UCRL-JC-153187

LAWRENCE
LIVERMORE
NATIONAL
LABORATORY

The Performance of Parallel Disk Write Methods for Linux Multiprocessor Nodes

G. D. Benson, K. Long, and P. S. Pacheco

May 7, 2003

105h European Parallel Virtual Machine/Message Passing Interface Conference, Venice, Italy, September 29 – October 3, 2003

The Performance of Parallel Disk Write Methods for Linux Multiprocessor Nodes[⋆]

Gregory D. Benson, Kai Long, and Peter S. Pacheco

Keck Cluster Research Group
Department of Computer Science
University of San Francisco
2130 Fulton Street, San Francisco, CA 94117-1080
{benson,klong,peter}@cs.usfca.edu

Abstract. Despite increasing attention paid to parallel I/O and the introduction of MPI-IO, there is limited, practical data to help guide a programmer in the choice of a good parallel write strategy in the absence of a parallel file system. In this study we experimentally evaluate several methods for implementing parallel computations that interleave a significant number of contiguous or strided writes to a local disk on Linux-based multiprocessor nodes. Using synthetic benchmark programs written with MPI and Pthreads, we have acquired detailed performance data for different application characteristics of programs running on dual processor nodes. In general, our results show that programs that perform a significant amount of I/O relative to pure computation benefit greatly from the use of threads, while programs that perform relatively little I/O obtain excellent results using only MPI. For a pure MPI approach, we have found that it is best to use two writing processes with `mmap()`. For Pthreads it is usually best to use `write()` for contiguous data and `writev()` for strided data. Codes that use `mmap()` tend to benefit from periodic syncs of the data of the data to the disk, while codes that use `write()` or `writev()` tend to have better performance with few syncs. A straightforward use of ROMIO usually does not perform as well as these direct approaches for writing to the local disk.

[⋆] **Disclaimer.** This document was prepared as an account of work sponsored by an agency of the United States Government. Neither the United States Government nor the University of California nor any of their employees, makes any warranty, express or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise, does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government or the University of California. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States Government or the University of California, and shall not be used for advertising or product endorsement purposes.

1 Introduction

Many parallel simulation problems require a large number of writes to disk. Almost any time-dependent differential equation solver will interleave computation with disk writes [9]. Examples are climate modeling, simulation of groundwater flow, and circuit simulation. Such simulations require frequent writes to disk because core memory must be used for support of the computation or simply too much data is generated to fit into an in-core buffer. The resulting data files can be used for simulation analysis and visualization. For example, we have developed a parallel program called NeuroSys [6] that, given a small set of initial conditions, will simulate the behavior of a large network of biologically accurate neurons. The output data consists of numerical data, such as membrane voltages, that represent the state of each neuron. A moderate sized simulation can result in several gigabytes of data.

This study investigates low-level, parallel disk write techniques for Linux clusters. In particular, we focus on methods that optimize the usage of dual-processor compute nodes with dedicated disks. We are interested in finding the most efficient methods for pushing large amounts of data onto the dedicated disks. Our goal is to establish the best write methods for a wide range of applications. Our results can be used to guide both application programmers and systems programmers. In addition, the methods we study can be employed in parallel programs written using Pthreads, MPI, and OpenMP.

We have developed a synthetic benchmarking program that allows us to experimentally evaluate the performance of different methods given different application characteristics. The methods we evaluate consist of matching different Linux write mechanisms, such as `write()`, `writev()`, `mmap()`, and file syncing, with different strategies for parallelizing computation and I/O among multiple processes or threads on a multiprocessor node. For example, a straightforward, but sometimes inefficient strategy is to have two independent threads or MPI processes perform both computation and I/O on behalf of the simulation. It turns out that on a node with a single disk, uncoordinated file sync requests can compete for the disk and result in poor disk utilization. We also give preliminary results for the MPI-IO function, `MPI_File_write_all()` [3].

We also consider different simulation characteristics such as the ratio of I/O to computation on each simulation step. If a simulation may produce very little output per simulation step, it is more important to ensure a balance in computation among the threads or processes and it is unlikely that the disk write strategy will greatly impact application performance as long as the OS can buffer write requests and issue the disk writes asynchronously. However, as the output per simulation step increases it becomes less likely that the OS will be able to buffer all of the write requests, and it will be required to push the buffers to the disk. In this case, the write mechanism usage can determine how and when buffers are sync to disk.

Another application characteristic we consider is contiguous writes versus strided writes. Scientific simulations will often use strided writes to extract significant values from large vectors. The `write()` system call is well suited for contiguous writes, but not for strided writes unless memory copying is used within the application. Both `writev()` and `mmap()` files accommodate strided data. In MPI-IO, derived datatypes can be used to identify the strided data.

In our study we use a typical hardware and software configuration: However, our testbed configuration represents a typical Linux cluster installation. a Linux cluster of nodes each with dual Pentium III 1GHz processors and an 18GB SCSI disk. Our testbed nodes run Linux kernel version 2.4.20 and we use the ext3 file system. For the benchmarks using MPI, we use MPICH-GM.1.2.5..8 and shared memory communication. For MPI-IO we use ROMIO 1.2.5.1. For parallel applications we assume that processes or threads running on a single node will write directly to the local disk. Such applications can benefit from the inherent I/O parallelism and achieve better communication bandwidth by eliminating I/O traffic on the network. We have not experimented with other I/O configurations such as NFS mounted file systems or parallel file systems such as PVFS [2] backed by one or more dedicated I/O nodes.

Our results show that when a program performs a large amount of I/O relative to computation, its performance can be substantially improved if it uses Pthreads. This result is in accordance with previous research in which threads are used to support asynchronous I/O [4]. However, programs with relatively little I/O obtain excellent performance using only MPI. Indeed, such programs may actually obtain better performance than a threaded program. For pure MPI programs, it is usually better to have both processes writing to the disk, and using the system call `mmap()` instead of `write()` or `writev()` usually results in better performance. On the other hand, programs that use Pthreads — especially programs with a large amount of I/O — often obtain the best results with `write()` or `writev()`. Furthermore, for programs that use Pthreads the relative performance of one and two writers can depend heavily on both the amount of I/O relative to computation and the layout of the data in memory. Programs that use `mmap()` instead of `write()` or `writev()` perform best if data is regularly synced to disk. Programs that use `write()` or `writev()` usually obtain little benefit from periodic syncing. If consistent performance is more important than raw speed, it is often better to use `mmap()` rather `write()` or `writev()`. A straightforward use of the ROMIO [8] implementation MPI-IO does not perform as well as the direct approaches for writing to the local disk.

This work was motived by a practical need to implement efficient parallel disk writing in a locally developed scientific application, NeuroSys [6]. We found very little guidance in the research literature to help us optimize its I/O. Ideally, we would have liked to find experience summaries on a Linux cluster similar to ours (a cluster of 64 dual-processor nodes connected by Myrinet). A notable parallel I/O benchmark is b_eff_io [7], which is used to characterize the total parallel I/O performance of a parallel system. Unlike our benchmarks, which serve to identify an optimal disk write method for a specific system, b_eff_io is suited for comparing the I/O bandwidth of different systems. Furthermore, b_eff_io does not simulate computation, so while it can give an indication of the raw I/O throughput of a system, it does not reveal how well a system can overlap I/O with computation.

Our work is complementary to work in parallel file systems [2] and MPI-IO implementations [8]. The results of our parallel write experiments can be used to help guide in implementation of these I/O systems on Linux clusters. In addition, our benchmarks could be used as a framework for comparing the write performance of these I/O systems. However, earlier results [5] indicate that performance gains from mixing MPI with explicit threading is not worth the added programming complexity when the application involves primarily in core sparse-matrix computations. Our results show that for I/O intensive applications mixing threads with MPI is well worth the added complexity.

The rest of this paper is organized as follows. Section 2 describes the write methods we analyze in this paper. Section 3 describes our benchmarking approach. Section 4 evaluates our results and provides recommendations for application and system programmers who implement parallel I/O libraries. Finally, Section 5 makes some concluding remarks and discusses future work.

Acknowledgements. The first and third author thank the W.M. Keck Foundation for a grant to build a cluster for use in research and education and partial support of this research. The work of the third author was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract W-7405-ENG-48.

2 Parallel Write Methods

We refer to a *parallel write method* as a write mechanism combined with a parallelization strategy.

Disk Write Mechanisms Linux and most UNIX implementations support disk writing with the `write()` and `writev()` systems calls. Linux also supports memory mapped files with the `mmap()` system call. Both `write()` and `writev()` collect data from user space and transfer it to a kernel-space buffer. The buffer is usually written to the disk at a later time depending on system behavior and configuration. While `write()` supports the transfer of a single contiguous buffer, `writev()` allows a program to transfer a set of non-contiguous buffers. Using `mmap()` a program can establish a range of virtual memory in process address space to directly map to a named file. Thus writing to the file is as simple as writing directly to memory. The caching of memory mapped files is similar to ordinary file caching.

Programmers can affect when the kernel actually writes data to disk by "syncing" the data associated with a file. For writing with `write()` and `writev()` the `fsync()` system call can be used. On execution, `fsync()` blocks until all in-memory parts of a file are written to disk, including the file metadata. The `fdatasync()` system call is supposed to sync just the file data to disk, but as of Linux 2.4.20, `fdatasync()` is the same as `fsync()`, therefore introducing unnecessary disk writes. For `mmap()` regions, the `msync()` system call can be used. By syncing data to disk, it may be possible to achieve better disk write control and enable overlapping of computation with I/O. In a system with enough main memory, an application may write all its data to kernel buffers and only after the buffers have aged long enough or come under memory pressure, will the data be written to disk. Such behavior can result in large delays at the end of the program.

The `mmap()` system call can be used in two ways to write data to a file. The most straightforward technique is to use `mmap()` to map the entire length of the file into the user's virtual address space. In this mode, the user uses `mmap()` once and simply writes to contiguous memory locations. However, this approach limits the maximum file length to the amount of mappable virtual memory, which is just under 2 GB in Linux (for a default kernel configuration in Linux 2.4.20 and on a 32 bit x86 architecture). An alternate technique for writing files with `mmap()` is to map only a portion of the file into virtual memory. Once this *window* has been written to, it is unmapped with `munmap()`. The next window is mapped with `mmap()`. This approach does not place any limits on the maximum file size and it gives the programmer more control over the amount of memory used for file buffering.

The data to be written to a file may be *contiguous* or *strided*. Writing contiguous data is more straightforward than writing strided data. For example, using `write()` on a contiguous buffer simply requires the starting address and the length of the buffer to be written. Using `write()` on strided data requires several small writes or the non-contiguous data must be copied to an intermediate buffer. Alternatively, `writev()` can by used to directly write strided data. For `mmap()`, strided data must be copied into the mmapped memory region.

Parallelization Strategies When assigning work to a dual processor node, the goal is to best utilize the processors for both the computation and I/O operations. Such utilization can be achieved with separate threads using Pthreads or separate MPI processes. For example, if very little data needs to be written to disk, then it makes sense to divide the computation between two threads or two processes. However, as an application writes more data to disk relative to the amount of computation it becomes less clear how the work should be divided. One consideration is whether to write two individual files, one for each thread, or have the threads combine the output data and write to a single file. The latter case may work better on nodes with a single disk. The parallelization strategies used for our study are listed in Table 1.

3 Experimental Methodology

We have developed a set of synthetic benchmarks that employ the disk write implementations given in Section 2. In this discussion the term "thread" is used to mean either Pthread or MPI process. The programs take several parameters: **iters**, **comptime**, **bufsize**, **syncsize**, and **winsize**. The **iters** parameter determines how many loop iterations to simulate. Each iteration represents a possible iteration in a real simulation. The **comptime** parameter determines how much total computation time should be simulated across all loop **iters**. The computa-

MPI Implementations	
S1A	Two MPI processes compute and write to their own files. Each process performs periodic syncs to disk.
S1B	Similar to S1A except the periodic syncs are coordinated so that only one sync is issued at a time to the kernel.
S2	Two MPI processes compute. One of the processes is a designated writer. The non-writing process sends its data to the writing process on each iteration. The writer issues periodic syncs.
S2DT	Similar to S2 except that derived datatypes are used to send strided data from the non-writer to the writer.
S3	Two MPI processes. One computes and one writes. The compute process sends data to the write process on each iteration.
S3DT	Similar to S3 except that derived datatypes are used to send strided data from the compute process to the write process.
MPI-IO	Two MPI processes compute and write using the `MPI_File_write_all()` function. Derived datatypes are used for strided data.

Pthread Implementations	
S1A	Two threads compute and write their own files. Each thread performs periodic syncs to disk.
S1B	Similar to S1A except the periodic syncs are coordinated so that only one sync is issued at a time to the kernel.
S2	Two threads compute. One of the threads is the designated writer. The non-writing thread passes a buffer to the writing thread on each iteration. The writer issues periodic syncs.
S3	Two threads. One computes and one writes. The compute thread passes a buffer to the write thread on each iteration.
S5	Similar to S2 except a separate sync thread is used. The writing thread notifies the sync thread when the sync size is reached. The sync thread allows disk flushing to occur asynchronously with the computation.

Table 1. Implementation Details for the Parallel Write Methods

tion time per loop is **comptime / iters**. If there are two compute threads, the compute time per iteration is further divided between the two threads. If there are two write threads, the **bufsize** parameter is the amount of data to be written on each loop iteration by each thread. If there is only one write thread, then 2 × **bufsize** bytes are written by this thread on each iteration. Similarly, if there are two compute threads, each generates **bufsize** bytes of data for transferring to disk on each iteration, while if there is only one write thread, it will generate 2 × **bufsize** bytes. Thus, the total amount of data written to disk is 2 × **iters** × **bufsize**. Data is synced to disk every **syncsize** bytes if at all. Finally, **winsize** is used with `mmap()` to determine the maximum size of an mmap region. Note that **syncsize** should be less than or equal to **winsize.** For the strided benchmarks two additional parameters are used: **len** and **stride**. The **len** is the number of bytes in a contiguous chunk of data. The **stride** is the number of bytes between the beginnings of successive chunks.

In the benchmarks, the compute time is generated by filling the compute buffer and having the process spin on the `gettimeofday()` system call. In current versions of Linux, this function has microsecond resolution, which is more than adequate for our purposes.

Each synthetic benchmark follows a common template: (1) Create threads or processes; (2) Warm up disk with writes; (3) Barrier; (4) Loop of (4a) Simulate computation and fill buffer, (4b) Barrier, (4c) Write buffer or transfer buffer, (4d) Possible sync; and (5) A final sync to ensure all data is flushed to disk. To record times, we start timing after (3) and end timing after (5).

4 Experimental Results

Using the benchmarks described in Section 3 we have obtained performance results for each parallel write method. We use cluster nodes each with dual Pentium III 1GHz processors and an 18 GB SCSI disk. Our testbed nodes run Linux kernel version 2.4.20 and we use the ext3 file system. For the benchmarks using MPI, we use MPICH-GM.1.2.5..8 and shared memory communication. For MPI-IO we use ROMIO 1.2.5.1.

We have chosen parameters that are representative of realistic computations. One of our main goals is to explore the impact of computation time relative to I/O time on each of the parallel write methods. We have chosen parameters that range from no compute time to a relatively large amount of compute time. The intuition is that some methods may do a better job at overlapping computation and I/O. Another goal is to determine the interaction between write mechanisms and parallelization strategies for our cluster configuration.

The specific parameters consist of 1024 iterations with 64 KB buffers. The total amount of data written for each simulation is 128 MB. For S1A and S1B, this data will be split across two 64 MB files. For the other methods, this data will be stored in a single 128 MB file. We found that sync size and window sizes gave unpredictable results, so we experimented with a large range of values. We considered the following sync sizes (**syncsize**): 128 KB, 256 KB, 512 KB, 1 MB, 2 MB, 4 MB, 8 MB, 16 MB, 32 MB, 64 MB, and 128 MB. We also consider the following window sizes (**winsize**) for the `mmap()` write mechanism: 1 MB, 2 MB, 4 MB, 8 MB, 16 MB, 32 MB, 64 MB, and 128 MB. Note that we only run experiments with **syncsize** $\leq$ **winsize**. Each test is repeated 10 times and we compute a 10% trimmed mean. For the strided data we used a **stride** of 32 bytes and a **len** of 8 bytes. For the MPI-IO results, we used a call to `MPI_File_write_all()` during each loop, a single call to `MPI_File_sync()` upon completion of the loop, and the the default ROMIO hints.

The results for our MPI and Pthread experiments are listed in Figure 1. The data in the table include the total execution time in seconds (we do not include program startup), the best **syncsize** found for the method (f), and the best **winsize** found for `mmap()` methods (w).

Pthreads and MPI The results show that for our system configuration and the given parameters, if there is a large amount of I/O relative to computation, then using Pthreads provides much better performance than MPI. For contiguous data the best Pthread code for 0 seconds of compute provides a 60% improvement over the best MPI code for this time, and the best Pthread code for 2 seconds of compute provides a 28% improvement. For strided data the best Pthreads code for 0 and 2 seconds provided improvements of 36% and 10%, respectively. On the other hand, if the application is carrying out a relatively little amount I/O (8 and 32 second compute), then the best MPI implementations do as well or somewhat better than the best Pthread implementations. Our straightforward use of ROMIO was not competitive with the other methods unless the amount of computation was very large (32 seconds), and even in these cases it performed somewhat worse than MPI with `mmap()`.

Write method For MPI codes, `mmap()` always outperforms `write()` and `writev()`. For example, for a compute time of 2 seconds, the best strided `mmap()` code provided an 18% improvement over the best `writev()` code. For Pthreads, the situation is almost reversed: in almost every case `write()` or `writev()` outperformed the corresponding `mmap()` code. The exceptions occur for contiguous buffers and relatively large amounts of compute time (8 and 32 seconds). In these situations, S5 with mmap performed best among the Pthreads codes. We expect that for these relatively large amounts of compute time, the extra sync thread is able to provide a significant amount of overlap with computation.

Distribution of work among threads For the MPI codes, it is almost invariably the case that S1A or S1B performs best. In both methods, the processes perform equal amounts of computation and each writes to its own file. S1B differs from S1A in that it schedules the syncs, and, in most cases this seemed to either give some benefit or it didn't cause any degradation. The only clear exception to preferring S1A or S1B occurs in the strided benchmark with 2 seconds of compute time. In this situation, S3DT, is clearly superior.

For Pthreads making an optimal choice of the work distribution is more complicated. For contiguous data and small amounts of computation (0 and 2 seconds), we need to look at the variability in the times. Unfortunately, there wasn't enough space to report this, but the relatively large variability in the

Table 1: MPI Contiguous Data

Comp time	write() S1A	write() S1B	write() S2	write() S3	mmap() S1A	mmap() S1B	mmap() S2	mmap() S3	MPI-IO
0	4.97 64Mf	**4.95** 64Mf	5.28 128Mf	5.00 128Mf	4.72 8Mf 64Mw	4.39 2Mf 2Mw	**4.37** 512Kf 1Mw	5.05 512Kf 1Mw	5.28
2	**6.63** 64Mf	6.77 64Mf	7.35 128Mf	8.29 128Mf	6.78 2Mf 64Mw	**6.37** 2Mf 4Mw	6.67 4Mf 8Mw	7.96 256Kf 32Mw	6.81
8	**10.1** 64Mf	10.40 64Mf	10.70 128Mf	17.5 128Mf	9.54 256Kf 1Mw	**9.05** 2Mf 4Mw	9.22 256Kf 1Mw	16.4 256Kf 128Mw	10.70
32	33.10 64Mf	**33.00** 64Mf	33.60 128Mf	64.50 128Mf	**33.00** 512Kf 2Mw	**33.00** 2Mf 2Mw	33.3 256Kf 2Mw	64.4 256Kf 1Mw	33.70

Table 2: MPI Strided Data

Comp time	writev() S1A	writev() S1B	writev() S2	writev() S2DT	writev() S3	writev() S3DT	mmap() S1A	mmap() S1B	mmap() S2	mmap() S2DT	mmap() S3	mmap() S3DT	MPI-IO
0	8.35 64Mf	8.80 64Mf	10.60 128Mf	8.76 128Mf	12.30 128Mf	**7.60** 128Mf	9.54 512Kf 2Mw	**7.00** 1Mf 2Mw	8.59 256Kf 8Mw	7.62 512Kf 32Mw	9.14 128Kf 1Mw	7.30 512Kf 128Mw	9.71
2	9.82 64Mf	10.00 64Mf	11.00 128Mf	9.85 128Mf	11.50 128Mf	**9.75** 128Mf	9.20 128Kf 8Mw	8.64 512Kf 16Mw	9.67 256Kf 2Mw	8.54 512Kf 32Mw	9.51 512Kf 1Mw	**7.98** 512Kf 16Mw	10.00
8	**12.70** 64Mf	12.80 64Mf	15.90 128Mf	13.00 128Mf	19.90 128Mf	18.60 128Mf	**11.60** 1Mf 1Mw	11.90 4Mf 4Mw	14.70 128Kf 1Mw	12.20 512Kf 2Mw	18.70 128Kf 128Mw	17.70 128Kf 128Mw	15.00
32	**36.20** 64Mf	**36.20** 64Mf	40.10 128Mf	36.50 128Mf	67.30 128Mf	66.10 128Kf	35.50 256Kf 1Mw	**35.10** 4Mf 4Mw	38.70 512Kf 4Mw	35.50 256Kf 1Mw	66.70 256Kf 128Mw	65.70 128Kf 128Mw	38.70

Table 3: Pthreads Contiguous Data

Comp time	write() S1A	write() S1B	write() S2	write() S3	write() S5	mmap() S1A	mmap() S1B	mmap() S2	mmap() S3	mmap() S5
0	2.27 64Mf	2.30 64Mf	2.01 128Mf	**1.75** 64Mf	3.95 128Mf	5.20 32Mf 64Mw	5.07 32Mf 64Mw	5.37 64Mf 64Mw	5.39 64Mf 64Mw	**3.58** 8Mf 16Mw
2	**4.56** 64Mf	4.97 64Mf	4.84 128Mf	4.66 64Mf	6.10 128Mf	7.31 32Mf 64Mw	7.28 16Mf 64Mw	7.35 2Mf 4Mw	8.51 2Mf 4Mw	**5.11** 32Mf 64Mw
8	**9.91** 64Mf	9.92 64Mf	9.90 128Mf	16.23 64Mf	10.32 128Mf	12.93 64Mf 64Mw	12.85 64Mf 64Mw	13.13 64Mf 64Mw	16.43 128Kf 1Mw	**9.31** 1Mf 64Mw
32	33.43 64Mf	33.48 64Mf	**33.32** 128Mf	64.12 128Kf	33.22 256Kf	37.01 64Mf 64Mw	36.36 256Kf 1Mw	37.09 64Mf 64Mw	64.12 256Kf 1Mw	**33.12** 512Kf 16Mw

Table 4: Pthreads Strided Data

Comp time	writev() S1A	writev() S1B	writev() S2	writev() S3	writev() S5	mmap() S1A	mmap() S1B	mmap() S2	mmap() S3	mmap() S5
0	4.68 64Mf	**4.65** 64Mf	7.15 128Mf	7.19 128Mf	9.27 128Mf	7.46 32Mf 64Mw	7.37 32Mf 64Mw	10.06 32Mf 64Mw	10.07 32Mf 64Mw	**6.94** 4Mf 64Mw
2	8.46 64Mf	8.18 64Mf	9.83 128Mf	**7.22** 128Mf	9.99 128Mf	9.93 32Mf 64Mw	9.92 32Mf 64Mw	11.93 512Kf 1Mw	10.07 512Kf 1Mw	**8.58** 2Mf 64Mw
8	**12.45** 64Mf	12.48 64Mf	14.93 64Mf	16.37 64Mf	15.41 128Mf	15.81 64Mf 64Mw	15.47 256Kf 1Mw	18.28 64Mf 64Mw	17.30 128Kf 4Mw	**14.48** 1Mf 64Mw
32	**36.26** 64Mf	36.28 64Mf	38.32 128Mf	64.13 128Kf	38.17 128Kf	39.86 64Mf 64Mw	39.73 64Mf 64Mw	42.25 512Kf 1Mw	64.13 256Kf 1Mw	**38.38** 512Kf 16Mw

Fig. 1. Expermental Results for Parallel Write Methods (times in seconds, f is sync size, w is window size, K is kilobytes, M is megabytes)

times indicated that there was little reason to prefer any one of S1A, S1B, S2, or S3. As we noted earlier, for contiguous data and large amounts of compute time, S5 gives the best performance with Pthreads. For strided data, it is usually the case that S1A or S1B is superior. The exception is S3, which is clearly superior for 2 seconds of compute time.

Syncing and Window Size Syncing allows the application to force the kernel to flush disk buffers to disk. However, we found that attempting to perform `fdatasync()` with `write()` or `writev()` with the benchmarks almost invariably resulted in worse performance than simply using one sync at the end of the benchmark. The degradation in performance is likely due to the current implementation of `fdatasync()` in the Linux kernel, which is really an `fsync()` [1]. The exception to this occurs with S3 and Pthreads. Evidently with a single compute thread and a single I/O thread, the threaded implementations benefit from an extra call to `fsync()`.

For `mmap()`, it is clear that periodic syncing provides significant benefits to performance. In the case of the contiguous MPI benchmarks, a 2 MB sync size provided the best performance. For the other benchmarks, though, a range of sync sizes resulted in optimal performance: We found that a sync size of 512 Kbytes to 4 Mbytes gave the best results.

The range of optimal window sizes is even larger than the range of optimal sync sizes. Each size we tested, from 1 MB to 128 MB, was in some situation optimal. Perhaps noteworthy in this context is the fact that it was almost always the case that the optimal window size for a given situation was less than the file size. So unmapping and remapping windows provides a clear benefit.

Variability As we noted earlier, we do not have sufficient space to report the variability in the data we collected. We found a very wide range: the difference between the fastest and slowest times for a given benchmark could vary by as much as 300%. In general, however, it seems that for a given method, the use of `mmap()` results in somewhat less variation.

5 Conclusions and Future Work

Our study experimentally compares the performance of several parallel disk write methods using both MPI and Pthreads. We provide experimental results for one specific hardware configuration, a configuration that is typical of Linux clusters. Each benchmark was run on a constrained set of simulation parameters. The parameter set we studied was quite large, and we found that no one approach to I/O invariably results in the best performance. In particular, we have shown that using Pthreads gives the best performance when the ratio of I/O to computation is high, but that MPI provides excellent performance when the ration of I/O to computation is low. We also found considerable differences in the performance of `write()`, `writev()`, and `mmap()`, and in the use of various other parameters. This suggests that any optimal implementation of I/O for dual processor nodes will need to use adaptive methods to identify the best combination of parameters.

Many directions for future work are available. Specifically, we want to explore the new asynchronous I/O API (aio) for Linux kernels. We also would like to implement benchmarks using the `O_DIRECT` option for writing files. This option allows applications to bypass the kernel disk caching mechanism. As of the writing of this paper, the `O_DIRECT` option is not supported in the Linux 2.4.20 kernel for the ext3 file system. We also want to increase our parameter space in a sensible way to make our results more general. Ideally, we would like to use our benchmarks to find the proper parallel write method for a specific application.

References

1. Daniel P. Bovet and Marco Cesati. *Understanding the Linux Kernel*. O'Reilly & Associates, Inc., 2nd edition, 2001.
2. Philip H. Carns, Walter B. Ligon III, Robert B. Ross, and Rajeev Thakur. PVFS: A parallel file system for linux clusters. In *Proceedings of the 4th Annual Linux Showcase and Conference*, pages 317–327, Atlanta, GA, 2000. USENIX Association.
3. William Gropp, Ewing Lusk, and Rajeev Thakur. *Using MPI-2: Advanced Features of the Message Passing Interface*. Scientific and Engineering Computation. MIT Press, Cambridge, MA, USA, November 1999.
4. S. More, A. Choudhary, I. Foster, and M. Xu. MTIO: A multi-threaded parallel I/O system. In *Proceedings of the 11th International Parallel Processing Symposium (IPPS-97)*, pages 368–373, Los Alamitos, April 1–5 1997. IEEE Computer Society Press.
5. Leonid Oliker, Xiaoye Li, Parry Husbands, and Rupak Biswas. Effects of ordering strategies and programming paradigms on sparse matrix computations. *SIAM Review*, 44(3):373–393, September 2002.
6. P. Pacheco, M. Camperi, and T Uchino. PARALLEL NEUROSYS: A system for the simulation of very large networks of biologically accurate neurons on parallel computers. *Neurocomputing*, 32–33(1–4):1095–1102, 2000.
7. Rolf Rabenseifner and Alice E. Koniges. Effective file-I/O bandwidth benchmark. *Lecture Notes in Computer Science*, 1900:1273–1283, 2001.
8. Rajeev Thakur, William Gropp, and Ewing Lusk. On implementing MPI-IO portably and with high performance. In *Proceedings of the Sixth Workshop on Input/Output in Parallel a nd Distributed Systems*, pages 23–32, 1999.
9. Rajeev Thakur, Ewing Lusk, and William Gropp. I/O in parallel applications: The weakest link. *The International Journal of High Performance Computing Applications*, 12(4):389–395, Winter 1998.