# The Parallel Effective I/O Bandwidth Benchmark: b_eff_io

**Rolf Rabenseifner*** — **Alice E. Koniges****
**Jean-Pierre Prost***** — **Richard Hedges****

* *High-Performance Computing Center (HLRS), University of Stuttgart*
*Allmandring 30, D-70550 Stuttgart, Germany*

*rabenseifner@hlrs.de*

** *Lawrence Livermore National Laboratory, Livermore, CA 94550, U.S.A.*

*{koniges, richard-hedges}@llnl.gov*

*** *IBM T.J. Watson Research Center, Yorktown Heights, NY 10598, U.S.A.*

*jpprost@us.ibm.com*

ABSTRACT. *The parallel effective I/O bandwidth benchmark (b_eff_io) is aimed at producing a characteristic average number of the I/O bandwidth achievable with parallel MPI-I/O applications exhibiting various access patterns and using various buffer lengths. It is designed so that 15 minutes should be sufficient for a first pass of all access patterns. First results of the b_eff_io benchmark are given for the IBM SP, Cray T3E, Hitachi SR 8000, and NEC SX-5 systems, and a discussion follows about problematic issues of our current approach. We show how a redesign of our time-driven approach allows for rapid benchmarking of I/O bandwidth with various compute partition sizes. Next, we present how implementation specific file hints can be enabled selectively on a per access pattern basis, and we illustrate the benefit that hints can provide using the latest version of the IBM MPI-IO/GPFS prototype.*

KEYWORDS: *I/O Performance Analysis, MPI, Parallel I/O, Disk I/O, Hints, Benchmark, Bandwidth.*

2e soumission à *Calculateurs parallèles*, le November 20, 2001.

## 1. Introduction

Crucial to the ultimate useful performance of a cluster computing environment is the seamless transfer of data between memory and a filesystem. Most cluster applications would benefit from a decent parallel filesystem that allows the transfer of data to occur using standard I/O calls such as those implemented in the MPI-2 standard MPI-I/O [MPI 97]. However, since there is a variety of data storage and access patterns that span the gamut of cluster applications, designing a benchmark to aid in the comparison of filesystem performance is a difficult task. Indeed, many parallel I/O benchmarks and benchmarking studies characterize only the hardware and file system performance limits [DET 98, HAS 98, HO 99, JON 00, KOE 98]. Often, they focus on determining under which conditions the maximal file system performance can be reached on a specific platform. Such results can guide the user in choosing an optimal access pattern for a given machine and filesystem, but do not generally consider the needs of the application over the needs of the filesystem. Other benchmarks, such as BTIO [CAR 92], are combining numerical kernels with MPI-I/O. The MPI-I/O benchmark described in [LAN 98] uses low-level benchmarks and kernel style benchmarks. This project emphasizes the development of a test suite for MPI-I/O together with I/O performance issues.

In this paper, we describe the design and implementation of a parallel I/O benchmark useful for comparing filesystem performance on a variety of architectures, including, but not limited to cluster systems.

This benchmark, referred to as the parallel effective I/O bandwidth benchmark (b_eff_io in short), is aimed at:

a) getting detailed information about several access patterns and buffer lengths,

b) measuring a characteristic average number for the I/O bandwidth achievable with parallel MPI-I/O applications.

b_eff_io examines "first write", "rewrite", and "read" accesses, strided (individual and shared pointers) and segmented collective accesses to one shared file per application, as well as non-collective access to one file per process. The number of parallel accessing processes is also varied, and wellformed I/O is compared with non-wellformed I/O. On systems meeting the rule that the total memory can be written to disk in 10 minutes, the benchmark should not need more than 15 minutes for a first pass of all access patterns.

This paper is structured as follows. In Section 2, we describe the basic design criteria that influenced our choice of appropriate I/O patterns. In Section 3, we give the specific definition of the benchmark. Results of the benchmark on a variety of platforms are described in Section 4. In Section 5, we discuss a few problematic issues related to the benchmark definition. In Section 6, we show how our time-driven approach can be applied to allow for rapid benchmarking of various compute partition sizes. In Section 7, we present how MPI-I/O file hints can be selectively enabled on a per access pattern basis in an attempt to improve the parallel I/O bandwidth achieved for that access pattern, and we illustrate this technique by describing our current experimentation using the latest IBM MPI-I/O prototype on an SP system. Finally, we mention future work and conclude the paper.

## 2. Benchmark Design Considerations

We begin by considering the common I/O patterns of parallel applications. To standardize I/O requests, the MPI Forum introduced the MPI-I/O interface [MPI 97] that allows the user to make these requests using a convenient interface similar to standard MPI calls. Furthermore, MPI-I/O allows for optimization of accesses to the underlying filesystem (see for example [DIC 98, PRO 00, THA 99, THA1]), while retaining the uniformity of the basic MPI-I/O interface across platforms.

Based on this background, the parallel effective I/O bandwidth benchmark, should measure different access patterns, report these detailed results, and should calculate an average I/O bandwidth value that characterizes the whole system. This goal is analogous to the Linpack value reported in TOP500 [TOP500] that characterizes the computational speed of a system, and also to the effective bandwidth benchmark (b_eff), that characterizes the communication network of a distributed system [RAB1, SOL 99, SOL1]. Indeed, the experience with the design of the b_eff benchmark is influential on this I/O benchmark design.

However, the major difference between b_eff and b_eff_io is the magnitude of the bandwidth. On well-balanced systems in high performance computing we expect an I/O bandwidth which allows for writing or reading the total memory is approximately 10 **minutes**. For the communication bandwidth, the b_eff benchmark shows that the total memory can be communicated in 3.2 **seconds** on a Cray T3E with 512 processors and in 13.6 seconds on a 24 processor Hitachi SR 8000. An I/O benchmark measures the bandwidth of data transfers between memory and disk. Such measurements are highly influenced by buffering mechanisms of the underlying I/O middleware and filesystem details, and high I/O bandwidth on disk requires, especially on striped filesystems, that a large amount of data be transferred between such buffers and disk. Therefore an I/O benchmark must ensure that a sufficient amount of data is transferred between disk and the application's memory. The communication benchmark b_eff can give detailed answers in about 2 minutes. Later we shall see that b_eff_io, our I/O counterpart, needs at least 15 minutes to get a first answer.

### 2.1. *Multidimensional Benchmarking Space*

Often, benchmark calculations sample only a small subspace of a multidimensional parameter space. One extreme example is to measure only one point, e.g., a communication bandwidth between two processors using a ping-pong communication pattern with 8 Mbyte messages, repeated 100 times. For I/O benchmarking, a huge number of parameters exist. We divide the parameters into 6 general categories. At the end of each category in the following list, a first hint about handling these aspects in b_eff_io is noted. The detailed definition of b_eff_io is given in section 3.

1. Application parameters are (a) the size of contiguous chunks in memory, (b) the size of contiguous chunks on disk, which may be different in the case of scatter/gather access patterns, (c) the number of such contiguous chunks that are accessed with each call to a read or write routine, (d) the file size, (e) the distribution scheme, e.g., segmented or long strides, short strides, random or regular, or separate files for each node, and (f) whether or not the chunk size and alignment are wellformed, e.g., a power of two or a multiple of the striping unit. For b_eff_io, 36 different patterns are used to cover most of these aspects.

2. Usage aspects are (a) how many processes are used and (b) how many parallel processors and threads are used for each process. To keep these aspects outside of the benchmark, b_eff_io is defined as a maximum over these aspects and one must report the usage parameters used to achieve this maximum.

3. The major programming interface parameter is specification of which I/O interface is used: Posix I/O buffered or raw, special filesystem I/O of the vendor's file system, or MPI-I/O. In this benchmark, we use only MPI-I/O, because it should be a portable interface of an optimal implementation on top of Posix I/O or the special filesystem I/O.

4. MPI-I/O defines the following orthogonal aspects: (a) access methods, i.e., first writing of a file, rewriting, or reading, (b) positioning method, i.e., explicit offsets, individual or shared file pointers, (c) coordination, i.e., accessing the file collectively by a group of processes or noncollectively, (d) synchronism, i.e., accessing the file in a blocking mode or in a nonblocking mode. Additional aspects are: (e) whether or not the files are open *unique*, i.e., the files will not be concurrently opened by other open calls, and (f) which consistency is chosen for conflicting accesses, i.e., whether or not atomic mode is set. For b_eff_io there is no overlap of I/O and computation, therefore only blocking calls are used. Because there should not be a significant difference between the efficiency of using explicit offsets or individual file pointers, only the individual and shared file pointers are benchmarked. With regard to (e) and (f), *unique* and *nonatomic* are used.

5. Filesystem parameters are (a) which filesystem is used, (b) how many nodes or processors are used as I/O servers, (c) how much memory is used as bufferspace on each application node, (d) the disk block size, (e) the striping unit size, and (f) the number of parallel striping devices that are used. These aspects are also outside the scope of b_eff_io. The chosen filesystem, its parameters, and any usage of non-default parameters must be reported.

6. Additional benchmarking aspects are (a) repetition factors, and (b) how to calculate b_eff_io, based on a subspace of the parameter space defined above using maximum, average, weighted average, or logarithmic averages.

To reduce benchmarking time to an acceptable amount, one can normally only measure I/O performance at a few grid points of a 1-5 dimensional subspace. To analyze more than 5 aspects, usually more than one subspace is examined. Often, the common area of these subspaces is chosen as the intersection of the area of best results of the other subspaces. For example in [JON 00], the subspace varying the number of servers is obtained with segmented access patterns, and with well-chosen block sizes and client:server ratios. Defining such optimal subspaces can be highly system-dependent and may therefore not be as appropriate for b_eff_io designed for a variety of systems. For the design of b_eff_io, it is important to choose the grid points based more on general application needs than on optimal system behavior.

### 2.2. *Criteria*

The benchmark b_eff_io should characterize the I/O capabilities of the system. Should we use, therefore, only access patterns, that promise a maximum bandwidth? No, but there should be a good chance that an optimized implementation of MPI-I/O should be able to achieve a high bandwidth. This means that we should measure patterns that can be recommended to application developers.

An important criterion is that the b_eff_io benchmark should only need about 10 to 15 minutes. For first measurements, it need not run on an empty system as long as concurrently running other applications do not use a significant part of the I/O bandwidth of the system. Normally, the full I/O bandwidth can be reached by using less than the total number of available processors or SMP nodes. In contrast, the communication benchmark b_eff should not require more than 2 minutes, but it must run on the whole system to compute the aggregate communication bandwidth. Based on the rule for well-balanced systems mentioned in Section 2 and assuming that MPI-I/O will attain at least 50 percent of the hardware I/O bandwidth, we expect that a 10 minute b_eff_io run can write or read about 16 % of the total memory of the benchmarked system. For this estimate, we divide the total benchmark time into three intervals based on the following access methods: initial write, rewrite, and read. However, a first test on a T3E900-512 shows that based on the pattern-mix, only about the third of this theoretical value is transferred. Finally, as a third important criterion, we want to be able to compare different common access patterns.

## 3. Definition of the Effective I/O Bandwidth

The parallel effective I/O bandwidth benchmark measures the following aspects:

– a *set of partitions*;

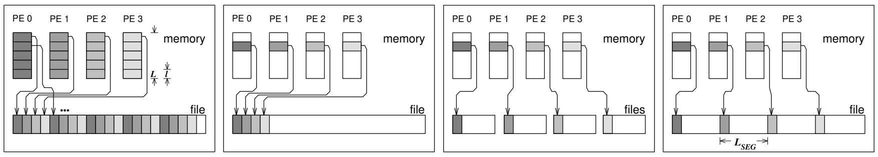
– the access methods *initial write*, *rewrite*, and *read*;

– the *pattern types* (see Fig. 1):

(0) strided collective access, scattering large chunks in memory to/from disk,

(1) strided collective access, but one read or write call per disk chunk,

(2) noncollective access to one file per MPI process, i.e., to separate files,




pattern type 0 pattern type 1 pattern type 2 pattern type 3/4

**Figure 1.** *Access patterns used in b_eff_io. Each diagram shows the data accessed by* **one** *MPI-I/O write call.*

(3) same as (2), but the individual files are assembled to one segmented file,

(4) same as (3), but the access to the segmented file is done collectively;

files are not reused across pattern types;

– the contiguous chunk size is chosen *wellformed*, i.e., as a power of 2, and *non-wellformed* by adding 8 bytes to the wellformed size;

– different chunk sizes, mainly 1 kB, 32 kB, 1 MB, and the maximum of 2 MB and $1/128$ of the memory size of a node executing one MPI process.

The entire list of access patterns is shown in Tab. 1. The column "type" refers to the pattern type. The column "$l$" defines the size of the contiguous chunks that are written from memory to disk and vice versa. The value $M_{PART}$ is defined as *max(2 MB, memory of one node / 128)*. This definition should reflect applications that write a significant part of the total memory to disk, but expecting that this is done by several I/O accesses, e.g., by writing several matrices to disk. The smaller chunk sizes should reflect the writing of fixed-sized short (1 kB), middle (32 kB) and long (1 MB) wellformed and non-wellformed (8 bytes longer) data-sets. The column "$L$" defines the size of the contiguous regions accessed at once in memory. In case of pattern type (0), non-contiguous file views are used. If $l$ is less than $L$, then in each MPI-I/O read/write call, the $L$ bytes accessed in memory are scattered/gathered to/from chunks of $l$ bytes at various locations on disk. In all other cases, a contiguous chunk written/read on disk has the same size as a contiguous region accessed in memory. This is denoted by ":=$l$" in the $L$ column. $U$ is a time unit.

Each access pattern is benchmarked by repeating the pattern for a given amount of time. This time is given by the allowed time for a whole partition (e.g., $T = 10$ minutes) multiplied by $U/\sum U/3$, as given in the table. A value of $U$=0 indicates that this pattern is executed only once to hide from the benchmark initialization effects of the subsequent patterns. The time-driven approach allows one to limit the total execution time. The total amount of data written to disk is not limited by the chunk sizes. It is only limited by $T$, the time allowed (= scheduled) for this benchmark, and by the disk-bandwidth of the system that is measured. For pattern types (3) and (4) a fixed segment size must be computed before starting the pattern of these types. Therefore, the time-driven approach is substituted by a size-driven approach, and the repeating factors are initialized based on the measurements for types (0) to (2).

| type | $l$ | | $L$ | $U$ |
|---|---|---|---|---|
| 0 | 1 MB | | 1 MB | 0 |
| | $M_{PART}$ | | :=$l$ | 4 |
| | 1 MB | | 2 MB | 4 |
| | 1 MB | | 1 MB | 4 |
| | 32 kB | | 1 MB | 2 |
| | 1 kB | | 1 MB | 2 |
| | 32 kB | +8B | 1 MB + 256B | 2 |
| | 1 kB | +8B | 1 MB + 8kB | 2 |
| | 1 MB | +8B | 1 MB + 8B | 2 |

| type | $l$ | | $L$ | $U$ |
|---|---|---|---|---|
| 1 | 1 MB | | :=$l$ | 0 |
| | $M_{PART}$ | | :=$l$ | 4 |
| | 1 MB | | :=$l$ | 2 |
| | 32 kB | | :=$l$ | 1 |
| | 1 kB | | :=$l$ | 1 |
| | 32 kB | +8B | :=$l$ | 1 |
| | 1 kB | +8B | :=$l$ | 1 |
| | 1 MB | +8B | :=$l$ | 2 |

| type | $l$ | | $L$ | $U$ |
|---|---|---|---|---|
| 2 | 1 MB | | :=$l$ | 0 |
| | $M_{PART}$ | | :=$l$ | 2 |
| | 1 MB | | :=$l$ | 2 |
| | 32 kB | | :=$l$ | 1 |
| | 1 kB | | :=$l$ | 1 |
| | 32 kB | +8B | :=$l$ | 1 |
| | 1 kB | +8B | :=$l$ | 1 |
| | 1 MB | +8B | :=$l$ | 2 |
| 3/4 | see type=2 | | | |
| | | | | $\sum U$ = 64 |

**Table 1.** *Details of access patterns used in b_eff_io.*

The b_eff_io value **of one pattern type** is defined as the total number of transferred bytes divided by the total amount of time from opening till closing the file. The b_eff_io value **of one access method** is defined as the average of all pattern types with double weighting of pattern type 0.[1] The b_eff_io value **of one partition** is defined as the average of the access methods with the weights 25 % for *initial write*, 25 % for *rewrite*, and 50 % for *read*. The b_eff_io **of a system** is defined as the maximum over any b_eff_io of a single partition of the system, measured with a scheduled execution time $T$ of at least 15 minutes. This definition permits the user of the benchmark to freely choose the usage aspects and enlarge the total filesize as desired. The minimum filesize is given by the bandwidth for an initial write multiplied by 300 sec (= 15 minutes / 3 access methods). The benchmark can be used on any platform that supports parallel MPI-I/O. On clusters of SMP nodes, the user of the benchmark can decide how many MPI processes should run on each node. The final b_eff_io value is defined as the maximum over all such usage aspects. These aspects must be reported together with the b_eff_io result. For using this benchmark to compare systems as in the TOP 500 list [TOP500] and in the TOP 500 cluster list [TFCC], more restrictive rules are necessary. They are described in Sect. 6.

1. The double weighting of pattern type 0 and the double weighting of the large chunk sizes (see $U = 2$ and 4 in Tab. 1) is used to reflect that this benchmark should measure the I/O performance of large systems in high performance computing which typically should be used with large I/O chunks or collective I/O (=pattern type 0).

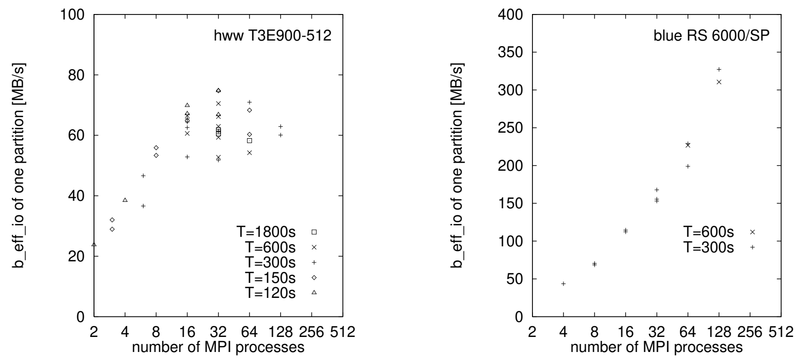



**Figure 2.** *Comparison of b_eff_io for different numbers of processes on T3E and SP, measured partially without pattern type 3.*

## 4. Comparing Systems Using b_eff_io

First, we test b_eff_io on two systems, the Cray T3E900-512 at HLRS/RUS in Stuttgart and an RS 6000/SP system at LLNL called “Blue Pacific.” The T3E is an MPP system, the IBM SP is a cluster of SMP nodes. In Sect. 4.1, we will also compare other systems from NEC, Hitachi and IBM, and in Sect. 7, we will show results on an IBM ASCI White system.

On the T3E, we use the tmp-filesystem with 10 striped Raid-disks connected via a GigaRing for the benchmark. The peak-performance of the aggregated parallel bandwidth of this hardware configuration is about 300 MB/s. The LLNL results presented here are for an SP system with 336 SMP nodes each with four 332 MHz processors. Since the I/O performance on this system does not increase significantly with the number of processors on a given node performing I/O, all test results assume a single thread on a given node is doing the I/O. Thus, a 64 processor run means 64 nodes assigned to I/O, and no requested computation by the additional 64*3 processors. On the SP system, the data is written to the IBM General Parallel File System (GPFS) called blue.llnl.gov:/g/g1 which has 20 VSD I/O servers. Recent results for this system show a maximum read performance of approximately 950MB/sec for a 128 node job, and a maximum write performance of 690MB/sec for 64 nodes [JON 00].[2] Note that these are the maximum values observed, and performance degrades when the access pattern and/or the node number is changed.

For this data on both platforms pre-releases (Rel. 0.x) of b_eff_io were used that had a different weighting of the patterns (type 0, type 1, etc). Therefore the values pre-

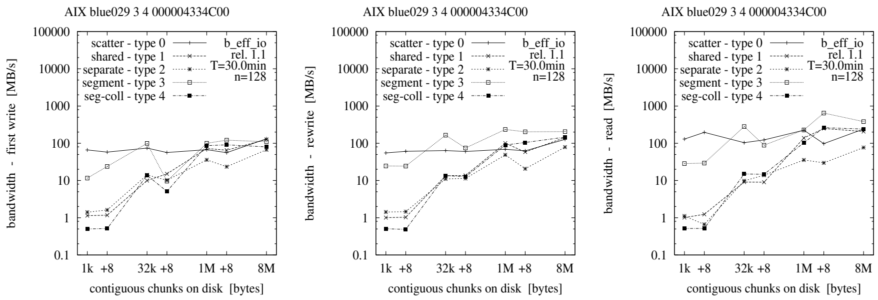
2. Upgrades to the AIX operating system and underlying GPFS software may have altered these performance numbers slightly between measurements in [JON 00] and in the current work. Additionally, continual upgrades to AIX and GPFS are bringing about improved performance overall.

sented in this section cannot be directly compared with the results in the next section. MPI-I/O was implemented with ROMIO [THA 99, THA1] but with different device drivers. On the T3E, we have modified the MPI Release mpt.1.3.0.2, by substituting the ROMIO/ADIO Unix filesystem driver routines for opening, writing, and reading files. The Posix routines were substituted by the asynchronous counter part, directly followed by the wait routine. This combination of asynchronous write (or read) and wait is not semantically identical to the Posix (synchronous) write or read. The Posix semantics does not allow on the T3E that the I/O generated by several processes can be done in parallel on several striping units, i.e., the I/O accesses are serialized. With the asynchronous counterpart, a relaxed semantics is used which allows parallel disk access from several processes [RAB3]. On the RS 6000/SP Blue Pacific machine, GPFS [GPF 00] is used underneath the MPICH version of MPI with ROMIO. Figure 2 shows the b_eff_io values for different partition sizes and different values of $T$, the time scheduled for benchmarking one partition. All measurements were taken in a non-dedicated mode.

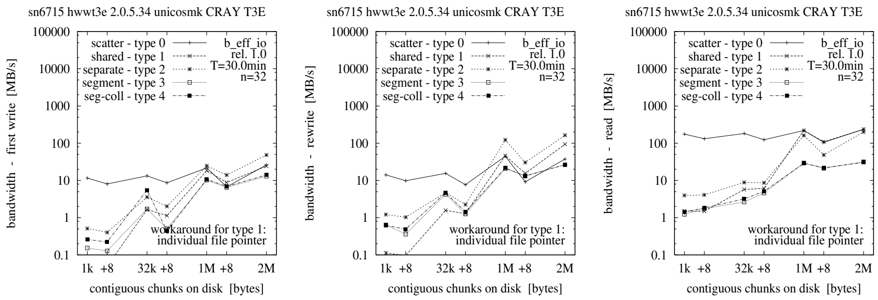
Besides the different absolute values that correlate to the amount of memory in each system, one can see very different behavior. For the T3E, the maximum is reached at 32 application processes, with little variation from 8 to 128 processes, i.e., the I/O bandwidth is a global resource. In contrast, on the IBM SP the I/O bandwidth tracks the number of compute nodes until it saturates. In general, an application only makes I/O requests for a small fraction of the compute time. On large systems, such as those at the High-Performance Computing Center in Stuttgart and the Computing Center at Lawrence Livermore National Laboratory, several applications are sharing the I/O nodes, especially during prime time usage. In this situation, I/O capabilities would not be requested by a significant proportion of the CPU's at the same time. "Hero" runs, where one application ties up the entire machine for a single calculation are rarer and generally run during non-prime time. Such hero runs can require the full I/O performance by all processors at the same time. The right-most diagram shows that the RS 6000/SP fits more to this latter usage model. Note that GPFS on the IBM SP is configurable, i.e., number of I/O servers and other tunables, and the performance on any given SP/GPFS system depends on the configuration of that system.

### 4.1. *Detailed Insight*

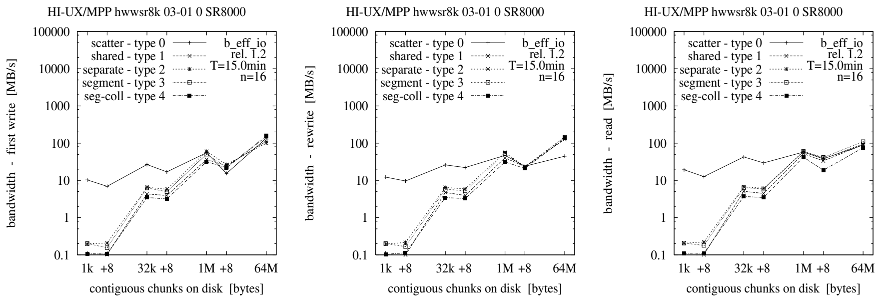
In this section, we present a detailed analysis of each run of b_eff_io on a partition. For each run of b_eff_io, the I/O bandwidth for each chunk size and pattern is reported in a table that can be plotted as in the graphs shown in each row of Figure 3. The three diagrams in each row show the bandwidth achieved for the three different access methods: writing the file the first time, rewriting the same file, and reading it. On each graph, the bandwidth is plotted on a logarithmic scale, separately for each pattern type and as a function of the chunk size. The chunk size on disk is shown on a pseudo-logarithmic scale. The points labeled "+8" are the non-wellformed counterparts of the power of two values. The maximum chunk size varies across systems because it is



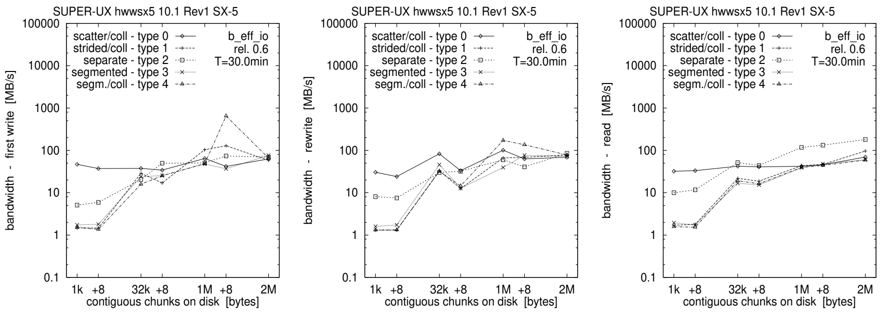

(a) 128 nodes on the "Blue Pacific" RS 6000/SP at LLNL, $T = 30$ min, b_eff_io = 63 MB/s




(b) 32 PEs on the T3E900-512 at HLRS, $T = 30$ min, b_eff_io = 57 MB/s




(c) 16 nodes on the Hitachi SR 8000 at HLRS, $T = 15$ min, b_eff_io = 41 MB/s




(d) 4 processors on the NEC SX-5Be/32M2 at HLRS, $T = 30$ min, b_eff_io = 60 MB/s

**Figure 3.** *Comparison of the results on SP, T3E, SR8000, and SX-5.*

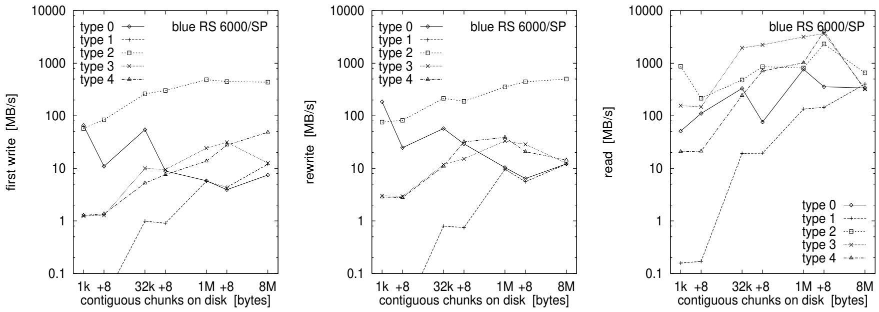



**Figure 4.** *128 nodes on the "Blue Pacific" RS 6000/SP at LLNL, with ROMIO.*

chosen to be proportional to the usable memory size per node to reflect the scaling up of applications on larger systems. On the NEC SX-5, a reduced maximum chunk size was used. Except on the NEC SX-5, we have used b_eff_io releases 1.x. On the IBM SP, a new MPI-I/O prototype was used [PRO 00]. This prototype is used for the development and improvement of MPI-I/O. Performance of the actual product may vary.

The four rows compare the I/O bandwidth on four different systems from IBM, Cray, Hitachi and NEC. The IBM SP and the Cray T3E are described in the previous section. The NEC SX-5 system at HLRS is a shared memory vector system. It is a cluster of two 16 processor SMP nodes with 32+48 GB memory, but the benchmark was done only on a single SMP node. It has four striped RAID-3 arrays DS 1200, connected by fibre channel. The NEC Supercomputer File System (SFS) is used. It is optimized for large-sized disk accesses. The filesystem parameters are: 4 MB cluster size (=block size), and if the size of an I/O request is less than 1 MB, then a 2 GB filesystem-cache is used. On the NEC SX-5, we use MPI/SX 10.1. The Hitachi SR8000 at HLRS is a 16 node system, that clusters SMP nodes each with 8 pseudo-vector CPUs and with 8 GB of memory.

First, notice that access pattern type 0 is the best on all platforms for small chunk sizes on disk. Thus all MPI-I/O implementations can effectively handle the 1 MB memory regions that are given in each MPI-I/O call to be scattered to disk or gathered from disk. In all other pattern types, the memory region size per call is identical to the disk chunk size, i.e., in the case of 1 kB or 32 kB, only a small or medium amount of data is accessed per call.

Note that due to the logarithmic scale, a vertical difference of only a few millimeters reflects an order of magnitude change. Comparing the wellformed and non-wellformed measurements, especially on the T3E, there are huge differences. Also on the T3E, we see a large gap between write and read performance in the scattering pattern type.

On the IBM SP MPI-I/O prototype, one can see that segmented non-collective pattern type 3 is also optimized. On the other hand, the collective counterpart is more than a factor of 10 worse. Such benchmarking can help to uncover advantages and weakness of an I/O implementation and can therefore help in the optimization process. Figure 4 and the first row of Figure 3 compare two different MPI-I/O implementations on top of the GPFS filesystem. ROMIO is used in the benchmarks shown in Figure 4. One can see that with ROMIO writing and reading of separate files result in best bandwidth values. IBM MPI-I/O prototype clearly shows in Figure 3 better results for access pattern type 0 and for access pattern type 3. File hints, introduced by the MPI-2 standard with the *info* argument, can be used in the IBM MPI-I/O prototype [PRO 00] in order to further optimize I/O performance for speficic access patterns. These hints are necessarily pattern specific, since if they worked for all pattern types, they would naturally be a part of the standard MPI-I/O implementation. In Section 7, we present how the next release of b_eff_io will enable the selective use of MPI-I/O implementation specific file hints on a per access pattern basis, and we illustrate the impact of using such hints by providing early results using the latest version of the IBM MPI-I/O prototype [PRO 01] on an ASCI White system.

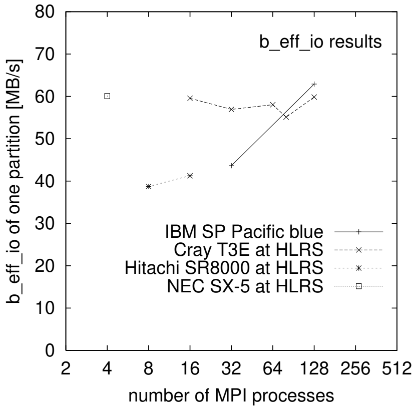



**Figure 5.** *Comparison of b_eff_io for various numbers of processes at HLRS and LLNL, measured partially without pattern type 3. b_eff_io releases 1.x were used, except for the NEC system (rel. 0.6).*

Figure 5 compares the final b_eff_io results on these four platforms. Values for other partition sizes are added. It should be noticed, that the IBM results in this figure can not be compared with the results in Figure 2, because for Figure 5, ROMIO and b_eff_io rel. 0.x are used instead of the IBM MPI-I/O prototype and b_eff_io rel. 1.1. In particular, b_eff_io rel. 0.x used an inappropriate weighting to compute the bandwidth average. Consequently, the absolute values were not comparable between different platforms. This problem was fixed in rel. 1.0. With the IBM MPI-I/O prototype, the bad effective I/O bandwidth for small partition sizes (as shown with ROMIO in Figure 2) could also be fixed.

In general, our results show that the b_eff_io benchmark is a very fast method to analyze the parallel I/O capabilities available for applications using the standardized MPI-I/O programming interface. The resulting b_eff_io value summarizes parallel I/O capabilities of a system in one significant I/O bandwidth value.

## 5. Discussion of b_eff_io

In this section, given the primary results of the benchmark, we reflect on some details of its definition. The design of the b_eff_io tries to follow the rules about MPI benchmarking defined by Gropp and Lusk [GRO 99], as well as Hempel [HEM 99], but there are a few problematic issues.

Normally, the same experiment should be **repeated** a few times to compute a **maximum bandwidth**. To achieve a very fast I/O benchmark suite, this methodology is substituted by **weighted averaging** over a medium number of experiments, i.e., the access patterns. (We note that in the case of the IBM SP data, repeated calculation of b_eff_io on different days produced nearly identical answers). The weighted averaging is done for each experiment after calculating the average bandwidth over all repetitions of the same pattern. Any maximum is calculated only after repeating the total b_eff_io benchmark itself. For this maximum, one may vary the number of MPI processes (i.e., the partition size), the schedule time $T$, and filesystem parameters.

The major problem with this definition is that one may use any schedule time $T$ with $T > 10$ minutes. First experiments on the T3E have shown that the b_eff_io value may have its maximum for $T = 10$ minutes. This is likely since for any larger time interval, the caching of the filesytem in the memory is reduced.

Indeed, **caching issues** may be problematic for I/O benchmarks in general. For example, Hempel [HEM 00] has reported that on NEC SX-5 systems other benchmark programs have reported a bandwidth significantly higher than the hardware peak performance of the disks. This is caused by a huge 4 GB memory cache used by the filesytem. In other words, the measurement is not able to guarantee that the data was actually written to disk. To help assure that data is written, we can add MPI_File_sync. The problem is, however, that MPI_File_sync influences only the consistency semantics. Calling MPI_File_sync after writing to a file guarantees that any other process can read this newly written data, but it does **not** guarantee that the data is stored on a permanent storage medium, i.e., that the data is written to disk. There is only one way to guarantee that the MPI-I/O routines have stored 95 % of the written data to disk. One must write a dataset 20 times larger than the memory cache length of the filesystem. This can be controlled by verifying that the datasize accessed by each b_eff_io access method is larger than 20 times of the filesystems' cache length.

The next problem arises from the **time driven approach** of the b_eff_io benchmark. Each access pattern is repeated for a given time interval, which is $T_{pattern} = T/3 * U/\Sigma U$ for each pattern. The termination condition must be computed after each call to a write or read routine. In all patterns defining a collective fileview or using collective write or read routines, the termination condition must be computed globally to guarantee that all processes are stopped after the same iteration. In the current version, this is done by computing the criterion only at a root process. The local clock is read after a barrier synchronization. Then, the decision is broadcasted to all other nodes. This termination algorithm is based on the assumption that a barrier followed by a broadcast is at least 10 times faster than a single read or write access. For example, the fastest access on the T3E for $L = 1$ kB regions is about 4 MB/s, i.e., 250 $\mu$s per call. In contrast, a barrier followed by a broadcast needs only about 60 $\mu$s on 32 PEs, which is not 10 times faster than a single I/O call. Therefore, this termination algorithm should be modified in future versions of our benchmark. Instead of computing the termination criterion at the end of each iteration, a geometric series of increasing repeating factors should be used: The repeating factor used in the first iteration must be small, because otherwise the execution time of the first iteration may be larger than the allowed time ($= U/\sum U/3$, see Sect. 3) on some slow platforms. After benchmarking this first iteration, the repeating factor can be increased to reduce the relative benchmarking overhead induced by the time measurements and barrier operation at the end of each iteration.

Pattern types 3 and 4 require a predefined **segment size** $L_{SEG}$ (see Figure 1). In the current version, for each chunk size "$l$", a repeating factor is calculated from the measured repeating factors of pattern types 0–2. The segment size is calculated as the sum of the chunk sizes multiplied by these repeating factors. The sum is rounded up to the next multiple of 1 MB. This algorithm has two drawbacks:

1. The alignment of the segments are multiples of 1 MB. If the striping unit is more than 1 MB, then the alignment of the segments is not wellformed.

2. On systems with **32 bit integer/int** datatype, the segment size multiplied by the number of processes ($n$) may be more than 2 GB, which may cause internal errors inside of the MPI library. Without such internal restrictions, the maximum segment size would be 16/$n$ GB, based on a 8 byte element type. If the segment size must be reduced due to these restrictions, then the total amount of data written by each process does not fit any longer into one segment.

On large MPP systems, it may also be necessary to reduce the **maximum chunk size** ($M_{PART}$) to 2/n GB or 16/n GB. This restriction is necessary for pattern types 0, 1, 3 and 4.

Another aspect is the mode used to open the benchmark files. Although we want to benchmark **unique** mode, i.e., ensure that a file is not accessed by other applications while it is open by the benchmark program, MPI_MODE_UNIQUE_OPEN must **not** be used because it would allow an MPI-I/O implementation to delay all MPI_File_sync operations until the closing of the file.

## 6. The Time-Driven Approach

Figure 2 shows interesting results. There is a difference between the maximum I/O bandwidth and the sampled bandwidth for several partition sizes. In the redesign from release 0.x to 1.x, we have incorporated that the averaging for each pattern type can not be done by using the average of the bandwidth values for all chunk sizes. The bandwidth of one pattern type must be computed as the total amount of transferred data divided by the total amount of time used for all chunk sizes. With this approach, it is possible to reduce caching effects and to allow a total scheduled time of 30 minutes for measuring all five patterns with the three access directions (write, rewrite, read) for **one** compute partition size.

The b_eff_io benchmark is proposed for the *Top 500 Clusters* list [TFCC]. For this, the I/O benchmark must be done automatically in 30 minutes for **three** different compute partition sizes. This can be implemented by reorganizing the sequence of the experiments. First, all files are written with the three different compute partition sizes, followed by rewriting, and then by all reading. Additionally, the rewriting experiments only use pattern type 0, to reduce the amount of time needed for each partition size without losing the chance to compare the *initial write* with the *rewrite* pattern. Therefore, the averaging process has also to be slightly modified. The average for writing patterns is done by weighting all 5 *initial write* patterns and the one *rewrite* pattern each with a factor of one. This implies that pattern type 0 is weighted double, as it is also done with the *read* pattern types. The b_eff_io value is then defined as the geometric mean of the writing and reading average bandwidth values. The geometric mean is used to guarantee that both bandwidth values (writing and reading) influence the final result in an appropriate way, even if one of the two values is very small compared to the other one. The arithmetic mean was not chosen, because it would always report a value larger than 50 % of the higher bandwidth value, even if the other bandwidth is extremely low.

Remembering that the b_eff_io benchmark has two goals, (a) to achieve a detailed insight into the performance of several I/O patterns, and (b) to summarize these benchmarks in one specific effective I/O bandwidth value, we offer the option to run this b_eff_io release 2.0 benchmark for a longer time period and with all rewriting patterns included. In this case, the scheduled time is enlarged to 45 minutes. The averaging process is not changed, i.e., the additional *rewriting* patterns do not count for the b_eff_io number, but with this option the benchmark can still be used to compare all *initial write* patterns with their *rewrite* counterparts.

For using b_eff_io as an additional metric in the Top 500 Clusters list, it is also necessary to define three partition sizes to get comparable results and to reduce the total time to run this benchmark on a given platform. The three partition sizes, i.e., the number of nodes of a cluster or the number of CPUs of an MPP or SMP system, are given by the following formulas:

– small = 2 ** ( round ( $log_2$(SIZE) * 0.35 ) )

– medium = 2 ** ( round ( $log_2$(SIZE) * 0.70 ) )

– large = SIZE,

with SIZE = size of MPI_COMM_WORLD and the following exceptions: small=1 if SIZE==3, and medium=3 if SIZE==4.[3] The three partition sizes should allow to analyze the behavior of the system when applications try to make I/O by using different numbers of CPUs as shown in Fig. 2 and in Sect. 4. The size of MPI_COMM_WORLD should reflect the total cluster system. If the system is used on a cluster of SMP nodes, it must be reported how many CPUs per node were used to run this benchmark.

## 7. The Influence of File Hints

The MPI-I/O interface allows for the specification of file hints to the implementation so that an optimal mapping of the application's parallel I/O requests to the underlying filesystem and to the storage devices below can be done. First benchmark tests with b_eff_io have shown a significant difference in the behavior of two different MPI-I/O implementations on top of the same filesystem. IBM's prototype implementation shows in the first row of Fig. 3 an optimum with the access pattern type 0, while ROMIO yields the best results with the access pattern type 2 (Fig. 4). IBM Research [PRO 00, PRO 01] developed an MPI-I/O prototype, referred to as MPI-IO/GPFS and using GPFS as the underlying file system, in order to investigate how file hints can be used to optimize the performance of various parallel I/O access patterns. They demonstrate important benefit of using file hints for specific access patterns and access methods (reading vs. writing).

We present here how the b_eff_io benchmark can be enhanced in order to enable the use of implementation specific MPI-I/O hints on a per access pattern type basis. We first describe our methodology. Then, we detail our on-going experimentation aimed at applying this methodology on the ASCI White RS 6000/SP system at Lawrence Livermore National Laboratory, using the latest version of IBM's MPI-IO/GPFS prototype. Preliminary results from this experimentation are then presented and demonstrate how the use of hints can lead to improved parallel I/O performance (i.e., higher b_eff_io numbers).

### 7.1. *Support for File Hints*

The support for file hints can be done with an additional options file, which specifies file hints to be set at file open time for each pattern type and each access method. Additionally, for each disk chunk size used, separate hints can be given. For some pattern type, these size dependent hints may be set together with the definition of an additional fileview.

3. The three values (35 %, 70 %, and 100 %) are motivated as a (nearly) linear splitting of the total number in the logarithmic scale. The zero value (i.e. $2^0 = 1$ CPU) is omitted, because large parallel systems are normally not dedicated for serial applications.

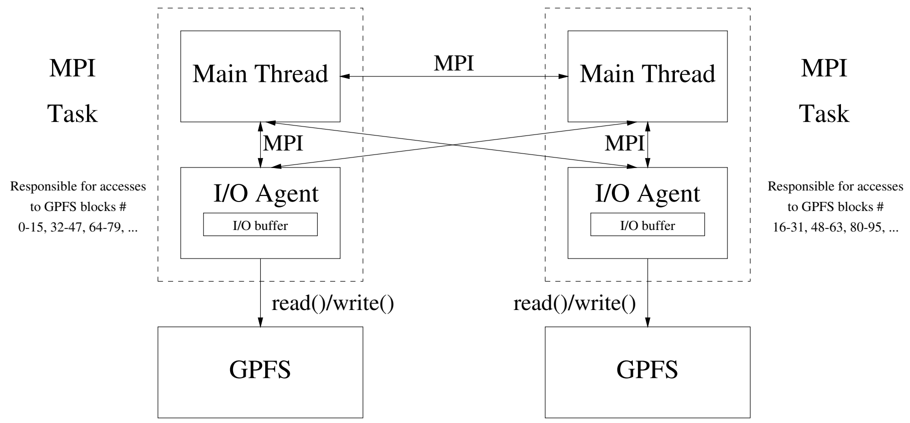



**Figure 6.** *GPFS block allocation used by MPI-IO/GPFS in data shipping mode (using the default stripe size)*

The usage of such file hints is automatically reported together with the benchmark results, allowing the user to get additional information about the optimal usage of the MPI-I/O implementation for specific access patterns, access methods, and disk chunk sizes.

### 7.2. *Experimentation with IBM's MPI-IO/GPFS Prototype*

The foundation of the design of MPI-IO/GPFS is the technique referred to as data shipping. This technique prevents GPFS file blocks to be accessed concurrently by multiple tasks, possibly residing on separate nodes. Each GPFS file block is bound to a single I/O agent, which is 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 file 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 file blocks to a set of I/O agents according to a round-robin striping scheme, illustrated in Figure 6. I/O agents are also responsible for combining data access requests issued by all participating tasks in collective data access operations.

On a per open file basis, the user can define the stripe size used in the binding scheme of GPFS blocks to I/O agents, by setting file hint *IBM_io_buffer_size*. The stripe size also controls the amount of buffer space used by each I/O agent in data acccess operations. Its default size is the number of bytes contained in 16 GPFS file blocks.

The user can enable or disable the MPI-IO data shipping feature, by setting file hint *IBM_largeblock_io* to *"false"* or *"true"*, respectively. Data shipping is enabled by default. When it is disabled, tasks issue read/write calls to GPFS directly. This saves the cost of transferring data between MPI tasks and I/O agents, but risks GPFS file block ping-ponging across nodes if tasks located on distinct nodes contend for GPFS file blocks in read-write or write-write shared mode (since GPFS performs coherent client caching across nodes). In addition, collective data access operations are implemented as noncollective operations when data shipping is disabled.

Therefore, it is recommended to disable data shipping on a file only when accesses to the file are performed in large chunks or when tasks access large disjoint partitions 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 file block access contention is not a concern.

For these reasons, it seems natural to set the *IBM_largeblock_io* hint to true if the access pattern is segmented (like for access pattern types 2, 3, and 4). For scattering access pattern types 0 and 1, it also seems natural to set *IBM_largeblock_io* to true for larger values of $l$ (e.g., 1MB and $M_{PART}$), and leave it to false for smaller values of $l$ (e.g., less than 1MB). Finally, in the latter case (pattern types 0 and 1, and smaller values of $l$), it is certainly interesting to experiment with various values for the *IBM_io_buffer_size* hint, one below the default value (e.g., a size of 4 GPFS file blocks), and one above the default value (e.g., a size of 64 GPFS file blocks). To use different file hints for the different patterns and chunk sizes, the hint value must be set to *"switchable"* on file open. The described hints must be specified when the file view is set for each pattern.

MPI-I/O-GPFS allows also to define the hint *IBM_sparse_access*, which can be set to true if the access pattern is collectively sparse (this does not apply to any pattern type used by b_eff_io); by default, the value of this hint is false. This hint is not used in the b_eff_io benchmark.

Our current experimentation is performed using IBM MPI-IO/GPFS on an ASCI White RS 6000/SP testbed system at Lawrence Livermore National Laboratory. This IBM SP system is composed of 67 SMP nodes each consisting of 16 375 MHz Power3 processors. There are 64 compute nodes, 1 login node, and 2 dedicated GPFS server nodes. Each node has 8 GB of memory. The GPFS configuration uses a GPFS file block size of 512 KB and has a pool of 100 MB on each of the nodes. It uses two VSD servers. Each VSD server serves 12 logical disks (RAID5 sets). The filesystem is comprised of 24 logical disks, of about 860 GB each. The transfer rate of the disks integrated as one system is approximately 700 MB/sec.

For these tests, a partition of 16 nodes is used with 4 MPI processes per node. b_eff_io 2.0 prototype is used. Two sets of file hints are used. The first set uses all default (IBM_largeblock_io = false and IBM_io_buffer_size = 8MB) file hint values. This is our **baseline**. The **second set** has the IBM_largeblock_io hint set to true. For pattern type 2 (*separated files*), the mode MPI_MODE_UNIQUE_OPEN is removed for both measurements. The **third set** has again the IBM_largeblock_io hint set to true and IBM_io_buffer_size hint set to the default (8 MB) for all patterns, except on the

| Access | set of hints | IBM_ large-block_ io | scatter type 0 MB/s | shared type 1 MB/s | sepa-rate type 2 MB/s | seg-mented type 3 MB/s | seg-coll type 4 MB/s | weighted average MB/s |
|---|---|---|---|---|---|---|---|---|
| write | baseline | false | 197 | 188 | 104 | 338 | 13 | 173 |
| | second | true | 284 | 277 | 435 | 541 | 524 | 391 |
| | third | mixed | 397 | 282 | 423 | 533 | 502 | 422 |
| rewrite | baseline | false | 190 | 172 | 83 | 186 | 11 | 139 |
| | second | true | 197 | 244 | 241 | 228 | 431 | 256 |
| | third | mixed | 449 | 265 | 234 | 297 | 297 | 332 |
| read | baseline | false | 122 | 130 | 96 | 408 | 11 | 148 |
| | second | true | 483 | 346 | 609 | 586 | 590 | 516 |
| | third | mixed | 441 | 282 | 579 | 554 | 544 | 473 |
| average | baseline | false | 157 | 155 | 95 | 335 | 12 | 152 |
| | second | true | 362 | 303 | 473 | 485 | 534 | 420 |
| | third | mixed | 432 | 278 | 454 | 485 | 472 | 425 |

**Table 2.** *Summary of the effect of hints on the total bandwidth on the ASCI White testbed.*

| set of hints | IBM_largeblock_io | b_eff_io MB/s |
|---|---|---|
| baseline | false (default) | 159.5 |
| second | true | 440.8 |
| third | mixed | 451.7 |

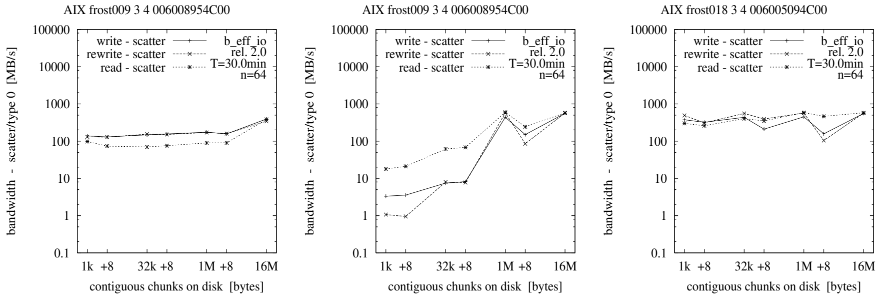
**Table 3.** *b_eff_io without and with IBM_largeblock_io hint on the ASCI White testbed.*

pattern types *scatter* and *shared* with the chunk sizes 1 kB, 1 kB + 8 bytes, 32 kB, and 32 kB + 8 bytes, where IBM_largeblock_io = false and IBM_io_buffer_size = 2 MB is used.[4] T is chosen to be 30 minutes per set of file hints.

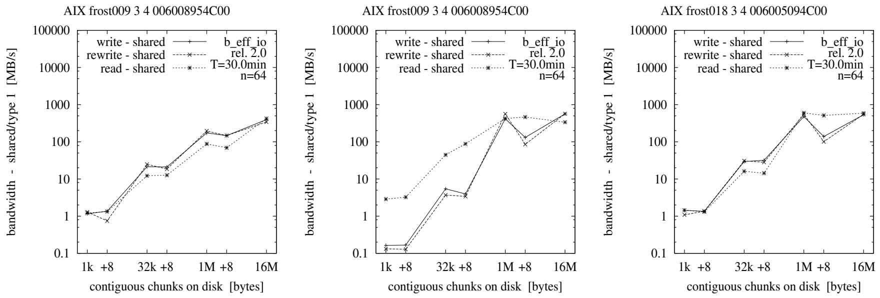
### 7.3. *Results with Hints Enabled*

The results are plotted in Figure 7, Table 2, and Table 3. In Figure 7, we can compare the baseline benchmarks (without hints) with the results for the second and third set of hints. The first row (a) compares the benchmark results for the pattern types 0 (*scatter*). We can see that the second set[5] improves the results for the large chunk sizes whereas we can see that for the chunk sizes less than 1 MB, the base-

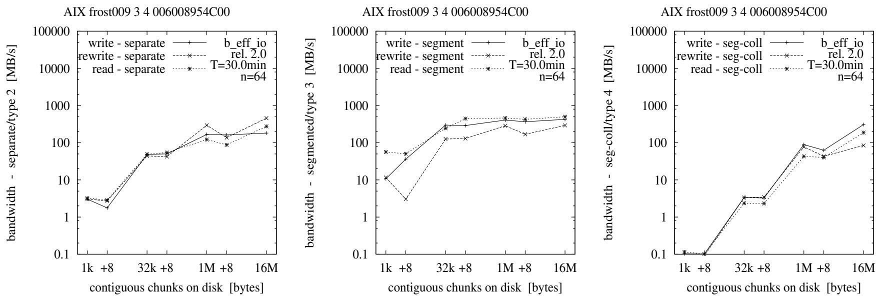
4. For this set of hints, the hints must be set to *switchable* when opening the file – it takes the default value on opening the file, and any subsequent call to MPI_FILE_SET_VIEW or MPI_FILE_SET_INFO allows to switch it to *true/false* as often as required.
5. IBM_largeblock_io = true and IBM_io_buffer_size = 8 MB (default) for all patterns.



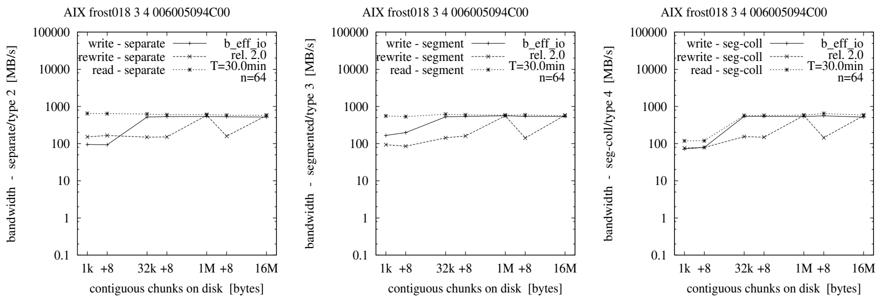

(a) Pattern types 0: baseline (left), second set of hints (middle), and third set (right).




(b) Pattern types 1: baseline (left), second set of hints (middle), and third set (right).




(c) Pattern types 2–4, without hints, i.e., IBM_largeblock_io = false.




(d) Pattern types 2–4, with third set of hints, i.e., IBM_largeblock_io = *true*.

**Figure 7.** *The effect of hints on the ASCI White RS 6000/SP testbed at LLNL.*

line results are clearly better. The right diagram uses the third set of hints. For the chunk sizes less than 1 MB, the hints are switched to IBM_largeblock_io = false and IBM_io_buffer_size = 2 MB. Comparing this set of hints with the baseline, we can see that the bandwidth could be improved for all chunk sizes, except at writing of non-wellformed chunks with 1 MB plus 8 bytes. The second row (b) shows for pattern types 1 (*shared*) that the hints give only a real advantage for 1 MB wellformed chunks. Rows (c) and (d) show the other pattern types, row (c) with the baseline and row (d) with the third set of hints.[6]The left diagram in rows (c) and (d) show pattern types 2 (*separated*). The hint improves the bandwidth in all disciplines. Looking at the *segmented* pattern type (3) in the middle diagrams one can see that the baseline could be improved by the hint mainly for short chunk sizes and read operation. In the collective case (type 4=*seg-coll*) in the right diagrams the baseline produces very poor results while the benchmark with hints enabled runs with good bandwidth. The curves clearly show that the third settings for the hints[7] have produced the best results for each chunk size.

The result for each pattern type and each access is summarized in a total bandwidth shown in Table 2. The total bandwidth is defined as the total number of transferred bytes divided by the total amount of time from opening till closing the file. The table shows, that in all cases, the hint IBM_largeblock_io = true improves the baseline result (IBM_largeblock_io = false). This means that some losses with IBM_largeblock_io = true in the area of chunk sizes less than 1 MB are more than only compensated by the advantages of this hint for larger chunk sizes. The weighted average in the last column is done as an arithmetic mean with the double weighted scatter pattern according to the time units used for the benchmarking. The average at the bottom row is done according to the formula ((write+rewrite)/2+read)/2 to guarantee that the write and rewrite patterns have not more weight than the read patterns. Looking at the 15 values representing the 5 different pattern types and the 3 access methods, the table shows that with the hint set to true, all bandwidth values are equal or larger than 197 MB/s. In the baseline results, only three values have reached this value of 197 MB/s. With this hint, the average line (bottom row) shows bandwidth values larger than 300 MB/s for all pattern types.

With the third set of hints, again the result could be improved mainly for type 0. For type 1, we can see a small reduction. The results for the second and third sets also show, that the detailed values are not exactly reproducible and that there may be a minor reduction of bandwidth due to the overhead for allowing hints to be switchable.

Table 3 shows that the b_eff_io value could be improved from 159.5 MB/s for the baseline to 440.8 MB/s if the IBM_largeblock_io hint is enabled and to 451.7 MB/s

6. The second set is omitted, because it uses the same hints as in the third set, except that they are not declared as switchable, and because it exhibits only minor differences from the results of the third set.
7. IBM_largeblock_io is set to false and IBM_io_buffer_size is reduced to 2 MB for the pattern types 0 (*scatter*) and 1 (*shared*) if the chunk size is less than 1 MB, otherwise IBM_largeblock_io = true and IBM_io_buffer_size = 8 MB.

with the third set of hints. Note, that in b_eff_io version 2.0, the definition of the final b_eff_io had to be slightly modified to allow fast benchmarking without measuring all rewrite patterns. In the new formula, b_eff_io is the geometric mean of the writing and reading average. The reading average was not changed. The writing average is the sum of the bandwidth values of *all write* pattern types plus the *rewrite scatter* pattern type, divided by 6. This means, that in the reading and in the writing average, the *scatter* pattern type is double weighted, in the first case by using the *write* and the *rewrite* results, in the second case by counting the *read-scatter* result twice in the average.[8]

This section has shown, that hints can significantly improve the parallel I/O bandwidth. This benchmark can be used to test the hints and can assist the process of defining an optimal default set of hints.

## 8. Future Work

We plan to use this benchmark to compare several additional systems. In a next stage, cluster type supercomputers should be compared with Beowulf type systems, built with off-the-shelf components and operated with a parallel filesystem, e.g., with the Parallel Virtual File System (PVFS) [CAR 00].

Although [CRA 95] stated that "the majority of the request patterns are sequential", we should examine whether random access patterns can be included into the b_eff_io benchmark. The b_eff_io benchmark was mainly developed to benchmark parallel I/O bandwidth, but it may be desirable to include also a shared data access, i.e., reading the same data from disk into memory locations on several processes; this pattern type is more like checking whether the optimization of the parallel MPI-I/O can detect such a pattern and can implements it by a single reading from disk (e.g., divided into several portions accessed by several processes) and then broadcasting the information to the other processes.[9]

The benchmark will also be enhanced to write an additional output that can be used in the SKaMPI *comparison page* [REU 98]. This combination should allow to use the same web interface for reporting I/O bandwidth (based on this b_eff_io) together with communication bandwidth (based on SKaMPI benchmark or on the parallel effective communication benchmark b_eff [RAB1, RAB 01]).

## 9. Conclusion

In this paper, we have described in detail b_eff_io, the parallel effective I/O bandwidth benchmark. We used this benchmark to characterize the I/O performance of

8. The value according to b_eff_io version 1.x is computed in the lower right corner of Table 2.
9. This type of a *shared* access pattern should not be confused with the *shared* filepointer used in pattern type 1.

common computing platforms. We have shown how this benchmark can provide both detailed insight into the I/O performance of high-performance platforms and how this data can be reduced to a single number averaging important information about that system's accumulated I/O bandwidth. We gave suggestions for interpreting and improving the benchmark, and for testing the benchmark on one's own system. We also showed how the power of MPI-I/O file hints can be taken advantage of on a selective basis in the benchmark. Although we have used the benchmark on small clusters with Linux and other operating systems, we typically find that the parallel filesystems on these clusters are not yet sufficiently advanced to support the full MPI-I/O standard efficiently. We expect that benchmarks such as the one described in this paper will help to spur the design of these systems and that the situation should remedy itself soon. The b_eff_io benchmark can be used together with the Linpack benchmark [TOP500] and the effective communication bandwidth benchmark b_eff [KON 01] to measure the balance of the accumulated computational speed (Linpack $R_{max}$), the accumulated communication bandwidth (b_eff) and the total I/O bandwidth (b_eff_io).

Acknowledgements

The authors would like to acknowledge their colleagues and all the people that supported this project with suggestions and helpful discussions. Work at LLNL was performed under the auspices of the U.S. Department of Energy by University of California Lawrence Livermore National Laboratory under contract No. W-7405-Eng-48.

## 10. References

[CAR 00] CARNS P. H., LIGON W. B. III, ROSS R. B., and THAKUR R., "PVFS: A Parallel File System For Linux Clusters", *Proceedings of the 4th Annual Linux Showcase and Conference*, Atlanta, GA, October 2000, p. 317–327, `http://parlweb.parl.clemson.edu/pvfs/`.

[CAR 92] CARTER R., CIOTTI R., FINEBERG S., and NITZBERG W., "NHT-1 I/O Benchmarks", *Technical Report RND-92-016*, NAS Systems Division, NASA Ames, November 1992.

[CRA 95] CRANDALL P., AYDT R., CHIEN A., and REED D., "Input-Output Characteristics of Scalable Parallel Applications", *Proceedings of Supercomputing '95*, ACM Press, Dec. 1995, `www.supercomp.org/sc95/proceedings/`.

[DET 98] DETERT U., "High-Performance I/O on Cray T3E", *40th Cray User Group Conference*, June 1998.

[DIC 98] DICKENS P. M., "A Performance Study of Two-Phase I/O", *Proceedings of Euro-Par '98, Parallel Processing*, Southampton, UK, 1998, D. Pritchard, J. Reeve Eds., LNCS 1470, p. 959–965.

[GPF 00] *IBM General Parallel File System for AIX: Installation and Administration Guide*, IBM Document SA22-7278-03, July 2000.

[GRO 99] GROPP W. and LUSK E., "Reproducible Measurement of MPI Performance Characteristics", *Recent Advances in Parallel Virtual Machine and Message Passing Interface, Proceedings of EuroPVM/MPI '99*, Barcelona, Spain, Sep. 26–29, 1999, J. Dongarra et al. Eds., LNCS 1697, p. 11–18, `www.mcs.anl.gov/mpi/mpptest/hownot.html`.

[HAS 98] HAAS P.W., "Scalability and Performance of Distributed I/O on Massively Parallel Processors", *40th Cray User Group Conference*, June 1998.

[HEM 99] HEMPEL R., "Basic Message Passing Benchmarks, Methodology and Pitfalls", *SPEC Workshop on Benchmarking Parallel and High-Performance Computing Systems*, Wuppertal, Germany, Sep. 13, 1999, `www.hlrs.de/mpi/b_eff/hempel_wuppertal.ppt`.

[HEM 00] HEMPEL R. and RITZDORF H., "MPI/SX for Multi-Node SX-5", *SX-5 Programming Workshop*, High-Performance Computing Center, University of Stuttgart, Germany, Feb. 14–17, 2000, `www.hlrs.de/news/events/2000/sx5.html`.

[HO 99] HO R. S. C., HWANG K., and JIN H., "Single I/O Space for Scalable Cluster Computing", *Proceedings of 1st IEEE International Workshop on Cluster Computing (IWCC '99)*, Melbourne, Australia, Dec. 2-3, 1999, p. 158–166, `http://andy.usc.edu/papers/IWCC'99.pdf`.

[JON 00] JONES T., KONIGES A. E., and YATES R. K., "Performance of the IBM General Parallel File System", *Proceedings of the International Parallel and Distributed Processing Symposium*, May 2000. Also available as UCRL JC135828.

[KOE 98] KOENINGER K., "Performance Tips for GigaRing Disk I/O", *40th Cray User Group Conference*, June 1998.

[KON 01] KONIGES A. E., RABENSEIFNERR., and SOLCHENBACH K., "Benchmark Design for Characterization of Balanced High-Performance Architectures", *Proceedings of the 15th International Parallel and Distributed Processing Symposium (IPDPS '01), Workshop on Massively Parallel Processing (WMPP)*, San Francisco, CA, Apr. 23–27, 2001, IEEE Computer Society Press.

[LAN 98] LANCASTER D., ADDISON C., and OLIVER T., "A Parallel I/O Test Suite", *Recent Advances in Parallel Virtual Machine and Message Passing Interface, Proceedings of EuroPVM/MPI '98*, Liverpool, UK, Sep. 1998, V. Alexandrov, J. Dongarra Eds., LNCS 1497, p. 36–44, 1998, `http://users.wmin.ac.uk/~lancasd/MPI-IO/mpi-io.html`.

[MPI 97] MESSAGE PASSING INTERFACE FORUM, *MPI-2: Extensions to the Message-Passing Interface*, July 1997, `www.mpi-forum.org`.

[PRO 00] PROST J.-P.,TREUMANN R., HEDGES R., JIA B., KONIGES A. E., and WHITE A., "Towards a High-Performance Implementation of MPI-IO on top of GPFS", *Proceedings of Euro-Par 2000*, Munich, Germany, Aug. 29 - Sep. 1, 2000.

[PRO 01] PROST J.-P., TREUMANN R., HEDGES R., JIA B., and KONIGES A. E., "MPI-IO/GPFS, an Optimized Implementation of MPI-IO on top of GPFS", *Proceedings of Supercomputing '01*, Denver, CO, Nov. 11–16, 2001.

[RAB 01] RABENSEIFNER R., and KONIGES A.E., "Effective Communication and File-I/O Bandwidth Benchmarks", *Recent Advances in Parallel Virtual Machine and Message Passing Interface, Proceedings of EuroPVM/MPI 2001*, Y. Cotronis, J. Dongarra Eds., LNCS 2131, p. 24–35, 2001.

[RAB1] RABENSEIFNER R., *Effective Bandwidth (b_eff) Benchmark*, `www.hlrs.de/mpi/b_eff/`.

[RAB2] RABENSEIFNER R., *Effective I/O Bandwidth (b_eff_io) Benchmark*, www.hlrs.de/mpi/b_eff_io/.

[RAB3] RABENSEIFNER R., *Striped MPI-I/O with mpt.1.3.0.1*, www.hlrs.de/mpi/mpi_t3e.html#StripedIO.

[REU 98] REUSSNER R., SANDERS P., PRECHELT L., and MÜLLER M., "SKaMPI: A detailed, accurate MPI benchmark", *Recent Advances in Parallel Virtual Machine and Message Passing Interface, Proceedings of EuroPVM/MPI '98*, 1998, LNCS 1497, p. 52–59, 1998, wwwipd.ira.uka.de/∼skampi/

[SOL 99] SOLCHENBACH K., "Benchmarking the Balance of Parallel Computers", *SPEC Workshop on Benchmarking Parallel and High-Performance Computing Systems*, Wuppertal, Germany, Sep. 13, 1999,

[SOL1] SOLCHENBACH K., PLUM H.-J., and RITZENHOEFER G., *Pallas Effective Bandwidth Benchmark – Source Code and Sample Results*, ftp://ftp.pallas.de/pub/PALLAS/PMB/EFF_BW.tar.gz.

[THA 99] THAKUR R., GROPP W., and LUSK E., "On Implementing MPI-IO Portably and with High Performance", *Proceedings of the Sixth Workshop on I/O in Parallel and Distributed Systems (IOPADS '99)*, May 1999, p. 23–32.

[THA1] THAKUR R., LUSK E., and GROPP W., *ROMIO: A High-Performance, Portable MPI-IO Implementation*, www.mcs.anl.gov/romio/.

[TFCC] *TFCC – IEEE Task Force on Cluster Computing*, www.ieeetfcc.org, and *Top Clusters*, www.TopClusters.org.

[TOP500] UNIVERSITIES OF MANNHEIM AND TENNESSEE, *TOP500 Supercomputer Sites*, www.top500.org.