Thread Scheduling for Out-of-Core Applications with Memory Server on Multicomputers

Yuanyuan Zhou, Limin Wang, Douglas W. Clark and Kai Li
Computer Science Department
Princeton University
Princeton, NJ 08544
{yzhou, lmwang, doug, li}@cs.princeton.edu

Abstract

Out-of-core applications perform poorly in paged virtual memory (VM) systems because demand paging involves slow disk I/O accesses. Much research has been done on reducing the I/O overhead in such applications by either reducing the number of I/Os or lowering the cost of each I/O operation.

In this paper, we investigate a method that combines fine-grained threading with a memory server model to improve the performance of out-of-core applications on multicomputers. The memory server model decreases the average cost of I/O operations by paging to remote memory, while the fine-grained thread scheduling reduces the number of I/O accesses by improving the data locality of applications. We have evaluated this method on an Intel Paragon with 7 applications. Our results show that the memory server system performs better than the VM disk paging by a factor of 5 for sequential applications and a factor of 1.5 to 2.2 for parallel applications. The fine-grained threading alone improves the VM disk paging performance by a factor of 10 and 1.2 to 3 respectively for sequential and parallel applications. Overall, the combination of these two techniques outperforms the VM disk paging by more than a factor of 12 for sequential applications and a factor of 3 to 6 for parallel applications.

1 Introduction

Out-of-core applications involve data sets that are too large to fit in main memory. Many astrophysics modeling, computational biology and engineering problems are examples of such applications [6, 7, 19]. For these applications, main memory simply serves as a small cache in the memory hierarchy, and the bulk of the data must reside on disks or other secondary storage. There are three categories of out-of-core applications: sequential applications, parallel applications, and message-passing parallel applications with uneven data partitioning. The last one is viewed as a separate category because its memory requirements on some nodes are greater than physical memory sizes, even though the total data set size does not exceed the total amount of memory.

The simplest way to solve the out-of-core problem is to run the original in-core program with the increased problem size, using underlying virtual memory (VM) support for demand paging. Although this approach requires no effort from application programmers, it typically has poor performance due to slow disk accesses, which are several orders of magnitude slower than memory accesses.

There are two approaches to reducing the I/O overhead in out-of-core applications. The first is to reduce the number of I/O accesses, which can be achieved by applications, compilers or runtime systems. Some scientific researchers use explicit I/O calls rather than transparent virtual memory paging because the application has better knowledge of its own data locality and reference pattern. However, this approach usually requires significant restructuring of the code, which can be a formidable task. Some compilers can reorder instructions to improve data locality [2, 13] or insert explicit I/O calls or prefetches into application codes [5, 14, 16, 22, 15, 12]. Most of these compiling techniques are limited by the alias analysis problem and therefore only useful for certain array codes.

The second approach is to reduce the cost of each I/O operation. *Memory server* is one such example. This technique extends the multicomputer memory hierarchy by using remote memory servers as distributed caches for disk backing stores [10]. It has been demonstrated that the memory server model can improve the performance of some out-of-core sequential applications. However, [10] has several limitations. First, it uses dedicated memory server nodes to service memory requests from the computation nodes. Second, for some applications with poor data locality, the paging overhead in the memory server system is still significantly high, up to 50% of the total execution time. This overhead can be reduced by improving applications' data locality. The third limitation is that no results have been shown with the memory server model for message-passing parallel applications.

A recent study has proposed a method to improve the cache locality of sequential programs by scheduling fine-grained threads [17]. The scheduling algorithm relies upon hints provided at the time of thread creation to determine thread execution order to improve data locality. This method can effectively reduce second level cache misses and consequently improve the performance of some untiled sequential applications. But their study can only deal with independent threads.

In this paper, we extend the fine-grained thread scheduling approach to handle thread dependencies and apply it in memory servers to reduce the capacity page misses for our-of-core applications. We also extend the memory server model to support both dedicated and non-dedicated memory server nodes. We have evaluated our method on an Intel Paragon with 7 applications. Our results show that the combination of these two techniques outperforms the traditional virtual memory disk paging by more than an order of magnitude for sequential applications and a factor of 3 to 6 for parallel applications.

2 Fine-grained Thread Scheduling

The fine-grained thread scheduling was originally proposed to improve data locality for caches [17]. Their results show that this method can significantly improve performance by reducing second-level cache misses. However, the proposed method is limited to threads that are independent of each other. In this paper, we extend the thread scheduling approach to handle dependent threads, and apply it in the memory server to reduce capacity page misses for out-of-core applications.

2.1 Original Idea

The original idea of fine-grained thread scheduling is to decompose a program into fine-grained threads and schedule these threads so as to improve the program's data locality. With fine-grained threads, for example, one can substitute a thread for the innermost loop of a program that may be causing second-level cache misses. The run-time thread scheduler determines an execution order with good data locality.

We borrow the example in paper [17] to illustrate the thread scheduling scheme. The example is a matrix multiply of two n by n matrices A and B with the result put in matrix C. In order to improve locality, B is transposed before and after the computation. One straightforward implementation uses the following nested loops:

for $i = 1$ **to** n
  for $j = 1$ **to** n
    $C[i,j] = 0;$
    for $k = 1$ **to** n
      $C[i,j] = C[i,j] + A[i,k] * B[j,k];$

The innermost loop computes the dot product of two n-element vectors. Fine-grained threading simply replaces the dot-product loop with a thread as following

for $i = 1$ **to** n
  for $j = 1$ **to** n
    $th = Fork(DotProduct, i, j, A[i], B[j]);$
$RunThreads();$

$DotProduct(i,j):$
  $C[i,j] = 0;$
  for $k = 1$ **to** n
    $C[i,j] = C[i,j] + A[i,k] * B[j,k];$

where $Fork$ creates a thread that will compute the specified dot product and returns a thread ID for the create thread, and $RunThreads$ runs each thread in some order determined by the scheduling algorithm. The last two arguments in the $Fork$ interface are reference hints for the thread system. These hints are used by the scheduler to approximately measure the cache affinity among threads.

Thread t_i is denoted by $t_i(a_{i1}, \ldots, a_{ik})$ where a_{ij} is the address of the j^{th} piece of data referenced by thread t_i during its execution. Thus, if n threads are executed in some order $t_1, t_2, \ldots, t_n$, they can be represented by the permutation:

$$\begin{array}{l} t_1(a_{11}, \ldots, a_{1k}), \\ t_2(a_{21}, \ldots, a_{2k}), \\ \quad\vdots \\ t_n(a_{n1}, \ldots, a_{nk}). \end{array}$$

The goal in scheduling n threads for cache locality is to find a permutation that minimizes the number of cache misses. We can view such a problem as a k-dimensional geometry problem [3]. A thread $t_i(a_{i1}, \cdots, a_{ik})$ is a point in the k-dimensional space where the coordinates of the point are $(a_{i1}, \cdots, a_{ik})$. The problem is then equivalent to finding a tour of the thread points in the space that satisfies the requirement of minimizing cache misses.

2.2 Original Thread Scheduling Algorithm

The main idea of the original thread scheduling algorithm is to schedule threads in "bins" so that when the threads in a bin are run they will not cause many capacity cache misses. The address space is first divided into blocks. The block size is set to be the cache size (main memory size in our case) divided by the number of reference hints. Threads accessing the same blocks are grouped into the same *bin*. When threads in a bin are run one after another, they will not cause many capacity misses. For each non-empty bin, the scheduler runs all threads it contains one after another until it becomes empty.

With two address hints, we constrain the general scheduling problem. Intuitively, the two hints might be the two largest objects referenced by the thread or the two objects most frequently referenced by the thread. Thus constrained, the scheduling problem then becomes finding a tour of points in a two-dimensional plane as shown in Figure 1, where each thread is represented as a point in the plane with coordinates defined by the two address hints. To minimize cache misses, the scheduling algorithm must find a tour that has a “cluster” property, i.e., threads that have the same or similar hints should be clustered together in the tour.

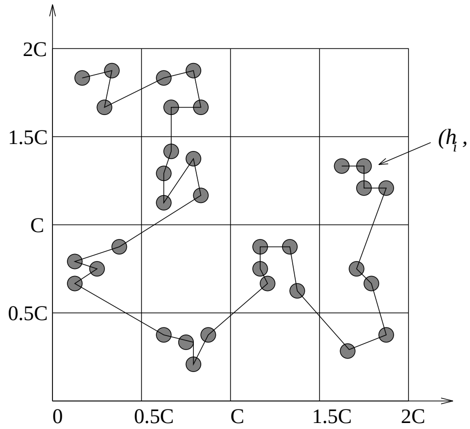

Figure 1: Scheduling threads in a 2-D plane.

To show how the thread scheduler works, let us consider a simple example of a 4×4 matrix multiplication ($C = A \times B$) program using the thread interface presented in section 2.1. The inner-loop forks off threads, each of which computes a dot product, in the following order:

$$t_1(a_1, b_1), t_2(a_1, b_2), t_3(a_1, b_3), t_4(a_1, b_4),$$
$$t_5(a_2, b_1), t_6(a_2, b_2), t_7(a_2, b_3), t_8(a_2, b_4),$$
$$t_9(a_3, b_1), t_{10}(a_3, b_2), t_{11}(a_3, b_3), t_{12}(a_3, b_4),$$
$$t_{13}(a_4, b_1), t_{14}(a_4, b_2), t_{15}(a_4, b_3), t_{16}(a_4, b_4)$$

where a_i and b_j are the i-th vector of matrix A and j-th vector of matrix B respectively.

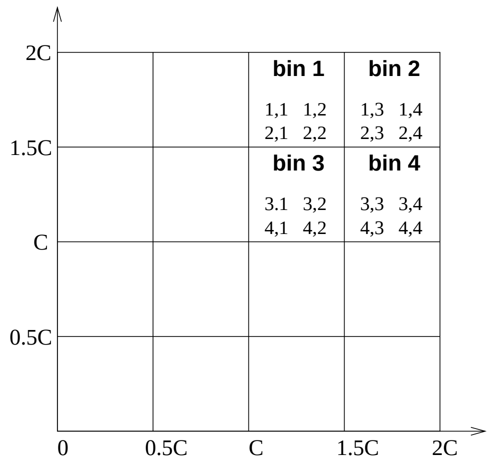

Figure 2: An example of scheduling threads. Each i, j pair represents a thread that computes the dot product for a_i and b_j.

Suppose the cache holds only four vectors and the dimension size of a block in the 2-D scheduling plane is one half the cache (or main memory) size. Then in one possible mapping threads fall into four blocks of the schedule plane, as shown in Figure 2.

The scheduler schedules all threads in each bin before moving to the next bin. The scheduler thus reorders the dot-products in such a way that the execution of the threads in each bin will not cause capacity cache misses. The scheduling order exhibits similar locality to that provided by course grain tiling either by hand or by a compiler.

Since the original study assumes independent threads, the implementation of the thread scheduler is relatively simple. All threads can be scheduled at fork time by hashing reference hints into bins, which is organized as a hash table. Once all threads are forked, *RunThreads* simply traverses all non-empty bins executing threads one after another. Further detail about the original scheduler can be found in [17].

2.3 Limitations of the Original Idea

The original approach has two limitations. First, it assumes that all threads are independent. This assumption only holds for a limited amount of applications. In some applications, a thread cannot start until another thread completes. For example, a thread t has to be executed after another thread t' is done if t needs to read the data produced by t' or t overwrites the data needed by t'. Many applications such as matrix factorization have data dependencies between loop iterations. The original approach does not work with such applications since it cannot support dependencies among threads.

The second limitation of the original approach is its bin traversing scheme. Although any bin traversal scheme always yields correct results, it may have different performance. Because the original scheduling algorithm uses a hash scheme, the bin traversing sequence is very random. The main drawback of this random traversing scheme is that it cannot reuse any data that are brought into the cache (or main memory) by preceding bins. A better alternative is to visit bins that share some reference hints one after another. In the above 4×4 matrix multiplication example, the traversing sequence **bin 1, bin 2, bin 3, bin 4** has fewer cache (or page) misses than the sequence **bin 1, bin 4, bin 2, bin 3**. When **bin 2** follows **bin 1**, vectors a_1 and a_2 are still in the cache (or main memory).

2.4 Thread Dependencies

We extend the original thread system to support applications with thread dependencies.

Consider a simplified Gaussian elimination of matrix A:

for $i = 1$ **to** n
  for $j = i + 1$ **to** n (1)
    $a[i, j] = a[i, j]/a[i, i];$
  for $j = i + 1$ **to** n
    for $k = i + 1$ **to** n (2)

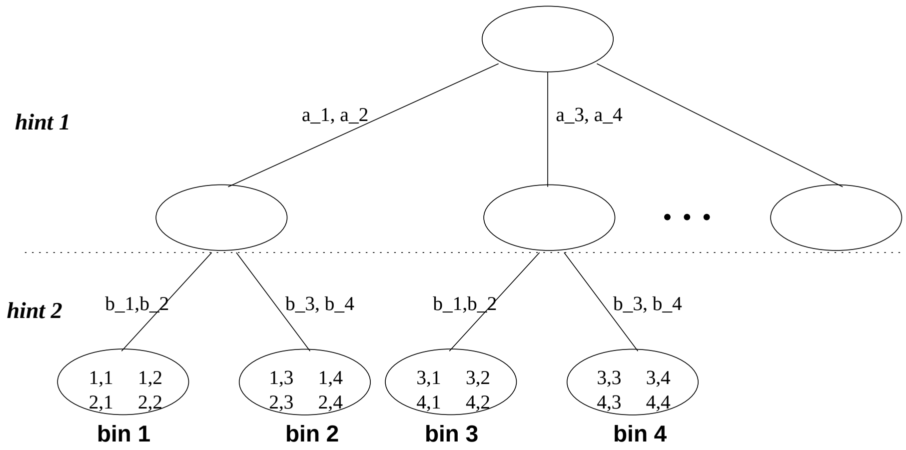

Figure 3: Scheduling tree (Only ready threads are inserted into the tree. Edges represents address hints and ovals represents bins, which are traversed according to the tree nodes traversal algorithm).

$$a[j, k] = a[j, k] - a[i, k] * a[j, k];$$

Just as in the matrix multiplication example, we can turn inner loops (1) and (2) into threads:

$$
\begin{array}{l}
\textbf{for } i = 1 \textbf{ to } n \\
\quad th_{i,i} = Fork(Pivot, i, a[i]); \\
\quad \textbf{for } j = i + 1 \textbf{ to } n \\
\quad\quad th_{j,i} = Fork(Compute, i, j, a[i], a[j]); \\
RunThreads();
\end{array}
$$

$$
\begin{array}{l}
Pivot(i): \\
\textbf{for } j = i + 1 \textbf{ to } n \\
\quad a[i, j] = a[i, j]/a[i, i];
\end{array}
$$

$$
\begin{array}{l}
Compute(i, j): \\
\textbf{for } k = i + 1 \textbf{ to } n \\
\quad a[j, k] = a[j, k] - a[i, k] * a[j, k];
\end{array}
$$

where $Pivot()$ replaces inner loop (1) and $Compute()$ replaces inner loop (2).

In contrast to matrix multiplication, this application has thread dependencies. For example, thread $th_{j,i}$ depends on thread $th_{i,i}$ and $th_{j,i-1}$ since it needs to use the ith and jth row, which are produced by thread $th_{i,i}$ and $th_{j,i-1}$, respectively. Similarly, thread $th_{i,i}$ has to execute after thread $th_{i,i-1}$. In order to guarantee the correct execution order of threads, we need to specify dependency relations at thread creation time so the thread system can schedule threads according to the dependency specification. One way to achieve this is to extend the $Fork$ interface to allow applications to specify dependencies. The threaded code for Gaussian Elimination is then changed to:

$$
\begin{array}{l}
\textbf{for } i = 1 \textbf{ to } n \\
\quad th_{i,i} = Fork(Pivot, i, a[i], th_{i,i-1}); \\
\quad \textbf{for } j = i + 1 \textbf{ to } n \\
\quad\quad th_{j,i} = Fork(Compute, i, j, a[i], a[j], \\
\quad\quad\quad th_{i,i}, th_{j,i-1}); \\
RunThreads();
\end{array}
$$

2.5 Scheduling with Dependencies

Once an application has specified thread dependencies, the challenge is to design a scheduling algorithm that improves the program's data locality without violating thread dependencies. In the original study, fine-grained threads are used to reduce capacity cache misses, which cost too little to justify any sophisticated scheduling algorithms. However, since our study uses fine-grained threading to reduce capacity *page* misses, a relative complicated scheduling scheme can be used for the sake of better performance.

The first modification to the original scheduling algorithm is to schedule threads at run time rather than at fork time. A ready set is introduced to record all ready threads. A thread is ready if and only if all threads it depends on have been executed. Threads in the ready set can be executed in any order without violating the correctness of the program. Each thread has a dependency counter. The dependency counter for a thread is decremented by one when one of its dependencies completes. When the dependency counter reaches zero, the thread will be put into the ready set.

The second extension is a better bin traversal scheme that takes advantage of the data affinity among bins. Our scheduler uses a fat tree rather than a simple hash table as the main data structure. Each node in the tree represents a bin. The tree is organized in a way such that siblings share some reference hints. The main advantage for this tree scheme is that when the scheduler uses the tree traversing algorithm to move from one bin to another, some data in the cache (or main memory) can be reused. In addition, the tree structure makes it more efficient and less memory consuming to support scheduling with thread dependencies. This scheme can also easily support variable number of reference hints.

Figure 3 shows the tree structure of the 4×4 matrix multiplication example. The scheduling tree contains only threads that are ready to execute. Once a thread's dependency counter reaches zero, it is inserted into the tree with address hints as indices. The depth of the tree equals the

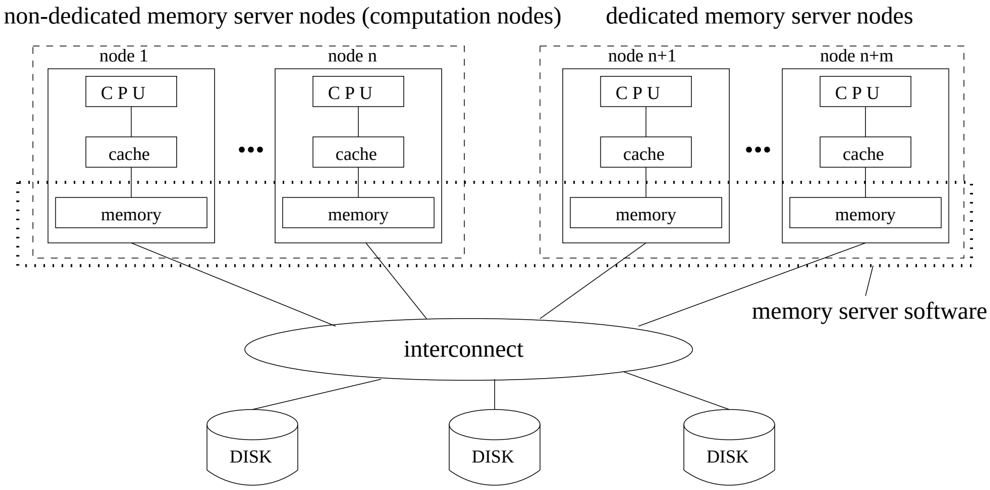

Figure 4: The Memory Server Model

number of address hints. Nodes are allocated on demand to reduce memory overhead. The scheduler traverses the whole tree to move from one bin to another bin. The node traversing sequence gives the bin scheduling sequence to be **bin 1, bin 2, bin 3, bin 4**.

3 Memory Server Model

Memory server is a virtual memory management technique that can be used to reduce the number of disk accesses for out-of-core applications on multicomputers [10]. It has been demonstrated that the memory server model outperforms the traditional virtual memory disk paging by a factor of 3 for some out-of-core sequential applications with good data locality.

In this paper, we assume large-scale distributed memory multicomputers as the architectural framework. Each node consists of a processor, a local memory unit, and a network interface. Nodes in the system are interconnected through a scalable, high bandwidth routing network. Disks are attached only to some nodes.

Our memory server model supports both dedicated and non-dedicated memory server nodes. Dedicated memory server nodes are only used for serving remote memory requests. Non-dedicated memory server nodes are computation nodes at the same time. In other words, all the physical memory in all computation nodes is managed together as a whole. By doing this, we can reduce the number of disk accesses for message-passing parallel applications with uneven data partitioning. Figure 4 shows an example of our memory server model with n computation nodes (non-dedicated memory server nodes) and m dedicated memory server nodes.

When a computation node exhausts its local memory, it swaps out some victim pages to other nodes. Since each computation node is also a memory server node, the physical memory at each node may contain both local and remote data. A simple replacement policy is always to replace remote pages first. When the amount of free memory is below some threshold, the oldest remote page will be returned back to the owner of that page, and the owner can send this page to some other node or disk. When no remote page is present, a local page is chosen to be the victim according to some approximate LRU algorithm.

One alternative replacement strategy is global LRU among both local and remote pages. This method can reduce the total number of disk accesses at the cost of maintaining some global timing information about memory references. Since this approach would involve operating system support to collect memory reference history information, we use the first replacement policy in our prototype implementation.

On a page miss, the computation node fetches the missing page from a remote node or a disk. Each node has a page table that records the status and current locations for its own pages. After the missing page is fetched back to its owner, it is deleted from the remote memory. In our memory server model, there is always at most one copy in memory for any page. The main advantage of this policy is that it can save memory space. The drawback is that clean pages also need to be transferred to new locations when swapping.

In general, the memory server model can reduce the average page miss penalty. This can be explained from two aspects. First, for applications with reasonable temporal locality, most of paging in the memory server model is from remote memory rather than disks. Although a node's non-resident pages can be in either remote memory or disks, the memory server replacement policy guarantees that pages in disks are much older than those in remote memory. Therefore, if the application has good locality, it is more likely to fetch pages from remote memory than from disks. Second, fetching a page from remote memory is two orders of magnitude faster than paging from disks. For example, transferring an 8-Kbyte page from a remote memory takes about 100 microseconds on the Intel Paragon, whereas reading a page from a disk takes around 10 milliseconds. As a result of these two reasons, memory server has significantly smaller average page miss penalty than VM disk paging.

4 Implementation

To evaluate our method of combining fine-grained threading with memory server, we need a platform that can support both sequential and message passing parallel applications. The Intel paragon exactly matches our requirement [11]. The memory server model implementation requires a small modification to the kernel, while the thread system runs at user level.

4.1 Experiment Environment

The Intel Paragon multicomputer used in our experiments consists of 10 nodes for computation. Each node has one compute processor and a communication co-processor. Both processors are 50 MHz i860 microprocessors with 16 Kbytes of data cache and 16 Kbytes of instruction cache [11]. The two processors share 64 Mbytes of local memory, with around 56 Mbytes memory available to user applications. The memory bus provides a peak bandwidth of 400 MBytes/sec. The nodes are interconnected with a wormhole routed 2-D mesh network whose peak bandwidth is 200 Mbytes/sec per link [23].

The operating system is a micro-kernel based version of OSF/1 with multicomputer extensions for a parallel programming model and the NX/2 message passing primitives. The co-processor runs exclusively in kernel mode, and it is dedicated to communication. The one-way message-passing latency of a 4-byte NX/2 message on the Paragon is about 50 μsec [18]. The transfer bandwidth for large messages depends on data alignment. When data are aligned properly, the achievable maximal bandwidth at user level is 175 Mbytes/sec. Without proper alignment, the peak bandwidth is about 45 Mbytes/sec.

4.2 Thread System Implementation

The implementation of the thread system is straightforward for sequential applications. Applications are manually modified to create millions of small threads. The thread scheduler runs at user level. Applications' virtual address space is divided into 16-Mbyte blocks. In order to reduce the thread system memory overhead, the scheduler is triggered to start running threads when the number of created threads has reached some threshold.

For parallel applications using NX/2 message passing primitives [18], our thread system faces the challenge of supporting communication among nodes while improving data locality. A reordering of communication events that satisfies the local thread dependencies may still generate incorrect results or deadlocks. For example, the following code

Node 0	Node 2
`send(TYPE1, ...);`	send(TYPE1, ...);
`recv(TYPE2, ...);`	recv(TYPE2, ...);

can proceed normally. But, if the *send* and *recv* are scheduled in a switched order on node 0, and the code on node 1 remains unchanged, a deadlock will occur.

Our solution is to also treat *send* and *recv* pairs on different nodes as dependencies. The *recv* thread cannot execute until the corresponding *send* thread on another node completes. This method can only reorder local threads but does not require global coordination for scheduling. The main drawback of this method is that it does not consider parallelism. Sometimes, it is better to sacrifice data locality to schedule a *send* thread early so other nodes waiting for the message can proceed.

	fork	run	total
time (μs)	0.013	0.006	0.02

Table 1: Thread overhead in microseconds.

Table 1 reports the overhead to fork and run a null thread on the Intel Paragon. The thread overhead is calculated using a simple loop that creates one million threads to call the null procedure and then runs them. The threads' references are evenly distributed across the address space. The total overhead for one thread is only 0.02 milliseconds, which is two orders of magnitude less than the page miss penalty in most of paging systems. This indicates that the overhead can be easily justified by eliminating just one page miss for each thread.

4.3 Memory Server Implementation

To implement the memory server model on the Intel Paragon, one straightforward approach is to use the external memory management (EMM) mechanism in OSF/1 (derived from Mach 3.0). The advantage of this approach is that the implementation can be done at user level. However, our prototype implementation using EMM shows that this method is very inefficient due to the high operating system overhead. Another approach to implement the memory server model is to change the VM system of the Mach kernel to page to/from remote memory rather than disk. But this approach is hard to debug and has poor portability. Therefore, we implemented the memory server model at user level by adding a new system call to deal with virtual to physical page mapping.

	VM disk paging	EMM	MS
time (ms)	13	6.5	1.4

Table 3: The average page fault time.

Physical memory is managed by the memory server library. To do this at user level, the memory server library needs to

- catch page faults. When a page is swapped out to remote memory or disks, its access protection is turned off. The first access to this page after it swaps out triggers a page fault. The page fault handler fetches the

	handler invoking	request transfer	context switch	page transfer	page protection	total
time (ms)	0.3	0.05	0.7	0.1	0.2	1.4

Table 2: The breakdown for handling a page fault in MS

missing page from remote memory or disks, changes its protection and restarts the faulting instruction.

- pin all memory. This is needed to avoid swapping by the VM system.
- map virtual pages to physical frames. When the local memory is exhausted, the memory sever system swaps out a page and gives its physical frame to the faulted page. This can be easily achieved in a system that supports memory mapped files. Due to the limitation of the Paragon system, we have to add one system call $remap\_addr(addr1, addr2)$ to the Mach kernel. This system call maps $addr1$'s physical frame to $addr2$.

The memory server system uses NX messages to transfer requests and pages. All computation nodes run two kernel threads, one for application computation and one for serving remote memory requests.

Table 3 compares the average page fault handling time with virtual memory management (VM) disk paging, the memory server implementation using external memory management (EMM), and the memory server implementation with small modifications to the kernel (MS). The MS implementation has the fastest average page fault handling time, 8 times faster than VM disk paging and 4 times faster than EMM.

Table 2 shows the time breakdown for handling a page fault in the MS implementation. Transferring a page only costs 10% of the total page fault handling time, while operating system related overhead takes other 90%. This indicates that the memory server performance can be improved significantly by reducing the operating system overhead.

5 Performance

To evaluate the performance of our implementation, we have used three sequential applications (Matrix multiply, Gaussian elimination and Cholesky factorization), the parallel versions of these three applications and one parallel application with uneven data partition (N-Body).

Table 4 gives the problem sizes, data set sizes and execution time with MS for the seven applications. We choose the problem sizes large enough so that the whole data sets cannot fit in physical memory. All applications take several hours to several days to execute with MS.

5.1 Sequential Applications

Figure 6 shows the performance and time breakdown for three sequential applications with the traditional VM disk paging (VM), our memory server implementation (MS), VM

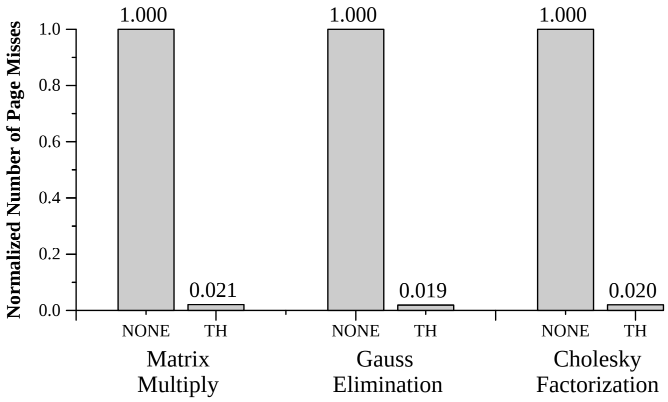

Figure 5: Number of page misses with sequential applications.

disk paging with threading (TH-VM) and MS with threading (TH-MS). All the execution time is normalized to that with VM disk paging. The MS performance is measured using 8 dedicated server nodes. Because it takes more than 10 days to run the non-threaded versions of our applications with VM disk paging, we estimate the performance with VM using the following formula:

$$\#misses(MS) \times missPenalty(VM)$$

The assumption made here is that VM will generate the same number of page misses as the memory server. This is true unless the application is very non-deterministic. Numbers with MS, TH-VM and TH-MS are measured with experiments.

The memory server model performs dramatically better than VM with all three sequential applications. As we might expect, most of the execution time with VM is spent on paging to/from disks. Figure 6 shows that the VM paging overhead takes more than 97% of the total execution time. MS reduces this overhead by a factor 7 for the three applications because on average MS handles a page miss 8 times faster than VM (Table 3). As a result, MS outperforms VM by more than a factor of 5.

Fine-grained threading also significantly improves performance. As shown in Figure 5, the threaded versions (TH-VM and TH-MS) have 97% fewer page misses than the non-threaded versions(VM and MS). As a result, fine-grained threading alone reduces the paging time with VM by a factor of 20 to 40. The overhead of the thread system is less than 25% of the execution time for Gaussian and Cholesky. For matrix multiply, the overhead of threading is negligible because this application has no thread dependencies. In addition, the fine-grained threading also effectively reduces the number of cache and TLB misses for matrix multiply. This

applications		problem size	data set size (Mbytes)	execution time (s)
sequential applications	matrix multiply	4,096 x 4,096	384	162,364.4
	Gaussian elimination	4,096 x 4,096	128	58,864.8
	Cholesky factorization	4,096 x 4,096	128	28,256.7
parallel applications	matrix multiply	8,192 x 8,192	1,536	514,756.8
	Gaussian elimination	8,192 x 8,192	512	202,169.6
	Cholesky factorization	8,192 x 8,192	512	123,143.3
irregular applications	N-Body	80,000 bodies 30 iterations	300 total 70 max,28 min	19,640.1

Table 4: The problem sizes, data set sizes and execution time with MS for seven applications.

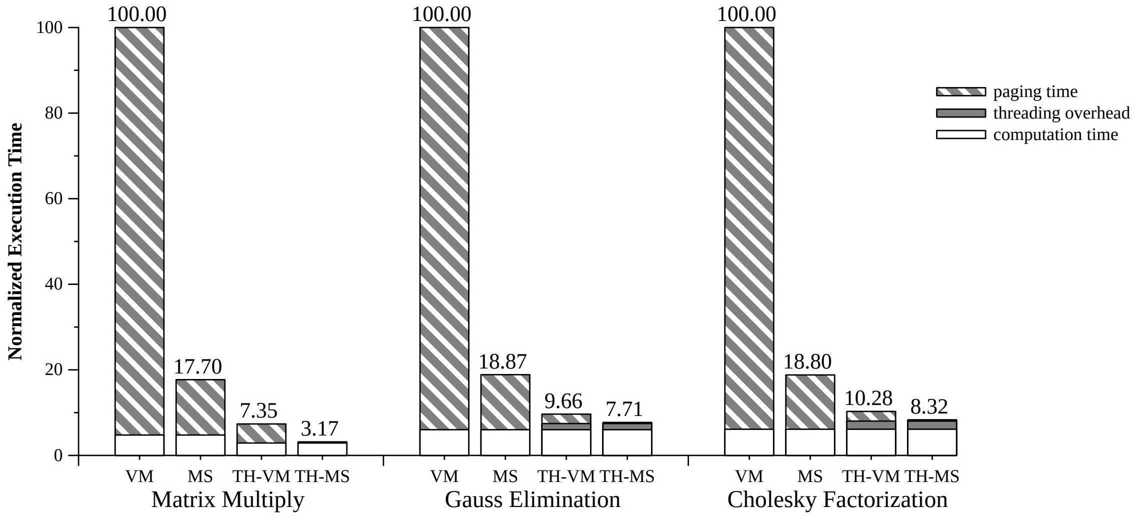

Figure 6: Sequential applications performance (VM stands for the traditional virtual memory disk paging, MS for our memory server implementation, TH-VM for fine-grained threading with VM disk paging and TH-MS for fine-grained threading with memory servers.

is indicated by the decreased computation time with TH-VM and TH-MS. Consequently, fine-grained threading improves VM performance by more than an order of magnitude.

Overall, the combination of fine-grained threading and memory server performs 12 to 30 times better than traditional VM. The paging overhead is reduced from 97% with VM to less than 10% of the total execution time.

5.2 Parallel Applications

Our experiments use the message passing code of the three sequential applications. The matrix multiply application partitions the result matrix in a row-wise fashion. The other two applications are parallelized by partitioning the matrix by interleaving row-wise. All three applications have thread dependencies. The experiments run on 8 computation nodes and 2 dedicated memory server nodes.

The combination of MS and fine-grained threading outperforms VM by a factor of 3 to 6. As shown in Figure 7, MS reduces the paging overhead by a factor of 1.7 to 2.5, which results in a factor of 1.5 to 2.2 improvement in overall performance. Although fine-grained threading reduces the average number of page misses per node by a factor of 2 to 20 as shown in Figure 8, it only improves the end performance with VM disk paging by a factor 1.2 to 3. The main reason is the inter-node data dependencies in parallel applications. A page miss on node 1 not only slows down this node execution but also delays the computation of other nodes who are waiting for node 1 to produce and send results.

The performance improvement by the memory server model and fine-grained threading is less pronounced with parallel applications than with sequential applications. The main reason is the serialization effect at dedicated memory server nodes. In our experiments, 2 dedicated memory servers need to service 8 computation nodes. In addition, the fine-grained threading cannot work as effectively with parallel applications as with sequential applications because dependencies among threads on different nodes limit the number of reorderable threads at any one time.

5.3 Irregular Applications

We use the N-Body application to demonstrate that our memory server model can help reduce the paging overhead for irregular applications with uneven data partitioning. We do not have the results with fine-grained threading for this application. But we are in the process of changing the

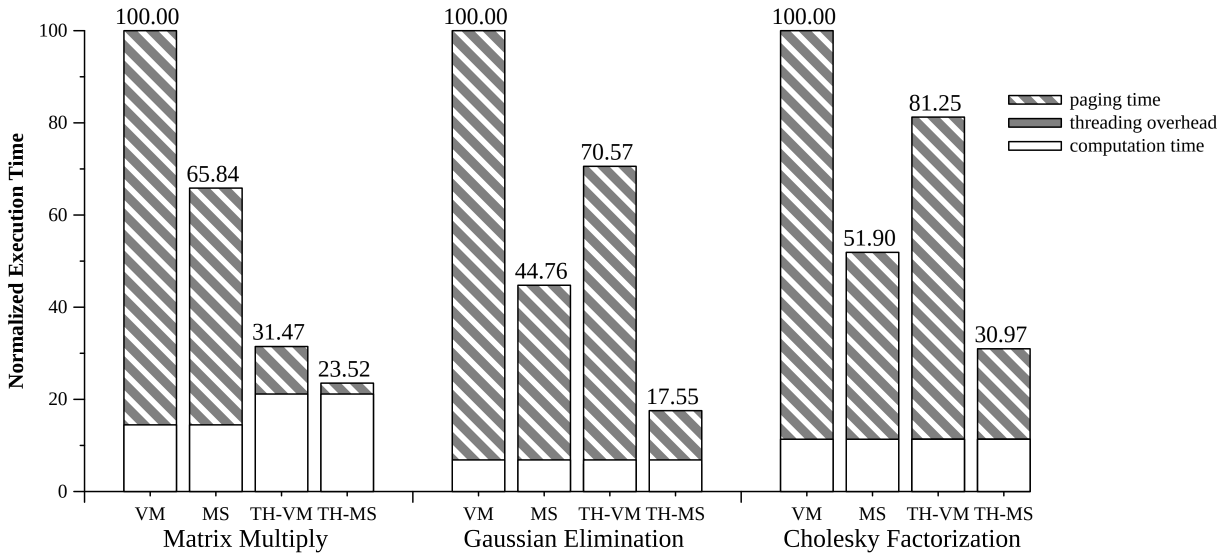

Figure 7: Parallel applications performance.

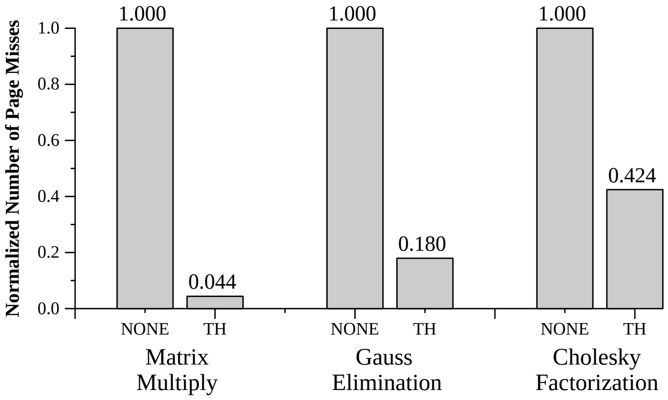

Figure 8: Number of page misses with parallel applications.

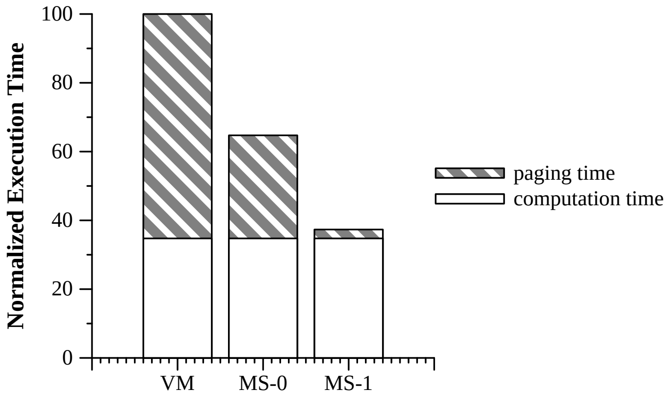

Figure 9: N-Body performance (MS-0 means 8 computation nodes with no dedicated memory server, and MS-1 means 8 computation nodes with 1 dedicated memory server).

application to use fine-grained threads.

The N-Body application is widely used in many areas of science and engineering, including astrophysics, molecular biology and even graphics. It simulates the interactions among system of particles over a number of time steps, using the Barnes-Hut hierarchical N-body method [1]. Our N-Body application is based on Salmon's implementation with message passing [24]. It uses the orthogonal recursive bisection partitioning technique. It is parallelized in a way that the work on each node is well balanced. If all particles have the same mass, all particles are evenly distributed among all nodes. Otherwise, imbalance in data partitioning can occur. A black hole in a galaxy is such an example. Our input simulates such a scenario. The total memory requirement is around 300 Mbytes, and the data partitioning is uneven among 8 nodes, from 28 Mbytes to 70 Mbytes. This causes paging in nodes whose data partitions exceed the physical memory limit, while some other nodes still have more than 20 Mbytes of free physical memory available.

Figure 9 shows the performance of the N-Body application on 8 computation nodes with VM disk paging, memory server with no dedicated server nodes (MS-0) and memory server with one dedicated server node (MS-1). The memory server reduces the paging overhead with VM by a factor of 2 without using extra nodes. As a result, the memory server running on the same number of nodes runs a factor of 1.5 faster than VM. With the memory server, applications' in-core problem sizes are limited by the total amount of memory on all nodes rather than the amount of memory on a single node.

Using one dedicated memory server node (MS-1) substantially outperforms the memory server without dedicated server nodes (MS-0). The paging overhead is reduced from 46% of the execution time in MS-0 to less than 10% in MS-1. With no dedicated server nodes, paging requests to a remote node need to interrupt the computation of this node, while in MS-1, all paging requests are first sent to the dedicated memory server node. Because the context switch and message notification on the Intel Paragon are very expensive, the memory server system overhead and the average page miss penalty with MS-0 are much greater than with MS-1. Consequently, MS-1 runs twice as fast as VM.

6 Related Work

Researchers in the local area network community haven taken a similar approach to the memory server model [4, 9, 21, 8], using the memory available on other nodes as backing store. All these studies use a cluster of workstations and run sequential or distributed applications, while our memory server model runs on a multicomputer for sequential and parallel applications. Therefore, we have different design tradeoffs.

Memory server for multicomputers was first proposed in [10]. They presented the model and discussed several design issues including computation vs. memory server nodes, page mapping and replacement, and caching and prefetching. They also showed preliminary results of a prototype implementation on an Intel iPSC/860 for sequential applications. Our memory server model can support both dedicated and non-dedicated memory server nodes. Our experiments include sequential, parallel and irregular applications. In addition, we use fine-grained threads to reduce the number of page misses.

Much research has been done in the area of improving the performance of out-of-core applications. A lot of scientific researchers working on out-of-core applications typically write a separate version with explicit I/O calls to achieve reasonable performance. Writing an out-of-core version is a formidable task because it often involves significant restructuring of the code, and in some cases can have a negative impact on the numerical stability of the algorithm [25]. The approach presented in this paper provides a potentially simpler alternative. It is conceivable that this technique could be used by a compiler, but that investigation is beyond the scope of this paper.

An alternative application-specific method is to provide a special library for applications to hide the details of reading and writing from disk. For example, "Manual paging" [20] is used in hierarchical tree based applications to increase spatial and temporal locality by controlling the swapping policy in the library. This approach shares some similar design issues with our memory server model. The difference is that this method is usually application-specific, whereas our memory server is more general.

Compiling for out-of-core applications tends to focus on two areas: using various tiling techniques to improve data locality [2, 13] and inserting explicit I/O calls or I/O prefetches into application codes [5, 14, 16, 22, 15, 12]. Most of these compiling techniques are limited by the alias analysis problem and therefore only useful for certain array codes.

Fine-grained thread scheduling was originally proposed in [17]. It was used to reduce second level cache misses. In this paper, we apply the idea in reducing page misses in our memory server system. We also extend the thread system to support thread dependencies.

7 Conclusions and Limitations

This paper describes a method that uses fine-grained threads with the memory server model to reduce the number of disk accesses for out-of-core applications. Our implementation on the Intel Paragon has shown that the memory server mechanism lowers the page miss penalty of the traditional VM disk paging by a factor of 8. The results with six out-of-core applications shows that the thread system effectively reduces the number of page misses by more than 97% for sequential applications and 58-96% for parallel applications. The combination of these two mechanisms outperforms VM disk paging by more than an order of magnitude for sequential applications and a factor of 3 to 6 for parallel applications.

The memory server model can also help parallel applications with uneven data partitioning, such as the N-Body application. Our experiments with this application have shown that with the same number of multicomputer nodes, the memory server system performs 50% faster than the traditional VM disk paging. With one dedicated memory server node, the memory server system runs twice as faster as VM.

Our study has several limitations. Our experiments are conducted on an old platform. However, since network performance is increasing at almost the same speed as processor performance, we expect that our results can still apply to current parallel systems. We have not compared our results with explicit I/O or compiler-inserted I/O versions for out-of-core applications. We have also not compared the fine-grained thread scheduling with manually tiled version. Because of the platform limitation, we are unable to investigate the effectiveness of fine-grained thread scheduling for reducing second level cache misses in addition to memory swapping.

8 Acknowledgement

This work benefitted greatly from discussions the authors had with Tracy J. Kimbrel and Sandy Irani on the thread scheduling algorithm. We are also grateful to Jaswinder Pal Singh for his help with the N-Body application.

This project is sponsored in part by the Scalable I/O Initiative Effort under DARPA grant DABT63-94-0049 and grants from Sandia National Lab and Lawerance Livermore National Lab.

References

[1] Josh Barnes and Piet Hut. A hierarchical $O(N \log N)$ force calculation algorithm. *Nature*, 324:446–449, December 4 1986.

[2] Rajesh Bordawekar, Alok Choudhary, and J. Ramanujam. Automatic Optimization of Communication in Compiling

Out-of-core Stencil Codes. In *Proceedings of the 10th ACM International Conference on Supercomputing*, pages 366–373, Philadelphia, PA, May 1996. ACM Press.

[3] Bernard Chazelle. Private Communication. 1996.

[4] D. Comer and J. Griffoen. A New Design for Distributed Systems: The Remote Memory Model. In *Proceedings of the USENIX Summer Conference*, pages 127–135, Anaheim, California, June 1990.

[5] Thomas H. Cormen and Alex Colvin. ViC*: A Preprocessor for Virtual-Memory C*. Technical Report PCS-TR94-243, Dartmouth College, 1994.

[6] Phyllis E. Crandall, Ruth A. Aydt, Andrew A. Chien, and Daniel A. Reed. Input/Output Characteristics of Scalable Parallel Applications. In *Proceedings of Supercomputing '95*, San Diego, CA, December 1995. IEEE Computer Society Press.

[7] Juan Miguel del Rosario and Alok Choudhary. High Performance I/O for Parallel Computers: Problems and Prospects. *IEEE Computer*, 27(3):59–68, March 1994.

[8] M. J. Feeley, W. E. Morgan, F. H. Pighin, A. R. Karlin, and H. M. Levy. Implementing Global Memory Management in a Workstation Cluster. In *Proc. of the 15th ACM Symp. on Operating Systems Principles (SOSP-15)*, pages 201–212, December 1995.

[9] E. Felten and J. Zahorjan. Issues in the Implementation of a Remote Memory Paging System. Technical Report 91-03-09, University of Washington, March 1991.

[10] Liviu Iftode, Karin Petersen, and Kai Li. Memory Servers for Multicomputers. In *Proceedings of the IEEE Spring COMPCON '93*, pages 538–547, February 1993.

[11] $i860^{TM}$ XP Microprocessor. Programmer's Reference Manual, 1991.

[12] Mahmut Kandemir, Rajesh Bordawekar, and Alok Choudhary. Data Access Reorganizations in Compiling Out-of-Core Data Parallel Programs on Distributed Memory Machines. In *Proceedings of the Eleventh International Parallel Processing Symposium*, pages 559–564, April 1997.

[13] Mahmut Kandemir, Rajesh Bordawekar, Alok Choudhary, and J. Ramanujam. A Unified Tiling Approach for Out-of-Core Computations. In *Sixth Workshop on Compilers for Parallel Computers*, pages 323–334, Aachen, Germany, December 1996. Forschungzentrum Julich GmbH. Also available as Caltech Technical Report CACR 130.

[14] Ken Kennedy, Charles Koelbel, and Mike Paleczny. Scalable I/O for Out-of-Core Structures. Technical Report CRPC-TR93357-S, Center for Research on Parallel Computation, Rice University, November 1993. Updated August, 1994.

[15] Todd C. Mowry, Angela K. Demke, and Orran Krieger. Automatic compiler-inserted I/O prefetching for out-of-core applications. In *Proceedings of the 1996 Symposium on Operating Systems Design and Implementation*, pages 3–17. USENIX Association, October 1996.

[16] Michael Paleczny, Ken Kennedy, and Charles Koelbel. Compiler Support for Out-of-Core Arrays on Data Parallel Machines. In *Proceedings of the Fifth Symposium on the Frontiers of Massively Parallel Computation*, pages 110–118, McLean, VA, February 1995.

[17] James Philbin and et al. Thread Scheduling for Cache Locality. In *Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 60–71, Cambridge, Massachusetts, 1–5 October 1996. ACM Press.

[18] R. Pierce and G. Regnier. The Paragon Implementation of the NX Message Passing Interface. In *Proceedings of the Scalable High-Performance Computing Conference*, May 1994.

[19] James T. Poole. Preliminary Survey of I/O Intensive Applications. Technical Report CCSF-38, Scalable I/O Initiative, Caltech Concurrent Supercomputing Facilities, Caltech, 1994.

[20] John Salmon and Michael Warren. Parallel Out-of-core Methods for N-body Simulation. In *Proceedings of the Eighth SIAM Conference on Parallel Processing for Scientific Computing*, March 1997.

[21] B. N. Schilit and D. Duchamp. Adaptive Remote Paging for Mobile Computers. Technical report, Department of Computer Science, Columbia University, New York, U.S., 1991.

[22] Rajeev Thakur, Rajesh Bordawekar, Alok Choudhary, Ravi Ponnusamy, and Tarvinder Singh. PASSION Runtime Library for Parallel I/O. In *Proceedings of the Scalable Parallel Libraries Conference*, pages 119–128, October 1994.

[23] Roger Traylor and Dave Dunning. Routing Chip Set for Intel Paragon Parallel Supercomputer. In *Proceedings of Hot Chips '92 Symposium*, August 1992.

[24] Michael S. Warren and John K. Salmon. Astrophysical N-Body Simulations Using Hierarchical Data Structures. In *Proceedings Supercomputing '92*, pages 570–576, Minn., MN, November 1992. IEEE.

[25] David Womble, David Greenberg, Rolf Riesen, and Stephen Wheat. Out of Core, Out of Mind: Practical Parallel I/O. In *Proceedings of the Scalable Parallel Libraries Conference*, pages 10–16, Mississippi State University, October 1993.