

# Throttling I/O Streams to Accelerate File-I/O Performance

Seetharami Seelam<sup>1,\*</sup>, Andre Kerstens<sup>2,\*</sup>, and Patricia J. Teller<sup>3</sup>

<sup>1</sup> IBM Research, Yorktown Heights, NY 10598, USA  
sseelam@us.ibm.com

<sup>2</sup> SGI, Sunnyvale, CA 94085, USA  
kerstens@sgi.com

<sup>3</sup> The University of Texas at El Paso (UTEP), El Paso, TX 79968, USA  
pteller@utep.edu

**Abstract.** To increase the scale and performance of high-performance computing (HPC) applications, it is common to distribute computation across multiple processors. Often without realizing it, file I/O is parallelized with the computation. An implication of this is that multiple compute tasks are likely to concurrently access the I/O nodes of an HPC system. When a large number of I/O streams concurrently access an I/O node, I/O performance tends to degrade, impacting application execution time. This paper presents experimental results that show that *controlling* the number of file-I/O streams that concurrently access an I/O node can enhance application performance. We call this mechanism file-I/O stream throttling. The paper (1) describes this mechanism and demonstrates how it can be implemented either at the application or system software layers, and (2) presents results of experiments driven by the cosmology application benchmark MADbench, executed on a variety of computing systems, that demonstrate the effectiveness of file-I/O stream throttling. The I/O pattern of MADbench resembles that of a large class of HPC applications.

## 1 Introduction

Many parallel high-performance computing (HPC) applications process very large and complex data sets. Often the file I/O needed to store and retrieve this data is parallelized along with the computation. This generates multiple I/O streams (e.g., one for each MPI task) that concurrently access the system's I/O nodes. Because the number of I/O nodes usually is much smaller than the number of compute processors, even when only one such application is executing on a system, often an I/O node is concurrently accessed by multiple streams. Due to the advent of multicore processors, as the number of processors in next-generation systems grows to solve larger problem sizes and as the ratio of compute processors to I/O nodes increases, the number of I/O streams concurrently accessing an I/O node is destined to increase. Currently, this ratio is greater than 40 for some high-end systems [1].

---

\* This work was done while the author was affiliated with UTEP.

In some cases, e.g., when the I/O nodes are RAIDs, I/O parallelization can result in improved I/O performance. But, as indicated in [2] and as confirmed by our experimental results, I/O performance drops rather precipitously in many cases. When the number of I/O streams exceeds a certain threshold, application execution times are impacted negatively – this is due to events that prevent full I/O utilization, e.g., seek and rotational latencies, failure to take advantage of prefetching, and/or insufficient I/O request frequency.

For HPC applications similar to MADCAP CMB [3], traditional parallelization of file I/O can result in this behavior, which will be exacerbated in next-generation systems. The challenge is, given a workload, schedule I/O so that as much I/O-node bandwidth as possible is realized. Although this paper does not address this challenge optimally, it describes and gives evidence of a potential solution, i.e., dynamic selection of the number of concurrently-active, synchronous, file-I/O streams. We call this control mechanism *file-I/O stream throttling*.

The potential effectiveness of file-I/O stream throttling is demonstrated in Figure 2(a). The figure compares MADbench<sup>1</sup> execution times when one of 16, four of 16, eight of 16, and 16 of 16 MPI tasks generate I/O streams that concurrently access an I/O node. The depicted experimental results show that careful selection of the number of concurrently-active MADbench file-I/O streams can improve application execution time: in this case, the optimal number of I/O streams is four of 16. In comparison, with one of 16 or 16 of 16 (default parallelization), execution time increases by 18% and 40%, respectively.

The paper describes this potential I/O performance problem in detail, shows its impact on MADbench, which represents the I/O behavior of a large application class, presents the file-I/O stream throttling mechanism, and demonstrates the effect of this mechanism on I/O performance and application execution time. I/O-stream throttling can be performed at the application or system software layers. The former is achieved by having the application code specify the number of streams concurrently accessing an I/O node; this is the method that is incorporated in the MADbench code. Throttling via system software could be implemented by stream-aware I/O schedulers or file-system policies. The strengths and weaknesses of each method are described in the paper.

The remainder of the paper is organized as follows. Section 2 describes related work, Section 3 describes two different stream throttling methods, and Section 4 describes MADbench. Section 5 presents the systems used for experimentation, while the results and their implications are presented in Section 6. Section 7 concludes the paper.

## 2 Related Work

There are several studies that concentrate on optimizing the parallel file-I/O performance of HPC applications by providing libraries and parallel file systems, and

---

<sup>1</sup> MADbench is a benchmark that represents the I/O behavior of the MADCAP CMB application.

exploiting advancements in storage technologies [4,5,6]. Other optimization techniques include exploiting access patterns to assist file-system prefetching, data sieving, and caching [7], overlapping computation with I/O [8], and employing asynchronous prefetching [9]. The latter technique is particularly suitable for HPC applications, which generally have very regular data access patterns that facilitate the identification of the data needed for subsequent computations. Given sufficient disk system bandwidth, prefetching may minimize the effect of I/O-node performance on application performance. However, prefetching too aggressively increases memory traffic and leads to other performance problems [10]. In addition, as mentioned above, when bandwidth is limited, the problem of multiple streams concurrently accessing an I/O node is likely; this can result in a loss of the benefits of prefetching.

Our contribution differs from these methods but is complementary to the data sieving, data prefetching, and caching approaches. The focus of our work is on runtime control of the number of file-I/O streams that concurrently access an I/O node. Such control can minimize expensive disk-head positioning delays and, thus, increase both I/O and application performance.

### 3 File-I/O Stream Throttling

File-I/O stream throttling controls the number of synchronous I/O streams concurrently accessing the I/O system. In general, it must consider various *application characteristics*, e.g., request characteristics (sequential or random) and *system characteristics*, e.g., the numbers of I/O and compute nodes, numbers of processors and tasks on a compute node, storage system configuration, compute-node memory size, and application data size. The large number of characteristics and the complexities associated with them make dynamic I/O-stream throttling a challenging task, one that has not yet been accomplished.

However, many HPC applications (see, e.g., [3,11,12]) have sequential data layouts, where each requested data stream is sequentially accessed from the storage system, and tend to read in an iterative fashion. For such data layouts and read behaviors, the number of streams that should concurrently access an

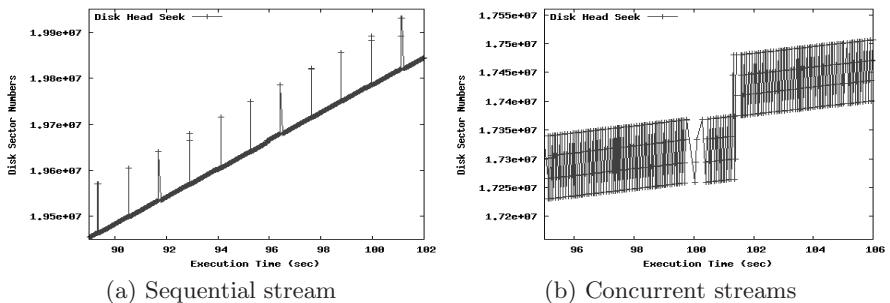


Fig. 1. Seek behavior of the disk head when using one or multiple I/O streams

I/O node is dependent only on the I/O node configuration; the complexities associated with application characteristics need not be considered. In this case, given a single-disk I/O node, I/O performance can be enhanced by minimizing expensive disk-head positioning delays; our experiments demonstrate this. For example, consider an application with a computational loop that is iterated multiple times by four MPI tasks that access unique files stored in an interleaved fashion on a shared disk. When all four streams concurrently access the disk with no coordination the disk head continuously seeks between files; this behavior is illustrated in Figure 1(b). In contrast, as shown in Figure 1(a), when the number of streams that concurrently access the disk is one (the four streams access the disk one after another) the disk head hardly seeks. In this particular example, one stream results in the best performance because the storage system is one simple disk, the bandwidth of which is well utilized by the one I/O stream. In contrast, an advanced storage system with multiple disks needs multiple streams to utilize the available bandwidth. Thus, one should not conclude from this example that using one stream is the best for all storage systems.

### 3.1 Application-Layer Stream Throttling

One way to change (throttle) the number of streams concurrently accessing an I/O node is to wrap the parallelized I/O routines with code that implements a token-passing mechanism [3] – this approach is used by some HPC applications, including MADbench. If one token is used by all MPI tasks then access to storage is sequentialized. At the other extreme, if the number of tokens is equal to the number of MPI tasks, then all I/O streams can concurrently access storage.

The token-passing approach has two problems. First, it involves changes to the application code, which often is targeted to run on various compute- and I/O-node configurations. As shown in this paper, the selection of the number of concurrently-active I/O streams depends on application I/O behavior and system characteristics. Thus, selecting the number requires extensive knowledge of both, as well as the ability to develop configuration-specific code that is tailored to the various systems. As demonstrated in [3], it is not impossible to throttle the number of I/O streams at runtime within an application. However, the use of higher-level I/O libraries such as MPI-I/O can make this very challenging.

Second, I/O throttling is applicable only to synchronous I/O streams – not to asynchronous I/O streams. A synchronous request blocks the execution of the requesting task until it is satisfied by the underlying I/O system, while an asynchronous request does not block task execution. In fact, asynchronous requests often are queued and delayed in the system’s underlying memory buffer cache and scheduled to the relatively slower I/O system at times that provide opportunities for improved I/O performance. It is well known that larger asynchronous request queues provide better opportunities for optimizing disk head movement and improving I/O system performance. Thus, throttling the number of asynchronous request streams is dangerous – it may constrain the number of streams that generate data to the memory buffer cache and, thus, the number of outstanding asynchronous requests in the buffer cache queue. Due to these two

problems, application-layer I/O-stream throttling is not always the easiest and most scalable option. Other alternatives, such as throttling in the OS, file system, and/or middleware, should be explored to improve I/O-node performance. In the next section, we discuss how I/O-stream throttling can be realized in the OS layer of an I/O node.

To properly understand the effect of stream throttling at the application layer, we must ensure that the number of synchronous I/O streams are not throttled at other layers between the application and storage system. For example, I/O scheduling algorithms in the OS often tend to minimize the positional delays on the disk by servicing requests in a pseudo shortest-*seek-first* manner, rather than a fair *first-come-first-served* manner – this is true of our test OS, Linux. Fortunately, the latest versions of Linux allow users to select among four I/O scheduling algorithms, one of which is a completely fair queuing algorithm (CFQ); we use CFQ to demonstrate our application throttling approach.

### 3.2 System-Layer Stream Throttling

In Linux, the Anticipatory Scheduler (AS), unlike CFQ and other I/O schedulers, has the ability to detect sequentiality in an I/O stream and allow a stream that exhibits spatial locality of reference to continue to access the storage system for a certain period of time. This throttling reduces disk-head positioning delays even when multiple streams compete for the disk. In addition, the AS detects asynchronous and synchronous streams and throttles only the synchronous streams. We use AS to demonstrate the effect of system-software throttling.

The main advantage of the system-layer approach, over the application-layer approach, is that it is transparent to the application and application programmer. There are many ways to implement I/O-stream throttling at the system software layer, including the scheduler approach that we use in our experiments. Disk controllers on modern I/O systems have intelligence built into them. This means that I/O-stream throttling implemented at the controller could be more beneficial than if implemented at the I/O-node OS software layer. Also, due to adaptive prefetching policies, some disk controllers can handle multiple streams very efficiently. For instance, [2] shows that storage devices can handle as many as three streams efficiently. However, when the number of disk cache segments is less than the number of streams, performance degrades significantly. Unfortunately, the number of concurrently-active I/O streams that the device can handle effectively is not always communicated to the OS software layer. Thus, in comparison to throttling at the device, throttling at the OS layer may lose some benefits. Also, since the AS is designed only to throttle down the number of streams to one and is not able to judiciously change the number of concurrently-active I/O streams, this scheduler will not be optimal for all system configurations. Currently, a scheduler capable of judiciously selecting the number of concurrently-active I/O streams per I/O node is under investigation.

An experimental evaluation of our system-layer throttling approach is described in Section 6.2. These experiments use the AS and are driven by the

unthrottled, concurrently-active file-I/O streams of MADbench – every MPI task of MADbench concurrently generates I/O.

## 4 MADbench Cosmology Application Benchmark

MADbench [3] is a light-weight version of the Microwave Anisotropy Dataset Computational Analysis Package (MADCAP) Cosmic Microwave Background (CMB) power spectrum estimation code [13]. It was developed using MPI and is one of a set of benchmarks used by NERSC at LBNL to make procurement decisions regarding large-scale DoE computing platforms. MADbench represents a large class of applications that deal with Cosmic Microwave Background Analysis. A significant amount of HPC resources are used by cosmologists, e.g., about 17% at NERSC. The I/O characteristics of MADbench are similar to those of many other applications and benchmarks, such as SMC (electron molecule collision), SeisPerf/Seis (seismic processing), CAM, FLASH [12], and codes based on the Hartree Fock algorithm like NWChem and MESSKIT [14,11]. This indicates that this study, which is based on MADbench, has relevance to a large class of applications and can be used to improve the I/O performance of applications with I/O behavior that is amenable to file-I/O stream throttling.

MADbench has three distinct I/O phases: (1) **dSdC** (write only): Each MPI task calculates a set of dense, symmetric, positive semi-definite, signal correlation derivative matrices and writes these to unique files. (2) **invD** (read only): Each MPI task reads its signal correlation matrices from the corresponding unique files and for each matrix it calculates the pixel-pixel data correlation (dense, symmetric, positive definite) matrix **D** and inverts it. (3) **W** (read only): Each MPI task reads its signal correlation matrices and for each matrix performs dense matrix-matrix multiplications with the results of **InvD**. Each of these three I/O phases are comprised of the following five steps: (1) computation, (2) synchronization (3) I/O, (4) synchronization, and (5) computation (except for **dSdC**). The time that each task spends in the first (second) sync step can be used to identify computational (I/O) imbalance among the tasks.

MADbench includes the following parameters that can be used to generate configurations that differ with respect to the stress they apply on the I/O system:

- **no\_pix**: size(in pixels) of the pixel matrix. The size,  $S$ , in KBytes is  $\frac{1}{8}(no\_pix)^2$ . With  $P$  tasks, each task accesses a file of  $\frac{S}{P}$  KBytes.
- **no\_bin**: number of chunks of data in a file, where the chunk size,  $C$ , is  $(S/P)/no\_bin$ . The choice of **no\_bin** impacts the layout and the interleaving of the files on the disk. The number of chunks of data in a file equals the number of iterations of the three phases.
- **no\_gang**: number of processors for gang scheduling. In our case there is no gang scheduling, i.e., **no\_gang** = 1.
- **blocksize**: ScaLAPACK block size.

In our experiments we use **no\_bin** = 16 and **blocksize** = 32KB. These values are comparable to the values used in the original MADbench study [3]. The value

for `no_pix` is based on the system's memory size and must be set large enough to stress the system's storage subsystem. If `no_pix` is too small, all the data will fit in memory and the I/O subsystem will not be stressed.

Because of the MADbench lock-step execution model, all tasks concurrently read or write their data from or to storage, creating multiple concurrently-active I/O streams. Two additional input parameters, `RMOD` and `WMOD`, can be used to control the number of concurrently-active reader and concurrently-active writer streams, respectively. At most, one out of `RMOD` (`WMOD`) tasks are allowed to concurrently access the I/O system. With  $N$  tasks, the total number of concurrent readers is  $\lceil \frac{N}{RMOD} \rceil$ . MADbench can be executed in two modes: computation and I/O, or I/O only [3]; we use both in our experiments.

## 5 Experimental Systems

Table 1 describes the three systems used in our experiments (Intel Xeon, IBM p690, and Scyld Beowulf cluster) in terms of the number of processors, memory capacity per node, I/O system configuration, and I/O system throughput. The Intel Xeon system, which runs Linux 2.6, contains dual processors (2.80 GHz Pentium 4 Xeons with Hyper Threading) and is attached to a 7,200 RPM 10.2GB EIDE disk. The IBM p690 is a 16-way POWER4 SMP, which also runs Linux 2.6. Experiments on the p690 are of two types w.r.t. memory size and I/O system configuration: memory size is either 16GB or 2GB, while the I/O system is either a 7,200 RPM 140GB SCSI disk or a 350GB DS4300 RAID-5 comprised of six 15,000 RPM SCSI disks. Each node of the Scyld Beowulf cluster contains two 2.0 GHz AMD Opteron processors and 4GB memory; it runs a custom Linux 2.4 kernel from Scyld. Experiments on this system also are of two types, but w.r.t. the I/O system configuration: either a 1.4TB NFS-mounted RAID-5 disk system comprised of six 7,200 RPM 250GB SATA disks or a local 7,200 RPM 120GB SATA scratch disk per node. On all the systems, to eliminate the interference of I/O requests from OS activities, MADbench is configured to access a storage system different than that hosting the OS. In addition, on all systems, to remove buffer cache effects, before an experiment is started, the MADbench storage system is unmounted and remounted.

**Table 1.** I/O- and compute-node configurations of systems

System	Nodes	CPUs/ Node	Memory/ Node(GB)	I/O System(s)	Throughput (MB/s)
Intel Xeon	1	4	1	EIDE disk	23
IBM p690	1	16	16/2	SCSI disk/RAID-5	35/125
Beowulf Cluster	64	2	4	SATA/RAID-5	45

## 6 Results of Experiments

To understand the importance of I/O performance, we conducted an experiment with MADbench running in computation-and-I/O mode on the Xeon system. In this case, I/O time, which depends on the number of concurrently-active readers and writers, ranges from 14% to 37% of total execution time. Borrill, et al. [3] report that on larger systems with  $\geq 256$  processors I/O consumes 80% of execution time. This indicates that for this class of applications I/O contributes significantly to application execution time.

To understand the true impact of I/O performance, one must ensure that the data footprint does not fit in the main memory buffer cache. Otherwise, requests are satisfied by the buffer cache rather than the I/O system. With the explosive growth of the data needs of HPC applications, it is rare that the data ever fits in main memory. Thus, the remainder of our experiments ensure that the data footprint is larger than the combined size of the main memory of the participating compute nodes.

We conducted two sets (base and base+) of experiments driven by MADbench in I/O-only mode. Each experiment in the *base set* has one or all readers and writers. The *base+* set is the base set plus the following experiments: (1) two readers/one writer, (2) four readers/one writer, and (3) eight readers/one writer.

### 6.1 Application-Layer Stream Throttling

The experiments discussed here use application throttling to control the number of concurrently-active MADbench readers and writers. To isolate the effect of application throttling, these experiments use the CFQ I/O scheduler, which provides relatively fair disk access to all concurrently-active MADbench I/O streams and does not inherently perform any I/O-stream throttling, as does AS.

The base set of experiments was conducted on four system configurations, while the base+ set was conducted on two. Rows E1 through E3 of Table 2 show the input parameters for the base set of experiments and rows E4 and E5 show this for the base+ set. Table 2 also shows the best and second-best number of readers/writers for all experiments. Results of each experiment are presented in terms of total execution time and I/O times (LBSTIO and TIO). Since each MPI task waits at a barrier after each iteration of each MADbench phase, LBSTIO represents the average load balancing time, including barrier synchronizations. TIO is the average task I/O time. Note that LBSTIO depends on the number of concurrently-active I/O streams and the underlying I/O scheduler. With a “fair” I/O scheduler, if all tasks are allowed to access the I/O system concurrently, all finish at approximately the same time, hence, the amount of time each waits at a barrier for other tasks to finish will be minimal. In contrast, as the number of concurrently-active I/O streams decreases, tasks scheduled earlier finish earlier and experience longer delays at a barrier waiting for other tasks that are scheduled later. Because of this, we expect an increase in LBSTIO, indicating that some tasks finish their I/O earlier than others.

**Table 2.** Parameter values for experiments

Exp. #	System/Storage	# of tasks	no_pix	Data Size (GB)	Memory (GB)	Two Best Readers/Writers
E1	p690/single disk	16	5,000	2.98	16	16/16 and 1/16
E2	Xeon/single disk	4	3,000	1.1	1	1/1 and 1/4
E3	p690/single disk	16	5,000	2.98	2	1/1 and 1/16
E4	Cluster/RAID-5/NFS	16	25,000	74.5	64	8/1 and 4/1
E5	p690/RAID-5	16	10,000	11.9	2	2/1 and 4/1

**Intel Xeon System:** The results of the base set of experiments run on this system are shown in Table 3. Since the system contains dual processors, the number of logical processors and, thus, the number of MPI tasks, is four. As shown in Table 3, the default practice of all readers/all writers (4/4), results in the longest application execution time. In contrast, employing one reader, reduces execution time by as much as 22%. Note, however, that this decrease in execution time is accompanied by a nine-fold increase in the LBSTIO time in the W (InvD) phase, resulting in load balancing issues.

Given a fixed number of readers, the number of writers has a negligible impact on total execution time – less than or equal to 5%. In contrast, given a fixed number of writers, one reader, versus “all readers”, in this case four, improves execution time by as much as 19.5%.

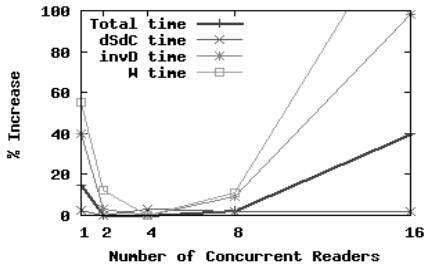
**Table 3.** Application-layer stream throttling: Total execution and I/O times for base set of experiments run on Intel Xeon system

Number of Writers	Number of Readers	Time (s)							
		Total	dSdC		InvD		W		
		TIO	LBSTIO	TIO	LBSTIO	TIO	LBSTIO		
4	4	243	12	9	83	3	77	1	
1	4	235	12	5	83	2	77	1	
4	1	199	12	12	53	13	45	10	
1	1	189	18	9	48	10	42	10	

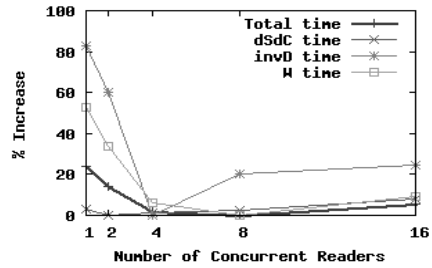
**IBM p690 – Single Disk/2GB Memory:** For this system the number of processors (number of MPI tasks) is 16. Similar to the results associated with the Xeon system, given a fixed number of writers, one reader, versus “all readers”, improves execution time. In this case, execution time increases by as much as 14% with 16 readers. Again, with this execution time improvement comes an increase in the I/O load-balancing time, LBSTIO; it increases from less than ten seconds to as much as several tens of seconds. Similar to the results of the experiments run on the Xeon system, given a fixed number of readers, the number of writers has a negligible impact (less than 3%) on total execution time.

**Table 4.** Application-layer stream throttling: Total execution and I/O times for base+ set of experiments run on Beowulf cluster with RAID-5

Number of Writers	Number of Readers	Time (s)							
		Total	dSdC		InvD		W		
			TIO	LBSTIO	TIO	LBSTIO	TIO	LBSTIO	
16	16	6353	1330	66	2150	72	2210	68	
16	1	8690	1320	67	1815	1596	1826	1606	
1	1	6973	1559	1071	1148	793	1146	794	
1	2	6406	1543	1000	1746	554	1128	568	
1	4	5741	1583	1004	1073	273	1077	265	
1	8	5627	1590	1035	1255	17	1249	17	
1	16	6131	1781	1155	1374	12	1319	28	



(a) IBM p690 with RAID-5



(b) Beowulf with RAID-5

**Fig. 2.** MADbench total and phase execution times for base+ set of experiments run with different numbers of concurrently-active readers on p690 with RAID-5 and Beowulf cluster with RAID-5

**Beowulf Cluster with RAID-5:** The results of the base+ set of experiments run on the Scyld Beowulf cluster with a 1.4TB NFS-mounted RAID-5 disk system comprised of six 7,200 RPM 250GB SATA disks are shown in Table 4. Only 16 processors, i.e., one processor on each of 16 compute nodes (16 MPI tasks) were employed in the experiments. As shown in row E4 of Table 2, each of the 16 nodes has 4GB memory (64GB total), thus, a larger problem size ( $no\_pix = 25,000$ ) is used in order to force disk accesses.

Since each node runs one MPI task and the RAID-5 supports multiple concurrent streams, we expected that given a fixed number of readers, 16 (all) writers, rather than one, would result in better execution time for the write phase (dSdC). One writer is not likely to fully utilize the storage system bandwidth and “all writers” results in larger I/O queues of asynchronous requests, which is known to improve I/O performance [15]. As shown in Table 4, “all writers”, as compared to one, does significantly reduce the dSdC write-phase I/O time (TIO) by over 25% and results in better use of storage system bandwidth. A total of 74.5 GB is written in 1,900 seconds, which translates to 40MB/s, which is 90% of peak bandwidth (45 MB/s). In contrast, one writer takes about 3,000 seconds,

which translates to 25.5 MB/s, which is less than 60% of peak bandwidth. In addition, as shown in Table 4, “all writers” results in relatively smaller load imbalance times (LBSTIO) in the dSdC phase. However, the reduction in the dSdC phase execution time due to the use of “all writers” does not translate into a smaller application execution time because “all writers” has a negative impact on the I/O times of the subsequent read phases: given a fixed number of readers, “all writers” results in 60%, 69%, and 20% increases in the InvD and W TIO times, and total execution time, respectively. We speculate that this is due to data placement on the disk system. When multiple streams compete for storage, the file system tends to allocate chunks to the streams in pseudo round-robin fashion; this results in files that, although contiguous in memory, reside on non-contiguous disk blocks. Thus, with multiple writers block allocation seems to be random, which would significantly impact the subsequent read performance. To circumvent the random placement, we use only one writer for our other experiments. We anticipate that a single writer will remove randomness in block allocation and result in consistent execution times.

Given one writer, we next identify the optimal number of concurrently-active readers (1, 2, 4, 8, or 16) for this system. For each number, Figure 2(b) gives the percentage increase in MADbench’s total and phase execution times; Table 4 contains the data. The sweet spot is eight. In comparison, with 16 readers, the total execution time is about 8% higher; with two, 13% higher; with four, less than 2% higher; and with one, more than 20% higher. With one reader the disk system is underutilized (less than 85% peak bandwidth) because there are not enough I/O requests; with 16, there are enough I/O requests but the disk system spends a significant amount of time seeking to different locations (positional delays) to service readers. In addition, as pointed out in [2], it is likely that 16 reader streams are more than the number of prefetch streams that the storage system can handle; modern storage arrays can prefetch four to eight sequential streams [2]. With eight readers, however, the disk system is well utilized (close to 100% of peak bandwidth), which only means that the positional delays are minimized and the storage system prefetching policies are effective. In summary, as compared to the all writers/all readers (one writer/one reader) case, with one writer and eight readers, application execution time improves by as much as 11% (by about 20%) – see Table 4 and Figure 2.

**IBM p690: RAID-5/2GB Memory:** We repeated the `base+` set of experiments described in the last section on the IBM p690 with a high-end 350GB RAID-5 I/O system. This is done to see if improved execution time results from throttling down the number of readers, i.e., not using “all readers”, in other types of storage systems. Note that the manufacturer of the p690’s storage system is different than that of the cluster’s. In addition, the p690’s storage system is comprised of more expensive hardware, has different disks, and has a faster disk spindle speed (15,000 RPM as compared to 7,200 RPM). The p690’s storage system gets a peak bandwidth of 125MB/s, while that of the cluster gets 45 MB/s. However, both systems have the same number of disks (six) and use the same organization (RAID-5). Thus, we expected and got similar results. The

results, shown in Figure 2(a), show that the sweet spot is four concurrently-active readers. In comparison, 16 readers increases the execution time by about 40%; one increases it by over 13%; and for four and two readers, the difference is less than 2%. Reasoning and analysis similar to that presented in Beowulf cluster section can be used to explain these performance results.

Two interesting observations can be made by comparing Figures 2(a) and 2(b). First, examining the slope of the execution curves between 8 and 16 readers (although, we do not have the data for the intermediate points, it is not required for this observation), one could easily conclude that for MADbench the cluster's storage system handles high concurrency better than the p690's storage system. In comparison, the performance of the p690's storage system drops rather dramatically. Second, examining the slope of the execution time curves between two and four readers, at a concurrency of two readers, the cluster's storage system seems more sensitive. We must acknowledge that since the file system plays a major role in performance, and since these systems have different file systems, this may not be attributable to the storage system alone.

## 6.2 System-Layer Stream Throttling

As discussed in Section 3.2, our experiments with system software I/O-stream throttling are implemented at the OS layer using the Linux 2.6 Anticipatory Scheduler (AS). This I/O scheduler does not throttle the number of concurrently-active writers; it throttles only the number of concurrently-active readers. In addition, it is designed to reduce the number of reader streams to one. We are working on a scheduler that allows throttling to an arbitrary number of streams.

Using the AS, which is available only under Linux, our system-layer throttling experiments reduce to experiments with "all writers" and one reader on the Intel Xeon and IBM p690 systems. For comparison purposes, we also present the no-throttling case, which is implemented using the CFQ I/O scheduler, and the best case from the application-layer throttling experiments. The results of our experiments indicate that the AS throttling behaves in a way that is consistent with the results of our application-layer throttling experiments. With few exceptions, the number of writers (fixed in this case and variable in application throttling) has no significant impact on application execution time and one reader is a good choice.

On the Intel Xeon system, although using the AS (System) decreases execution time by 12%, as compared to the unthrottled case (UT), application throttling performs best; its execution time is about 22% less than that of UT and 11% less than that of System. Thus, it appears that system throttling may provide a good alternative between no throttling and application throttling. Similarly, experiments conducted on the p690, for which the number of processors and MADbench MPI tasks is 16, shows that, although using the AS (System) decreases execution time by 5%, as compared to the UT, application throttling performs best; its execution time is about 12% less than that of UT and 8% (Xeon: 11%) less than that of System. However, in both experiments, system throttling results in much smaller LBSTIO times. This is because, as

compared to application throttling, the AS provides fine-grained throttling of streams and, thus, results in much lower I/O load imbalances in MADbench's read phases.

This gives further credence to the indication that application-transparent, system throttling may provide a good alternative between no throttling and application throttling. It is worth remembering that a big advantage of system throttling is that it is transparent to the application programmer.

## 7 Summary and Conclusions

Our results indicate that synchronous I/O-stream throttling can improve execution time significantly for a large class of HPC applications. This improvement is at least 8% and at most 40% for different systems executing MADbench. The best number of concurrent readers is one for the single-disk case, four for the 16-processor SMP with a state-of-the-art RAID, and eight for a Beowulf cluster with a RAID. This shows that stream throttling depends on application I/O behavior as well as the characteristics of the underlying I/O system. On the other hand, we see that the effects of asynchronous I/O-stream throttling are indirect and are not very well understood. On systems with a single disk the number of writers has a negligible effect on total application execution time and on the write-phase time. However, on systems with a RAID, although the number of writers does not impact the write-phase time, it impacts the subsequent read performance and, hence, the total execution time.

We explored I/O-stream throttling at the application layer and, in a limited form, at the system software layer (i.e., via the Linux Anticipatory Scheduler, which can dynamically reduce the number of streams to one). The former requires changes to the application code, while the latter is transparent to the application and can be implemented in the OS, file system, and/or middleware.

In summary, our work indicates that too few or too many concurrently-active I/O streams can cause underutilization of I/O-node bandwidth and can negatively impact application execution time. A general mechanism within system software is needed to address this problem. We are developing a dynamically-adaptive stream throttling system that will reside in the OS or file system of I/O nodes. Because of the load balancing issues discussed in the paper, we plan to provide controls to enable and disable this adaptation.

## Acknowledgements

This work is supported by DoE Grant No. DE-FG02-04ER25622, an IBM SUR grant, and UTEP. We thank D. Skinner, J. Borrill, and L. Oliker of LBNL for giving us access to MADbench and D. Skinner and the UTEP DAiSES team, particularly, Y. Kwok, S. Araunagiri, R. Portillo, and M. Ruiz, for their valuable feedback.

## References

1. Oldfield, R., Widener, P., Maccabe, A., Ward, L., Kordenbrock, T.: Efficient data movement for lightweight I/O. In: HiPerI/O. Proc. of the 2006 Workshop on High-Performance I/O Techniques and Deployment of Very-Large Scale I/O Systems (2006)
2. Varki, E., Merchant, A., Xu, J., Qiu, X.: Issues and challenges in the performance analysis of real disk arrays. *IEEE Trans. on Parallel and Distributed Systems* 15(6), 559–574 (2004)
3. Borrill, J., Carter, J., Oliker, L., Skinner, D.: Integrated performance monitoring of a cosmology application on leading HEC platforms. In: ICPP 2005. Proc. of the 2005 Int. Conf. on Parallel Processing, pp. 119–128 (2005)
4. Chen, Y., Winslett, M., Cho, Y., Kuo, S.: Automatic parallel I/O performance optimization in Panda. In: Proc. of the 7th IEEE Int. Symp. on High Performance Distributed Computing, IEEE Computer Society Press, Los Alamitos (1998)
5. Thakur, R., Gropp, W., Lusk, E.: Data sieving and collective I/O in ROMIO. In: Proc. of the 7th Symp. on the Frontiers of Massively Parallel Computation, pp. 182–189 (February 1999)
6. Chen, P., Lee, E., Gibson, G., Katz, R., Patterson, D.: RAID: High performance, reliable secondary storage. *ACM Computing Surveys* 26(2), 145–185 (1994)
7. Madhyastha, T., Reed, D.: Exploiting global input/output access pattern classification. In: Proc. of SC 1997, pp. 1–18 (November 1997)
8. Ma, X., Jiao, X., Campbell, M., Winslett, M.: Flexible and efficient parallel I/O for large-scale multi-component simulations. In: Proc. of the Int. Parallel and Distributed Processing Symp. (April 2003)
9. Sanders, P.: Asynchronous scheduling of redundant disk arrays. *IEEE Trans. on Computers* 52(9), 1170–1184 (2003)
10. Aggarwal, A.: Software caching vs. prefetching. In: ISMM '02. Proc. of the 3rd Int. Symp. on Memory Management, pp. 157–162 (2002)
11. Kendall, R., et al.: High performance computational chemistry: an overview of NWChem a distributed parallel application. *Computer Phys. Comm.* 128, 260–283 (2000)
12. Weirs, G., Dwarkadas, V., Plewa, T., Tomkins, C., Marr-Lyon, M.: Validating the Flash code: vortex-dominated flows. *APSS* 298, 341–346 (2005)
13. Carter, J., Borrill, J., Oliker, L.: Performance characteristics of a cosmology package on leading HPC architectures. In: Bougé, L., Prasanna, V.K. (eds.) HiPC 2004. LNCS, vol. 3296, pp. 176–188. Springer, Heidelberg (2004)
14. Crandall, P., Aydt, R., Chien, A., Reed, D.: Input/output characteristics of scalable parallel applications. In: Proc. of SC 1995 (1995)
15. Seelam, S., Babu, J., Teller, P.: Rate-controlled scheduling of expired writes for volatile caches. In: QEST 2006. Proc. of the Quantitative Evaluation of Sys-Tems (September 2006)