Designing Practical Efficient Algorithms for Symmetric Multiprocessors* (Extended Abstract)

David R. Helman and Joseph JáJá

Institute for Advanced Computer Studies &
Department of Electrical Engineering,
University of Maryland, College Park, MD 20742
{helman, joseph}@umiacs.umd.edu
www.umiacs.umd.edu/~{helman, joseph}

Abstract. Symmetric multiprocessors (SMPs) dominate the high-end server market and are currently the primary candidate for constructing large scale multiprocessor systems. Yet, the design of efficient parallel algorithms for this platform currently poses several challenges. In this paper, we present a computational model for designing efficient algorithms for symmetric multiprocessors. We then use this model to create efficient solutions to two widely different types of problems - linked list prefix computations and generalized sorting. Our novel algorithm for prefix computations builds upon the sparse ruling set approach of Reid-Miller and Blelloch. Besides being somewhat simpler and requiring nearly half the number of memory accesses, we can bound our complexity *with high probability* instead of merely *on average*. Our algorithm for generalized sorting is a modification of our algorithm for sorting by regular sampling on distributed memory architectures. The algorithm is a stable sort which appears to be asymptotically faster than any of the published algorithms for SMPs. Both of our algorithms were implemented in C using POSIX threads and run on four symmetric multiprocessors - the IBM SP-2 (High Node), the HP-Convex Exemplar (S-Class), the DEC AlphaServer, and the Silicon Graphics Power Challenge. We ran our code for each algorithm using a variety of benchmarks which we identified to examine the dependence of our algorithm on memory access patterns. In spite of the fact that the processors must compete for access to main memory, both algorithms still yielded scalable performance up to 16 processors, which was the largest platform available to us. For some problems, our prefix computation algorithm actually matched or exceeded the performance of the standard sequential solution using only a single thread. Similarly, our generalized sorting algorithm always beat the performance of sequential merge sort by at least an order of magnitude, even with a single thread.

* Supported in part by NSF grant No. CCR-9627210 and NSF HPCC/GCAG grant No. BIR-9318183. It also utilized the NCSA HP-Convex Exemplar SPP-2000 and the NCSA SGI/CRAY POWER CHALLENGEarray with the support of the National Computational Science Alliance under grant No. ASC970038N.

1 Introduction

Symmetric multiprocessors (SMPs) dominate the high-end server market and are currently the primary candidate for constructing large scale multiprocessor systems. Yet, the design of efficient parallel algorithms for this platform currently poses several challenges. The reason for this is that the rapid progress in microprocessor speed has left main memory access as the primary limitation to SMP performance. Since memory is the bottleneck, simply increasing the number of processors will not necessarily yield better performance. Indeed, memory bus limitations typically limit the size of SMPs to 16 processors. This has at least two implications for the algorithm designer. First, since there are relatively few processors available on an SMP, any parallel algorithm must be competitive with its sequential counterpart with as little as one processor in order to be relevant. Second, for the parallel algorithm to scale with the number of processors, it must be designed with careful attention to minimizing the number and type of main memory accesses.

In this paper, we present a computational model for designing efficient algorithms for symmetric multiprocessors. We then use this model to create efficient solutions to two widely different types of problems - linked list prefix computations and generalized sorting. Both problems are memory intensive, but in different ways. Whereas generalized sorting algorithms typically require a large number of memory accesses, they are usually to contiguous memory locations. By contrast, prefix computation algorithms typically require a more modest quantity of memory accesses, but they are are usually to non-contiguous memory locations.

Our novel algorithm for prefix computations builds upon the sparse ruling set approach of Reid-Miller and Blelloch [13]. Unlike the original algorithm, we choose the ruling set in such a way as to avoid the need for conflict resolution. Besides making the algorithm simpler, this change allows us to achieve a stronger bound on the complexity. Whereas Reid-Miller and Blelloch claim an *expected* complexity of $O\left(\frac{n}{p}\right)$ for $n >> p$, we claim a complexity *with high probability* of $O\left(\frac{n}{p}\right)$ for $n > p^2 \ln n$. Additionally, our algorithm incurs approximately half the memory costs of their algorithm, which we believe to be the smallest of any parallel algorithm we are aware of. Our algorithm for generalized sorting is a modification of our algorithm for sorting by regular sampling on distributed memory architectures [10]. The algorithm is a stable sort which appears to be asymptotically faster than any of the published algorithms we are aware of.

Both of our algorithms were implemented in C using POSIX threads and run on four symmetric multiprocessors - the IBM SP-2 (High Node), the HP-Convex Exemplar (S-Class), the DEC AlphaServer, and the Silicon Graphics Power Challenge. We ran our code for each algorithm using a variety of benchmarks which we identified to examine the dependence of our algorithm on memory access patterns. In spite of the fact that the processors must compete for access to main memory, both algorithms still yielded scalable performance up to 16 processors, which was the largest platform available to us. For some problems, our prefix computation algorithm actually matched or exceeded the performance of the standard sequential solution using only a single thread. Similarly, our generalized sorting algorithm always beat the performance of sequential merge sort by at least an order of magnitude, which from our experience is the best sequential sorting algorithm on these platforms.

The organization of our paper is as follows. **Section 2** presents our computational model for analyzing algorithms on symmetric multiprocessors. **Section 3** describes our prefix computation algorithm for this platform and its experimental performance. Similarly, **Section 4** describes our generalized sorting algorithm for this platform and its experimental performance.

2 A Computational Model for Symmetric Multiprocessors

For our purposes, the cost of an algorithm needs to include a measure that reflects the number and type of memory accesses. Given that we are dealing with a multi-level memory hierarchy, it is instructive to start with a brief overview of a number of models that have been proposed to capture the performance of multilevel hierarchical memories.

Many of the models in the literature are specifically limited to two-level memories. Aggarwal and Vitter [3] first proposed a simple model for main memory and disks which recognized the importance of spatial locality. In their uniprocessor model, a constant number of possibly non-contiguous blocks, each consisting of B contiguous records, can be transferred between primary and secondary memory in a single I/O. Vitter and Shriver [17] then proposed the more realistic D-disk model, in which secondary storage is managed by D physically distinct disk drives. In this model, D blocks can be transferred in a single I/O, but only if no two blocks are from the same disk. For both of these models, the cost of accessing data on disk was substantially higher than internal computation, and, hence, the sole measure of performance used is the number of parallel I/Os.

Alternatively, there are a number of models which allow for any arbitrary number of memory levels. Focusing on the fact that access to different levels of memory are achieved at differing costs, Aggarwal et al. [1] introduced the Hierarchical Memory Model (HMM), in which access to location x requires time $f(x)$, where $f(x)$ is any monotonic nondecreasing function. Taking note of the fact that the latency of memory access makes it economical to fetch a block of data, Aggarwal, Chandra, and Snir [2] extended this model to the Hierarchical Memory with Block Transfer Model (BT). In this model, accessing t consecutive locations beginning with location x requires time $f(x) + t$.

These models both assume that while the buses which connect the various levels of memory might be simultaneously active, this only occurs in order to cooperate on a single transfer. Partly in response to this limitation, Alpern et al. [4] proposed the Uniform Memory Hierarchy Model (UMH). In this model, the l^{th} memory level consists of $\alpha\rho^l$ blocks, each of size ρ^l, and a block of data can be transfered between level $l+1$ and level l in time $\rho^l/b(l)$, where $b(l)$ is the bandwidth. The authors of the UMH model stress that their model is an attempt to suggest what should be possible in order to obtain maximum performance. Certainly, the ability to specify the simultaneous, independent behavior of each bus would maximize computer performance, but as the authors acknowledge this is beyond the capability of current high-level programming languages. Hence, the UMH model seems unnecessarily complicated to describe the behavior of existing symmetric multiprocessors.

All the models mentioned so far focus on the relative cost of accessing different levels of memory. On the other hand, a number of shared memory models have focused instead on the contention caused by multiple processors competing to access main memory. Blelloch et al. [6] proposed the (d,x)-BSP model, an extension to the Bulk Synchronous Parallel model, in which main memory is partitioned amongst px banks. In this model, the time required for execution is modeled by five variables, which together describe the amount of time required for computation, the maximum number of memory requests made by a processor, and the maximum number of requests handled by a bank. The difficulty with this model is that the contention it describes depends on specific implementation details such as the memory map, which may be entirely beyond the control of the algorithm designer. A more general version of this model was suggested by Gibbons et al. [7]. Known as the Queue-Read Queue-Write (QRQW) PRAM model, it decomposes an algorithm into a series of synchronous steps. The time required for a given step is simply the maximum of the time required by any processor for computation, the number of memory accesses made by any processor, and the maximum number of requests made to any particular memory location. By focusing only on those requests which go to the same location, the QRQW model avoids implementation details such as the memory map, which makes it more appropriate as a high-level model. On the other hand, references which go to the same bank of memory but not to the same location can be just as disruptive to performance, and so ignoring details of the memory architecture can seriously limit the usefulness of the model.

In our SMP model, we acknowledge the dominant expense of memory access. Indeed, it has been widely observed that the rapid progress in microprocessor speed has left main memory access as the primary limitation to SMP performance. The problem can be minimized by insisting where possible on a pattern of contiguous data access. This exploits the contents of each cache line and takes full advantage of the pre-fetching of subsequent cache lines. However, since it does not always seem possible to direct the pattern of memory access, our complexity model needs to include an explicit measure of the number of non-contiguous main memory accesses required by an algorithm. Additionally, we recognize that efficient algorithm design requires the efficient decomposition of the problem amongst the available processors, and, hence, we also include the cost of computation in our complexity.

More precisely, we measure the overall complexity of an algorithm by the triplet $\langle M_A, M_E, T_C \rangle$, where M_A is the maximum number of accesses made by any processor to main memory, M_E is the maximum amount of data exchanged by any processor with main memory, and T_C is an upper bound on the local computational complexity of any of the processors. Note that M_A is simply a measure of the number of non-contiguous main memory accesses, where each such access may involve an arbitrary sized contiguous block of data. While we report T_C using the customary asymptotic notation, we report M_A and M_E as *approximations* of the actual values. By approximations, we mean that if C_A or C_V is described by the expression $\left(c_k x^k + c_{(k-1)} x^{(k-1)} + ... + c_0 x^0\right)$, then we report it using the approximation $\left(c_k x^k + o\left(x^k\right)\right)$. We distinguish between memory access cost and computational cost in this fashion because of the dominant expense of memory access on this architecture. With so few processors available, this coefficient is usually crucial in determining whether or not a parallel algorithm can be a viable replacement to the sequential alternative. On the other hand, despite the importance of these memory costs, we report only the highest order term, since otherwise the expression can easily become unwieldy.

In practice, it is often possible to focus on either M_A or M_E when examining the cost of algorithmic alternatives. For example, we observed when comparing prefix computation algorithms that the number of contiguous and non-contiguous memory accesses were always of the same asymptotic order, and therefore we only report M_A, which describes only the much more expensive non-contiguous accesses. Subsequent experimental analysis of the step-by-step costs has validated this simplification. On the other hand, algorithms for generalized sorting are usually all based on the idea of repeatedly merging sorted sequences, which are accessed in a contiguous fashion. Moreover, since our model is concerned only with the cost of main memory access, once values are stored in cache they may be accessed in any pattern at no cost. As a consequence, the number of non-contiguous memory accesses are always much less than the number of contiguous memory accesses, and in this situation we only report M_E, which includes the much more numerous contiguous memory accesses. Again, subsequent experimental analysis of the step-by-step costs has validated this simplification.

3 Prefix Computations

Consider the problem of performing a prefix computation on a linked list of n elements stored in arbitrary order in an array X. For each element X_i, we are given $X_i.succ$, the array index of its successor, and $X_i.data$, its input value for the prefix computation. Then, for any binary associative operator $\otimes$, the prefix computation is defined as:

$$X_i.\mathit{prefix} = \begin{cases} X_i.\mathit{data} & \text{if } X_i \text{ is the head of the list.} \\ X_i.\mathit{data} \otimes X_{(pre)}.\mathit{prefix} & \text{otherwise.} \end{cases}, \qquad (1)$$

where *pre* is the index of the predecessor of X_i. The last element in the list is distinguished by a negative index in its successor field, and nothing is known about the location of the first element.

Any of the known parallel prefix algorithms in the literature can be considered for implementation on an SMP. However, to be competitive, a parallel algorithm must contend with the extreme simplicity of the obvious sequential solution. A prefix computation can be performed by a single processor with two passes through the list, the first to identify the head of the list and the second to compute the prefix values. The pseudocode for this obvious sequential algorithm is as follows:

- **(1):** Visit each list element X_i in order of ascending array index. If X_i is not the terminal element, then label its successor with index $X_i.succ$ as having a predecessor.
- **(2):** Find the one element not labeled as having a predecessor by visiting each list element X_i in order of ascending array index - this unlabeled element is the head of the list.
- **(3):** Beginning at the head, traverse the elements in the list by following the successor pointers. For each element traversed with index i and predecessor *pre*, set *List[i].prefix_data* = *List[i].prefix_data* $\otimes$ *List[pre].prefix_data*.

To compute the complexity, note that Step (1) requires at most n non-contiguous accesses to label the successors. Step (2) involves a single non-contiguous memory access to a block of n contiguous elements. Step (3) requires at most n non-contiguous memory accesses to update the successor of each element. Hence, this algorithm requires approximately $2n$ non-contiguous memory accesses and runs in in $O(n)$ computation time.

According to our model, however, the obvious algorithm is not necessarily the best sequential algorithm. The non-contiguous memory accesses of Step (1) can be replaced by a single contiguous memory access by observing that the index of the successor of each element is a unique value between 0 and $n-1$ (with the exception of the tail, which by convention has been set to a negative value). Since only the head of the list does not have a predecessor, it follows that together the successor indices comprise the set $\{0, 1, .., h-1, h+1, h+2, .., n-1\}$, where h is the index of the head. Since the sum of the complete set $\{0, 1, .., n-1\}$ is given by $\frac{1}{2}n(n-1)$, it easy to see that the identity of the head can be found by simply subtracting the sum of the successor indices from $\frac{1}{2}n(n-1)$. The importance of this lies in the fact that the sum of the successor indices can be found by visiting the list elements in order of ascending array index, which according to our model requires only a single non-contiguous memory access. The pseudocode for this improved sequential algorithm is as follows:

- **(1):** Compute the sum Z of the successor indices by visiting each list element X_i in order of ascending array index. The index of head of the list is $h = \left(\frac{1}{2}n(n-1) - Z\right)$.
- **(2):** Beginning at the head, traverse the elements in the list by following the successor pointers. For each element traversed with index i and predecessor *pre*, set *List[i]._prefix_data* = *List[i].prefix_data* $\otimes$ *List[pre].prefix_data*.

Since this modified algorithm requires no more than approximately n non-contiguous memory accesses while running in $O(n)$ computation time, it is optimal according to our model.

The first fast parallel algorithm for prefix computations was probably the list ranking algorithm of Wyllie [18], which requires at least $n \log n$ non-contiguous accesses. Other parallel algorithms which improved upon this result include those of Vishkin [16] ($5n$ non-contiguous accesses), Anderson and Miller [5] ($4n$ non-contiguous accesses), and Reid-Miller and Blelloch [13] ($2n$ non-contiguous accesses - see [8] for details of this analysis). Clearly, however, none of these match the memory requirement of our optimal sequential algorithm.

3.1 A New Algorithm for Prefix Computations

The basic idea behind our prefix computation algorithm is to first identify the head of the list using the same procedure as in our optimal sequential algorithm. We then partition the input list into s sublists by randomly choosing exactly one *splitter* from each memory block of $\frac{n}{(s-1)}$ elements, where s is $\Omega(p \log n)$ (the list head is also designated as a splitter). Corresponding to each of these sublists is a record in an array called *Sublists*. We then traverse each of these sublists, making a note at each list element of the index of its sublist and the prefix value of that element within the sublist. The results of these sublist traversals are also used to create a linked list of the records in *Sublists*, where the input value of each node is simply the sublist prefix value of the last element in the previous sublist. We then determine the prefix values of the records in the *Sublists* array by sequentially traversing this list from its head. Finally, for each element in the input list, we apply the prefix operation between its current prefix input value (which is its sublist prefix value) and the prefix value of the corresponding *Sublists* record to obtain the desired result.

The pseudo-code of our algorithm is as follows, in which the input consists of an array of n records called *List*. Each record consists of two fields, *successor* and *prefix_data*, where *successor* gives the integer index of the successor of that element and *prefix_data* initially holds the input value for the prefix operation. The output of the algorithm is simply the *List* array with the properly computed prefix value in the *prefix_data* field. Note that as mentioned above we also make use of an intermediate array of records called *Sublists*. Each *Sublists* record consists of the four fields *head*, *scratch*, *prefix_data*, and *successor*, whose purpose is detailed in the pseudo-code.

- **(1):** Processor P_i $(0 \leq i \leq p-1)$ visits the list elements with array indices $\frac{in}{p}$ through $\left(\frac{(i+1)n}{p} - 1\right)$ in order of increasing index and computes the sum of the successor indices. Note that in doing this a negative valued successor index is ignored since by convention it denotes the terminal list element - this negative successor index is however replaced by the value $(-s)$ for future convenience. Additionally, as each element of *List* is read, the value in the successor field is preserved by copying it to an identically indexed location in the array *Succ*. The resulting sum of the successor indices is stored in location i of the array Z.
- **(2):** Processor P_0 computes the sum T of the p values in the array Z. The index of the head of the list is then $h = \left(\frac{1}{2}n(n-1) - T\right)$.

- **(3):** For $j = \frac{is}{p}$ up to $\left(\frac{(i+1)s}{p} - 1\right)$, processor P_i randomly chooses a location x from the block of list elements with indices $\left((j-1)\frac{n}{(s-1)}\right)$ through $\left(j\frac{n}{(s-1)} - 1\right)$ as a splitter which defines the head of a sublist in *List* (processor P_0 chooses the head of the list as its first splitter). This is recorded by setting *Sublists[j].head* to x. Additionally, the value of *List[x].successor* is copied to *Sublists[j].scratch*, after which *List[x].successor* is replaced with the value $(-j)$ to denote both the beginning of a new sublist and the index of the record in *Sublists* which corresponds to its sublist.
- **(4):** For $j = \frac{is}{p}$ up to $\left(\frac{(i+1)s}{p} - 1\right)$, processor P_i traverses the elements in the sublist which begins with *Sublists[j].head* and ends at the next element which has been chosen as a splitter (as evidenced by a negative value in the *successor* field). For each element traversed with index x and predecessor *pre* (excluding the first element in the sublist), we set *List[x].successor* = *-j* to record the index of the record in *Sublists* which corresponds to that sublist. Additionally, we record the prefix value of that element within its sublist by setting *List[x].prefix_data* = *List[x].prefix_data* ⊗ *List[pre].prefix_data*. Finally, if x is also the last element in the sublist (but not the last element in the list) and k is the index of the record in *Sublists* which corresponds to the successor of x, then we also set *Sublists[j].successor* = k and *Sublists[k].prefix_data* = *List[x].prefix_data*. Finally, the *prefix_data* field of *Sublists[0]*, which corresponds to the sublist at the head of the list is set to the prefix operator identity.
- **(5):** Beginning at the head, processor P_0 traverses the records in the array *Sublists* by following the successor pointers from the head at *Sublists[0]*. For each record traversed with index j and predecessor *pre*, we compute the prefix value by setting *Sublists[j].prefix_data* = *Sublists[j].prefix_data* ⊗ *Sublists[pre].prefix_data*.
- **(6):** Processor P_i visits the list elements with array indices $\frac{in}{p}$ through $\left(\frac{(i+1)n}{p} - 1\right)$ in order of increasing index and completes the prefix computation for each list element x by setting *List[x].prefix_data* = *List[x].prefix_data* ⊗ *Sublists[-(List[x].successor)].prefix_data*. Additionally, as each element of *List* is read, the value in the successor field is replaced with the identically indexed element in the array *Succ*. Note that is reasonable to assume that the entire array of s records which comprise *Sublists* can fit into cache.

We can establish the complexity of this algorithm with high probability - that is with probability $\geq (1 - n^{-\epsilon})$ for some positive constant ϵ. But before doing this, we need the results of the following Lemma, whose proof has been omitted for brevity [8].

Lemma 1. *The number of list elements traversed by any processor in Step (4) is at most* $\alpha\frac{n}{p}$ with high probability, *for any* $\alpha(s) \geq 2.62$ *(read* $\alpha(s)$ *as "the function* α *of* s*"),* $s \geq (p \ln n + 1)$*, and* $n > p^2 \ln n$*.*

With this result, the analysis of our algorithm is as follows. In Step (1), each processor moves through a contiguous portion of the list array to compute the sum of the indices in the *successor* field and to preserve these indices by copying them to the array *Succ*. When this task is completed, the sum is written to the array Z. Since this is done in order of increasing array index, it requires only three non-contiguous memory accesses to exchange approximately $\frac{2n}{p}$ elements with main memory and $O\left(\frac{n}{p}\right)$ computation time. In Step (2), processor P_0 computes the sum of the p entries in the array Z. Since this is done in order of increasing array index, this step requires only a single non-contiguous memory accesses to exchange p elements with main memory and $O(p)$ computation time. In Step (3), each processor randomly chooses $\frac{s}{p}$ splitters to be the heads of sublists. For each of these sublists, it copies the index of the corresponding record in the *Sublists* array into the successor field of the splitter. While the *Sublists* array is traversed in order of increasing array index, the corresponding splitters may lie in mutually non-contiguous locations and so the whole process may require $\frac{s}{p}$ non-contiguous memory accesses to exchange $\frac{2s}{p}$ elements with main memory and $\frac{s}{p}$ computation time. In Step (4), each processor traverses the sublist associated with each of its $\frac{s}{p}$ splitters, which together contain at most $\alpha(s)\frac{n}{p}$ elements *with high probability*. As each sublist is completed, the prefix value of the last element in the subarray is written to the record in the *Sublists* array which corresponds to the succeeding sublist. Since we can reasonably assume that $(s << n)$ and can therefore ignore the cost of writing to the *Sublists* array, this step requires approximately $\alpha(s)\frac{n}{p}$ non-contiguous memory accesses to exchange approximately $\alpha(s)\frac{n}{p}$ elements with main memory and $O\left(\frac{n}{p}\right)$ computation time *with high probability* . However, it is important to note that an $\frac{s}{n}$-biased binomial process requires *on average* $\frac{n}{s}$ events before encountering the first success and so *on average* each processor traverses about $\frac{n}{p}$ list elements (which is what we observe experimentally in the next section). In Step (5), processor P_0 traverses the the linked list of s records in the *Sublists* array established in Step (4) to compute their prefix values, which requires s non-contiguous memory accesses to exchange s elements with main memory and $O(s)$ computation time. Finally, in Step (6), each processor completes the prefix values for a contiguous chunk of the input list by first looking up the prefix value of the record in *Sublists* which maps to the head of its sublist. Since we make the reasonable assumption that the entire array of s records which comprise *Sublists* will fit into the cache, which is the case for all three platforms considered in this paper and the choices for n, accessing the prefix values in the *Sublists* array will only require s non-contiguous memory accesses (non-contiguous because we are assuming they are accessed in the order of request). As the computation of the prefix value for an element is completed, the correct value is restored to its *successor* field from the array *Succ*. Hence, this step will require approximately $(s+1)$ non-contiguous memory accesses to exchange approximately $\frac{2n}{p}$ elements with main memory and $O\left(\frac{n}{p}\right)$ computation time. Thus, *with high probability*, the overall complexity of our prefix computation algorithm is given by

$$T(n, p) = \langle M_A(n, p); M_E(n, p); T_C(n, p) \rangle \tag{2}$$

$$= \left\langle \alpha(s)\frac{n}{p}; \left((\alpha(s) + 4)\frac{n}{p}\right); O\left(\frac{n}{p}\right)\right\rangle \tag{3}$$

for $\alpha(s) \geq 2.62$, $s \geq (p \ln n + 1)$, $n >> s$, and $n > p^2 \ln n$. Noting that the relatively expensive M_A non-contiguous memory accesses comprise a substantial proportion of the M_E total elements exchanged with memory, and recalling that *on average* each processor traverses only about $\frac{n}{p}$ elements in Step (4), we would expect that in practice the complexity of our algorithm can be characterized by

$$T(n, p) = \langle M_A(n, p); T_C(n, p) \rangle \tag{4}$$

$$= \left\langle \frac{n}{p}; O\left(\frac{n}{p}\right)\right\rangle, \tag{5}$$

Notice that our algorithm's requirement of approximately n non-contiguous memory accesses is nearly half the cost of Reid-Miller and Blelloch and compares very closely with the requirements of the optimal sequential algorithm.

3.2 Performance Evaluation

Both our parallel algorithm and the optimal sequential algorithm were implemented in C using POSIX threads and run on an IBM SP-2 (High Node), an HP-Convex Exemplar (S-Class), a DEC AlphaServer 2100A system, and an SGI Power Challenge. To evaluate these algorithms, we examined the prefix operation of of floating point addition on three different benchmarks, which were selected to compare the impact of various memory access patterns. These benchmarks are the **Random [R]**, in which each successor is randomly chosen, the **Stride [S]**, in which each successor is (wherever possible) some stride S away, and the **Ordered [O]**, in which which element is paced in the array according to its rank. See [8] for a more detailed description and justification of these benchmarks.

The graphs in Fig. 1 compare the performance of our optimal parallel prefix computation algorithm with that of our optimal sequential algorithm. Notice first that our parallel algorithm almost always outperforms the optimal sequential algorithm with only one or two threads. The only exception is the [O] benchmark, where the successor of an element is always the next location in memory. Notice also that for a given algorithm, the [O] benchmark is almost always solved more quickly than the [S] benchmark, which in turn is always solved more quickly than the [R] benchmark. A step by step breakdown of the execution time for the HP-Convex Exemplar in Table 1 verifies that these differences are entirely due to the time required for the sublist traversal in Step (4). This agrees well with our theoretical expectations, since in the [R] benchmark, the location of the successor is randomly chosen, so almost every step in the traversal involves accessing a non-contiguous location in memory. By contrast, in the [O] benchmark, the memory location of the successor is always the successive location in memory, which in all likelihood is already present in cache. Finally, the [S] benchmark is designed so that where possible the successor is always a constant stride away. Since for our work this stride is chosen to be 1001, we might expect that each successive memory access would be to a non-contiguous memory location, in which case the [S] benchmark shouldn't perform any better than the [R] benchmark. However, cache modeling reveals that as the the number of samples increases, the number of cache misses decreases. Hence, for large value of s, the majority of requests are already present in cache, which explains the superior performance of the [S] benchmark. Finally, notice that, in Table 1, the n noncontiguous memory required by the [R] benchmark in Step (4) consume on average almost five time as much time as the $4n$ contiguous memory accesses of Steps (1) and (6). Taken as a whole, these results strongly support the emphasis we place on minimizing the number of non-contiguous memory accesses in this problem.

Table 1. Comparison of the time (in seconds) required as a function of the benchmark for each step of computing the prefix sums of 4M list elements on an HP-Convex Exemplar, for a variety of threads.

Step:	**Number of Threads & Benchmark**														
	[1]			**[2]**			**[4]**			**[8]**			**[16]**		
	[R]	**[S]**	**[O]**	**[R]**	**[S]**	**[O]**	**[R]**	**[S]**	**[O]**	**[R]**	**[S]**	**[O]**	**[R]**	**[S]**	**[O]**
(1)-(3):	0.59	0.87	0.66	0.34	0.40	0.34	0.18	0.21	0.18	0.10	0.12	0.10	0.08	0.08	0.08
(4):	6.69	1.86	2.33	3.40	1.08	1.17	1.75	0.57	0.59	0.96	0.31	0.30	0.74	0.22	0.18
(5):	0.01	0.12	0.01	0.01	0.04	0.01	0.01	0.05	0.01	0.01	0.06	0.01	0.01	0.02	0.01
(6):	0.69	0.75	0.69	0.37	0.38	0.35	0.21	0.20	0.19	0.11	0.12	0.11	0.09	0.12	0.08
Total:	7.97	3.60	3.68	4.12	1.91	1.87	2.14	1.03	0.97	1.19	0.60	0.52	0.92	0.41	0.35

The graphs in Figs. 2 and 3 examine the scalability of our prefix computation algorithm as a function of the number of threads. Results are shown for a variety of problem sizes on both the HP-Convex Exemplar and the the IBM SP-2 using the [R] benchmark. Bearing in mind that these graphs are log-log plots, they show that for large enough inputs, the execution time decreases as we increase the number of threads p, which is the expectation of our model. For smaller inputs and larger numbers of threads, this inverse relationship between the execution time and the number of threads deteriorates. In this case, such performance is quite reasonable if we consider the fact that for small problem sizes the size of the cache approaches that of the problem. This introduces a number of issues which are beyond the intended scope of our computational model.

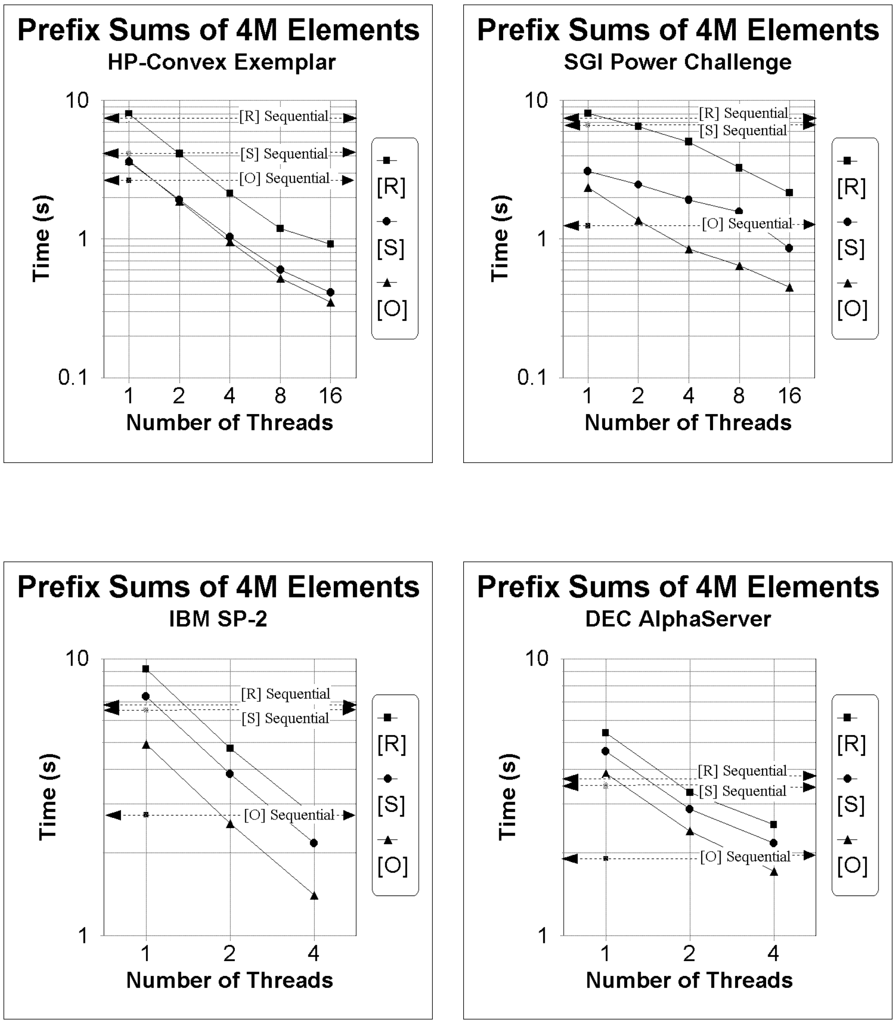

Fig. 1. Comparison between the performance of our parallel algorithm and our optimal sequential algorithm on four different platforms using three different benchmarks.

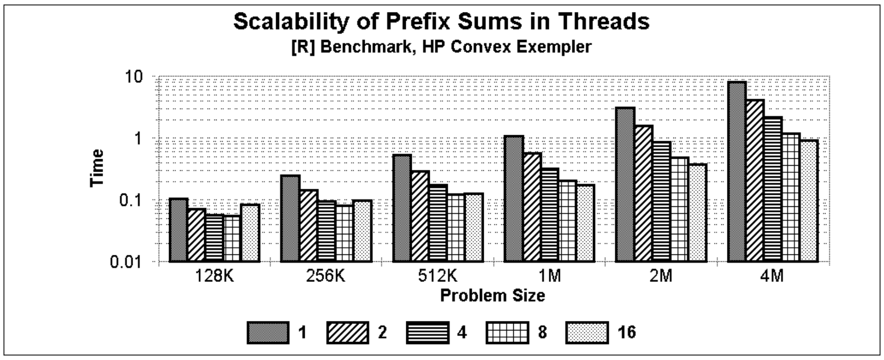

Fig. 2. Scalability of our prefix computation algorithm on the HP-Convex Exemplar with respect to the number of threads, for differing problem sizes.

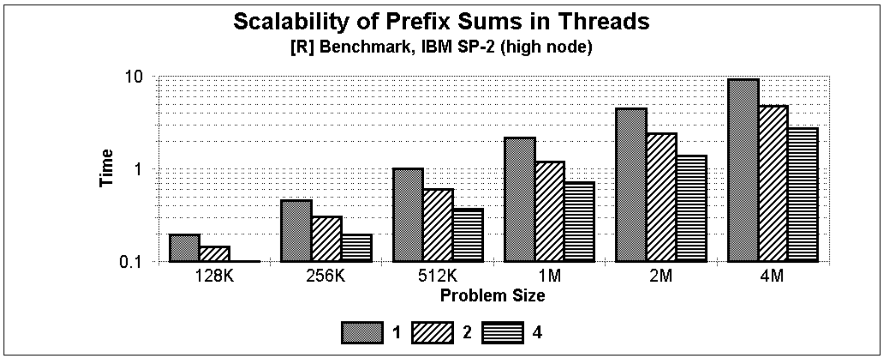

Fig. 3. Scalability of our prefix computation algorithm on the IBM SP-2 with respect to the number of threads, for differing problem sizes.

4 Generalized Sorting

Consider the problem of sorting n elements equally distributed amongst p processors, where we assume without loss of generality that p divides n evenly. Any of the algorithms that have been proposed in the literature for sorting on hierarchical memory models can be considered for possible implementation on an SMP. However, without modifications, most are unnecessarily complex or inefficient for a relatively simple platform such as ours. A notable exception is the algorithm of Varman et al. [15]. Yet another approach is an adaptation of our sorting by regular sampling algorithm [10, 9], which we originally developed for distributed memory machines.

4.1 A New Algorithm for Generalized Sorting

The idea behind sorting by regular sampling is to first partition the n input elements into p memory-contiguous blocks and then sort each of these blocks using an appropriate sequential algorithm. Then, a set of $p-1$ *splitters* is found to partition each of these p sorted sequences into p subsequences indexed from 0 up to $(p-1)$, such that every element in the i^{th} group is less than or equal to each of the elements in the $(i+1)^{th}$ group, for $(0 \leq i \leq p-2)$. Then the task of sorting merging the p subsequences with a particular index can be turned over to the correspondingly indexed processor, after which the n elements will be arranged in sorted order. One way to choose the splitters is by regularly sampling the input elements - hence the name Sorting by Regular Sampling. As modified for an SMP, this algorithm is similar to the parallel sorting by regular sampling (PSRS) algorithm of Shi and Schaeffer [14]. However, unlike their algorithm, our algorithm accommodates the presence of duplicate values without the overhead of tagging each element.

However, while our algorithm will efficiently partition the work amongst the available processors, it will not be sufficient to minimize main memory accesses unless we also carefully specify how the sequential tasks are to be performed. Specifically, straightforward binary merge sort or quick sort will require $\log \frac{n}{p}$ memory accesses for each element to be sorted. Thus, a more efficient sequential sorting algorithm running on a single processor can be expected to outperform a parallel algorithm running on the relatively few processors available with an SMP, unless the sequential steps of the parallel algorithm are properly optimized. Knuth [11] describes a better approach for the analogous situation of external sorting. First, each processor partitions its $\frac{n}{p}$ elements to be sorted into blocks of size $\frac{C}{2}$, where C is the size of the cache, and then sorts each of these blocks using merge sort. This alone eliminates $\log\left(\frac{C}{4}\right)$ memory accesses for each element. Next, the sorted blocks are merged z at a time using a tournament of losers, which further reduces the memory accesses by a factor of $\log z$. To be efficient, the parameter z must be set less than $\frac{C}{L}$, where L is the cache line size, so that the cache can hold the entire tournament tree plus a cache line from each of the z blocks being merged. Otherwise, as our experimental evidence demonstrates, the memory performance will rapidly deteriorate. Note that this approach to sequential sorting was simultaneously utilized by LeMarca and Ladner [12], though without noting this important limitation on the size of z.

The pseudocode for our algorithm is as follows:

- **(1)** Each processor P_i $(0 \leq i \leq p-1)$ sorts the subsequence of the n input elements with indices $\left(\frac{in}{p}\right)$ through $\left(\frac{(i+1)n}{p} - 1\right)$ as follows:
  - **(A)** Sort each block of m input elements using sequential merge sort, where $m \leq \frac{C}{2}$.
  - **(B)** For $j = 0$ up to $\left(\frac{\log(n/pm)}{\log(z)} - 1\right)$, merge the sorted blocks of size (mz^j) using z-way merge, where $z < \frac{C}{L}$.

– **(2)** Each processor P_i selects each $\left(\frac{in}{p} + (j+1)\frac{n}{ps}\right)^{th}$ element as a sample, for $(0 \leq j \leq s-1)$ and a given value of s $\left(p \leq s \leq \frac{n}{p^2}\right)$.
– **(3)** Processor $P_{(p-1)}$ merges the p sorted subsequences of samples and then selects each $((k+1)s)^{th}$ sample as Splitter$[k]$, for $(0 \leq k \leq p-2)$. By default, the p^{th} splitter is the largest value allowed by the data type used. Additionally, binary search is used to compute for the set of samples with indices 0 through $((k+1)s-1)$ the number of samples Est$[k]$ which share the same value as Splitter$[k]$.
– **Step (4):** Each processor P_k uses binary search to define an index $b_{(i,k)}$ for each of the p sorted input sequences created in Step (1). If we define $T_{(i,k)}$ as a subsequence containing the first $b_{(i,k)}$ elements in the i^{th} sorted input sequence, then the set of p subsequences $\{T_{(0,k)}, T_{(1,k)}, ..., T_{((p-1),k)}\}$ will contain all those values in the input set which are strictly less than Splitter$[k]$ and *at most* $\left(\text{Est}[k] \times \frac{n}{ps}\right)$ elements with the same value as Splitter$[k]$. The term *at most* is used because there may not actually be this number of elements with the same value as Splitter$[k]$.
– **Step (5):** Each processor P_k merges those subsequences of the sorted input sequences which lie between indices $b_{(i,(k-1))}$ and $b_{(i,k)}$ using p-way merge.

Before establishing the complexity of this algorithm, we need the results of the following lemma, whose proof has been omitted for brevity [10]:

Lemma 2. *At the completion of the partitioning in Step (4), no more than* $\left(\frac{n}{p} + \frac{n}{s} - p\right)$ *elements will be associated with any splitter, for* $\left(p \leq s \leq \frac{n}{p^2}\right)$ *and* $n \geq ps$.

With this result, the analysis of our algorithm is as follows. In Step (1A), each processor moves through a contiguous portion of the input array to sort it in blocks of size m using sequential merge sort. If we assume that $\left(m \leq \frac{C}{2}\right)$, this will require only a single non-contiguous memory accesses to exchange $\frac{2n}{p}$ elements with main memory and $O\left(\frac{n}{p}\log m\right)$ computation time. Step (1B) involves $\frac{\log(n/pm)}{\log(z)}$ rounds of z-way merge. Since round j will begin with $\frac{n}{pmz^j}$ blocks of size mz^j, this will require at most $\frac{2nz}{pm(z-1)}$ non-contiguous memory accesses to exchange $\frac{2n\log(n/pm)}{p\log(z)}$ elements with main memory memory and $O\left(\frac{n}{p}log\left(\frac{n}{pm}\right)\right)$ computation time. The selection of s noncontiguous samples by each processor in Step (2) requires s non-contiguous memory accesses to exchange $2s$ elements with main memory and $O(s)$ computation time. Step (3) involves a p-way merge of blocks of size s followed by p binary searches on segments of size s. Hence, it requires approximately $p\log(s)$ non-contiguous memory accesses to exchange approximately $2sp$ elements with main memory and $O(sp\log p)$ computation time. Step (4) involves p binary searches by each processor on segments of size $\frac{n}{p}$ and hence requires approximately $p\log\left(\frac{n}{p}\right)$ non-contiguous memory accesses to exchange approximately $p\log\left(\frac{n}{p}\right)$ elements with main memory and $O\left(p\log\left(\frac{n}{p}\right)\right)$

computation time. Step (5) involves a p-way merge of p sorted sequences whose combined length from Lemma (2) is at most $\left(\frac{n}{p} + \frac{n}{s} - p\right)$. This requires approximately p non-contiguous memory accesses to exchange approximately $2\left(\frac{n}{p} + \frac{n}{s}\right)$ elements with main memory and $O\left(\frac{n}{p}\right)$ computation time. Hence, the overall complexity of our sorting algorithm is given by

$$\begin{aligned} T(n,p) &= \langle M_A(n,p); M_E(n,p); T_C(n,p)\rangle \\ &= \left\langle \left(\frac{nz}{pm(z-1)} + s + p\log\left(\frac{n}{p}\right)\right); \right. \\ &\qquad \left. \left(\left(2\frac{\log(n/pm)}{\log(z)} + 2\right)\frac{n}{p} + 2\frac{n}{s}\right); O\left(\frac{n}{p}\log n\right)\right\rangle \end{aligned} \tag{6}$$

for $\left(p \le s \le \frac{n}{p^2}\right)$, $n \ge ps$, $m \le \frac{C}{2}$, and $z \le \frac{C}{L}$. Since the analysis suggests that the parameters m and z should be as large as possible subject to the stated constraints while selecting s so that $\left(p << s << \frac{n}{p}\right)$, we would expect that in practice the complexity of our algorithm could be characterized as

$$T(n,p) = \langle M_E(n,p); T_C(n,p)\rangle \tag{7}$$

$$= \left\langle \left(4 + 2\frac{\log(n/pm)}{\log(z)}\right)\frac{n}{p}; O\left(\frac{n}{p}\log n\right)\right\rangle. \tag{8}$$

4.2 Performance Evaluation

Our sorting algorithm was implemented in C using POSIX threads and run on an IBM SP-2 (High Node), an HP-Convex Exemplar (S-Class), a DEC AlphaServer 2100A system, and an SGI Power Challenge. We ran our code using six widely different double precision floating point benchmarks which were selected to test the dependence of our algorithm on the input distribution. A detailed description and justification of these benchmarks is presented in [9]. The results in Table 2 verify that as expected performance does not significantly depend on the input distribution. Because of this independence, the remainder of this section will only discuss performance on the single benchmark [U], in which the input data forms a uniform random distribution.

Table 2. Sorting doubles (in seconds) using 4 threads on a DEC AlphaServer 2100A.

Input Size	Benchmark					
	[U]	[G]	[Z]	[WR]	[DD]	[RD]
512K	0.397	0.394	0.320	0.421	0.337	0.348
1M	0.868	0.856	0.741	0.844	0.724	0.710
2M	1.64	1.72	1.39	1.73	1.40	1.51
4M	3.50	3.47	3.00	3.52	3.01	2.98

Table 3 displays the times required to sort 4M *doubles* (i.e. double precision floating point values) on the HP-Convex Exemplar using a single thread as a function of m and z. Notice first that performance suffers dramatically when the block size reaches 1MB (128K eight byte double precision numbers), which is the limit of the single level cache on the Exemplar. This is expected, since sorting a block in Step (1A) now requires that data be repeatedly swapped to main memory. Consider also the data for a given block size - say 1K. The execution time drops as we move from $z = 2$ to $z = 16$. This is reasonable since we require 12 rounds of 2-way merge, 6 rounds of 4-way merge, 4 rounds of 8-way merge, and only 3 rounds of 16-way merge, and each round of z-way merge is obviously another round where all the input elements must be brought in from main memory. Moving from $z = 16$ to $z = 32$ has little effect on the execution time since it does nothing to reduce the memory requirements, but moving to $z = 64$ saves a round of memory access and, hence, the execution time is further reduced. However, the most dramatic illustration of the importance of minimizing secondary memory access can be found by comparing the optimal sorting time of 15.86 seconds for $m = 2K$ and $z = 2048$ with the time of 39.25 seconds required to sort using only binary merge sort. Reducing memory access by a combination of block sorting and z-way merging improved the performance by 60%. Clearly, such results strongly support the attention that we place in this algorithm on the number of contiguous memory accesses.

Table 3. Time (in seconds) required on the HP-Convex Exemplar to sort 4M doubles using a single thread as a function of M and z.

Block	**Denomination of z-Way Merge**											
Size	**2**	**4**	**8**	**16**	**32**	**64**	**128**	**256**	**512**	**1024**	**2048**	**4096**
1K	30.43	21.49	19.22	17.66	17.87	16.20	16.74	16.91	16.87	16.82	16.73	16.29
2K	29.14	21.65	19.15	18.02	17.95	16.63	16.91	16.98	16.79	18.29	15.86	
4K	27.96	20.55	19.62	18.29	16.65	17.00	17.14	17.06	16.86	15.91		
8K	27.59	21.55	19.18	19.27	17.90	18.07	18.04	17.84	17.08			
16K	26.69	20.73	19.84	18.21	18.50	18.53	18.43	17.83				
32K	27.77	23.14	22.14	20.81	20.88	20.90	20.41					
64K	29.98	25.46	24.26	24.51	24.51	23.06						
128K	37.54	34.19	33.26	33.36	31.84							
256K	39.85	36.51	36.74	35.37								
512K	39.78	37.81	36.54									
1M	39.53	37.62										
2M	39.25											
4M	38.86 - (No z-way merge is necessary for this block size)											

A slightly more complicated picture of the role m and z emerges from Table 4, which displays the times required to sort 4M doubles on the IBM SP-2 using a single thread as a function of m and z. Again, performance suffers dramatically when the block size reaches 1MB (128K eight byte double precision numbers), which is the limit of the secondary cache on this platform. But consider the data for a given block size - say 256. The execution time drops as we move from $z = 2$ to $z = 128$. This is reasonable since we require 14 rounds of 2-way merge, 7 rounds of 4-way merge, 5 rounds of 8-way merge, 4 rounds of 16-way, 3 rounds of 32-way merge and 64-way merge, and only 2 rounds of 128-way merge, and each round of z-way merge is obviously another round where all the input elements must be brought in from main memory. We would then expect that moving from $z = 128$ to $z = 16384$ would have little effect on the execution time since it does nothing to reduce the memory requirements, but this turns out not to be the case. The explanation lies in recalling that, unlike the Exemplar, the SP-2 has both a primary and a secondary cache. An efficient implementation of the z-way merge in Step (1B) would fill this 16 KB 4-way set associative primary cache with the entire tree of losers (z 12 byte records) plus a cache line (32 bytes) from each of the z sequences being merged. For $z = 256$, this primary cache is essentially filled, and cache misses to secondary cache become an issue. Finally, note the difference between the optimal sorting time of 10.24 seconds for $m = 256$ and $z = 128$ with the time of 28.29 seconds required to sort using only binary merge sort. Here, reducing memory access by a combination of block sorting and z-way merging improved the performance by 64%. Again, such results strongly support the attention that we place in this algorithm on the number of contiguous memory accesses.

Table 4. Time (in seconds) required on the IBM SP-2 to sort 4M doubles using a single thread as a function of M and z.

Block	**Denomination of z-Way Merge**														
Size	**2**	**4**	**8**	**16**	**32**	**64**	**128**	**256**	**512**	**1024**	**2048**	**4096**	**8192**	**16384**	**32768**
128	23.72	15.98	13.40	12.13	11.17	11.34	11.55	10.34	10.85	12.06	13.83	15.43	17.18	20.81	20.03
256	22.71	14.86	13.45	12.32	11.20	11.51	10.24	10.59	11.26	12.27	13.71	15.37	18.37	17.12	
512	21.70	15.07	13.66	12.51	11.39	11.71	10.47	11.02	11.58	12.27	13.71	15.33	15.34		
1K	20.74	14.10	12.68	11.43	11.68	10.50	10.82	11.13	11.37	12.19	13.64	13.60			
2K	20.05	14.62	13.00	11.78	12.13	11.00	11.31	11.42	11.67	12.44	12.23				
4K	20.45	14.93	14.49	13.38	12.31	12.54	12.64	12.88	13.04	12.08					
8K	19.84	15.58	14.00	13.91	12.84	13.10	13.26	13.40	12.68						
16K	19.51	15.18	14.58	13.46	13.61	13.82	13.94	12.65							
32K	19.70	16.62	15.90	14.71	15.18	15.22	13.99								
64K	21.13	17.68	17.17	17.26	17.79	16.66									
128K	24.23	21.81	20.73	21.04	19.89										
256K	26.45	24.42	24.62	23.55											
512K	27.95	27.00	26.00												
1M	28.10	27.26													
2M	28.16														
4M	28.29 - (No z-way merge is necessary for this block size)														

The graphs in Figs. 4 and 5 examines the scalability of our sorting algorithm as a function of the number of threads, for a variety of problem sizes. Bearing in mind that these graphs are log-log plots, they show that for large enough inputs, the execution time decreases as we increase the number of threads p, which is the expectation of our model. For smaller inputs on the HP-Convex Exemplar, this inverse relationship between the execution time and the number of threads deteriorates when we move to 16 threads. This explanation for this problem may lie in the fact that when we moved to 16 threads on this platform, the data suddenly became very erratic, perhaps because some threads now had to compete with operating system processes for access to one of the 16 processors.

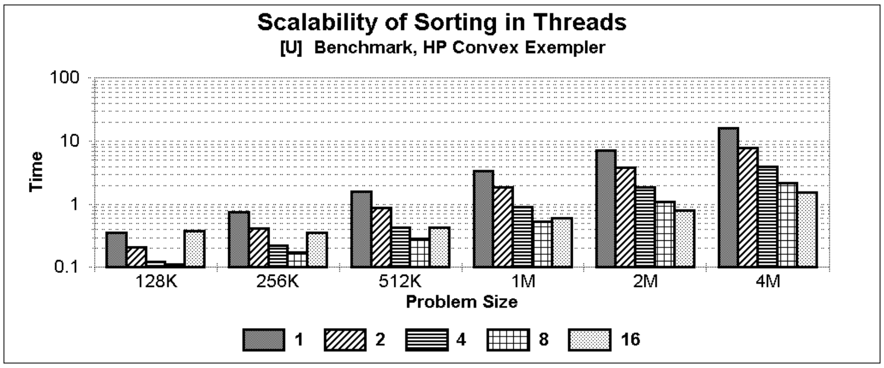

Fig. 4. Scalability of our generalized sorting algorithm on the HP-Convex Exemplar with respect to the number of threads, for differing problem sizes.

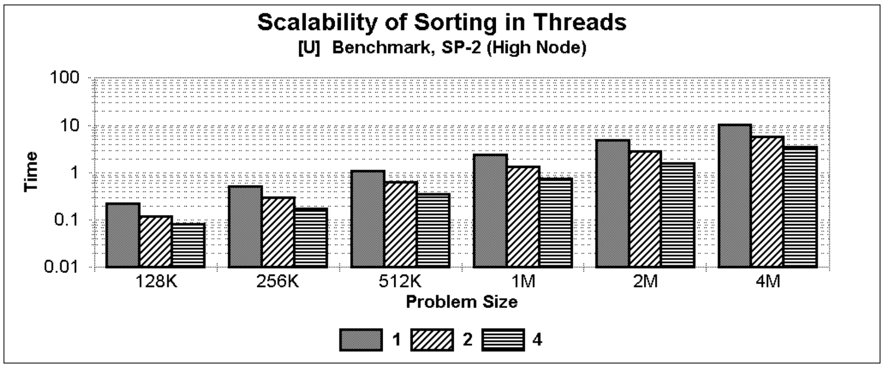

Fig. 5. Scalability of our generalized sorting algorithm on the IBM SP-2 with respect to the number of threads, for differing problem sizes.

References

1. A. Aggarwal, B. Alpern, A. Chandra, and M. Snir. A Model for Heirarchical Memory. In *Proceedings of the 19th Annual ACM Symposium of Theory of Computing*, pages 305–314, May 1987.
2. A. Aggarwal, A. Chandra, and M. Snir. Heirarchical Memory with Block Transfer. In *Proceedings of the 28th Annual IEEE Symposium on Foundations of Computer Science*, pages 204–216, October 1987.
3. A. Aggarwal and J. Vitter. The Input/Output Complexity of Sorting and Related Problems. *Communications of the ACM*, 31:1116–1127, 1988.
4. B. Alpern, L. Carter, E. Feig, and T. Selker. The Uniform Memory Hierarchy Model of Compuatation. *Algorithmica*, 12:72–109, 1994.
5. R. Anderson and G. Miller. Deterministic Parallel List Ranking. In *Proceedings Third Aegean Workshop on Computing, AWOC 88*, pages 81–90, Corfu, Greece, June/July 1988. Springer-Verlag.
6. G.E. Blelloch, P.B. Gibbons, Y. Matias, and M. Zagha. Accounting for Memory Bank Contention and Delay in High-Bandwidth Multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 8(9):943–958, 1997.
7. P.B. Gibbons, Y. Matias, and V. Ramachandran. The Queue-Read Queue-Write PRAM Model: Accounting for Contention in Parallel Algorithms. *SIAM Journal on Computing*, 1997. To appear.
8. D.R. Helman and J. JáJá. Prefix Computations on Symmetric Multiprocessors. Technical Report CS-TR-3915 and UMIACS-TR-98-38, UMIACS and Electrical Engineering, University of Maryland, College Park, MD, July 1998.
9. D.R. Helman and J. JáJá. Sorting on Clusters of SMPs. In *Proceedings of the 12th International Parallel Processing Symposium*, Orlando, Florida, April 1998.
10. D.R. Helman, J. JáJá, and D.A. Bader. A New Deterministic Parallel Sorting Algorithm With an Experimental Evaluation. *ACM Journal of Experimental Algorithmics*, 3(4):1–24, 1998.
11. D.E. Knuth. *The Art of Computer Programming: Sorting and Searching*, volume 3. Addison-Wesley Publishing Company, Reading, MA, 1973.
12. A. LaMarca and R. Ladner. The Influence of Caches on the Performance of Sorting. In *roceedings of the Eighth Annual Symposium on Discrete Algorithms*, pages 370–377, January 1997.
13. M. Reid-Miller. List Ranking and List Scan on the Cray C90. *Journal of the Computer and System Sciences*, 53:344–356, 1996.
14. H. Shi and J. Schaeffer. Parallel Sorting by Regular Sampling. *Journal of Parallel and Distributed Computing*, 14:361–372, 1992.
15. P. Varman, B. Iyer, and S. scheufler. A Multiprocessor Algorithm for Merging Multiple Sorted Lists. In *Proceedings of the 1990 International Conference on Parallel Processing*, pages 22–26.
16. Uzi Vishkin. Randomized Speed-Ups in Parallel Computation. In *Proceedings of the Sixteenth ACM Symposium on Theory of Computing*, pages 230–239, Washington, D.C., 1984.
17. J. Vitter and E. Shriver. Algorithms for Parallel Memory I: Two-Level Memories. *Algorithmica*, 12:110–147, 1994.
18. J.C. Wyllie. *The Complexity of Parallel Computations*. PhD thesis, Department of Computer Science, Cornell University, Ithica, NY, 1979.