M. Nikseresht, D. Hutchinson and A. Maheshwari, "Experiments With A Parallel External Memory System", 14th Annual IEEE International Conference on High Performance Computing (HiPC), Goa, India, December 2007.

Experiments with a Parallel External Memory System*

Mohammad R. Nikseresht[1], David A. Hutchinson[2], and Anil Maheshwari[1]

[1] School of Computer Science, Carleton University
[2] Dept. of Systems and Computer Engineering, Carleton University

Abstract. The theory of bulk-synchronous parallel computing has produced a large number of attractive algorithms, which are provably optimal in some sense, but typically require that the aggregate random access memory (RAM) of the processors be sufficient to hold the entire data set of the parallel problem instance. In this work we investigate the performance of parallel algorithms for extremely large problem instances relative to the available RAM. We describe a system, Parallel External Memory System (PEMS), which allows existing parallel programs designed for a large number of processors without disks to be adapted easily to smaller, realistic numbers of processors, each with its own disk system. Our experiments with PEMS show that this approach is practical and promising and the run times scale predictable with the number of processors and with the problem size.

1 Introduction

In this work we investigate the performance of parallel algorithms for extremely large problem instances relative to the available Random Access Memory (RAM). Using theoretical results of [1, 2], we transform parallel algorithms designed for a large number of processors without disks to smaller, realistic numbers of processors, each with its own disk system.

External Memory (EM) Algorithms: These algorithms are designed so that their run times scale predictably even as the size of their data increases far beyond the size of internal RAM. This huge data size requires that such algorithms optimize the transfer of data between RAM and some sort of secondary memory devices, typically moveable-head disk drives. The time to access an element of data at an arbitrary position on a moveable-head disk is several orders of magnitude greater than for RAM access. In addition, the time to set up the data transfer (disk head movement, rotational delay) is much larger than the time to actually transfer the data. Data items are grouped into *blocks* and accessed block-wise by efficient external memory algorithms in order to amortize the setup time over a large number of data items.

* This work was partially supported by the National Sciences and Engineering Research Council of Canada (NSERC) and by the High Performance Computing Virtual Laboratory (HPCVL).

A critical quality in a good EM algorithm is its locality of reference to items in a disk block (or collection of disk blocks). EM algorithms are designed to use all, or a significant fraction, of the elements in a block while the block is in memory, avoiding the cases where the same block must be brought into memory many times. The operating system's data cache may interfere with the operation of an EM algorithm, as it may make useless copies of data blocks, and occupy RAM that could be used more effectively for other purposes. EM algorithms manipulate huge data volumes relative to their RAM size, and so the time to move data between RAM and disk dominates the running time in most cases. For this reason, the Parallel Disk Model (PDM) [3] uses the number of distinct disk block input or output (I/O) operations as a measure of the goodness of an EM algorithm. If D disks are present and the block size is B items, a single I/O operation can transfer BD items in parallel in this model. An optimal EM algorithm is one that achieves the minimum number of I/O operations possible.

Coarse Grained Parallel Algorithms: These algorithms have a number of interesting and useful properties for our purposes: (a) the processors perform multiple rounds of computation separated by communication of interim results, (b) during each computation round, the processors perform a chunk of computational work. For this period of time the processors operate completely independently, and are restricted to accessing the data in their own local memories. and (c) between computation rounds the processors synchronize and exchange information via a communication network.

The simulation techniques described in [1] permit coarse grained parallel algorithms to be executed efficiently on machines with p *real* processors rather than the number of processors v required by the original algorithm, where $1 \leq p \leq v$. We refer to the original v processors as *virtual* processors. The tradeoff is that each of the real processors must have enough disk space to store the internal memory for v/p of the virtual processors plus the messages that would be sent between the virtual processors in each communication round. The real processors must have enough RAM to represent at least one virtual processor at a time, and that should be at least DB items in size so swapping between virtual processors is I/O efficient. Finally, the communication should be I/O efficient, meaning that at least DB blocks should be sent and received by each virtual processor in each communication round. Intuitively, the simulation technique works because the coarse grained property of the original parallel algorithm ensures that the requisite locality of reference is present in the resulting EM algorithm. We can "trade-off" some parallelism for blocked I/O but retain some parallelism to take advantage of multiple real machines and the scalability of multiple disk systems. The theoretical guarantees of asymptotically optimal parallelism and I/O in the resulting algorithms makes them interesting for practical use, since their parameters (v, p, B, D) can be scaled to fit the parallel hardware at hand.

Previous Work: The work of this paper extends the results of previous implementation work reported in [2] from a single processor to multiple real processors.

LEDA-SM [4], TPIE [5] and STXXL [6] are I/O workbenches developed to explore external memory algorithm implementations for sequential machines. TPIE provides prototype implementations of many contemporary EM algorithms but currently it does not offer algorithms for multiple processors or multiple disks. STXXL provides a mechanism for asynchronous I/O and therefore allows overlapping of I/O and computation. It handles multiple disks but it does not itself support multiple processor algorithms. SSCRAP is a framework for implementing parallel coarse grained algorithms and has been extended [7] to support the simulation of certain parallel algorithms in external memory with reference to the theoretical framework of [1]. However, it is not clear whether SSCRAP handles communication traffic between virtual processors in an I/O-optimal way, or whether very large problem instances relative to RAM size have been tested.

Organization and Contributions of this Paper: This paper takes a step forward in determining the practicality of the simulation approach. In Sect. 3, we describe a framework that allows existing MPI-based implementations of CGM algorithms (or BSP algorithms with appropriate parameters) to be executed efficiently on a machine with fewer real processors but with disk storage. Such MPI-based programs are modified in the following ways: (a) calls to the MPI library are replaced by calls to corresponding procedures in the PEMS library and (b) calls to the C dynamic memory management routines are replaced by calls to corresponding PEMS routines. In Sect. 4, we report preliminary timing results for sorting that are comparable with TPIE and STXXL, two contemporary workbenches for high performance I/O experiments. We compare single processor instances of sorting on these workbenches with sorting using various numbers of real processors running a CGM sample sort implementation. While our timing results lagged those of TPIE and STXXL for a single processor, we are able to surpass both by adding more processors. Our experiments show that this approach is practical and promising and the run times scale predictable with the number of processors and with the problem size.

2 Preliminaries

In this section we present the main ideas behind the simulation technique proposed in [1]. It optimizes blockwise data access and disk I/O and at the same time utilizes multiple processors connected via a communication network or shared memory. The Bulk-Synchronous Parallel (BSP) model [8] consists of v processor/memory components, a *router* that delivers messages in a point to point fashion, and a facility to synchronize all processors. Each processor has a unique label in the range $0, 1, \ldots, v-1$. Computation proceeds in a succession of *supersteps* separated by synchronizations, usually divided into *communication* and *computation supersteps*. In computation supersteps processors perform local computations on data that is available locally at the beginning of the superstep and issue send operations. Between computation supersteps, a communication superstep is performed, where each processor exchanges data with its peers, via

the router. This is done through an *h-relation*, where $O(h)$ data are sent and received by every processor in a superstep. In addition to the parameters v and h, BSP uses two additional parameters. The parameter g is the time required to send a single word of data between two processors, where time is measured in number of CPU operations, and the parameter L is the minimum setup time or latency of a superstep, measured in CPU operations.

The technique in [1] simulates a v (virtual) processor BSP algorithm $\mathcal{A}'$, executing a problem of size N, which communicates via h-relations of size $h = O(\frac{N}{v})$ with λ supersteps/rounds, local memory size μ, computation time $\beta + \lambda L$, communication time $g\alpha + \lambda L$ as a p (real) processor EM-BSP algorithm $\mathcal{A}$ with $\frac{v}{p}\lambda$ rounds, computation time $\frac{v}{p}(\beta + O(\lambda\mu)) + \frac{v}{p}\lambda L$, communication time $\frac{v}{p}g\alpha + \frac{v}{p}\lambda L$, and I/O time $\frac{v}{p}G \cdot O(\lambda\frac{\mu}{B}) + \frac{v}{p}\lambda L$ for $M = \Theta(\mu)$, $N = \Omega(vB)$, $B = O(\frac{N}{v^2})$, $p < v$. The parameter G is the ratio between the local computational capacity and the local I/O capacity and the parameter B is the disk block size.

Next we sketch the main steps of the simulation. We distribute v virtual processors evenly on p real processors. Each real processor i, $0 \leq i \leq p$ executes the following steps (for a single processor simulation, we set $p = 1$, $i = 0$ and omit Step 5):

For j = 0 to $\frac{v}{p} - 1$ do in parallel on each real processor i

1. Read the context of virtual processor $i\frac{v}{p} + j$ from the local disk.
2. Read any messages addressed to virtual processor $i\frac{v}{p} + j$ from the local disk.
3. Simulate the computation superstep of virtual processor $i\frac{v}{p} + j$, collecting all generated messages in the local internal memory.
4. Send all generated messages to the required (real) destination processors.
5. Receive all messages addressed to real processor i on behalf of virtual processors $i\frac{v}{p}$ to $(i+1)\frac{v}{p} - 1$ in local internal memory and write them to the local disk.
6. Write the changed context for virtual processor $i\frac{v}{p} + j$ back to the local disk.

---

3 Software Design

In this section we explain our software design and show how different components work and interact together in the Parallel External Memory System (PEMS). In PEMS a virtual processor is represented by a user space thread. The input to our system is an existing MPI program implementing a coarse grained BSP algorithm. In such a program MPI is responsible for interprocessor communications. Most of the MPI functions are for sending and receiving messages. In PEMS, each MPI call is replaced by call to a corresponding PEMS service. These PEMS calls may in turn incorporate MPI calls. The communication between virtual processors is managed using the disk, memory buffers and communication network. This is the main challenge in the design and implementation of PEMS.

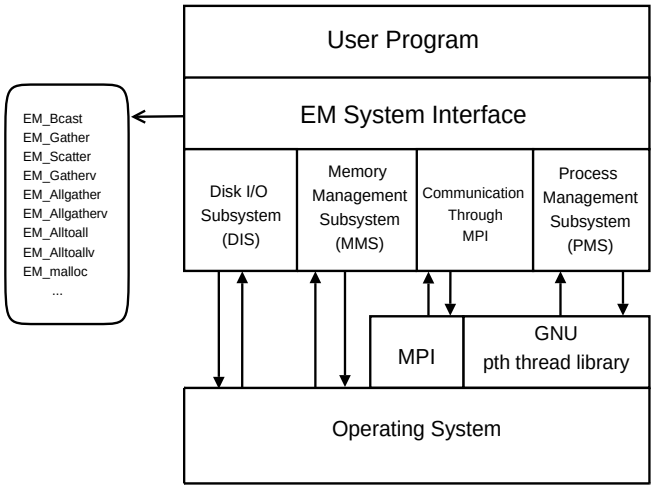

Fig. 1. PEM System Software Layers

To date, our focus has been on confirming the high level behavior of the simulation approach, that is to confirm that the behavior of PEMS scales predictably as problem sizes and the number of real processors vary. This involves both running time and disk space usage behaviors. We have not been overly concerned about optimization in this initial version of PEMS.

Figure 1 shows the layered software design of PEMS. The user program is shown as the top layer in the diagram. The EM System Interface (Layer 2) provides the external memory communication primitives. All MPI and memory allocation calls in the original user program are replaced by a call to a similar PEMS function in this layer. Layer 3 has four major components. The Disk I/O Subsystem (DIS) is responsible for reading and writing data blocks to disk efficiently. The Memory Management Subsystem (MMS) is responsible for allocating memory for virtual processor data and for swapping this data when required. The Process Management Subsystem (PMS) is responsible for creating, scheduling and synchronizing virtual processors. The Open MPI library [9] is used for communication between real processors and also for starting the software on multiple machines. In subsequent subsections we discuss layers 2 and 3 in more detail.

Virtual Processor Simulation and Process Management Subsystem (PMS): The GNU pth thread library [10] is used for representing virtual processors as threads. Each virtual processor is a user level thread. Each thread runs a copy of the modified MPI-based user program. Modifications include replacement of MPI and memory allocation calls with the corresponding PEMS calls. The PMS consists of a set of functions to initialize, start, synchronize, manage, and schedule threads. These functions are implemented using the GNU pth thread library which is responsible for the creation and cooperative schedul-

ing of threads. At the end of every computation superstep each virtual processor calls _EM_yield(). This first invokes the services of MMS. The data for the current thread must be swapped out and the data for the new thread must be swapped in. Then the services of the thread library are invoked to perform a context switch and transfer control to the new thread. Coordination of different processes on different machines is achieved using the open MPI library [9]. Each virtual processor has a global identification number which we will call its PEMS rank, as well as a local thread number within its user space process, and an MPI rank that identifies which MPI process contains it. The PEMS rank of a virtual processor is computed based on the MPI rank of the process and its local thread number. When virtual processors are communicating, their PEMS ranks are used to distinguish them from each other. It is possible to change the total number of virtual processors by changing the number of virtual processors in each MPI process or by changing the number of MPI processes which are running on different machines.

Memory Management Subsystem (MMS): As PEMS runs, context switching between threads occurs whenever a virtual processor reaches the end of a computation superstep. The simulation technique resizes each virtual processor (thread) to use the available physical memory. To accommodate many such threads it would normally be necessary for the operating system to allocate memory from its swap space or virtual memory. Each time a thread starts execution, the operating system would swap in the needed pages of data from the disk swap area. This is done based on a page fault mechanism which may not be efficient for our purposes. Even if the operating system was able to perform swapping efficiently, the total amount of memory needed for all virtual processors could be well beyond the virtual address space of the combined operating system and the hardware. This is especially true for 32-bit machines. Therefore, we need complete control over the PEMS physical memory and all of the disk activities related to the execution of the PEMS program.

The Memory Management Subsystem determines how much physical memory is available at the beginning of the program execution (initialization phase). Half of this memory is used as a buffer for communication purposes and for assembling and disassembling messages. The other half is reserved for use by virtual processors; they can allocate memory from this area by calling EM_malloc().

Disk I/O Subsystem (DIS): This layer allows the higher layers of the system to be independent of specific operating system I/O calls. This makes the system more flexible and portable, and potentially allows PEMS to interface to third party I/O packages. The DIS currently contains implementations of direct I/O based on the native Linux direct I/O which is supported in kernel versions 2.6 and above, asynchronous I/O and synchronous I/O. Currently, PEMS uses direct synchronous I/O for swapping and buffered synchronous I/O for messaging.

EM System Interface and Messaging: This layer is the most important layer in our implementation. While the contexts (local data) of each virtual processor can be swapped between RAM and disk using efficient streaming I/O, this is not genereally true for delivering messages between virtual processors in the simulation. All of the PEMS functions have the same number and type of parameters as their MPI counterparts. This makes it easier to transform an MPI program into a PEMS program. In fact we believe that there are simple automated ways of doing this. We have also implemented EM data types corresponding to the MPI data types (e.g. MPI_INT, MPI_CHAR). As mentioned earlier, communication is done through the disk, memory buffer, network, or a combination of them depending on the communication function. If two communicating virtual processors are on the same machine then they communicate through the disk or memory buffer, depending on the communication function invoked. If they are on two different machines they communicate through the network as well. PEMS splits the available physical memory into two partitions, each equal to the data memory size of a virtual processor. One partition is used by the virtual processors and the other partition is used by PEMS as a large buffer. Any message which fits into this buffer is kept there instead of being written to the disk. Since no virtual processor can receive a message bigger than its data memory size, it is possible to keep most of the messages in this buffer. The exceptions are messages generated by the functions EM_Alltoall and EM_Alltoallv. In this type of communication, potentially all of the virtual processors are generating messages which are destined to all other virtual processors and each virtual processor may receive different sizes and numbers of messages. Since it is not possible to keep all of these messages in the shared memory buffer, they are written to the disk. The destination virtual processor will read the relevant messages from the disk when it becomes active. This requires maintaining information pertaining to where these messages are stored on the disks, as well as efficient retrieval mechanisms of the disk blocks corresponding to these messages.

We now turn to communication between virtual processors running on different physical machines. The sending virtual processor generates a message and calls the relevant PEMS function for communication. PEMS determines the MPI rank of the destination virtual processor and sends the message to the real processor on which it is hosted. The destination virtual process (or thread) may not be running at the time that the messages arrives, however. The current running virtual processor at the destination machine receives the messages on behalf of the destination virtual processor and saves the messages in the memory buffer or on the disk. The actual receiver can read the messages later whenever it becomes active.

The most complicated communication primitive is EM_Alltoallv. This primitive allows all virtual processors to send messages to all other virtual processors. The messages can be of different sizes and may be made up of multiple packets. The only restriction is that they need to fit into the local buffers of each virtual processor. As real processor p_i simulates $\frac{v}{p}$ virtual processors, in each round of communication it will receive all messages addressed to the virtual processors

which it simulates. It is possible, in our current implementation, that a user thread representing virtual processor v_{ij}, associated with the real processor p_i, is called on to handle many messages addressed to the virtual processors in p_i. The total size of these messages may be more than the total physical memory of p_i. More precisely, let the data memory of each virtual processor be μ_d, then the maximum message size is bounded by μ_d. There are $\frac{v}{p}$ virtual processors in each real processor p_i, so p_i may receive a total of $\min(p, \frac{v}{p}) \times \mu_d$ messages which may be beyond the real processor's RAM capacity. If we use the personalized communication algorithm by Bader et.al. [11] then the message size for each virtual processor is at most $\frac{\mu_d}{v} + \frac{v-1}{2}$. Then the maximum message size for each real processor is $O(\min(p, \frac{v}{p}) \times \frac{\mu_d}{v})$ which is $O(\mu_d)$. This is within reasonable limits and can be handled by a real processor. During the personalized communication the memory buffer is used as a working area for assembling and disassembling messages and packets.

3.1 Discussion on Design Choices

In this section we highlight some of the design alternatives for different components of the PEM system and provide some reasoning behind our choices.

Kernel Space Threads versus User Space Threads: In our implementation each virtual processor is simulated as a user space thread. We could have used kernel space threads. There are advantages and disadvantages in using one over the other. We decided to use the user space threads to shorten development time; they are easier to manage and synchronize, switching between threads is faster and coding and debugging is easier. We chose the GNU pth thread library for our implementation. This library is portable, has a Posix pthread interface as well as has its own specific interface.

Choices for Communication: We categorize two types of communication. Communication between virtual processors within a real processor is called *internal communication*, whereas communication between virtual processors on different processors is called *external communication*.
Option 1: We assign the job of communication (either internal or external) to a separate process (a kernel level thread). As virtual processors generate data for communication, they send them to the communication process, which decides if those data should be communicated internally or externally. Here we also need an interprocess communication mechanism to send data from the simulating process to the communication process.
Option 2: As the internal communication is fairly simple, each virtual processor can submit its internal communication to the DIS and send its external communication to the communication process as in Option 1.
Option 3: Internal communication is done as in Option 2 but external communication is done by the main thread of the process which simulates virtual processors. This avoids the need for interprocess communication. In this approach the main thread also has a synchronizing role. Before allowing the next

superstep to start, it waits until communication data from all virtual processors are written to the disk and all external communication data are received by peer processors and written to the disk or the memory buffer.
Option 4: Internal and external communication is done by the current running threads on all machines. Each running virtual processor calls the communication routines as subroutines. Communication routines classify messages, communicate with other running virtual processors and gather all the messages sent to their real processor (on behalf of all virtual processors being simulated in this real processor). We chose to use this approach in the current version of PEMS for the following reasons: First, messages to other machines are sent out as they are generated so there is no additional disk I/O or buffer activity for them. Second, there is no need for a separate process to do the communication. Third, each time virtual processors communicate through MPI, they can also go through a synchronization phase with other virtual processors and in fact all real processors can be synchronized without the need to wait for the main threads of the processors.

Direct and Asynchronous I/O: Direct I/O allows an I/O operation to be performed directly on a user space buffer without additional buffering by the operating system. Incorporating Linux direct I/O imposes constraints on the buffer size, alignment and also on the file offset especially when we want to use disk I/O functions such as pread or pwrite. The buffer size must be a multiple of the disk block size, it must also be aligned on a memory address which is a multiple of the disk block size. The same alignment constraint applies to file offset on a read or write operation. This complicates the PEMS versions of MPI-like collective primitives, as we often need to read and write at arbitrary file offsets and at arbitrary addresses in the buffer space. As a result, we decided to use direct I/O for swapping the context data and buffered I/O for communication purposes.

4 Experiments

Objectives: The main objective of our present work has been to see whether the simulation technique as proposed in [1, 2] is practical. At this stage of PEMS development, our experiments focus primarily on scalability of performance when the problem size and number of processors are varied. In order to shorten development time we have not yet placed much emphasis on optimizing our use of low level services such as asynchronous I/O and and kernel threads. While our experiments show running times for sorting with TPIE and STXXL, we include these measurements only as general reference points that highlight room for improvement in our single processor results.

Experimental Setup: Single processor tests are performed on the following configuration: AMD Opteron 2.4GHz CPU with 2GB of RAM, 3 Hard Disks each with at least 30GB of free space, two partitions across two disks configured as software RAID 0, hard disk bandwith is 71MB/Sec, RAID bandwith is 112

Mb/Sec (measured by hdparm utility), file system is EXT3, GNU/Linux 64bit v. 2.6.15 operating system and we used gcc v.4.1.1. Multiple processor experiments are performed on a cluster of Linux machines with the following hardware and software configuration[3]. Intel 2 × Xeon 2.0 GHz CPUs with 1.5 GB of RAM (only one processor was used in practice), one hard disk each with at least 46GB of free space, hard disk bandwith of 45Mb/Sec (measured by hdparm utility), GNU/Linux 32bit kernel version 2.6.9-42.0.2. ELsmp, EXT2 file system, gcc v 3.4.6 compiler and Gigabit ethernet for the communication between machines.

We use parallel sample sort to test our implementation. In our experiments we have simulated parallel sample sort [12] as an external memory algorithm. For comparison, we include timings from TPIE's [5] test_sort and STXXL's [6] test_sort1. We have slightly modified test_sort and test_sort1 functions to restrict them to one round of sorting (with no extra tests). All of the test programs use 128MB of RAM for sorting. The record size is 4 bytes and timing includes the data generation, but none of the test programs create a separate output file. PEMS uses 128MB of memory but 64MB of this memory is used as buffer and shared memory for communication, and only 64MB is used for sorting. (In PEMS, we have disabled the extra memory so that the operating system cannot use it for caching.)

Test data was generated using the C or C++ standard random generator or the package specific random generator method. We do not use any special integer sorting techniques in the parallel sample sort algorithm or in other test programs. Operating system swap has been disabled in all of the tests.

Experimental Results: Figure 2 shows running times for PEMS sample sort on 1, 2, 4, 8 and 16 real processors. It also includes running time of TPIE and STXXL external memory sort test programs on a single processor. We include the TPIE and STXXL results only as general reference points that highlight room for improvement in our single processor results. The reader should not draw conclusions about the relative running times of TPIE and STXXL sort, for instance, as we have not ensured that this is a fair comparison. The TPIE and STXXL sort programs are 3 to 5 times faster than our single processor sort with this version of PEMS, but as more real processors are added, PEMS becomes faster. At 7 billion integers, the running time of PEMS with 16 processors reduces by 65%, 53%, and 48%, in comparison to TPIE, STXXL and PEMS with 8 processors, respectively. The PEMS running times increase almost linearly with problem size when the number of real processors is fixed.

Limitations: An important limitation of PEMS is its internal disk usage. It needs $\frac{v}{p} \times \mu_d$ for swapping of data on each real processor. Here μ_d represents data memory of each virtual processor. It also reserves $2\frac{v}{p} \times \mu_d$ for communication on each real processor. We are currently investigating ways to reduce this requirement on the reservation of the disk space for the messages.

Concluding Remarks: PEMS is runtime library for creating parallel external memory programs from implementations of BSP-like coarse grained parallel al-

[3] This is part of HPCVL Lab www.hpcvl.org.

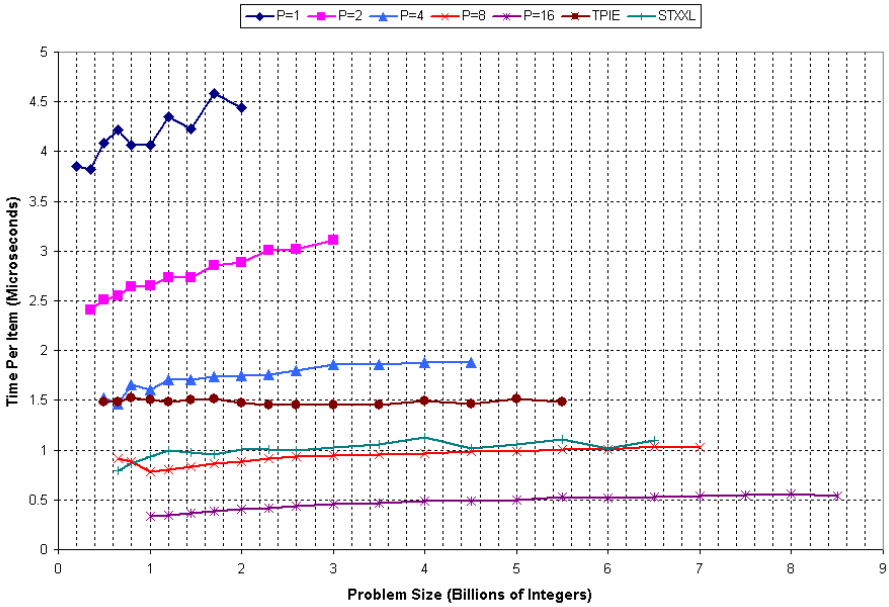

Fig. 2. Wall clock timings for sorting. The X-axis is the problem size in billions of integers. The Y-axis is wall clock time per sorted item in microseconds.

gorithms. The primary application area is problems that require processing of massive amounts of data. We see our work as relevant from several practical perspectives: a class of theoretically optimal parallel algorithms can be scaled to fit the hardware at hand, using both processors and disks; for a large set of problems for which suitable parallel algorithms exist, I/O optimal parallel algorithms can be used instead of single processor EM algorithms; and, both computational and I/O load are spread over multiple machines, contributing to the scalability.

Our experiments show several important properties of the PEMS approach. First of all the methodology is practical. Secondly, increasing the number of real processors decreases the running time in a predictable way. The ability to exploit parallel machine resources such as disks gives the ability to handle extremely large data sets on practical architectures such as a network of workstations. On such a system, one can inexpensively add computational power, RAM, disks, and bandwidth between RAM and disk by adding machines to a network. Using a coarse grained parallel algorithm and PEMS, our experiments in this paper suggest that one can take advantage of all of these resources, as well as adapting the theory of coarse grained parallel algorithms to the reality of a smaller number of real processors, each with a disk system. We believe that our experiments with sorting, for instance, showed good speedups in parallel running time primarily due to disk parallelism. More computationally intensive applications may be able to also make use of the additional computation power.

This brings us to several suggestions for further work: (1) PEMS algorithms have the disadvantage that they may do more I/O, than a conventional single processor EM algorithm due to the need to swap the contexts of virtual processors between RAM and disk. Our experiments suggest that this can be offset by the scalability of I/O bandwidth in our model. However, since I/O is so prevalent in PEMS we expect that improving the low level I/O performance may make a significant improvement in running times. To this end we plan to investigate the use of asynchronous (no-wait) I/O in PEMS. This would allow the overlapping of computation and I/O and the use of multiple independent parallel disks on each real processor. We noticed that STXXL [6] has a well designed and efficient asynchronous parallel disk I/O layer which can be used without calling its higher level functions. (2) We plan to further investigate the use of kernel threads in the asynchronous sending and receiving of communication traffic between virtual processors. (3) With multiple core CPUs becoming a commodity, adjustments should be considered for PEMS to take full advantage of symmetric multiprocessor machines. (4) In order to study the behavior of PEMS when simulating different algorithms more examples should be implemented. To this end we have implemented and tested a randomized list ranking algorithm and our preliminary results are promising (see [13]).

References

1. Dehne, F.K.H.A., Dittrich, W., Hutchinson, D.A., Maheshwari, A.: Bulk synchronous parallel algorithms for the external memory model. Theory Comput. Syst. **35**(6) (2002) 567–597
2. Hutchinson, D.A.: Parallel Algorithms in External Memory. PhD thesis, School of Computer Science, Carleton University (1999)
3. Vitter, J.S., Shriver, E.A.M.: Algorithms for parallel memory, I: Two-level memories. Algorithmica **12**(2–3) (1994) 110–147
4. Crauser, Mehlhorn: LEDA-SM: Extending LEDA to secondary memory. In: WAE: International Workshop on Algorithm Engineering, LNCS (1999)
5. TPIE: (`http://www.cs.duke.edu/TPIE/`)
6. Dementiev, R., Kettner, L., Sanders, P.: STXXL: Standard template library for XXL data sets. In: ESA. Volume 3669 of LNCS. (2005) 640–651
7. Gustedt, J.: Towards realistic implementations of external memory algorithms using a coarse grained paradigm. In: ICCSA (2). LNCS 2668 (2003) 269–278
8. Valiant, L.G.: A bridging model for parallel computation. Communications of the ACM **33**(8) (1990) 103–111
9. Open MPI: (`http://www.open-mpi.org/`)
10. GNU Pth - The GNU Portable Threads: (`http://www.gnu.org/software/pth/`)
11. Bader, D.A., Helman, D.R., JáJá, J.: Practical parallel algorithms for personalized communication and integer sorting. ACM JEA **1** (1996) 3
12. Shi, H., Schaeffer, J.: Parallel sorting by regular sampling. Journal of Parallel and Distributed Computing **14** (1992) 361–372
13. Nikseresht, M.R.: A parallel external memory system. Master's thesis, School of Computer Science, Carleton University (2007)