even aware that file hints are being used on her behalf. The implementation triggers the file hint only when it estimates the hint would benefit the application, and may restrict the use of the file hint to specific conditions controlled by user specified hints (eg the GPFS multiple access range hint is only used by MPI-IO/GPFS when file hint IBM_largeblock_io is not set or set to "false"). The MPI_FILE_GET_INFO function allows the user to discover the hints that have been defaulted, accepted, or rejected for a given file.

3.2 Robust Error Reporting Capability

Certain extensions to the base functionality of MPI-IO were added to MPI-IO/GPFS, in support of better error detection, recovery and debugging. These additions fall into two broad categories, collective communication robustness and MPI-IO specific tracing. The addition of MPI-IO to the MPI standard alters the model of MPI programs. In the MPI-1 standard, there were, practically speaking, no MPI communication calls the user could expect to recover from. The default for MPI errors was to abort the job and MPI-1 programs tend not to check return codes. Unlike with communication errors, it is reasonable to expect a program with a failing I/O operation to recover and continue. Care should be taken in an MPI-IO implementation to prevent errors on one task causing hangs on other tasks. The enhancements to the collective operations allow better program error recovery, while the tracing functions allow a programmer more detailed information for error determination.

3.2.1 Collective communication enhancements in MPI-IO/GPFS

In MPI-IO, there are errors that a reasonable application would expect to deal with, e.g., an invalid path name on an MPI_FILE_OPEN call or a write operation to a read-only file system. A robust implementation of MPI-IO should allow an error to occur on one or more tasks yet allow the program as a whole to continue to run without hangs or unexpected results.

Many of the MPI-IO calls are collective, which leads to three distinct kinds of errors:

- The underlying I/O fails.
- Collective call arguments are inconsistent or invalid.
- Communication errors.

Communication errors remain fatal with no user recovery. Fortunately, such errors are rare. The other two classes of errors may be handled by users.

On any MPI call, arguments are locally checked for validity. When the call is an MPI-IO collective one, MPI-IO/GPFS is designed so that no caller with an invalid argument will return from the call until all the other tasks in the file handle group have been notified that there has been an error. If any task were to return immediately after detecting an argument error, without participating in the collective portion of the MPI-IO operation while the other tasks went forward, the other tasks of the file handle group would hang. This hang is prevented by storing the result of local argument validity check in a variable and doing an allreduce operation on it. The collective I/O operation proceeds only when it is known at all tasks that each task has been called and has valid arguments.

Most of the collective MPI-IO calls have some arguments which must be identical across all of the tasks in the file handle group. For performance reasons, it is desirable to avoid the overhead of consistency checking for each I/O call in a working program. On the other hand, consistency errors may lead to subtle problems which are not easily traceable. Within both collective communication and collective operations performed on behalf of MPI-IO operations, an environment variable, MP_EUIDEVELOP is provided for extended consistency checking. When develop mode is turned on by setting MP_EUIDEVELOP=DEVELOP, each argument that must be identical across all callers is verified in the following manner.

- An array is filled with all of the arguments which must be checked for consistency.
- This array is sent to the left neighbor and received from the right neighbor. Task 0 sends to task *p-1*, *p* being the number of tasks in the file handle group.
- Each element of the received array is compared to each element of the sent array. An error flag is set when any mismatch is found.
- MPI_Allreduce is called with the error flag as input to inform all tasks whether there was a neighbor to neighbor inconsistency between any pair of tasks.

An obvious optimization is to combine the reduction operations for develop mode consistency checking and regular validity checking into a single operation. This allows the consistency checking to be done at an incremental cost of a single MPI_Sendrecv operation.

3.2.2 MPI-IO/GPFS tracing

Since a data access operation may involve several I/O agents which might all encounter an error when accessing GPFS, MPI-IO/GPFS had to, as a quality of implementation matter, find a way to report all occurring errors at the file system level. Unfortunately, only one error code can be returned by an MPI-IO function call. Therefore, we defined a tracing scheme that would allow users to have access to all errors which occurred on any I/O agent during their application run.

To enable MPI-IO/GPFS tracing, the user sets an environment variable selecting either of two modes. In one mode, only errors which occur during the application run are reported. In the other mode, along with error tracing, all read/write calls to GPFS are reported with their return codes and the parameters passed to the calls. In the latter mode, the user can optionally set a second environment variable to limit the read/write operation tracing to operations performed above a specific file offset. By default, the read/write operation tracing starts at offset zero. Error records and read/write records are stored in one file per job task, in directory /tmp on each node. The file naming convention ensures information pertaining to the I/O agent at each task will be written to a separate file.

4 Performance Measurements

This section presents in detail the experimentation that we carried out on one of the three 488 quad-processor node ASCI Blue Pacific systems at Lawrence Livermore National Laboratory and on an 8 node SP system at the IBM Research Center in Yorktown Heights. First, each benchmark is described. Then, for each benchmark, the platform used for the experimentation is presented, and the experimentation parameters and results are detailed. Graphs showing actual bandwidths are provided to demonstrate the benefit of the MPI-IO/GPFS high performance features introduced in Section 3.1. The first two benchmarks, run at Livermore, used MPI-IO/GPFS on top of GPFS 1.2 which does not include the prototyped enhancements discussed in Section 3.1.1. The second two were run with MPI-IO/GPFS on top of the prototype version of GPFS on a much smaller machine inside IBM.

4.1 Benchmark Description

Let us first describe the four benchmarks used in our experimentation. Each is aimed at evaluating the benefit of the high performance features of MPI-IO/GPFS for a particular class of application I/O patterns. For all tests, the metric is read or write bandwidth expressed as Mb/second for the job as a whole. The benchmarks will run with any number of tasks and provide every task with the same amount of I/O to do. In our tests, the file size scales with the number of tasks. 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. In MPI semantic, the return from an MPI write operation does not guarantee that data has been committed to disk. To ensure that the entire write time is counted, each test does an MPI_File_sync and MPI_Barrier before taking the end time stamp. In noncollective I/O it is possible that some tasks do I/O faster than others and an argument could be made for averaging the times. The synchronizations ensure, in both collective and noncollective tests, we measure job time which we consider most meaningful.

4.1.1 Contiguous Benchmark

In the contiguous benchmark, each task reads or writes sequentially a block of data from/to a single contiguous region of a preexisting file. Region size, type of file access (read or write), and type of I/O operation (collective or noncollective) are parameters to the benchmark program. The number of tasks in the parallel job together with the region size determine the size of the file that is accessed. Use of either the IBM_largeblock_io or the IBM_io_buffer_size file hint is controlled with environment variable settings.

The program uses the MPI_MODE_UNIQUE_OPEN file mode when opening the file. Since the data pattern consists of contiguous pieces of data, the benchmark does not use MPI_Datatypes to create a mapped view of the file; instead each task calculates the offset at which to access the file, based on its task number and the region size. Measurement is over the following sequence of steps which occur on each task:

- read/write the data buffer to/from the task's assigned region of the file
- call MPI_File_sync
- call MPI_Barrier

and the result is expressed in megabytes per second for the job.

4.1.2 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 holes or overlap. In the case of writes, the benchmark program creates the file. Parameters control file size, type of file access (read or write), type of I/O operation (collective or noncollective). Environment variables may be set to govern the use of the IBM_largeblock_io or the IBM_io_buffer_size file hints, and to indicate whether MPI_File_preallocate should be used to preallocate space for the new file in the case of writes. This benchmark opens the file with mode MPI_MODE_UNIQUE_OPEN, to indicate that no other user will simultaneously access the file.

The benchmark program reads or writes the file region by region, using a two gigabyte region size. Each task is assigned some set of 1024-byte blocks to read or write in each region, and creates an MPI_Datatype to map those blocks into their proper locations in the current region. This is accomplished by passing the MPI_Datatype that maps the blocks as the filetype argument to MPI_File_set_view. A two gigabyte region size is used because it is the largest region possible with a 32-bit MPI library. The span of addresses that can be mapped with an MPI_Datatype is constrained by the size of an MPI_Aint variable, so a 32-bit MPI mandates a maximum region size of two gigabytes.

Since the tasks read or write different patterns of blocks at each iteration, the list of block assignments is inspected before each read or write, to determine the range and position of the blocks in the current region and to create the associated filetype. Measurement is over the following sequence of steps which occur at each task:

- loop on the number of two-gigabyte regions in the file:
  - scan the list of block assignments for blocks belonging to the current region
  - create the filetype for the current region
  - set the current file view
  - read/write the assigned blocks (collectively or noncollectively)
- end loop
- call MPI_File_sync
- call MPI_Barrier

and the result is expressed in megabytes per second for the job.

4.1.3 GPFS File Partitioning Benchmark

This benchmark measures the benefit of using GPFS file partitioning with MPI-IO/GPFS. Each task reads or writes the same small number of non-overlapping noncontiguous data blocks from/to the file. The benchmark loops on the data access operation. Starting addresses of all blocks are generated randomly across the file and scattered among tasks. To simplify observations of how GPFS file partitioning affects I/O performance, only noncollective data access operations are used in this benchmark.

The size of each block and the total number of blocks is specified by two benchmark parameters, so that scope and amount of GPFS file partitioning can be controlled precisely. Another parameter specifies whether to use GPFS file partitioning. Note that if the IBM_largeblock_io hint is set to "true" this parameter will be ignored and data will be accessed without using GPFS file partitioning.

By comparing the bandwidths obtained with GPFS file partitioning enabled (mode MPI_MODE_UNIQUE_OPEN set when the file is open) with those obtained with file partitioning disabled (mode MPI_MODE_UNIQUE_OPEN not set), on various numbers of tasks, for read and write operations separately, we can evaluate the performance improvement resulting from the use of GPFS file partitioning.

4.1.4 Multiple Access Range Benchmark

This benchmark measures the benefit of using the GPFS multiple access range hint within MPI-IO/GPFS. This benefit strongly depends on the application I/O pattern, on the strategy MPI-IO/GPFS applies to prefetch/release GPFS blocks, and on the actions GPFS takes in response to the number of blocks MPI-IO/GPFS asks it to prefetch or release. By varying the data access parameters, such as the total amount of data to be accessed, the amount of data to be accessed in each read/write operation, the distribution of the data to be accessed onto the file as well as among the tasks, the I/O buffer size used by each I/O agent, and the strategy MPI-IO/GPFS applies for prefetching/releasing GPFS blocks, this benchmark can collect bandwidth information for various interactions of these factors.

In this benchmark, each task reads or writes a number of non-overlapping GPFS blocks from/to the file. GPFS blocks are randomly assigned to tasks. Only sub-blocks scattered throughout each GPFS block are read/written. The number of GPFS blocks to be accessed, the number of sub-blocks to be read/written within each GPFS block, as well as the size of a sub-block are adjustable. An environment variable provides control of the number of blocks MPI-IO/GPFS tries to access before prefetching new GPFS blocks and releasing the GPFS blocks just accessed. A value of zero for this environment variable disables the use of the GPFS multiple access range hint. MPI derived datatypes are used to describe the layout of the sub-blocks to be accessed within each GPFS block.

As in the discontiguous benchmark, the construction of the filetype used for setting the file view for a specific data layout (controlled by the number of sub-blocks per GPFS block and their size) is included in the timing interval. Comparison of resulting bandwidths for various numbers of tasks allows to evaluate the impact of the prefetching policy enabled by MPI-IO/GPFS through the use of the GPFS multiple access range hint.

4.2 Benchmark Results

This section presents, for each benchmark, the hardware platform and versions of the key software components used in our experimentation. The performance data is presented in graphs, in the appendix at the end of the paper, to facilitate the comparison of the aggregated bandwidths obtained with or without use of the high performance features described in Section 3.1.

Sections 4.2.1 and 4.2.2 are based on performance measurements performed on IBM SP systems that are part of the ASCI Blue Pacific complex at Lawrence Livermore National Laboratory [ASCI99]. Sections 4.2.3 and 4.2.4 are based on performance measurements performed on an 8 node system at the IBM T.J. Watson Research Center.

The ASCI Blue Pacific systems are based on 4-way nodes utilizing 332 Mhz 604e Power2 processors. Each of the nodes has 1.5 Gb of memory. The particular system hosting the experimentation includes 425 compute nodes and 56 I/O nodes. The GPFS configuration includes 38 VSD servers. GPFS page pools occupy 50 MB on each compute node. The software configuration includes AIX 4.3.2+, PSSP 3.1 and GPFS 1.2. A more detailed description of the file system configuration and GPFS performance statistics can be found elsewhere [Jon00].

While it is preferable to make performance measurements on a dedicated system, each of the ASCI Blue systems is so heavily used that this is not possible. Thus, these measurements were carried out on the system in normal production operation. Despite this normal workload, efforts were made to make the measurements at times when other jobs were not contending for the GPFS file system. The peak performance on the contiguous benchmark with data shipping could be somewhat higher than our numbers, since it is affected by switch traffic and we are not in a dedicated environment. The peak performance we report for no data shipping is unlikely to change in a dedicated environment because our results are very reproducible. In order to obtain an estimate of the peak performance of the file system, we ran each of our experiments several times and then report the highest value as listed in the tables below. We also use the data obtained from several replications of the same runs in order to evaluate the impact of contention on the measurement

results.

The contiguous and discontiguous benchmarks were run on 4, 8, 16, 32, 64, and 128 MPI tasks. Each MPI task had an entire node of the machine dedicated to it. Measurements for MPI-IO/GPFS are compared to the same tests using the ROMIO (MPICH) implementation [ThaF99], [ThaM99], and to analogous tests utilizing a traditional POSIX I/O interface. By analogous, we mean that the POSIX tests use the same data layouts (across task memory and in the file) as the MPI-IO tests.

For all of the MPI-IO/GPFS measurements and for the POSIX version of the contiguous benchmark, I/O bandwidth was measured for seven replications of the same experiment. For the POSIX version of the discontiguous benchmark, the maximum achieved I/O bandwidths (< 10 MB/sec) made multiple measurements prohibitive. In this case, a single replication of each experiment was performed. The maximum measured aggregated I/O bandwidth is reported for each combination of benchmark, read or write operation, MPI-IO implementation, and number of tasks.

4.2.1 Contiguous Benchmark

For the contiguous benchmark (section 4.1.1) a region size of 333 MB was used. Each MPI task wrote a single region. In this situation, noncollective operations give the best performance. The data transfers are independent and can proceed in parallel with no coordination.

To illustrate the effect of using MPI data shipping (controlled by the IBM_largeblock_io file hint), Figure 8 compares the results of running the contiguous benchmark with MPI data shipping enabled (IBM_largeblock_io set to "false") to the same runs with MPI data shipping disabled (IBM_largeblock_io set to "true"). MPI data shipping is enabled by default in MPI-IO/GPFS but the default in the benchmark is to override that. For comparison, we also include results obtained with ROMIO and POSIX.

With data shipping disabled, the performance of MPI-IO/GPFS is strikingly similar to the ROMIO and POSIX versions of the benchmark. These three versions exhibit a saturation of GPFS's ability to process the data with saturation occurring between 32 and 64 MPI tasks.

For the contiguous benchmark with large file regions, contention of tasks writing to the same GPFS block is non-existent. Therefore, the logic and data movement involved in data shipping is purely overhead. For task numbers below 64, performance is less with data shipping enabled. The overhead of data shipping has an interesting effect for larger task numbers (64 and 128). We surmise that the overhead reduces the rate of queueing requests to GPFS providing a sort of flow control which leads to better performance with these large task numbers.

In Figure 9, results of the contiguous benchmark for read operations are displayed. Again with data shipping disabled, MPI-IO/GPFS performs like the POSIX or ROMIO versions of the benchmark. In the case of reads, the data layout is such that no data shipping is needed for optimal performance. Flow control (or lack of it) is not an issue for the read tests. Enabling data shipping introduces additional overhead, and the read performance with data shipping enabled is consistently lower than with the other three versions of the benchmark.

The range of measured write bandwidths for the contiguous benchmark was observed to depend on the status of data shipping (controlled by the value of the IBM_largeblock_io hint). This variability of measured performance is particularly evident for larger numbers of tasks. In Figure 10, we produce the standard deviation, σ, of the aggregated write bandwidth for the two cases,

$$\sigma = \sqrt{\frac{n\Sigma x^2 - (\Sigma x)^2}{n(n-1)}}$$

where x is a single bandwidth measurement and n is the number of measurements. (We removed one data point from our sample that had a very low value potentially due to system problems.) The standard deviation with data shipping enabled exceeds the standard deviation with data shipping disabled for experiments with 32 or greater number of tasks. For 128 tasks, the standard deviation with data shipping is 35 times larger.

The data is for MPI-IO/GPFS only, since the results for ROMIO and POSIX are very similar to those obtained with MPI-IO/GPFS and data shipping disabled.

A possible explanation for the dependence on data shipping, of the variability (expressed in terms of the standard deviation) of the measured bandwidth is the following. When data shipping is enabled, all data must be sent from the tasks to the I/O agents, and write calls are issued by the I/O agents. We notice an increasing variability in the bandwidth with an increasing number of tasks since there is more contention for switch access and because all tasks are trying to send data concurrently to each I/O agent. When data shipping is disabled, i.e., the IBM_largeblock_io is set to "true", there is no message passing occurring between the MPI tasks and the I/O agents during data access operations. Here, tasks are issuing I/O calls to GPFS directly. In this latter case, the variability in the measurements goes down to an almost insignificant level at larger numbers of tasks. The turnover in the standard deviation curve without data shipping occurs when the write rates begin to saturate because of the lack of flow control in GPFS 1.2. In summary, users on a production machine such as this one may see a large variability in their write performance for large numbers of tasks with data shipping, and significantly reduced variability in their write performance without data shipping.

4.2.2 Discontiguous Benchmark

The discontiguous benchmark (described in Section 4.1.2) makes use of the optimized collective operations with data shipping enabled. The file size in this benchmark amounts to 333 MBytes per MPI task. Again, the POSIX and ROMIO versions of this benchmark are compared to MPI-IO/GPFS.

In Figure 11, results of the write performance tests for the discontiguous benchmark are presented. Initially, the most striking aspect of the plot is the dismal performance when using the straight POSIX interface. It is however not so surprising that writing to a GPFS file system in one KB block leads to very poor performance. Experimental results show that MPI-IO/GPFS with data shipping disabled leads to qualitatively similar performance.

In Figure 12, results of the read performance for the discontiguous benchmark are presented. Again, the performance of the POSIX version of the test shows poor performance as we would expect. MPI-IO/GPFS and ROMIO show good performance, achieving over 375 MB/sec and 250 MB/sec respectively.

ROMIO shows substantial optimization of the I/O in comparison to the POSIX results. The performance of ROMIO shows qualitatively different behaviors for the write and the read tests. Where performance of the read test is scaling well, performance of the write test peaks in the neighborhood of 32 nodes and then declines as the number of tasks increases.

We suspect that the additional load to GPFS of the read - modify - write cycle for data sieving in ROMIO has led to the same sort of flow control issues noted in the write performance of the contiguous test. [ThaF99].

MPI-IO/GPFS shows the best write performance in the discontiguous benchmark reaching over 300 MB/sec and the best performance in the read test, reaching over 375 MB/sec. This demonstrates the benefit of using data shipping combined with collective operations, leading to good scalability for both read and write operations.

4.2.3 GPFS File Partitioning Benchmark

The GPFS file partitioning benchmark sets the IBM_largeblock_io hint to "false" and the IBM_io_buffer_size to 16MB. The MPI_MODE_UNIQUE_OPEN file mode is set (resp. unset) when opening the file in order to enable (resp. disable) GPFS file partitioning. The GPFS file partitioning benchmark was run on 1, 2, 4, and 8 tasks, with one task per node. 10000 8KB blocks were read or written by each task from or to the file. I/O bandwidth was measured for seven replications of the same experiment.

The results for write operations are shown in Figure 13. Clearly, enabling GPFS file partitioning results

in substantial I/O performance improvement in the case of writes.

The results for read operations are shown in Figure 14. There is no significant difference between enabling and disabling GPFS file partitioning in the case of reads, since tokens granted once are never revoked when file blocks are accessed only in read-read sharing mode.

4.2.4 Multiple Access Range Benchmark

The multiple access range benchmark was run on 1, 2, 4, and 8 tasks. The number of GPFS blocks accessed by each task was 100 and the size of each sub-block was 4KB. Combinations of various numbers of sub-blocks (1, 2, 8, 16) and various values for *max_acc* (0, 1, 2, 4, 8) were experimented with. 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 of the multiple access range benchmark obtained for 8 tasks and 16 sub-blocks accessed in each GPFS block are shown in Figure 15 for the various values of *max_acc*. Results of seven replications of the same experiment were averaged to get the bandwidth data. Substantial I/O performance improvement can be observed, comparing the result of enabling the use of the GPFS multiple access range hint (*max_acc* set to 1, 2, 4, 8) and of disabling it (*max_acc* set to 0). It also clearly appears that lower values for *max_acc* lead to better performance improvement, especially for reads which benefit the most from the use of the GPFS multiple access range hint (as already observed in Section 2.2).

Figure 16 shows the bandwidths obtained for the various numbers of tasks with 16 sub-blocks accessed in each GPFS block and the values of 0 (no use of GPFS multiple access range hint) and 2 (a maximum of two GPFS blocks are accessed between two prefetching operations) for *max_acc*. The results show that the benefit produced by using the GPFS multiple access range hint increases with the number of tasks accessing the file, again more noticeably for reads.

5 Conclusion

This paper illustrates how careful or even clever use of file hints can allow users to improve the performance of their MPI-IO applications on top of the IBM General Parallel File System. Data shipping is a clear winner when data is read-write or write-write shared among several tasks. Increasing the stripe size and the I/O agent buffer size leads to better performance, provided the application can spare the buffer space. We also show how the MPI-IO/GPFS prototype itself makes use of GPFS hints in order to optimize prefetching and minimize the number of requests made to the storage device servers. Lastly, MPI-IO/GPFS robustness and user-friendliness is illustrated through its error reporting capability, which prevents deadlocks which may result from errors occurring on a subset of the tasks participating in a collective I/O operation.

We are currently investigating whether double buffering at the I/O agent can lead to increased performance. We are also examining whether data sieving, already used in ROMIO [ThaF99], could be beneficial to our prototype implementation. We are planning to study whether feedback from GPFS about the block hit ratio induced by its buffer cache replacement and prefetching policies or about the I/O request service time at the server side can be exploited by MPI-IO so that it can adapt its own prefetching policy or help GPFS control the I/O request flow to an overloaded server.

We are also starting to define synthetic benchmarks which can, through the setting of multiple parameters study and compare the performance of various MPI-IO implementations for several classes of multi-threaded applications. The number of threads accessing file data concurrently allows to control the level of overlap between computation and I/O exhibited by the application and to evaluate the impact on the application performance of thread scheduling policies.

6 Acknowledgements

The authors would like to thank Robert Kim Yates for his thorough reviewing of the paper and for the valuable comments he provided. The ASCI Blue Pacific system and the SP2 system at the IBM T.J. Watson were used for the experimentation. We also thank other users of these systems for their patience and understanding, as well as the administrators of these systems for their assistance in setting up the appropriate system configuration.

References

[ASCI99] *www.llnl.gov/asci/.*

[Cor96] P. Corbett, D. Feitelson, S. Fineberg, Y. Hsu, B. Nitzberg, J.-P. Prost, M. Snir, B. Traversat, and P. Wong, Chapter in *Input/Output in Parallel and Distributed Computer Systems*, Ravi Jain, John Werth, and James C. Browne, Eds., Kluwer Academic Publishers, June 1996, pp. 127–146.

[For97] Message Passing Interface Forum, *Mpi-2: A message-passing interface standard*, Standards Document 2.0, University of Tennessee, Knoxville, July 1997.
www.mpi-forum.org/docs/docs.html.

[GPFS98] *IBM General Parallel File System for AIX: Installation and Administration Guide*, IBM Document SA22-7278-02, October 1998.
www.rs6000.ibm.com/resource/aix_resource/sp_books/gpfs/install_admin/install_admin_v1r2/gpfs1mst.html.

[Intro] *An Introduction to GPFS R1.2.*
www.rs6000.ibm.com/resource/technology/paper1.html.

[Jon00] T. Jones, A. Koniges, and K. Yates, *Performance of the IBM General Parallel File System*, Proc. International Parallel and Distributed Processing Symposium, May 2000. Accepted.

[Patent] Patent US5950199, Schmuck et al., *Parallel file system and method for granting byte range tokens*, IBM Almaden Research, San Jose, CA 95120. Issued July 1999.

[ThaF99] R. Thakur, W. Gropp, and E. Lusk, *Data Sieving and Collective I/O in ROMIO*, Proc. 7th Symposium on the Frontiers of Massively Parallel Computation, February 1999, pp. 182–189.

[ThaM99] R. Thakur, W. Gropp, and E. Lusk, *On Implementing MPI-IO Portably and with High Performance*, Proc. of the Sixth Workshop on I/O in Parallel and Distributed Systems, May 1999, pp. 23–32.

[Vesta] P. Corbett and D. Feitelson, *Design and Implementation of the Vesta Parallel File System*, Proc. of the Scalable High-Performance Computing Conference, May 1994, pp. 63–70.

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

Appendix

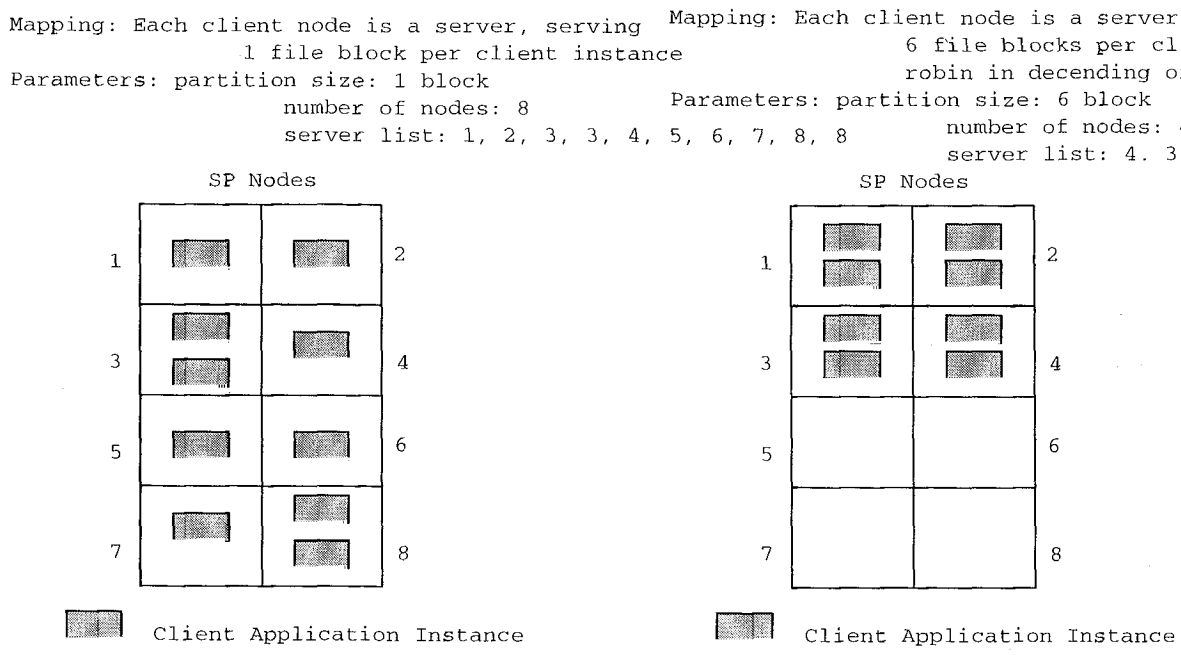

Figure 1: Sample mapping used by GPFS file partitioning

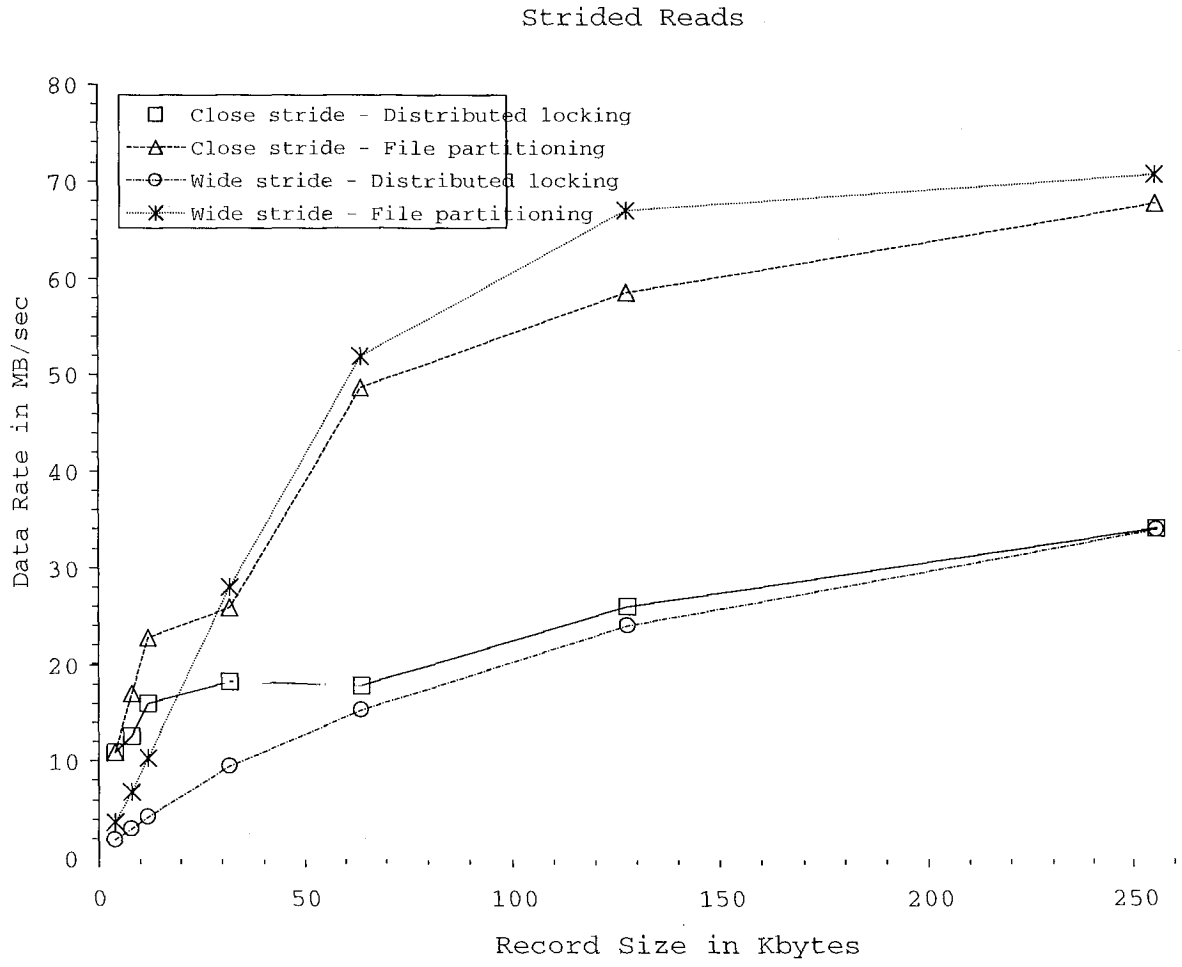

Figure 2: Read performance achieved by GPFS file partitioning

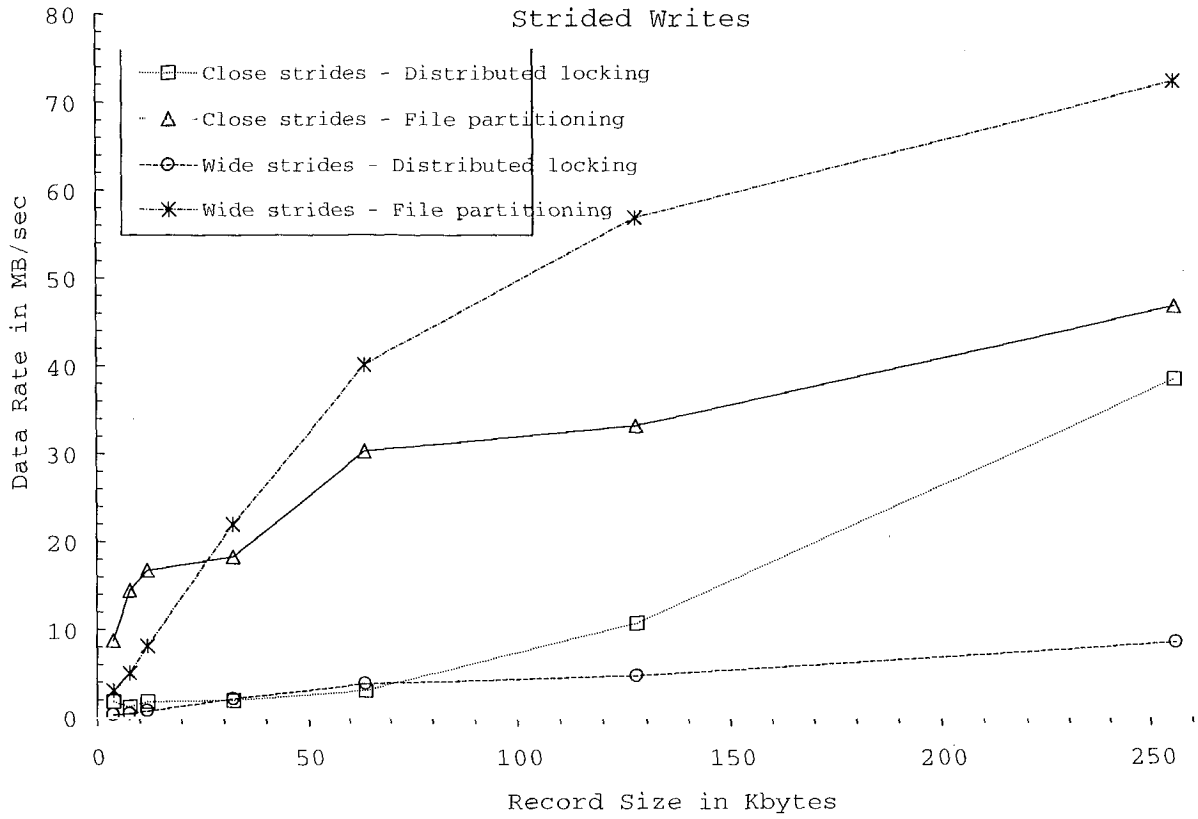

Figure 3: Write performance achieved by GPFS file partitioning

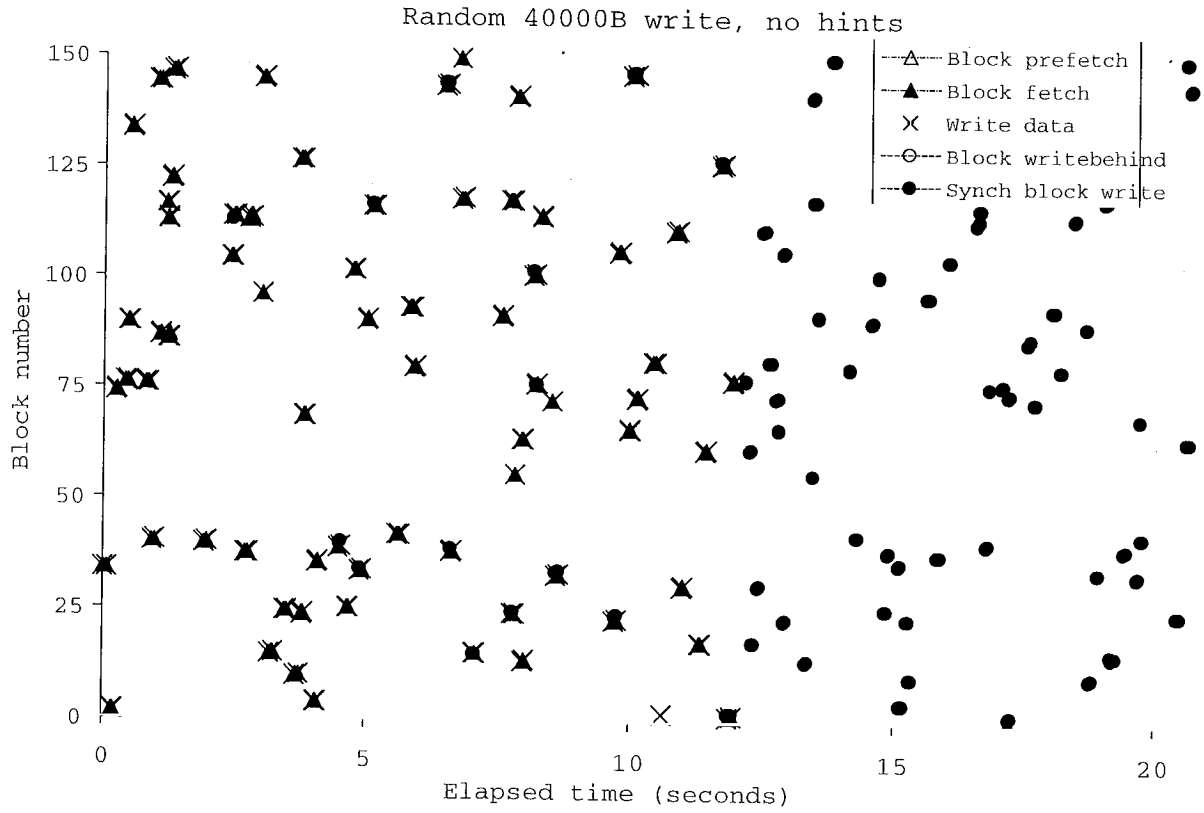

Figure 4: Effects on GPFS prefetch for random workloads

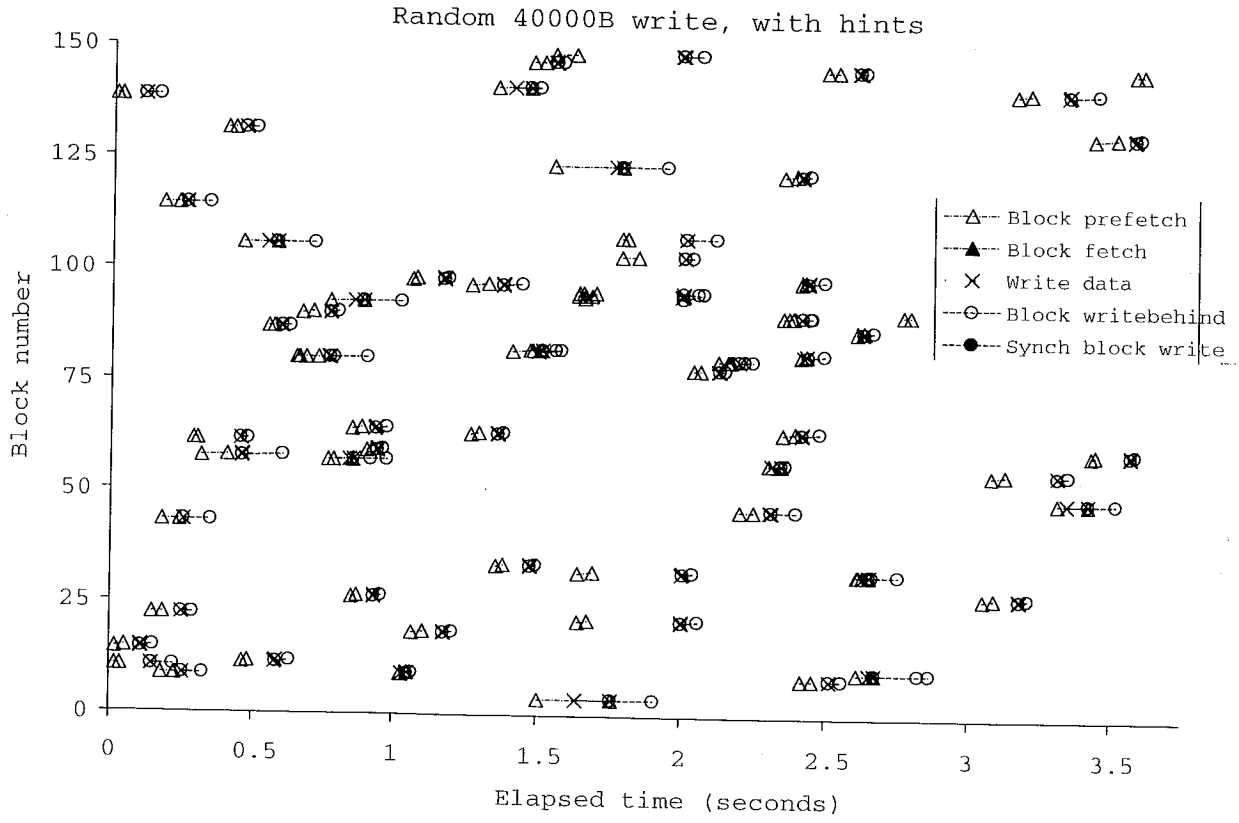

Figure 5: Effects on GPFS prefetch for random workloads with hints

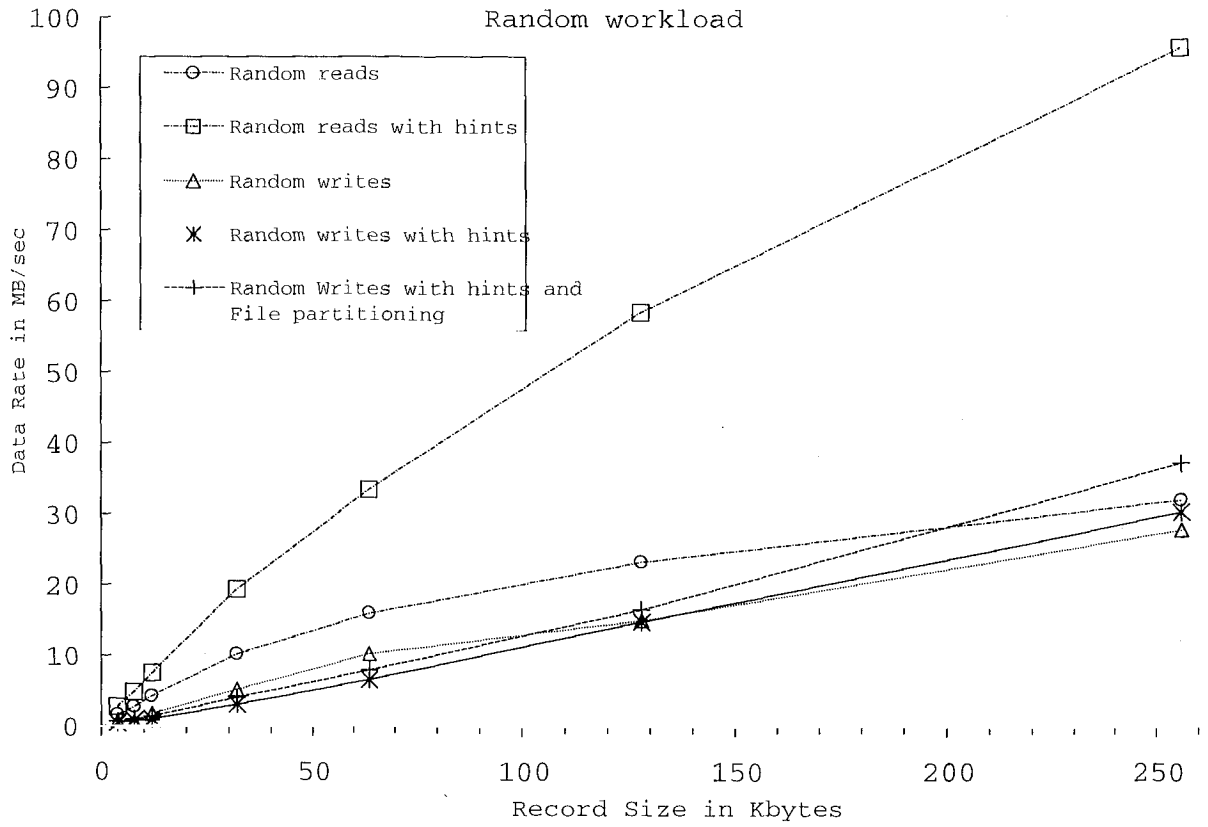

Figure 6: Read and Write performance achieved by GPFS on Random workloads with and without hints, and with hints and file partitioning

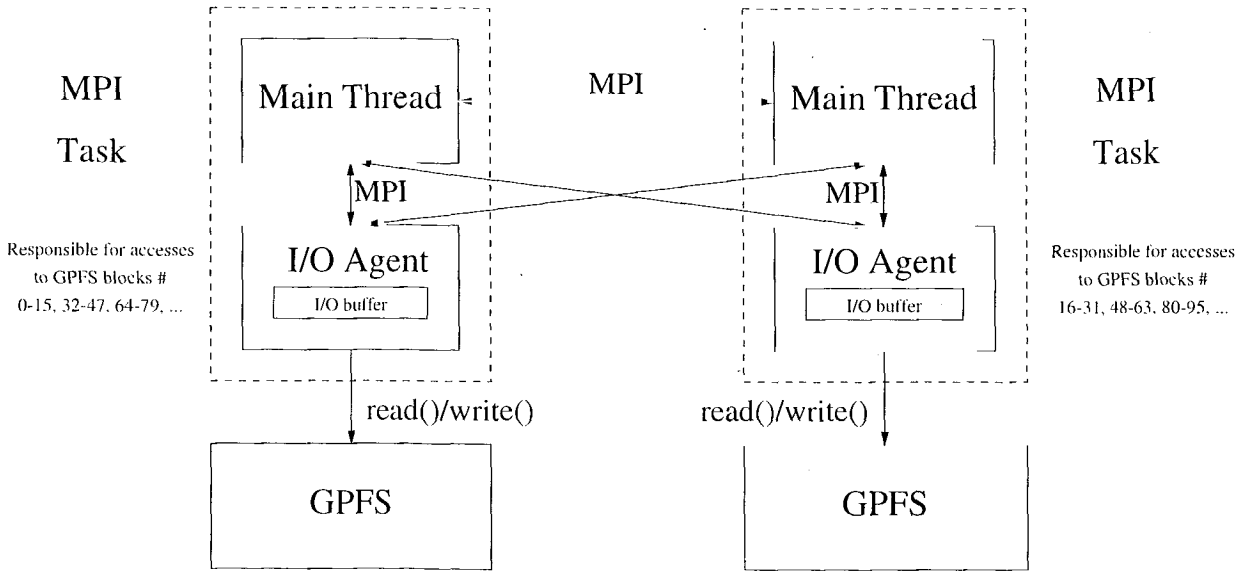

Figure 7: GPFS block allocation used by MPI-IO/GPFS in data shipping mode (using the default stripe size)

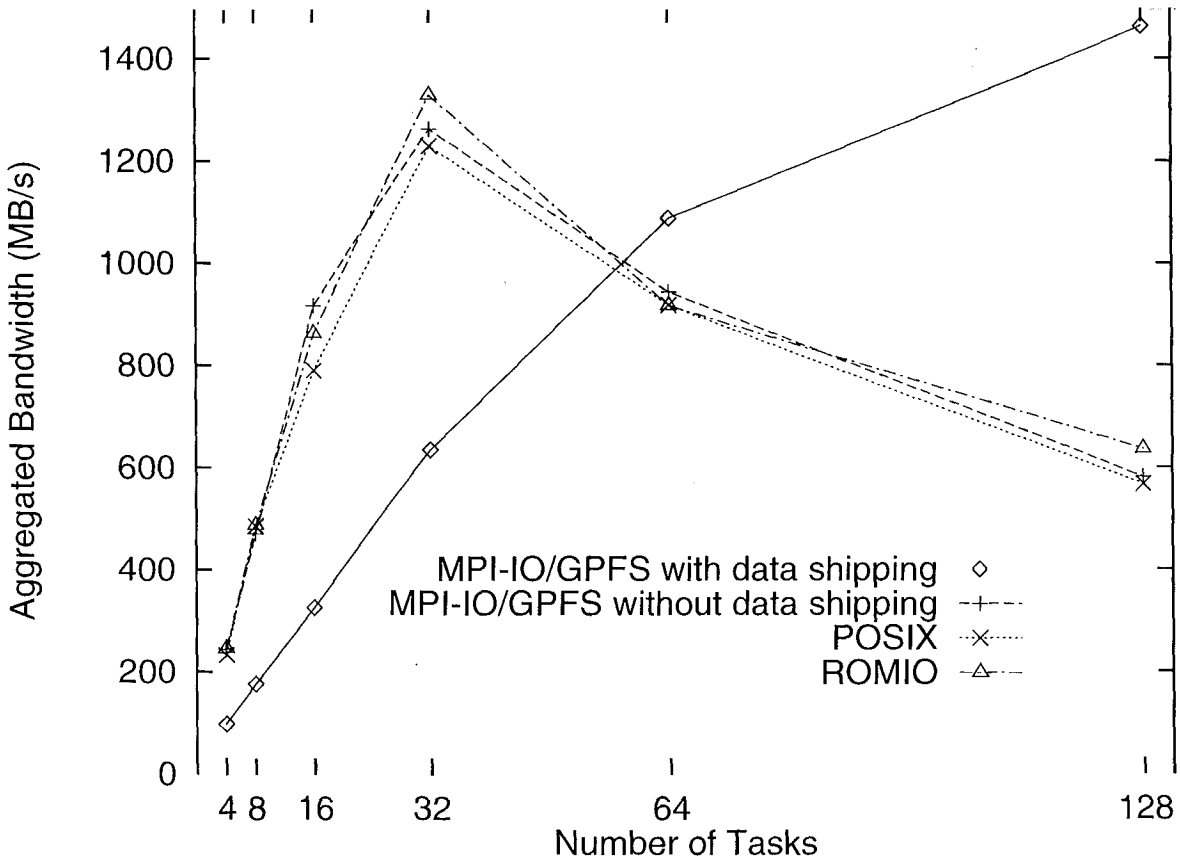

Figure 8: Maximum bandwidths measured for write operations in the contiguous benchmark

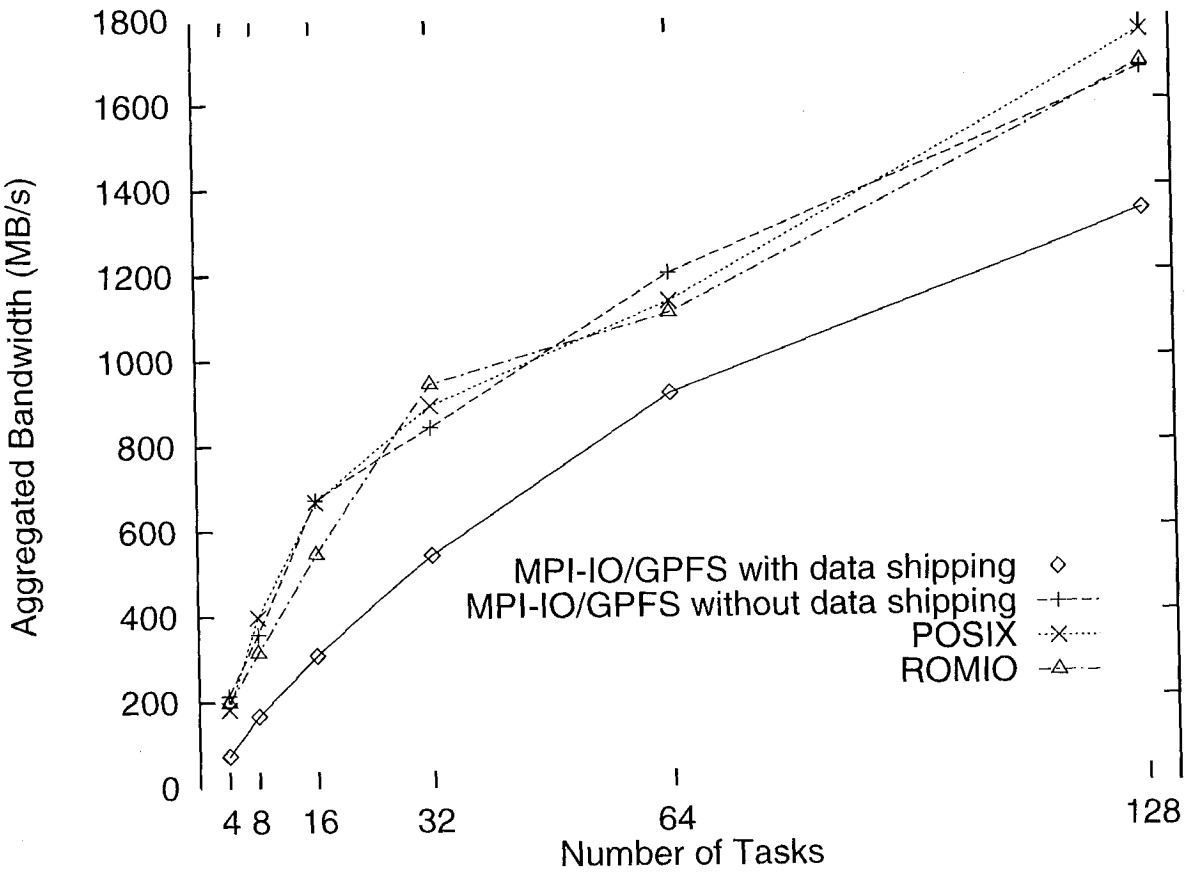

Figure 9: Maximum bandwidths measured for read operations in the contiguous benchmark

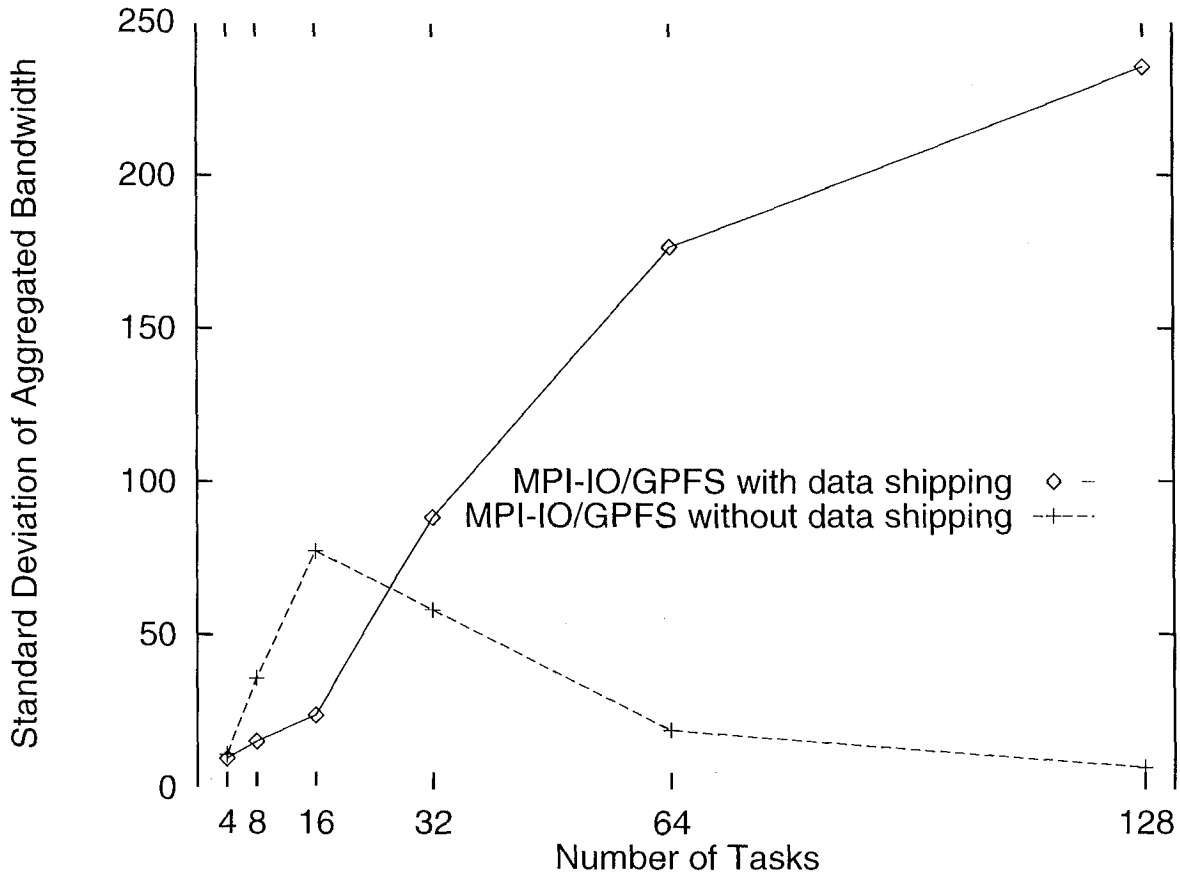

Figure 10: Effect of data shipping on measured standard deviation for the contiguous write benchmark

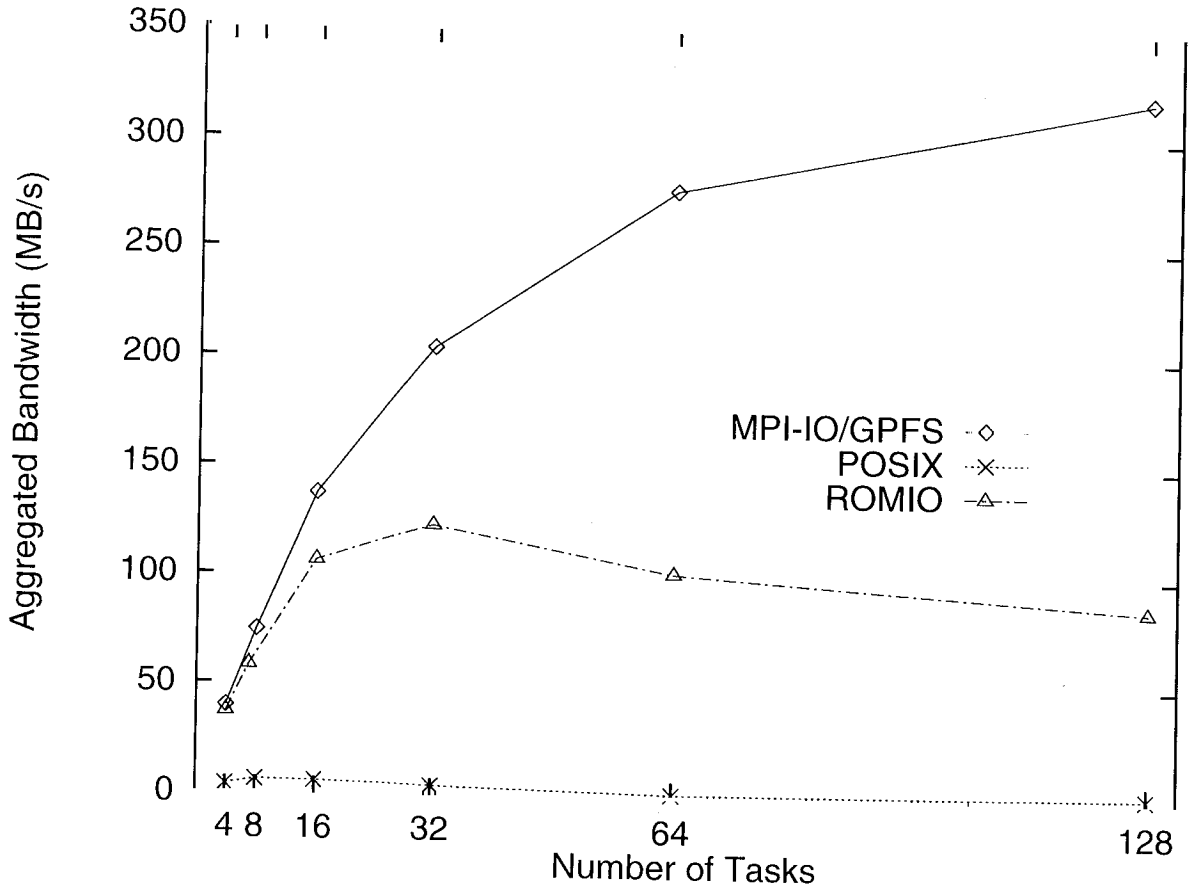

Figure 11: Maximum bandwidths measured for write operations in the discontiguous benchmark

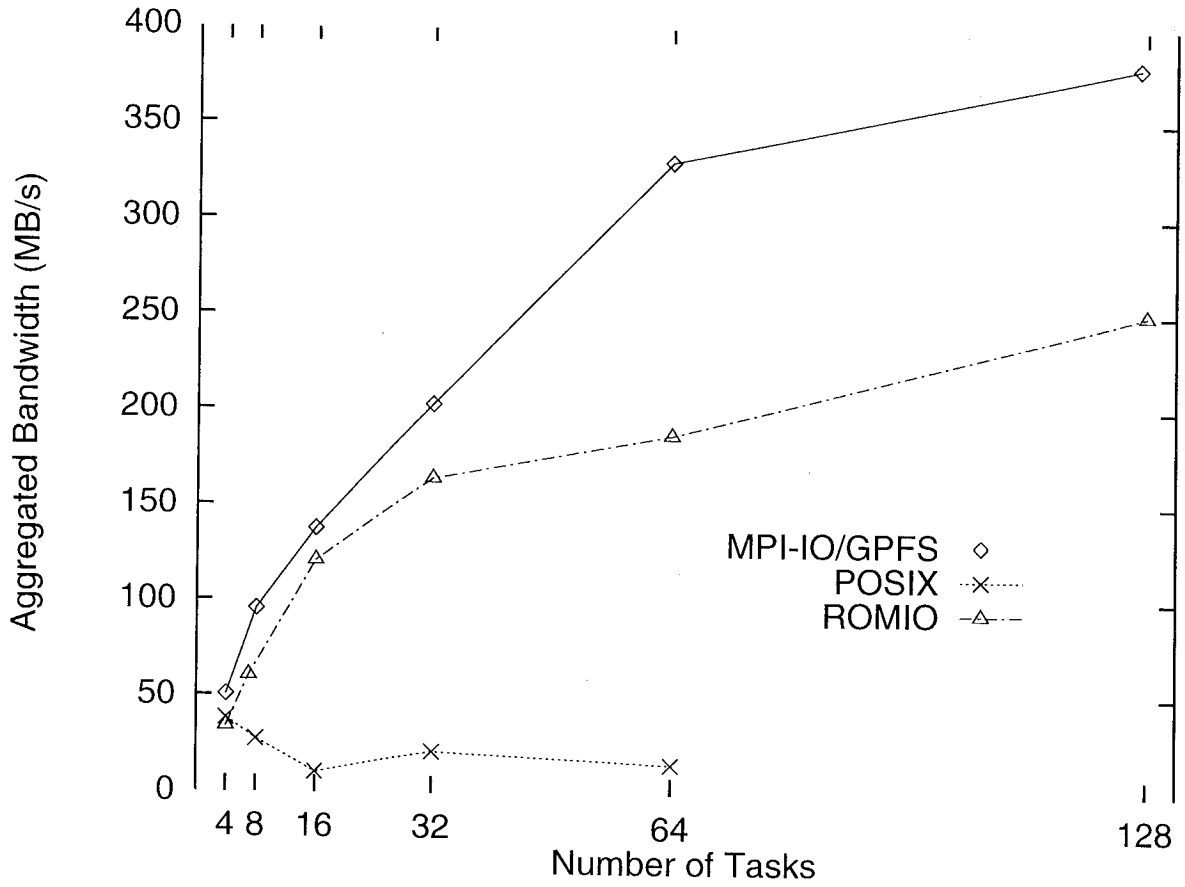

Figure 12: Maximum bandwidths measured for read operations in the discontiguous benchmark

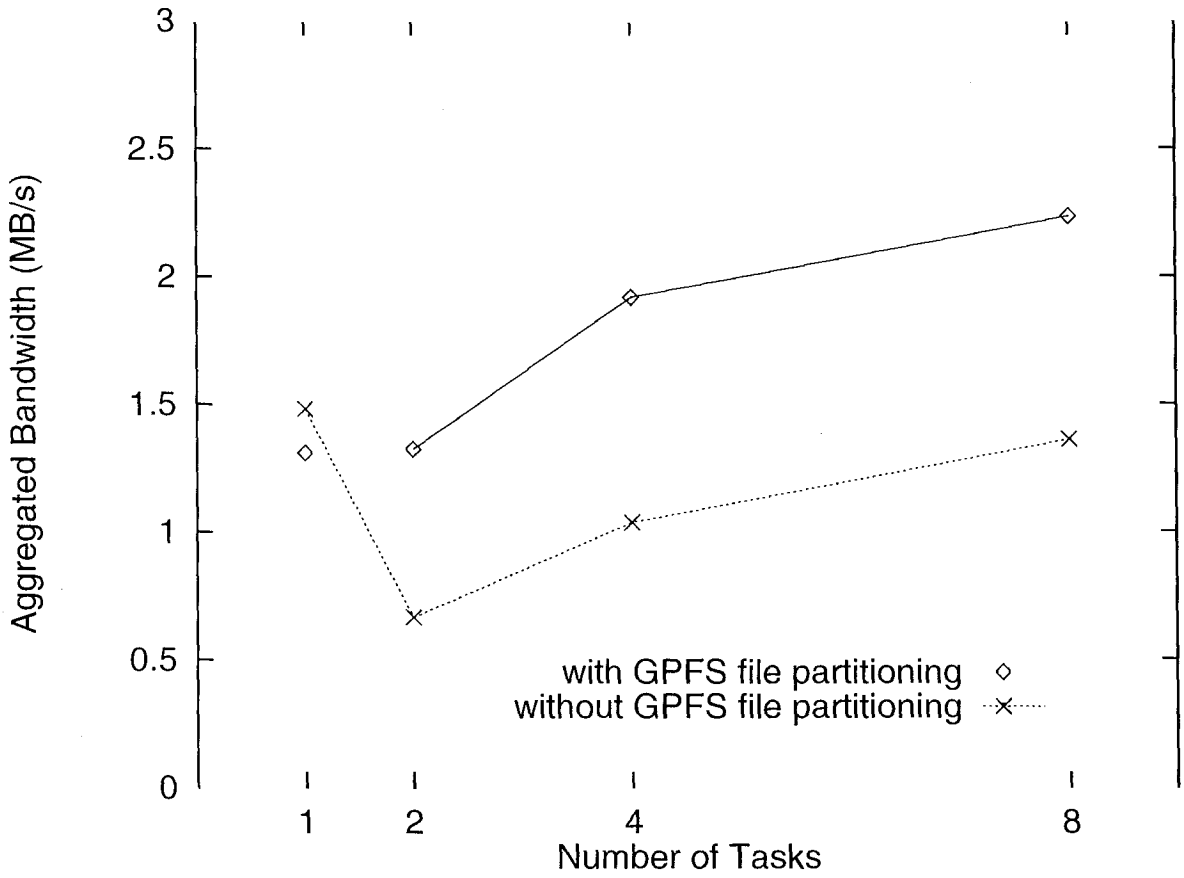

Figure 13: Averaged bandwidths measured for write operations in the GPFS file partitioning benchmark

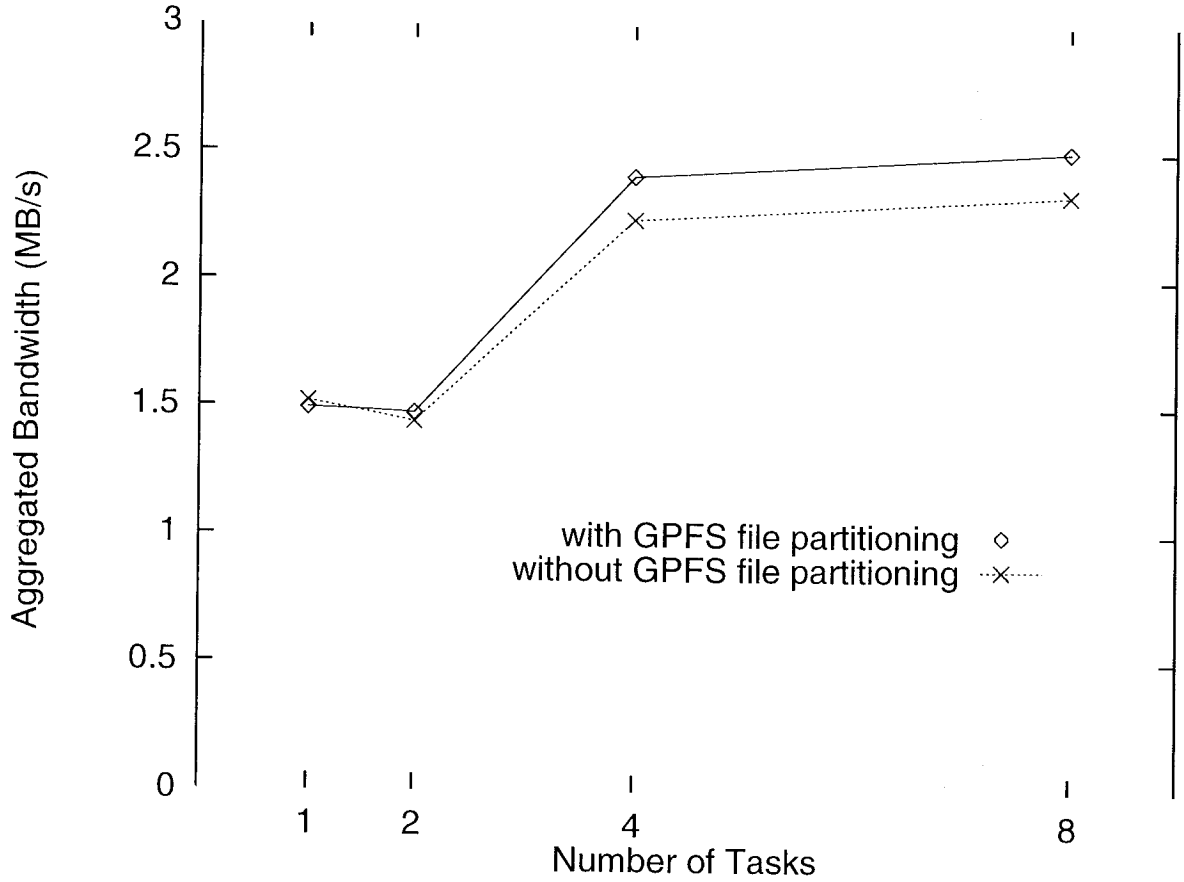

Figure 14: Averaged bandwidths measured for read operations in the GPFS file partitioning benchmark

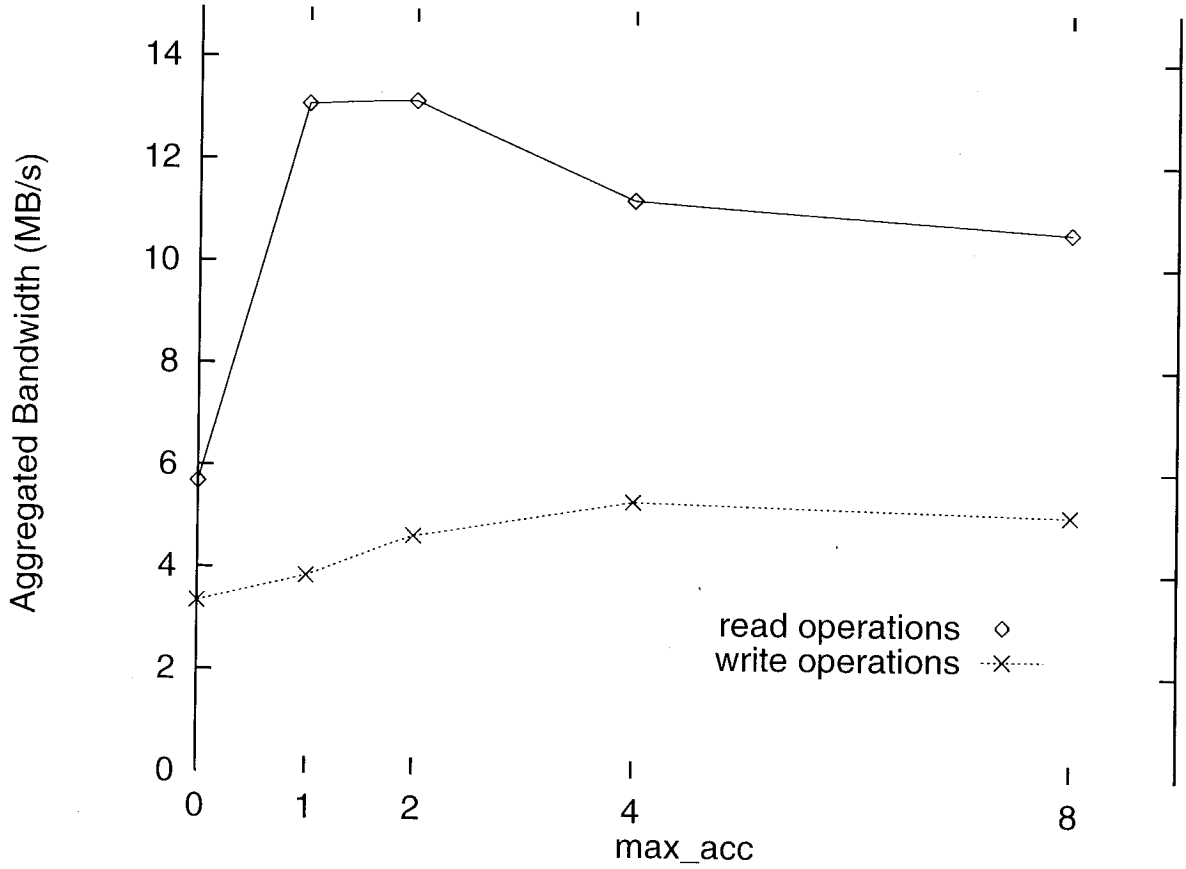

Figure 15: Averaged bandwidths measured on 8 tasks for various values of *max_acc* when each task accesses 16 4KB sub-blocks out of each of 100 GPFS file blocks

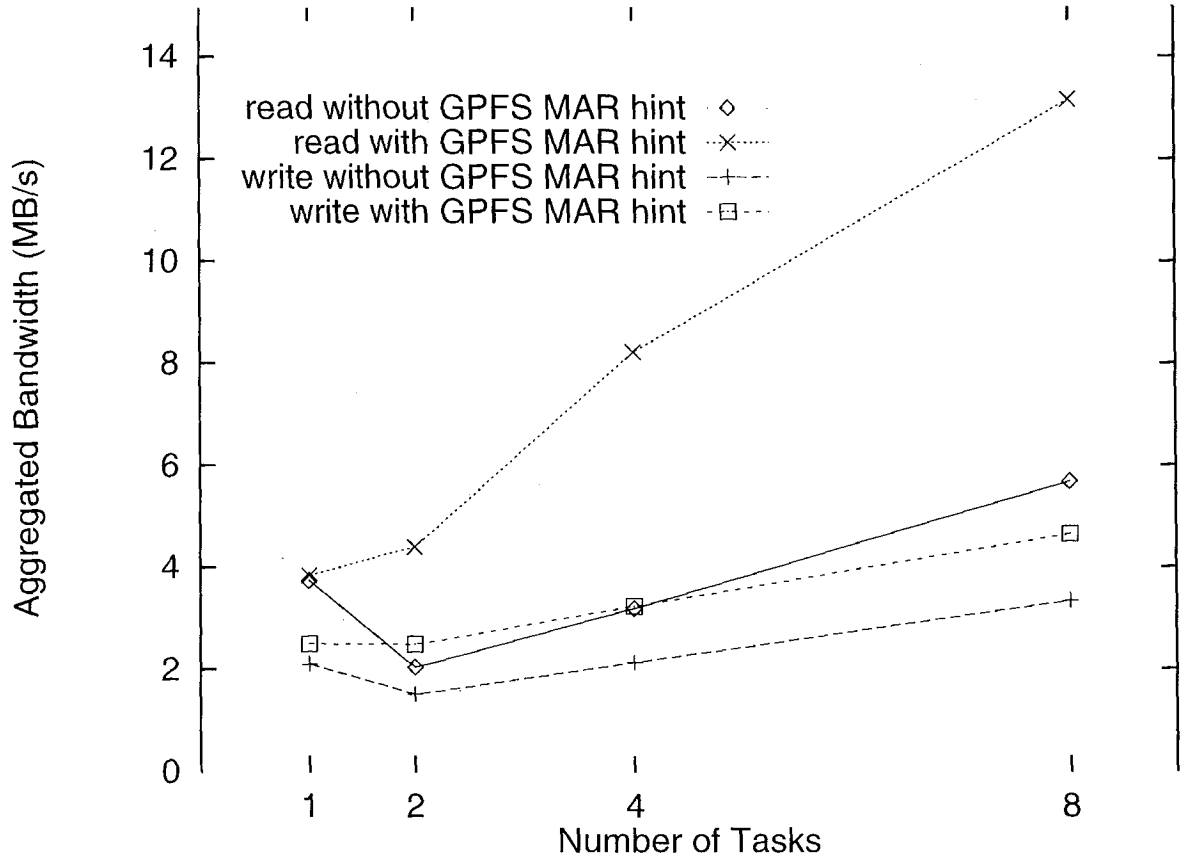

Figure 16: Averaged bandwidths measured without the use of the GPFS multiple access range hint and when a maximum of 2 GPFS blocks are accessed between two prefetching operations – each task accesses 16 4KB sub-blocks out of each of 100 GPFS file blocks