Understanding and Improving High-Performance I/O Subsystems

FINAL REPORT

(8/1/93-9/30/96)

Tarek A. El-Ghazawi, Associate Research Professor
and
Gideon Frieder, A. James Clark Professor and Dean

Department of Electrical Engineering and Computer Science
School of Engineering and Applied Science
The George Washington University
Washington, D.C. 20052
(202)994-5507
E-mail: *tarek@seas.gwu.edu*

Students/Research Assistants:
Mike R. Berry and Sorin Nastea

This research program has been conducted in the framework of the NASA Earth and Space Science (ESS) evaluations led by Dr. Thomas Sterling. In addition to the many important research findings for NASA and the prestigious publications, the program has helped orienting the doctoral research program of two students towards parallel input/output in high-performance computing. Further, the experimental results in the case of the MasPar were very useful and helpful to MasPar with which the P.I. has had many interactions with the technical management. The contributions of this program are drawn from three experimental studies conducted on different high-performance computing testbeds/platforms, and therefore presented in 3 different segments as follows.

1. Evaluating the parallel input/output subsystem of a NASA high-performance computing testbeds, namely the MasPar MP-1 and MP-2;
2. Characterizing the physical input/output request patterns for NASA ESS applications, which used the Beowulf platform; and
3. Dynamic scheduling techniques for hiding I/O latency in parallel applications such as sparse matrix computations. This study also has been conducted on the Intel Paragon and has also provided an experimental evaluation for the Parallel File System (PFS) and parallel input/output on the Paragon.

This report is organized as follows. The summary of findings discusses the results of each of the aforementioned 3 studies. Three appendices, each containing a key scholarly research paper that details the work in one of the studies are included.

SUMMARY OF FINDINGS

MasPar Evaluations

This work has shown that programmers of I/O-intensive scientific applications can tune their programs to attain good I/O performance when using the MasPar. They should be at least aware of their I/O configuration, the specific I/O RAM size and how it is locally partitioned in an attempt to partition data into files that can fit into the I/O RAM. The work further establishes that system managers are also encouraged to understand the I/O resource requirements of the applications running on their machines and tune the I/O RAM configuration for best performance. In specific, partitioning the I/O RAM among disk reads, disk writes, data processing unit (DPU) to front end communications, and interprocessor communications should be based on an understanding of the most common needs of the local application domain. Finally, the work has demonstrated that a full MasPar configuration with MPIOC^{TM} and a full I/O RAM has potential for delivering scalable high I/O performance. However for this to happen the I/O RAM management should make good attempt to prefetch anticipated data. Further, the I/O RAM partitioning strategy should be more flexible by using cache blocks for different purposes as dynamically needed by the applications. At the least, files smaller than the I/O RAM size should be cacheable. Finally, the sustained performance of the disk arrays remains to be the clear bottleneck and is likely to limit the overall performance of parallel I/O systems for some time to come. For more details on this study, refer to appendix A.

Physical I/O Requests of ESS Applications

This work has clearly shown that device driver instrumentation has the ability to distinguish among the different activities in the system, small explicit requests (less than page size), paging(4KB each in this case), and large objects (such as images). Further, it was shown ESS codes have high spatial I/O access locality, 90% of accesses into 10% of space. On the other hand, temporal locality was measured as frequency of accesses and observed to be as high as 6 repeated accesses per second. In general, astrophysics simulation codes (PPM and Nbody) have similar I/O characteristics and have shown very little I/O requirements for the used problem sizes. Wavelet code, however, required a lot of paging due to the use of many different files for output and scratch pad manipulations, and could benefit from some tuning to improve data locality. It is therefore advised that a strategy for file usage and explicit I/O requests for this code be developed to do so. On the system side, Linux tends to allow larger physical requests when more processes are running, by allocating additional blocks for I/O. It is therefore recommended that Linux file caching should be further investigated and optimized to suite the big variability in the physical requests of NASA ESS domain. This study was done in collaboration with Mike R. Berry (GWU).

Dynamic I/O Scheduling and PFS Evaluations

Using the worker-manager paradigm, we have introduced a dynamic I/O scheduling algorithm which maximizes I/O latency hiding by overlapping with computations at run-time, and is also capable of balancing the total load (I/O and processing). Using sparse matrix applications as a test case, we have shown empirically that such scheduling can produce performance gains in excess of 10%. Much higher improvement rates are expected when the non-zero elements are distributed in a skewed manner in sparse matrix applications. The end to end (including I/O) scalability of such applications was studied and was shown to be very satisfactory under these scheduling schemes. In addition, the Paragon parallel file system (PFS) was evaluated and its various ways of performing collective input/output were studied. It was shown that the performance of the various calls depend heavily on the way of managing the file pointer(s). Calls that allow concurrent asynchronous access of processors to their respective blocks, but provide ordering at the user level performed better than the rest. This study was conducted in collaboration with Sorin Nastea (GMU) and Ophir Frieder (GMU).

LIST OF PUBLICATIONS FROM PROJECT

PAPERS

[1] Mike Berry and Tarek El-Ghazawi. " Parallel Input/Output Characteristics of NASA Science Applications". Proceedings of the International Parallel Processing Symposium (IPPS'96), IEEE Computer Society Press. Honolulu, April 1996.

[2] Tarek El-Ghazawi. "Characteristics of the MasPar Parallel I/O System". Frontiers'95, IEEE Computer Society, McLean, VA, February 1995.

[3] S. Nastea, O. Frieder, and T. El-Ghazawi. " Parallel Input/Output Impact on Sparse Matrix Compression". Proceedings of the Data Compression Conference (DCC'96), IEEE Computer Society Press. Snowbird, April 1996.

TECHNICAL REPORTS

[4] Tarek El-Ghazawi. " I/O Performance Characteristics of the MasPar MP-1 Testbed. CESDIS TR-94-111

[5] Tarek El-Ghazawi. "Characteristics of the MasPar Parallel I/O System". CESDIS TR-94-129.

[6] M. Berry and T. El-Ghazawi. "An Experimental Study of the I/O Characteristics of NASA Earth and Space Science Applications". CESDIS TR-95-163.

[7] S. Nastea, T. El-Ghazawi, and O. Frieder. "Parallel Input/Output Issues in Sparse Matrix Computations". CESDIS TR-96-170.

[8] M. Berry and T. El-Ghazawi. "MIPI: Multi-level Instrumentation of Parallel Input/Output". CESDIS TR-96-176.

SUBMITTED MANUSCRIPTS

[9] S. Nastea, T. El-Ghazawi, and O. Frieder. "Parallel Input/Output Issues in Sparse Matrix Computations". Submitted to IPPS'97.

[10] S. Nastea, T. El-Ghazawi, and O. Frieder. "Parallel Input/Output for Sparse Matrix Computations". Submitted to the Journal of Supercomputing.

[11] S. Nastea, T. El-Ghazawi, and O. Frieder. "Impact of Data Skewness on the Performance of Parallel Sparse Matrix Computations". Submitted to the IEEE Transactions on Parallel and Distributed Systems.

APPENDIX A:

MASPAR MP-1 AND MP-2 I/O EVALUATIONS

Appeared in the IEEE CS Frontiers of the Massively Parallel Computations Proceedings, February 1995

Characteristics of the MasPar Parallel I/O System

Tarek A. El-Ghazawi[1]

Department of Electrical Engineering and Computer Science
The George Washington University
Washington, D.C. 20052
tarek@seas.gwu.edu

Abstract

Input/output speed continues to present a performance bottleneck for high-performance computing systems. This is because technology improves processor speed, memory speed and capacity, and disk capacity at a much higher-rate. Developments in I/O architecture have been attempting to reduce this performance gap. The MasPar I/O architecture includes many interesting features. This work presents an experimental study of the dynamic characteristics of the MasPar parallel I/O. Performance measurements were collected and compared for the MasPar MP-1 and MP-2 testbeds at NASA GSFC. The results have revealed many strengths as well as areas for potential improvements and are helpful to software developers, systems managers, and system designers.

1. Introduction

This study is aimed at the experimental performance evaluation of the MasPar I/O subsystem. Many I/O benchmarks have been developed and used by researchers for such investigations. A representative set of them is discussed here. These benchmarks can be divided into two main categories: application benchmarks and synthetic benchmarks. Selecting one over the other depends on the goals of the evaluation study. This is because application benchmarks are unique in providing performance results that are understandable to application-domain experts. Meanwhile, synthetic benchmarks (particularly those that are based on parameterized workload models [Jain91]) have the potential for providing insightful understanding of the architectural characteristics of the underlying I/O subsystem.

Among the I/O application benchmarks are Andrew [Howard88], TPC-B[TPCB90], and Sdet [Gaede81,Gaede82]. Andrew is more of a file system benchmark. As a workload, it copies a file directory hierarchy, then reads and compiles the new copy. Clearly, much of this work is not necessarily I/O. TPC-B [TPCB90] is a transaction processing benchmark aimed at evaluating machines running database systems in a banking environment. Despite being I/O intensive, TPC-B takes into account database software. The workload generates, using simulated customer random requests, debit-credit transactions to read and update account, branch, and teller balances. Sdet is another I/O benchmark which is produced by the System Performance Evaluation Cooperative (SPEC). SPEC produces independent standard benchmarks [Scott 90] and it is well known for its CPU performance benchmark, SPEC Release 1.0. Sdet is a part of their System Development Multitasking (SDM) suite. The workload is based on software-development environment including text editing and compiling.

Many synthetic I/O benchmarks were also developed. The list includes IOStone [Park90], LADDIS [Nelson92], and Willy [Chen92]. IOStone generates request patterns that approximate locality properties. IOStone, however, does not consider concurrency. All requests are generated using a single process. LADDIS, on the other hand, is a multi-vendor benchmark intended to evaluate Client/Server Network File Systems (NFS) and is actually based on NFSStone [Shein89]. Willy, however, represents a good step on the right direction. Willy is based on a parametrized workload model and allows for gaining architectural insight from evaluations. It is described [Chen92] as self-scaled and predictive. Scaling, in the context of the author, refers to changing one workload parameter while fixing the rest. Prediction refers to the ability to predict the performance under workloads that were not actually used, but should be within

[1]This work is supported by NASA HPCC Program for Basic Research through CESDIS University Program in High-Performance Computing, grant # 5555-18

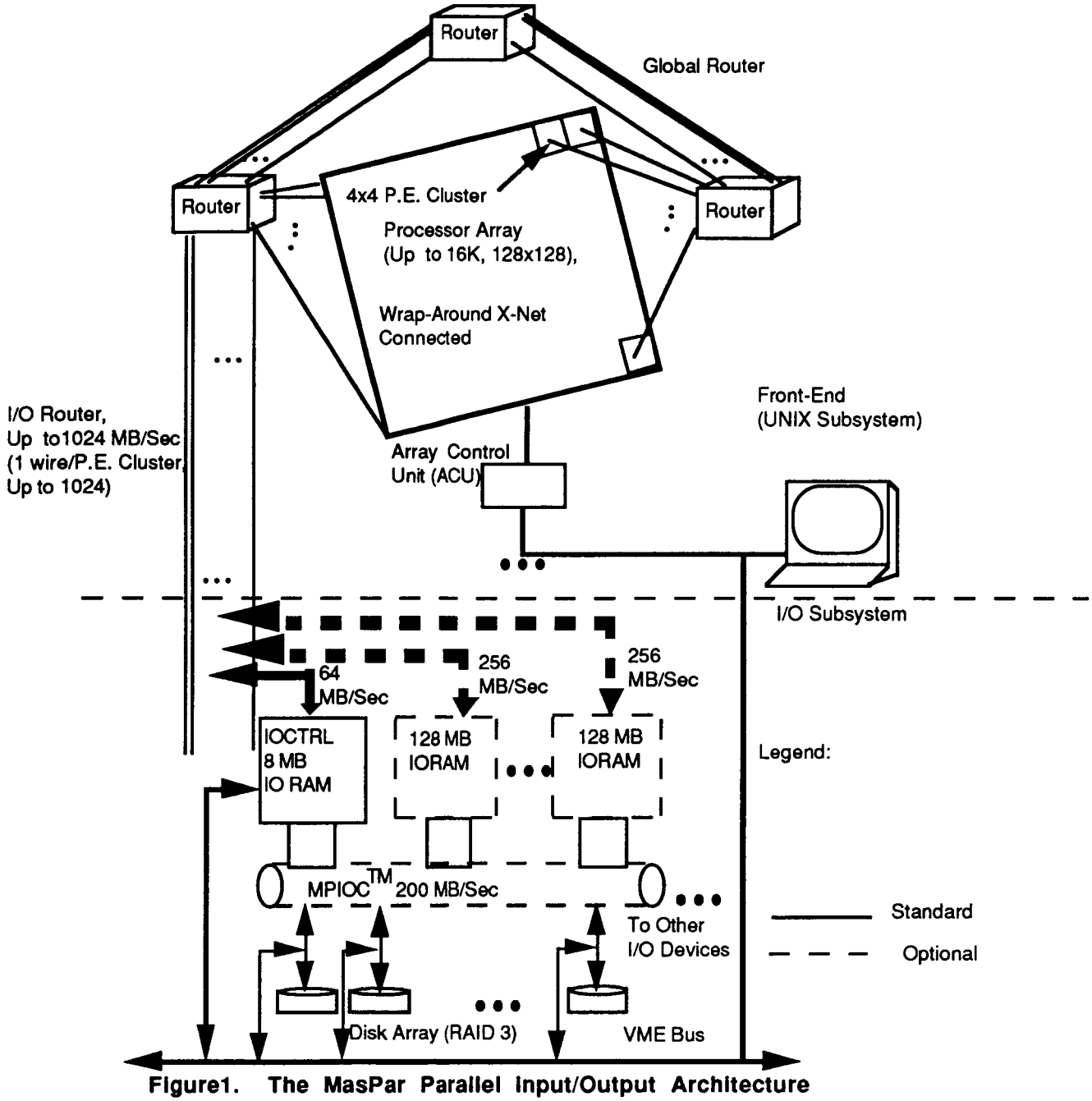


Figure1. The MasPar Parallel Input/Output Architecture

10%-15% of previously used workloads. Willy mainly generates workloads for workstations and multiprocessor systems with a few number of processors.

Since the goal of this study is to explore the characteristics of the MasPar parallel I/O subsystem, synthetic benchmarking was used to accomplish that in a massively parallel SIMD architecture. The workloads generated were, therefore, designed to isolate the behaviors of the PEs-to-I/O communication channel speed, the I/O cache management, and the disk array. In that sense, the workloads generated here are closely related to those produced by Willy.

This paper is organized as follows. Section 2 discusses the MasPar general architecture, while the two main I/O subsystem configuration alternatives are discussed in section 3. Sections 4 and 5 present the experiments as well as the experimental results, for the MasPar MP-1 and MP-2 respectively. Conclusions and general remarks are given in Section 6.

2. The MasPar Architecture

MasPar computer corporation currently produces two families of massively parallel-processor computers, namely the MP-1 and the MP-2. Both systems are essentially similar, except that the second generation (MP-2) uses 32-bit RISC processors instead of the 4-bit processors used in MP-1. The MasPar MP-1 (MP-2) is a SIMD machine with an array of up to 16K processing elements (PEs), operating under the control of a central array control unit (ACU), see figure 1. The processors are interconnected via the X-net into a 2D mesh with diagonal and toroidal connections. In addition, a multistage interconnection network called the global router (GR) uses circuit switching for fast point-to-point and permutation transactions between distant processors. A data broadcasting facility is also provided between the ACU and the PE. Every 4x4 neighboring PE's form a cluster which shares a serial connection into the global router. Using these shared wires, array I/O is performed via the global router, which is directly connected to the I/O RAM as shown in figure 1. The number of these wires, thus, grows as the number of PEs providing potential for scalable I/O bandwidth. Data is striped across the MasPar disk array (MPDA), which uses a RAID-3 configuration, typically with two optional parity disks and one hot standby. For more information on the MasPar, the reader can consult the MasPar references cited at the end of this study [Blank90][MasPar92][Nichols90].

3. MPIOC^{TM} Versus the PVME I/O Configurations

Based on the I/O structure, the MasPar computers can be divided into two categories: the MasPar I/O channel (MPIOC^{TM}) configuration and the parallel VME (PVME) configuration. It should be noted, however, that the two configurations are not mutually exclusive and they coexist in the MPIOC^{TM} configurations. The PVME, however, is the standard MasPar I/O subsystem and has no MPIOC^{TM}. It should be noted that MasPar I/O subsystem architecture is the same for both the MP-1 and the MP-2 series.

3.1 The PVME I/O Subsystem Configuration

PVME, or parallel VME, is actually the MasPar name for the I/O controller, a VME bus, and a limited amount (8 MB) of I/O RAM. The PVME I/O configuration is the MasPar standard I/O subsystem and it includes all I/O components shown in solid lines in figure 1. The PVME configuration, therefore, is the most common I/O subsystem on MasPar computers.

3.2 The MPIOC^{TM} Configuration

This is the more expensive configuration and, thus, the one which has the potential to offer the higher I/O bandwidth due to the high speed MasPar channel, MPIOC^{TM}, shown in dotted lines in figure 1. Disk controllers interface directly to the channel. A small fraction of MasPar installations have MPIOC^{TM}-based I/O subsystems. Examples are the the current MasPar facility at NASA GSFC and the installations at Lockheed and Lawrence Berkeley Labs. The I/O controller (IOCTLR) provides an 8 Mbytes of I/O RAM connected directly to 64 wires from the global router, as was the case in the PVME. Optional I/O RAMs can be added to the MPIOC^{TM} using either 32 Mbyte or 128 Mbyte modules. All memory is Error Code Correction (ECC)-protected. Each additional I/O RAM, up to a total of four, connects to a different set of 256 wires of the global router, in full PE configurations of 16K PEs. More I/O RAM modules, however, can be added in parallel for a total of 1 Gbyte of I/O RAM at most[MasPar92]. The MPIOC^{TM} is functionally similar to a HIPPI channel with 64 bit data bus and can transfer data up to a maximum rate of 200 MB/Sec.

While the number of links from the PE array to the I/O subsystem scales with the size of the array providing up to 1024 wires for the 16K processor system, the I/O RAM size can grow up to 1 Gbytes. However, since each of the 128 (32) Mbyte modules is linked by 256 wires to the I/O-Global Router link

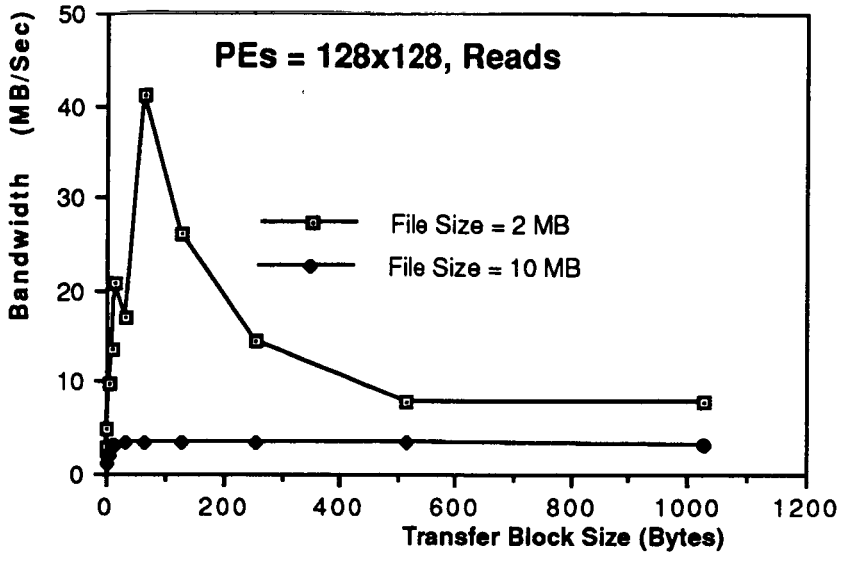


Figure 2. Effects of the transfer block size using the full MP-1 array

(1024 wires), we believe that an MPIOC™ with .5 Gbyte made out of 128 Mbyte modules has the potential for the best scalability characteristics per dollar. However, more I/O RAM still means more files (or bigger files) in the cache and thus better I/O performance.

3.3. Scope and Methodology of this Study

This work was conducted using two case study configurations at NASA GSFC. The first is a PVME configured MP-1 which was used to generate the results in section 4. Last Spring, this installation was upgraded to an MPIOC configured MP-2, which was used to generate the results in section 5. These will be referred to as the MP-1 and the MP-2 in the rest of this paper for simplicity. The MP-1 had 16K PEs and a disk array model DA3016 with 16 disks arranged into two banks, two parity disks, and one hot standby providing a total capacity of 11 GB. The I/O RAM was limited to the 8MB supplied by the IOCTLR. The MasPar published peak bandwidth for the PVME is 16 MB/Sec. In the PVME configurations, only 64 wires are used to connect the PE array to I/O system through the global router, regardless of the number of PEs in the array.

The MP-2 upgrade has also 16K PEs and equipped with an I/O channel (MPIOC) and a 32 Mbyte I/O RAM module. This implies that 256 wires of the global router are connecting the PE array to the MPIOC. Two RAID-3 disk arrays (type DK516-15) are also included, each of which has 8 disks and delivers a MasPar published sustained performance of 15 Mbytes/sec.

Wall clock time was used to time all measured activities. Unless otherwise is stated, measurements were filtered to remove any unusual observations. Filtering was accomplished by taking the average of an interval of m observations. This process was repeated n times and the median of these averages was used to represent the observation. This has also given the cache the opportunity to warm up and augment its performance into the measurements, for files of sizes that fit into the cache. Files of sizes greater than that were not able to take advantage of the IORAM. In each experiment, files that much smaller than the IORAM size as well as files that are much larger than the IORAM were used to represent the entire span of performance observations that one can get out of scientific sequential workloads. Measurements were collected using parallel read and write system calls. Therefore, this study reflects the performance as seen through the MasPar file system (MPFS).

In the context of this work, and unless otherwise is stated, the terms I/O RAM, cache, disk cache, and I/O cache indicate the solid state memory interfacing the I/O subsystem to the MasPar PE array through the global router.

4. Experimental Measurements from the MP-1 Case Study

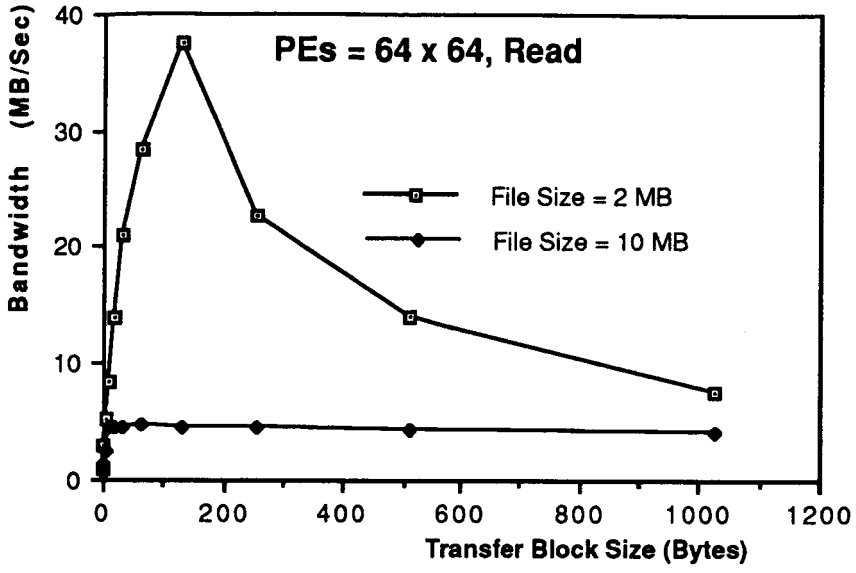


Figure 3. Effects of transfer block size using 64x64 MP-1 PEs

The designed experiments were intended to study the dynamic properties of the MasPar I/O system. In all experiments, the used metric was the bandwidth reported in MB/Sec. Workload was generated by changing a number of parallel I/O request attributes. Parameters that were varied in these experiments include: number of PEs, number of bytes to be transferred by each PE, the file size, the type of I/O operation (read, write, or read/write), and the number of active files. Generated accesses were sequential which is the case in the majority of scientific applications [Miller91].

4.1. Effects of Transfer Block Size

The experiment of figure 2 is intended to study the effect of the transfer sizes on the performance, with the I/O RAM warmed up. All 16k PEs are used. Performance using the 2 MByte file is far better than that of the 10 Mbyte, which suggests that while the 2 Mbyte file seem to fit into the I/O RAM, the 10 Mbyte far exceeds the size to the I/O RAM. The 2 Mbyte file shows an I/O bandwidth peak of about 42 Mbytes/Sec, when the transfer sizes were kept at 64 byte per PE.
Figure 3, however, reports the results of a similar experiment, except that only a grid of 64X64 (4K) PEs is used. Graph is similar to previous one except that the peak is smaller and it occurs at a transfer size of 128 byte per PE, or a total transfer size of 512 Kbyte. Since the total transfer size in the previous case was 1 Mbyte, the peak was expected here at 256 byte per PE. This suggests that either: (1) the link scales down with the lower number of processors; or (2) the processors or their memories are not fast enough. The first possibility is excluded for PVME since the link between the PE and the I/O subsystem is fixed at 64 wires (MBytes per second). Thus, it seems that the limiting factor here was the processors' local memories speed. Comparing the optimal transfer size and the peak performance in this case and the previous one, leads to believe that the MasPar possess favorable scalability characteristics that should be further studied.

4.2. Dynamic Behaviors of I/O Caches

Dynamic cache size can be affected by the specifics of the implementation. The experiments in this section are designed to focus on the dynamic cache attributes such as the effective read and write cache sizes, as well as prefetching and write-behinds.

4.2.1 Effective Cache Sizes

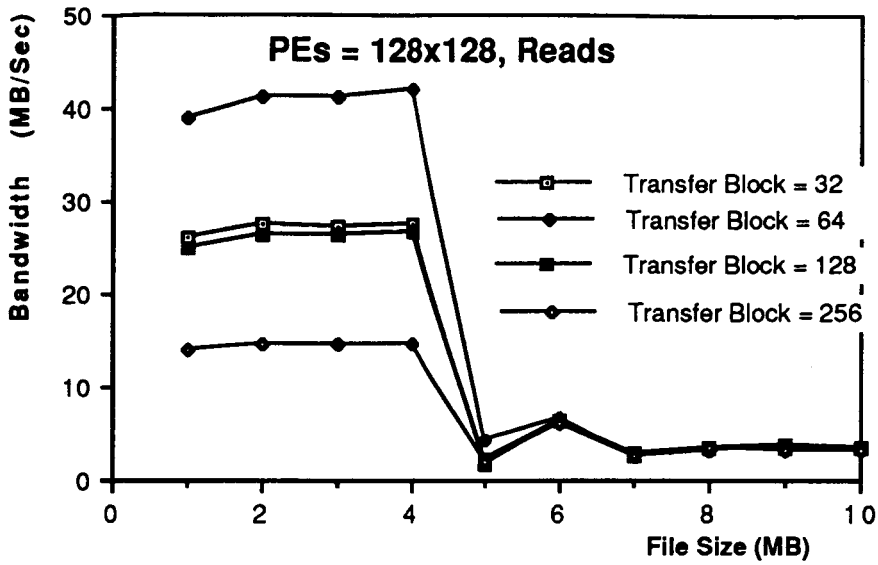


Figure 4. Effective I/O Read Cache on the PVME Configured MP-1

Figure 4 presents the results of the first experiment in this group, which investigates the effective I/O cache read size. The transfer size of 64 bytes per PE offers the best match for the 16K PE. Regardless of the used transfer sizes, performance degrades rapidly when file sizes exceed the 4 Mbyte boundary which indicates that the effective read cache size is 4 Mbyte.

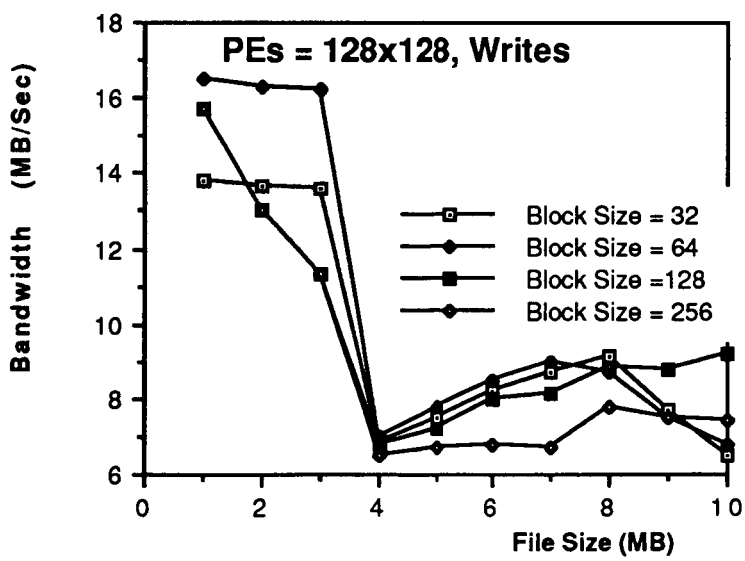


Figure 5. Effective I/O Write Cache on the PVME Configured MP-1

The write counterpart on the previous experiment is reported in figure 5. Results are very similar to those of the read case with a few exceptions. When the transfer size is big enough to result in an overall transfer size that exceeds the file size, some processors will have to remain idle and the overall performance will degrade. This resulted in an early performance degradation in the case of 128 and 256 byte transfers. This situation does not arise in the reads, where the transferred bytes are distributed over all enabled processors. Furthermore, performance degrades when file sizes exceed the 3 Mbyte limit. This indicates that the cache write size is 3 Mbytes. The 4 Mbyte effective read cache size and the 3 Mbyte effective write size were due to the system set up of the 8 Mbyte I/O RAM. The I/O RAM can be partitioned into buffers that are to

enhance disk reads, disk writes, processor array to front end communications, and processor to processor communications. At the time of testing, the I/O RAM was configured to set up 4 Mbyte for disk reads, 3

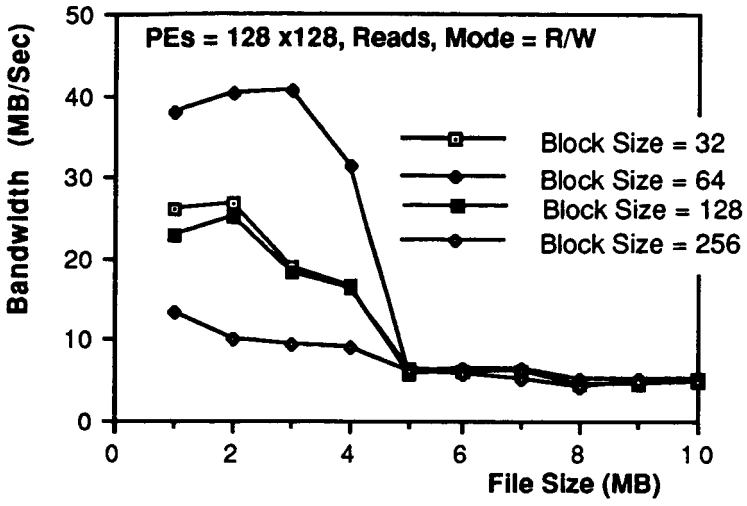


Figure 6. Effect of Read/Write Mode on Reads

Mbyte for disk writes, and 1 Mbyte for communications with the front end [Busse93]. Finally, the bandwidth for larger files that can not fit into the cache is slightly higher than the bandwidth of similar file sizes in the case of read. This is mainly due to the write-behind which allows the executing process to write to the I/O RAM. Writing to the disk proceeds later asynchronous of the processing.

Since different cache blocks seem to be used for readable and writeable files, it became of interests to see the effect of opening a file with a read/write mode on read operations. The experiment of figure 6 examines this behavior. Performance of reads in this case falls between the read and the write performance.

4.2.2 File Prefetching

Individual measurements were collected with and without flushing the I/O RAM in between. It was found that the only form prefetching is to leave files in the cache for future references once they are read, provided the file in question fits into the allocated part of the I/O RAM. This is done even after the file is closed by the application. There was no indication from our measurements that a part of a file is cached if the entire file size is too big to fit into the cache.

4.3. I/O Scalability

In our context, scalability refers to the ability of the I/O bandwidth to increase as the number of processors participating in I/O activities grow. The experiment of figure 7, was designed to study the scalability characteristics of the I/O subsystem. The size of the system, no. of PEs, under study is changed here by changing the active set enabling only a subset of processors. The X-axis represents the dimensionally "n" of the enabled "nxn" submesh. The system exhibits good I/O scalability as long as files fit into the cache. Spikes of unusually high performance were noticed at 32x32, 64x64, 96x96, and 128x128 processor subsystems. These subsystems are all multiples of 32x32 (1 K processors) which suggests that optimal performance is reached when the used processors are multiples of 1k. The reason for this is the fixed I/O-Router link of 64 wires in the PVME configuration. Recall that each one of these wires can connect, via the global router, to a cluster of 16 processors. Thus, the 64 wires can connect to a 1K partition of the processor array. Therefore, when a multiple of 1K processors are performing I/O, all wires of the link are used up all the time providing the potential for maximum bandwidth. The 96x96 processor case, although meets the criteria for high performance, does not show as much improvement in performance as the other three points. This behavior remains hard to explain. This, however, is the only case where this multiple of

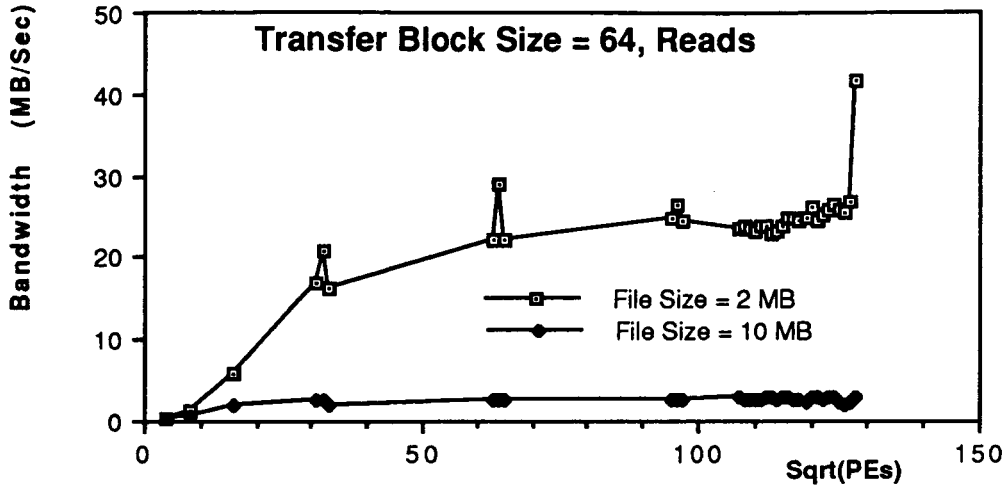


Figure 7. Bandwidth Scalability in the MP-1/PVME with the Increase in PEs I/O

1K is not a power of two, which might have resulted in some mismatching with other internal design or packaging properties. Outside the cache, performance is degraded and the bandwidth does not seem to scale with the increase in the number of processors involved in the I/O.

5. MP-2 Case Study Measurements

A representative set of measurements were obtained once the MP-1 was upgraded to an MP-2 with MPIOC and I/O RAM. The measurements were designed to highlight the important aspects of the upgrade. Discovering the effective cache sizes and assessing the scalabiltiy were clear targets to see how the configuration relates to its predecessor and how the performance of the MPIOC relates to that of the standard PVME. The effective read cache size measurements are shown in figure 8. The performance of reads drops significantly when file sizes exceed 12 Mbytes. Thus, the effective cache size is 12 Mbytes for reads. Transfer blocks of sizes 64 remain to do well but their good performance could also be obtained by using blocks of 128 bytes instead. The 12 M byte was also found to be the entire allocation for the MasPar file system (MPFS) out of the used 32 Mbyte I/O RAM. When file sizes exceed the 12 Mbyte limit, the sustained disk array speed is about 10 Mbyte/ Sec which is 33% less than the published rate. However, the published rate of 15 Mbyte/Sec is achievable with very large files as will be shown later.

Effective write cache size, as seen in figure 9, remains at 3 Mbyte even with the increased caching space due to the I/O RAM module. Furthermore, the write performance is about one order of magnitude worse than that of the read.

Prefetching is not different from the first case study and is still following the same simple strategy of leaving a previously read file in the cache. Prefetching on this system was again studied by collecting individual (non averaged) measurements with and without cache flushing in between.

Scalabiltiy measurements were collected for two files of sizes 10 Mbyte and 100 Mbyte under parallel read operations and plotted in figure 10. This figure resembles figure 7 in the general form but now with much greater values. For the 100 Mbyte case, the system runs at the speed of the disk array which now demonstrates a sustained performance equal the published 15 Mbyte/Sec. The 10 Mbyte file displays a great positive spike at 128x128 PEs. This is consistent with figure 7 and the fact that in this new configuration our I/O RAM module provide 256 wires that support 4K PEs through the global router.

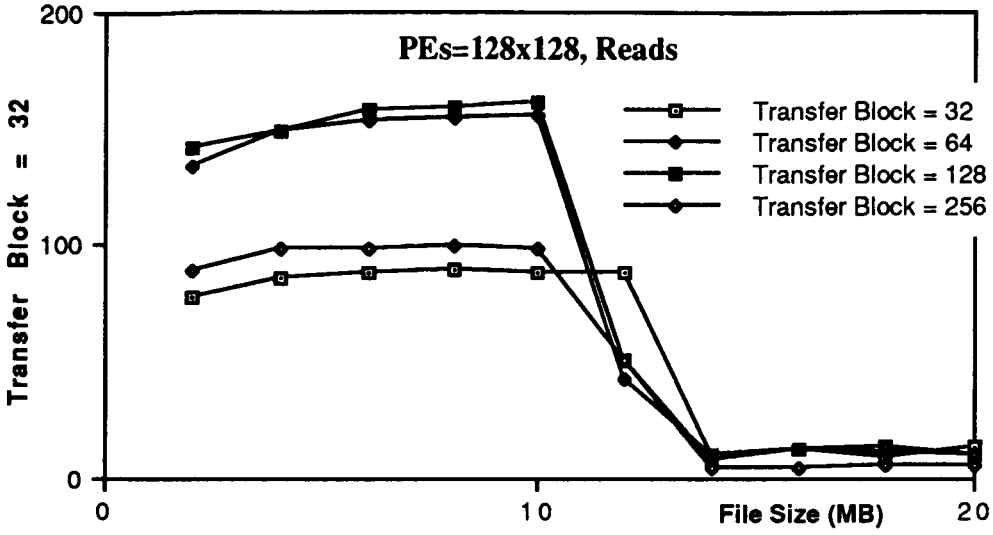


Figure 8. Effective Read Cache for the MP-2/MPIOC Case Study

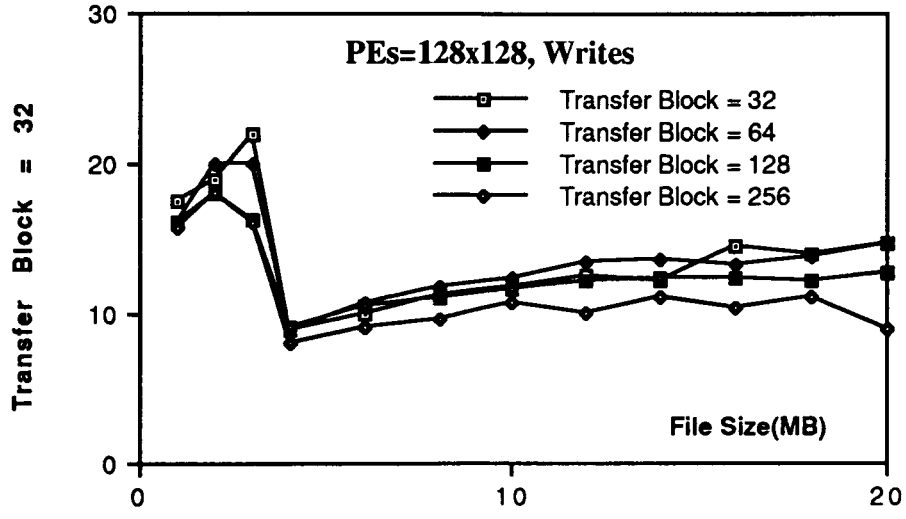


Figure 9. Effective Write Cache for the MP-2/MPIOC Case Study

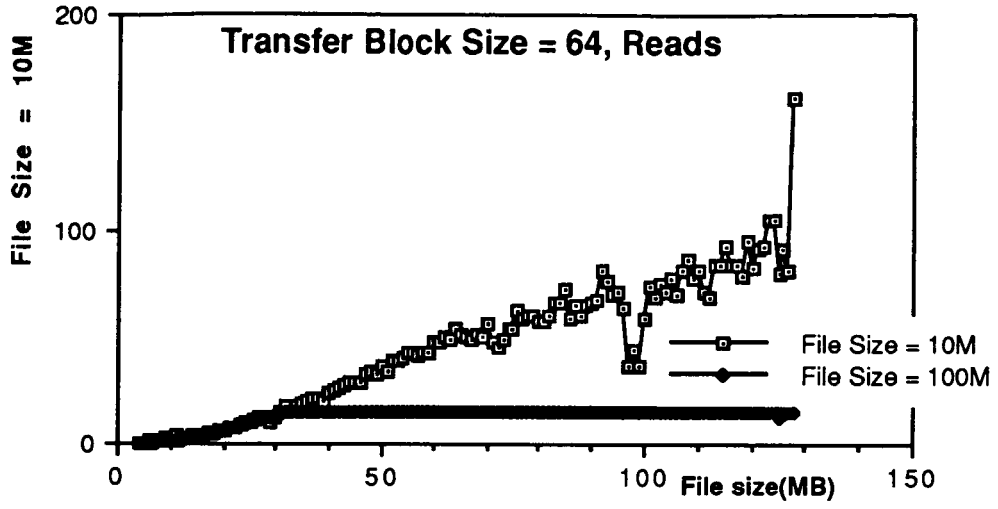


Figure 10. I/O Scalability for the MP-2/MPIOC configuration

Thus, positive spikes are expected at dimensionalities that provide multiples of 4K, namely 64x64 and 128x128. No such spike was observed however at 64x64. The negative spikes are basically due to the systems activities at the time of the measurements.

6. Conclusions

The MasPar has been known for its cost efficiency, ease of use, and computational performance. This work has shown that programmers of I/O-intensive scientific applications can tune their programs to attain good I/O performance when using the MasPar. They should be at least aware of their I/O configuration, the specific I/O RAM size and how it is locally partitioned in an attempt to partition data into files that can fit into the I/O RAM. The work further establishes that system managers are also encouraged to understand the I/O resource requirements of the applications running on their machines and tune the I/O RAM configuration for best performance. In specific, partitioning the I/O RAM among disk reads, disk writes, data processing unit (DPU) to front end communications, and interprocessor communications should be based on an understanding of the most common needs of the local application domain. Finally, the work has demonstrated that a full MasPar configuration with MPIOC^{TM} and a full I/O RAM has potential for delivering scalable high I/O performance. However for this to happen the I/O RAM management should make good attempt to prefetch anticipated data. Further, the I/O RAM partitioning strategy should be more flexible by using cache blocks for different purposes as dynamically needed by the applications. At the least, files smaller than the I/O RAM size should be cacheable. Finally, the sustained performance of the disk arrays remains to be the clear bottleneck and is likely to limit the overall performance of parallel I/O systems for some time to come.

References

[Anon85] Anon et al., "A Measure of Transaction Processing Power", Datamation, April 1985, 112-118.
[Blank90] Tom Blank, "The MasPar MP-1 Architecture", Proc. of the IEEE Compcon, Feb. 1990.
[Busse93] T. Busse, Personal Communications, MasPar Co., August 1993.
[Chen92] P. Chen, Input/Output Performance Evaluation: Self-Scaling Benchmarks, Predicted Performance, Technical Report, University of California at Berkeley, Rep. No. UCB/CSD-92-714, November 1992.
[Jain91] R. Jain, The Art of Comp. Sys. Performance Analysis, John Wiley, NY:1991.
[Gaede81] S. Gaede, "Tools for Research in Computer Workload Characterization", Experimental Comp. Performance and Evaluation, 1981. D. Ferrari and M. Spadoni, eds.

[Gaede82] S. Gaede, "A Scaling Technique for Comparing Interactive System Capacities", 13th Int. Conf. on Management and Performance Evaluation of Computer Systems, 1982, 62-67.
[Grand92] Grand Challenges 1993: High Performance Computing and Communications, A Report by the Committee on Physical, Mathematical, and Engineering Sciences, Federal Coordinating Council for Science, Engineering, and Technology, 1992.
[Howard88] J. H. Howard et al., "Scale and Performance in a Distributed File System", ACM Trans. on Computer Systems, February 1988, 51-81.
[Katz89] R. H. Katz, G. A. Gibson, and D. A. Patterson, "Disk System Architecture for High Performance Computing", Proc. of the IEEE, Dec.1989, 1842-1858.
[Krystynak93] J. Krystynak and B. Nitzberg, " Performance Characteristics of the iPSC/860 and CM-2 I/O Systems", Proc. of the 7th IPPS, April 1993.
[MasPar92] MasPar Technical Summary. MasPar Corporation, November 1992.
[McDonell87] K. J. McDonell, " Taking Performance Evaluation out of the Stone Age", Proceedings of the Summer Usenix Technical Conference, Phoenix, June 1987, 407-417.
[Miller91] E. L. Miller and R. H. Katz, "Input/Output Behavior of Supercomputing Applications", Proc. of Supercomputing'91, 567-576.
[Nelson92] B. Nelson, B. Lyon, M. Wittle, and B. Keith, "LADDIS--A Multi-Vendor and Vendor-Neutral NFS Benchmark",Uniforum Conference, January 1992.
[Nickolls90] The Design of the MasPar MP-1: A Cost Efficient Massively Parallel Computer, Proc. of the IEEE Compcon, Feb. 1990.
[Park90] A. Park and J. C. Becker, "IOStone: A Synthetic File System Benchmark", Computer Architecture News, June 1990, 45-52.
[Reddy90] A. L. N. Reddy and P. Banerjee, "A Study of I/O Behavior of the Perfect Benchmarks on a Multiprocessor", Proc. of the 17th Int. Symp. on Computer Architecture, May 1990, 312-321.
[Saavedra-Barrera89] R. H. Saavedra-Barrera, A. J. Smith, and E. Miya, "Machine Characterization Based on an Abstract High-Level Language Machine," IEEE Trans. Comp., vol. 38, No. 12, Dec. 1989.
[SPEC91] SPEC SDM Release 1.0 Manual, System Perf. Evaluation Cooperative, 1991.
[Shein89] B. Shein, M. Callahan and P. Woodbuy, "NFSStone-- A Network File Server Performance Benchmark", Proc. of Usenix Summer Tech. Conf., June 1989, 269-275.
[TPCB90] TPC Benchmark B Standard Specification, Transaction Processing Council, August 1990.

APPENDIX B:

PHYSICAL I/O REQUESTS OF ESS APPLICATIONS

In Collaboration with Mike Berry (GWU)

An Experimental Study of Input/Output Characteristics of NASA Earth and Space Sciences Applications

Michael R. Berry Tarek A. El-Ghazawi

Department of Electrical Engineering and Computer Science
The George Washington University
Washington, D.C. 20052
{mrberry,tarek}@seas.gwu.edu

Abstract

Parallel input/output (I/O) workload characterization studies are necessary to better understand the factors that dominate performance. When translated into system design principles this knowledge can lead to higher performance/cost systems. In this paper we present the experimental results of an I/O workload characterization study of NASA Earth and Space Sciences (ESS) applications. Measurements were collected using device driver instrumentation. Baseline measurements, with no workload, and measurements during regular application runs, were collected and then analyzed and correlated. It will be shown how the observed disk I/O can be identified as block transfers, page requests, and cache activity, and how the ESS applications are characterized by a high degree of spatial and temporal locality.

1. Introduction

In recent literature, the I/O performance bottleneck has been extensively addressed. It is clear that the current trends in technology will continue to increase the performance gap between processing and I/O. However, the improvement of parallel I/O architectures and file systems can help in reducing this gap. Improving these architectures, with cost-efficient solutions, requires an in-depth understanding of the I/O characteristics and resource needs of the underlying applications. Capitalizing on the most common and dominant machine behaviors thus allows significant performance benefits to be achieved at relatively low cost. In this paper we discuss empirical results from our workload characterization study conducted in the NASA ESS application domain, revealing the important I/O workload characteristics and the underlying factors.

I/O workload characterization requires a methodology and a tool for measuring I/O activities. Instrumentation can be accomplished at one or more system levels, including application code, I/O libraries, file systems, device drivers, and hardware monitoring of I/O channels and system bus. Instrumentation of each level can reveal disparate data.

The workload presented to the I/O subsystem is a combination of requests generated by both the application and operating system. Therefore, we chose to use device driver instrumentation of the hard disk sub-system, to capture both applications and system I/O activities. In addition, we used a set of experiments designed to aid in distinguishing the I/O behaviors due to the operating system from those that are directly generated by the applications. Device driver instrumentation does require access to the operating system source code, which is generally hard to acquire from most vendors. Therefore, the experimental network of workstations (NOW) system, Beowulf [1], at NASA Goddard was selected as the platform for this study largely due to the availability of its operating system source code. Three typical ESS applications also from NASA provided the workload.

This paper is organized as follows. Section 2 reviews some of the related work focusing on recent I/O workload characterization studies conducted in the context of parallel systems. Section 3 describes the methodology used including the instrumentation technique, experiments, measurements and information sought. In section 4, the experimental results are presented and discussed. A summary and conclusions are given in section 5.

This research is supported by the CESDIS University Program in High-Performance Computing under USRA subcontract 5555 #18.

1063-7133/96 $5.00 © 1996 IEEE
Proceedings of IPPS '96

2. Related work

There have been several previous studies, both experimental and theoretical, that have examined the issue of I/O workload characterization. In this section we describe the related research, highlighting the objectives, methods, and results of those studies.

The I/O behavior of parallelized benchmarks on an Alliant multiprocessor emulator was examined in [2]. They found the applications exhibited sequential I/O characteristics. I/O access rates and patterns were determined for a Cray YMP in [3] using C library instrumentation. This work categorized three general classes of I/O access patterns: required (any I/O at program start-up and termination), checkpoint (I/O to save minimum data for program restart), and data staging (I/O needed when memory requirements are more than physical memory, e.g., paging). Pasquale and Polyzos in [4] studied the static and dynamic I/O characteristics of scientific applications in a production environment on a Cray YMP, and concluded the intensive I/O applications had a regular access pattern. The architectural requirements of eight parallel scientific applications were evaluated on nCube and Touchstone Delta machines in [5]. This study described the temporal patterns in I/O accesses and rates. A parallel I/O modeling and interface methodology is discussed in [6], along with the parallel I/O requirements observed at the Argonne National Lab. In [7] system architecture issues concerning parallel I/O on massively parallel processors (MPPs) are discussed. The need for comprehensive workload characterization through instrumentation studies of multiple platforms and applications is emphasized.

In [8] I/O workload characteristics were presented for a parallel file system on an iPSC/860 running parallel scientific applications in a multiprogramming production environment. File usage and size, read and write request sizes, request spacing in a file, access patterns, locality, and design implications for parallel file systems are presented. In a related study, [9] characterized control-parallel and data-parallel user-program I/O on a CM-5. These studies of the CHARISMA project comprise a solid body of work in characterizing a file system's I/O workload requirements.

In [10] the parallel I/O workloads of four applications running on a parallel processor with the Vesta file system are characterized. This study used six R/S 6000's connected with an SPn network (same network that is used in IBM's SP1 and SP2 machines) and the Unified Tracing Environment (UTE) to perform the I/O characterization which showed I/O request sizes and rates, and data sharing characteristics. This study supported the I/O characterization results concerning request sizes and rates reported in [8,9]. In [11] the instrumented versions of three scientific applications with high I/O requirements were run on an Intel Paragon XP/S. This study characterized the parallel I/O requirements and access patterns. In [12] VanderLeest used instrumented I/O library calls, kernel initiated tracing, and a bus analyzer to study I/O resource contention.

The work in [8,11,12] are the most closely related efforts to ours. Our work differs from that of these studies in that we are using device driver instrumentation instead of I/O library instrumentation. The hybrid instrumentation implemented in [12] is in the form of a bus analyzer at the lowest level, and library instrumentation at the high end. That work was not conducted on parallel systems, nor did it examine scientific applications with parallel I/O.

3. Workload characterization methodology

In this section we discuss the characterization method and the rationale behind the selections that we made. These elements include, the objectives of the study, the hardware platform selected, the applications used to provide the workload excitations, the method of monitoring the I/O and collecting the measurements, the specific experiments that were performed, the data collected, and the information generated.

3.1 Objectives

Most I/O workload characterization efforts have focused on measuring explicit I/O requests to data files, ignoring system activities. Therefore, in this work we pay particular attention to the total workload which is ultimately presented to the I/O subsystem. Such a workload consists of explicit application I/O, pure systems activities, and system activities generated in response to the needs of the applications. We especially recognize the benefit of being able to characterize this total I/O workload generated, as well as the elementary factors that give rise to this overall behavior. Accordingly we have captured trace file data on all of the system's I/O activity at the disk level. From the analysis of this data we characterize the system's I/O behavior to aid in its understanding and in the development of more efficient systems.

3.2 Platform

In order to measure the I/O activities at the physical level, we implemented disk drive instrumentation. Instrumenting disk device drivers required access to the

operating system source code from any target parallel platform that we considered. Since most such code is proprietary we found it very difficult to obtain. Consequently, we decided to use the experimental parallel testbed, Beowulf [1], built at NASA Goddard, which uses the Linux operating system. The prototype Beowulf system, which we used, is a parallel workstation cluster with 16 Intel DX-4 100 MHz subsystems, each with 16 MB of RAM, a 500 MB disk drive, and 16 KB of primary cache, connected with two parallel Ethernet networks. In addition to the Linux operating system, the Beowulf system has PVM for inter-processor communication, and can use PIOUS [13] as a parallel file system for coordinated I/O activities. Since Linux's GNU licensing policy allows public access to the source code, we were afforded the opportunity to develop and use device driver instrumentation. This consideration was a prime motivator in the selection of this parallel system.

3.3 Applications

Three representative parallel applications were selected from the NASA ESS domain. These are a piece-wise parabolic method (PPM) code, a wavelet decomposition code, and an N-body code. The PPM code is an astrophysics application that solves Euler's equations for compressible gas dynamics on a structured, logically rectangular grid [14]. Our study used four 240x480 grids per processor. This code has been used primarily for computational astrophysics simulations, such as supernova explosions, non-spherical accretion flows, and nova outbursts.

Wavelet transformation codes are used extensively at NASA Goddard for ESS satellite imagery applications such as image registration and compression, of such images as from the Landsat-Thematic Mapper [15]. The version of the code we used decomposed a 512x512 byte image. N-body simulations have been used to study a wide variety of dynamic astrophysical systems, ranging from small clusters of stars to galaxies and the formation of large-scale structures in the universe. Our N-body code uses an oct-tree algorithm with 8K particles per processor, which resulted in 303 million total particle interactions [16].

3.4 Instrumentation

The parallel I/O performance data described and depicted in the following section was collected using an instrumented disk device driver running on each of the workstation nodes. Each workstation's IDE disk device driver was modified to capture trace data on all I/O activity requested of the hard disk sub-system. The read and write handlers in the IDE disk device driver were instrumented to capture the requested level of instrumentation. All read or write requests sent to the disk drive generated a trace entry consisting of a time-stamp, the disk sector number requested, a flag indicating either a read or write request, and a count of the remaining I/O requests to be processed.

The I/O instrumentation traces were buffered by the kernel message handling facility through the proc filesystem [17], and were eventually written to disk. Using the proc filesystem allowed the trace data to be transported from kernel space into user memory in /proc, without the need to develop and integrate additional specialized kernel code. Buffering the traces through the proc filesystem allowed the captured data to be stored quickly in memory, with the flexibility and ease of retrieving the data from what appeared to be a regular file in the proc filesystem. The level of instrumentation was controlled through the use of an ioctrl call. This allowed the instrumentation to be turned off and on, without the need to reboot the cluster with the desired instrumented or non- instrumented kernel.

3.5 Experiments

The instrumentation was turned on and trace file data was collected for I/O requests during four basic experiments. The first consisted of gathering data while no user applications were running. This allowed us to measure the quiescent I/O level, with which we could compare to the I/O activity measured while applications were running and a user induced I/O load was present. The next three experiments involved running each of the three applications described above, one at a time. These experiments were intended to reveal the individual contribution of each application to the overall behavior. The final experiment was to collect data while all three applications were running simultaneously. This experiment created an I/O load resulting from a combination of different applications, to emulate a typical production environment.

3.6 Metrics

A number of metrics were used in characterizing the I/O in this study, including I/O request size, the distribution of requests by disk sectors, and the average time between consecutive accesses to the same sector. Spatial locality information was developed from the distribution of requests by sector number, and temporal locality data was produced from the measurements of time elapsed between accesses to a particular sector.

4. Workload characterization

4.1 Base line

The first part of our study focused on the analysis of I/O activity while no user applications were running. Figure 1 covers this period of inactivity and shows I/O accesses concentrated around a few sectors, which is consistent with logging and table lookup activities that are normally part of routine kernel work occurring all of the time. These I/O requests can be seen as horizontal lines. The predominate I/O request size observed during this period is 1KB [18]. A few instances of small multiples of 1KB requests were also seen. This 1KB request size matches the disk systems block size of 1KB, and is indicative of small I/O requests generating I/O transfers of the smallest possible physical request size.

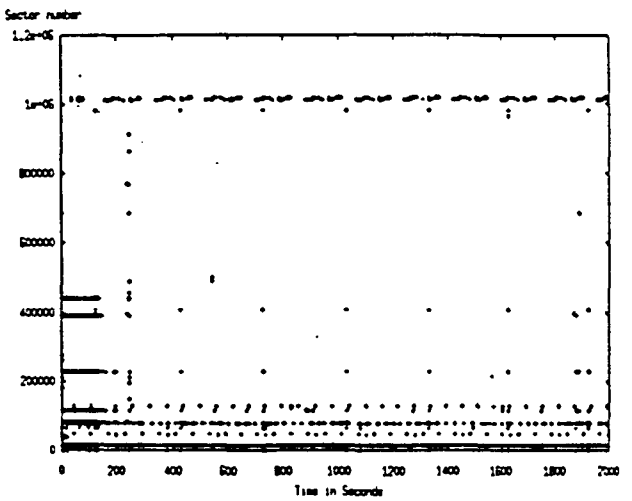


Figure 1. I/O Requests (baseline)

4.2 Single applications

Piece-wise parabolic method: The I/O during this application is relatively low with no paging activity occurring while this program is running, except briefly toward the end. As can seen in Figure 2, the paging activity is denoted by a 4KB request at approximately 230 seconds from the beginning of the execution time. The 1KB block I/O requests are very prevalent, consistent with kernel activity, low user program I/O demand, and small infrequent requests.

Wavelet: Figure 3 presents the I/O activity that was observed while the wavelet decomposition application was running. In Figure 3, a frequent request size of 4KB can be observed, which indicates a high rate of paging. The paging requirements of the wavelet program are due

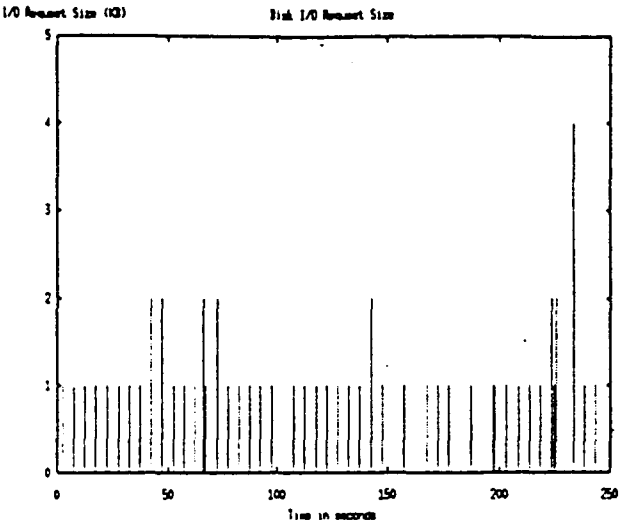


Figure 2. Request Size (PPM)

to the large program space and image data requirements. A spike of I/O activity occurs at approximately 50 seconds into the execution. This is generated by the higher request sizes occurring while the data file is being read. Requests approaching 16 KB are observed during

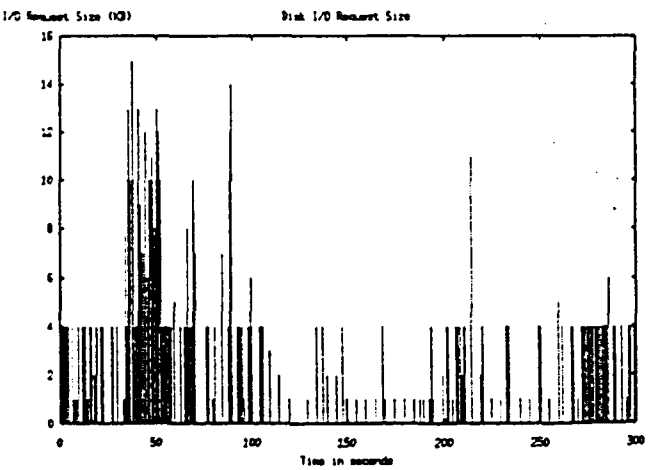


Figure 3. Request Size (wavelet)

this period, and are a result of the 16 KB cache on Beowulf. As a stream of data is being read at this point of execution, cache is repeatedly filled with the new data. Interference from system activities keep the request size from reaching and maintaining the full 16 KB cache size. A lull in the I/O activity is the next significant feature of this application, indicating that the computational phase is underway. Note that there are few page requests (at 4KB) during this period. This is caused by system memory maintaining the working set of instructions and data, without the previous higher need for new data and

instructions.

N-body: In Figure 4 the consistent 1 KB block I/O is visible, with more 2 KB requests and a few page swaps (or 4KB requests) than occurred during PPM. The higher computational requirements of the N-body problem cause more frequent page faults than PPM, to maintain the working set, but the overall activity is obviously much less than that of the wavelet program with its large data requests.

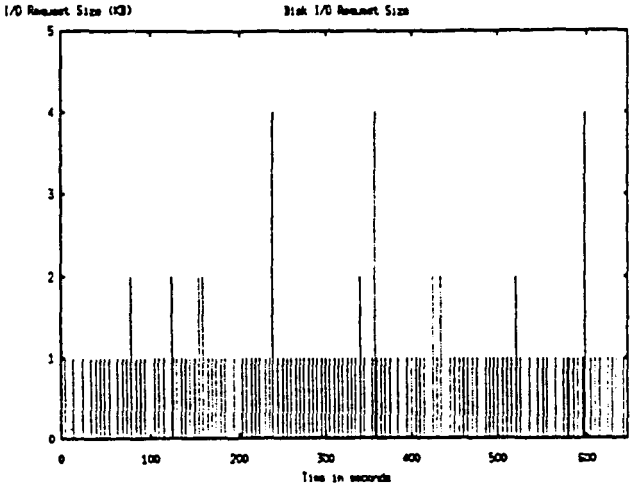


Figure 4. Request Size (N-Body)

4.3 Combined applications

Figures 5 and 6 show the resultant I/O from running all three applications simultaneously. The resultant I/O request sizes shown in Figure 5 reflect the simultaneous demand on the I/O by all three applications. The 1 KB requests are maintained throughout this period, with a much higher occurrence of 4 KB requests, reflecting the greater load. The dramatic increase in request size at approximately 50 seconds, is primarily due to the image being read in the wavelet application, but the combined effect of the applications have driven the total request sizes much higher than when the applications were run independently. Request sizes in the 16 KB to 32 KB range shown in Figure 5 are attributed to an increased I/O buffer size when the wavelet data file is read.

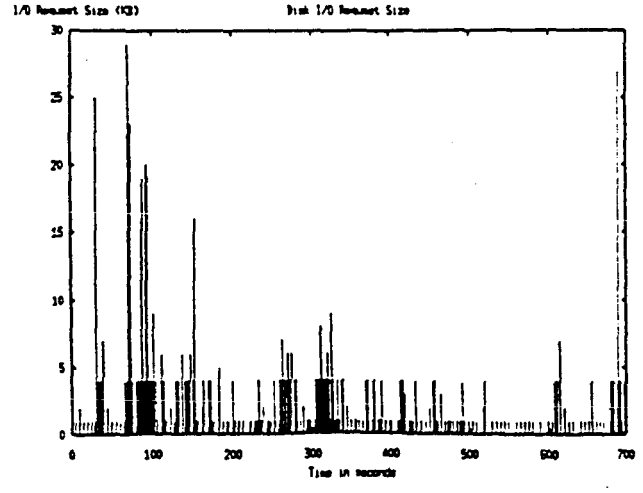


Figure 5. Request Size (combined)

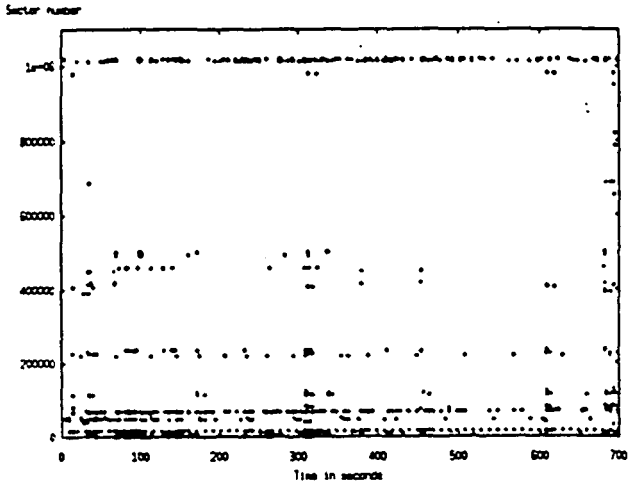


Figure 6. I/O Requests (combined)

Figure 6 also shows a correspondingly higher amount of request activity, primarily in the lower sector numbers. The clumping of requests seen in Figure 6 matches the periods of greater request activity seen in Figure 5. The distribution of I/O requests between reads and writes that occurred during each application (average per disk) and during 2000 seconds of baseline inactivity is shown in Table 1. System and instrumentation logging account for the almost exclusive amount of writes that was measured

Application	reads	writes	requests per sec	total requests
Baseline	0%	100%	0.9	1782
PPM	4%	96%	1.4	358
Wavelet	49%	51%	294.5	88342
N-Body	13%	87%	1.0	638

Table 1. I/O Requests

in all but the wavelet experiment. (Note: I/O instrumentation did not measurably change the execution time of any of the applications.) The difference in the relative percentages between reads and writes for the wavelet application is because this program is the only one that has significant input data, in this case from its imagery data file. The overall low request activity in the PPM and N-body applications, and the low percentage of reads, is a result of both of these programs being simulations with no input data, with and only short statistical summaries being written.

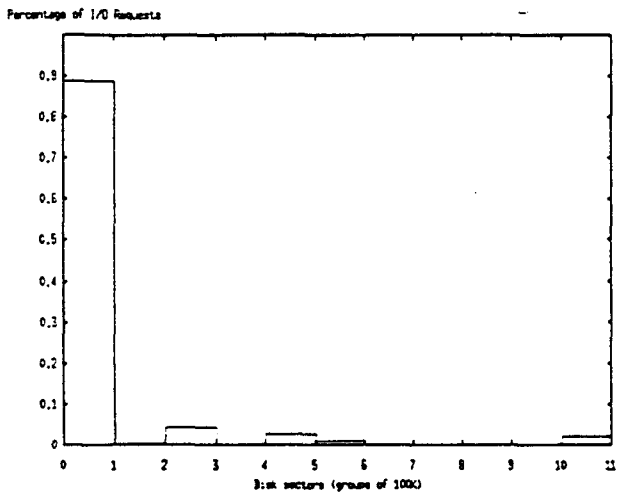


Figure 7. Spatial Locality (combined)

Figure 7 shows the spatial locality as a percentage of I/O requests occurring within a band of sectors. In this figure, sectors have been combined into bands of 100K each. The higher incidence of I/O activity in the lower

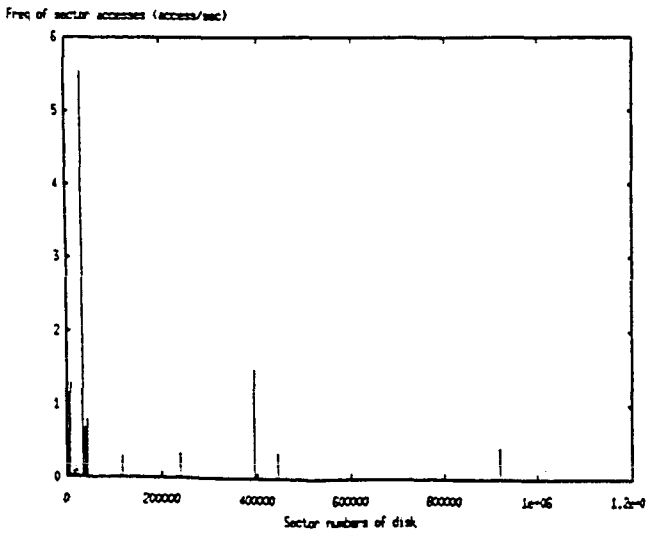


Figure 8. Temporal Locality (combined)

sector numbers is caused by the user programs and data, swap file space, and kernel file data mainly residing in these locations on the disk. Figure 8 shows temporal locality as a characterization derived from data also collected while running the combined application experiment. Temporal locality is expressed as the frequency of accesses (per second) to the same sector on disk. These access frequencies were averaged over the 700 seconds required to run the combined experiment. Figure 8 also shows most of the I/O occurred at the lower sector numbers. The most frequently accessed sector location was approximately 45000, and the next most frequent at just under 400000.

5. Summary and conclusions

This study has aimed at characterizing the parallel I/O workload generated by some of NASA's ESS applications. This was accomplished by instrumenting the disk device driver and capturing trace information on the total load of the emulated production environment as observed by the I/O subsystem. Experiments were conducted to reveal the elementary contributions of the individual applications, system activities, and the combined characteristics of a multiprogramming load with several applications running concurrently.

The proposed instrumentation technique has been able to identify different I/O activities based on the observed request sizes, that fell into three primary categories. First, small requests which were observed as 1KB physical requests. Second, paging activities which were observed as 4 KB requests. Third, large I/O requests distinguished by sizes approaching multiples of 16 KB, indicating most of the 16 KB cache data was being replaced.

With the ability to closely observe I/O activities, the I/O attributes and request patterns were monitored and characterized. It was shown that in the absence of applications, system activities of small request sizes appear at low and high sector numbers due to system logging. Intensive data set manipulation applications such as the wavelet image processing code were distinguished with heavy paging in the beginning of the application to build the working set of the code and large data structures, as well as with large explicit request sizes approaching multiple cache block size, when the image data was read. Limited paging activities still occurred to maintain the working set, followed by a heavier activity toward the end of the application run. Both N-body and PPM are simulation codes, and have shown behaviors that are similar. In general, these two codes have very limited I/O activities, most of which is implicit. The explicit I/O is due to writing the final simulation results into output files. A very small amount of paging activity was

observed as a result of processing in these applications.

In addition to the request size characteristics, the ESS applications' I/O exhibited substantial locality properties. The spatial locality of the combined workload, almost follows the 10-90 rule. Temporal locality analysis revealed some hot spots on the disk. Also, a relatively high ratio of writes, as compared to other domain applications, were observed in the ESS applications, particularly in wavelet. Our next step is to integrate these data into a parameter set that can be used for system design and tuning of parallel systems and applications.

Acknowledgments

The authors would like to thank Terrence Pratt for encouraging and supporting this research at its inception. We also thank Thomas Sterling for his support and valuable input; Don Becker for his invaluable explanations of Linux; Chance Rescheke and Dan Ridge for their assistance with the Beowulf NOW; and Charlie Packer, Udaya Ranawake, Kevin Olson and Jacqueline Lemoigne for their help with the NASA ESS applications.

References

[1] Sterling, T., Becker, D., Savarese, D., Dorband, J., Ranawake, U., Packer, C. Beowulf: A Parallel Workstation for Scientific Computation. In *Proc. of the 1995 International Conference on Parallel Processing*, Vol I, pp. 11-14, 1995.

[2] Reddy, A., Banerjee, P. A Study of I/O Behavior of Perfect Benchmarks on a Multiprocessor. *Computer Architecture News*, 18(2): 312-321, June 1990.

[3] Miller, E., Katz, R. Input/Output Behavior of Supercomputing Applications. In *Proc. of Supercomputing '91*, pp. 567-576, Nov 1991.

[4] Pasquale, B., Polyzos, G. A Static Analysis of I/O Characteristics of Scientific Applications in a Production Workload. In *Proc. of Supercomputing '93*, pp. 388-397, Nov 1993.

[5] Cypher, R., Ho, A., Konstantinidou, S., Messina, P. Architectural Requirements of Parallel Scientific Applications with Explicit Communication. *Computer Architecture News*, 21(2): 2-13, June 1993.

[6] Galbreath, N., Gropp, W., Levine, D. Applications-Driven Parallel I/O. In *Proc. of Supercomputing '93*, pp. 462-471, Nov 1993.

[7] del Rosario, J., Choudhary, A. High-Performance I/O for Massively Parallel Computers: Problems and Prospects. *IEEE Computer*, 27(3): pp. 59-68, Mar 94.

[8] Kotz, D., Nieuwejaar, N. File-System Workload on a Scientific Multiprocessor. *IEEE Parallel & Distributed Technology*, 3(1): 51-60, Spring 1995.

[9] Purakayastha, A., Ellis, C., Kotz, D., Nieuwejaar, N., Best, M. Characterizing Parallel File-access Patterns on a Large-scale Multiprocessor. In *Proc. of the 9th International Parallel Processing Symposium*, pp. 165-172, April 1995.

[10] Baylor, S., Wu, C. Parallel I/O Workload Characteristics using Vesta. In *Proc. of the 3rd Annual Workshop on Input/Output in Parallel and Distrib. Systems*, pp. 16-29, April 1995.

[11] Crandall, P., Aydt, R., Chien, A., Reed, D. Input/Output Characteristics of Scalable Parallel Applications. In *Proc. of Supercomputing '95*, pp. 613-619, December 1995.

[12] Vander Leest, S. Measurement and Evaluation of Multimedia I/O Performance. Ph.D. dissertation. Department of Electrical and Computer Engineering, University of Illinois Urbana-Champaign, 1995.

[13] Moyer, S., Sunderam, V. PIOUS: A Scalable Parallel I/O System for Distributed Computing Environments. In *Proc. of the Scalable High-Performance Computing Conference.*, pp. 71-78, 1994.

[14] Fryxell, B., Taam, R., Numerical Simulations of Non-Axisymmetric Accretion Flow. *Astrophysical Journal*, 335: 862-880, 1988.

[15] El-Ghazawi, T., Le Moigne, J. Multi-Resolution Wavelet Decomposition n the MasPar Massively Parallel System. CESDIS Technical Report, TR-94-122.

[16] Olson, K., Dorband, J. An Implementation of a Tree Code on a SIMD Parallel Computer. *Astrophysical Journal Supplement Series*, 94: 117-125, September 1994.

[17] Johnson, M., The Linux Kernel Hacker's Guide. Chapel Hill, NC. http://sunsite.unc.edu/mdw/linux.html, 1993.

[18] Berry, M., El-Ghazawi, T. An Experimental Study of Input/Output Characteristics of NASA Earth and Space Sciences Applications. CESDIS Technical Report, TR-95-163.

APPENDIX C:

DYNAMIC I/O SCHEDULING AND PFS EVALUATIONS

In Collaboration with Sorin Nastea (GMU)
and
Ophir Frieder (GMU)

Final Version Was
Submitted to Journal of Supercomputing, June 1996

Parallel Input/Output for Sparse Matrix Computations

Sorin G. Nastea †
Department of Computer Science
George Mason University
Fairfax, Virginia, 22030-4444
E-mail: snastea@cs.gmu.edu
Phone: (703) 993-1536

Tarek El-Ghazawi‡
Department of Electrical Engineering and Computer Science
The George Washington University
Washington DC, 20052
E-mail: tarek@seas.gwu.edu
Phone: (202) 994-5507

Ophir Frieder†
Department of Computer Science
George Mason University
Fairfax, Virginia, 22030-4444
E-mail: ophir@cs.gmu.edu
Phone: (703) 993-1540

---

†This work is supported in part by the National Science Foundation under contract number IRI-9357785.

‡Supported by CESDIS/USRA subcontract number 5555-18

Abstract

Sparse matrix computations have many important industrial applications and are characterized by large volumes of data. Due to the lagging input/output (I/O) technology, compared to processor technology, the negative impact of input/output could be challenging to the overall performance of such applications. In this work, we empirically investigate the performance of typical parallel file system options for performing parallel I/O operations in sparse matrix applications and select the best suited one for this application. We introduce a dynamic scheduling method to further hide I/O latency. We also investigate the impact of parallel I/O on the overall performance of sparse-matrix vector multiplications.

Our experimental results using the Intel Paragon and standard matrix data will show that, by using our technique, tangible performance gains can be attained beyond what parallel I/O system calls alone may offer. For some data sets, it is possible to significantly ease the I/O bottleneck through latency hiding and amortization over increased computations to a limit that can preserve the scalability characteristics of the computational activities. The results will also empirically shed some light on the pros and cons associated with the different parallel file system calls supported by modern parallel systems, such as the Intel Paragon.

1. Introduction

A matrix is called sparse if a relatively small number of the matrix elements are non-zero [12]. Sparse matrices are very efficient for accommodating a variety of applications, including engineering, medical, and military data. Commonly performed matrix computations include: eigenvalues and eigenvectors computations, matrix multiplication, or solving systems of linear equations. Cheung and

Reeves [1] categorize Sparse Matrix Applications (SMA) into three fundamental classes: 1) SMA with **regular sparse patterns**, in which matrices have a regular structure, such as banded, triangular, or (block) diagonal; 2) SMA with **random sparse patterns**; 3) **Dense applications with sparse computation**, in which, although dense matrices are used, the problem deals only with a small, limited part of the data. We will focus on the second category, considered as the most general case. A number of sparse matrix compression formats exist. It is generally acknowledged that it is more efficient to deal with matrices in the compressed format for at least two reasons: 1) saving disk storage and memory space; and 2) saving execution time, as only non-zero elements participate in computations.

We empirically investigate the performance of typical parallel file system options for performing parallel I/O operations in sparse matrix applications. Nitzberg and Fineberg [15] presented an overview of raw I/O bandwidth of typical parallel systems using synthetic workloads, including the Paragon. Our investigations go beyond the study of the raw I/O performance, to include the interaction of I/O with scalable computations. In specific, we study the impact of I/O on the overall performance of typical sparse matrix computations.

One successful way to enlarge the bandwidth of I/O systems is to access the data before they are actually required by the processing nodes in computations. The technique is generally known as prefetching, and its application is strictly dependent on the application access patterns. Recently, Arunachalam, Choudhary, and Rullman [Aru96] describe the design and implementation of a prefetching strategy and provide measurements and evaluation of the file system with and without the prefetching capability. They found that, by using prefetching, a maximum speedup of 7.7 could be attained for 8 processing nodes and 8 I/O nodes. Even if we also use prefetching as a way of

boosting I/O bandwidth, some important aspects differentiate our work than the one performed by Arunachalam *et al.* Thus, instead of using variable delays as simulated **load-balanced** computations, we use real applications, including one with **inherent load-imbalances**, such as the sparse matrix-vector multiplication. Therefore, our research goal is to go beyond just measuring system's capabilities, to test our I/O bandwidth improving solutions and the response of the Paragon PFS in complex real-life situations. We introduce techniques and methods to significantly ease the I/O bottleneck through latency hiding and load balancing to a limit that can preserve the scalability characteristics of the computations. Also, we base most of our experiments on the M_ASYNC file access mode, proven to yield best performance [15, 7, 16] and because of its suitability for MIMD-type implementations. On the other hand, Arunachalam *et al.* based their tests on M_RECORD, which is most suitable for SIMD implementations.

The remainder of this paper is structured as follows. In Section 2, we present the two Sparse Matrix Applications (SMA) (compression and multiplication) used to study the performance of the I/O system. In Section 3, we describe our experimental testbed, the parallel platform and the matrix data set. In Section 4, we present the Paragon PFS file access modes and our I/O latency hiding methods. Finally, in Sections 5 and 6 we present our experimental results and conclusions, respectively.

2. Sparse Matrix Applications

Our purpose is to find appropriate solutions and techniques to enhance the I/O performance on parallel computers, for applications such as sparse matrix computations. Therefore, we investigated the I/O bottleneck for two typical Sparse Matrix Applications (SMA): sparse matrix

compression and sparse matrix-dense vector multiplication.

2.1 Sparse matrix compression

The quality of compression formats should be judged by considering a number of criteria, including: the compression ratio (i.e. the ratio between the sizes of the matrix in the compressed and in the extended format, respectively), the availability of compressed matrix elements to participate in efficient algorithmic constructions, and the possibility of modifying, extending, and regenerating the original matrix. Some of the most used sparse matrix compression techniques include the Scalar ITPACK [3,10, 11], Horowitz [5, 9], Vector ITPACK [12], ITPLUS [4, 10], and ITPER (ITPACK permuted blocks) [6, 10]. There is no globally accepted best storage technique. The selection of the compression format depends on the actual distribution of non-zero elements within the matrix and on the application requirements. In our experiments, we used Scalar ITPACK compression format, given its suitability for general sparse pattern matrices. This compression technique yields good compression ratio and enables efficient algorithmic constructions.

$$A = \begin{bmatrix} 2 & 0 & 5 & 0 & 0 \\ 8 & 3 & 0 & 7 & 0 \\ 0 & 6 & 2 & 0 & 1 \\ 9 & 0 & 0 & 1 & 0 \\ 0 & 7 & 0 & 0 & 2 \end{bmatrix}$$

$$a[nz] = [2\ \ 5\ \ 8\ \ 3\ \ 7\ \ 6\ \ 2\ \ 1\ \ 9\ \ 1\ \ 7\ \ 2]$$

$$ja[nz] = [1\ \ 3\ \ 1\ \ 2\ \ 4\ \ 2\ \ 3\ \ 5\ \ 1\ \ 4\ \ 2\ \ 5]$$

$$ia[N+1] = [1\ \ 3\ \ 6\ \ 9\ \ 11\ \ 13]$$

Figure 1. Example illustrating scalar ITPACK format

We illustrate the **Scalar ITPACK** format through an example in Figure 1, in case of a matrix stored row-wise. It stores values and information regarding the position within the matrix of the non-zero elements into three vectors, as follows: 1st vector stores non-zero values; 2nd vector stores their

column indices; and 3rd vector stores indices of elements in 1st and 2nd vectors corresponding to beginning of rows.

2.2 Sparse matrix-dense vector multiplication

Additionally from the sparse matrix compression, we used the sparse matrix-dense vector multiplication as scalable computation. The algorithm multiplies matrix elements compressed

Procedure 2: multiplication

INPUT:
A[N, M] - matrix compacted according to the scalar ITPACK format:
a[nz] - contains all **nz** non-zero values ;
ja[nz] - contains the corresponding column indexes of all the elements in vector a[ ];
ia[N+1] - contains the indices of elements in vector a[ ] that correspond to new rows.
x[M] - an M x 1 input vector;

OUTPUT:
y[N] - an N x 1 output vector.

ALGORITHM:

```
for ( i =1, N )
        temp=0;
        for ( k = ia [i], ia[i+1] - 1 )
                temp = temp + a[k] * x [ ja[k]];
        endfor
        y [ i] = temp;
endfor
```

Figure 2: Sparse matrix-dense vector multiplication algorithm

according to the Scalar ITPACK compression scheme (Figure 2) and it is considered a typical sparse matrix multiplication scheme [6].

3. Experimental testbed

3.1 The Paragon system

Our experiments were performed on an Intel Paragon parallel computer with 64 processing nodes, among whom 56 are compute nodes. Each node is based on an Intel i860 processor, having at least 16 MBytes of RAM on-board. The underlying topology of this MIMD machine is a mesh that enables up to 160 MBytes/s of inter-node communications bandwidth. Large files can be stored on a Parallel File System (PFS), organized as a two disks system, each of them a RAID 3. File contents are striped over disks with stripe sizes equal to 64 KBytes. Conceptually, the system combines fine-grained parallelism within each RAID 3 with coarse-grained parallelism at PFS level. At the concrete level, the system represents a multi-level striping implementation to achieve a better distribution of load over I/O nodes. We capture in Figure 3 the hierarchical structure of a Paragon PFS with two I/O nodes each one controlling a disk system.

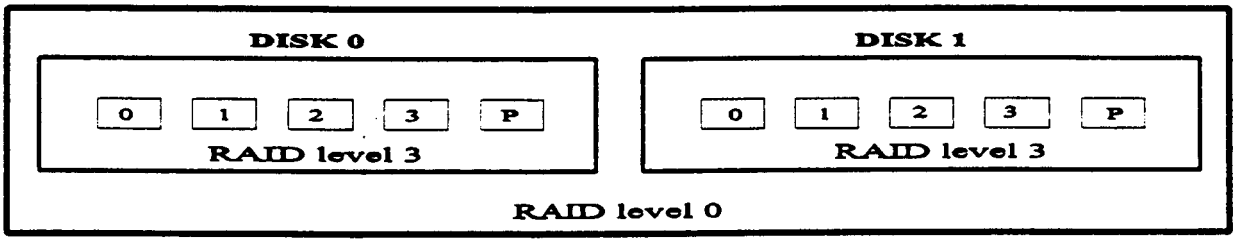


Figure 3. Hierarchical structure of a Paragon PFS with two disk systems

3.2 The matrix data

In our experiments, we use several sparse matrices selected from the Harwell-Boeing sparse matrix collection [2,3]. We summarize the main characteristics of these matrices in Table 1. The

Harwell-Boeing collection is a set of benchmark matrices collected from challenging practical applications of typical computational problems. Both the User's Guide and the collection are available on the Internet free of charge.

Table 1. Statistical data of sparse matrices selected from the Harwell-Boeing collection

Name	Type	Order	Non-zero's	Sparsity
ORANI 678	Unsymmetric	2529	90158	0.014
PSMIGR 1	Unsymmetric, mostly block-diagonal	3140	543162	0.055
BCSSTK28	Symmetric	4410	219024	0.011

Table 2. PFS file access modes main characteristics

PFS file access mode	File pointer	File access policy	Degree of synchronization	Typical applications
M_UNIX	independent, multiple	random access first-come, first-served basis atomicity ensured	small	read/write on disjoint areas
M_LOG	single, shared	first-come, first-served basis access by order of issuing the call	medium	writing log files
M_SYNC	single, shared	synchronized access by node number	high	round-robin data read/write
M_RECORD	multiple	concurrent access *appearing* to have been done by node number	high	more efficient round-robin data read/write
M_GLOBAL	single, shared	data read by one node and broadcast to all others	high	reading shared data
M_ASYNC	independent, multiple	random, concurrent access non-atomic writes	small	complete flexibility of implementation

4. Parallel File System (PFS) file access modes and I/O latency hiding

4.1 Supported PFS file access modes on an Intel Paragon

The Intel Paragon supports the following PFS file access modes: M_UNIX, M_LOG, M_SYNC, M_RECORD, M_GLOBAL, and M_ASYNC. The main differences consist of the way the contents of the file pointer is maintained and the degree of file access synchronization.

In Table 2, we summarize the main characteristics of the PFS file access modes. We have assumed the following interpretation for the degrees of inter-node synchronization:

1) small - only **open** and **close** calls are synchronizing;

2) medium - additionally, calls like **fseek** and **eseek** are synchronizing;

3) high - most or all calls are synchronizing, including the **read/write** calls.

By *synchronizing calls* we understand two things. First, corresponding syncronizing calls have to exist in the code run by all compute nodes in a compute partition. Second, the system executes them in some particular way. Thus, if calls such as *fopen*, *fclose*, *fseek*, or *eseek* are involved, all processors execute them at the same time, performing the same action (such as moving the file pointer to a same file location). On the other hand, if *fread* or *fwrite* calls are involved, the calls are scheduled based on node number and the operations are performed at file locations given by the node number and size of the read/write call.

Some of the file access modes have some typical applications. For example, the M_LOG mode is most suitable for creating and maintaining log files, whereas M_GLOBAL has its best application in implementing a variation of collective read of a file, when all nodes are reading the same information from disk, but only one node is actually performing the read followed by a broadcast of the read data through inter-node message-passing. Our implementations are aimed

at setting a fair basis of performance comparison for all these modes rather than providing the most suitable application for each of the parallel file access modes. A complete description of the PFS file access modes can be found in the Paragon User's Guide manual [14].

The sparse matrices are uncompressed and resident on disk. Before compression and/or computations, such as matrix multiplication, are performed, data have to be read into main memory. Some of the PFS file access modes (M_LOG, M_SYNC, M_RECORD, and M_GLOBAL) make either all nodes truly share the same pointer or make the seek operations transparent to the user (M_RECORD). All of these file access modes, with the exception of M_LOG, offer some means of synchronizing the calling nodes. M_LOG is designed for implementing log files, therefore the access to the file is truly on a first-come, first-served basis, with single shared file pointer. Because in our case the nodes have to read specific data from disk rather than to write on it, some file pointer alignment information must be exchanged by the nodes. Each node has to be aware of the index of the rows it processes to ensure the correctness of the results. In the case of M_SYNC and M_RECORD file access modes, the access to the file is done by node number. Thus, for M_SYNC accessing nodes are truly synchronized by node number. M_RECORD is a special case: even if the file pointer is distributed, compute nodes do not have full control on the contents of the file pointer and fseek and eseek calls are synchronizing. Performing read-write is similar to M_SYNC, but the access of nodes to the file just looks to have been by done by node number, but it actually is on a first-come, first-served basis. To have this opportunity, additional constraints are imposed, such that each node must perform the same type of operation in a read/write session and use the same buffer length. For some applications, in which the file size is not evenly divisible by the number of processors multiplied by the read-block size, errors may occur if these restrictions are not met and

make M_RECORD mode impractical. One way of solving this problem of incompatibility between the PFS characteristics and the application requirements is to artificially enlarge the size of the file such that, in the read session, all nodes are fetching the same number of bytes. This anomaly is corrected in the computations phase when fudge data are simply discarded.

In the case of M_UNIX and M_ASYNC modes, multiple file pointers have to be appropriately maintained. As no synchronization restrictions are imposed, both these two file access modes enable flexible scheduling of file access and, therefore, they support MIMD-type implementations.

4.2 Additional latency hiding

A typical way of hiding the I/O latency is the use of asynchronous read. Modern MIMD machines have dedicated hardware facilities for achieving the message passing task and for interfacing the I/O devices. This architectural concept enables processing nodes to carry on their computations without being directly involved in communications and I/O tasks. Ideally, the full benefit from an asynchronous I/O call is attained when the I/O operation has the same length as the computations performed between two consecutive asynchronous calls. In such a case, I/O is fully overlapped with computations. Unfortunately, this is not always easy to reach in practice, and application programs should make every effort to achieve the maximum possible degree of overlapping.

Load-balancing is an important issue in computations and I/O as well. The structure of the Paragon PFS itself contributes decisively at ensuring an even distribution of load over I/O nodes. However, especially for a PFS with a large number of I/O nodes, the completion of some I/O calls

can be delayed primarily because the PFS is a shared resource with other users. Therefore, some compute nodes that posted the delayed calls can experience significant load imbalances. Moreover, computations themselves can contribute and expand the existing load-imbalance. Thus, we perform two common computations with sparse matrices: compression and the multiplication with a variable number of vectors. As opposed to the compression, that does not raise special load-balancing problems, the multiplication is typically a good example of a potential source for load-imbalances. Additionally, the size of the multiplication and its associated challenges can be easily scaled by changing the number of multiplying vectors.

We compare two approaches for allocating the data to compute nodes to solve the parallel sparse matrix compression and multiplication:

(1) each node generates, based on its node index, the rows it reads. Thus, the first node reads the first n rows, the second node reads the next n, and so on. At the next read cycle, the first node reads rows: $N*n, \ldots, (N+1)*n - 1$. As a rule, node j reads within read cycle i (if it is the last read cycle, some nodes may not read at all, and one node may read less) rows: $(N*i+j)*n, \ldots, (N*i+j+1)*n-1$. Thus, each node reads disjoint areas from the file. The physical access to the file is imposed by the selected PFS file access mode. The advantages are the following: the method enables large size read sessions and the nodes extract the information about the read data based on their node index. The main disadvantage of this method is that it does not attempt to evenly distribute the load onto processing nodes. Based on the actual distribution of the data elements, some nodes may take longer than others to process a particular subset of the matrix elements. We call this approach, that is mainly a static allocation, the **worker-worker approach (W-W)**. This approach is aimed at solving problems with no or with insignificant load-imbalances.

(2) each node informs the major node when it is able to process new data (**master-worker approach M-W**). As a principle, any worker that is ready to accept new data, sends a READY message to the master node. If the master node, that manages the allocation of rows to workers, still has unprocessed rows to allocate, sends back a GO message in which includes the starting row index and the number of allocated rows. Once the EOF is reached, the master node broadcasts a STOP message. The algorithm efficiently mixes asynchronous message passing and I/O calls with computations for best performance. We present both the master and the corresponding worker algorithms in Figure 4. This algorithm is a more elaborate alternative for potential sources of load-imbalances.

One difference between the two approaches is the need for a coordinating master node in the second approach. In the first approach, the indices of rows fetched and processed by each compute node can be determined according to the node number of each processor. In a dynamic allocation, a coordinator is necessary to arbitrate requests for more work from compute nodes and keep track of the allocated row indices and of the remaining work. A master node can be a service node, while the workers are compute nodes. The advantage of this procedure is that any delayed node is not delaying the whole process. As the I/O calls and the associated computations generally take longer to complete than the short message inter-change, the message passing that referees the data allocation is carried on in the background. Therefore, there is no measurable penalty paid for the communication between processing nodes (workers) and the managing node (master). Whenever a master node is involved in coordinating a parallel application, the concern that the master node may become a serious bottleneck is raised. In this situation, the load associated with the master node is minor as compared to the load allocated to compute nodes that deal both with

```
Algorithm: master
        compute the number of I/O reads
        for (all I/O reads)
                receive a READY message from a worker
                identify the node that sent the READY message
                send back GO message with row identifiers
        endfor
        for (all processing nodes)
                receive a READY message from a worker
                identify the node that sent the READY message
                send back STOP message with row identifiers
        endfor


Algorithm: worker
        post an asynchronous receive for a STOP message
        read asynchronously the first set of data based on its node index
        post an asynchronous receive GO message
        send READY message to master node
        Terminate=FALSE
        while not (Terminate)
                for ( ; ; )
                        if GO message received from master node
                                post a new asynchronous receive GO message
                                send READY message to master node
                                break from for loop
                        endif
                        if STOP message received
                                cancel GO receive message
                                Terminate=TRUE
                                break from for loop
                        endif
                endfor
                wait until previous I/O read ends
                if not (Terminate)
                        post a new asynchronous read from disk for next compute iteration
                endif
                perform computations on data previously read from disk
        endwhile
```

Figure 4. Dynamic allocation of I/O and computations

I/O and computational tasks. Therefore, the major node is not impeding on the scalability of the overall execution. To implement the dynamic (or Master-Worker) allocation, two memory buffers

may be used. At any moment in the infinite for loop, while one buffer is being filled with data from disk, the other one, already containing data, is being used in the computations phase.

5. Experimental results

5.1 Performance measurements of the PFS file access modes

In this section, we compare the performance of the PFS file access modes available on an Intel Paragon. A short preview at these results appeared in the Proceedings of the DCC'96 [7].

Suppose we solve the problem of compressing a sparse matrix, resident in extended format on the hard disk. Computation and I/O operations must be performed by a number of processors grouped into a computing partition. The compression and, also, other computations, like the matrix-vector multiplication, can be embedded in the dynamic allocation algorithm presented in Figure 4 as the computation taking place at the worker node level.

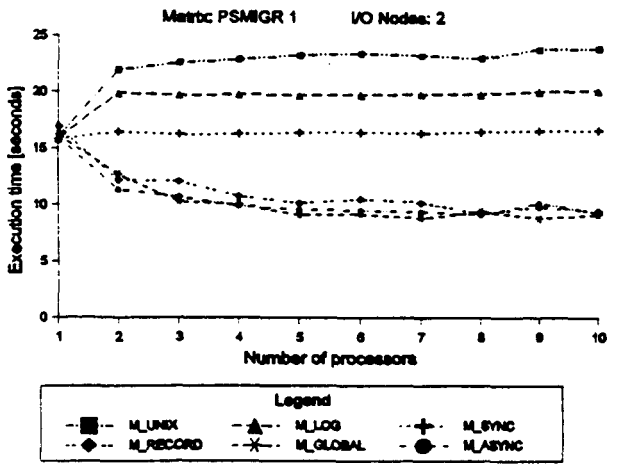
The purpose of our experiments is to show how computations and I/O intensive applications can be handled efficiently. Therefore, we tested in our experiments all possible PFS file access modes, and we compared their achieved performance. We respected some principles in designing the compression process for implementations involving each of the PFS file access modes:

1) Each node reads an integer number of rows, as the compression is row-oriented. This generally contradicts the constraint to read in integer number of stripe-sizes to achieve high performance. We compromised on these issues, in the sense that each node reads at one time a number of rows whose combined size is closest to the optimal read size.

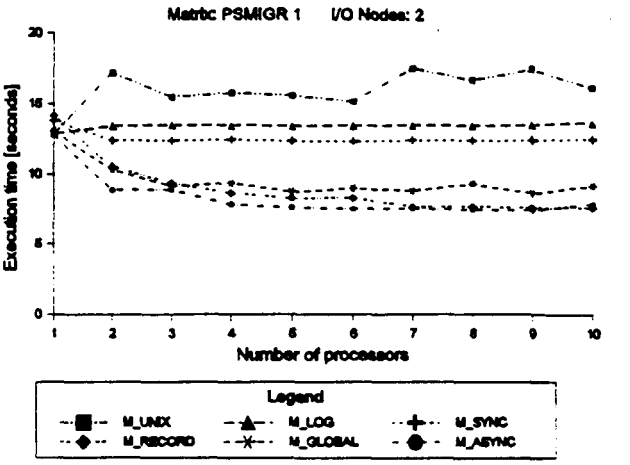
2) Asynchronous message passing and asynchronous I/O reads are efficiently used. They enable

the computations to occur concurrently with inter-node communications and parallel I/O.

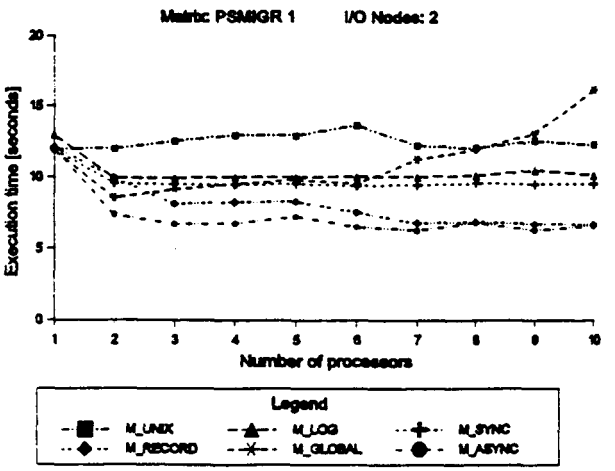
In Figure 5, we present our results regarding the comparative performance evaluation of the PFS file access modes. We have used 4 different read-sizes, multiple of the stripe-size (64 KBytes): 1 (64 KBytes), 2 (128 KBytes), 4 (256 KBytes), and 6 (384 KBytes) stripe-sizes. Note that our system has 2 I/O nodes. The best overall performance is obtained with the M_ASYNC PFS file access mode (Figure 5). The explanation is that all restrictions that apply to the other PFS file access modes are lifted in this case. Good performance is achieved with M_GLOBAL with a read-size equal to a stripe-size (Figure 5a). The explanation is that only one node is actually reading the data, followed by the data replication on the inter-node communications network, thus avoiding all contentions in accessing the disk. As each node processes the same amount of data during one read cycle, the actual read-size is the basic read-size multiplied by the number of processors. However, this feature becomes an aggravating issue once the read-size increases, as it tends to trigger paging, thus diminishing the overall performance. This behavior is clearly illustrated when the read sizes are equal to multiple stripe-sizes (Figure 5 c, d). File access with M_SYNC and M_LOG is highly synchronized. The reading with M_SYNC is done by node number, while with M_LOG is performed on a first-come, first-served basis. Therefore, the degree of concurrency allowed is much smaller. However, M_SYNC performs better than M_LOG, because it enables a higher degree of concurrent access of nodes to the file. Similarly to M_LOG, file access with M_UNIX is also performed on a first-come, first-served basis. Not surprisingly, its performance is the worse compared to all other PFS file access modes, because it allows the least concurrency to file access. Worth noting that a significant penalty is paid because of the existence of multiple file pointers, that have to be maintained individually, compared to the single, shared file



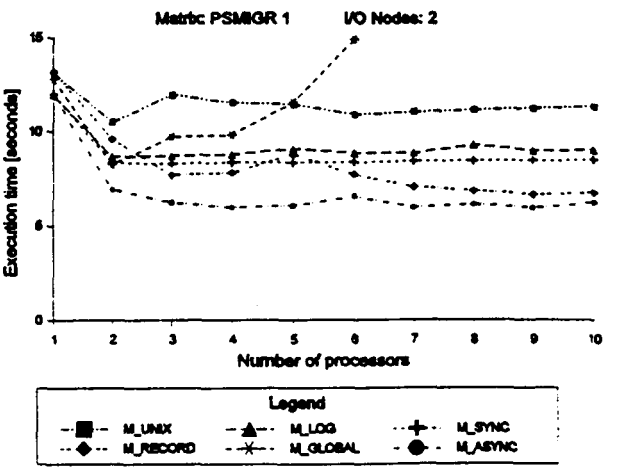
a) Read size = STRIPE_SIZE = 64 KB



b) Read size = STRIPE_SIZE*IO_NODES = 128 KB



c) Read size = 2*STRIPE_SIZE*IO_NODES = 256 KB



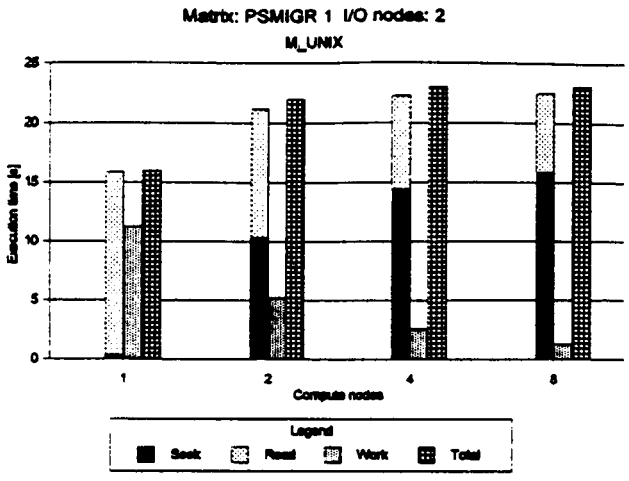
d) Read size=3*STRIPE_SIZE*IO_NODES=384 KB

Figure 5. Compression and associated I/O execution time for matrix PSMIGR 1

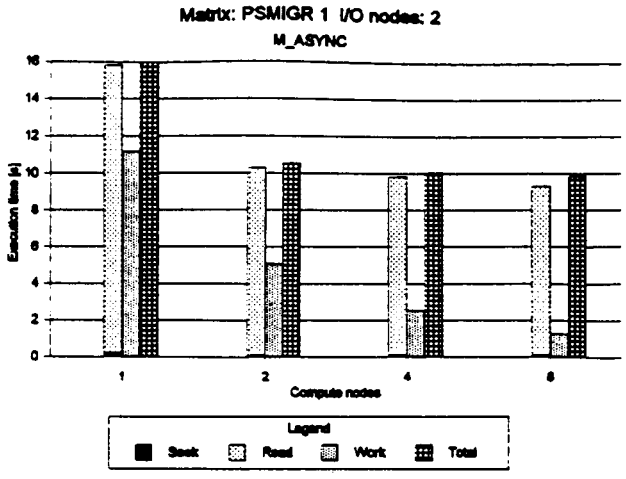
pointer modes. M_RECORD also yields better performance than M_SYNC because it allows a higher degree of concurrent access to the file. M_ASYNC and M_UNIX are very similar in some aspects, but they produce completely different results because of the lack of constraints in the M_ASYNC case. A dramatic improvement is recorded for the M_ASYNC in the time to move the file pointer. Thus, the total time to move the file pointer for the M_UNIX ranges between 3.7 and 16.7 seconds for the studied cases, representing from less than 35% to up to more than 75% of the whole I/O cost. On the other hand, the same operation takes only tens of a second for implementations using the M_ASYNC PFS file access mode and is negligible as compared to the entire I/O cost (less than 1% of the entire I/O cost). To explain more thoroughly the way timing results are made up, in Figure 6 we show the compression time results based on the M_ASYNC mode detailed by I/O and processing components for both M_UNIX and M_ASYNC modes. An estimation of the overall execution time is:

$$T_{total} = T_{seek} + \max \{T_{work}, T_{I/O\_read}\} \quad (1)$$

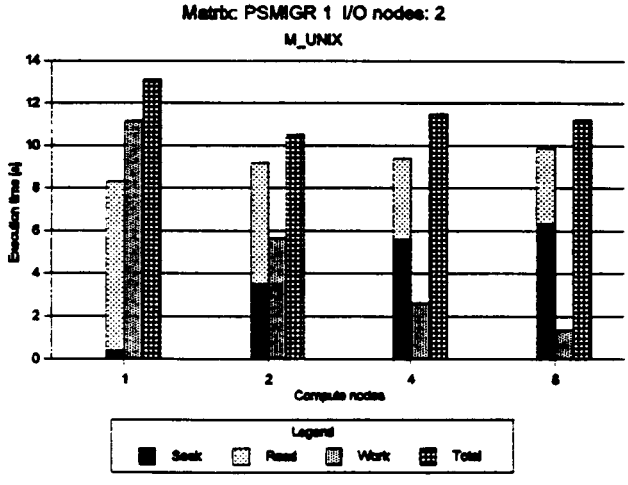
For a read-block size equal to 64 KBytes, the read time is completely above the computations time. Therefore, the overall execution time is imposed by the I/O performance. For a read-block size equal to 384 KBytes and M_ASYNC mode, the overall results scale for a small number of processors ($p \leq 2$), but the I/O time becomes preponderant for large number of processors. This behavior is due to the unscalable characteristics of the PFS that diminishes overall performance for insufficient amounts of computations. The charts emphasize the scalability of the computation as compared to the saturation of the I/O performance. In Figure 6, we also illustrate another interesting aspect: the cost of seeking overwhelms the cost for read for M_UNIX and large number



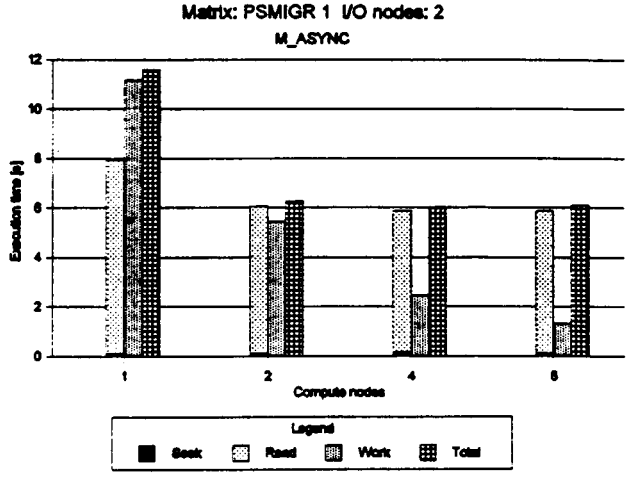
Read size: 64 KB



Read size: 64 KB



Stripe size: 384 KB



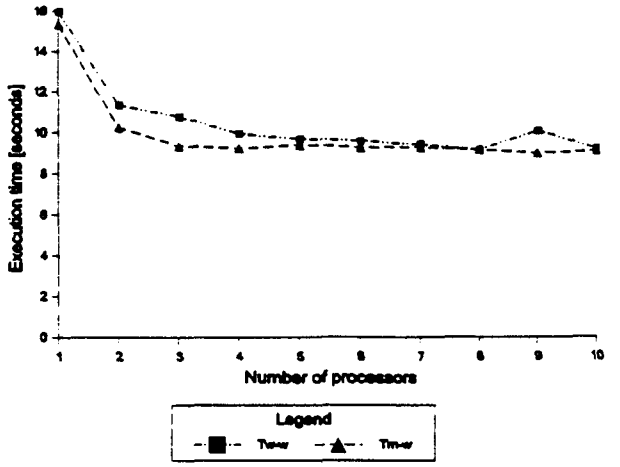
Read size: 384 KB

Figure 6. Comparison of the structure of the overall execution time for M_UNIX and M_ASYNC

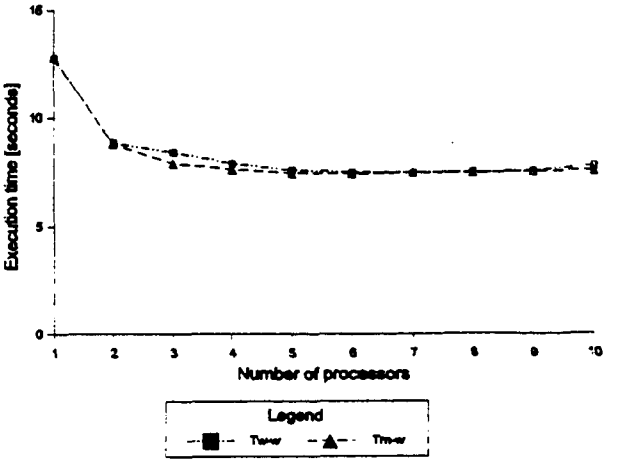
of processors.

5.2 Effect of the additional I/O latency hiding

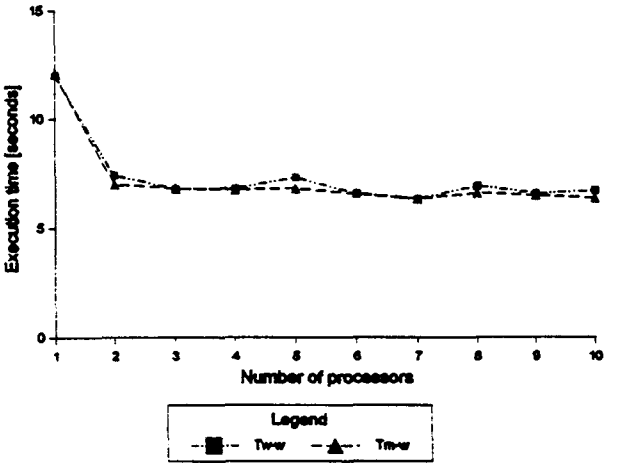
As previously shown, the I/O bottleneck impacts directly on the overall performance. Therefore, the application designer should use any available techniques that speed up the I/O and/or hide its latency. In Section 4.2, we present a method that uses dynamic allocation (the Master-Worker approach) as a meaningful attempt to achieve a better distribution of the work over compute and I/O nodes. The method aims at avoiding the idle times of both compute and I/O nodes due to any unevenness of the data distribution. In Figure 7, we compare the performance obtained with the static and dynamic allocations for different read sizes. We show that the dynamic allocation produces better performance in all 6 cases. Once again, the improvements are not based on a faster I/O, but on a better use of allocations. In this section, our tests are based on operations, such as the I/O and compression, that, normally, do not raise special problems of load-balancing. However, as the load on I/O can generate from other sources than our job as well, the dynamic allocation proves to be useful. In Table 3, we capture some statistical data on this improvement. The measured improvement due to dynamic allocation (the Master-Worker approach) is up to 19.79 %. In this case, load-balancing is aimed at smoothing sudden load-unevenness due to external sources rather than internal ones, therefore it helps prevent loss of performance due to random rather that persistent and systematic stimuli. We expect this technique to have an even larger and more conclusive impact when allocated loads have various weights, such as some types of computations involving sparse matrices. We further address this issue in Section 5.3.



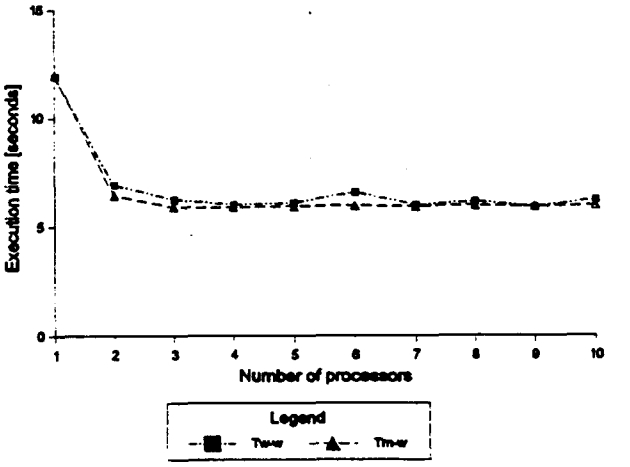
a) Read size =STRIPE_SIZE = 64 KB



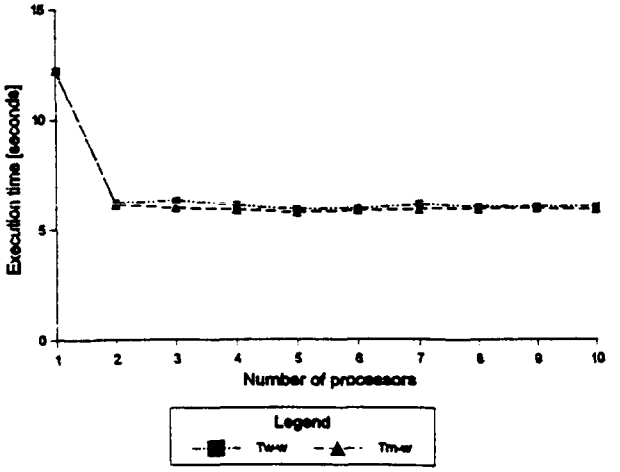
b) Read size = STRIPE_SIZE*IO_NODES = 128 KB



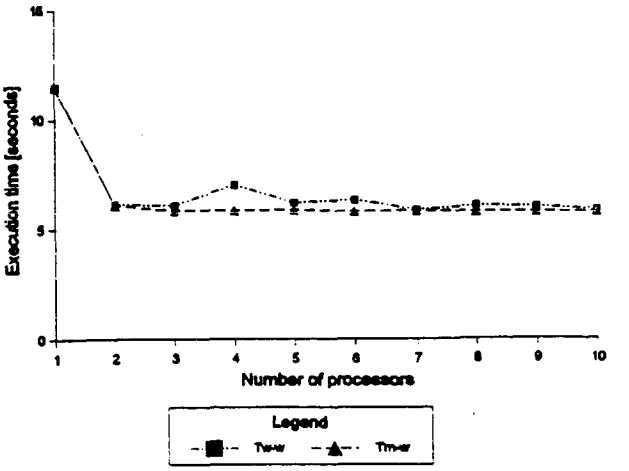
c) Read size =2* STRIPE_SIZE*IO_NODES = 256 KB



d) Read size=3*STRIPE_SIZE*IO_NODES=384 KB



e) Read size =4* STRIPE_SIZE*IO_NODES = 512 KB



f) Read size = 5*STRIPE_SIZE*IO_NODES = 640 KB

Figure 7. Comparison between the static and the dynamic allocations

Table 3. Improvement of the performance with the dynamic allocation vs. static allocation

Increase in performance	Read sizes					
	64 KB	128 KB	256 KB	384 KB	512 KB	640 KB
Highest [%]	15.97	6.49	7.53	10.25	5.04	19.79
Lowest [%]	1.5	0.4	0.15	0	1.16	0.16

5.3 Effect of the multiplication size on scalability and amortization of I/O

As we previously stated, computations with high degree of inherent concurrency scale well, compared to I/O operations. To study the effect of the size of the problem on the overall performance of computations and I/O, we have increased the complexity of the computations part. Thus, we have combined the compression of a sparse matrix with the multiplication of this matrix (in the compressed format) with a number of dense vectors. The sparse matrix-dense vector multiplication has two interesting effects. First, the operation itself raises load balancing problems, as the computation load depends on the number of non-zero elements contained in each amount of data read from disk by each node. Depending on the distribution of non-zeros per each row, the variations of load can become non-negligible. Second, by varying the number of vectors, we can conveniently modify the ratio between the amounts of computations and I/O.

In Figure 8, we capture the interrelation among overall scalability, I/O, and amortization of I/O with increased computations. As expected, the overall results scale well as long as the scalable computation part surpasses the I/O part. Each of the curves in Figure 8 has a scalable segment and a saturated one. One interesting result is that the amortization is achieved at an extremely reasonable size of the multiplication problem (vectors$\geq$ 100). This is the direct outcome from the

combination of techniques used to increase the performance of the I/O itself, such as asynchronous read and dynamic allocation. The conclusion to be drawn from these results is that, if the I/O operation can be overlapped (by using asynchronous calls) with scalable computations, there exists some size of the scalable operation for which the overall results become scalable.

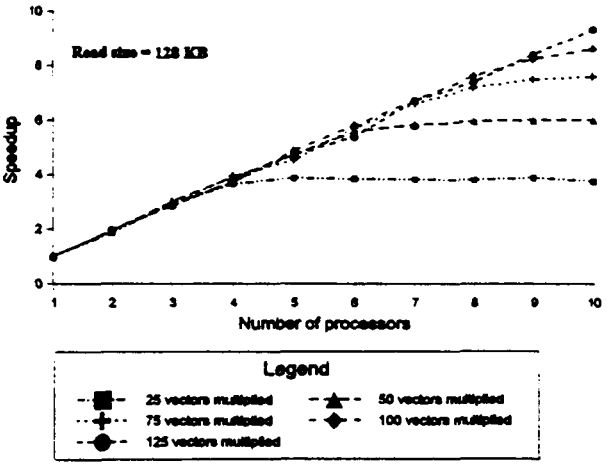
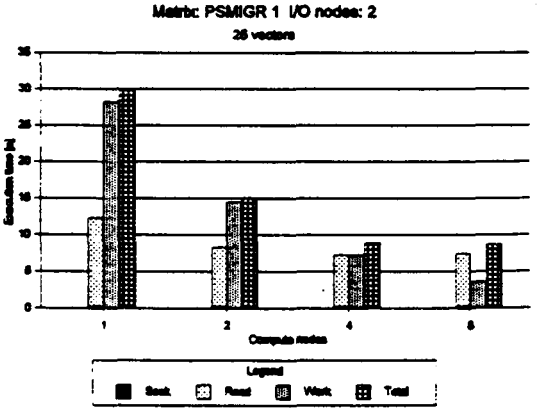
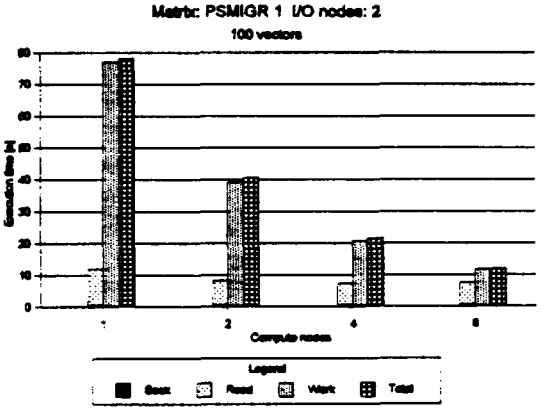


Figure 8. Impact of the problem size on scalability and I/O amortization



Read size: 128 KB



Read size: 128 KB

Figure 9. Structure of the overall results for compression and multiplication

To show more clearly how these results were obtained, in Figure 9 we detail the components of the overall execution time. Thus, for 25 multiplying vectors, the time to perform

computations is above the time for I/O only for 4 compute nodes or less. As the amount of computations increases (100 vectors), the scalable computations cost surpasses the I/O cost for the entire range of 10 compute nodes. Thus, overall scalable performance can be obtained when an unscalable operation, such as I/O, can yield enough work to overlap with and completely hide behind the scalable computations. Therefore, when two operations, among whom one is scalable and the other one is unscalable and whose costs can overlap, pass the scalable characteristics of the overwhelming operation cost to the entire process for a wide range of compute nodes.

To generate the results in Figure 8, we used the dynamic allocation. We show once again a comparison between the two possible types of allocation, given the increased interest due to the potential load-balancing problems embedded into the sparse matrix multiplication. In Figure 10, we plot the speed up results for I/O, compression, and multiplication with 125 vectors with both the static and dynamic allocations. In addition to what was presented in Figure 6, the multiplication itself involves a random amount of computations. Therefore, variations of processor behavior may occur, and the dynamic allocation is aimed at covering the possible node delays. The differences in speed up obtained using the static and dynamic allocations are more obvious for larger number of processors because load-imbalances have a larger relative effect for smaller absolute work loads. The dynamic

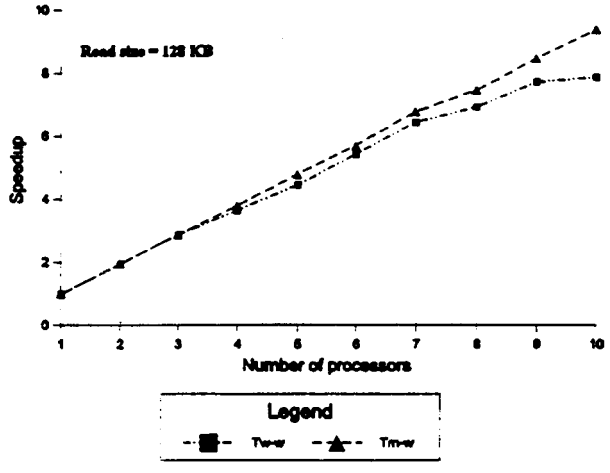


Figure 10. Comparison of the static (w-w) and dynamic (m-w) allocations of I/O, compression and multiplication of 125 vectors for a 3140 x 3140 sparse matrix

(m-w) allocation yields a speed up of 9.405 for 10 nodes, compared to a speed up of 7.921 obtained with the static (w-w) allocation for the same number of processing elements. Figure 10 shows that, to achive high performance, it is not enough to make the scalable operation preponderant, but also to ensure a high degree of scalability to the computations themselves by appropriately choosing the load-balancing techniques according to the problem and execution model.

6. Conclusions

We have studied the effect of the I/O bottleneck on the performance of some basic sparse matrix operations, such as the compression and the multiplication. Our experiments were performed on an Intel Paragon MIMD machine. In these experiments, we used benchmark matrices selected from the Harwell-Boeing collection. We compared the performance of all applicable PFS file access modes and we showed empirically the performance characteristics of each of them in real-life applications. It was particularly shown that, although M_GLOBAL does better for smaller collective reads, due to the ease in managing the single file pointer, it results in loss of performance in larger read sizes, due to local memory size constraints. M_ASYNC, on the other hand, can schedule the multiple file pointers intelligently, performing better than M_GLOBAL for large I/O read sizes. We introduced a dynamic allocation that takes place in the background of the I/O operation. We show that execution time improvements of 10% or more can be obtained with this technique, and we expect even better behavior on more skewed data distributions. Also, we studied the effect of scalable computation on hiding the I/O bottleneck, and we showed that for

moderate sizes of scalable problems, the I/O latency can be effectively hidden using a combination of asynchronous calls and dynamic load balancing.

References

[1] Cheung, A. L., Reeves, A. P., *Sparse data representation,* Proceedings Scalable High Performance Computing Conference, 1992.

[2] Duff, I. S., *Sparse matrix test problems,* ACM Transactions on Mathematical Software, Vol. 15, No. 1, 1-14, 1989.

[3] Duff, I. S., Grimes, R.G., Lewis, J. G., *User's guide for the Harwell-Boeing Sparse Matrix Collection,* CERFACS Report TR/PA/92/86, 1992.

[4] Fernandes, P, Girdinio, P., *A new storage scheme for an efficient implementation of the sparse matrix-vector product,* Parallel Comp. 12 (1989) 327-333.

[5] Horowitz, E., Sahni, S., *Fundamentals of data structures*, Computer Science Press, Rockville, MD, 1983.

[6] Nastea, S. G., Frieder, O., El-Ghazawi, T., *Sparse matrix multiplication on highly parallel computers*, Proceedings of the 10th Int'l Conference on Control Systems and Computer Science, Bucharest, Romania, 1995.

[7] Nastea, S. G., El-Ghazawi, T., Frieder, O., *Parallel Input/Output Impact on Sparse Matrix Compression*, Proceedings of the IEEE Data Compression Conference, Snowbird, Utah, 1996.

[8] Paolini, G. V., Santangelo, P., *A graphic tool for the structure of large sparse matrices*, IBM Technical Report, ICE-0034 IBM ECSEC Rome (1989).

[9] Park, S. C., Draayer, J. P., Zheng, S. Q., *Fast sparse matrix multiplication,* Computer Physics

Communications, Vol. 70, 1992.

[10] Peters, A., *Sparse matrix vector multiplication technique on the IBM 3090 VP,* Parallel Computing 17, 1991.

[11] Petiton, S., Saad, Y., Wu, K., Ferng, W., *Basic Sparse matrix computations on the CM-5*, International Journal of Modern Physics C vol. 4, No. 1, 63-83, 1993.

[12] Press, W. H., Flannery, B. P., Tenkolski, S. A., Weterling, W. T., *Numerical recipes. The art of scientific computing*, Cambridge University Press, pp. 64-73, 1986.

[13] Rothberg, E., Schreiber, R., *Improved load balancing in parallel sparse Choleski factorization*, "Supercomputing '94", Washington D.C., 1994.

[14] *** Intel Paragon User's Guide, 1995.

[15] Nitzberg, B., Fineberg, S. A., Parallel I/O on Highly Parallel Systems, Tutorial notes, Proceedings "Supercomputing '94", Washington D. C., 1994.

[16] Arunachalan, M., Choudhary, A., Rullman, B., *Implementation and evaluation of prefetching in the Intel Paragon Parallel File System*, Proceedings of the 10th Int'l.Parallel Processing Symposium, Honolulu, Hawaii, April 15-19, 1996.

NASA

Report Documentation Page

1. Report No.	2. Government Accession No.	3. Recipient's Catalog No.

4. Title and Subtitle
Understanding and Improving High-Performance I/O Subsystems

5. Report Date

6. Performing Organization Code

7. Author(s)
Tarek A. El-Ghazawi
Gideon Frieder
A. James Clark

8. Performing Organization Report No.

10. Work Unit No.

9. Performing Organization Name and Address
Department of Electrical Engineering and Computer Science
School of Engineering and Applied Science
The George Washington University
Washington, D.C. 20052

11. Contract or Grant No. NAS5-32337
USRA subcontract No. 5555-18

12. Sponsoring Agency Name and Address
National Aeronautics and Space Administration
Washington, DC 20546-0001
NASA Goddard Space Flight Center
Greenbelt, MD 20771

13. Type of Report and Period Covered Final
August 1, 1993 - September 30, 1996

14. Sponsoring Agency Code

15. Supplementary Notes
This work was performed under a subcontract issued by
Universities Space Research Association
10227 Wincopin Circle, Suite 212
Columbia, MD 21044 Task 16

16. Abstract This report is divided into three parts: Characteristics of the MasPar Parallel I/0 System, An Experimental Study of Input/Output Characteristics of NASA Earth and Space Science Applications, and Parallel Input/Output for Sparse Matix Computations. Detailed below is the abstract for the first topic discussed.

Input/Output speed continues to present a performance bottleneck for high-performance computing systems. This is because technology improves processor speed, memory speed and capacity, and disk capacity at a much higher rate. Developments in I/0 architecture have been attempting to reduce this performance gap. The MasPar I/0 architecture includes many interesting features. This work presents an experimental study of the dynamic characteristics of the MasPar MP-1 and MP-2 testbeds at NASA GSFC. The results have revealed many strengths as well as areas for potential improvements and are helpful to software developers, systems managers, and system designers.

17. Key Words (Suggested by Author(s))	18. Distribution Statement	
	Unclassified--Unlimited	
19. Security Classif. (of this report)	**20. Security Classif. (of this page)**	**21. No. of Pages**	**22. Price**
Unclassified	Unclassified	52	

NASA Form 1626 Oct 86