ANALYZING THE I/O BEHAVIOR OF SUPERCOMPUTER APPLICATIONS

Ethan L. Miller and Randy H. Katz

University of California at Berkeley
Berkeley, California

ABSTRACT

This paper describes the collection and analysis of supercomputer I/O traces on a Cray Y-MP. Analyzing these traces, which came primarily from programs with high I/O requirements, shows the file system I/O patterns that these applications exhibit.

INTRODUCTION

Over the last few years, CPUs have seen tremendous gains in performance. I/O systems and memory systems, however, have not enjoyed the same rate of increase. As a result, supercomputer applications are generating more data, but I/O systems are becoming less able to cope with this huge volume of information. Multiprocessors are exacerbating this problem, as the number of disks and tape drives in the I/O system and thus aggregate I/O bandwidth, increase. Bandwidth is not usually scaled up at the same rate as the aggregate processing speed, however. According to Amdahl's metric, each MFLOP should be accompanied by one Mbit/second of I/O. Providing this requires correct matching of bandwidth capability to application bandwidth requirements, and using buffering to reduce the peak bandwidth that the I/O system must handle. To better determine the necessary hardware bandwidth and software buffer sizing and policies, applications' I/O patterns must first be analyzed. This requires gathering I/O traces from real applications, which we have done.

BACKGROUND

Supercomputer environment

The majority of the I/O traces gathered in this study came from applications running on the Cray Y-MP 8/8128 at NASA Ames [1,2]. This computer has eight processors, each with a 6-ns cycle time. The system has a total of 128 MW (each word is eight bytes long) shared among the eight processors. The I/O system has 35.2 GB of storage on high-speed disks, each capable of sustaining 9.6 MB/second; a 256-MW solid-state disk (SSD) acting as an operating system-managed cache for a single filesystem; and several terabytes of nearline and offline tape storage. The tape storage is divided into two parts—a nearline storage facility called the Mass Storage System (MSS-II) [3], which can automatically mount tapes with requested data, and the extensive offline tape library which requires operator intervention. The NASA Cray system already has the maximum configuration of Y-MP memory, so I/O problems cannot be alleviated simply by adding more primary memory.

The UNICOS process scheduling mechanism at NASA affects the way programmers choose to structure their implementations, and thus I/O demands seen by the operating system. Batch jobs, which include any program requiring over 10 minutes of Cray CPU time, are queued according to two resource requirements—CPU time and memory space. As the Cray Y-MP does not have virtual memory, all of a program's memory must be contiguously allocated when the program starts up. To simplify memory allocation, each queue is given a fixed memory space to run its programs in. A job ready to run and residing in memory is run on any of the eight processors that are available. It runs until it must wait for a disk I/O, at which time it is suspended. The program remains in memory while it waits, and another program that is ready to run is given to that processor. Since there are eight processors, there must be at least eight jobs in memory to keep all the processors busy, assuming no jobs use more than one processor. In practice, $n+1$ or $n+2$ jobs in memory will keep n processors busy. To insure this many ready jobs despite memory space limitations, a single queue may have multiple jobs in memory at once, especially if the queue is small. Thus, for a given CPU time requirement, turnaround time is shortest for the job that requires the least memory. Programmers take advantage of this by structuring their program to use smaller in-memory data structures while using the I/O system for staging.

Because of their high-speed vector processing ability, supercomputers are ideally suited to problems that require manipulations of large arrays of data [4]. These problems include computational chemistry, computational fluid dynamics, structural dynamics, and seismology, to name a few. These disciplines all require large numbers of floating-point computations that are usually vectorizable over large data sets: from hundreds of megabytes up to tens or hundreds of gigabytes for some seismic computations. In most cases, the application performs multiple iterations over the data set, as when simulating a model through time. The next section will discuss the applications traced for this study.

CH3039-5/91/0000/0051/$01.00 © 1991 IEEE

This paper appeared in the Eleventh IEEE Symposium on Mass Storage Systems, Monterey, CA, October 1991.

APPLICATIONS TRACED

We gathered traces from a variety of applications running on Cray Y-MPs. The majority of these programs were run at NASA Ames. We chose to trace applications with high I/O rates, both in megabytes per second and I/Os per second. While many supercomputer applications do not perform large amounts of I/O, we decided to concentrate on those that do a lot of I/O. I/O-intensive applications stress the I/O system more, revealing performance bottlenecks. Those that perform little I/O are easy to characterize, as will be shown by the two traces that did little I/O.

The first group of applications is the climate models. These included **gcm** (General Circulation Model) and **ccm** (Community Climate Model). The two applications differed in their approach to data organization—**gcm** kept all of its data in memory, while **ccm** used less memory and staged data to and from disk. **Venus** was also a "climate" model which modeled Venus' atmosphere instead of Earth's. It required far more I/O than the other two models because it used a very small in-memory array. While it did more I/O than the other two programs, it often had a shorter turnaround time since it was able to fit in the queue for jobs requiring little memory.

Bvi and **les** were other computational fluid dynamics (CFD) applications which modeled smaller systems. **Bvi** modeled the interaction between a helicopter blade and the air around it. It used a relatively large in-memory array, but still did a large amount of I/O. It was the only program that was explicitly designed to use the SSD. This was reflected in the small transfers the program made; since they were to the low-latency SSD rather than the high-latency disk, the program did not incur a high per-access penalty. **Les**, another CFD program, also requested a lot of I/O. It was the only program that explictly used asynchronous I/O; as a result, it rarely had to wait for data to be staged into or out of memory.

Forma used sparse matrices to solve a structural dynamics problem. This program was ported (with few changes) from a Cray 1, a machine that was even more memory-limited than the Cray Y-MP. Thus, it only used 2 MW of main memory, but did huge amounts of I/O to shuttle pieces of the matrices in and out of memory.

The final program, **upw**, performed an approximate polynomial factorization. It did the least I/O of any program traced. The program read a small input file, computed for ten CPU minutes, and wrote out its answer. While no other program we traced showed this behavior, it is a representative I/O pattern for some applications.

Basic information about the applications is summarized in Figure 1. **Running time** is the amount of CPU time (not wall time) each application required. All of the other numbers are relative to running time. **Total data size** is the sum of the sizes of all the files the application accessed.

TRACING METHODS

Information traced

The traces gathered included two types of data. First, they included information sufficient to reconstruct the sequence of I/O system calls that an application made. Second, they included wall clock timestamps so the actual performance (including other programs) could be measured. We focused on the first type of data.

Each trace record contained a file identifier, I/O size, and I/O offset within the file. The file identifier was generated by the system call that opened the file; if a file was opened twice by the same program, it had two different file identifiers. All records also had three timestamps. Two of these were in wall clock time—I/O start time and I/O duration. These measured the wall clock time between I/Os and how long the program was suspended for each I/O. The third timestamp recorded CPU time between I/O requests. This timestamp was independent of how heavily loaded the system was, as it only measured how much CPU time the application used between I/Os. This timestamp was the most important in analyzing the traces,

Application	Running Time (sec)	Total data size (MB)	Total I/O done (MB)	Number of I/Os	Avg I/O size (MB)	MB/sec	I/Os/sec
bvi (CFD)	1258	171.0	22,835.0	1,380,457	0.016	18.20	1097.0
ccm (climate)	205	11.6	1,812.0	54,125	0.031	8.80	264.0
forma (structural)	206	30.0	15,155.0	475,826	0.030	73.60	2310.0
gcm (climate)	1897	229.0	266.2	7,953	0.031	0.14	4.2
les (large eddy)	146	224.0	7,803.0	22,384	0.317	53.40	153.0
venus (climate)	379	55.2	16,712.0	34,904	0.032	44.10	92.0
upw (polynomial)	596	- - -	61.5	1,840	0.445	0.10	3.1

Figure 1. Characteristics of the traced applications.

as it was the only one that showed program behavior rather than system load.

Trace gathering methods

All of the data gathered on the Cray Y-MP were logical-level traces. This data included logical file numbers, file offsets, request sizes, and wall clock and process clock timestamps. Because no collected data were internal to the operating system (as physical disk block numbers would be), all the data could be collected by code running at user level. Thus, no modifications to the operating system were necessary. This was a distinct advantage on the Cray, since it would have been very difficult to obtain the dedicated time necessary to debug changes to the operating system.

Instead of modifying the operating system, we changed the user libraries dealing with I/O. Cray provides data collection hooks in standard Unicos 5.0 system libraries. These hooks provide aggregate data on a program's I/O, such as the total number of bytes requested and the average and maximum times to do an I/O. Major events such as file opens and process forks are also recorded. We modified the libraries to record information for each individual **read** and **write** call as well. This information was buffered up and sent, via Unix pipe, to a process that collected the data and wrote it out to a trace file. The tracing mechanism and trace format are described in more detail in [5].

There was very little CPU overhead required to collect traces in this manner. The tracing code was small relative to the operating system code executed for each system call, and was only active during system calls. This limited overheads to less than 20% of I/O system call time. Total overhead depended on how many times I/O system calls were made, but was typically only a few percent of execution time.

I/O PATTERN ANALYSIS

There has been much analysis of overall supercomputer performance in both I/O and CPU usage. However, the I/O usage studies have focused primarily on overall system performance over relatively long periods, ranging from many minutes to several weeks, as in [6]. These studies are useful for evaluating current systems, but are ill-suited for inferences to future systems. Many parameters may change, and it is hard to isolate the parameters that affect system performance. Studying the individual applications can make it easier to predict the interactions between applications for a new system.

Types of application I/O

All of the I/O accesses made by the programs can be divided into three types—required, checkpoint, and data staging. Required I/Os are similar to hardware cache misses called *compulsory* in [7]. They consist of I/Os that must be made to read a program's initial state from the disk and write the final state back to disk. For example, a program might read a configuration file and an initial set of data points. It would write out the final set of data points along with graphical and textual representations of the results after it finished. These I/Os, however, do not contribute much to the I/O rate between CPU and disk. For a program that runs 200 seconds, reading 50 MB of configuration data and writing 100 MB of output, the overall data rate is 0.75 MB/second. This rate is sustainable by most workstations. While the peak I/O rates at the start and the end are high, they will only occupy a small fraction of the total running time of the program. **Upw** and **gcm** are examples of programs that only do compulsory I/O.

Checkpoints, the second type of I/O, are used to save the state of a computation in case of a hardware or software error which would require the simulation to be restarted. A checkpoint file generally consists of some subset, possibly complete, of the program's in-memory data. Checkpoints are generally made every few iterations, though making them too often slows the program down unnecessarily. The application writer balances the cost of writing the checkpoint against the cost of redoing lost iterations of the simulation. The likelihood of failure determines the number of iterations between checkpoints. Since checkpoints occur multiple times per program, they add more to the bandwidth requirement than required I/O, but they also do not place a continuous high demand on the I/O system. For a program that saves 40 MB of state every 20 CPU seconds, the average I/O rate is only 2 MB/second, far less than the maximum rate most supercomputers provide. As with required I/Os, dealing with peak rates may present a problem, but since the I/Os occur relatively infrequently, it is easy to have another program ready to run (and not in the checkpoint stage itself) while the first program is waiting for checkpoint I/Os to complete.

The third type of I/Os are those done because the memory allocated to the problem is insufficient to hold the entire problem. These I/Os are the equivalent of paging under a paging virtual memory operating system, but they are generally done under program control because many supercomputers lack paging. Even when paging exists, the program is better able than the operating system to predict which data it will need. Unlike the other two types of I/O above, memory-limitation I/O must be done on every iteration of the algorithm. The entire data set is usually

shuttled in and out of memory at least once, and perhaps more often. If each data point consists of 3 words and requires 200 floating-point operations, there must be 24 bytes of I/O for every 200 FLOPS (this is quite close to Amdahl's metric, which would require 200 bits, or 25 bytes of I/O for those 200 FLOPS). For a 200-MFLOP processor, the average sustained rate will be almost 25 MB/second, far more than either the compulsory I/O data rate or the checkpoint I/O data rate. Peak rates are higher still, and in fact are higher than 200 MB/second of requests sustained over several CPU seconds.

I/O access characteristics

The I/O accesses that the applications make can be characterized in several ways. These included the total amount of I/O, the read/write ratio both overall and for given files, and the size of each individual I/O, again overall and for each file. In looking at these characteristics, however, only "large" files were considered. In most cases, these files were over a few megabytes long, and some were hundreds of megabytes long. While "small" files, which include parameter files and human-read text output, are important, they do not contribute much to the overall I/O that a supercomputer application must do, as their contribution is dwarfed by accesses to large machine-generated data files.

Prog	Reads (MB/sec)	Writes (MB/sec)	Avg I/O size (KB)	R/W ratio
bvi	12.3000	5.340	16.1	2.310
ccm	4.2500	3.960	31.9	1.070
forma	62.2000	5.680	30.4	11.00
gcm	0.0107	0.120	31.9	0.089
les	24.0000	25.200	325.0	0.950
upw	0.0012	0.010	32.7	0.120
venus	26.4000	14.700	456.0	1.800

Figure 2. Data rates of the traced applications.

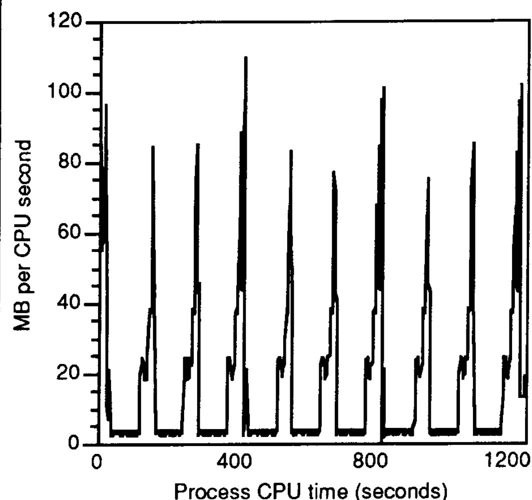
The only applications that had read/write ratios much under one were **gcm** and **upw**, as can be seen in Figure 2. They were the programs that did not do much I/O in the first place, since they did little I/O other than compulsory writes. The programs that did higher amounts of I/O had higher read/write ratios because, for those programs, the disk was used to hold large parts of the array. For each cycle of the algorithm, each section of the data is written once. However, that data may be read more than once so it can be used in the computation in different places. This pattern will remain no matter how large the memory of the system gets, since a larger memory will simply encourage larger problems, which will keep the same patterns. A file cache will not greatly change the read/write ratio to disk. The files are usually so large that they will not fit into the cache. Since the entire file is both read and written each iteration, there are no "hot" blocks that can remain in the cache between iterations. A cache might, however, decrease the read/write ratio to disk slightly because "paging" the data array might show spatial locality for reads.

Access size varied between programs, but was relatively constant within programs. The access size was completely under the programmer's control, so it varied according to how the algorithm was implemented. As seen in Figure 2, accesses on the large files ranged from 32 KB to 512 KB. The notable exception was **bvi**, which used the SSD for most of its "disk" accesses. There was no seek penalty for the SSD, so the small I/O penalty was much less than it would be for a normal disk. A SSD access still paid operating system overhead and transfer time, but it did not incur any latency as a disk access would.

Cycles in program I/O

Since all of the programs implemented iterative algorithms, the programs' I/O patterns followed cycles that matched the iterations of the program. Often, the data in the files would be read in the same sequence and with the same I/O request size each cycle. Even when the sequence was not the same between cycles, each program had a typical I/O request size that stayed constant throughout the program. Times of high data request rates also followed a pattern; request rate peaks were generally evenly spaced through the program's execution.

I/O was bursty, as expected, but the bursts came in cycles. The demand patterns for all of the cycles in a single application were remarkably similar, as Figures 3 and 4 show.



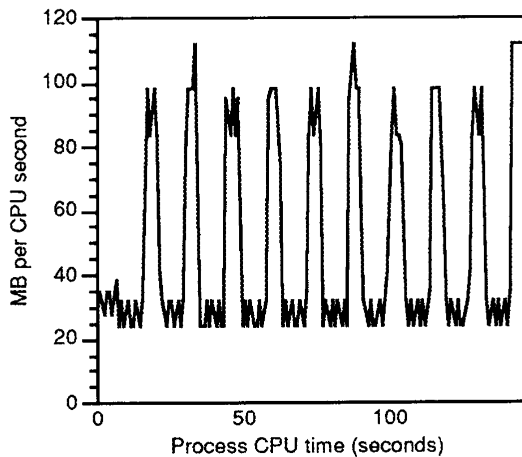


Figure 4. Data rate over time for les.

File reference patterns also followed cycles. This was especially true for algorithms that operated on an unchanging array that was larger than the program's memory size. For such applications, the reference patterns were essentially identical from cycle to cycle. For other applications, the array might change between iterations of the algorithm. For example, a common method in a CFD problem is to create more data points in areas of interest for detailed examination. Since these areas cannot be predicted in advance, the program itself identifies the areas and creates more points, changing the data array and the disk reference patterns.

Mass storage implications

The mass storage systems attached to a supercomputer are only involved in some of the I/Os that applications request. In particular, each file is read from the mass storage system (MSS) and written at most once to the MSS for a single execution of an application. For the MSS, then, the most important number is the data set size. While the total I/O done varies from 61 MB to 22 GB, the data set sizes vary much less—from 55 MB to 229 MB. This suggests that MSS I/O requests are more closely linked to processor speed than they are to memory size. In a system with an infinitely large memory, only compulsory and checkpoint I/Os would be requested. This would eliminate a large fraction of the I/Os to disk. However, the data rate to the MSS, which is proportional to the amount of compulsory I/O, would remain constant. Higher MFLOPS would increase the data rate to the MSS, though, since the CPU could process more data in a given time, thus increasing the amount of compulsory I/O a program would need.

CONCLUSIONS

While much attention has been given to CPU performance in supercomputers, the I/O system, which includes the file cache, SSD, disks, and tape storage, will play an increasingly larger role in utilizing the CPU efficiently.

We have classified application I/Os into three categories—required, checkpoint, and data staging—and shown how memory size and CPU speed are likely to affect each category. An analysis of the data shows that data staging I/O dominates when it is present. If an application requests data staging I/O, its read/write ratio is greater than 1; otherwise, its read/write ratio will be less than 1. I/O requests are often sequential and cyclical; this information can be used to predict a program's I/O demands and anticipate them, thus reducing the load on the I/O system.

REFERENCES

1. *NAS User Guide, Version 5.0*, NAS Systems Division, NASA Ames Research Center, Moffett Field, CA, January 1990.

2. Bonifas, Cathy, "Searching For a Unix Mass Storage System For a Supercomputer Environment," *Digest of Papers*, Proc. Tenth IEEE Symposium on Mass Storage Systems, May 1990, pp. 129-133, (1990).

3. Henderson, Robert, and Alan Poston, "MSS-II and RASH," *Conference Proceedings*, Winter 1989 USENIX Technical Conference, pp. 65-84.

4. Peterson, et. al., "Supercomputer Requirements for Selected Disciplines Important to Aerospace," *Proceedings of the IEEE*, Vol. 77, No. 7 (July 1989), pp. 1038-1054.

5. Miller, Ethan, "Input/Output Behavior of Supercomputing Applications," *Tech. Report No. UCB/CSD 91/616*, University of California, Berkeley, January 1991.

6. Williams, Elizabeth, et. al., "The Characterization of Two Scientific Workloads Using the Cray X-MP Performance Monitor," *Proceedings of Supercomputing '90*, November, 1990.

7. Hill, Mark, "Aspects of Cache Memory and Instruction Buffer Performance," Ph.D. dissertation, *Tech. Report No. UCB/CSD 87/381*, University of California, Berkeley, November 1987.