I/O-Efficient Scientific Computation Using TPIE

Darren Erik Vengroff*
Dept. of Electrical Engineering
Evans Hall
Univ. of Delaware
Newark, DE 19716
voice: (302) 831–2405
FAX: (302) 831–4316
email: vengroff@ee.udel.edu

Jeffrey Scott Vitter†
Dep. of Computer Science
Box 90129
Duke University
Durham, NC 27708–0129
voice: (919) 660–6548
FAX: (919) 660–6502
email: jsv@cs.duke.edu

July 25, 1996

Abstract

In recent years, I/O-efficient algorithms for a wide variety of problems have appeared in the literature. Thus far, however, systems specifically designed to assist programmers in implementing such algorithms have remained scarce. TPIE is a system designed to fill this void. It supports I/O-efficient paradigms for problems from a variety of domains, including computational geometry, graph algorithms, and scientific computation. The TPIE interface frees programmers from having to deal not only of explicit read and write calls, but also the complex memory management that must be performed for I/O-efficient computation.

In this paper, we discuss applications of TPIE to problems in scientific computation. We discuss algorithmic issues underlying the design and implementation of the relevant components of TPIE and present performance results of programs written to solve a series of benchmark problems using our current TPIE prototype. Some of the benchmarks we present are based on the NAS parallel benchmarks [5], while others are of our own creation.

We demonstrate that the CPU overhead required to manage I/O is small and that even with just a single disk the I/O overhead of I/O-efficient computation ranges from negligible to the same order of magnitude as CPU time. We conjecture that if we use a number of disks in parallel this overhead can be all but eliminated.

*Supported in part by the U.S. Army Research Office under grant DAAH04–93–G–0076 and by the National Science Foundation under grant DMR–9217290. Portions of this work were conducted while visiting the University of Michigan and Duke University.

†Supported in part by the National Science Foundation under grant CCR–9007851 and by the U.S. Army Research Office under grants DAAL03–91–G–0035 and DAAH04–93–G–0076.

1 Introduction

The Input/Output communication between fast internal memory and slower secondary storage is the bottleneck in many large-scale applications. The significance of this bottleneck is increasing as internal computation gets faster and parallel computing gains popularity [17]. CPU bandwidth is currently increasing at a rate of 40–60% per year, versus an annual increase in bandwidth of 7–10% for disk drives [18]. Main memory sizes are also increasing, but not fast enough to meet the needs of many large-scale applications. Additionally, main memory is roughly two orders of magnitude more expensive than disks. Thus, if I/O-efficient code can be written so as to provide performance near that obtained by solving the same problem on a machine with a much larger RAM, a great deal of money can be saved.

Up to this point, a great many I/O-efficient algorithms have been developed. The problems that have been considered include sorting and permutation-related problems [1, 2, 14, 15, 22], computational geometry [3, 4, 11, 23] and graph problems [7]. Until recently, there had been virtually no work directed at implementing these algorithms. Some work has now begun to appear [6, 19], but to the best of our knowledge no comprehensive package designed to support I/O-efficient programming across multiple platforms and problem domains has appeared. One goal of our ongoing research is to remedy this problem. Towards that end, we are developing TPIE, a transparent parallel I/O environment designed to facilitate the implementation of I/O-efficient programs.

In this work, we describe a series of experiments we have run using a prototype implementation of the TPIE interface. The experiments were chosen as models of common operations in scientific codes. Several of the experiments are on NAS parallel benchmarks designed to model large-scale scientific computation [5]. The results of our experiments demonstrate that I/O-efficient programs can be written using a high-level, portable, abstract interface, yet run efficiently.

In Section 2, we introduce the parallel I/O model of computation on which the algorithms that TPIE implements are based. In Section 3, we describe the TPIE system itself and the structure of our current prototype. In Section 4, we discuss the benchmarks we implemented, the algorithms that TPIE uses, and the performance of our implementations. Finally, we list a number of open problems worthy of examination in the continued pursuit of I/O-efficient computation.

2 The Parallel I/O Model of Computation

The algorithms TPIE uses are typically based on those designed for the parallel I/O model of computation [22]. This model abstractly represents a system having one or more processors, some fixed amount of main memory, and one or more independent disk drives. It is described by the following parameters:

$$N = \#\text{ of items in the problem instance}$$
$$M = \#\text{ of items that can fit into main memory}$$

B = # of items per disk block
D = # of disks.

We define an I/O operation, or simply an I/O for short, to the be process of transferring exactly one block of data to or from each disk. The I/O complexity of an algorithm is simply the number of I/Os it performs.

In discussing the I/O complexity of algorithms, we make every effort to avoid the use of big-Oh asymptotic notation. Instead, we are interested in as exact a figure as possible for the number of I/Os an algorithm will use for a given problem size and run-time environment. In some cases, the relative effect of rounding up certain quantities to the their integral ceilings can be significant, for example, when quantities round to small natural numbers. This effect is typically ignored when big-Oh notation is used. We are careful in this work to consider ceilings explicitly when their effect is significant; we use the ceiling notation $\lceil x \rceil$ to denote the smallest integer that is $\geq x$.

3 TPIE: A Transparent Parallel I/O Environment

TPIE [20, 21] is a system designed to assist programmers in the development of I/O-efficient programs for large-scale computation. TPIE is designed to be portable across a variety of platforms, including both sequential and parallel machines with both single and multiple I/O devices. Applications written using TPIE should continue to run unmodified when moved from one system that supports TPIE to another. In order to facilitate this level of portability, TPIE implements a moderately sized set of high-level access methods. The access methods were chosen based on their paradigmatic importance in the design of I/O-efficient algorithms. Using these access methods, we can implement I/O-efficient algorithms for many problems, including sorting [2, 15, 16, 22], permuting [8, 9, 10, 22], computational geometry problems [3, 4, 11, 23], graph-theoretic problems [7]. and scientific problems [8, 13, 22].

Because such a large number of problems can be solved using a relatively small number of paradigms, it is important that the access method implementations remain flexible enough to allow application programs a great deal of control over the functional details of the computation taking place within the fixed set of paradigms. To accomplish this, TPIE takes a somewhat non-traditional approach to I/O. Instead of viewing computation as an enterprise in which code reads data, operates on it and then writes results, we view it as a continuous process in which program objects are fed streams of data from an outside source and leave trails of results behind them. The distinction is subtle, but significant. In the TPIE model, programmers don't have to worry about making explicit calls to I/O subroutines or managing internal memory data structures in a run-time dependent environment. Instead, they merely specify the functional details of the computation they wish to perform within a given paradigm. TPIE then choreographs an appropriate sequence of data movements to keep the computation fed.

TPIE is implemented in C++ as a set of templated classes and functions and a run-time library. Currently, a prototype implementation supports access to data

stored on one or more disks attached to a workstation.[1] In the future, we plan to port the interface to larger multiprocessors and/or collections of workstations connected to a high-speed LAN. From the programmer's perspective, very little will change when the system moves to parallel hardware. All the same access methods will continue to exist, and applications will still be written with a single logical thread of control, though they will be executed in a data parallel manner.

The current TPIE prototype is a modular system with three components. The Access Method Interface (AMI) provides the high-level interface to the programmer. This is the only component with which most programmers will need to directly interact. The Block Transfer Engine (BTE) is responsible for moving blocks of data to and from the disk. It is also responsible for scheduling asynchronous read-ahead and write-behind when necessary to allow computation and I/O to overlap. Finally, the Memory Manager (MM) is responsible for managing main memory resources. All memory allocated by application programs or other components of TPIE is handled by the MM. In the case of application programs, this is facilitated through the use of a global `operator new()` in the TPIE library.

The AMI supports access methods including scanning, distribution, merging, sorting, permuting, and matrix arithmetic. In order to specify the functional details of a particular operation, the programmer defines a particular class of object called a scan management object. This object is then passed to the AMI, which coordinates I/O and calls member functions of the scan management object to perform computation. Readers interested in the syntactic details of this interaction are referred to the TPIE Manual, a draft version of which is currently available [21].

4 TPIE Performance Benchmarks

The benchmarks we implemented work with four of the basic paradigms TPIE supports: scanning, sorting, sparse matrices, and dense matrices. The benchmarks illustrate important characteristics not only of the TPIE prototype and the platform on which the tests were run, but also of I/O-efficient computation in general. In the exposition that follows we will discuss both.

Two of the benchmarks are based on the NAS parallel benchmarks set [5], which consists of kernels taken from applications in computational fluid dynamics. Besides being representative of scientific computations, these benchmarks also provide reference output values that can be checked to verify that they are implemented correctly. In addition to the NAS benchmarks, there are two new benchmarks designed to further exercise TPIE's matrix arithmetic routines.

Each of the benchmarks is accompanied by a graph illustrating the performance of one or more TPIE applications written to execute it. The graphs show both overall wall time and CPU time on the y-axis, as plotted against various problem sizes on the x axis. Given adequate and appropriately utilized I/O bandwidth, the wall time

[1] The following workstation/OS combinations are supported: Sun Sparcstation/SunOS 4.x, Sun Sparcstation/Solaris 5.x, DEC Alpha/OSF/1 1.x and 2.x, HP 9000/HP-UX.

and CPU time curves would be identical; therefore, getting them as close together as possible is an important performance goal.[2]

All of the benchmarks were run on a Sun Sparc 20 with a single local 2GB SCSI disk. The operating system was Solaris 5.3. Aside from the test application being run, the system was idle. In all cases, TPIE was configured to restrict itself to using 4 megabytes of main memory. I/O was performed in blocks of 64KB, with read-ahead and write-behind handled by a combination of TPIE and the operating system. The reason we used such a large block size was so that our computations would be structurally similar, in terms of recursion and read/write scheduling, to the same applications running on a machine with $D = 8$ disks and a more natural block size of 8KB. On such a system, I/O performance would increase by a factor of close to 8, whereas internal computation would be essentially unaffected.

4.1 Scanning

The most basic access method available in TPIE is scanning, which is implemented by the polymorphic TPIE entry point `AMI_scan()`. Scanning is the process of sequentially reading and/or writing a small number of streams of data. Essentially any operation that can be performed using $O(N/DB)$ I/Os can be implemented as a scan. This includes such operations as data generation, prefix sums, element-wise arithmetic, inner products, Graham's scan for 2-D convex hulls (once the points are appropriately sorted), selection, type conversion, stream comparison, and many others. The functional details of any particular scan are specified by a scan management object.

4.1.1 Scanning Benchmark

Because scanning is such a generic operation, we could have chosen any of a very wide variety of problems as a benchmark. We chose the NAS benchmark NAS EP [5] for two reasons: it was designed to model computations that actually occur in large-scale scientific computation; and it can be used to illustrate an important class of scan optimizations called scan combinations.

The NAS EP benchmark generates a sequence of independent pairs of Gaussian deviates. It first generates a sequence of $2N$ independent uniformly distributed deviates using the liner congruential method [12]. Then, it uses the polar method [12] to generate approximately $(\pi/4)N$ pairs of Gaussian deviates from the original sequence of uniform deviates.

Performance of our TPIE implementation of NAS EP is shown in Figure 1. There are three sets of curves, labeled "TPIE, 2 Scans," "TPIE, Optimized," and "Single Variable."

[2]One obvious way to bring these curves together is to increase the CPU time by performing additional or less efficient computation. Clearly, this is not the mechanism of choice. Instead, we seek to reduce the overall time by reducing the amount of I/O and/or improving increasing the overlap between computation and asynchronous I/O.

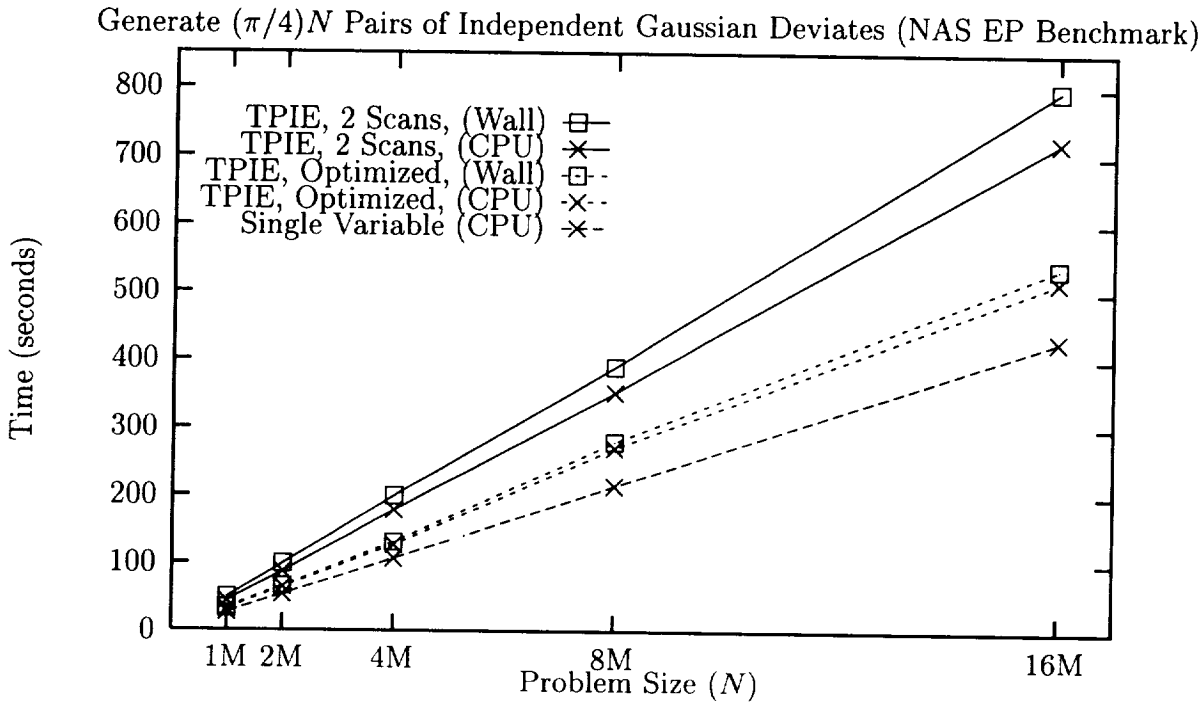

Figure 1: NAS EP Benchmark

The distinction between the 2 scan TPIE curves and the optimized TPIE curves is that in the former, two separate scans are performed, one to write the uniformly distributed random variates and the other to read the uniformly distributed random variates and write the Gaussian pairs, whereas in the latter, the two steps are combined into a single scan. As expected, the optimized code outperforms the unoptimized code.

This difference is significant not so much because it tells programmers they should combine scans, as because of the fact that scan combination is a relatively straightforward optimization that can be automated by a preprocessor. Such a preprocessor would parse the C++ text of a program and, where possible, construct hybrid scan management objects. The scans would then be replaced by a single scan using the hybrid object. Additionally, scans can often be piggy-backed on many other types of operations, such as merges, distributions, sorts, and permutations.

Returning to Figure 1, the single variable curve plots the CPU performance of a C++ program that does not perform any I/O at all, using TPIE or any other system. Instead, each pair of random variates is simply written over the previously generated pair in main memory. The purpose of this curve is to illustrate a fundamental lower bound on the CPU complexity of generating the variates. By comparing this to the CPU curves of the TPIE implementations, we can see that the CPU overhead associated with scheduling and performing I/O, communicating between the components of TPIE, and interacting with the user supplied scan management object is quite small. In the optimized case it amounts to approximately 20%.

4.2 Sorting

Sorting is a fundamental subroutine in many computations. There are a tremendous number of sorting algorithms which support many different models of computation and assumptions about input format and/or key distribution. In this section we discuss a number of issues related to sorting in external-memory, both theoretical and practical.

4.2.1 I/O-Efficient Sorting Algorithms

With rare exception[3], I/O-efficient comparison sorts fall into one of two categories, merge sorts and distribution sorts. Merge sorts work from the bottom up, sorting small subfiles and then combining them into successively larger sorted files until all the data is in one large sorted file. Distribution sorts work from the top down by computing medians in the data and then distributing the data into buckets based on the median values. The buckets are then recursively sorted and appended to one another to produce the final output. The I/O structure of radix sort resembles that of distribution sort, except that the entire set of keys is involved in $O((\lg N/M)/(\lg M/DB))$ large $O(M/DB)$-way distribution steps.

One common technique for dealing with multiple disks in parallel is to stripe data across them so that the heads of the D disks are moved in lock step with one another, thereby simulating a single large disk with block size DB. On a striped disk system, the I/O complexity of merge sorting N objects is

$$2\frac{N}{DB}\left\lceil \log_{(M/2DB)} N/M \right\rceil = 2\frac{N}{DB}\left\lceil \frac{\lg(N/M)}{\lg(M/2DB)} \right\rceil . \tag{1}$$

Each item is read once and written once in each pass, and all reads and writes are fully blocked. The logarithmic factor is the number of levels of recursion required to reduce merge subproblems of size M into the final solution of size N. Each stream is double buffered, hence we can merge $M/(2DB)$ streams at a time. If we are able to compute medians perfectly with no additional cost, as in the case where the keys are uniformly distributed, we can perform distribution sort in this same bound.

Asymptotically, the I/O bound (1) is not optimal for sorting. By using the D disks independently, we can do distribution sort in

$$2k\frac{N}{DB}\left\lceil \frac{\lg(N/M)}{\lg(M/2B)} \right\rceil \tag{2}$$

I/Os, where $k \geq 1$ is a constant whose value depends on the complexity of finding the medians, the quality of the medians as partitioning elements, and how evenly the buckets are distributed over the D disks. Although the the denominator in (2) is larger than the denominator in (1) by an additive term of $\lg D$, the leading constant factor in (2) is larger than that of (1) by a multiplicative factor of k. A number of

[3]For a recent example, see [3].

independent disk distribution sort algorithms exist [1, 15, 16, 22], with values of k ranging from approximately 3 to 20.

Before implementing an external sort on parallel disks, it is useful to examine the circumstances under which the I/O complexity (2) for using the disks independently is less than the I/O complexity (1) with striping. If we neglect the ceiling term for the moment, algebraic manipulation tells us that it is better to use disks independently when

$$\left(\frac{M}{2B}\right)^{\left(\frac{k-1}{k}\right)} < D.$$

Thus, D must be at least some root of $M/2B$. The critical issue now becomes the value of k. If $k = 1$ (i.e., if we do not need extra I/Os to compute $M/2B$ medians that partition the data evenly and if each resulting bucket is output evenly among the D disks), it is better to use disks independently. However, if $k = 4$, we need $D > (M/2B)^{3/4}$ in order for using disks independently to be worthwhile, which is not the case in current systems. For this reason, TPIE implements both merging and distribution in a striped manner. Work is continuing on developing practical methods that use disks independently.

Another important aspect of the behavior of I/O-efficient algorithms for sorting concerns the behavior of the logarithmic factor $\lceil \lg(N/M)/\lg(M/2DB) \rceil$ in the denominator of (1). The logarithmic term represents the number of merge passes in the merge sort, which is always integral, thus necessitating the ceiling notation. The ceiling term increases from one integer to the next when N/M is an exact power of $M/(2DB)$. Thus over very wide ranges of values of N, of the form $M^i/(2DB)^i < N/M \leq M^{i+1}/(2DB)^{i+1}$, for some integer $i \geq 1$, the I/O complexity of sorting remains linear in N. Furthermore, the possibility of $i \geq 3$ requires an extremely large value of N if the system in question has anything but the tiniest of main memories. As a result, although the I/O complexity of sorting is not, strictly speaking, linear in N, in practice it often appears to be.

4.2.2 Sorting Benchmark

The NAS IS benchmark is designed to model key ranking [5]. We are given an array of integer keys $K_0, K_1, \ldots K_{N-1}$ chosen from a key universe $[0, U)$, where $U \ll N$. Our goal is to produce, for each i, the rank $R(K_i)$, which is the position K_i would appear in if the keys were sorted. The benchmark does not technically require that the keys be sorted at any time, only that their ranks be computed. As an additional caveat, each key is the average of four random variates chosen independently from a uniform probability distribution over $[0, U)$. The distribution is thus approximately normal. Ten iterations of ranking are to be performed, and at the beginning of each iteration an extra key is added in each distant tail of the distribution.

In order to rank the keys, we sort them, scan the sorted list to assign ranks, and then re-sort based on the original indices of the keys. In the first sort, we do not have a uniform distribution of keys, but we do have a distribution whose probabilistic structure is known. Given any probabilistic distribution of keys with cumulative

distribution function (c.d.f.) F_K, we can replace each key value k_i by $k_i' = F_K(k_i)$ in order to get keys that appear as if chosen at random from a uniform distribution on $[0, 1]$. Because the keys of the NAS IS benchmark are sums of four independent uniformly distributed random variates, their c.d.f. is a relatively easy to compute piecewise fourth degree polynomial.

For the sake of comparison, we implemented this first sort in four ways, using both merge sort and three variations of distribution sorting. One distribution sort, called CDF1, assumed that the keys were uniformly distributed. The next CDF4, used the fourth degree c.d.f. mentioned above to make the keys more uniform. Finally, as a compromise, CDF2 used a quadratic approximation to the 4th degree c.d.f. based on the c.d.f. of the sum of two independent uniform random variables.

In the second sort, the indices are the integers in the range $[0, N)$, so we used a distribution sort in all cases. The rationale behind this was that distribution and merging should use the same amount of I/O in this case, but distribution should require less CPU time because it has no need for the main-memory priority queue that merge sorting requires.

The performance of the the various approaches is shown in Figure 2. As we expected, merge sort used more CPU time than any of the distribution sorts and the more complicated the c.d.f. we computed the more CPU time we used. When total time is considered, merge sort came out ahead of the distribution sorts. This appears to be the result of imperfect balance when the keys are distributed, which causes an extra level of recursion for a portion of the data. Interestingly, the quality of our c.d.f. approximation had little effect on the time spent doing I/O. We conjecture that this would not be the case with more skewed distributions, such as exponential distributions. We plan experiments to confirm this. The jump in the total time for the merge sort that occurs between 8M and 10M is due to a step being taken in the logarithmic term in that range.

4.3 Sparse Matrix Methods

Sparse matrix methods are widely used in scientific computations. A fundamental operation on sparse matrices is that of multiplying a sparse $N \times N$ matrix A by an N-vector x to produce an N-vector $z = Ax$.

4.3.1 Sparse Matrix Algorithms

Before we can work with sparse matrices in secondary memory, we need a way of representing them. In the algorithms we consider, a sparse matrix A is represented by a set of nonzero elements E. Each $e \in E$ is a triple whose components are $\text{row}(e)$, the row index of e in A, $\text{col}(e)$, the column index of e in A, and $\text{value}(e)$, the value of $A[\text{row}(e), \text{col}(e)]$.

In main memory, sparse matrix-vector multiplication can be implemented using Algorithm 1. If the number of nonzero elements of A is N_z, then Algorithm 1 runs in $O(N_z)$ time on a sequential machine.

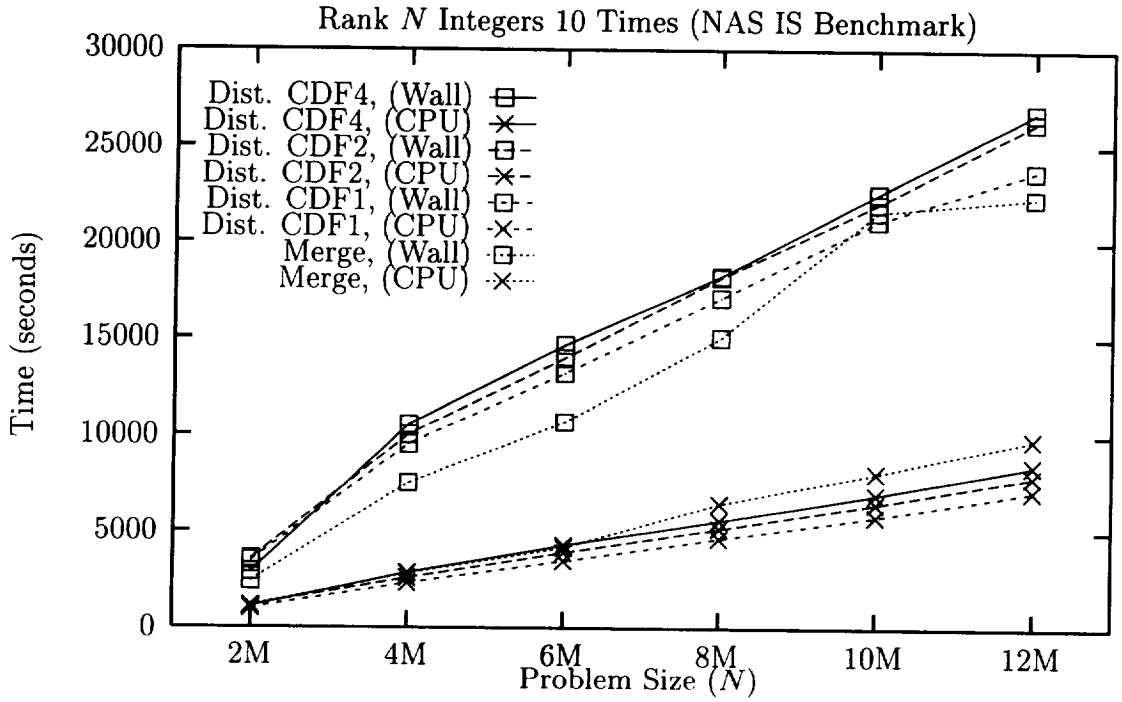

Figure 2: NAS IS benchmark performance

In secondary memory, we can also use Algorithm 1, but I/O performance depends critically on both the order in which the elements of A are processed and which of components of z and x are in main memory at any given time. In the worst case, every time we reference an object it could be swapped out. This would result in $3N_z$ I/Os.

In order to guarantee I/O-efficient computation, we reorder the elements of A in a preprocessing phase. In this preprocessing phase, A is divided into N/M separate $M \times N$ sub-matrices A_i, called bands. Band A_i contains all elements of A from rows iM to $(i+1)M-1$ inclusive. Although the dimensions of all the A_i are the same, the number of nonzero elements they contain may vary widely. To complete the preprocessing, the elements of each of the A_i are sorted by column.

Once A is preprocessed into bands, we can compute the output sub-vector

$$z[iM \ldots (i+1)M-1]$$

from A_i and x using a single scan, as shown in Algorithm 2. If we ignore the preprocessing phase for a moment and assume that the elements of x appear in order in external memory, the I/O complexity of Algorithm 2 is $N_z/DB + \lceil N/M \rceil N/DB + N/DB$. The entire preprocessing phase can be implemented as a single sort on the nonzero elements of A, with band index being the primary key and column being a secondary key. This takes $2N_z/DB \left\lceil \frac{\lg(N_z/M)}{\lg(M/2DB)} \right\rceil$ I/Os, as explained in Section 4.2.1. Note, however, that the preprocessing only has to be done a single time for a given matrix A. After that, the main phase of the algorithm can be executed repeatedly for many different vectors x. This is a common occurrence in iterative methods.

```
(1)  z ← 0;
(2)  foreach nonzero element e of A do
(3)        z[row(e)] = z[row(e)] + value(e) × x[col(e)];
(4)  endforeach
```

Algorithm 1: An algorithm for computing $z = Ax$ where A is a sparse $N \times N$ matrix and x and z are N-vectors.

4.3.2 The SMOOTH Benchmark

TPIE supports sparse matrices at a high level as a subclass of AMI streams. The nonzero elements of a sparse matrix are stored in the stream as (row, column, value) triples as described in the preceding section. AMI entry points for constructing sparse matrices as well as multiplying them by vectors are provided.

In order to test the performance of TPIE sparse matrices, we implemented a benchmark we call SMOOTH, which models a finite element computation on a 3-D mesh.

The SMOOTH benchmark implements sparse matrix-vector multiplication between a $N \times N$ matrix with $27N$ nonzero elements and a dense N-vector. The result is then multiplied by the matrix again. Ten iterations are performed.

The performance of SMOOTH is shown in Figure 3. Although we do ten iterations of multiplication, and only preprocess once, the total time with preprocessing is significantly higher that that of the multiplication iterations alone. As expected, I/O is not a major contributor to this difference, because sorting only requires a small number of linear passes through the data. The big difference is in CPU time. The additional CPU time used in preprocessing the sparse matrix is roughly twice the CPU time used in all ten iterations of the multiplication.

4.4 Dense Matrix Methods

Dense matrices appear in a variety of computations. Like sparse matrices, they are often multiplied by vectors, and banding techniques similar to those discussed in the previous section can by used. Another fundamental operation is multiplication of two $K \times K$ matrices A and B to produce $C = AB$.

4.4.1 Dense Matrix Algorithms

Asymptotically I/O-optimal multiplication of two $K \times K$ matrices over a quasiring can be done in $\Theta(K^3/\sqrt{M}DB)$ I/Os [22]. There are at least two simple algorithms that achieve this bound. The first algorithm, Algorithm 3, uses a recursive divide-and-conquer approach. The second algorithm, Algorithm 4, also partitions the input matrices, but all partitioning is done up front in a single permutation of each matrix.

```
    // Preprocessing phase:

(1) foreach nonzero element e of A do
(2)         Put e into A_{⌊row(e)/M⌋};
(3) endforeach
(4) for i ← 0, N/M do
(5)         Sort the elements of A_i by column;
(6) endfor

    // Main algorithm:

(7) Allocate a main memory buffer z_M of M words;
(8) for i ← 0 to ⌈N/M⌉ do
(9)         z_M ← 0;
(10)        foreach nonzero element e of A_i do
(11)                z_M[row(e) − iM] = z_M[row(e) − iM] + value(e) × x[col(e)];
(12)        endforeach
(13)        Write z_M to z[iM ... (i + 1)M − 1];
(14) endfor
```

Algorithm 2: An I/O-efficient algorithm for computing $z = Ax$ where A is a sparse $N \times N$ matrix and x and z are N-vectors.

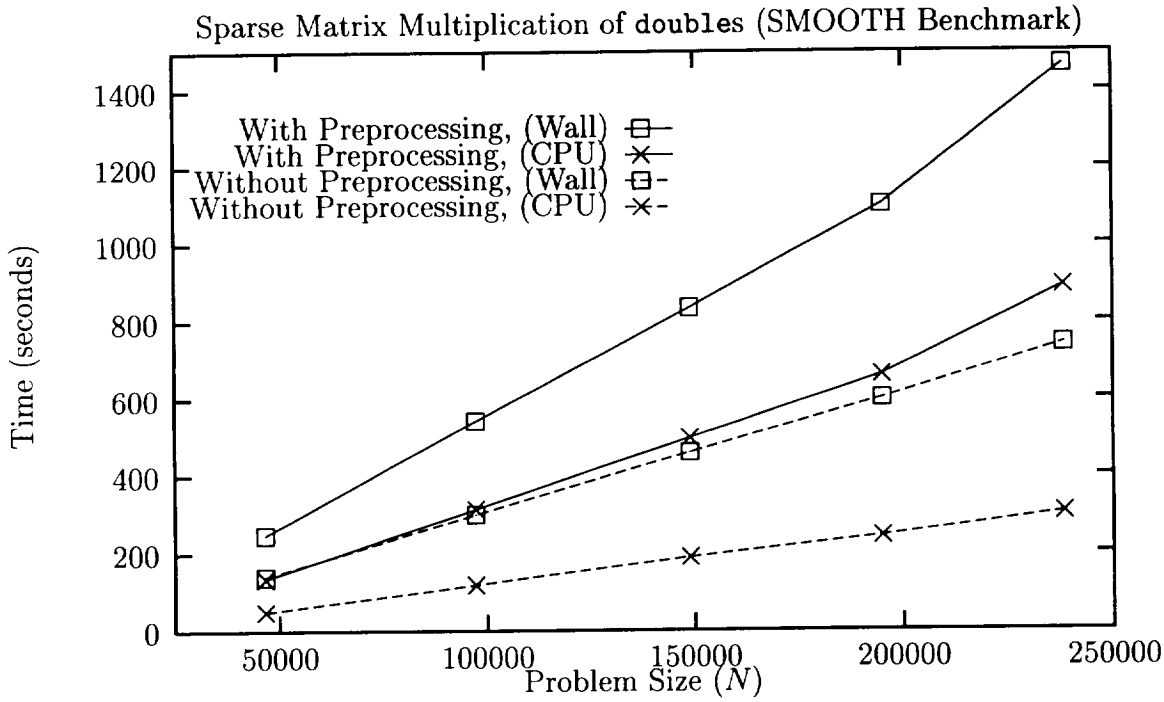

Figure 3: SMOOTH Benchmark

The matrix product is then produced iteratively, and a single final permutation returns it to canonical order. Both algorithms assume the input matrices are stored in row major order.

The I/O complexity of Algorithm 3 is

$$\frac{12\sqrt{3}K^3}{\sqrt{M}DB},$$

while that of Algorithm 4 is

$$\frac{2\sqrt{3}K^3}{\sqrt{M}DB} + \text{prep}(K),$$

where $\text{prep}(N)$ is the I/O complexity of the preprocessing and postprocessing steps, which can be done by sorting the K^2 elements of the three matrices, giving us

$$\text{prep}(K) = 6\frac{K^2}{DB}\left\lceil 2\frac{\lg(K/M)}{\lg(M/2DB)}\right\rceil.$$

In special circumstances, when B, D, K, and $\sqrt{M}$ are all integral powers of two, the pre- and post-processing are bit-matrix multiply complement permutations, which can be performed in fewer I/Os than sorting [10].

```
(1)  if 3K^2 ≤ M then
(2)      read A and B into main memory;
(3)      compute C = AB in main memory;
(4)      write C back to disk;
(5)  else
(6)      partition A at row K/2 and column K/2;
(7)      label the four quadrant sub-matrices A_{1,1}, A_{1,2}, A_{2,1}, and A_{2,2} as shown in Figure 4;
(8)      partition B into B_{1,1}, B_{1,2}, B_{2,1}, and B_{2,2} in a similar manner;
(9)      permute all sub-matrices of A and B into row major order;
(10)     Perform the (11)-(14) using recursive invocations of this algorithm
(11)         C_{1,1} ← A_{1,1}B_{1,1} + A_{1,2}B_{2,1};
(12)         C_{1,2} ← A_{1,1}B_{1,2} + A_{1,2}B_{2,2};
(13)         C_{2,1} ← A_{2,1}B_{1,1} + A_{2,2}B_{2,1};
(14)         C_{2,2} ← A_{2,1}B_{1,2} + A_{2,2}B_{2,2};
(15)     Reconstruct C from its sub-matrices C_{1,1}, C_{1,2}, C_{2,1}, and C_{2,2};
(16)     permute C back into row major order;
(17) endif
```

Algorithm 3: A recursive divide-and-conquer approach to matrix multiplication. Two $K \times K$ input matrices A and B are multiplied to produce $C = AB$.

4.4.2 Dense Matrix Benchmark

TPIE has high-level support for dense matrices over arbitrary user-defined quasirings. Operations supported include initialization, element-wise arithmetic, and matrix-matrix multiplication. Matrix-matrix multiplication uses Algorithm 4. Separate AMI entry points are available for the preprocessing permutation and the iterative multiplication itself, allowing a matrix to be preprocessed once and then multiplied by a number of other matrices.

We implemented a benchmark, called DENSE, which constructs two $K \times K$ matrices, preprocesses them, and then multiplies them. Times were recorded for both the total benchmark and for the multiplication only. The results are shown in Figure 6. As expected, the CPU time required to multiply the matrices follows a cubic path. Because of read-ahead, I/O is almost fully overlapped with computation, making the CPU and total time curves virtually indistinguishable. The cost of preprocessing the matrices is approximately one third of the cost of multiplying them. Thus if several multiplications are done with the same matrix amortization greatly reduces this cost.

5 Conclusions

We have presented a series of results demonstrating that I/O-efficient computation can be made practical for a variety of scientific computing problems. This computation is made practical by TPIE, which provides a high level interface to computational

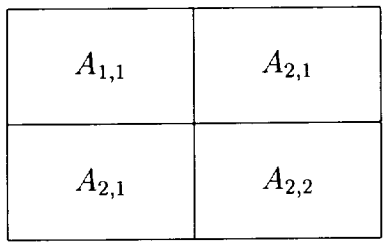

Figure 4: Partitioning a matrix into quadrants

(1) partition A into $K/\sqrt{M/3}$ rows and $K/\sqrt{M/3}$ columns of
sub-matrices each having $\sqrt{M/3}$ rows and $\sqrt{M/3}$ columns;
// step (1) is shown in Figure 5
(2) partition B in a manner similar to A;
(3) permute all $A_{i,j}$ and $B_{i,j}$ into row major order;
(4) **foreach** i, j **do**
(5) $C_{i,j} \leftarrow \sum_k A_{i,k} B_{k,j}$;
(6) **endforeach**
(7) reconstruct C from all $C_{i,j}$;
(8) permute C back into row major order;

Algorithm 4: An iterative approach to matrix multiplication.

paradigms whose I/O complexity has been carefully analyzed. Using TPIE results in only a small CPU overhead versus entirely in core implementation, but allows much larger data sets to be used. Additionally, for the benchmarks we implemented, I/O overhead ranged from being negligible to being of the same order of magnitude as internal computation time.

If we replace the single disk on which the tests were run with D disks, where D is on the order of 8, we conjecture that the I/O time required in our computations could be reduced by a factor very close to D. In applications like DENSE, where I/O overhead is already negligible, little would change, but in applications like NAS IS and NAS EP, we would see a dramatic reduction in I/O overhead. As discussed in Section 4, CPU time should not change appreciably. Recalling that a portion of the I/O that would be reduced by a factor of D is already overlapping with computation, we expect that in many case the I/O overhead (the portion that does not overlap with CPU usage) could be eliminated. We plan to assemble a parallel disk system to evaluate this conjecture experimentally.

In addition to the problems discussed here, there are many other scientific computations that we believe can benefit from careful analysis and I/O-efficient implementation. These include LUP decomposition, FFT computation, and multi-grid

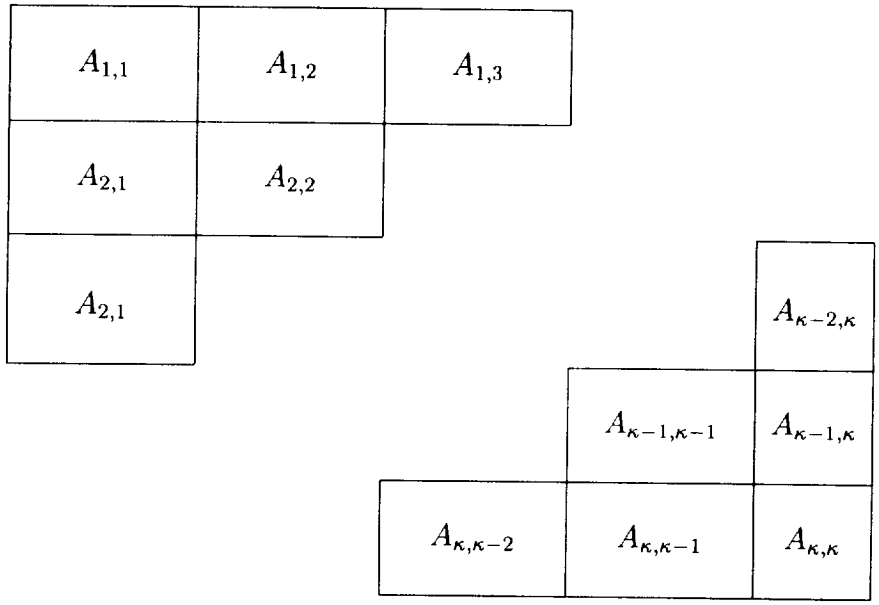

Figure 5: Partitioning a matrix into sub-matrices in step (1) of Algorithm 4. Each sub-matrix $A_{i,j}$ has $\sqrt{M/3}$ rows and $\sqrt{M/3}$ columns. The number of sub-matrices across and down A is $\kappa = K/\sqrt{M/3}$.

methods, all of which we plan to explore as the TPIE project continues. We also plan to investigate the construction of a scan combining preprocessor as described in Section 4.1.1.

Complementing this high level work, there are a number of potentially interesting I/O related research topics concerning how environments like TPIE should interact with operating systems. These include models of application controlled virtual memory and the behavior of TPIE applications in multiprogrammed environments where the main memory available to a given process may vary over time.

In closing, we are encouraged by the results we have presented, which demonstrate that I/O efficient computation using an abstract, high level model is practical. It is important to realize, however, that this research is only in its infancy, and that many more questions, both theoretical and practical, remain to be answered.

6 Acknowledgments

We thank Yi-Jen Chiang, Liddy Shriver, and Len Wisniewski for helpful discussions on implementing I/O-efficient algorithms. We also thank Jim Baker, Philip Klein, and Jon Savage for suggesting that we study sparse matrix methods.

We also thank Yale Patt and the staff of the ACAL lab at the University of Michigan for the facilities and support that allowed much of this research to take place.

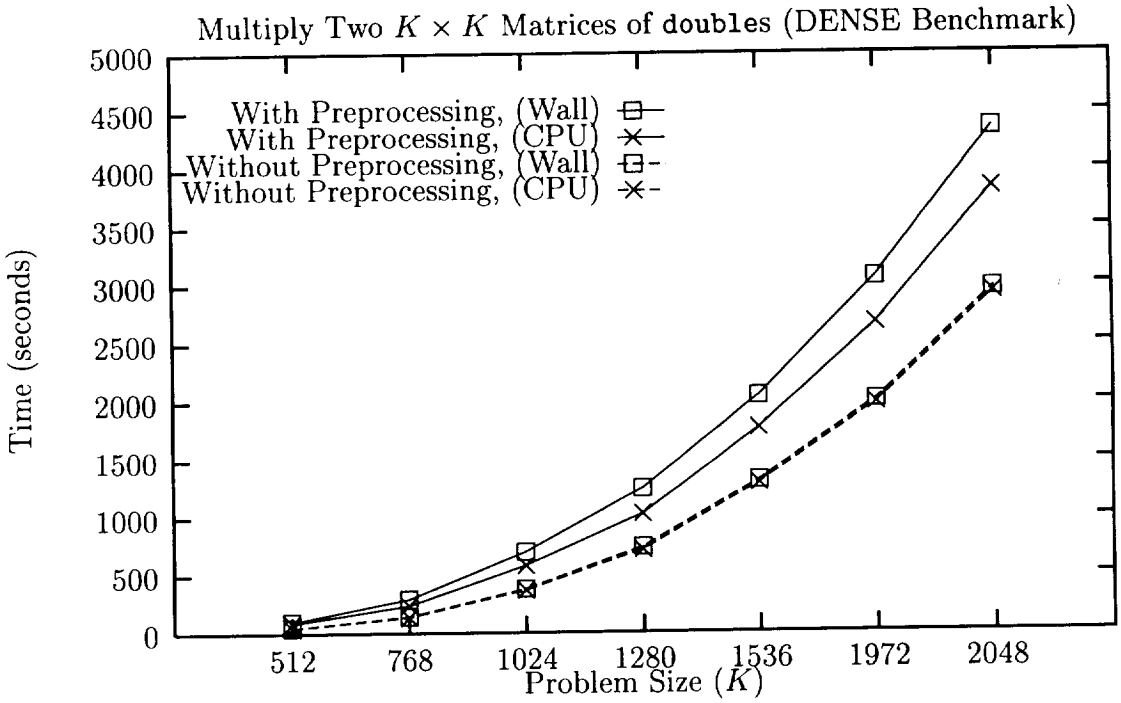

Figure 6: DENSE Benchmark

References

[1] A. Aggarwal and G. Plaxton. Optimal parallel sorting in multi-level storage. In *Proc. 4th Annual ACM-SIAM Symp. on Discrete Algorithms*, Arlington, VA, 1994.

[2] A. Aggarwal and J. S. Vitter. The input/output complexity of sorting and related problems. *Communications of the ACM*, 31(9):1116–1127, 1988.

[3] L. Arge. The buffer tree: A new technique for optimal I/O-algorithms. Technical Report RS-94-16, BRICS, Univ. of Aarhus, Denmark, 1994.

[4] L. Arge, D. E. Vengroff, and J. S. Vitter. External-memory algorithms for processing line segments in georgraphic information systems. submitted, 1995.

[5] D. Bailey, E. Barszcz, J. Barton, D. Browning, R. Carter, L. Dagum, R. Fatoohi, S. Fineberg, P. Fredrickson, T. Lansinki, R. Schrieber, H. Simon, V. Venkatakrishnan, and S. Weeratunga. The NAS parallel benchmarks. Technical Report RNR-94-007, RNR, March 1994.

[6] Y.-J. Chiang. Experiments on the practical I/O efficiency of geometric algorithms: Distribution sweep vs. plane sweep. In *Proc. 1995 Workshop on Algs. and Data Structures. (WADS)*, 1995.

[7] Y.-J. Chiang, M. T. Goodrich, E. F. Grove, R. Tamassia, D. E. Vengroff, and J. S. Vitter. External-memory graph algorithms. In *Proc. ACM-SIAM Symp. on Discrete Alg.*, pages 139–149, 1995.

[8] T. H. Cormen. *Virtual Memory for Data Parallel Computing.* PhD thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, 1992.

[9] T. H. Cormen. Fast permuting in disk arrays. *Journal of Parallel and Distributed Computing*, 17(1–2):41–57, Jan./Feb. 1993.

[10] T. H. Cormen, T. Sundquist, and L. F. Wisniewski. Asymptotically tight bounds for performing BMMC permutations on parallel disk systems. Technical Report PCS-TR94-223, Dartmouth College Dept. of Computer Science, July 1994.

[11] M. T. Goodrich, J.-J. Tsay, D. E. Vengroff, and J. S. Vitter. External-memory computational geometry. In *IEEE Foundations of Comp. Sci.*, pages 714–723, 1993.

[12] D. E. Knuth. *The Art of Computer Programming: Seminumerical Algorithms*, volume 2. Addison Wesley, 2d. ed. edition, 1981.

[13] C. E. Leiserson, S. Rao, and S. Toledo. Efficient out-of-core algorithms for linear relaxation using blocking covers. In *IEEE Foundations of Comp. Sci.*, pages 704–713, 1993.

[14] M. H. Nodine and J. S. Vitter. Large-scale sorting in parallel memories. In *Proc. 3rd ACM Symp. on Parallel Algorithms and Architectures*, pages 29–39, 1990.

[15] M. H. Nodine and J. S. Vitter. Deterministic distribution sort in shared and distributed memory multiprocessors. In *Proc. 5th ACM Symp. on Parallel Algorithms and Architectures*, June 1993.

[16] M. H. Nodine and J. S. Vitter. Paradigms for optimal sorting with multiple disks. In *Proc. of the 26th Hawaii Int. Conf. on Systems Sciences*, Jan. 1993.

[17] Y. N. Patt. The I/O subsystem—a candidate for improvement. *IEEE Computer*, 27(3):15–16, Mar. 1994.

[18] C. Ruemmler and J. Wilkes. An introduction to disk drive modeling. *IEEE Computer*, 27(3):17–28, Mar. 1994.

[19] E. A. M. Shriver and L. F. Wisniewski. Choreographed disk access using the whiptail file system: Applications. manuscript.

[20] D. E. Vengroff. A transparent parallel I/O environment. In *Proc. 1994 DAGS Symposium on Parallel Computation*, July 1994.

[21] D. E. Vengroff. *TPIE User Manual and Reference.* Duke University, 1995. Available via WWW at `http://www.cs.duke.edu:~dev/tpie.html`.

[22] J. S. Vitter and E. A. M. Shriver. Algorithms for parallel memory I: Two-level memories. *Algorithmica*, 12(2), 1994.

[23] B. Zhu. Further computational geometry in secondary memory. In *Proc. Int. Symp. on Algorithms and Computation*, 1994.