Out of Core Sorting on Beowulf Class Computers

Matthew M. Cettei
Lucent Technologies
Suite 105
8000 Regency Parkway
Cary, NC 27511
919-388-2656
mcettei@lucent.com

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

October 12, 1998

Abstract

Beowulf class parallel computers have shown impressive performance for specific applications and have become a popular choice for groups who need high performance computing resources on a tight budget. At the same time, it is still unclear how many fundamental applications map to this platform. One such application is that of sorting. More expensive clusters of high-performance workstations using proprietary networks have shown excellent sorting capabilities. However, this application has yet to be explored on Beowulf workstations, especially in the context of data sets larger than core memory. In this paper we study two algorithms for sorting, focusing on their performance on a Beowulf workstation as problem size approaches and exceeds core memory size.

1 Introduction

Sorting is one of the most fundamental applications of computers, as it is required for the many database and storage systems operating today. One observable trend certain to continue is the exponential growth of data set sizes. To keep pace with this trend, the use of parallel machines has also increased. Often sorting requires rearranging large records of data, and even parallel machines can quickly run out of available memory. This paper looks at sorting techniques which can run on data sets larger than the total memory size and their performance on Beowulf-class computers.

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 massively parallel computers [1]. These clusters are often built using existing workstations which are used as interactive systems during the day, 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 that emphasizes dedicated resources and a private system area network for communication [2]. 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). Experiments have shown Beowulf workstations capable of providing high performance for applications in a number of problem domains.

Typical scientific applications that are well suited for execution on parallel machines require large amounts of data. Unfortunately, rapid improvements in processor execution rates have far outstripped the progress of I/O systems, most notably disk access rates. In order to bridge the gap between these rates of progress, new methods of I/O have been developed to take full advantage of the network bandwidth and multiple I/O resources in parallel systems. Parallel disk systems, such as RAIDs, provide increased I/O bandwidth and data protection through redundancy [3]. However, RAIDs still rely on a single point of access to the I/O system. Parallel file systems remove this bottleneck by splitting I/O requests between multiple nodes which handle I/O. These nodes, known as *I/O nodes*, access their disks in parallel and take advantage of the network bandwidth, providing parallel points of access to the I/O system. Parallel file systems thus provide a necessary capability for many application domains, including out of core methods.

Many parallel algorithms rely on the data set fitting in the available memory on the parallel machine. As data sets grow beyond memory capabilities, algorithms must be utilized that work on data sets beyond the core memory size. While virtual memory systems allow for memory to spill over onto disk space, VM often imposes a high performance penalty. As an alternative, explicit out of core algorithms recognize that parallel processes rarely require all of their data in memory at one time and can read sections of such data from disk at the necessary time. Traditional sequential OOC algorithms do not port well to parallel machines because many commercial parallel machines have poor I/O characteristics, which have adverse effects on OOC applications. In contrast to these machines, Beowulf workstations have better relative I/O characteristics because each node contains a disk. Thus OOC algorithms might map more effectively to this type of system.

One application that can take advantage of parallel I/O systems and out of core algorithms is sorting. Sorting requires large amounts of I/O and has proven well-suited to networks of workstations [4], which exhibit many of the characteristics of Piles-of-PCs. This paper presents two out of core sorting algorithms and their performance on a Beowulf machine running a parallel file system. The focus of this study will be on the behavior of these algorithms with problem sizes that approach and exceed the core memory size. The next section will delve into the work already performed in this area and how it relates to the work presented herein. Section 3 will describe the algorithms tested and the experimental methods, and Section 4 will present the results.

2 Background

In order to efficiently perform sorting operations on Beowulf workstations, it is important to match the algorithm to the system software and architecture characteristics. Here we discuss the particulars of the Beowulf workstation, the parallel file system, parallel sorting, and the role that OOC computation will play.

2.1 Beowulf Machines

The Beowulf workstation is a fairly new concept in the realm of parallel computing [5]. A Beowulf workstation is a dedicated set of PCs built from commodity parts connected by an inexpensive dedicated system area network, combined with a set of freely available software to provide an operating system, compilers, and message passing system. The ideal software for this type of distributed machine would allow the user to view the system as a single machine by coordinating processes among the nodes. This set of software is continuously enhanced by the growing community of Beowulf users, who generally make their additions freely available. The use of commodity off-the-shelf parts allows the most recent technology to be included in a machine being built. Massively parallel machine development has been hindered in the past by the technology curve in that by the time the machine is built, some of the hardware, especially the processors, is obsolete. Beowulf nodes can be assembled and upgraded like workstation PCs, and no custom hardware is needed to assemble a system.

The goal of a programmer on a Beowulf workstation is to develop algorithms that take advantage of a

Beowulf's strengths, such as fast processors and a distributed I/O system, while compensating for its weaknesses, namely commodity networking hardware not designed for parallel computing. Presently, Beowulf workstations have been used to process N-body algorithms [6], electromagnetic codes, and systems of equations with Gauss-Seidel methods [7].

2.2 Parallel Virtual File System

In parallel applications, I/O generally occurs at three points: initially reading the data set, writing out the solution data, and, in the case of out of core applications, reading and writing intermediate data. The limiting factor of a parallel disk-to-disk sorting application is often the I/O system, especially with out of core sorts.

Certain basic principles of parallel I/O persist through most attempts to create a useful parallel file system. *Declustering* involves spreading a file across a set of disks, in order to increase the total bandwidth when accessing a chunk of the file. *Striping* is a declustering scheme where file clusters are interleaved round-robin across a set of disks [8]. A large access to a striped file may cause several disks to respond, thereby taking advantage of the greater network and disk bandwidth. Also, if placed correctly, a striped file should improve data locality, as each node could have part of its data set on a local disk.

The Parallel Virtual File System (PVFS), developed at Clemson University, takes a streams-based approach to parallel I/O. It is one of the few parallel file systems designed specifically for a cluster of workstations environment. The system is user-level and consists of a manager daemon process, which runs on any single node, and a set I/O daemons (IOD), which run on any node used for I/O. The set of I/O nodes can overlap the set of compute nodes. TCP is used to communicate with compute processes via a set of library calls. The manager daemon coordinates file opens and closes, checks permissions and performs most of the operations not requiring a read or write.

The IODs communicate directly with compute processes when performing a read or write. PVFS uses UNIX socket commands for communication and, therefore, is portable to most UNIX systems. It has been tested on a number of Linux systems and a DEC Alpha cluster. The IOD takes a set of request parameters and performs the set of sequential disk accesses, coordinating the transfer from disk to network. To improve network performance, data is packed into large packets before being sent over the network. The request parameters for PVFS are not dependent on the disk distribution, so multi-strided requests can be filled without data sieving. PVFS is an effective parallel file system providing consistency and speed for parallel applications with large amounts of I/O.

One of the open issues in parallel I/O is the proper allocation of nodes as I/O nodes and compute nodes. Most parallel machines partition their nodes into compute (or work) nodes and auxiliary nodes. With dedicated clusters, it may be more efficient to use nodes for both compute and I/O work, because each node has a local disk. Kotz has examined using compute nodes to do I/O work on a massively parallel machine using his disk-directed I/O paradigm [9]. Those results show that the processors can continue to run between 50% and 85% efficiency while servicing I/O requests, depending on the types of requests. These tests were run on a parallel machine simulation with a set of I/O access traces. The use of nodes for both computation and I/O was also explored by Cettei et al. [7], where it was found that for an OOC Gauss-Seidel iterative solver performance was highest when I/O and compute nodes were overlapped. This issue will be examined in this paper as well in the context of sorting applications.

2.3 Parallel Sorting

Research related to parallel sorting is widespread but mostly relates to the traditional fast network parallel machines, while very little work has focused on a clustered computing environment. Still, many of the algorithm studies can transfer loosely to a cluster of workstations, although efficient network use is more important on a Pile-of-PCs architecture than on a massively parallel machine. Most of the sequential sorting algorithm complexities were first reported by

Knuth[10], and most work since has focused on performing sorts on various architectures. The hypercube algorithm put forth by Abali et al. [11] performs a quicksort on each node, then performs a Fast Partition algorithm to balance the load on each processor before passing data to other processors. This algorithm is similar to the bucket sort on a cluster of machines examined in this paper. Wen has shown an efficient parallel algorithm for merging multiple lists on a concurrent-read exclusive-write parallel random access machine (CREW PRAM) [12], which has similarities with the mergesort presented here.

The Network of Workstations (NOW) project at UC-Berkeley has provided the best non-commercial disk-to-disk sorting performance to date [4]. Using a network of 95 Sun workstations and Myrinet network, the NOW Sort group won the Indy MinuteSort award for largest sort in one minute. The NOW Sort uses a simple bucket sort algorithm and assumes a uniform distribution. Given P workstations, the problem is partitioned into P buckets, based on the distribution. Each workstation sorts its initial partition and distributes the resulting buckets to the other processors. The initial sort used is a bucket/partial radix sort, which was found to be superior to a quicksort and a quicksort over buckets for their implementation. To perform read accesses, the NOW Sort uses the mmap() command with madvise() in order to maximize performance. The key values are split into buckets and sent to other nodes, where the local data is sorted and written to disk. In terms of overlapping communication with I/O, the best results were found with a multi-threaded version using a reader thread and a send thread.

The one-pass NOW Sort was found to be nearly perfectly scalable up to 32 processors on their hardware. The NOW Sort group developed a two-pass sort in order to operate on a data set that was OOC. The two-pass consists of several bucket sort runs followed by a mergesort. This sort scales fairly well, although the parallel version performs well below their one-pass sort. The work by the NOW Sort group most closely matches the work presented here.

2.4 Out of Core Computation

Out of core algorithms are defined as algorithms which can run on data sets larger than the main memory size. The Beowulf architecture is particularly well-suited for OOC programs because often this shifting of data to and from disk can happen locally. There are two approaches to dealing with OOC problems: the use of virtual memory and explicit out of core solutions.

Virtual memory is an operating system feature that allows a larger "virtual" memory space than the size of physical memory. This is accomplished by the use of local disk as a buffer area for regions of memory not currently in use. Nothing need be done by the user to enable virtual memory; the kernel simply moves sections of memory onto disk when it needs more space in physical memory. This makes the coding of OOC problems trivial; however, often performance suffers the moment the swapping of memory onto disk begins.

The primary characteristic of explicit out of core programming is that the user manages memory use. To this end, the programmer partitions the problem into sections that can fit into memory and handles shifting sections in and out of memory during the computation. This can be troublesome to code if library support is not available, but performance is often superior to virtual memory solutions.

This paper builds on previous work studying the performance of OOC implementations by focusing on OOC sorting. Previous work by the Parallel Architecture Research Lab (PARL) group at Clemson compared an explicit OOC implementation of a Gauss-Seidel Iterative Solver with another version that was not designed to run OOC. The non-OOC solver ran well for small matrix sizes that fit in memory, but when virtual memory was needed, the explicit OOC algorithm executed much more quickly [7]. Kotz studied the use of disk-directed I/O with an OOC LU decomposition problem, finding that DDIO did improve performance of the application over using traditional caching [13]. Salmon and Warren studied parallel OOC methods for N-body simulation using tree codes, and found the OOC tree codes essentially reduced to OOC sorting [6].

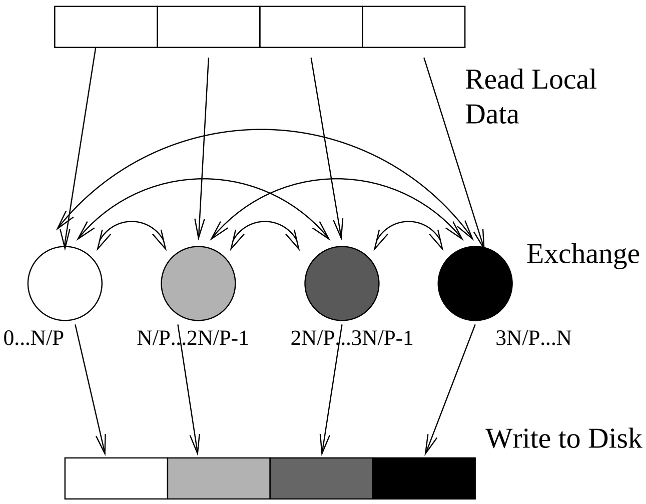

Figure 1: Diagram of data flow for parallel bucket sort.

3 Out of Core Sorting

Previous sorting work has shown that near-optimal algorithms can be developed for parallel sorting. As sort data size requirements grow, methods of performing sorts out of core become necessary. The focus of this paper is the dedicated cluster computer, specifically the Beowulf class computer. In this section the complexity of OOC sorting on dedicated clusters will be discussed, as well as the experimental setup.

3.1 Bucket Sort

The bucket sort tested here was deliberately coded to be nearly identical to the NOW Sort discussed in the previous chapter. The steps of the bucket sort are:

- Read $\frac{N}{P}$ records from disk with PVFS
- Partition data into P buckets and exchange with other processors
- Quicksort bucket and write back to disk

A diagram of the data flow in this algorithm is given in Figure 1. The obvious advantage is that there is only one read step and one write step, so the I/O is minimized, but the memory requirements are tied to the problem size. Each compute node must send a one-to-one message to each processor, at an average of $\frac{N*K}{P^2}$ bytes per message, where K is the record size in bytes. Each processor sends *P-1* messages, so the total network traffic is approximately $N * K$ bytes.

The bucket sort utilizes two buffers on each node to hold data before and after sorting. It has been shown by Nyberg et al. [14] that sorting via pointers is considerably faster than sorting large records; therefore, our implementation uses an additional buffer of pointers on which the sort is performed. With a uniform distribution each processor's bucket is assured to be $\frac{N*K}{P}$ size. This leads to a total memory requirement of:

$$\frac{N}{P} * (2 * K + sizeof(ptr))$$

For a non-uniform distribution the memory requirements would vary depending on how much the data is skewed, as some buckets would be much larger than others. The NOW group has implemented a random sampling into their bucket sort to approximate the data range and minimize the bucket size variance; this was not included in our implementation, and non-uniform distributions are not covered in this work.

3.2 Mergesort Algorithms

The second algorithm tested is based on a mergesort scheme. It has no reliance on data distribution and can also operate on any sized data set. The sort data begins and ends distributed via PVFS striped files. This is not an optimal algorithm by any means, but it has the interesting characteristic that it relies on a buffer whose size is independent of problem size. Given P processors, N records of size K, and a buffer of B bytes, the steps of this mergesort are:

- Quicksort $\frac{N}{P}$ records in B-sized sections and write back to disk
- Mergesort the $\frac{N*K}{P*B}$ B-sized sections and write back to disk. Each processor should have $\frac{N}{P}$ sorted records on disk.
- Arrange processors into a tree structure and perform a mergesort. Each processor merges two streams and sends them to the next tree level. See Figure 2.
- Write back to disk at the final two processors.

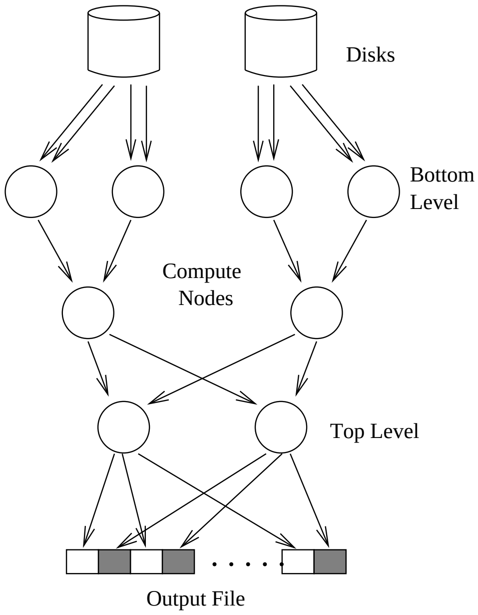

Figure 2: Diagram of data flow for mergesort algorithm

The mergesort requires roughly $3 * B$ bytes of memory on each machine in the cluster, and empirical testing found that a buffer size of 1MB was adequate. Assuming that the number of processors is a power of two, the tree structure above leaves two processors at the top level. Because the transfer to disk is the bottleneck in the process, the final processor merge is duplicated, and each processor writes half of the final data (in interleaved accesses) back to the disk. The buffer value is used to determine how much data is merged at one time. Each of the first-level processors read two B-sized sections from disk, mergesorts the sets, and sends them onto the next level. The first-level processors will therefore mergesort a total of $\frac{2N}{P}$ records. The final merge is the limiting factor of the algorithm, as it always involves just two processors, and thus limits the scalability of the algorithm. However, it serves as a reasonable algorithm for our purposes.

The mergesort algorithm presented above operates out of core, so it requires more I/O than an in-core algorithm. During the first phase, when each node sorts its own partition, the partition must be read and written completely $\log\lceil\frac{N*K}{B*P}\rceil$ times. The second phase, where the merge tree is built, requires a read of $N * K$ bytes by $\frac{P}{2}$ processors and a write of the same amount by 2 processors. The entire sort data set must be sent in a node-to-node communication $\log P$ times, with message sizes of B bytes. Much of this communication will be overlapped with other communications and the disk reads, so that while the first row of processors continues to merge B-sized buffers from disk, it will be piping $2B$-sized buffers to the next level in the tree. For sizable data sets, all processors in the tree will be busy for some period of time.

During our tests we also use this algorithm with buffer sizes set to hold the entire data set in core. This allows us to explore the behavior of the algorithm when using VM by increasing the problem size.

3.3 The Grendel Machine

The Beowulf machine on which this work was performed is a 17-node cluster connected by a Bay Networks fast ethernet switch. Each node has the following specifications:

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

One node runs the PVFS manager daemon and handles interactive connections while the other nodes can be used as compute nodes, I/O nodes, or both. Each node runs Linux v2.0.27 with a Tulip driver by Donald Becker. The PVFS file system was used to make all I/O accesses, and the Parallel Virtual Machine (PVM) [15] provided message-passing between compute processes. The IDE disks provide approximately 4.5MBps with sustained writes and 4.2MBps with sustained reads, as reported by Bonnie, a popular UNIX file system performance benchmark. When idle, approximately 6MB of memory are used on each node by the kernel and various system processes, including PVFS and the PVM daemon.

4 Results

This section will discuss the experimental results of tests performed on the two algorithms presented in the previous chapter. Each individual test was run five times, with an average of these results presented here. Unless otherwise noted, the sort keys were four bytes in each test, record sizes were kept at 128 bytes, all I/O was performed on the nodes used for computation, and buffer sizes of 1MB were used in the explicit OOC algorithm. The results fit in the following major categories:

- explicit OOC and virtual memory performance for the mergesort
- performance of algorithms with a uniform key distribution
- effects of compute node and I/O node overlap

The standard performance metric is the total execution time for the disk-to-disk sort and the comparative speedup, where applicable.

4.1 Explicit OOC and Virtual Memory Mergesort

The purpose of these experiments was to determine how well the explicit OOC mergesort scaled (for the available number of nodes) and how it compared to the same algorithm using an in-core data set (and virtual memory when necessary). The virtual memory version, because of limits on swap space, would only run up to a problem size of 256K records of 128 bytes on four nodes. Figure 3 demonstrates the speedup of the OOC parallel mergesort over the virtual memory parallel mergesort on the same number of nodes as the data set is made larger. The performance is nearly equivalent up to 128K elements, after which the VM version is much slower.

The OOC mergesort competed well with the virtual memory version on smaller data sets, and far outperformed it on larger sets. The OOC mergesort performed well on data sets sizes up to 1GB on 16 nodes, while the virtual memory version took hours or days for tests requiring more than 64MB of memory per node.

Figure 4 shows the speedup of the OOC parallel mergesort over the OOC sequential version of the sort for various array sizes. The mergesort does not provide linear speedup because the final writing, performed by two processors in this configuration, is a bottleneck and because for a fraction of the time spent in the tree phase not all processors are busy. However, as the data set size is increased, the overall efficiency improves. This is due to two factors: there is more overlap of communication and computation in the tree phase, and data transfer in this environment is more efficient with larger blocks.

4.2 OOC Mergesort vs. Bucket Sort

This section compares the OOC mergesort with the one-pass bucket sort presented by Arpaci-Dusseau et

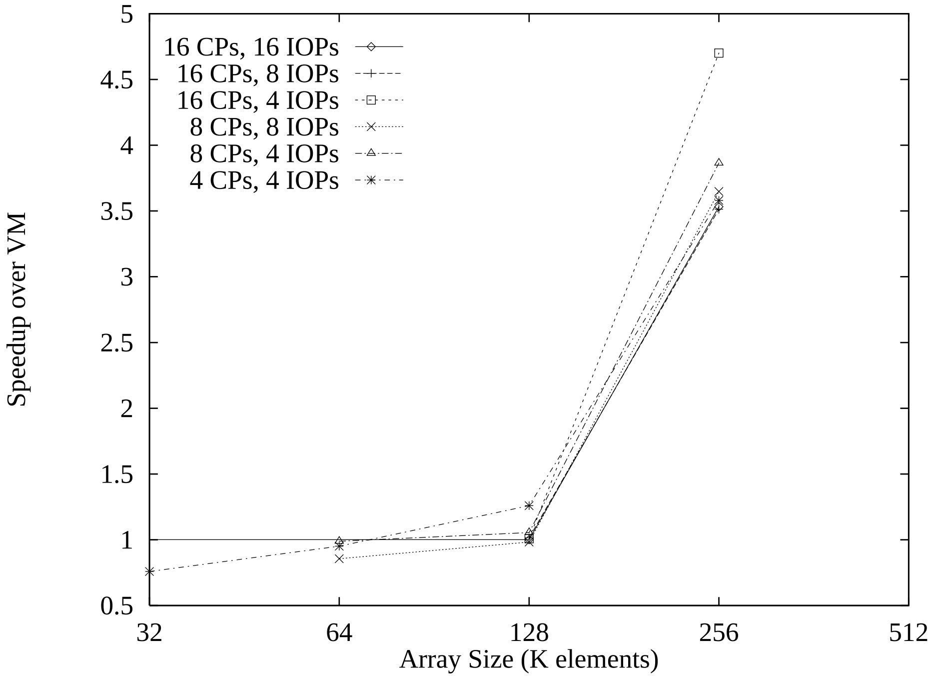

Figure 3: Speedup of the OOC mergesort over the virtual memory version.

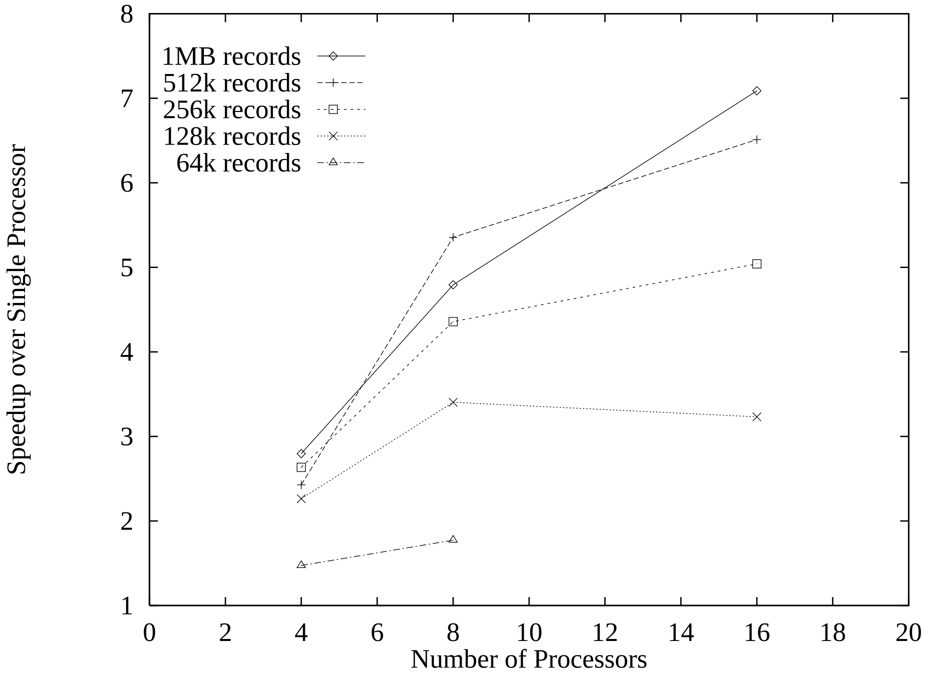

Figure 4: Speedup of the OOC mergesort over the sequential single processor.

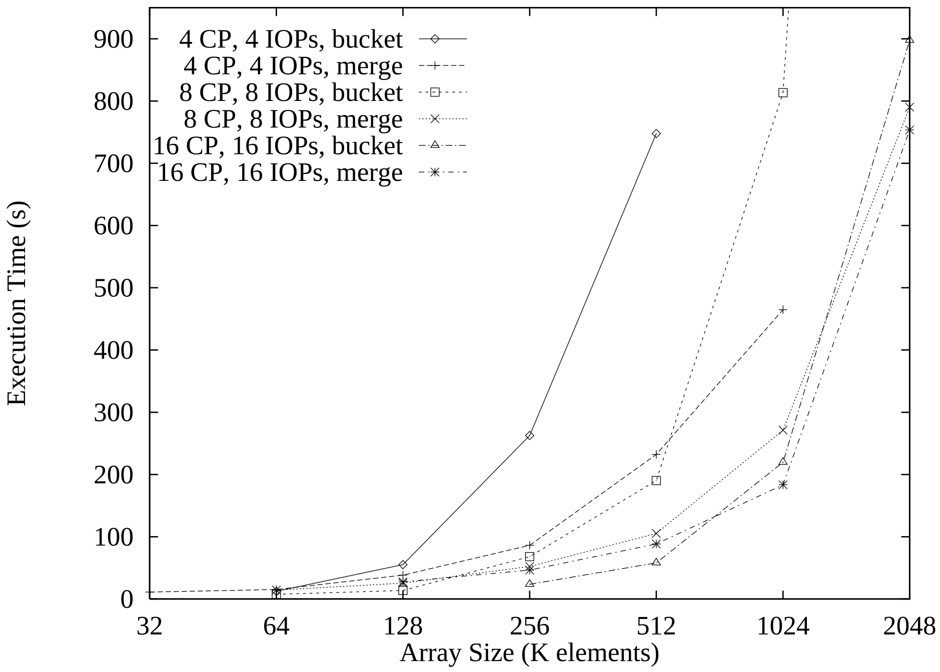

Figure 5: Merge and bucket sort performance on a uniform distribution.

al. [4] on both a uniform distribution and a skewed distribution. Figure 5 shows the comparative performance with a uniform distribution. The bucket sort does perform better on the smaller data sets, but the merge performs better as the data set size increases. In particular, the performance of the bucket sort begins to drop when each node is responsible for 128K of 128 byte records. This corresponds to a memory usage of 36MB by the bucket sort for holding local records. With I/O being performed on these same nodes, contention begins to occur for pages of memory as dirty buffers accumulate.

4.3 Sharing Compute and I/O Nodes

These experiments were designed to examine the use of overlapping I/O and compute nodes. Figure 6 summarizes these results. In general, adding additional nodes to serve as separate I/O nodes did improve the performance of the algorithm. However, in each case, for a fixed number of resources the algorithm performed significantly better when nodes were used for both I/O and computation. When we consider the amount of explicit disk I/O occurring in these sorts, this seems obvious. For the mergesort operating on 256K of 128 byte records, the data set size is 32MB. This corresponds to 96MB of writes and 96MB of reads throughout the execution time of the application. For 16 nodes, this is only 12MB of I/O per node, which we would expect to take only 3 seconds of the approximately 50 second execution time based on the characteristics of our disks. Thus the nodes are free to compute roughly 94% of the time.

5 Conclusions

This paper has shown that an out of core sorting algorithm is superior to approaches that rely on virtual memory for problem sizes that approach or exceed core memory size and has comparable performance for smaller sizes. The most interesting point here is that even a small amount of necessary I/O, such as the

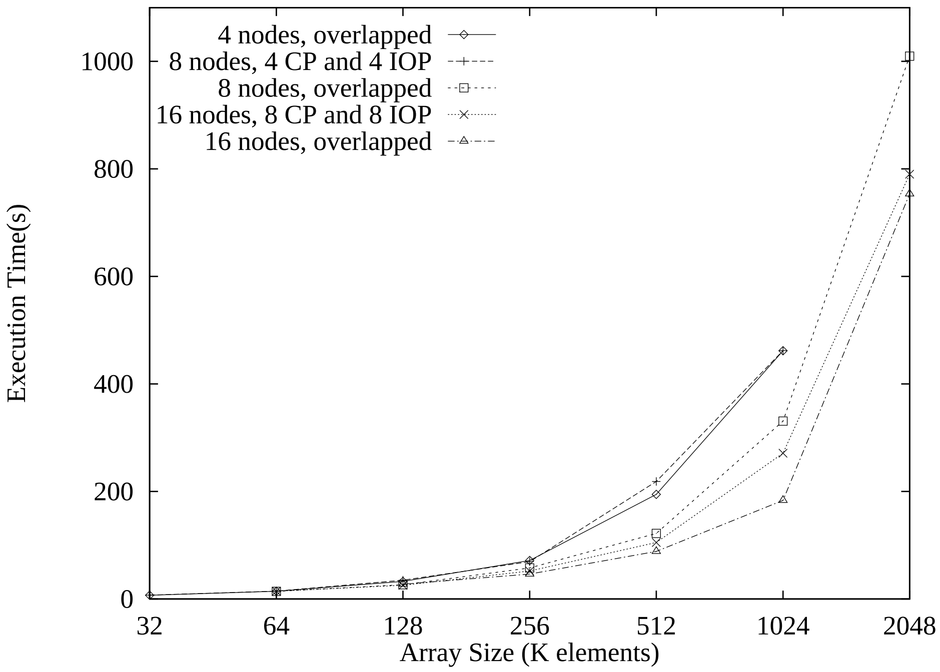

Figure 6: Using I/O nodes as compute nodes.

192MB of I/O over 50 seconds in the 256K record case, can result in a tremendous performance hit when instead of explicitly performing the I/O, the VM system is left to its own devices. It is also interesting to note that our OOC mergesort algorithm was operating with 3MB of memory for record storage during most of the tests and was able to sort data sets of 1GB, indicating that it is not necessary to hold large portions of the data set in core. File system size constraints limited our ability to test beyond this size.

However, the execution of parallel sorting algorithms out of core still presents several open issues. First, the scalability of this mergesort algorithm is questionable, and new approaches to sorting that retain an independent buffer size should be studied in this environment. Second, further testing should be performed to push the limits of the I/O subsystems to better determine when it is appropriate to overlap I/O and compute nodes and when it is not. Finally, it is still much more convenient to use VM rather than perform explicit I/O; more progress needs to be made in order to simplify the process of writing explicit OOC applications so that the performance benefits are more easily obtained.

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] D. Patterson, G. Gibson, and R. Katz, "A case for redundant arrays of inexpensive disks (RAID)," in

Proceedings of the ACM SIGMOD International Conference on Management of Data, (Chicago, IL), pp. 109–116, ACM Press, June 1988.

[4] A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, D. E. Culler, J. M. Hellerstein, and D. P. Patterson, "High-Performance Sorting on Networks of Workstations," in *Proceedings of the 1997 ACM SIGMOD Conference*, pp. 243–254, 1997.

[5] T. Sterling, D. J. Becker, D. Savarese, J. E. Dorband, U. A. Ranawake, and C. V. Packer, "Beowulf: A parallel workstation for scientific computation," in *Proceedings of the 1995 International Conference on Parallel Processing*, 1995.

[6] 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.

[7] M. Cettei, W. B. L. III, and R. Ross, "Support for parallel out of core applications on beowulf workstations," in *Proceedings of the 1998 IEEE Aerospace Conference*, 1998.

[8] P. Dibble, M. Scott, and C. Ellis, "Bridge: A high-performance file system for parallel processors," in *Proceedings of the Eighth International Conference on Distributed Computer Systems*, pp. 154–161, June 1988.

[9] D. Kotz and T. Cai, "Exploring the use of I/O nodes for computation in a MIMD multiprocessor," in *Proceedings of the IPPS '95 Workshop on Input/Output in Parallel and Distributed Systems*, pp. 78–89, April 1995.

[10] D. E. Knuth, *The Art of Computer Programming: Sorting and Searching (Volume 3)*. Addison-Wesley, 1973.

[11] B. Abali, F. Ozguner, and A. Bataineh, "Balanced parallel sort on hypercube multiprocessors," *IEEE Transactions on Parallel and Distributed Systems*, vol. 4, pp. 572–581, May 1993.

[12] Z. Wen, "Multiway merging in parallel," *IEEE Transactions on Parallel and Distributed Systems*, vol. 7, pp. 11–17, January 1996.

[13] D. Kotz, "Disk-directed I/O for an out-of-core computation," in *Proceedings of the Fourth IEEE International Symposium on High Performance Distributed Computing*, pp. 159–166, August 1995.

[14] C. Nyberg, T. Barclay, Z. Cvetanovic, J. Gray, and D. Lomet, "AlphaSort: A RISC Machine Sort," in *Proceedings of 1994 ACM SIGMOD Conference*, May 1994.

[15] V. Sunderam, "Pvm: A framework for parallel distributed computing," *Concurrecy: Practice and Experience*, pp. 315–339, December 1990.