Tuning the Performance of I/O Intensive Parallel Applications

Anurag Acharya Robert Bennett
Michael Beynon Jeff Hollingsworth
Assaf Mendelson Joel Saltz
Alan Sussman Mustafa Uysal

CRPC-TR95632-S
October 1995

Center for Research on Parallel Computation
Rice University
6100 South Main Street
CRPC - MS 41
Houston, TX 77005

Tuning the Performance of I/O Intensive Parallel Applications[1]

Anurag Acharya, Mustafa Uysal, Robert Bennett, Assaf Mendelson, Michael Beynon, Jeff Hollingsworth, Joel Saltz, Alan Sussman

Dept of Computer Science, University of Maryland, College Park MD 20742
{acha,uysal,robertb,assaf,beynon,hollings,saltz,als}@cs.umd.edu

Abstract

Getting good I/O performance from parallel programs is a critical problem for many application domains. In this paper, we report our experience tuning the I/O performance of four application programs from the areas of sensor data processing and linear algebra. After tuning, three of the four applications achieve effective I/O rates of over 100Mb/s, on 16 processors. The total volume of I/O required by the programs ranged from about 75MB to over 200GB. We report the lessons learned in achieving high I/O performance from these applications, including the need for code restructuring, local disks on every node and overlapping I/O with computation. We also report our experience on achieving high performance on peer-to-peer configurations. Finally, we comment on the necessity of complex I/O interfaces like collective I/O and strided requests to achieve high performance.

1 Introduction

I/O has been identified as a major obstacle to achieving high performance from parallel computers. As a result, significant effort has been put into trying to improve parallel I/O systems. To date, most of the effort has focused on observing the I/O behavior of existing applications and on trying to improve the ability of I/O systems to execute existing applications[2, 3, 4, 6, 8]. We decided to take a different approach; rather than tuning the I/O system to optimize fixed applications, we concentrated on tuning the applications to improve their I/O performance (and hopefully also improve their execution time). Our goal was to find out what strategies were required to achieve good I/O performance for these applications, and to identify common strategies that work for a variety of applications. We were also interested in seeing what support from I/O libraries and file-systems was necessary to achieve good performance.

We conducted our experiments on a parallel computer with an aggressive I/O configuration that has multiple disks on every node and a theoretical peak I/O performance of 640MB/s for 16 nodes. A widely used micro-benchmark indicated the maximum application-level bandwidth to be 400MB/s using the Unix raw disk interface and 270MB/s using the Unix file-system interface. With this configuration, we were able to eliminate under-configured hardware I/O systems, a common cause of I/O bottlenecks, as the source of poor performance.

With increasing I/O demand, it is becoming attractive to spread I/O throughout a parallel system instead of using dedicated I/O and compute nodes. In our experimental configuration, every node has equal amounts of memory and large amounts of local disk; in addition to computation, each node can perform I/O for itself and other nodes. This configuration allows us to investigate the performance of different applications for both peer-to-peer and traditional client-server architectures.

For our study, we selected I/O intensive applications from two areas: sensor data processing (earth science) and out-of-core sparse matrix factorization (scientific computation). The earth science applications are currently in production use at NASA Goddard Space Flight Center and the out-of-core sparse matrix factorization applications have been developed at the University of Maryland with a near-term goal of solving

[1]This research was supported by ARPA under contract No. #DABT63-94-C-0049, Caltech Subcontract #9503, by NASA under contract No. NASA #NAS5-32337, USRA/CESDIS Subcontract #555541 and by NSF under grant #ASC9318183. Hollingsworth was supported in part by a UMCP General Research Board award.

very large submarine structural acoustics problems. I/O is required in these applications for accessing pre-existing data, for intermediate results (i.e., for *out-of-core* processing) and for producing final output.

For each program the objective was simple: make it run as fast as possible and keep track of what was required to achieve this speed. The results of this exercise are encouraging. Foremost, we were able to obtain effective I/O rates of over 100MB/s for three out of four applications. We also observed several common characteristics in the ways in which we achieved high performance from our applications. First, although it appeared that the initial versions of some of the applications would benefit from complex I/O interfaces, such as strided requests, after our tuning, relatively simple I/O primitives proved to be sufficient. Second, local disks on compute nodes were required to achieve good performance for all of the applications. Third, overlapping I/O with computation was essential for all applications studied. Both prefetching initiated by applications and write-behind provided by the operating system were successfully used.

The rest of this paper is divided into six sections. In the next section, we describe our experimental configuration. Section 3 reports on our efforts to characterize our configuration using micro-benchmarks. In Sections 4 and 5 we describe each of our application areas, report the I/O performance, and discuss the steps required to achieve the performance. Section 6 describes the lessons we learned tuning the applications. Finally, Section 7 summarizes our work.

2 Systems background

To provide a context for our experiments, we briefly describe the experimental environment. We first present the configuration of the parallel machine. Subsequently, we describe the two filesystems and the I/O library available on the machine.

All experiments were run on an IBM SP-2 running the AIX 3.2.5 operating system. The SP-2 consists of RS/6000 workstations connected together by a high performance switch. Each node is connected to the switch by a 40MB/s bi-directional link. The machine consists of 16 *thin* nodes. Each node is identically configured and contains one Power2 processor and 64MB of main memory. Each node also has an aggressive I/O subsystem consisting of two fast-wide SCSI buses each with three 2.2GB IBM Starfire 7200 SCSI disks. Each disk has a rated minimum bandwidth of 8MB/s, and each SCSI bus has a maximum bandwidth of 20MB/s. The overall system has 96 disks totaling over 200GB, and a theoretical maximum aggregate disk bandwidth of 640MB/s.

To explain our observed performance, it is helpful to understand the architecture of individual nodes. Figure 1 shows the node configuration. The processor is connected to its main memory by a 500MB/s memory bus. All I/O, including communication over the high performance switch, uses the 80MB/s MCA (micro-channel) bus. I/O destined for remote nodes must travel over the MCA bus at least twice, once from the SCSI adaptor to local memory and once from local memory to the High Performance Switch adaptor.

The default filesystem provided with AIX is the journaled file system (JFS). JFS is similar to most traditional UNIX file-systems, but includes a finite log to record transaction information to ensure file system integrity in case the system crashes before information has been written to disk. Each disk contains one filesystem[2].

Parallel I/O File System (PIOFS) is the native parallel filesystem on the SP-2, and is implemented as kernel extensions to AIX. It exports a Unix-like interface, with additional calls to create and manage striped files. It has two modes of operations, a **cautious** mode that serializes concurrent updates and a **reckless** mode that allows multiple processes to simultaneously update the same file. In addition, PIOFS allows multiple processes opening a file to create customized views; this facility can be used to partition the file. Files are striped across a user-defined number of the nodes in a round-robin fashion. If the striping unit is larger than 64KB, PIOFS stripes it across disks within individual nodes in 64KB blocks. It uses JFS to access disks, and UDP to communicate between client and server processes. On our platform, PIOFS uses the High Performance Switch for communication and has been configured on two disks (one per SCSI bus) on each node.

Jovian-2 is a multi-threaded parallel I/O library developed at the University of Maryland. It provides an interface similar to the POSIX `lio_listio()` interface, which allows multiple I/O requests to be issued

[2]AIX 4.1 is able to stripe filesystems across multiple disks in a single node. It has only recently become available for the SP-2 and has not yet been installed on this machine.

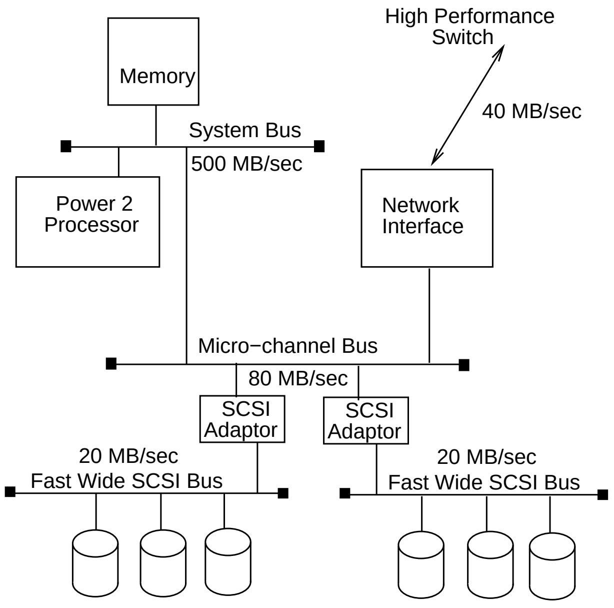


Figure 1: Organization of an SP-2 node.

with a single call. Jovian-2 consists of two parts; the client proxy, which runs in the same thread as the application, and a separate server thread. The server thread can serve requests from both local and remote processes; the local requests are handled as a special-case for fast processing. Jovian-2 is able to take advantage of multi-disk configurations. It allows the application process running on each node to control the scheduling of the associated server thread. Our current implementation assumes that a standard Unix file system with asynchronous I/O calls is available on individual nodes. On the SP-2, our implementation uses the high-speed user-space communication primitives provided by IBM's Message Passing Library (MPL).

3 Micro-benchmarks

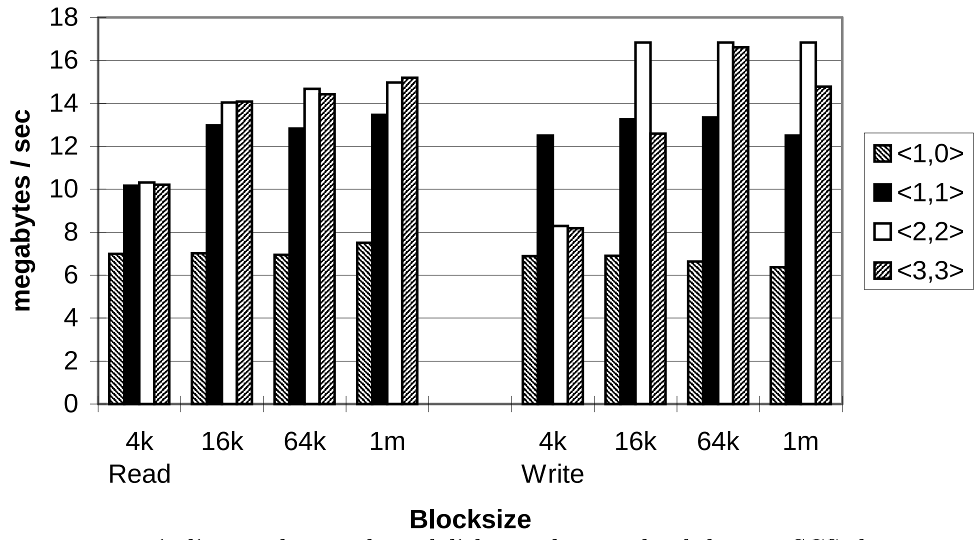
Before trying to tune applications, we conducted a series of tests to determine the maximum application-level bandwidths provided by our experimental configuration. These bandwidths provide a more realistic baseline to evaluate the performance of applications than theoretical disk transfer rates. We varied several parameters including request size, number of disks, and number of pending I/O requests. The results of these tests provided two pieces of information. First, we measured the maximum bandwidth that application processes could hope to get from the I/O system. Second, we determined the parameters and configurations that provide the best performance. This information was used to tune the applications to effectively use the I/O system.

3.1 Journaled File System

To measure single node JFS performance, we used a modified version of the widely used `iozone` benchmark[3]. `iozone` determines the maximum application level I/O bandwidth by making a sequence of contiguous write requests followed by a sequence of contiguous read requests. Our version of this program supports multi-disk configurations and can generate multiple simultaneous requests per disk. It issues all requests in using a single `lio_listio()` call and waits for all of them to complete before issuing the next set.

We performed experiments for request sizes between 4KB and 4MB; with one to six outstanding requests per disk and six different disk configurations. For each experiment, the benchmark wrote and read back a

[3] `iozone` is available on many ftp archives, such as *ftp://ftp.cs.umn.edu/packages/FreeBSD/2.0.5-RELEASE/ports/utils/iozone*.



$< x, y >$ indicates the number of disks used on each of the two SCSI buses.

Figure 2: Maximum application-level I/O rates for JFS

70MB file. A 70MB file was used to ensure we measured disk activity and not file cache performance. Along the request size dimension, the bandwidth curves saturate around 1MB requests. Increasing the number of outstanding requests did not provide much benefit. In most cases, the best performance was achieved with just one or two outstanding requests. Of the various configurations tested, the configuration with two disks per SCSI bus provided the best performance in almost all cases.

Figure 2 presents the maximum application-level read and write bandwidths for a set of request sizes on four disk configurations. We repeated these experiments using the Unix raw disk interface instead of JFS. The maximum read bandwidth achieved was 25.2MB/s and the maximum write bandwidth was 23.5MB/s. Although the raw disk configuration provided noticeably higher throughput, we decided that any potential benefit from using raw disks would be offset by the loss in functionality of not having a UNIX filesystem.

3.2 Jovian-2

To test the performance of the `Jovian-2` parallel I/O library for various I/O configurations, we performed experiments for three kinds of configurations (1) local access, (2) client-server and (3) peer-to-peer. Request size was scaled to take advantage of disk parallelism – for a configuration that used d disks, a request of $stripe\_size \times d$ KB was used. For the local access and the one-client-one-server cases, a single processor wrote and read back a 70MB file. A 70MB file was used to ensure we measured disk activity and not file cache performance. In addition, we read a 70MB file on every node between every write experiment and the subsequent read experiment. For larger configurations, the file size was scaled to ensure at least 70MB of data per node. For configurations with multiple clients (or peers), non-overlapping requests were generated.

Results for a representative subset of the experiments are presented in Table 1. For non-local requests, Jovian-2 reads files from disk using JFS and delivers them to the requesting application using MPL. To provide an upper bound for the performance of the library, we computed the maximum communication bandwidth for 128KB messages to be 32MB/s[4]. Combining this maximum communication bandwidth with the maximum application-level I/O bandwidth via JFS (from Table 2), we arrive at 10.3MB/s as an upper bound for non-local bandwidth. In comparison, Jovian-2 achieved a read bandwidth of 9.3MB/s for a one server, one client configuration. For configurations with relatively large I/O bandwidth – more servers than clients and large peer-to-peer systems, the read and write bandwidths are comparable. For configurations with relatively small I/O bandwidth, write bandwidth is much higher than read bandwidth, by taking advantage of write-behind. For peer-to-peer configurations using n nodes, $1/n$ of I/O was local and the rest

[4] This computation was done by measuring the time required to send one million 128KB messages between a pair of nodes.

Four-way striping on each node (2 disks per SCSI bus), 128KB stripes

Configuration	local	c1-s1	c1-s2	c1-s4	c1-s8	c3-s1	c7-s1	c6-s2	c5-s3	c4-s4
Read bandwidth	14.3	9.3	17.5	21.2	22.8	8.5	8.9	16.6	23.3	25.9
Write bandwidth	16.7	8.7	16.3	20.6	21.6	9.66	19.8	24.8	30.0	27.9

Configuration	pp-2	pp-4	pp-8
Read bandwidth	13.8	16.5	26.0
Write bandwidth	21.2	26.2	33.2

Two-way striping on each node (1 disk per SCSI bus), 64KB stripes

Configuration	local	c1-s1	c1-s2	c1-s4	c1-s8	c3-s1	c7-s1	c6-s2	c5-s3	c4-s4
Read bandwidth	8.4	6.7	11.0	14.9	19.8	6.6	5.5	9.7	12.1	17.1
Write bandwidth	13.0	7.8	14.1	19.7	20.1	4.1	5.3	13.5	19.7	23.0

Configuration	pp-2	pp-4	pp-8
Read bandwidth	8.5	10.9	16.5
Write bandwidth	18.6	20.7	21.7

All bandwidths are aggregate and are in MB/s. cx-sy corresponds to a client-server configuration with x clients and y servers; pp-x corresponds to a peer-to-peer configuration with x peers. For comparison, the maximum JFS bandwidths for four disks per node are 15.0MB/s (read) and 16.8MB/s (write) and for two disks per node are 13.5MB/s (read) and 13.4MB/s (write). The two disk, 64 KB stripe size experiments were done to allow a comparison with PIOFS.

Table 1: Micro-benchmark performance for Jovian-2

remote.

3.3 PIOFS

The experiments performed for PIOFS were similar to those described in the previous subsection. The node-level striping unit was set to 128 KB. Since PIOFS has been configured on two disks per node, this automatically gets striped over them in 64 KB blocks. The file size and request size were scaled as described above. A single PIOFS file was written and read back in *reckless* mode which does not enforce sequential atomicity and provides higher performance. All configurations with multiple clients (or peers) issued non-overlapping requests.

A comparison of the bandwidths achieved using PIOFS for various configurations with the bandwidths achieved using Jovian-2 on two disks indicates that Jovian-2 performs better for local accesses and client-server configurations (except for the skewed 7-client-1-server case). This is as expected because Jovian-2 special-cases local accesses and uses a faster communication medium. Comparison of performance on peer-to-peer configurations yields mixed results. Jovian-2 achieves better write bandwidths whereas PIOFS achieves better read bandwidths, in particular for the larger configurations. We are as yet unable to explain this behavior.

For our study, we use Jovian-2 for the earth science applications; the sparse matrix factorization applications directly use JFS and manage their own I/O.

4 Earth Science Applications

The two earth science programs, `pathfinder` and `climate`, included in our application suite constitute a processing chain for Advanced Very High Resolution Radiometer (AVHRR) satellite data. Pathfinder AVHRR

Configuration	local	c1-s1	c1-s2	c1-s4	c1-s8	c1-12	c3-s1	c7-s1	c6-s2	c5-s3	c4-s4
Read bandwidth	4.7	4.6	7.2	9.6	10.9	11.2	4.1	4.1	9.0	11.5	16.0
Write bandwidth	5.7	6.1	7.4	9.8	10.7	11.4	7.0	7.1	13.2	19.0	14.0

Configuration	pp-2	pp-4	pp-8
Read bandwidth	6.4	15.8	22.4
Write bandwidth	10.6	11.4	17.6

All bandwidths are aggregate and are in MB/s. cx-sy corresponds to a client-server configuration with x clients and y servers; pp-x corresponds to a peer-to-peer configuration with x peers. For comparison, the maximum JFS bandwidths for for two disks per node are 13.5MB/s (read) and 13.4MB/s (write).

Table 2: Micro-benchmark performance for PIOFS.

data sets are global, multichannel data from NOAA meteorological satellites. `Pathfinder` and `climate` are currently in production use by NASA's Goddard Distributed Active Archive Center and, together, are representative of a large class of NASA earth science applications. Furthermore, the structure of these applications is similar to the large set of programs currently being developed to process data from the Earth Observation System satellites.

Pathfinder: This program processes AVHRR global area coverage (4.4 km resolution) data. The input to the program consists of a set of files each containing data corresponding to a little more than one orbit. There are fourteen files per daily data set, which totals about 680MB. `Pathfinder` performs calibration for instrument drift, topographic correction, masking of cloudy pixels, registration of sensor readings with locations on the ground and compositing of multiple pixels corresponding to the same ground location. In addition to satellite sensor data, it reads several fixed size ancillary files which add up to about 100MB. The output is a twelve band 5004x2168 Interrupted Goode Homolosine image of the world at 8km resolution. The size of the output image is about 228MB.

The sensor data is read in scan line by scan line, in chunks of 128 lines, each line consisting of 3584 bytes. Ancillary files are not loaded in memory. Instead, the data from these files is read when needed. For composition, `Pathfinder` maintains an out-of-core intermediate version of the image. This image is repeatedly referenced since `pathfinder` accumulates multiple sensor data values into each pixel of the intermediate image. After each chunk is processed, each data value is mapped into the intermediate image and is compared with the corresponding pixel. If the new value is "better", it is copied into the pixel. In effect, the accumulation phase of `pathfinder` implements an out-of-core max-reduction. Once all input data has been processed, the final image is produced by rewriting the partitioned image from local disks and local memories into persistent storage. The total I/O performed by an optimized sequential version, including I/O for out-of-core accesses, is over 28GB.

Although processing of AVHRR global area coverage data is representative of many earth science applications, some similar programs process even more data. For example, the input volume is sixteen times higher for AVHRR local area coverage data (1.1 km resolution); the corresponding output size for the global 1km data products is sixty-four times larger. For MODIS, the primary instrument of the Earth Observation System, both the input and the output are at least two orders of magnitude larger than AVHRR global data.

Climate: This program produces the AVHRR Land Climate data product. It processes the 8km resolution global image generated by `pathfinder` into a vegetation index on a latitude-longitude grid. Vegetation index maps are used to track global land cover change. Input to `climate` is the twelve-band 228MB file generated by `pathfinder`, of which the program uses three bands totalling 54MB. In addition, `climate` reads 21.5MB from an ancillary file. The total I/O volume for `climate` is 75.5MB.

I/O Optimizations for the sequential version:

- In both programs, input was being read one scan line at a time (3.5KB for `pathfinder` and 10KB for `climate`). We aggregated input reads to 512KB in both cases.
- A recurrent I/O pattern in both programs was the embedding of small I/O requests in the innermost

loops. Each such occurrence generated nested sequences of small requests with fixed strides. This occurs in three situations: (1) reading of topographic data, (2) reading of the land-sea mask and (3) reading and writing of data for the out-of-core max-reduction. The request size was almost always two bytes and the subsequent seek distance was 20MB. Relatively straightforward loop restructuring transformations were sufficient to aggregate the I/O and move it to the outermost loop. In the first two cases, the I/O was converted to block reads, whereas for the out-of-core max-reduction, it was converted to block read-modify-writes.

- All I/O in both programs was buffered, using `fread/fwrite` with default buffer size (4KB). In most cases, including the pattern described above, buffering was inappropriate because individual requests were small (two bytes) and the distance between successive accesses (20MB) was larger than practical buffer sizes. After the aggregation optimization mentioned above, the size of the buffer required was variable. We replaced the buffered I/O calls by their unbuffered analogues.

Parallelization:

The computations in `climate` map a twelve band 5004x2168 Interrupted Goode Homolosine image onto a 180 by 360 degree latitude/longitude world map. The mapping between input and output does not vary with the input data. The output data set is partitioned between processors. Input data is sent to the processor responsible for the corresponding partition of the output and appropriate calculations are carried out. Since the relationship between the input and output datasets is known a-priori, we were able to partition the output dataset to optimize load balance.

In `pathfinder`, the map between the input and the output datasets is data-dependent. A `pathfinder` input dataset consists of orbital sensor data, the output dataset is an Interrupted Goode Homolosine image used by `climate`. In `pathfinder`, the output dataset is partitioned between processors using equal-sized strips. Each processor subsamples the input dataset to determine which portion of the input data is to be read by individual processor. For a variety of reasons, the load balance is data dependent. `Pathfinder` only processes a subset of of its input, e.g. night-time and ocean data is not processed.

Both `pathfinder` and `climate` accumulate multiple input values into each element of the output dataset. In `pathfinder`, the local portions of the output image are too large to be stored in main memory, so successive updates to the output image are formulated as out-of-core computation. In both programs, once the updates are completed, the final result is produced by concatenating the individual partitions.

Use of Parallel I/O:

We replaced the Unix I/O calls by calls to Jovian-2. All ancillary files were striped across four local disks. For `pathfinder`, the intermediate image file used for the out-of-core max-reduction was also striped across four local disks. The input and output files were striped across all the server disks (four per server). Stripe size in all cases was 128KB.

4.1 Results and Analysis

Experiments: We ran `pathfinder` and `climate` for a daily data set on a variety of configurations. An unmodified version of `pathfinder` ran for 18,800 seconds on a single processor of the SP-2. Of this, about 13,600 seconds (76%) of the time was spent waiting for I/O; 580 seconds for input, 50 seconds for output and the remaining 12,970 seconds of I/O for the out-of-core max-reduction.

Table 3 shows the breakdown of total I/O volume for the parallel version of `pathfinder`. The volume changes with configuration for two reasons. First, every processor subsamples all files to determine the map from the input and output coordinates; this leads to an increase in the amount of input data read as the number of processors grows. This growth can be avoided by partitioning task of determining the map between input and output coordinates and having each processor report its share of the map to all other processors. Second, the size of the block to read during the reduction phase is determined by the bounding box around the pixels that are to be updated. Since the pixels to be updated are sparsely distributed, finer partitions of the intermediate image file are able to eliminate holes, reducing the total volume of I/O for this phase. The volume of intermediate reads is consistently much higher than the volume of intermediate writes. This is because some on-disk data corresponding to the entire chunk of 128 scan lines being processed is needed for the max-reduction. Writes happen only if at least one of the current on-disk pixel values need to be updated, and then only for the bounding box around the pixels to be updated.

Config (client/server)	3/1	7/1	6/2	5/3	11/1	10/2	9/3	15/1	14/2	13/3
Input	1,508	3,100	2,700	1,838	4,751	4,334	3,924	6,350	5,952	5,561
Intermediate read	20,341	12,306	13,143	11,261	10,507	10,699	11,094	9,275	9,600	9,893
Intermediate write	6,301	4,493	4,701	3,697	3,973	4,039	4,130	3,683	3,871	3,809
Total	28,378	20,126	20,771	17,024	19,459	19,300	19,376	19,536	19,651	19,492

All volumes are in megabytes. Output volume is 228MB for all configurations.

Table 3: Breakdown of total I/O volume for `pathfinder`

The breakdown of total I/O volume for `climate` does not change with configuration. The I/O for `climate` consists almost exclusively of read requests – total local I/O (for ancillary files) is about 21.5MB and total non-local I/O (input data) is 54MB. The output volume is 130KB.

Early results indicated that there was a large difference between the performance of peer-to-peer and client-server configurations for programs that have substantial computation. With abbreviated input (one orbit file instead of fourteen) `pathfinder` running on a four processor peer-to-peer configuration was able to achieve only a 400KB/s per processor non-local read bandwidth. With the same input, it was able to achieve a per processor non-local read bandwidth of 6MB/s on a client-server configuration of the same size (three clients, one server). The write bandwidths on peer-to-peer configurations were better at 3MB/s but were much lower than the 7.4MB/s achieved on a client-server configuration of same size. A point to note here is that non-local reads are used to read the satellite data in chunks of 128 scan lines and are interspersed with computation, whereas non-local writes are used for final output and are grouped. The total execution time for an abbreviated `pathfinder` run (single orbit file) on a four processor peer-to-peer configuration was 510 seconds, of which 320 seconds was I/O waiting time. In comparison, the execution time on a three-client-one-server configuration was 290 seconds, of which 103 seconds was I/O waiting time. In contrast, the Jovian-2 micro-benchmark, which does no computation, achieved comparable performance on both peer-to-peer and client-server configurations (see Table 1). This might seem to indicate that only programs that do nothing but I/O are able to achieve good I/O bandwidths on peer-to-peer configurations; however, Section 5 provides a counter-example. It provides performance results for a program that performs substantial computation as well as substantial I/O on a peer-to-peer configuration. It achieves good performance by using extensive information about future I/O requirements and a sender-push I/O model. We shall revisit the issue of I/O performance on peer-to-peer configurations later in this paper (Sections 5 and 6). For the rest of the experiments with `pathfinder` and `climate`, we restricted ourselves to client-server configurations.

Figure 3 shows a breakdown of execution time for `pathfinder` for a set of client-server configurations. It provides three points of interest. First, `pathfinder` is now compute-bound. Except for the 15-client-1-server case, I/O waiting time is less than 25% of the total time. In many cases, it is substantially less (10% in one case). Second, a small number of servers is optimal. Increasing the number of servers beyond two for any of the configurations provided no benefit and actually increased the execution time for the twelve and sixteen processor configurations. Third, the execution time does not reduce significantly from the twelve processor to the sixteen processor configuration. There are two reasons for this. First, since all processors do the partitioning independently, the amount of input data read during the partitioning phase increases with the number of processors. Second, as the number of processors grows, each chunk (128 input scan lines) is partitioned between more processors. Each processor that processes a part of a chunk has to unpack, parse and map the entire chunk before it is able to isolate its portion. Therefore, the total amount of processing done on every chunk grows roughly with the number of processors. As was mentioned earlier, growth in the amount of total input read volume can be avoided by partitioning the task of determining the map between input and output coordinates, followed by a global exchange of information. Growth in the total amount of computation done per chunk can be avoided by using a pre-computed map from input scan lines to the

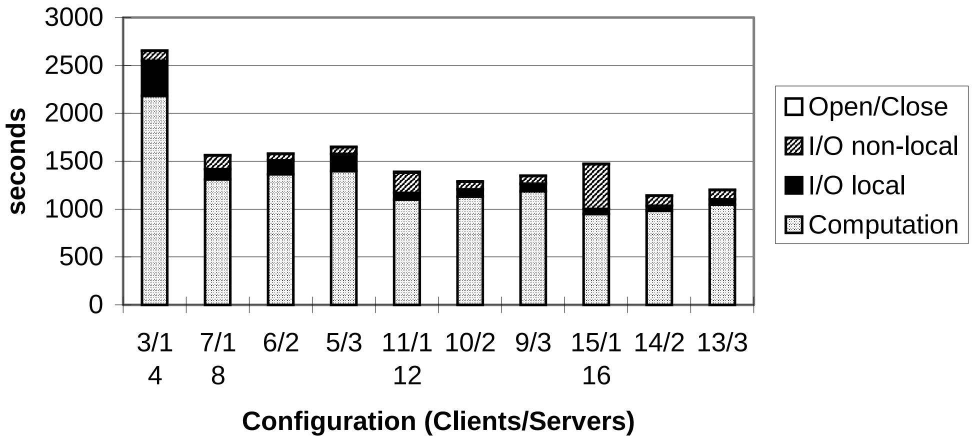


Figure 3: Breakdown of execution time for `pathfinder`

output image. This woukd require restructuring of code to process variable-size chunks. The map from input scan lines to the output image can either be computed during the preprocessing phase of `pathfinder` or can be computed as a part of the processing from raw readings (level 0 in NASA parlance) to the orbit data file (level 1b in NASA parlance) which occurs prior to `pathfinder`.

Table 4 presents the aggregate bandwidths sustained by `pathfinder` for different kinds of I/O. Recall that both input and output I/O are non-local, whereas both intermediate reads and writes are to local disks. It is interesting to note that for configurations with many clients and few servers, the aggregate I/O rate achieved is greater than the value indicated by the micro-benchmark results presented in Table 1. This is made possible by the fact that, beyond an initial barrier for configuration purposes, all client processes are independent. This allows different clients to utilize the server(s) at different times. In contrast, the Jovian-2 micro-benchmark does nothing but I/O and all the clients are continuously making requests. Another point of interest is that the intermediate requests have substantial locality and are able to take good advantage of the operating system file cache for both reuse and write-behind. This is facilitated by the parallelization scheme, which ensures that each processor gets only the data that maps into its segment of the output image.

Config (client/server)	3/1	7/1	6/2	5/3	11/1	10/2	9/3	15/1	14/2	13/3
Input	18.0	25.4	45.6	38.5	26.4	60.0	60.3	15.0	65.8	68.9
Output	22.2	34.3	55.8	40.0	41.8	108.0	94.5	63.0	126.0	135.2
Intermed read	85.8	246.4	192.6	151.5	510.4	453.0	388.8	634.5	644.0	600.6
Intermed write	65.7	169.4	147.2	118.0	278.3	256.0	230.4	406.5	372.4	338.0
Overall	64.8	96.2	122.1	103.5	85.0	163.0	162.5	43.8	161.6	170.0

All rates are in MB/s. Aggregate I/O rate is computed by multiplying the per processor sustained I/O rate by the number of clients. Per processor sustained I/O rate is computed as the sum of I/O volumes for all clients divided by the sum of time spent in I/O routines by all clients.

Table 4: Aggregate I/O rates for `pathfinder`

Figure 4 shows the breakdown of execution time for `climate`. Both computation and I/O for `climate` scale quite well. The total I/O time was consistently about 4-5% of the total computation time for all the configurations we tested it on. We did not run `climate` on larger configurations, since the individual partitions of the I/O data would become very small. Recall that the total I/O volume for `climate` is 75.5MB. Table 5 presents the aggregate I/O rates for `climate`. Similar to the results for `pathfinder`, asynchronous requests allow `climate` to achieve a larger aggregate bandwidth than was indicated by the Jovian-2 micro-benchmark.

Table 6 presents end-to-end I/O rates for both `pathfinder` and `climate`. It shows that both programs,

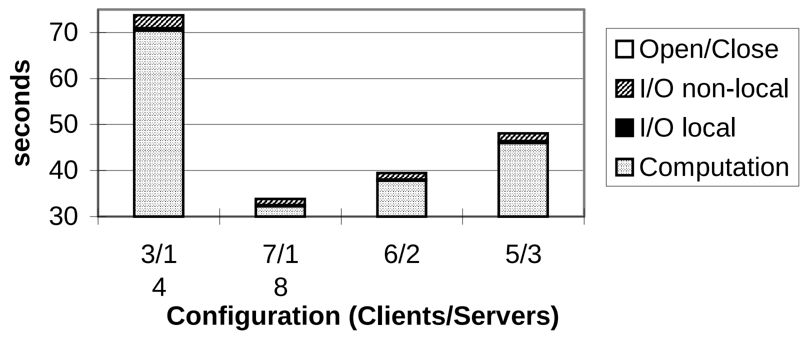


Figure 4: Breakdown of execution time for `climate`

which are now compute-bound, achieve an average I/O rate of over 26MB/s over their entire execution time.

Config (client/server)	3/1	7/1	6/2	5/3
Local I/O	35.1	77.0	64.2	56.5
Non-local I/O	17.7	25.2	34.2	31.5
Overall	19.2	27.8	36.0	32.7

Table 5: Aggregate I/O rates for `climate`, in MB/s: Aggregate I/O rate is computed by multiplying the per processor sustained I/O rate by the number of clients. Per processor sustained I/O rate is computed as the sum of I/O volumes for all clients divided by the sum of time spent in I/O routines by all clients.

Config (client/server)	3/1	7/1	6/2	5/3	11/1	10/2	9/3	15/1	14/2	13/3
pathfinder	11.8	17.9	17.5	16.2	20.8	22.7	21.7	18.8	26.5	25.5
climate	1.15	2.53	2.29	1.92	–	–	–	–	–	–

Table 6: End-to-end I/O rates for `pathfinder` and `climate`. All rates are in MB/s. The rate is computed by dividing the total data volume by the total execution time.

5 Out-of-core Sparse Matrix Factorization

Many scientific and engineering applications require the solution of very large sparse linear systems. The largest sparse system (with 10% sparsity and double precision complex arithmetic) that can be solved on current supercomputers consists of about 250,000 equations (assuming a total memory pool of 50GB). The demands of some applications are far beyond that limit. In particular, submarine structural acoustics problems can require the solution of sparse linear systems with 2-3 million equations. Such applications require efficient out-of-core methods. We have implemented an out-of-core parallel sparse Cholesky factorization, along with an associated pair of programs for parallel symbolic factorization and parallel matrix partitioning. We selected Cholesky factorization (`factor`) and the sparse matrix partitioner (`partitioner`) for our study.

Sparse Cholesky factorization arises in the direct solution of symmetric positive-definite systems of linear equations. The Cholesky factor of a symmetric positive definite matrix A is a lower triangular matrix L with positive diagonal, such that $A = LL^T$. Our parallel out-of-core sparse Cholesky factorization is a parallelization of a left-looking supernodal Cholesky factorization algorithm [10]. This particular formulation of Cholesky factorization enables the use of efficient dense linear algebra kernels [5], as well as large transfers between secondary storage and primary memory.

```
1 for i = 1 to S do
2   for all $S_j$ with $j < i$ and $S_{ij} \neq \emptyset$
3     Read $S_j$
4     Update $S_i$ with $S_j$
5     Discard $S_j$
6   Factorize $S_i$
7   Write $S_i$ to disk
```

Figure 5: Out-of-core Sparse Cholesky Factorization

The sparse matrix partitioner performs two functions. First, it computes and allocates space for *fill-ins*, meaning locations in A that are originally zero, but will become nonzero after the factorization. Second, it distributes the the sparse matrix (both its structure and data) to the local disks of the processors. `Partitioner` has two input files. The initial matrix file contains the matrix structure and the nonzero locations of A. The index file contains the sparsity structure of L (which is obtained from symbolic factorization). The index file is distributed among the processors, with each processor having a block of columns.

We use a 2-D partitioning strategy, originally developed in [14]. The processors are organized in a $k \times m$ grid. Let $P_{i,j}$ denote the processor number at the ith row and jth column of the processor grid. Supernode i of matrix A is mapped to processors in the $(i \ mod \ m)^{th}$ column of the processor grid. A supernode is further partitioned within the processors in a column of the processor grid, such that block j of supernode i is mapped to processor $P_{j \ mod \ k, i \ mod \ m}$. This mapping ensures that communication takes place only within the processors placed in the same column or in the same row of the processor grid. Hence, each processor communicates with at most $k + m$ other processors.

The key observation we make for `factor` is that the sparsity structure of the matrix is statically determined and does not change over the computation. Therefore the exact sequence of supernode update operations is known in advance before the computation starts. This information can be exploited when performing step 3 of algorithm (Figure 5) in two ways :**(1) Prefetching:** to overlap the I/O of S_j with the ongoing-computation, and **(2) Caching:** to avoid the re-reading of supernodes to be used in the near future.

Parallelism in `factor` is achieved at several levels both in computation and I/O. First, since each supernode is partitioned among k processors, updates to a supernode are performed in parallel. Second, multiple supernodes can be updated in parallel, as long as the dependences are satisfied. That is, supernode S_j can update supernode S_i only after the factorization (step 6) is performed on S_j. In our implementation, at most m supernodes are updated in parallel, where m is the horizontal dimension used in the processor grid. Third, each supernode is striped across k processors. The stripe size depends on the sparsity of the supernode and is determined by `partitioner`. Fourth, asynchronous I/O primitives are used to overlap the computation with I/O. The prefetch mechanism uses a pre-computed schedule to issue as many asynchronous I/O requests as are allowed by the available memory, *in the order* that they will be used. We have not attempted to improve the communication balance for the `factor`. Our relatively simple technique provided acceptable performance for moderately unstructured matrices but did not perform well on *sara-2*, which is very sparse.

`Partitioner` has two phases with similar I/O access patterns. The first phase sequentially reads the index file to extract the supernodal structure of the matrix. The input requests consist of large sequential chunks (25MB) from a single file, using blocking I/O calls. Almost all the I/O is local in the first phase, except the columns on the processor borders. I/O in the second phase also uses large sequential requests, but uses two local files and a smaller request size (5MB). This incurs a few extra disk seeks as compared to the first phase.

5.1 Results and Analysis

Table 7 presents the characteristics of the input matrices. *Skirt* and *sara-1* have similar characteristics, except that the latter needs twice as much space to store a nonzero. *Sara-2* has a low number of unknowns per element, which results in thin supernodes in contrast to the relatively large supernodes of the first two matrices. This means that the integer index storage per nonzero for the sparse representation is much

Matrix	N	$\lvert A \rvert$	$\lvert L \rvert$	Type	Description
skirt	45,361	1.3M	45.8M	Real	NASA
sara-1	44,856	2.6M	30.4M	Complex	Structural Acoustics
sara-2	80,651	2.9M	28.4M	Complex	Structural Acoustics

Table 7: Characteristics of input matrices. N is the number of equations. $|A|$ denotes the number of nonzeros in the input matrix, and $|L|$ is the number of nonzeros in the Cholesky factor.

	Partitioner		Factorization	
Matrix	Read	Write	Read	Write
skirt	381	403	20,200	377
sara-1	488	534	49,000	509
sara-2	301	1,939	220,800	838

Table 8: Application I/O volumes, in MB, for 16 processors.

higher in sara-2 than for the first two matrices. Table 8 shows the total amount of I/O performed by both applications for the test matrices. `Partitioner` output volume and, consequently, `factor` input volume increases with the number of processors, so the table shows 16 processor data. Again we see that sara-2 is different from the other two input data sets, with the input to `partitioner` being smaller, but the matrix structure produces both a large output set from `partitioner`, and correspondingly greater data read by `factor`.

For all the experiments in this section, we have not used local disk striping; each node uses a single local disk. A key data structure in `factor` is the *elimination tree* [9] generated by the symbolic factorization using the structure of the sparse matrix. This structure contains dependency information between different supernodes and allows all processors to generate a schedule for their I/O and communication. Each processor issues prefetch I/O requests based on the schedule generated from the elimination tree and availability of memory space. A static prefetch horizon of two dependency levels per supernode is used, generating at most $2m$ outstanding read requests on each processor. The prefetch horizon was determined empirically and depends on the per-processor physical memory available for user programs as well as the relative I/O and computation rates. I/O accesses in `factor` can be both local and non-local; step 3 of Figure 5 requires non-local I/O when S_j is not stored on local disk. Data prefetched on behalf of other processors is injected into the communication network when the local computation reaches an appropriate point in the schedule. Table 9 shows the *effective* I/O rates seen by `factor`, computed by dividing the total volume of I/O by the time spent waiting for the prefetch requests. By successfully utilizing the available data access information, we were able to hide much of the I/O latency in `factor`.

Table 10 presents the sustained I/O rates achieved by `partitioner`. The index file is read both in phase 1 and phase 2, producing caching effects in phase 2 for smaller inputs on the largest processor configurations. In all the cases, phase 2 has benefited from the file cache, and in the largest configuration, where the per processor data size is smallest, the file cache successfully avoided physical I/O. `Partitioner` performs a large number of unbuffered, small-size writes, which results in the relatively poor performance of writes, as compared to reads.

The execution time breakdowns for `partitioner` and `factor` are shown in Figures 12 and 11, respectively. The breakdown for `factor` shows that the computation scales linearly with the number of processors, and

	4 nodes		8 nodes		16 nodes	
Matrix	Read	Write	Read	Write	Read	Write
skirt	47.4	83.6	87.6	172.1	121.6	242.7
sara-1	76.3	101.5	96.2	199.7	157.3	277.3
sara-2	-	-	-	-	96.1	182.7

Table 9: Aggregate sustained I/O rates for `factor`, in MB/s.

Matrix	4 nodes			8 nodes			16 nodes		
	Read-1	Read-2	Write	Read-1	Read-2	Write	Read-1	Read-2	Write
skirt	14.1	19.4	9.9	43.0	45.8	19.7	108.8	380.9	41.1
sara-1	12.3	15.9	9.3	18.6	29.5	18.7	106.1	430.7	15.1
sara-2	20.4	49.6	2.5	42.1	76.4	16.4	103.2	297.8	15.5

Table 10: Aggregate sustained I/O rates for `partitioner`, in MB/s.

Matrix	4 nodes				8 nodes				16 nodes			
	Total	I/O	Comp	Comm	Total	I/O	Comp	Comm	Total	I/O	Comp	Comm
skirt	5,652	436	3,096	2103	3,351	278	1,591	1,459	1,800	204	798	778
sara-1	13,062	674	8,790	3,502	8,643	548	4,443	3,597	5,074	384	2,263	2,354
sara-2	-	-	-	-	-	-	-	-	11,023	2,321	3,764	5,300

Table 11: Execution time breakdown for `factor`, in seconds

that the I/O also scales reasonably well, in all cases consuming a fairly small percentage of the total program time. However, communication does not scale well for all input data sets for `factor`. Note that the sara-2 input data set was only run on 16 processors, because the program runs for too long on fewer processors. Even for this very sparse data set, the I/O takes only about 20% of the total execution time. The breakdown for `partitioner` shows that I/O is a small fraction of the total execution time, and that the I/O scales well with increasing numbers of processors. In fact, the I/O scales better than the computation. For these input sets, the I/O always consumed less than 25% of the total program execution time, and often significantly less. From this data, we can conclude that the I/O optimizations applied for these programs have ensured that I/O is not a bottleneck in performance.

As was mentioned in Section 4, results from our earth science applications and micro-benchmarks suggests that programs that do substantial computation as well as substantial I/O would not be able to achieve good I/O bandwidths on peer-to-peer configurations. However, `factor` provides a counter-example[5]. It uses extensive information about future I/O requirements to schedule I/O requests, and a sender-push model to transfer the data to the destination. Since all peers in the configuration have the schedule, they can determine, at each stage, who they should be receiving data from.

6 Lessons Learned

This section presents the lessons we learned from the studies presented in this paper. We present our experiences as well as guidelines for obtaining high I/O performance for I/O intensive applications.

Code restructuring is important:

- For the applications we have studied, it is not difficult to restructure the code to coalesce small requests into much larger ones. We believe that many I/O intensive applications are currently not designed to generate large I/O requests, but relatively little effort is required to modify them to do so. In other

[5]All experiments on `factor` and `partitioner` were run on peer-to-peer configurations.

Matrix	4 nodes		8 nodes		16 nodes	
	Total	I/O	Total	I/O	Total	I/O
skirt	309.6	77.6	222.4	38.4	162.0	19.3
sara-1	522.3	79.2	401.7	40.1	319.8	24.3
sara-2	2,568.4	541.7	2,101.2	319.2	1,568.4	177.9

Table 12: Execution time breakdown for `partitioner`, in seconds

words, the problem is not that large I/O requests cannot be generated, but that programmers have not considered the problem of optimizing their applications to take advantage of the performance benefits provided by larger requests.

- For the applications we have studied, information about future requests is available and can be used to prefetch data. For `pathfinder` and `climate`, all processors subsample all input files in the partitioning phase and at the end of this phase, every processor has complete information about its future requests. For `factor` and `partitioner`, the sequence of requests is available from the elimination-tree structure generated by symbolic factorization. Furthermore, three of the applications in this study (all the ones that write a significant volume of data), can be structured to take good advantage of write-behind provided by the operating system file cache.
- For the applications we have studied, it is possible to partition the *out-of-core* intermediate data so that each processor reads and writes to its own local disk(s). As can be expected, and as we have shown in Section 3, bandwidths for local disk access are substantially higher than the bandwidths for non-local accesses. In addition, local accesses are guaranteed not to interfere with I/O requests from other processors. This increases the utility of the file cache and makes the overall behavior of the application becomes more predictable. Exploiting locality in this manner is beneficial for *out-of-core* applications[1, 2, 12] on both client-server and peer-to-peer configurations. In either configuration, exploiting locality improves I/O performance as well as total execution time.

Diskful machines are important:

As shown by the results in Sections 4 and 5, local disks attached to compute nodes can help convert programs that request tens to hundreds of gigabytes of I/O into compute-bound problems. In combination with code restructuring to exploit locality, diskful machines can improve both the I/O performance and the overall execution time for out-of-core applications.

Complex I/O interfaces are not required:

- After code restructuring, most requests in the studied applications were large. For large requests, the interface is usually less important.
- Small strided requests were a recurrent pattern in the original versions of `pathfinder` and `climate`. Nested-strided requests [11] have been proposed for just such patterns. However we found that these patterns were caused by the embedding of small I/O requests in the innermost loops. Relatively straightforward loop restructuring, including interchanging the order of nested loops [15] and fusing multiple requests were sufficient to coalesce these requests into large block I/O requests.
- None of the applications studied required collective I/O [1, 3, 13]. This is not surprising given the size of the requests after code restructuring. All of the applications are parallelized in SPMD fashion. In our earth science applications all processes are independent (apart from initial and possibly final synchronization). Independent I/O requests were able to utilize the servers when they would have been idle in a collective I/O model (see Section 4).

Good performance on peer-to-peer systems is possible:

Our experience with applications that do substantial I/O and computation on peer-to-peer configurations was mixed. On one hand, the performance of `pathfinder` on peer-to-peer configurations was poor; on the other hand, `factor` achieved excellent performance. `Pathfinder` used a general-purpose parallel I/O library which served requests as they arrived and had no information about future I/O requests. In contrast, the I/O module in `factor` had access to extensive information about future requests and was able to control the scheduling of I/O requests. This allowed it to overlap the I/O and communication with the computation.

The problem of achieving good computation performance on processors that are serving data to others has been previously noted by Kotz and Cai [7]. In their experiments on a cluster of RS6000s, they found that serving off-processor I/O requests can slow a relatively simple parallel program by between 17% and 98%. Given the trend towards commodity hardware, we believe peer-to-peer configurations are important for applications that do substantial I/O and computation. We are currently looking at this problem in greater detail.

7 Conclusions

In this paper we have shown that I/O intensive parallel applications can be optimized so that I/O is not the limiting factor in their performance. The results from both micro-benchmarks and complete applications, run on an IBM SP-2 with multiple disks per node, show that we can achieve high I/O rates from the hardware and into the applications. We have been able to convert programs with very large I/O requirements whose performance appears to be limited by the I/O capabilities of the parallel machine into compute-bound programs. Our experience has shown that achieving high I/O performance does not require complex I/O strategies; rather, appropriate restructuring of the applications to use local secondary storage for staging intermediate results and producing relatively small numbers of large I/O requests allows an I/O library or the vendor file-system to provide a high I/O bandwidth to the application. In addition, overlapping the I/O with computation, either in the application or in the operating system, provided large performance benefits. For the applications we have studied, this benefit derives from the out-of-core nature of the algorithms used, which are required because of the extremely large data sets to be processed. However, these out-of-core algorithms did not require complex I/O interfaces to achieve high I/O bandwidths. A relatively simple interface like POSIX `lio_listio()` is adequate as long as the application and I/O system are configured properly.

Acknowledgements

We would like to thank Tonjua Hines and Steve Kempler from the Earth Science Data & Information Systems Project (ESDIS) at NASA Goddard Space Flight Center for many invaluable discussions about NASA's remote sensing data processing requirements, and for helping us gain access to NASA programs. We would also like to thank Peter Smith and Mary James of the Goddard Distributed Active Archive Center (DAAC) for providing the Pathfinder and Climate programs.

References

[1] Robert Bennett, Kelvin Bryant, Alan Sussman, Raja Das, and Joel Saltz. Jovian: A framework for optimizing parallel I/O. In *Proceedings of the 1994 Scalable Parallel Libraries Conference*, pages 10–20. IEEE Computer Society Press, October 1994.

[2] R. Bordawekar, A. Choudhary, K. Kennedy, C. Koelbel, and M. Paleczny. A model and compilation strategy for out-of-core data parallel programs. In *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming (PPOPP)*, pages 1–10. ACM Press, July 1995. ACM SIGPLAN Notices, Vol. 30, No. 8.

[3] Alok Choudhary, Rajesh Bordawekar, Michael Harry, Rakesh Krishnaiyer, Ravi Ponnusamy, Tarvinder Singh, and Rajeev Thakur. PASSION: Parallel and scalable software for input-output. Technical Report SCCS-636, NPAC, September 1994. Also available as CRPC Report CRPC-TR94483.

[4] P. Crandall, R. Aydt, A. Chien, and D. Reed. Input/output characteristics of scalable parallel applications. In *Proceedings Supercomputing'95*, December 1995. To appear.

[5] J.J.Dongarra, J.DuCroz, I.S.Duff, and S.Hammarling. A set of level 3 basic linear algebra subprograms. *ACM Transactions on Mathematical Software*, 16(1):1–17, 1990.

[6] David Kotz. Disk-directed I/O for MIMD multiprocessors. In *Proceedings of the 1994 Symposium on Operating Systems Design and Implementation*, pages 61–74. ACM Press, November 1994.

[7] David Kotz and Ting Cai. Exploring the use of I/O nodes for computation in a MIMD processor. In *Proceedings of the IPPS'95 Third Annual Workshop on Input/Output in Parallel and Distributed Systems (IOPADS)*, pages 78–89, April 1995.

[8] David Kotz and Nils Nieuwejaar. File-system workload on a scientific multiprocessor. *IEEE Parallel & Distributed Technology*, 3(1):51–60, Spring 1995.

[9] Joseph W. H. Liu. The role of elimination trees in sparse factorization. (11):134–172, 1990.

[10] Esmond G. NG and Barry W. Peyton. Block sparse cholesky algorithms on advanced uniprocessor computers. 14(5):1034–1056, September 1993.

[11] Nils Nieuwejaar and David Kotz. Low-level interfaces for high-level parallel I/O. In *Proceedings of the IPPS'95 Third Annual Workshop on Input/Output in Parallel and Distributed Systems (IOPADS)*, pages 47–62, April 1995.

[12] M. Paleczny, K. Kennedy, and C. Koelbel. Compiler support for out-of-core arrays on parallel machines. In *Proceedings of the Fifth Symposium on the Frontiers of Massively Parallel Computation*, pages 110–118. IEEE Computer Society Press, February 1995.

[13] Juan Miguel del Rosario and Alok N. Choudhary. High-performance I/O for massively parallel computers: Problems and prospects. *IEEE Computer*, 27(3):59–68, March 1994.

[14] Edward Rothberg and Anoop Gupta. An efficient block-oriented approach to parallel sparse cholesky factorization. pages 503–512, Portland, OR, November 1993.

[15] Michael Wolfe. *High Performance Compilers for Parallel Computing.* Addison-Wesley, 1995.