MTC Envelope: Defining the Capability of Large Scale Computers in the Context of Parallel Scripting Applications

Zhao Zhang
Department of Computer Science
University of Chicago
zhaozhang@uchicago.edu

Daniel S. Katz
Computation Institute
University of Chicago &
Argonne National Laboratory
d.katz@ieee.org

Michael Wilde
Computation Institute
University of Chicago &
Argonne National Laboratory
wilde@mcs.anl.gov

Justin M. Wozniak
Mathematics and Computer Science Division
Argonne National Laboratory
wozniak@mcs.anl.gov

Ian Foster
Computation Institute
University of Chicago &
Argonne National Laboratory
foster@anl.gov

ABSTRACT

Many scientific applications can be efficiently expressed with the parallel scripting (many-task computing, MTC) paradigm. These applications are typically composed of several stages of computation, with tasks in different stages coupled by a shared file system abstraction. However, we often see poor performance when running these applications on large scale computers due to the applications' frequency and volume of filesystem I/O and the absence of appropriate optimizations in the context of parallel scripting applications.

In this paper, we show the capability of existing large scale computers to run parallel scripting applications by first defining the MTC envelope and then evaluating the envelope by benchmarking a suite of shared filesystem performance metrics. We also seek to determine the origin of the performance bottleneck by profiling the parallel scripting applications' I/O behavior and mapping the I/O operations to the MTC envelope. We show an example shared filesystem envelope and present a method to predict the I/O performance given the applications' level of I/O concurrency and I/O amount. This work is instrumental in guiding the development of parallel scripting applications to make efficient use of existing large scale computers, and to evaluate performance improvements in the hardware/software stack that will better facilitate parallel scripting applications.

Categories and Subject Descriptors

D.4.8 [**Operating systems**]: Performance—*Measurements, Modeling and prediction*

General Terms

Design; Performance

HPDC'13, June 17–21, 2013, New York, NY, USA.

Keywords

Many-task computing, Parallel scripting, Shared file system

1. INTRODUCTION

Many-task computing applications [25] link existing parallel or sequential programs via a filesystem abstraction. Task dependencies are represented by file production and consumption. Parallel scripting is a powerful and convenient tool to construct such applications without modifying the original programs and can naturally maximize execution parallelism. Such parallel scripting applications have been widely used by scientists in fields of astronomy [24, 14, 15], biological science [26, 22, 2, 18], chemistry [12], earth science [17, 11], economics [10], material science [9] and many others. And researchers have enabled such applications on many types of platforms such as clusters, clouds, supercomputers, and grids [37, 1, 3, 19, 34, 36, 35].

On the platforms where there is shared file system that can be used for inter-task communication, a task reads one or several files as input, executes for a while, then writes several files as output. Tasks in later stages might consume the files produced in previous stages as input files. Parallel scripting applications usually have a large number of tasks, and both filesystem access frequency and I/O amount are not well addressed by existing shared file systems and I/O modules. To make good use of a large scale computer, a programmer should have a picture of how the machine facilitates his application, or how his application fits the machine. For example, parallel scripting programmers should know how many times a task could read a file within unit time and how many bytes a task could write to shared filesystem given the concurrency. Unfortunately, we don't have a well-defined shared filesystem benchmark suite that shows the capacity of parallel scripting applications.

This paper's goal is to define and quantitatively evaluate such large scale computers' capacity of parallel scripting applications, which we call the envelope. Our approach is to first understand the applications' I/O behavior and identify the performance metrics that characterize the application I/O performance on large scale computers. Second, we measure those performance metrics on existing large scale system with scales to define the MTC envelope on each scale

Table 1: Basic Application Stage Statistics

	Montage				BLAST			CyberShake		
stage	mProject	mDiff	mFit	mBack	formatdb	blastall	merge	extract	seis	peakGM
tasks	1319	3883	3883	1297	63	1008	16	393	11868	11868
input files	2638	7766	7766	1297	63	4032	1008	1179	35604	11868
output files	2594	3883	7746	1297	252	1008	16	786	11868	11868
input amount (GB)	3.2	2.2	35.8	5.8	4.0	64.5	0.9	41.3	2655.2	0.4
output amount (GB)	10.9	0.001	3.6	5.4	4.7	0.9	0.06	40.1	0.3	0.03

as well as the scalability of existing shared file systems. The third step is to map the applications' behavior on the measured envelope. By doing this, we can classify the application by the factor(s) that bound its I/O performance. The last step is to deliver an guide to show the estimated I/O time consumption given the application's I/O concurrency and I/O amount.

More specifically, we pick three typical parallel scripting applications: Montage [14], BLAST [2] and CyberShake PostProcessing [17]. We run all stages of those applications and trace the filesystem related system calls, then align the trace of the tasks in any one stage with the system call sequence from the profile of that stage. We identify four common concurrent filesystem accesses from those application profiles: file open, file create, read, and write. The parallel scripting applications feature a multi-read-single-write I/O pattern, where a file can be read many times, but only written once. Within the multi-read pattern, we further classify read access as 1-to-1 read and N-to-1 read. The 1-to-1 read refers to the case where each task reads one distinct file, while the N-to-1 read refers to the cases where many tasks read one common file.

We therefore define the MTC envelope as eight curves, showing

- file open operation throughput,
- file creation operation throughput,
- 1-to-1 read data throughput,
- 1-to-1 read data bandwidth,
- N-to-1 read data throughput,
- N-to-1 read data bandwidth,
- write data throughput, and
- write data bandwidth

on the y axis, versus the number of GPFS clients on the x axis.

With these eight curves measured on the GPFS system of the Intrepid BG/P supercomputer, we are able to classify those application stages by the factors that bound them. An application can be metadata bound or I/O bound. In the I/O bound case, they can further be classified as operation throughput bound, data bandwidth bound or concurrency bound. To guide users programming parallel scripting applications on systems with shared filesystems, we compile benchmark performance data on operation concurrency and I/O amount into two heat maps of projected I/O throughput and bandwidth for each filesystem. This allows the programmer to tell if the application can run efficiently on a given computer, what is an efficient scale for the application, and how much time the I/O operations will take.

In this first attempt to characterize the MTC envelope of large-scale systems we assume two realistic constraints: we evaluate a set of existing, unmodified application scripts as they are currently implemented by their science communities (but transcribed into Swift for ease of testing); similarly we measure the behavior of the current production BG/P filesystem with its current configuration, typical background workload, and performance limitations. Note that we run many of our performance measurements on the I/O nodes (referred to as GPFS clients). The reader should consider that each of these I/O nodes is associated with 64 compute nodes, or 256 compute cores.

Regarding definitions, throughout the paper, we refer to the entire parallel script used to execute a scientific task as the *application*, and examine in detail the various stages of processing within these application scripts. A *task* within these stages refers to the invocation of a single application *program* by the parallel script. Further, we model and examine the *throughput* of the application script in terms of the rate at which it performs I/O *operations*, and the *bandwidth* in terms of the rate of data transfer in units of (scaled) bytes per second.

The contributions of this work include a novel approach to understand the concurrency of parallel scripting applications I/O behavior, a suite of performance metrics that characterize parallel scripting applications' I/O behavior and measure later system improvements, and a guide for parallel scripting application writers to make better use of existing hardware-software stack.

2. APPLICATION I/O PROFILE

We profile the I/O behavior of the parallel stages of Montage, BLAST, and CyberShake (ten stages total). Table 1 shows basic statistics: number of tasks, number of input/output files, and total input/output amounts. Note that the ratio between the input amounts and the number of input files denotes how many bytes are read from each input file, while a file or part of a file can be read many times.

2.1 Application Configuration

For Montage, we run a 6x6 degree mosaic centered at galaxy m101. Montage has four parallel stages: mProject, MDiff, mFit, and mBackground (referred to as mBack in this paper for reasons of space.) For BLAST, we search the first 256 sequences in the NCBI nr database against the database itself. BLAST has three parallel stages: formatdb, blastall, and merge. For CyberShake, we run the post-processing workload against the geographic site TEST. CyberShake has three parallel stages: extract, seis, and peak (referred to as peakGM in this paper for reasons of clarity.)

2.2 Application Profiling Methodology

For each application stage, we run all tasks in parallel and trace all system calls during execution. We initially

attempted to align the execution trace with the absolute time, as shown in Figure 1, and found two problems. First, run time is dependent on the machine where the application stage is run, so the profile on Machine A is not necessarily valid on Machine B. Second, the filesystem is usually shared by multiple users, so if we align the execution trace by time, the shared filesystem access delay from other users will add noise, and we won't be able to observe the spike of operations that would otherwise be concurrent. Note that we aggregated file creation and file open over one second intervals. Most of the time, the frequency of open is identical to the frequency of create, as in most of the traces for peakGM, the open and creation occur in the same second.

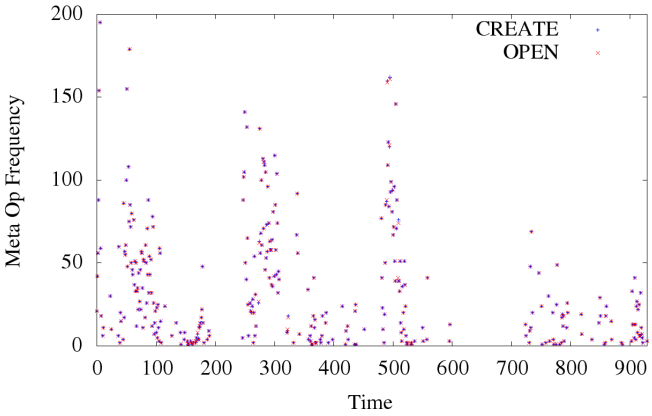

Figure 1: peakGM metadata operations vs. time

We therefore decided to align the system call trace with the order of the system call sequence, rather than with time. The significant benefit of this is that the profile represents the characteristics of the application, independent of any machine, and the time can be projected when we map the filesystem operation onto specific machines. We are thus able to see spikes of concurrent metadata operations in reads and writes.

2.3 Metadata Operation Distribution

Among the ten application stages, we see two major patterns of metadata operation distribution. Figure 2 shows file creation and open distribution over the system call sequence: file creation and file open accesses are two spikes. When those two sets of operations are executed on a system with a shared file system, they will first be limited by the concurrency bound, then they will be limited by throughput at a particular scale. A second pattern is shown in Figure 3, where the file creations are spread across a time range, which mean that at any given time, the concurrency bound of file creation in mProjectPP is 303, though there are 2,594 file creations in total.

In those reads, we see three access patterns for input files: some files are read by only one task in a stage, some files are read by all tasks in a stage, and some files are read by a number of tasks less than the total number of tasks in a stage. We refer those three cases as 1-to-1, N-to-1, and Few-to-1 respectively. Table 2 shows the detailed statistics of the metadata operation spikes as well as the maximum concurrency. Those access patterns suggestion opportunities for potential read optimization in I/O middleware. For

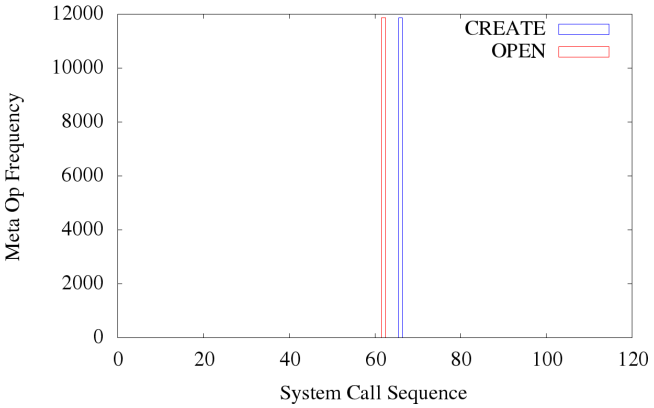

Figure 2: peakGM metadata operations vs. system call sequence

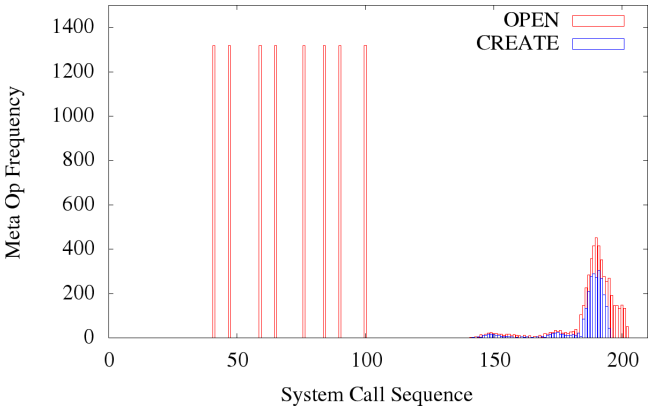

Figure 3: mProjectPP metadata operations vs. system call sequence

example, the N-to-1 reads can be replaced by a single read by one of the tasks, followed by a broadcast to all other tasks with a parallel algorithm. MPI-collective functions [32] can apply such optimizations, but this breaks the independent MTC paradigm.

2.4 I/O Distribution

To show the I/O traffic of each application stage, we record the number of bytes of each read() and write() system call. Then we align this I/O traffic to the system call sequence to determine its concurrency. In Figures 4-9, the vertical lines show the total number of bytes read/written at any given system call, and the points show the average I/O amount. Table 3 shows the trace length (number of system calls), the maximum read/write concurrency, and the maximum I/O traffic.

One interesting I/O pattern in the mProjectPP, mFit, and mBack stages is shown in Figures 4, 5, and 6, where the reads and writes tend to transfer the same amount of I/O traffic per system call, and the number is close to 64 KBytes. It is likely that a buffer of size 64 KBytes is used for file access. We see the same pattern for reads in the formatdb, blastall, and merge stages, shown in Figures 7, 8, and 9. However, the write traffic in formatdb and blastall comprises

Table 2: Application Metadata Operation Stats

	Montage				BLAST			CyberShake		
stage	mProject	mDiff	mFit	mBack	formatdb	blastall	merge	extract	seis	peakGM
open spikes	8	1	10	4	1	4	63	1	1	1
Max. open concurrency	1319	3883	3883	1297	63	1008	16	393	11868	11868
create spikes	0	1	0	0	3	1	1	0	0	1
Max. create concurrency	303	3883	445	113	63	1008	16	81	768	11868
1-to-1 read spikes	4	1	6	4	1	0	63	1	0	1
N-to-1 read spikes	4	0	4	0	0	1	0	0	0	0
Few-to-1 read spikes	0	0	0	0	0	3	0	0	1	0

Table 3: Application I/O Stats

	Montage				BLAST			CyberShake		
stage	mProject	mDiff	mFit	mBack	formatdb	blastall	merge	extract	seis	peakGM
trace length	388	80	328	201	488145	17977	3206	35048	393	11868
Max. read concurrency	1319	3883	3883	1297	63	1008	63	16	393	11868
Max. write concurrency	1246	913	1931	1216	63	1008	16	16	183	768
Max. bytes read/call (MB)	86.4	250.3	254.5	84.9	4.1	6.6	1	1.0	129.0	777.8
Max. bytes written/call (MB)	81.5	0.3	110.2	79.6	2.6	0.2	16	1.0	6.0	18.4

many small writes of hundreds of bytes. The reason is that these two stages process the protein sequence in a sequential order, and each sequence has hundreds of bytes. The I/O traffic patterns in the extract, seis and peakGM stages are all unique, yet they feature high concurrency and are data-intensive.

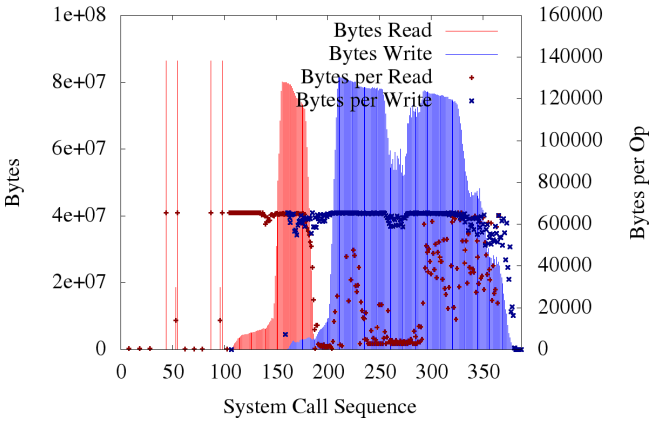

Figure 4: mProjectPP I/O traffic profile

3. MTC ENVELOPE

The goal of the MTC envelope is to show the capacity of large scale computers in the context of parallel scripting applications. Section 2 showed us that the I/O performance of an application stage is first bound by concurrency, which determines the metadata throughput, I/O throughput and bandwidth at a certain scale. Then the I/O performance is bound by metadata throughput. For either read or write traffic, the stage will query the metadata to find out the location of the actual files or create entries in metadata server if the files do not exist in the write case. Finally, the I/O performance is bound by I/O throughput for small files and by I/O bandwidth for large files.

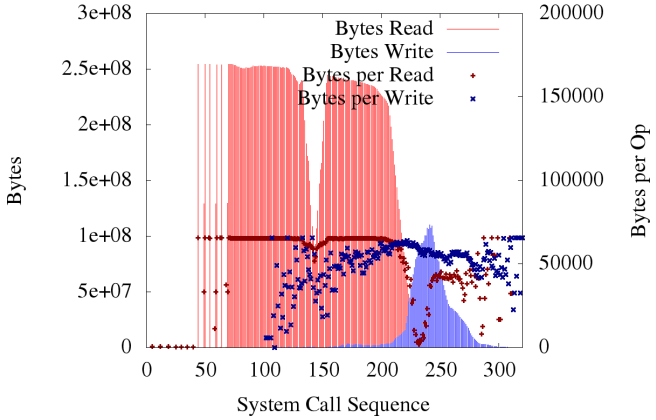

Figure 5: mFit I/O traffic profile

3.1 Definition

We define the MTC envelope for a fixed number of computing resources as a set of eight performance metrics:

- file open operation throughput,
- file creation operation throughput,
- 1-to-1 read data throughput,
- 1-to-1 read data bandwidth,
- N-to-1 read data throughput,
- N-to-1 read data bandwidth,
- write data throughput, and
- write data bandwidth

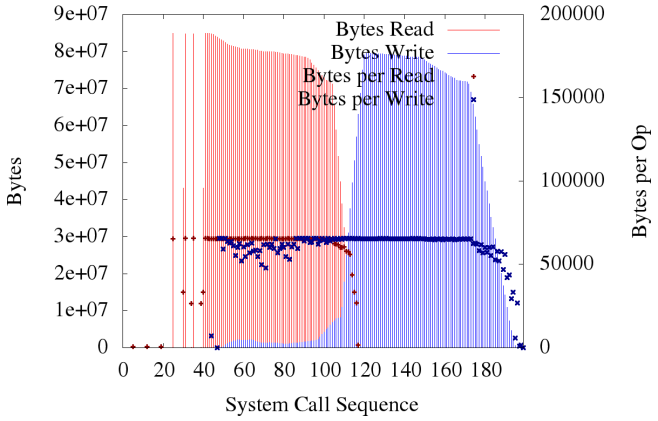

Figure 6: mBack I/O traffic profile

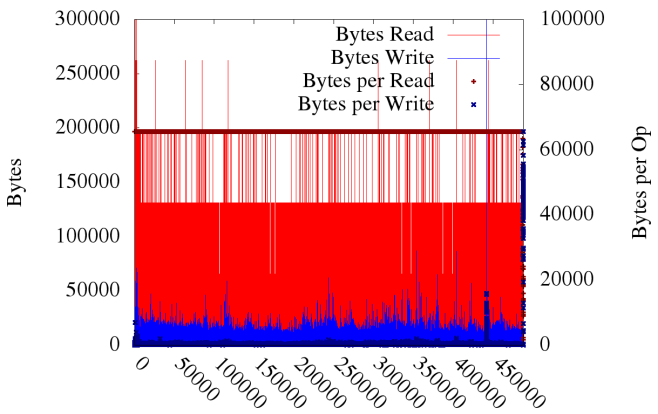

Figure 7: formatdb I/O traffic profile

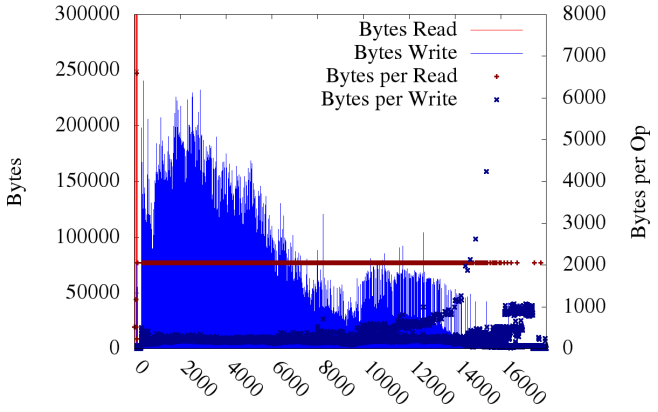

Figure 8: blastall I/O traffic profile

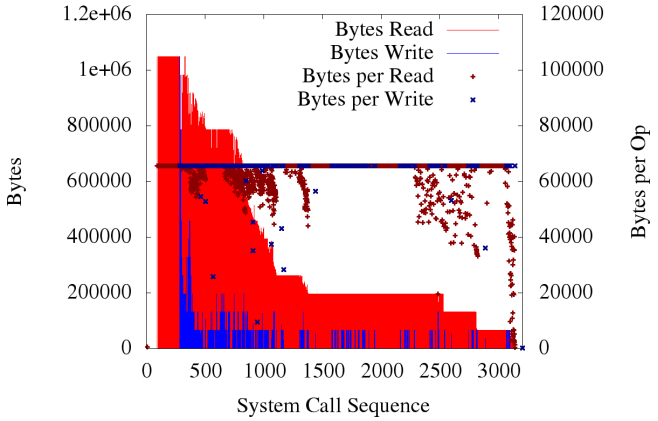

Figure 9: merge I/O traffic profile

In §2, we also discussed the Few-to-1 case, but here we simplify this by merging Few-to-1 into 1-to-1.

A system will exhibit different MTC envelopes at different scales. Its throughput and bandwidth will change as the level of I/O concurrency changes. Therefore, we further define the MTC envelope for a given large scale system as eight curves (rather than eight points), where each curve reflects the change of one of the performance metric along the scales.

3.2 Measurements

In this paper, we measure the GPFS deployment on the Intrepid BG/P at Argonne National Laboratory as an example. This GPFS deployment has one metadata server and 128 IBM x3545 file servers, each with two 2.6-GHx Dual Core CPU and 8 GB RAM. The compute nodes reach the shared file system through I/O nodes, where each I/O node is associated with 64 compute nodes. In this paper, we run our envelope measurements directly on the I/O nodes.

We run our I/O benchmark workloads on the I/O nodes, where each I/O node is a GPFS client. The performance test space for metadata operations is {create, open}×{1, 2, 4, 8, 16, 32, 64, 128, 256} clients, while the performance test space for I/O is {read, write}×{1 KB, 128 KB, 1 MB, 16 MB}×{1, 2, 4, 8, 16, 32, 64, 128, 256} clients. The total number of performance test jobs is 630, and the whole test takes about 2.2 million core hours on the BG/P. Note that we used a shell script as our test framework to represent a typical parallel scripting application, and that we ran on a multiuser system, not a dedicated system. Therefore, we are not measuring the peak performance of the I/O system, but rather the performance that parallel scripting applications experience when sharing the filesystem with other users.

3.3 Metadata Operation Throughput

We measure file creation throughput by asking all GPFS clients to create independent files in one single directory. In all test cases, the total number of file creations is 8,192. Using a similar strategy, we measure file open throughput by creating 8,192 files in one directory, then ask all GPFS clients to query the files, with each file only queried once.

Figure 10 shows the scaling of throughput of file creation and open in a single directory. Both creation and open throughput increase linearly to eight GPFS clients. Then the rate of increase slows down from eight clients to 32 clients. Above 32 clients, throughput starts to slow down. The reason for the creation slowdown is that the GPFS metadata server uses a locking mechanism to resolve concurrent create operations in one directory, and 64 and more

clients triggers the concurrency bottleneck of the locking mechanism. The query slowdown is due to a concurrency bottleneck of the tested GPFS deployment.

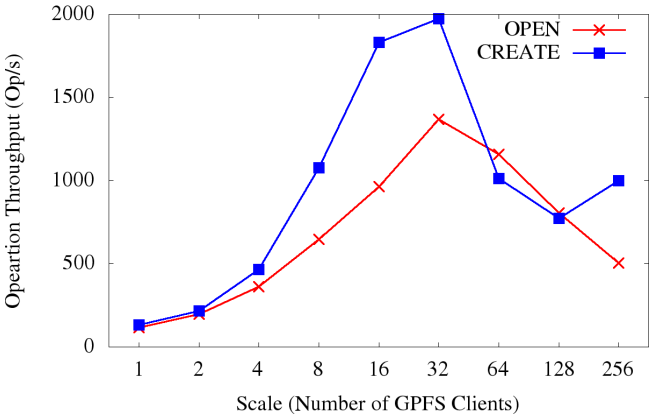

Figure 10: Metadata Operation Throughput

3.4 1-to-1 Read Performance

In the 1-to-1 read benchmark measurement, we first create 8,192 files with size of {1 KB, 128 KB, 1 MB, 16 MB} in one directory. Then we start a number of GPFS clients to simultaneously read those files. With N clients, each client reads $8,192/N$ distinct files by copying the file from GPFS to its local RAM disk. Figures 11 and 12 show the 1-to-1 read operation throughput and data bandwidth respectively.

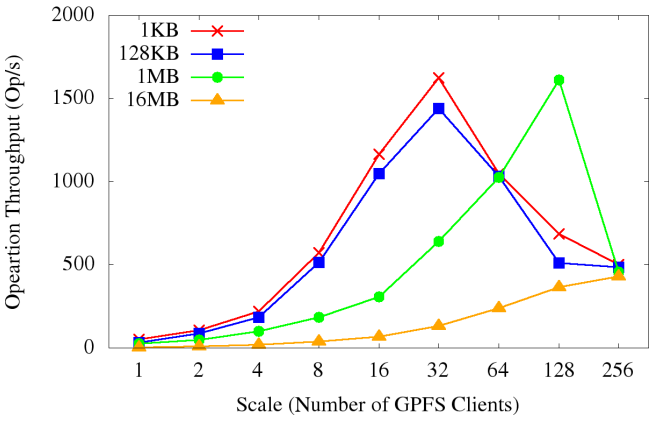

Figure 11: 1-to-1 Read Operation Throughput

In Figure 11, the 1 KB and 128 KB 1-to-1 reads are dominated by latency, as seen by the closeness of the two curves. For these two reads, the throughput increases up to 32 GPFS clients, then starts to slow down. This slowdown is similar to that of the open operation in §3.3, and is likely due to the same cause. As the cost of the small file read is mostly that of determining the file's location by querying the metadata server, the throughput of read is about 25% lower than that of open due to the extra overhead that comes from the actual data transfer. 1 MB reads reach the throughput bound at a larger scale than 1 KB and 128 KB reads because the data transfer has less traffic congestion than open

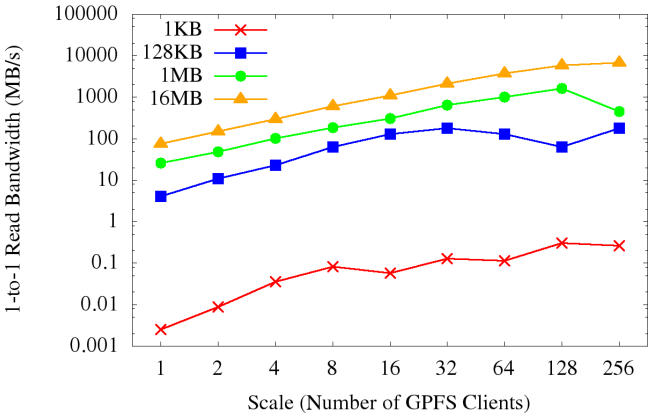

Figure 12: 1-to-1 Read Data Bandwidth

does. With larger file size, the 16 MB reads have not become throughput bound with 256 clients.

Peak GPFS read performance has been documented at 62,000 MB/s [4]. Figure 12 (log scale) does not manage to reach this limit; the peak bandwidth for 16 MB read is 6,880 MB/s. For the bandwidth dominated 1 MB and 16 MB read tests, the performance scales up nearly linearly from one to 128 clients, then the 1 MB read reaches its ceiling, while 16 MB read performance is still increasing.

3.5 N-to-1 Read Performance

Here, we first create a single file with size {1 KB, 128 KB, 1 MB, 16 MB}. Then we let a number {1, 2, 4, 8, 16, 32, 64, 128, and 256} of GPFS clients concurrently read the file. With N clients, each client will read the same file $8,192/N$ times.

Figures 13 and 14 show the N-to-1 read operation throughput and data bandwidth respectively. 1 KB and 128 KB reads reach peak throughput with 64 clients, which is twice as many clients as where the 1-to-1 read throughput peaks. One potential explanation of the improvement is metadata caching in the GPFS client, where when file is read multiple times, the client does not have to re-query the metadata server. However, with 128 clients, the performance is bound by the read concurrency of the shared file system.

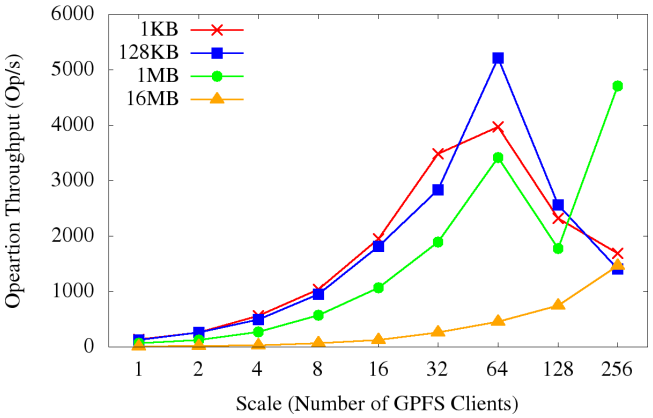

Figure 13: N-to-1 Read Operation Throughput

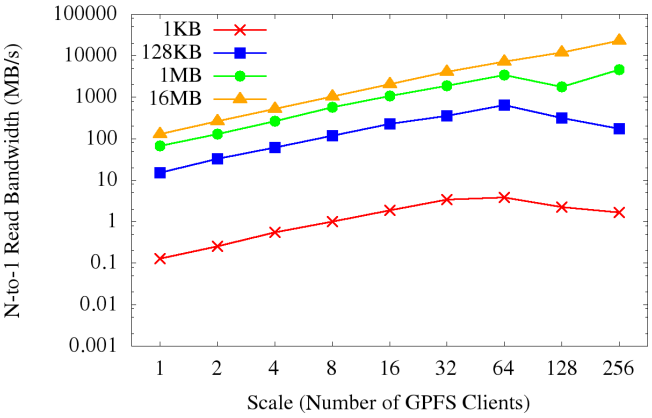

Figure 14: N-to-1 Read Data Bandwidth

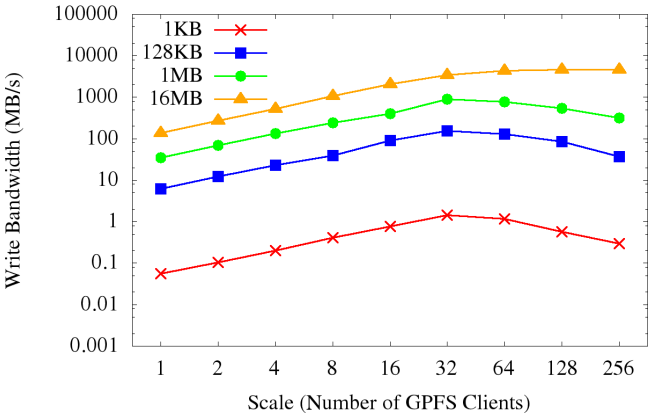

Figure 16: Write Data Bandwidth

3.6 Write Performance

We run the write performance benchmark in the space of {1 KB, 128 KB, 1 MB, 16 MB}×{1, 2, 4, 8, 16, 32, 64, 128, 256} clients. To focus only on write performance, we initially create 8,192 empty files in one directory, and then each client writes to a mutually exclusive group of the files. With N clients, each client writes to $8,192/N$ distinct files.

As shown in Figure 15, the 1 KB, 128 KB, and 1 MB writes reach the operations throughput bound at 32 clients, which is identical to the peak of 1-to-1 reads. In Figure 16, the 16 MB write reaches the data bandwidth bound at 64 clients, yielding a data bandwidth of 3,497 MB/s.

One interesting observation of Figures 11 and 15 is that the operations throughput is independent of the file size. This suggests that 256 clients limit the throughput of GPFS to approximately 470 Op/s for read and 180 Op/s for write. This is a significant characteristic and a limiting factor of the envelope.

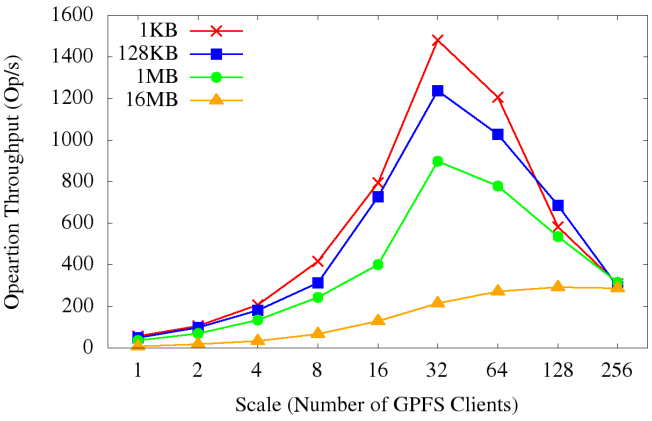

Figure 15: Write Operation Throughput

3.7 Envelope Summary

We have defined the MTC envelope as eight performance metrics of file open throughput, file creation throughput, 1-to-1 read throughput, 1-to-1 read bandwidth, N-to-1 read throughput, N-to-1 read bandwidth, write throughput, and write bandwidth. We show the envelope on 1, 2, 4, 8, 16, 32, 64, 128, and 256 clients as a Kiviat diagram in Figure 17. This shows the performance on each metric relative to that on one client.

Scaling up to 16 clients shows generally good performance. Overall, most of the metrics reach a peak at some number of clients and then decrease, with the exception of N-to-1 read bandwidth, which has not yet reached a peak using 256 clients. The throughput metrics, for both metadata and transfer, appear to have the worst scalability. 1-to-1 read bandwidth and write bandwidth appear to have reached their peak with 256 clients. Our characterization of the MTC envelope indicates that while the BG/P I/O system can sustain a very large I/O data bandwidth, its peaks in operation throughput per second remain a limiter of MTC application performance.

4. PUTTING THE APPLICATION IN THE ENVELOPE

So far, we have measured the performance metric suite at multiple scales, which is enough to characterize the I/O performance of the system for running parallel scripting applications. Now, we examine predicting I/O time consumption and I/O performance bounding factors for the parallel scripting applications themselves.

4.1 I/O Time Consumption

To ease the work of predicting I/O time consumption of parallel scripting applications on the shared filesystem, we first example write on GPFS.

Assume one stage of an application has N tasks, with each writing one output file of size D bytes at the same time, and that the stage runs on a group of computing resources that has C shared filesystem clients. To evaluate the I/O consumption of this stage, we first find the I/O bandwidth B, throughput T_w, and metadata creation throughput T_m at scale C. All writes will come as $\lceil N/C \rceil$ rounds. In each round, metadata creation takes C/T_m time, and the write time consumption is C/T when throughput dominates or $C * D/B$ when bandwidth dominates. So the write time consumption can be approximately expressed as

$$Time = \lceil N/C \rceil * (C/T_m + C * D/B) \quad (1)$$

when the write is bandwidth bound, or as

$$Time = \lceil N/C \rceil * (C/T_m + C/T_w) \quad (2)$$

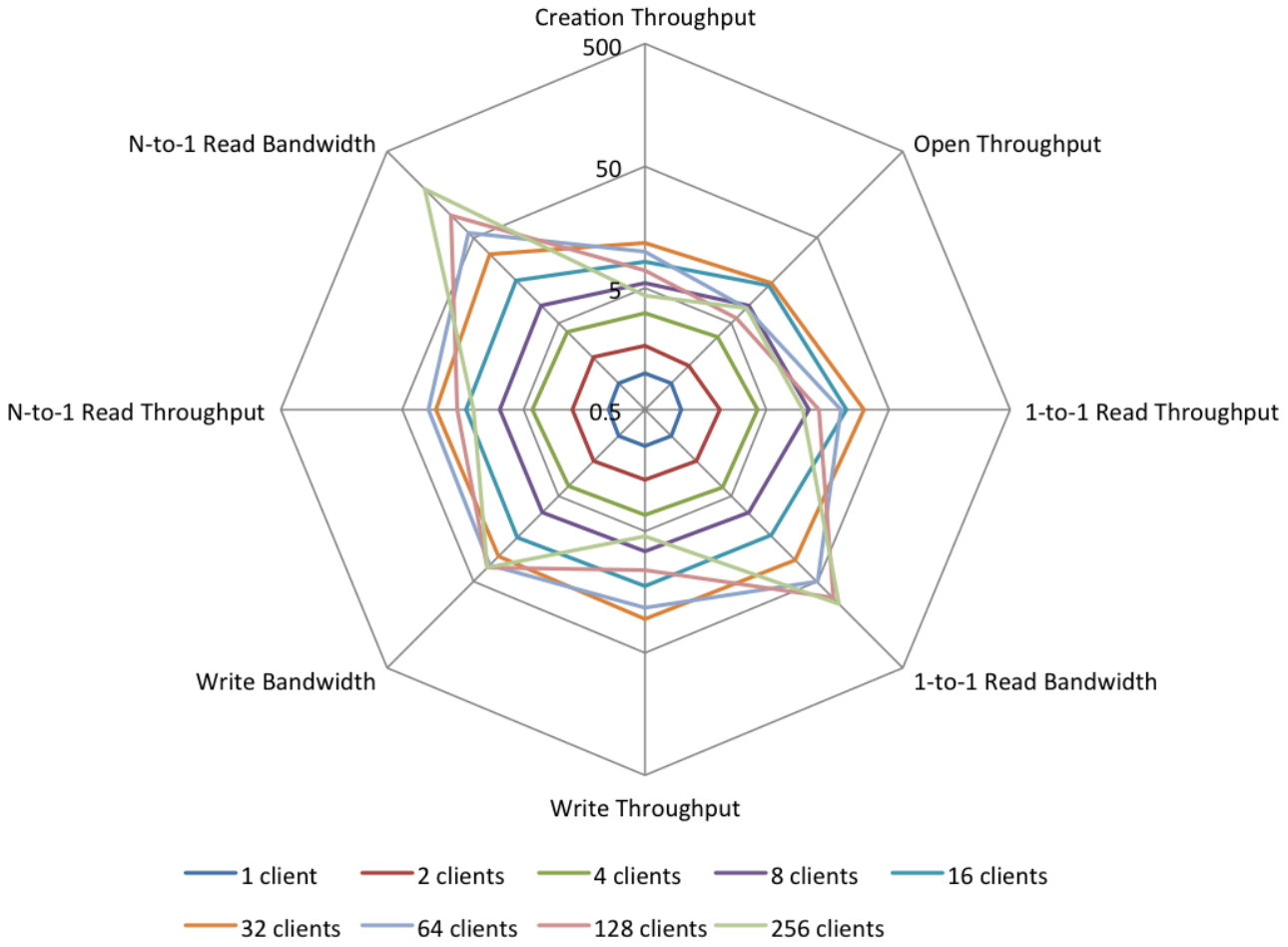

Figure 17: MTC Envelope (log scale). The performance on each metric (0.5 - 500) is relative to that on one client.

when the write is throughput bound. Higher throughput and bandwidth are desired for parallel scripting applications in order to reduce runtime.

We plot two heat maps of concurrent write bandwidth and throughput with our benchmark numbers and Equations 1 and 2. Figure 18 shows the bandwidth distribution with various file size on multiple scales, while Figure 19 shows the throughput distribution with the same input as Figure 18. When used to predict write performance, the two heat maps yield the same predictions, however, in Figure 18, the difference among the file size of 1 KB, 128 KB and 1 MB is barely discernible.

For a read workload, we denote N as the number of tasks, with each reading a file of size D_1 bytes at the same time and a common input file of size D_N shared among all tasks. On a group of computing resources that has C shared filesystem clients, the MTC envelope delivers 1-to-1 read bandwidth of B_1, throughput of T_1, N-to-1 read bandwidth of B_N and throughput of T_N. The read time consumption can be approximately expressed as

$$Time = \lceil N/C \rceil * (\max(C/T_1, C * D_1/B_1) + \max(C/T_N, C * D_N/B_N)) \quad (3)$$

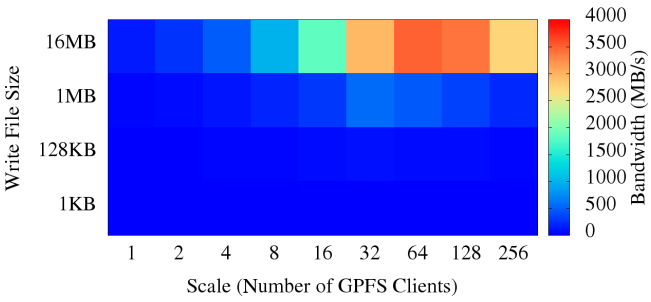

Figure 18: Heat map of write bandwidth

We do not compile the 1-to-1 read and N-to-1 read performance into heat maps because the benchmark numbers are measured directly. Users can refer to Figures 11, 12, 13, 14 for numbers to project read time consumption.

Using such information, a parallel scripting application programmer could obtain a rough picture of how much time was taken by I/O in the execution. For example, assuming we have an application stage with 8,192 tasks, each writing one output file of size 16 MB, and running on 64 GPFS

Table 4: Application I/O Performance Decomposition and Bounding Factor(s) Classification (bw=bandwidth, tput=throughput) on 32 GPFS Clients

		Montage				BLAST			CyberShake		
		mProject	mDiff	mFit	mBack	formatdb	blastall	merge	extract	seis	peakGM
Read	Metadata	**69.7%**	**82.0%**	23.3%	24.1%	1.7%	6.0%	1.9%	3.1%	1.4%	**82.4%**
	I/O	30.3%	18.0%	**76.7%**	**75.9%**	**98.3%**	**94.0%**	**98.1%**	**96.9%**	**98.6%**	17.6%
	Bound. Fact.	bw	tput	bw	bw	bw	bw	bw	bw	bw	tput
Write	Metadata	37.4%	**52.0%**	**52.0%**	37.7%	11.9%	**52.0%**	40.8%	4.7%	**52.0%**	**52.0%**
	I/O	**62.6%**	48.0%	48.0%	**62.3%**	**88.1%**	48.0%	**59.2%**	**95.3%**	48.0%	48.0%
	Bound. Fact.	bw	tput	tput	bw	bw	tput	bw	bw	tput	tput

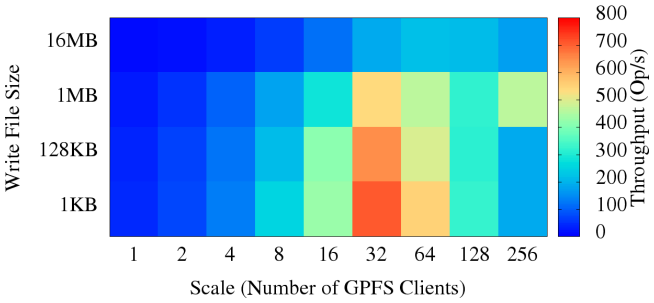

Figure 19: Heat map of write throughput

clients, then the write consumption can be estimated by the following steps. 16 MB writes are bandwidth dominated, so we look at the bandwidth heat map, and find that the bandwidth is about 3500 MB/s. The write has 8,192*16 MB of data to write, so it will take 8,192*16/3,500 = 37 seconds. Alternatively, if a programmer with a requirement for the I/O time might first determine what file size and scale combinations for the I/O time are valid on the shared file system, and then could choose from the candidate combinations.

4.2 I/O Performance Bounding Factors

To begin to investigate the I/O performance bounding factor(s) of the example applications, we map the applications' I/O behavior to the MTC envelope that we measured on GPFS on 32 clients. Table 4 illustrates the time consumption decomposition and bounding factors for each applications stage. The calculations were done based on the profile of each application and Equations 1 and 2. The MTC envelope on 32 GPFS clients shows file creation throughput of 1,367 Op/s, file open throughput of 1,974 Op/s, 1-to-1 read performance of 1,624 reads/s and 2,123 MB/s, N-to-1 read performance of 3,489 reads/s and 4,118 MB/s, and write performance of 1,481 writes/s and 3,429 MB/s.

Given an application stage profile, we compute the the read I/O consumption as the difference between the time consumption of reads and metadata operations. Write I/O consumption can be measured directly as shown in §3.6. We also compute the time consumption numbers from Equations 1 and 2 and determine which is larger to decide if an operation is bandwidth bound or throughput bound.

We see from the read time distribution for mProject and mDiff in Table 4 that for small file I/O, even though the metadata processing takes more time than the I/O, we can determine if the I/O is bandwidth-bound or throughput-bound. If we perform the same calculation on the MTC envelope on 64 clients, we would see a dramatical decrease of metadata and I/O throughput, which is the concurrency bounding factor of the applications' I/O behavior.

We will continue analysis of application performance within the constraints of the envelope in our future work.

4.3 I/O Performance Prediction

To predict the I/O performance of a given application or a stage of the application, we define two models.

- Coarse model: The coarse model maps the I/O traffic of an application stage to the **peak** metadata operation throughput and read/write bandwidth (MB/s) and operation throughput (Op/s) at a given scale.
- Fine model: Instead of just looking at the peak measured performance in the MTC envelope, the fine model takes into account the impact of the I/O file sizes on I/O bandwidth and throughput. We assume the bandwidth and throughput distribution along files sizes are linear. For example, Figure 12 shows the bandwidth of 1-to-1 read for file sizes of 1 KB, 128 KB, 1 MB, and 16 MB. The 1-to-1 read bandwidth for a file size of 8.5 MB is calculated as the average of the bandwidth of 1 MB and 16 MB. For multiple input and output files, we only compute the metadata operation time once, as the subsequent metadata operation congestion will be reduced by the diffusion of operations in those rounds, due to delays in the earlier round(s). The bandwidth factor is computed as the sum of time used for each file.

We compare the coarse model, the fine model, and the measured application I/O performance for three sample stages: Montage:mBack, CyberShake:peakGM and Montage:mProjectPP. These represent an I/O bandwidth-bound workload, a metadata throughput-bound workload, and a mixed bounding factor workload. We run 1024 tasks of each stage on 32 GPFS clients.

Table 5 shows the errors predicted by the coarse and fine models. We show the average of five runs of each application on 32 GPFS clients. Both the coarse and fine prediction of mProjectPP and mBack are based on the file access bandwidth, while the predictions of peakGM are based on the file access throughput because the file sizes in this stage are small.

We observe that the fine model predicts the stages with bandwidth-dominant read performance (mProjectPP and

Table 5: Application I/O Performance Prediction Error on 32 GPFS Clients

Stage	Phase	Time (secs)	Coarse Model Error	Fine Model Error
mProjectPP	Input	83.0	-60.9%	**+0.4%**
	Output	87.8	**+18.7%**	+45.7%
mBack	Input	116	-29.7%	**+17.5%**
	Output	48.6	**+32.0%**	+86.6%
peakGM	Input	12.2	+65.9%	+69.5%
	Output	42.7	+8.0%	+8.0%

mBack) well, while the coarse model predicts bandwidth-dominant write well. So our assumption in the fine model that the read bandwidth is linear with file size appears correct. The similar assumption in the fine model for write bandwidth fails. But the assumption in the coarse model of rounding the file size to the closest bandwidth of file size of {1 KB, 16 KB, 1 MB, 16 MB} seems to work well.

We also observe that the fine model predicts read better than write for bandwidth-dominant I/O. The reason is that in our runs, the first round of reads starts at the same time and the model correctly predicts that. On the other hand, write concurrency overhead is somewhat neutralized by the preceding reads, so we have more concurrency overhead in our models than actually exists.

A third observation is that for throughput-dominant reads in peakGM, neither model works well, as this stage is metadata operation bound, and it is more sensitive to the current workload of the machine. While the models seem to reasonably predict the throughput-dominant write in peakGM, the measured performance's standard deviation is 11.7 seconds with the average time consumption of 42.7 seconds.

5. RELATED WORK

Examples of alternative approaches to profiling the I/O performance of applications on specific systems include work on general HPC applications [30] and on FLASH [28]. In comparison, our approach can be used to predict I/O performance on different systems.

Previous benchmark include MADbench2 [6], which originated from the cosmic microwave background application MADCAP [5]. It measures the system's unique (one-file-per-process) and shared file (one-file-for-all-processes) access performance. It could also measure the improvement of the asynchronous I/O with computation and I/O overlapped on the timeline. However, it does not work well for applications that are metadata-intensive. The Effective I/O Benchmark [23] is a purely synthetic benchmark with predefined configurations that runs for a limited time to output a score as the performance metric. It is difficult to understand real application performance using the Effective I/O Benchmark. The SPIObench [31] and PIORAW [33] benchmarks are synthetic benchmarks to measure the concurrent read/write operations on unique files. They can measure the 1-to-1 read performance, but similarly to MADbench2, they overlook metadata-intensive applications. The IOR [13] benchmark is a highly-parameterized HPC storage benchmark, and previous work [29] has used it to model and predict the HPC application I/O performance with specified parameters combinations. With slightly modification, the IOR benchmark can be used a standard test framework for MTC envelope. It also comes with a limitation of MPI-dependency.

Many research groups have observed that metadata throughput is a bottleneck in today's shared file system. GIGA+ [21, 20] explores the idea of scalable directories in a hierarchical layout, and the work has been migrated in OrangeFS (PVFS) [8]. ZHT [16] distributes the metadata storage fully over storage servers, which could significantly improve the metadata operation throughput. Our previous AME work [36] has a similar metadata server design, but it is has a simplified implementation without the notion of directories. The work of Data Diffusion [27] and AMFS [35] explore the benefit of addressing data locality in parallel scripting applications. In the context of file system benchmarking, Data Diffusion [27] and AMFS [35] enhance the read/write bandwidth by redirecting the I/O to local storage. A similar approach has been implemented in HDFS [7] to facilitate MapReduce applications. The performance improvements to the applications from scalable metadata access and locality can be quantitatively profiled with the MTC envelope.

6. CONCLUSION

In this paper, we first studied the I/O behavior of parallel scripting applications that use a shared filesystem abstraction for inter-task communication on large scale computers. Then we defined the MTC envelope as a set of eight performance metrics: metadata query throughput, metadata creation throughput, 1-to-1 read throughput, 1-to-1 read bandwidth, N-to-1 read throughput, N-to-1 read bandwidth, write throughput, and write bandwidth. We also showed that the MTC envelope is sufficient to characterize the I/O behavior of parallel scripting application. Taking GPFS on BG/P as an example, we benchmarked the MTC envelope of GPFS at multiple scales, and studied its scalability. We believe the envelope model may be useful as a guide for aggregating many smaller operations into fewer larger ones, replacing shared filesystem accesses with scalable local filesystem usage, and spreading shared filesystem accesses in a manner that reduces locking and other forms of resource contention. Finally, we presented a way to convert performance measurements into heat maps that can guide programmers to make better use of existing shared filesystems for parallel scripting applications.

We believe this work can also benefit other types of applications such as HPC and MapReduce, as the same approach can be applied to those applications and hardware-software stacks. We are in the era of Big Data where many interesting applications are inevitably data-intensive, so such shared filesystem studies in other communities will help them use the current I/O and storage in a more efficient way.

Acknowledgements

This work was supported in part by the U.S. Department of Energy under the ASCR X-Stack program (contract DE-SC0005380) and contract DE-AC02-06CH11357. Computing resources were provided by the Argonne Leadership Computing Facility. We thank the ZeptoOS team in Argonne's MCS Division for their effective and timely support. We also thank the ALCF support team at Argonne. Some work by DSK was supported by the National Science Foundation, while working at the Foundation. Any opinion, finding, and

conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

7. REFERENCES

[1] M. Albrecht, P. Donnelly, P. Bui, and D. Thain. Makeflow: A portable abstraction for data intensive computing on clusters, clouds, and grids. In *Workshop on Scalable Workflow Enactment Engines and Technologies (SWEET) at ACM SIGMOD*, 2012.

[2] S. F. Altschul, W. Gish, W. Miller, E. W. Myers, and D. J. Lipman. Basic local alignment search tool. *Journal of Molecular Biology*, 215:403–410, 1990.

[3] Amazon simple workflow service. `http://aws.amazon.com/swf/`.

[4] Argonne Leadership Computing Facility. Intrepid file system information. `https://www.alcf.anl.gov/resource-guides/intrepid-file-systems`.

[5] J. Borrill. MADCAP: The microwave anisotropy dataset computational analysis package. In *Proceedings of the 5th European SGI/Cray MPP Workshop*, 1999.

[6] J. Borrill, L. Oliker, J. Shalf, and H. Shan. Investigation of leading HPC I/O performance using a scientific-application derived benchmark. In *Supercomputing, 2007. SC '07. Proceedings of the 2007 ACM/IEEE Conference on*, pages 1 –12, nov. 2007.

[7] D. Borthakur. HDFS architecture. `http://hadoop.apache.org/hdfs/docs/current/hdfs_design.pdf`.

[8] P. H. Carns, W. B. Ligon, III, R. B. Ross, and R. Thakur. PVFS: a parallel file system for linux clusters. In *Proc. of the 4th annual Linux Showcase & Conf. - Volume 4*, pages 28–28. USENIX Association, 2000.

[9] M. W. Deem, R. Pophale, P. A. Cheeseman, and D. J. Earl. Computational discovery of new zeolite-like materials. *J. of Phys. Chem. C*, 113:21353–21360, 2009.

[10] J. Elliott, I. Foster, K. Judd, E. Moyer, and T. Munson. CIM-EARTH: Philosophy, Models, and Case Studies. Technical Report ANL/MCS-P1710-1209, Argonne National Laboratory, Mathematics and Computer Science Division, Feb 2010.

[11] R. Graves, T. Jordan, S. Callaghan, E. Deelman, E. Field, G. Juve, C. Kesselman, P. Maechling, G. Mehta, K. Milner, D. Okaya, P. Small, and K. Vahi. CyberShake: A physics-based seismic hazard model for southern California. *Pure and Applied Geophysics*, Online Fir:1–15, May 2010.

[12] G. Hocky, M. Wilde, J. Debartolo, M. Hategan, I. Foster, T. Sosnick, and K. Freed. Towards petascale ab initio protein folding through parallel scripting. argonne. Technical Report ANL/MCS-P1612-0409, Argonne National Laboratory, Mathematics and Computer Science Division, April 2009.

[13] The ASCI I/O stress benchmark. `http://www.llnl.gov/asci/purple/benchmarks/limited/ior/`.

[14] J. C. Jacob, D. S. Katz, G. B. Berriman, J. C. Good, A. C. Laity, E. Deelman, C. Kesselman, G. Singh, M.-H. Su, T. A. Prince, and R. Williams. Montage: a grid portal and software toolkit for science-grade astronomical image mosaicking. *Intl. J. of Comp. Sci. and Eng.*, 4(2):73–87, 2009.

[15] D. S. Katz, J. C. Jacob, G. B. Berriman, J. Good, A. C. Laity, E. Deelman, C. Kesselman, and G. Singh. A comparison of two methods for building astronomical image mosaics on a grid. In *Proc. 2005 Intl. Conf. on Par. Proc. Works.*, pages 85–94, 2005.

[16] T. Li, X. Zhou, K. Brandstatter, D. Zhao, K. Wang, A. Rajendran, Z. Zhang, and I. Raicu. ZHT: A light-weight reliable persistent dynamic scalable zero-hop distributed hash table. In *to appear in Parallel & Distributed Processing Symposium (IPDPS), 2013 IEEE 27th International*. IEEE, 2013.

[17] P. Maechling, E. Deelman, L. Zhao, R. Graves, G. Mehta, N. Gupta, J. Mehringer, C. Kesselman, S. Callaghãn, D. Okaya, H. Francoeur, V. Gupta, Y. Cui, K. Vahi, T. Jordan, and E. Field. SCEC CyberShake workflows – automating probabilistic seismic hazard analysis calculations. In I. J. Taylor, E. Deelman, D. B. Gannon, and M. Shields, editors, *Workflows for e-Science*, pages 143–163. Springer, 2007.

[18] D. R. Mathog. Parallel blast on split databases. *Bioinformatics*, 19(14):1865–1866, 2003.

[19] A. Matsunaga, M. Tsugawa, and J. Fortes. Cloudblast: Combining mapreduce and virtualization on distributed resources for bioinformatics applications. In *eScience, 2008. eScience '08. IEEE Fourth International Conference on*, pages 222 –229, dec. 2008.

[20] S. Patil and G. Gibson. Scale and concurrency of giga+: file system directories with millions of files. In *Proceedings of the 9th USENIX conference on File and stroage technologies*, FAST'11, pages 13–13, Berkeley, CA, USA, 2011. USENIX Association.

[21] S. V. Patil, G. A. Gibson, S. Lang, and M. Polte. Giga+: scalable directories for shared file systems. In *Proceedings of the 2nd international workshop on Petascale data storage: held in conjunction with Supercomputing '07*, PDSW '07, pages 26–29, New York, NY, USA, 2007. ACM.

[22] A. Peters, M. E. Lundberg, P. T. Lang, and C. P. Sosa. High throughput computing validation for drug discovery using the DOCK program on a massively parallel system. In *Proc. 1st Annual Midwest Symposium on Computational Biology and Bioinformatics*, September 2007.

[23] R. Rabenseifner and A. Koniges. Effective file-I/O bandwidth benchmark. In *Euro-Par 2000 Parallel Processing*, volume 1900, pages 1273–1283. Springer Berlin / Heidelberg, 2000. 10.1007/3-540-44520-X_179.

[24] I. Raicu, I. Foster, A. Szalay, and G. Turcu. Astroportal: A science gateway for large-scale astronomy data analysis. In *Proc. TeraGrid Conf. 2006*, 2006.

[25] I. Raicu, I. Foster, and Y. Zhao. Many-task computing for grids and supercomputers. In *Proc. of Many-Task Comp. on Grids and Supercomputers, 2008*, 2008.

[26] I. Raicu, Z. Zhang, M. Wilde, I. Foster, P. Beckman, K. Iskra, and B. Clifford. Toward loosely coupled programming on petascale systems. In *Proc. IEEE/ACM Supercomputing 2008*, November 2008.

[27] I. Raicu, Y. Zhao, I. T. Foster, and A. Szalay. Accelerating large-scale data exploration through data diffusion. In *Proc. of 2008 Intl. Work. on Data-Aware Dist. Comp.*, DADC '08, pages 9–18. ACM, 2008.

[28] R. Ross, D. Nurmi, A. Cheng, and M. Zingale. A case study in application I/O on Linux clusters. In *Proceedings of the 2001 ACM/IEEE conference on Supercomputing (CDROM)*, pages 11–11. ACM, 2001.

[29] H. Shan, K. Antypas, and J. Shalf. Characterizing and predicting the I/O performance of HPC applications using a parameterized synthetic benchmark. In *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, SC '08, pages 42:1–42:12, Piscataway, NJ, USA, 2008. IEEE Press.

[30] E. Smirni and D. Reed. Lessons from characterizing the input/output behavior of parallel scientific applications. *Performance Evaluation*, 33(1):27–44, 1998.

[31] Streaming parallel I/O benchmark. `http://www.nsf.gov/pubs/2006/nsf0605/spiobench.tar.gz`.

[32] R. Thakur and R. Rabenseifner. Optimization of collective communication operations in MPICH. *Intl. J. of High Perf. Comp. Applications*, 19:49–66, 2005.

[33] The PIORAW Test. `http://cbcg.lbl.gov/nusers/systems/bassi/code_profiles.php`.

[34] M. Wilde, I. Foster, K. Iskra, P. Beckman, Z. Zhang, A. Espinosa, M. Hategan, B. Clifford, and I. Raicu. Parallel scripting for applications at the petascale and beyond. *Computer*, 42:50–60, 2009.

[35] Z. Zhang, D. Katz, J. Wozniak, A. Espinosa, and I. Foster. Design and analysis of data management in scalable parallel scripting. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, page 85. IEEE Computer Society Press, 2012.

[36] Z. Zhang, D. S. Katz, M. Ripeanu, M. Wilde, and I. Foster. AME: An anyscale many-task computing engine. In *6th Work. on Workflows in Support of Large-Scale Science (WORKS11)*, 2011.

[37] Y. Zhao, M. Hategan, B. Clifford, I. Foster, G. Von Laszewski, V. Nefedova, I. Raicu, T. Stef-Praun, and M. Wilde. Swift: Fast, reliable, loosely coupled parallel computation. In *Services, 2007 IEEE Congress on*, pages 199–206. IEEE, 2007.