Support for Parallel Out of Core Applications on Beowulf Workstations

Matthew M. Cettei Walter B. Ligon III Robert B. Ross
Parallel Architecture Research Lab
Clemson University
102 Riggs Hall
Clemson, SC 29634-0915
864-656-7223
mcettei@eng.clemson.edu walt@eng.clemson.edu rbross@eng.clemson.edu

Abstract— Beowulf workstations are a new approach to parallel computing that combine the characteristics of the Pile-Of-PC architecture with freely available software and provide high performance at a low cost for many applications. In order to provide this performance in the face of increasing problem sizes, efficient out of core (OOC) methods must be investigated. In light of shortcomings in operating system virtual memory performance, we have implemented a parallel file system and a new interface tailored to OOC algorithms to provide efficient and convenient access to disk resources in Beowulf workstations. This paper focuses on how these components extend the range of problem sizes which can be efficiently solved on Beowulf workstations in the context of a sample application, a Gauss-Seidel iterative solver. Execution times of the solver when using traditional virtual memory and the parallel file system are compared to characterize performance for various problem sizes, and it is concluded that the combination of parallel file system and new interface significantly increases the size of problems that can be efficiently solved.

Table of Contents

1. Introduction

As the popularity of parallel processing has increased, so has the need for low cost parallel computing resources. Clusters of workstations were one of the first attempts at providing parallel computing facilities at a lower cost than traditional massively parallel computers [1]. These clusters are often built using existing workstations which are used as interactive systems during the day. Machines are typically connected with Ethernet or a similar low-performance networking technology, can be heterogeneous in composition, and rely on extra software to balance the load across the machines in the presence of interactive jobs. The Pile-of-PCs architecture is an extension of the cluster of workstation concept emphasizing the dedication of the machines to the task of parallel processing and the use of inexpensive, non-proprietary hardware. A typical Pile-of-PCs consists of a cluster of machines dedicated as parallel processors, built entirely from commodity off the shelf parts, and employing a private system area network for communication [2]. The use of off-the-shelf parts results in systems that are tailored to meet the needs of the users, built using the most up-to-date technology at the time of purchase, and cost substantially less than previous parallel processing systems. The dedication of the machines to the task of parallel processing simplifies administration tasks and reduces the need for load balancing software.

The Beowulf workstation concept builds on the Pile-of-PCs concept by utilizing a freely available base of software including operating systems (e.g. Linux), message passing libraries (e.g. MPI and PVM), and compilers (e.g. gcc). The free availability of most system software source encourages customization and performance improvements. These improvements, along with new software developed by Beowulf users, are also typically made freely available and thus returned to the community [2]. Experiments have shown that Beowulf workstations can indeed provide high performance for real applications at a low cost, including applications such as N-body simulations [3], computational electromagnetics, and computational fluid dynamics.

Increases in processor performance have been dramatic over the last few years, especially in PC systems, greatly out pacing disk and memory performance. This processor improvement has decreased the time required for existing applications to run and at the same time has prompted new and larger applications to be developed. Many of these new applications operate on data sets much larger than those that could be held in-core, placing new I/O requirements on machines running these applications. Kernel virtual memory support provides a starting point for handling larger data sets in single process applications, but it falls short when multiple processes running on different machines with distributed memory all need access to the same data set.

In Beowulf workstations one of the most interesting and often underutilized resources in the system are the disks. Unlike most massively parallel machines, each node in a Beowulf workstation typically has its own disk and controller. However, often only a couple of these disks are used by taking advantage of a network file system such as NFS, because most Beowulf workstations do not provide an efficient and convenient means for combining these resources into a global pool. Thus application programmers must either rely on the operating system's demand paging on each local machine or hand code the I/O accesses and data transfer in order to use all the disk subsystems.

Parallel file systems are one approach to providing this global access to I/O resources in some systems. Parallel file systems are system software designed to distribute data among a number of I/O resources in a parallel computer system and to coordinate parallel access to that data by application tasks. However, availability and performance are only half of the picture; the user interface to this storage system must also make it convenient for application programmers to take best advantage of its potential. One solution is to provide multiple interfaces so that application programmers can choose which interface best fits their needs.

Although there is research taking place in parallel I/O and interfaces for I/O systems, much of the research has focused on massively parallel machines and distributed systems consisting of high end workstations. Little work thus far has examined these topics on Pile-of-PC systems or Beowulf workstations. These machines have their own unique characteristics which make it unclear how new parallel I/O techniques and interfaces will impact application development and performance in this environment, including large network packet sizes, large network latencies, off the shelf (OTS) operating systems, standard networking protocols, and evenly distributed I/O resources.

We are attempting to fill this niche by developing a system to provide convenient and efficient access to the I/O subsystem of a Beowulf workstation for use by programmers implementing out of core (OOC) applications. OOC applications are designed to explicitly handle data movement in and out of core memory avoiding the use of virtual memory. To do this we have constructed a parallel file system and interface libraries for use on Beowulf workstations, building both on previous parallel I/O research and our experiences with these systems. The libraries being developed include a UNIX interface with a partitioning extension, a scheduled I/O interface that implements a form of collective I/O, and a multi-dimensional block interface that is especially useful in implementing OOC applications. We are evaluating both the file system and these libraries using implementations of real parallel out of core applications on top of an operational Beowulf workstation, a machine consisting of seventeen Pentiums with a switched fast ethernet network that cost under $50K in 1996.

This paper describes our method for extending the range of efficiently solvable problems on a Beowulf workstation in the context of a sample application, a Gauss-Seidel iterative solver. This solution revolves around the use of two new software components, the Parallel Virtual File System (PVFS) and the Multi-Dimensional Block Interface (MDBI). PVFS is a parallel file system designed for use in Pile-of-PC and Beowulf workstation environments and provides the I/O accessibility needed for parallel OOC applications on the Beowulf workstation. The MDBI interface is a user interface designed for treating data files as multi-dimensional arrays of records. It allows the user to quickly specify accesses to blocks of records in a file and provides transparent user selectable buffering as well. These two components together provide the efficient and convenient access to data needed by the iterative solver for moving data elements in and out of core on compute nodes.

In Section 2 we will discuss the Parallel Virtual File System, including the basic components of the system, data layout on disks, and request structure and processing. Section 3 will cover the Multi-Dimensional Block Interface and discuss the benefits of using such an interface. The algorithm and implementation of the Gauss-Seidel iterative solver will be discussed in Section 4, and the results of our performance study are described in Section 5. Finally Section 6 will present our conclusions and directions of future work.

2. Parallel Virtual File System

Parallel file systems serve two primary functions: they allow data stored in a single logical file to be physically distributed among I/O resources in a machine and provide a mechanism for tasks in a parallel application to access the data concurrently and possibly independently. In theory, if the data is physically balanced among the I/O devices, the data requirements of the application are balanced between tasks, and the network bandwidth is sufficient, the system should provide effectively scalable I/O performance. In practice, however, there are a number of factors that can prevent many applications from achieving good scalability with parallel file systems. One source of problems is file system and application interaction, including mismatch between the physical layout of data and the distribution to the tasks [4], lack of coordination between application tasks resulting in poor disk utilization [5], and poor access patterns which result in

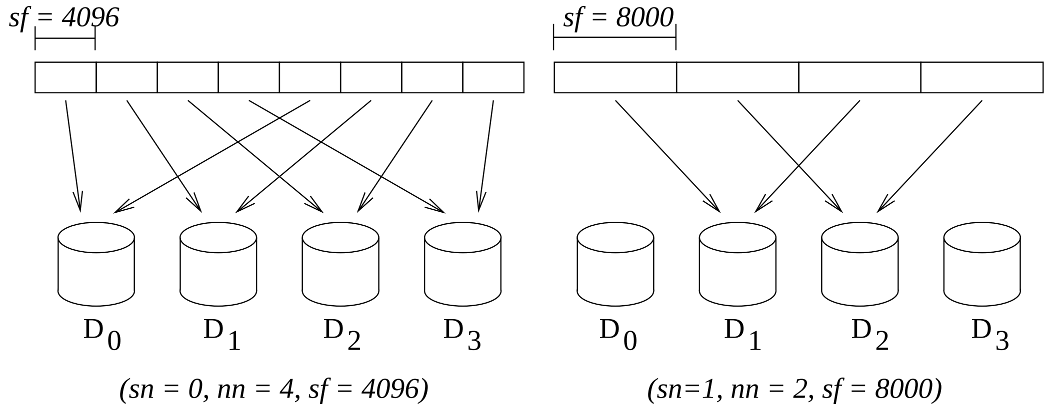

Figure 1: Striping Examples

large control message overhead. File system and operating system interaction, such as buffer space utilization and caching policy, is another potential problem area.

Our goal with PVFS is to provide a parallel file system tailored to the characteristics of a Pile-of-PCs with interfaces designed to mesh well with applications typical to our workload. The current system is the result of a number of experiments in both techniques for data transfer between application and file system and interfaces for interaction with the file system. One of the requirements of the PVFS project was for the system to remain a virtual file system; we wanted to build PVFS without modifying the existing system software on the platform using the native file system support to store data to disk. For this reason we chose to implement PVFS as a set of user-level daemons and an interface library that applications could use to interact with the system. The TCP protocol is used for all communication, and standard UNIX system calls and memory mapping of files are used to store data on disks. As a result, PVFS is able to operate in a variety of hardware and software environments, including common Beowulf workstation configurations.

There are two types of daemons used in the system, the management daemon and the I/O daemon. The management daemon is responsible for keeping track of metadata for the file system. Metadata is data that describes the characteristics of a file, including the owner, permissions, and striping of the file across the disks in the cluster. When a process attempts to create or open a PVFS file, the manager verifies that the process has permission to do so and passes this metadata to both the requesting process and the I/O daemons. The metadata is then available throughout the time the file is accessed without further communication with the manager, avoiding a potential bottleneck.

The I/O daemons run on each machine whose I/O subsystem is to take part in the parallel file system. Each I/O daemon is responsible for performing the disk accesses local to its machine. A simple request mechanism is available for specifying accesses for the IOD to perform, and applications directly contact these I/O daemons when reading and writing to avoid the bottleneck and latency problems of passing the requests through the manager.

Files stored on PVFS consist of an ordered set of stripes which are in turn made up of stripe fragments stored in files on disks in the cluster. Stripe fragments are distributed across I/O nodes using a round robin scheme. The starting node sn, the number of nodes used nn, and the stripe fragment size sf are all selectable by the user at the time the file is created. Figure 1 shows how a file, shown as a linear array of bytes, might be split into stripe fragments and distributed among the disks in a parallel file system for two (sn, nn, sf) combinations. In the first example, where $(sn = 0, nn = 4, sf = 4096)$, the file is divided into stripe fragments of 4096 bytes and spread across all of the four disks, starting with disk 0. In the second example, where $(sn = 1, nn = 2, sf = 8000)$, stripe fragments of 8000 bytes each are spread across two of the four disks in the file system starting with disk 1.

Because of the typically high network latencies in Pile-of-PC machines when using standard networking hardware and protocols, one of our primary concerns in designing a parallel file system for this environment was minimizing both control message overhead and the number of small messages. This has been addressed in PVFS by the use of the strided request mechanism. This mechanism allows for describing non-contiguous but regularly spaced regions in a file with a single set of parameters. Figure 2 shows an example strided region. Since studies have shown that as many as 80% of parallel file accesses use a strided pattern [6], providing this capability to access non-contiguous regions with a single request can significantly reduce the number of control messages and has the potential to increase message lengths by allow-

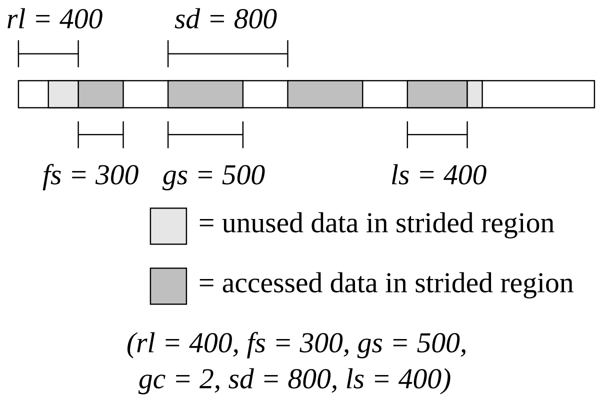

Figure 2: Accessing a Portion of a Strided Region

ing what would have been multiple small messages to be combined into a single, larger request. Other parallel file systems and interfaces supporting accesses to non-contiguous regions with single requests include Vesta [7], Panda [8], MPI-IO [9], and PASSION [10].

Our I/O daemon accepts strided requests in order to take advantage of this typical access pattern. Each read or write request consists of a set of six parameters:

- request location (rl) - location of start of request
- first size (fs) - size of starting partial block
- group size (gs) - size of each full block
- group count (gc) - number of full blocks
- stride (sd) - distance from start of one block to start of next
- last size (ls) - size of ending partial block

These parameters define a portion of a simple strided region of the PVFS file that is to be accessed. Figure 2 gives an example of how these parameters map into the PVFS file. In the example, a portion of a simple strided region of a file is accessed using the parameters ($rl = 400, fs = 300, gs = 500, gc = 2, sd = 800, ls = 400$). This capability makes it easy to build interfaces to extract records from a file or portions of multiple records (assuming a uniform size) in a cyclic fashion, often with a single request.

This simple request scheme is the building block of all our interfaces. For example this request scheme can be used to access a block of a two dimensional data set with a single request, as seen in Figure 3. Here the stride and group size are set so that only the data from a row that is in a selected block is accessed. The data set stored as a linear stream of $54,000$ bytes and is viewed as a matrix of nine rows of 6000 bytes each. A block consisting of 1000 bytes from each of three rows is selected for access.

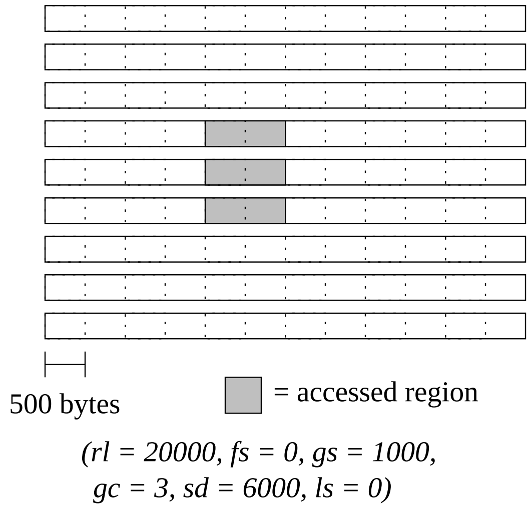

Figure 3: Accessing a Block of a 2-D Data Set

The rl value sets the start of the request to the location of the correct row element in the fourth row, and setting sd equal to the width of a row results in the same starting point for each access within the following rows. The gs value determines the amount of data accessed per row, and the gc value determines the number of rows from which data is accessed. In this case both fs and ls are set to zero because no "partial" strides are needed.

Application programmers can use this strided request mechanism via the partitioning extension of our UNIX-style interface, as described in [11]. Additionally, two scheduled I/O interfaces and the MDBI interface have been developed. The scheduled I/O interfaces implement a new approach to collective I/O and are still in an experimental phase. This paper will concentrate on the multi-dimensional block interface and how it can be used when implementing OOC applications.

3. Multi-Dimensional Block Interface

The issue of providing usable interfaces to parallel file systems is still an open topic. The most widely supported user interface to I/O systems is the UNIX interface. With this interface a file is viewed simply as a linear array of bytes of data, and operations are provided for seeking to positions in the file and reading and writing contiguous regions. However, it is difficult to use this interface as a parallel application interface for a number of reasons:

- Multiple operations are often needed to access multi-dimensional data
- Explicit seeks are needed to access partitioned data
- External synchronization is often required to manage access to shared data

These all result in more coding effort for the application programmer and also lead to two types of inefficiencies. First, additional overhead is caused by the number of system calls needed to perform the necessary seeks and accesses for partitioned or multi-dimensional datasets. Second, caching and prefetching by off-the-shelf file systems are often impaired by the access patterns of these applications, which often do not match the patterns of sequential applications.

Even the earliest parallel file system designers realized that this simple interface was inadequate for use in today's parallel applications. The interface options to some of the earliest parallel file systems such as Intel's CFS used various "modes" which determined how the accesses of various tasks in a parallel application mapped into the file [12]. This included global and independent file pointers and in some cases support for collective I/O or synchronization. This did give the programmer more options for describing and coordinating I/O; however, these "modes" provided no means for explicitly partitioning files between processes or describing the file in terms other than a linear sequence of bytes. Only recently have researchers realized the potential benefits of partitioning, strided requests, and the capability to more fully describe how and when data is to be accessed. This has led to many new approaches to parallel I/O interfaces, one of which is the idea that more application-specific interfaces should be developed. One common way to view data files is as a multi-dimensional matrix of some sort, so support for accessing files in this manner is an obvious approach. Examples of variants on this type of interface are the Panda interface for the iPSC/860 [8] and Vesta on the Vulcan multicomputer [7].

The MDBI interface is a library of calls designed to help in the development of OOC algorithms operating on multi-dimensional datasets by making it easier to manage the movement of data in and out of core. It allows the programmer to describe an open file as an N-dimensional matrix of elements of a specified size, partition this matrix into a set of blocks, then read or write blocks by specifying their indices in the correct number of dimensions. In addition, it supports buffering and read-ahead of blocks via the definition of "superblocks". The programmer specifies superblocks by giving their dimension in terms of the previously defined blocks of the file. Any time a block is accessed all other blocks in the superblock are read and held in a transparent user-space buffer on the compute node.

A set of parameters is first passed to the library to describe the logical layout and superblocks of the file:

- the number of dimensions, D,
- the size of a record, rs,
- D (block size, block count) pairs, giving the number of elements ne_i in a block and the number of blocks nb_i in the file $\forall i : 1 \leq i \leq D$, and
- D blocking factors $bf_{1..D}$, defining the size of superblocks in each dimension.

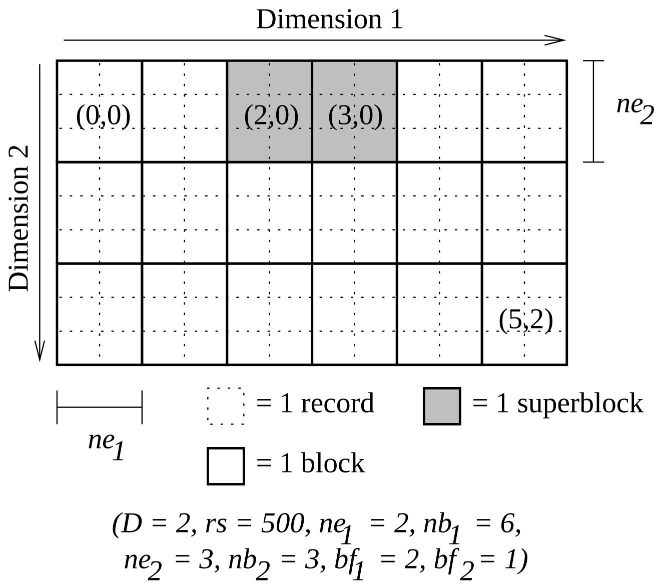

Figure 4: Specifying the Blocking of a 2-D Matrix

For example, Figure 4 shows the same file as in Figure 3 described to the library as a two dimensional matrix containing a 12×9 array of 500 byte records stored in row major order. This matrix is grouped into a 6×3 array of blocks, each of which is a 2×3 array of records.

Once the file has been described to the library, accesses to blocks can be specified simply by giving the coordinates. Blocks that are read are placed into a multidimensional array of records on the compute node of size $ne_1 \times ne_2 \times \cdots \times ne_D$. In addition blocks residing in the same superblock are also placed into a user-space buffer. For example, if block (2,0) were accessed in the matrix in Figure 4, block (3,0) would be read into the user-space buffer for the process as well, as it resides in the same superblock as defined by the blocking parameters.

The application library uses the coordinates of the request, the blocking values, the buffer factors, and the partitioning mechanism supported by PVFS to minimize the number of requests to the file system. In the two dimension case, accesses are converted into a strided access just as in Figure 3. This allows the entire block or superblock to be read with a single request. When the data is defined to be of more dimensions, multiple requests, a batch request, or a nested-strided request mechanism must be used.

```
DO {
  READ PARTITION DATA FOR NODE INTO MEMORY
  INVERT AND STORE DIAGONAL BLOCKS
  FOREACH ITERATION {
    FOREACH ROW OF BLOCKS IN PARTITION {
      CALCULATE NEW X VALUES FOR ROW
      EXCHANGE NEW X VALUES WITH OTHER TASKS
    }
  }
}
```

Figure 5: Pseudo-code for Virtual Memory Implementation

4. Gauss-Seidel Iterative Solver

Iterative solvers are used in a number of areas in order to find approximate solutions for systems of linear equations by solving the matrix equation $A\vec{x} = \vec{b}$. Our iterative solver uses a block-oriented Gauss-Seidel approach. The A matrix is decomposed into square blocks, and all operations are matrix-vector and vector-vector operations. In general, given an initial estimate of block-vector $\vec{x}$, a new approximation for subvector $\vec{x}_i^{new}$ is computed as:

$$\vec{x}_i^{new} = A_{ii}^{-1}\left(\vec{b}_i - \left(\sum_{j=0}^{i-1} A_{ij}\vec{x}_j^{old} + \sum_{j=i+1}^{m} A_{ij}\vec{x}_j^{old}\right)\right)$$

where $\vec{x}_i$ and $\vec{b}_i$ are subvectors of vectors $\vec{x}$ and $\vec{b}$ respectively, and A_{ij} is a submatrix of matrix A. When the RMS difference of $A\vec{x}$ and $\vec{b}$ falls below a desired level the computation is terminated. Convergence tests are not included in the application, but instead can be run periodically in parallel with the iterations. In order to compare the speed of different versions of the solver it was run for a fixed number of iterations instead of running until convergence, so the RMS difference was not calculated. The number of iterations was selected so that the iterative phase dominates the one-time parts of the program.

Our parallel algorithm uses a row-block partitioning with the A matrix broken into $m \times m$ blocks of $n \times n$ double precision complex elements. Figure 7 shows the partitioning of data for four compute nodes and data access during the update of a block of $\vec{x}$ by a single compute node. For each element $\vec{x}_i$, $\vec{x}_i^{new}$ is calculated using the in-core $\vec{x}$, in-core $\vec{b}_i$, A_{ii}^{-1}, and the blocks of one row of A. After n elements of $\vec{x}_i^{new}$ are calculated by each compute node they all exchange the new values via PVM messages to update their in-core $\vec{x}$ vector, and once all of $\vec{x}$ has been updated an iteration is complete.

Figure 5 gives the pseudo-code for the virtual memory (VM) implementation. In this implementation all partition data for a given task is read into memory before any calculations are made. This forces the operating system to manage the movement of data in and out of core when all physical memory is used. First each task calculates the inverses of its diagonal blocks of A and stores them in memory as well, then it proceeds to calculate new values for $\vec{x}$ elements, using matrix-vector and vector-vector operations on the rows of blocks. In this implementation the amount of data held on a machine is dominated by the rows of the A matrix which the processor will need, as the size of $\vec{x}$, $\vec{b}$, and the A^{-1} diagonal blocks is insignificant in comparison. The size of the in-core data is approximately:

$$IC_{vm} \approx \frac{M^2 \times E_{sz}}{N}$$

for an $M \times M$ matrix of E_{sz} byte elements on N processors. In all our tests, $E_{sz} = 16$ bytes, the size of a double precision complex value.

```
DO {
  SET BLOCKING AND BUFFER FACTORS
  INVERT AND STORE DIAGONAL BLOCKS
  FOREACH ITERATION {
    FOREACH ROW OF BLOCKS IN PARTITION {
      FOREACH BLOCK IN ROW {
        READ CURRENT BLOCK USING MDBI
        CALCULATE PARTIAL X VALUES
      }
      EXCHANGE NEW X VALUES WITH OTHER TASKS
    }
  }
}
```

Figure 6: Pseudo-code for MDBI Implementation

In the MDBI implementation of the algorithm, outlined in Figure 6, the diagonal blocks of A_{ii}^{-1} are calculated by the tasks and written back to a temporary file on the parallel file system before the iterations are started. The entire $\vec{x}$ vector and each compute node's portion of the $\vec{b}$ vector are kept in-core for the duration of the algorithm, while A_{ii}^{-1} and the blocks of A are read from disk when needed. However, the use of blocking factors can result in more than one block of A or A_{ii}^{-1} being held in-core on a compute node, so in this implementation the amount of data held on a machine is dependent on the blocking factors and buffer factors chosen for the A matrix file and the A^{-1} file. Since we are dividing A into square submatrices of $n \times n$ elements, the size of these in bytes is approximately:

$$IC_{mdbi} = n^2 E_{sz}(bf_{A_1} bf_{A_2} + bf_{A_1^{-1}} bf_{A_2^{-1}})$$

which is equivalent to:

$$IC_{mdbi} = \frac{M^2}{nb_A^2} E_{sz}(bf_{A_1} bf_{A_2} + bf_{A_1^{-1}} bf_{A_2^{-1}})$$

where $nb_A = nb_{A_1} = nb_{A_2}$.

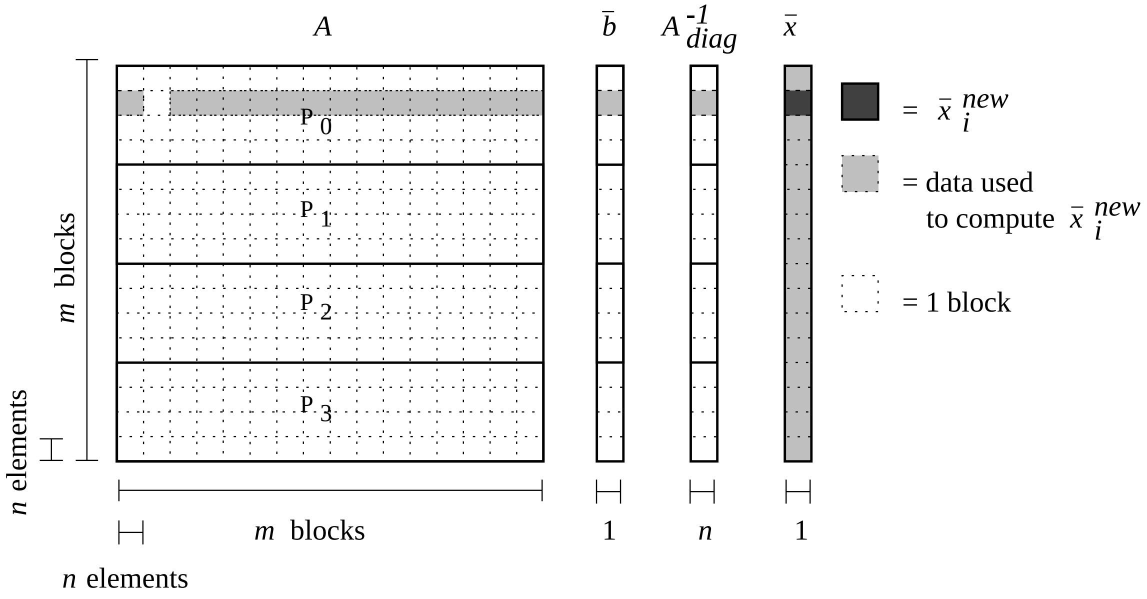

Figure 7: Data Distribution and Access for Iterative Solver

5. Results

This section describes our test environment, the testing performed, and the observations that we have made of the application during the study. The Beowulf machine used for testing consists of 17 workstations with the following:

- Pentium 150 MHz CPU
- 64 MB EDO DRAM
- 128 MB swap space
- 2.1 GB IDE disk
- Tulip-based 100 Mbit fast ethernet card

A Bay Networks fast ethernet switch in full-duplex mode is used to connect the machines. The machines all run Linux v2.0.27 with the replacement Tulip driver written by Donald Becker. The network is isolated from outside traffic.

Seven iterations were made for each test point in each graph. Of these the high and low execution times were thrown out, and the remaining five were averaged to give the values shown in the graphs. Files were stored on a varying number of disks, but the stripe size in each case was fixed at the size of one row of elements for matrices up to 4K $\times$ 4K and half of a row for larger matrices, so each I/O request was distributed evenly among all I/O nodes. In all MDBI runs a block count (nb_A) of 64 is used, which means that the A matrix was split into a 64 $\times$ 64 array of submatrices. The buffer factor for A (bf_A) was set to (64,1), so in all cases an entire row of blocks was buffered. The buffer factor for the A_{ii}^{-1} diagonal file ($bf_{A^{-1}}$) was also set to (64,1) so that the entire file would reside in core on each compute node. This simplifies the calculation of IC_{mdbi} for our tests to:

$$IC_{mdbi} = \frac{M^2}{2}$$

Our first three sets of tests were run to examine the range of problem sizes that could be efficiently solved using the two implementations of our iterative solver on four, eight, and sixteen compute nodes. Problem sizes were varied from 1K $\times$ 1K to 8K $\times$ 8K elements. Three configurations of the application were tested: one using virtual memory (VM), another using PVFS and the MDBI interface with data stored on compute nodes (MDBI), and the last using PVFS and the MDBI interface with data on separate I/O nodes (MDBI-S). Table 1 summarizes the results of these tests.

Figure 8 shows the execution times for the algorithm on four compute nodes. The results of the VM tests show that for problems of size 1K $\times$ 1K and 2K $\times$ 2K swapping is unneeded and the virtual memory version outperforms the MDBI version. With a 3K $\times$ 3K problem, approximately 36 MB of data are required for storage of the A matrix on each node. This appears to fit mostly in-core, although the fact that the MDBI version outperforms the VM version would indicate that perhaps some swapping is taking place. At 4K $\times$ 4K, $IC_{vm} = 64$ MB, the use of demand paging increases dramatically, and performance degrades significantly.

The MDBI implementation shows more consistent behavior, scaling well for the entire tested range. For the

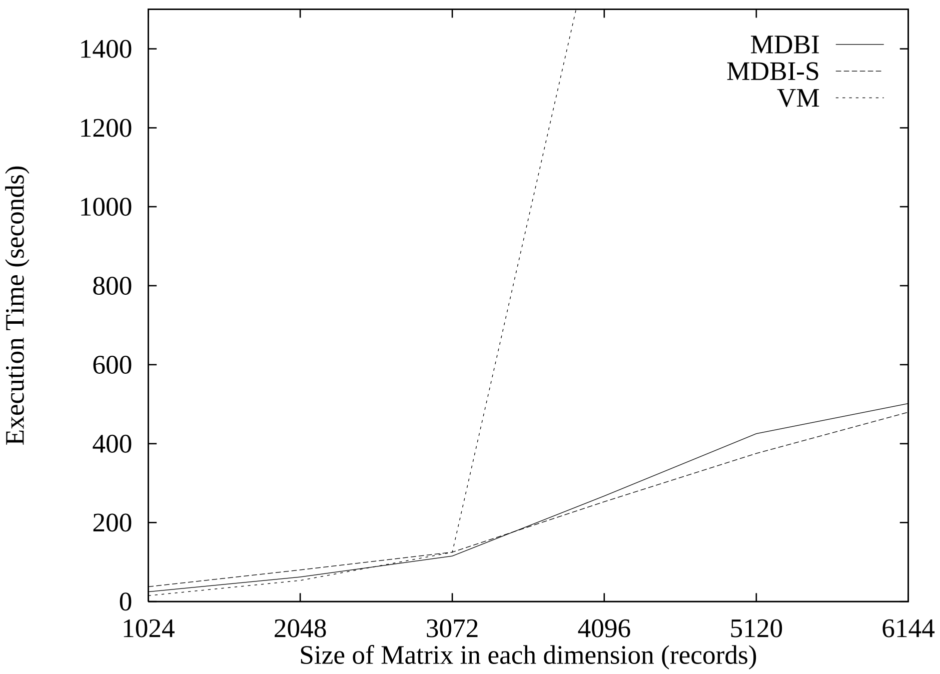

Figure 8: Execution Time vs. Problem Size for 4 Nodes

6K $\times$ 6K problem $IC_{mdbi} = 18$ MB, indicating that larger problems could be solved using the same blocking and buffer factors. It is important to note that IC_{mdbi} is independent of the number of nodes used in the computation; each task is caching a single row of blocks from A and the diagonals of A^{-1} regardless of how many tasks are participating in the computation.

Figure 9 shows this same series of tests on eight compute nodes. For problem sizes up to 5K $\times$ 5K the VM version soundly outperforms the MDBI version, in many cases running twice as fast. The reduction in memory utilization for disk buffers due to using all eight nodes to store file data increases the point at which demand paging becomes necessary so that with $IC_{vm} = 50$ MB for eight nodes with the 5K $\times$ 5K matrix size the VM algorithm still performs well. However, the 6K $\times$ 6K problem, with $IC_{vm} = 72$ MB, causes significant paging and results in an execution time of more than ten times that of the 5K $\times$ 5K problem. Again, the MDBI version of the code scales well as the problem size grows. Similar behavior is seen in the application using 16 compute nodes, with the execution time of the VM version jumping once the problem size grows larger than core memory.

The MDBI-S data points were gathered to investigate the validity of using compute nodes as I/O nodes. Often systems use separate nodes for I/O nodes, and we wanted to determine if this was a preferable configuration given a fixed set of resources. One can see that for small problem sizes using compute nodes as I/O nodes actually outperforms using separate I/O nodes, most likely because the locality of data combined with the abundance of free memory on the machines allows most file data to reside in cache. For larger problem sizes separate I/O nodes do result in better performance as the available core memory is doubled, increasing the amount of file data that can be cached. By comparing the four node MDBI-S data to the eight node MDBI data, however, we can see that using all nodes to perform computation is the right choice for this application.

Our goal in our second series of tests was to explore the performance implications of varying the number of I/O and compute nodes used. Figure 10 shows execution times for combinations of I/O and compute nodes on a 6K $\times$ 6K problem using the MDBI algorithm. Overlapping sets of nodes were used in all cases. There are two important characteristics of this system that are highlighted by this graph. First, the use of large numbers of I/O nodes when few compute nodes are used is extremely inefficient. There is overhead associated with establishing connections and passing control messages to and from each I/O node, and when the number of compute nodes is small this overhead can outweigh the benefits of the increased disk bandwidth. Second, for a given number of available nodes using all nodes for both computation and I/O is the best choice in all cases. This verifies the observations made in the previous set of tests. A more thorough study should be undertaken in order to better characterize the range of applications and problem sizes for which this configuration is optimal.

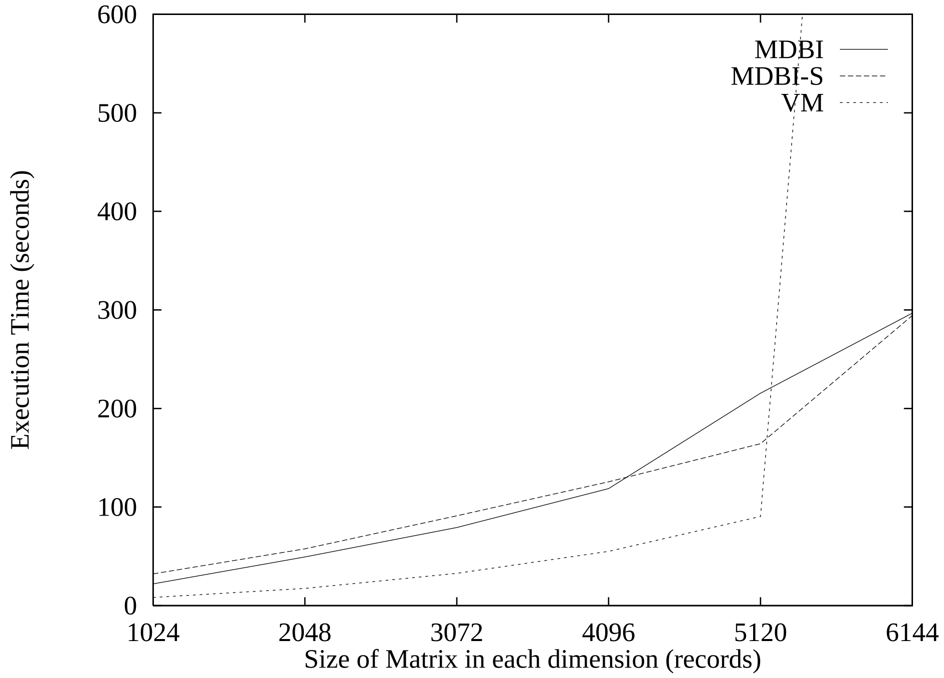

Figure 9: Execution Time vs. Problem Size for 8 Nodes

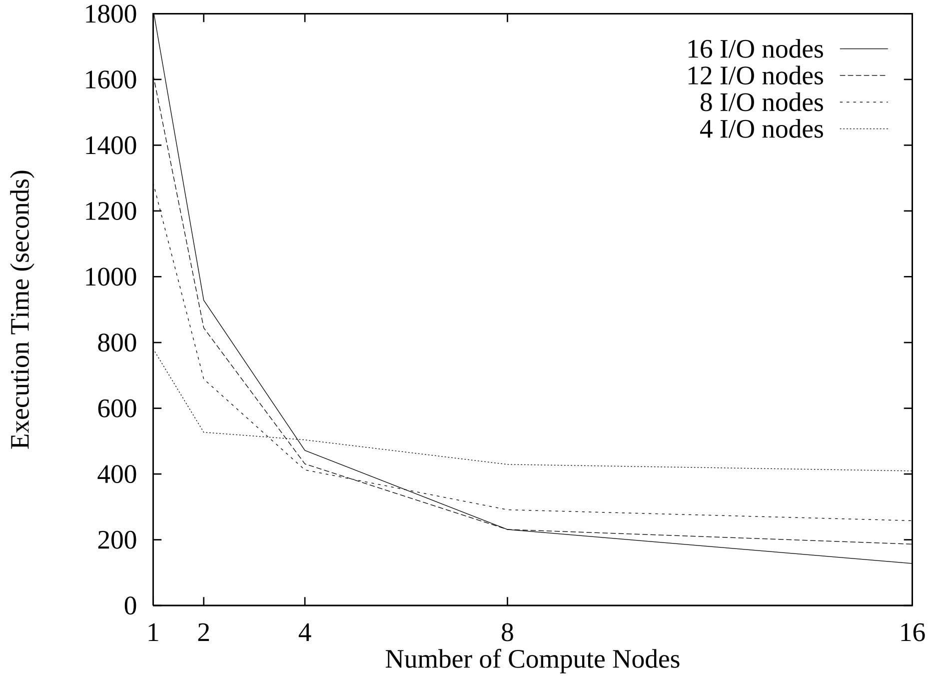

Figure 10: Execution Time vs. Number of Compute Nodes

Table 1: Execution Times for Iterative Solver

Size of A	4 Compute Nodes			8 Compute Nodes			16 Compute Nodes	
	MDBI	MDBI-S	VM	MDBI	MDBI-S	VM	MDBI	VM
1K × 1K	24.8	37.5	14.9	22.0	32.2	8.3	20.5	15.0
2K × 2K	62.1	80.1	53.6	49.4	57.5	17.4	37.1	18.6
3K × 3K	115.4	125.2	124.8	79.2	91.1	32.7	49.2	27.0
4K × 4K	267.2	252.8	1812.3	118.6	125.6	55.1	65.5	32.1
5K × 5K	425.3	375.1	3140.2	215.5	164.2	90.5	87.2	44.4
6K × 6K	501.5	479.7	4698.7	296.7	294.3	1928.3	139.3	57.4
8K × 8K	—	—	—	—	—	—	263.5	1518.8

6. Conclusions

When problem sizes grow to exceed core memory size, the effectiveness of solutions depends on timely access to the portions of data needed by application tasks. While the virtual memory support of many operating systems can enable applications to run out of core, we have shown that in some cases the operating system cannot provide the performance needed for these applications to run efficiently. Our tests show that for a sample application, the combination of a parallel file system and an easy to use interface can give the programmer the power to move data in and out of core in parallel applications in a timely fashion, extending the problem size that can be solved on a given set of resources. While this is only one application, the pattern of data access is common to many applications including telemetry processing, other iterative solving methods, and factorization methods. Thus this technique should be valid for a number of application domains. In addition the penalty for using the OOC method for small problem sizes is not tremendous, indicating that prototyping PVFS OOC applications using small data sets is feasible. However, it is clear that in-core solutions still have an advantage when they are applicable.

What has not been made clear by this study is the extent to which this range has been increased. For the choices of blocking and buffer factors used in our tests, problems of size up to 10K × 10K (1.6GB) could in theory be solved, giving a $IC_{mdbi} = 50$ MB. After this point new blocking and buffer factors would need to be chosen in order to avoid unwanted demand paging on our machines with 64 MB of memory. Machines with larger core memory sizes could tackle much larger problems.

Furthermore, there is much more work to be done in the area of interfaces. Collective I/O in particular has been shown to be of great benefit in many compute platforms. Our work in this area has shown performance improvements in some applications but has uncovered new problems with regards to operating system policies as well. Additional study is needed to fully characterize the implications of interacting with today's operating systems in the most efficient manner.

References

[1] K. Castagnera, D. Cheng, R. Fatoohi, E. Hook, B. Kramer, C. Manning, J. Musch, C. Niggley, W. Saphir, D. Sheppard, M. Smith, I. Stockdale, S. Welch, R. Williams, and D. Yip, "Clustered workstations and their potential role as high speed compute processors," Tech. Rep. RNS-94-003, NAS Systems Division, NASA Ames Research Center, April 1994.

[2] D. Ridge, D. Becker, P. Merkey, and T. Sterling, "Beowulf: Harnessing the power of parallelism in a pile-of-pcs," in *Proceedings of the 1997 IEEE Aerospace Conference*, 1997.

[3] J. Salmon and M. Warren, "Parallel out-of-core methods for N-body simulation," in *Proceedings of the Eighth SIAM Conference on Parallel Processing for Scientific Computing*, 1997.

[4] J. M. del Rosario, R. Bordawekar, and A. Choudhary, "Improved parallel I/O via a two-phase run-time access strategy," in *Proceedings of the IPPS '93 Workshop on Input/Output in Parallel Computer Systems*, (Newport Beach, CA), pp. 56–70, 1993. Also published in Computer Architecture News 21(5), December 1993, pages 31–38.

[5] D. Kotz, "Disk-directed I/O for MIMD multiprocessors," *ACM Transactions on Computer Systems*, vol. 15, pp. 41–74, February 1997.

[6] N. Nieuwejaar and D. Kotz, "Low-level interfaces for high-level parallel I/O," in *Input/Output in Parallel and Distributed Computer Systems* (R. Jain, J. Werth, and J. C. Browne, eds.), vol. 362 of *The Kluwer International Series in Engineering and Computer Science*, ch. 9, pp. 205–223, Kluwer Academic Publishers, 1996.

[7] P. F. Corbett and D. G. Feitelson, "The Vesta parallel file system," *ACM Transactions on Computer Systems*, vol. 14, pp. 225–264, August 1996.

[8] K. E. Seamons and M. Winslett, "An efficient abstract interface for multidimensional array I/O," in *Proceedings of Supercomputing '94*, (Washington, DC), pp. 650–659, IEEE Computer Society Press, November 1994.

[9] P. Corbett, D. Feitelson, S. Fineberg, Y. Hsu, B. Nitzberg, J.-P. Prost, M. Snir, B. Traversat, and P. Wong, "Overview of the MPI-IO parallel I/O interface," in *Input/Output in Parallel and Distributed Computer Systems* (R. Jain, J. Werth, and J. C. Browne, eds.), vol. 362 of *The Kluwer International Series in Engineering and Computer Science*, ch. 5, pp. 127–146, Kluwer Academic Publishers, 1996.

[10] R. Thakur, A. Choudhary, R. Bordawekar, S. More, and S. Kuditipudi, "Passion: Optimized I/O for parallel applications," *IEEE Computer*, vol. 29, pp. 70–78, June 1996.

[11] W. B. Ligon and R. B. Ross, "Implementation and performance of a parallel file system for high performance distributed applications," in *Proceedings of the Fifth IEEE International Symposium on High Performance Distributed Computing*, pp. 471–480, IEEE Computer Society Press, August 1996.

[12] R. Bordawekar, A. Choudhary, and J. M. D. Rosario, "An experimental performance evaluation of Touchstone Delta Concurrent File System," in *Proceedings of the 7th ACM International Conference on Supercomputing*, pp. 367–376, ACM Press, 1993.

For more information and source code for PVFS, see http://ece.clemson.edu/parl/pvfs/index.html. More information on PVM is available at http://www.epm.ornl.gov/pvm/pvm_home.html, and more information on MPI can be found at http://www.mcs.anl.gov/mpi/index.html.

Matthew Cettei *is a student at Clemson University working towards his Master's degree in Computer Engineering. His current interests are in parallel file systems and parallel sorting algorithms.*

Walter Ligon *received his Ph. D. in Computer Science from the Georgia Institute of Technology in 1992. Since then he has been at Clemson University where he is an Assistant Professor in the Department of Electrical and Computer Engineering. His current research interests are in parallel and distributed systems, I/O for parallel systems, reconfigurable computing, and problem solving environments.*

Robert Ross *received his B.S. in Computer Engineering from Clemson University in 1994. Currently he is working towards his Ph. D. in Computer Engineering at Clemson University. His interests include parallel file systems, scheduling algorithms, and visualization.*