Tuning the Performance of I/O-Intensive Parallel Applications *

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
[†]Center for Excellence in Space Data and Information Sciences
Goddard Space Flight Center, Greenbelt MD 20771

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 satellite-data processing and linear algebra. After tuning, three of the four applications achieve application-level I/O rates of over 100 MB/s on 16 processors. The total volume of I/O required by the programs ranged from about 75 MB to over 200 GB. 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 knowledge of future I/O requests. 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 one of the major obstacles to achieving high performance from parallel computers. As a result, significant effort has been put into trying to improve the performance of I/O on these machines. To date, most researchers have focused on observing the I/O behavior of existing applications and on trying to improve the ability of I/O systems available on parallel machines to execute these applications [2, 3, 4, 7, 9]. We take a different approach. Instead of assuming that the applications are fixed and that the I/O system alone is open to modification, we believe that both the applications and the I/O system have to be tuned to achieve good performance. In this paper, we concentrate on tuning the applications to improve their I/O performance, hopefully also improving their execution time. Our goal in this research 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 filesystems was necessary to achieve good performance.

To eliminate an under-configured I/O system, often a cause of I/O bottlenecks, as a cause of poor performance, we conducted our experiments on a parallel machine with an aggressive I/O configuration. Our experimental platform consisted of a 16-processor IBM SP-2 with six fast disks attached to every processor. A widely used micro-benchmark indicated the maximum application-level bandwidth to be 400 MB/s using the Unix raw disk interface and 270 MB/s using the Unix filesystem interface. All processors are connected to a high performance switch by a 40 MB/s bi-directional link. Each node on our platform has substantial resources and can perform I/O for itself and for other nodes. This configuration also allowed us to investigate the performance of applications on both peer-peer and client-server architectures.

For our study, we selected I/O-intensive applications from two areas: satellite-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 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. The results of this exercise are encouraging. Foremost, we were able to obtain application-level I/O rates of over 100 MB/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 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, information about future I/O requests was available for all applications and could be

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

To appear in the Fourth Annual Workshop on I/O in Parallel and Distributed Systems, May 27, 1996, Philadelphia

used to schedule the requests. 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 this performance. Section 6 describes the lessons we learned tuning the applications. Finally, Section 7 summarizes our work.

2 Systems background

All our experiments were performed on a 16-node IBM SP-2 running AIX 3.2.5. Each node is identically configured with one POWER2 processor, 64 MB of main memory, two fast-wide SCSI buses and a 40 MB/s bi-directional link to a multi-stage high-performance switch. Each SCSI bus has three 2.2 GB IBM Starfire 7200 SCSI disks. The peak bandwidth for each disk is 8 MB/s and the peak bandwidth for each SCSI bus is 20 MB/s. The overall system has 96 disks totaling over 200 GB, a peak aggregate disk bandwidth of 640 MB/s and a peak aggregate SCSI bandwidth of 480 MB/s. The SCSI buses and the network adaptor are connected to an 80 MB/s MicroChannel bus.

Each disk contains a separate filesystem. Although, 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 our machine.

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 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; 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 I/O server. 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 user-space communication primitives provided by IBM's Message Passing Library (MPL).

Jovian-2 is significantly different from the original Jovian I/O library [1]. These changes were prompted by our experience with real I/O-intensive applications and by changes in our experimental platform. There are four main differences. First, Jovian assumed a loosely synchronous model of computation and provided a *collective-I/O* interface; Jovian-2 makes no such assumption and provides a simple `lio_listio()`-like interface. Since all clients were guaranteed to participate in a request, Jovian servers would wait for requests from all their clients before issuing disk requests. Jovian-2 servers, on the other hand, try to keep the disk busy by issuing disk requests as soon as possible. Second, Jovian assumed that individual requests would be small and provided support for coalescing multiple requests. Jovian-2 assumes programmers are aware of the costs of I/O and individual I/O requests are of substantial size. Third, Jovian servers were implemented as separate processes; Jovian-2 servers are implemented as threads in the application's address-space. Finally, Jovian did not provide support for striping over multiple disks whereas Jovian-2 supports user-customizable striping.

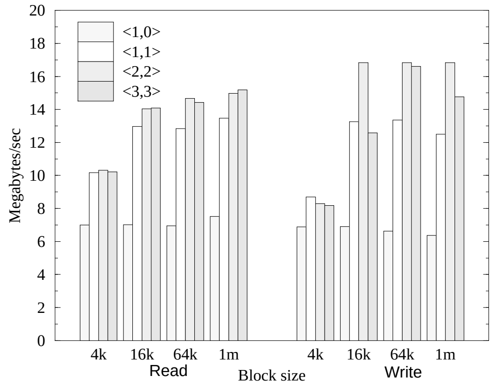

Figure 1: Maximum application-level I/O rates for JFS. `<x,y>` indicates the number of disks on each of the two SCSI buses.

3 Micro-benchmarks

We conducted a series of experiments using micro-benchmarks to characterize the performance of our experimental platform. These experiments had two goals. First, we wanted to determine the maximum application-level I/O bandwidth, which is a more realistic baseline to evaluate the performance of applications than peak disk or SCSI transfer rates. Second, we wanted to determine the parameters and configurations that provide the best performance. This information is needed to tune applications to effectively use the I/O system.

In this section we present results for both the native Unix filesystem available on the SP-2 and for Jovian-2. For our study, we used Jovian-2 for the earth-science applications and the Unix filesystem for the sparse-matrix factorization applications.

3.1 Journaled File System

The Journaled File System is the default Unix filesystem available on AIX 3.2. To measure single node JFS performance, we used a modified version of the widely used `iozone` benchmark [13]. `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 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 4 KB and 4 MB; with one to six outstanding requests per disk and six different disk configurations. For each experiment, the benchmark wrote and read back a 70 MB file. We used a 70 MB file to ensure we measured disk activity and not file cache performance. Along the request-size dimension, the bandwidth curves saturate around 1 MB 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. The additional bandwidth provided by adding a third disk on a SCSI bus did not increase performance because of contention between disks for the bus.

Figure 1 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 25.2 MB/s and the maximum write bandwidth was 23.5 MB/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.

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:

- Local-access: data is located on the same node that the I/O requests are made on.
- Client-server: the nodes participating in the experiment are partitioned into clients and servers as shown in Figure 2(a). Clients run both the application thread and the I/O-server thread whereas servers run only the I/O-server thread. All I/O requests are made on clients.
- Peer-peer: all nodes run both the application and the I/O-server thread as shown in Figure 2(b). I/O requests can originate from any node.

In these experiments, the primary parameter was the number of nodes that performed I/O. On each node that performed I/O, we used four disks (two on each SCSI bus), since that provided the best performance, as shown in Section 3.1. Both file size and request size were scaled with the number of disks. For the local-access and the one-client-one-server cases, a single processor wrote and read back a 70 MB file. For larger configurations, the file size was scaled to ensure at least 70 MB of data per node. file In addition, we read a 70 MB file on every node between every write experiment and the subsequent read experiment. For each configuration, the file was striped over all available disks – local disks for the local-access case, server disks for client-server configurations, and all disks on all the nodes for peer-peer configurations.

The request size was scaled to request at least one striping unit from every disk. For a configuration that used d disks, a request of $striping\_unit\_size \times d$ KB was used. For configurations with multiple clients (or multiple peers), non-overlapping requests were generated. For peer-peer configurations using n nodes, $1/n$ of I/O was local and the rest remote.

Results for a representative subset of the experiments are presented in Table 1. All measurements include time required to flush the file cache. For comparison, the maximum JFS bandwidths for four disks per node are 15.0 MB/s (read) and 16.8 MB/s (write). For non-local requests, Jovian-2 reads files from disk using JFS and delivers them to the requesting application using MPL. An upper bound for the bandwidth that can be achieved for individual non-local requests can be computed by adding the time taken to read a striping unit from the disk into the memory of the server node and the time taken to move it to the client node. We measured the maximum communication bandwidth for 128 KB messages to be 32 MB/s.[1] Combining this with the maximum application-level I/O bandwidth via JFS (from Table 1), we arrive at 10.3 MB/s as an upper bound for non-local bandwidth for individual requests. In comparison, Jovian-2 achieved a read bandwidth of 9.3 MB/s for a one-server-one-client configuration. For configurations with relatively large I/O bandwidth – more servers than clients and large peer-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, due to write-behind in the filesystem.

4 Earth-Science Applications

The two earth-science programs, included in our application suite, `pathfinder` and `climate`, constitute a processing chain for Advanced Very High Resolution Radiometer (AVHRR) data. The Pathfinder AVHRR data sets are global, multichannel data from NOAA meteorological satellites. Both `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 [6] satellites.

Pathfinder: This program is the first in the processing chain and processes AVHRR global area coverage data. Its input consists one or more *daily data sets* which contain satellite imagery and several ancillary files which contain topographic and cartographic data about the earth and information that helps determine the position of the satellite at any given time. Each *daily data set* contains fourteen files, each containing data corresponding to a little more than one orbit. The output of the program is a single multichannel image of the world. `Pathfinder` performs calibration for instrument drift, topographic correction, masking of cloudy pixels, registration of individual pixels with locations on the ground and compositing of multiple pixels corresponding to the same ground location. The size of one *daily data set* is about 680 MB, the total size of all ancillary files is about 100 MB and the size of the output is 228 MB.

Each orbit file consists of about 10,000-13,000 scan lines, each scan line containing 3584 bytes. Input data is read in scan line by scan line, in *chunks* of 128 lines. Processing is done one chunk at a time. Ancillary files are not cached in memory. Instead, the data from these files is read when needed. The calibration, correction, masking and registration operations depend only on the data in the chunk being processed; the composition operation combines data from multiple chunks. For the first four operations, `pathfinder` maintains large in-core scratch data-structures which are reused for every chunk. For composition, `pathfinder` maintains an out-of-core intermediate version of the composite image. After the first four operations have been performed on a chunk, each data value in the chunk is mapped into the

[1] This measurement was done by determining the time required to send one million 128 KB messages between a pair of nodes.

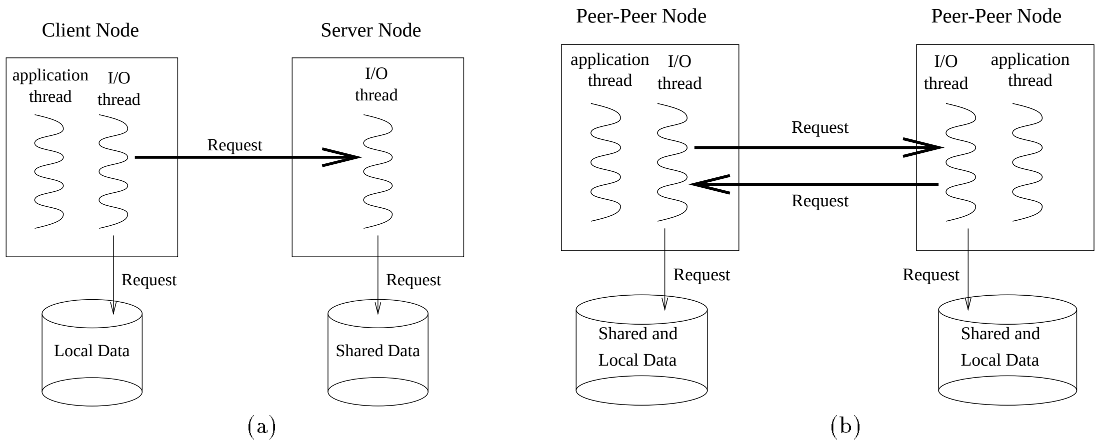

Figure 2: Configurations for (a) client-server and (b) peer-peer execution models

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

		one client,varying servers				4 nodes	8 nodes				Peer-peer		
Configuration	local	c1-s1	c1-s2	c1-s4	c1-s8	c3-s1	c7-s1	c6-s2	c5-s3	c4-s4	pp-2	pp-4	pp-8
Read bandwidth	14.3	9.3	17.5	21.2	22.8	8.5	8.9	16.6	23.3	25.9	13.8	16.5	26.0
Write bandwidth	16.7	8.7	16.3	20.6	21.6	9.7	19.8	24.8	30.0	27.9	21.2	26.2	33.2

Table 1: Micro-benchmark performance for Jovian-2. 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-peer configuration with x peers.

intermediate image and is compared with the corresponding pixel. If the new value is "better", it is copied into the pixel. In effect, the composition operation has been implemented as an out-of-core max-reduction. Note that the mapping between the pixels in the satellite images and the two-dimensional output image generated by `pathfinder` is complex and many-to-one. Once all input data has been processed, the intermediate image is scanned for pixels that have no data associated with them. This happens, for example, if the satellite image is clouded. Data for such pixels is computed by interpolation. Finally, the output image is generated by concatenating information about the data set with the intermediate image. The total I/O performed by an optimized sequential version, including I/O for out-of-core accesses, is over 28 GB.

Although processing of the 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 the AVHRR local area coverage data which is higher 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 the AVHRR global area coverage data processed by `pathfinder`.

Climate: This program processes the output of `pathfinder` and generates the AVHRR Land Climate data product. It performs a data selection and reduction operation. It extracts three frequency bands of the image generated by `pathfinder` and reduces them to a single band latitude-longitude grid. The output of `climate` is vegetation index map which is used to track global land cover change. Input to `climate` is the 228 MB file generated by `pathfinder`, of which the program reads 54 MB. In addition, `climate` reads 21.5 MB from an ancillary file. The output image is about 130 KB. The total I/O volume for `climate` is 75.5 MB.

I/O Optimizations:

- In both programs, input was being read one scan line at a time (3.5KB for `pathfinder`,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 20 MB. 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 the `stdio` library with 4 KB buffers. In most cases, including the patterns described above, this buffering was inappropriate. In the case of the patterns described above, individual requests were small (two bytes) and the distance between successive accesses (20 MB) was very

large. We replaced the buffered-I/O calls by their unbuffered analogues.

Parallelization:

In pathfinder, the map between the input satellite images and the output global composite image is data-dependent and cannot be computed a-priori. The amount of computation depends roughly upon the size of the input data processed. However, this relationship is weak as: (1) night images are not processed, (2) clouded images are partially processed and (3) ocean images are partially processed. The categorization of an image is also data-dependent and can be determined only after the data has been unpacked and partially processed.

We parallelized pathfinder by partitioning the output image in equal-sized horizontal strips. Each processor is responsible for all processing needed to generate its partition of the output image. We chose to partition the output image (instead of the input data) as this allows all combination operations[2] to be local to individual processors. No interprocessor communication is needed. We chose a horizontal partitioning scheme to take advantage of the row-major storage format used in all files (input, ancillary as well as output files). Horizontal striping allows I/O to be performed in large contiguous blocks.

Each processor computes the map from the input data set to the output image by subsampling (one scan line per chunk) all input files. It then reads the chunks that intersect with its partition of the output image. For each chunk, it maps each input pixel into the output image. Pixels that map into its partition are processed further, others are ignored. The individual partitions of the output image are also too large to be stored in main memory. Therefore, the composition operation is still out-of-core. Once all processing is completed, the final result is produced by concatenating the individual partitions.

In climate, the mapping between the pixels of the input image and those of the output image is data-independent and can be computed a-priori. The amount of computation to be done is proportional to the amount of input data. We parallelized climate by horizontally partitioning the output image. Each processor reads the data that maps to its partition of the output image. Load balance is achieved by ensuring that all processors read approximately equal amounts of data.

For both pathfinder and climate, the final image is generated by concatenating the images generated by individual processors.

Use of Parallel I/O:

In our experiments, we used four disks per node, two disks on each SCSI bus. We replaced calls to Unix I/O routines by calls to Jovian-2 routines. All ancillary files were replicated and striped across the four disks on every node. For client-server configurations, all input and output files were striped over all the disks of all server nodes; for peer-peer configurations, these files were striped over all participating disks. Every node running a pathfinder process (that is, the clients in client-server configurations and all participating nodes in peer-peer configurations) created a separate temporary file to hold its partition of the intermediate image. This file was striped over the four local disks. The striping-unit size in all cases was 128 KB.

[2] All input pixels that map to a single pixel in the output image are combined by a max-reduction operations to get the final value of an output pixel.

4.1 Results and Analysis

We ran pathfinder and climate for one *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 2 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 constructs the map from the input images to the output image by reading one scan line per chunk of 128 scan lines. As a result, the total amount of input grows with the number of processors that are running the application (clients in client-server, all nodes in peer-peer). This growth can be avoided by partitioning the task of constructing 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 that is read during the out-of-core max-reduction 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 has to be read to decide whether any pixels are to be updated. Writes are needed only if at least one of the pixels need to be updated, and then only for the bounding box around the pixels to be updated.

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 21.5 MB and total non-local I/O (input data) is 54 MB. The output volume is 130 KB.

Early results indicated that there was a large difference between the performance of peer-peer and client-server configurations for pathfinder. With abbreviated input (one orbit file instead of fourteen), pathfinder running on a four processor peer-peer configuration was able to achieve only a 400 KB/s per-processor non-local read bandwidth. With the same input, it was able to achieve a per-processor non-local read bandwidth of 6 MB/s on a client-server configuration of the same size (three clients, one server). The write bandwidth on a four-processor peer-peer configuration was better at 3 MB/s but was much lower than the 7.4 MB/s achieved on the corresponding client-server configuration of same size (three clients, one server). Note that the non-local reads are used to input the satellite data in chunks of 128 scan lines and are interspersed with computation, whereas the non-local writes are used for final output and are bunched together. The total execution time for an abbreviated pathfinder run (single orbit file) on a four processor peer-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

Config (client/server)	4 nodes	8 nodes			12 nodes			16 nodes		
	c3-s1	c7-s1	c6-s2	c5-s3	c11-s1	c10-s2	c9-s3	c15-s1	c14-s2	c13-s3
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

Table 2: Breakdown of total I/O volume (in MB) for `pathfinder`. Output volume is 228 MB for all configurations.

both peer-peer and client-server configurations (see Table 1). This might lead to speculation that applications that do significant amount of computation hamper I/O performance on peer-peer configurations. Section 5 provides a counter-example. It provides performance results for a program that performs substantial computation and I/O on a peer-peer configuration. It achieves good performance by using extensive global information about future I/O requirements and a one-sided communication model. Our current hypothesis is knowledge of future I/O requirements is necessary to achieve good computation and I/O performance on peer-peer configurations. We intend to test this hypothesis in our future research. For the rest of the experiments with `pathfinder` and `climate`, we limited ourselves to client-server configurations.

Figure 3 shows a breakdown of execution time for `pathfinder` for a set of client-server configurations. There are three interesting points. 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, for a given number of nodes, configurations with a small number of servers achieved the best performance. This is unsurprising as the bulk of the I/O is for intermediate read/write operations and is directed to local disks. It should be noted that increasing servers in a fixed size configuration has two conflicting effects: (1) it increases the bandwidth for non-local I/O (by increasing the number of nodes that act as servers) and (2) it decreases the bandwidth for local I/O (by reducing the number of clients). 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 configurations. There are two reasons for this. First, since all processors independently compute the map from input coordinates to output coordinates, 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 the portion it needs to process. 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 computing the map between input and output coordinates, followed by a global exchange of information. An even better solution would be to compute the map from satellite image coordinates to the output image coordinates during the process of converting raw sensor readings (level 0 data in NASA parlance) to the AVHRR orbital data files (level 1b data in NASA parlance). This conversion occurs earlier in the processing chain than `pathfinder`. This change would also help eliminate the growth in the total amount of computation as individual processors can now read only the portions of chunks that map to their partition of the output image. Note that this would require restructuring of `pathfinder` code to process variable-size chunks.

Table 3 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. A parallel-I/O library that provides a collective-I/O interface and coalescing of requests from multiple clients would usually not be able to do this as it would wait until requests are received from all clients before issuing any requests to the disks. A collective-I/O library that provided only partial coalescing and issued disk requests without waiting for all requests to arrive, would be able to utilize the server(s) over a longer time.

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 all intermediate I/O is to local disks and that each processor processes exactly the data that maps into its segment of the output image.

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

Table 5 presents end-to-end I/O rates for both `pathfinder` and `climate`. It shows that both programs are now compute-bound. It also shows that it is possible to achieve end-to-end I/O rates over 26 MB/s in earth-science applications.

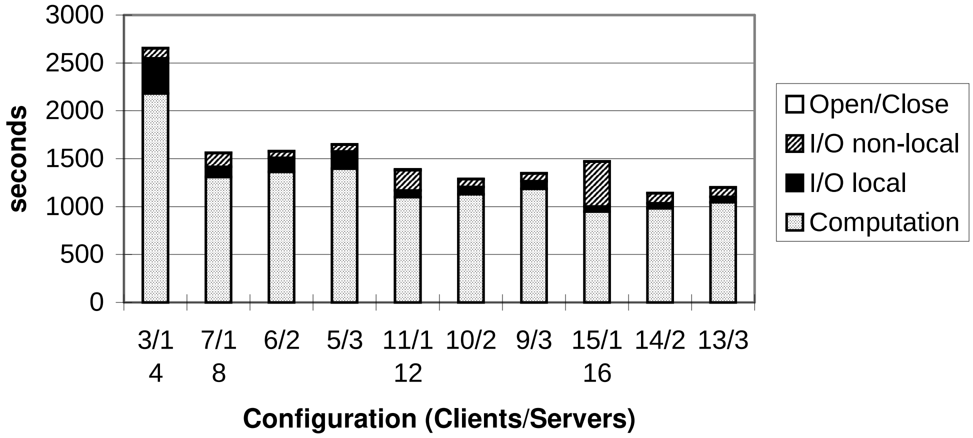

Figure 3: Breakdown of execution time for `pathfinder`. The numbers in the bottom row indicate the total number of nodes in the configuration. An unmodified version of `pathfinder` ran for 18,800 seconds on a single processor of the SP-2. Of this, about 13,600 seconds (76%) of the time was spent waiting for read/write operations.

	4 nodes	8 nodes			12 nodes			16 nodes		
Config (client/server)	c3-s1	c7-s1	c6-s2	c5-s3	c11-s1	c10-s2	c9-s3	c15-s1	c14-s2	c13-s3
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
Intermediate read	85.8	246.4	192.6	151.5	510.4	453.0	388.8	634.5	644.0	600.6
Intermediate 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

Table 3: Aggregate application-level I/O rates for `pathfinder`. All rates are in MB/s. The aggregate I/O rate is computed by multiplying the per processor application-level I/O rate by the number of clients. Per processor 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.

	4 nodes	8 nodes			12 nodes			16 nodes		
Config (client/server)	c3-s1	c7-s1	c6-s2	c5-s3	c11-s1	c10-s2	c9-s3	c15-s1	c14-s2	c13-s3
pathfinder	11.8	17.9	17.5	16.2	20.8	22.7	21.7	18.8	26.5	25.5
climate	1.2	2.5	2.3	1.9	–	–	–	–	–	–

Table 5: 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.

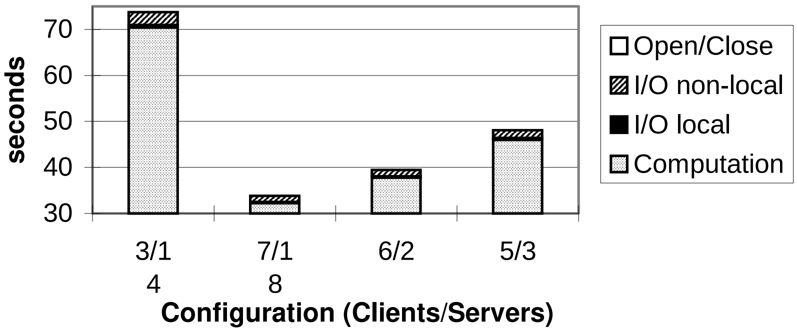

Figure 4: Breakdown of execution time for `climate`. The numbers in the bottom row indicate the total number of nodes in the configuration.

	4 nodes	8 nodes		
Config (client/server)	c3-s1	c7-s1	c6-s2	c5-s3
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 4: Aggregate application-level I/O rates for `climate`, in MB/s. The aggregate I/O rate is computed by multiplying the per-processor application-level I/O rate by the number of clients. Per-processor 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.

```
for i = 1 to S do
  for all S_j with j < i and S_ij ≠ ∅
    Read S_j
    Update S_i with S_j
    Discard S_j
  Factor S_i
  Write S_i to disk
```

Figure 5: Out-of-core Sparse Cholesky Factorization

5 Out-of-core Sparse-Matrix Factorization

Many scientific and engineering applications require the solution of very large sparse linear systems. Assuming a total memory pool of 50 GB, the largest sparse system (with 5% sparsity and double-precision complex arithmetic) that can be solved in-core on current supercomputers consists of about 250,000 equations.[3] 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 associated programs for parallel symbolic factorization and parallel matrix partitioning. We have selected two of these programs, the sparse-matrix partitioner (`partitioner`) and the Cholesky factorization programs (`factor`) for our study. Like `pathfinder` and `climate`, this pair of programs forms a processing chain.

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 [11]. This particular formulation of Cholesky factorization is based on *supernodes*. Each supernode is a set of contiguous columns such that every adjacent column in the set has an identical sparsity structure below the diagonal. Using supernodes enables the use of efficient dense linear-algebra kernels [5], as well as large transfers between secondary storage and primary memory. These applications assume a peer-peer configuration and directly use Unix I/O calls.

[3] We arrive at this number by calculating the number of double-precision complex values that a 50 GB memory will hold and by using this number and the sparsity to compute the corresponding number of equations. This number is an overestimate, as it ignores the memory required to hold the data-structures used to efficiently store the sparse matrix.

Partitioner: this program has two input files, the *matrix file* which contains the structure of the original matrix (A) and its non-zero values, and the *index file* which contains the sparsity structure of the factor L. The index file is generated by a symbolic factorization of A prior to the execution of `partitioner`. `Partitioner` performs two operations: (1) computing and allocating space for the *fill-ins*, which are locations in A that are originally zero, but will become non-zero (in L) after the factorization; and (2) distributing the Cholesky factor, L, to the processors participating in the factorization.

The Cholesky factor is partitioned using a 2-D strategy originally developed in [17]. The processors are organized in a $k \times m$ grid. Let $P_{r,q}$ denote the processor number at the rth row and qth 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 among 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.

This program 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. All requests in this phase are very large (25 MB) and contiguous. Furthermore, all requests are read requests and use blocking I/O calls. With the exception of the columns on the partition boundaries, all I/O in this phase is to the local disk. Requests in the second phase also access large contiguous chunks but the request size is smaller (5 MB) and two local files, instead of one, are used.

Factor: as mentioned above, this program implements left-looking supernodal Cholesky factorization. Figure 5 provides a high-level algorithm. 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 as long as the factorization (step 6) has been 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 possible given the memory constraints. The requests are issued in the order that the corresponding data 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.

A key data structure in `factor` is the *elimination tree* [10] generated during symbolic factorization using the structure of the sparse matrix. This structure contains dependency information between different supernodes and does not change over the course of the computation. Therefore the exact sequence of supernode update operations is known a-priori and can be used to generate a schedule for I/O and communication for all processors. This information can be exploited when performing step 3 of algorithm 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. 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 requests 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. Prefetched non-local data is injected into the communication network when the local computation reaches an appropriate point in the schedule.

5.1 Results and Analysis

We ran `partitioner` and `factor` on a variety of peer-peer configurations. In all configurations, we used only one local disk at each node. To evaluate the benefit of future knowledge provided by the elimination tree, we conducted experiments with two versions of `factor` – one that used the information for prefetching (`factor`) and the other that did not (`factor-np`). In `factor-np`, processors make explicit I/O requests to their peers and service the requests from their peers at particular points in the execution. Specifically, a processor is available to serve I/O requests from its peers either after step 4 of the algorithm outlined in Figure 5 or while it is waiting for the completion of its non-local I/O requests. As before, all computation on a supernode is performed at the processor in which the data is stored. Since I/O operations are performed on demand, non-local requests pay for one round-trip delay as well as any delay incurred by the processor that is servicing the request. Such delays may occur due to local computation at the service node and due to interactions with other peers.

	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 7: Application I/O volumes (in MB) for 16 processors.

We used three input matrices in our experiments – *skirt*, *sara-1* and *sara-2*. Table 6 presents some characteristics of these matrices. The first two, *skirt* and *sara-1* correspond to roughly the same number of equations but *skirt* has fewer non-zeroes and is more sparse. The third matrix, *sara-2* is similar in the number of non-zeroes to *sara-1* but corresponds to twice as many equations. As a result, *sara-2* is significantly more sparse than the other two, contains relatively thin supernodes and needs more space to store the sparse-matrix data-structures.

Table 7 shows the total amount of I/O performed by both applications for the three matrices. Since the *elimination tree* is replicated over all processors, the size of `partitioner`'s output, and consequently the size of `factor`'s input, increases with the number of processors. The table shows the I/O volumes for 16 processors, the largest possible configuration on our machine.

Table 8 presents the aggregate application-level I/O rates achieved by `partitioner`. This number is computed by dividing the total volume of I/O by the sum of the time spent issuing I/O requests and the time spent waiting for them to complete. The superlinear growth in the application-level I/O rate, seen for all three phases, is a caching effect. The *index file* is read in both the Read-1 and Read-2 phases. For the 16 processor configuration, the Read-2 phase is operating entirely out of the file cache. Similarly, the write phase benefits from the write-behind nature of the file cache. The performance of writes lags significantly behind that of reads because of a group of small unbuffered writes that dominate the write time. This effect can be easily eliminated by using a buffer to collect these writes (`stdio` should be adequate).

The breakdown of execution time for `partitioner` is shown in Table 9. It shows that (1) I/O is a small fraction (7.6%-25%) of the total execution time and (2) I/O scales well with increasing numbers of processors. In fact, I/O scales better than the computation for all three matrices. For these matrices, I/O takes less than 25% of the total program execution time, often significantly less.

Matrix	N	$\|A\|$	$\|L\|$	Type	Description
skirt	45,361	1.3×10^6	45.8×10^6	Real	NASA
sara-1	44,856	2.6×10^6	30.4×10^6	Complex	Structural Acoustics
sara-2	80,651	2.9×10^6	28.4×10^6	Complex	Structural Acoustics

Table 6: 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.

	4 nodes			8 nodes			16 nodes		
Matrix	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 8: Aggregate application-level I/O rates (MB/s) for `partitioner`.

	4 nodes		8 nodes		16 nodes	
Matrix	Total	I/O	Total	I/O	Total	I/O
skirt	309.6	77.6 (25.1%)	222.4	38.4 (17.3%)	162.0	19.3 (11.9%)
sara-1	522.3	79.2 (15.2%)	401.7	40.1 (10.0%)	319.8	24.3 (7.6%)
sara-2	2,568.4	541.7 (21.1%)	2,101.2	319.2 (15.2%)	1,568.4	177.9 (11.3%)

Table 9: Execution time breakdown for `partitioner`, in seconds

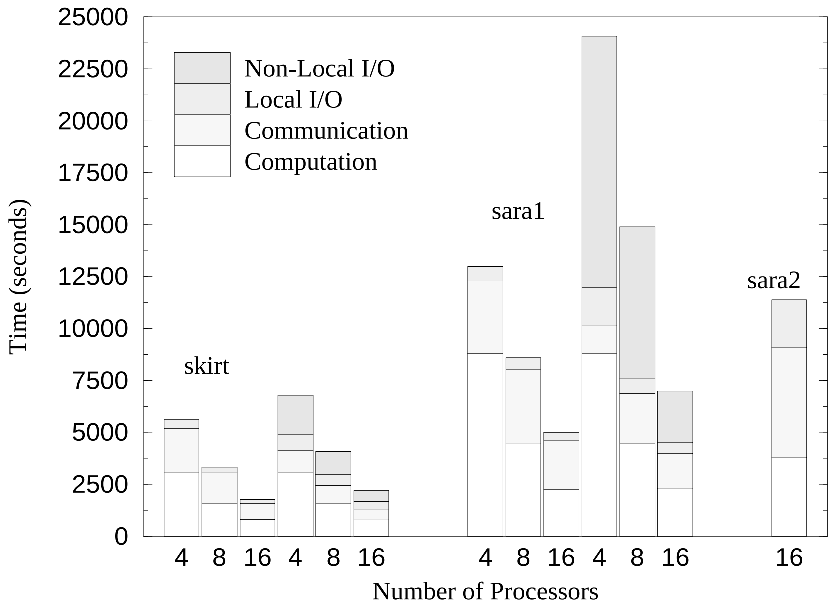

Figure 7: Execution time breakdown for `factor` and `factor-np`. The graph shows results for 4-, 8- and 16-processor configurations for both `factor` and `factor-np` on *skirt* and *sara-1*. For *sara-2*, only the results for `factor` on a 16-processor configuration are shown; factoring *sara-2* takes too long on other configurations to allow much experimentation. For *skirt* and *sara-1*, the first three bars show the results for `factor` and the second three bars show the results for `factor-np`.

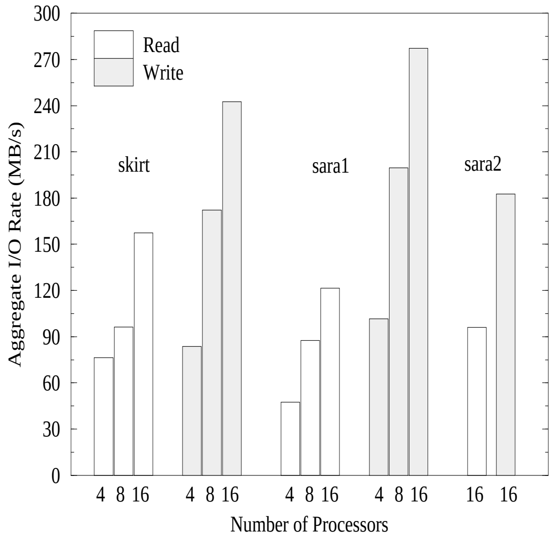

Figure 6: Aggregate application-level I/O rates for `factor`. Results are shown for 4-, 8- and 16-processor configuration for *skirt* and *sara-1*. For *sara-2*, only the results for the 16-processor configuration are shown; experiments for smaller configurations take very long to complete.

Figure 6 shows the aggregate application-level I/O rates seen by `factor`. It shows that `factor` is able to achieve an application-level read bandwidth up to 170 MB/s and an application-level write bandwidth up to 270 MB/s.

The execution time breakdowns for `factor` and `factor-np` are shown in Figure 7. In addition to showing how the different parts of each program scale, this graph also quantitatively demonstrates the utility of knowledge about future I/O requests.

The breakdown for `factor` shows that the computation scales linearly with the number of processors. The I/O performance scales fairly well but not as much as the computation. Communication, however, does not scale well. In all cases, I/O takes a fairly small percentage of the total execution time. 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 matrix, I/O takes only about 20% of the total execution time.

Comparison of these results with corresponding results for `factor-np` shows that lack of knowledge about future I/O requests can degrade performance by between 23 and 86%. Furthermore, the fraction of execution time spent waiting for I/O increased from between 1% and 7% for `factor` to between 40% and 56% for `factor-np`. Almost all of the increase in execution time is due to time spent waiting for non-local I/O requests to complete.

The slowdown was more pronounced (86%) for *sara-1* than for *skirt*. We speculate that this difference is due to the different amount of computation performed per supernode. Recall that a processor is available to serve I/O requests from other processors either after step 4 of the algorithm outlined in Figure 5 or while it is waiting for the completion of its own non-local I/O requests. The less sparse nature of *skirt* results in a more uniform distribution of per-supernode computation time. Also since nonzero precision in *skirt* is less than nonzero precision in *sara-1*, the average per-supernode computation is also smaller for *skirt*. In contrast, *sara-1* has a large variation in the per-supernode computation times caused by a few very large supernodes. As a result, in certain phases of the computation for *sara-1*, service of I/O requests from other processors is delayed for large intervals causing cascaded performance degradation of non-local I/O. As mentioned above, non-local I/O dominated the I/O costs for `factor-np`: for *sara-1*, 87% of the I/O time was spent for non-local I/O, constituting 50% of total runtime.

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. Based on our experience with these applications as well as our examination of other NASA satellite-data processing programs, we believe that, while many I/O-intensive applications are currently not designed to generate large I/O requests, relatively little effort is required to modify them to do so. In other 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 was available and could be used to prefetch data. For `pathfinder` and `climate`, processors subsample the input files in the partitioning phase. At the end of this phase, every processor has complete information about its future requests for input reads. For the modified version of the out-of-core max-reduction (where modification consisted of a pair of simple loop-splitting and loop-reordering transformations), information about updates to all frequency bands of the output image is known before any updates are performed. For `factor` and `partitioner`, the sequence of requests is available from the elimination-tree structure generated by symbolic factorization. Similar experiences have been reported by Patterson et al [15]. They report that after relatively simple loop-splitting transformations, significant knowledge about future I/O requests is available in all five I/O-intensive programs they studied.

  Furthermore, three of the applications in this study (all the ones that write a significant volume of data), could be structured to take good advantage of the 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 more predictable. Exploiting locality in this manner is beneficial for *out-of-core* applications [1, 2, 14] on both client-server and peer-peer configurations. In either configuration, exploiting locality improves I/O performance as well as total execution time.

Diskful machines are important:

Diskful machines (machines with local disks) allow problems to be partitioned such that most of the I/O requests are satisfied by local disks. As noted above, local disk accesses have a higher application-level bandwidth with the associated benefit of guaranteed lack of contention for the disk and the file cache. 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 [12] 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 loop splitting, interchanging the order of nested loops [18] 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, 16]. 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).

We recognize that this paper describes experiences with only four programs. However, we believe that a substantial class of I/O-intensive programs will be able to achieve good I/O performance with simple I/O interfaces. This belief is based on our examination of other NASA satellite-data processing programs and on the experiences reported by Patterson et al [15]. The characterization study by Crandall et al [4] provides another example. It describes a significantly different program running on a machine with much lower I/O bandwidth (the JPL terrain rendering application running on an Intel Paragon) that is able to achieve relatively good I/O performance with just asynchronous I/O requests. We speculate that with sufficient I/O bandwidth and efficient asynchronous I/O support and an interface similar to `lio_listio()`, most I/O-intensive programs will be able to achieve good I/O performance.

Good performance on peer-peer systems is possible:

Our experience with applications that do substantial I/O and computation on peer-peer configurations was mixed. On one hand, the performance of `pathfinder` on peer-peer configurations was poor; on the other hand, `factor` achieved excellent performance. The problem of achieving good computation performance on processors that are serving data to others has been previously noted by Kotz and Cai [8]. 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%. We believe that knowledge of future I/O requests (local and off-processor) is likely to be the key to achieving good I/O performance on peer-peer configurations. Our experiments with `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. The other version of `factor`, which did not take advantage of this knowledge, performed significantly worse.

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 microbenchmarks 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 filesystem 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. 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()` was adequate as long as the application and I/O system were configured properly.

Acknowledgements

We would like to thank our shepherd, David Kotz, for his detailed comments. 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] R. Bennett, K. Bryant, A. Sussman, R. Das, and J. 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*, pages 1–10. ACM Press, July 1995. ACM SIGPLAN Notices, Vol. 30, No. 8.

[3] A. Choudhary, R. Bordawekar, M. Harry, R. Krishnaiyer, R. Ponnusamy, T. 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.

[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] M. King. Earth Observation System Project Science homepage. *http://spso.gsfc.nasa.gov/spso_homepage.html*, 1995.

[7] D. 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.

[8] D. Kotz and T. 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*, pages 78–89, April 1995.

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

[10] J. W. H. Liu. The Role of Elimination Trees in Sparse Factorization. *SIAM Journal of Matrix Analysis and Applications*, (11):134–172, 1990.

[11] E. G. Ng and B. W. Peyton. Block Sparse Cholesky Algorithms on Advanced Uniprocessor Computers. *SIAM Journal on Scientific Computing*, 14(5):1034–1056, September 1993.

[12] N. Nieuwejaar and D. 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*, pages 47–62, April 1995.

[13] W. Norcutt. IOZONE benchmark program. Available at *ftp://ftp.cs.umn.edu/packages/FreeBSD/2.0.5-RELEASE/ports/utils/iozone*, 1991.

[14] 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.

[15] R. Patterson, G. Gibson, E. Ginting, D. Stodolsky, and J. Zelenka. Informed prefetching and caching. In *Proceedings of the Fifteenth ACM Symposium on Operating System Principles*, pages 79–95, December 1995.

[16] J. Rosario and A. Choudhary. High-performance I/O for massively parallel computers: Problems and prospects. *IEEE Computer*, 27(3):59–68, March 1994.

[17] Edward Rothberg and Anoop Gupta. An Efficient Block-Oriented Approach to Parallel Sparse Cholesky Factorization. In *Proceedings of Supercomputing '93*, pages 503–512, Portland, OR, November 1993.

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