Prefix Computations on 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

Abstract

We introduce a new optimal prefix computation algorithm on linked lists which 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. Moreover, whereas Reid-Miller and Blelloch targeted their algorithm for implementation on a vector multiprocessor architecture, we develop our algorithm for implementation on the symmetric multiprocessor architecture (SMP). These symmetric multiprocessors dominate the high-end server market and are currently the primary candidate for constructing large scale multiprocessor systems. Our prefix computation algorithm was 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 using a variety of benchmarks which we identified to examine the dependence of our algorithm on memory access patterns. For some problems, our algorithm actually matched or exceeded the performance of the standard sequential solution using only a single thread. Moreover, in spite of the fact that the processors must compete for access to main memory, our algorithm still achieved scalable performance with up to 16 processors, which was the largest platform available to us.

*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 CHALLENGE array with the support of the National Computational Science Alliance under grant No. ASC970038N.

1 Introduction

Prefix computations on linked structures are basic operations used for the manipulation of lists, trees, and graph data structures. However, beyond their obvious utility, the computational tasks involved are well-known to defy practical parallel implementation on coarse grain architectures, especially for a relatively small number of processors. The source of the difficulty lies in the fact that there is no obvious way to estimate the relative proximity of two list elements except by essentially solving the overall problem. Thus, there is no way to partition the list elements amongst the various processors that will avoid either irregular communication or memory access patterns. To make matters worse, there is very little computation with which to mask these costs. Finally, any parallel solution must compete against the obvious sequential solution, which is extremely simple with very small constants.

In this paper, we introduce a new prefix computation algorithm which builds upon the sparse ruling set approach of Reid-Miller and Blelloch [7]. 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 $n > p^2 \ln n$. At the same time, 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. Finally, whereas Reid-Miller and Blelloch targeted their algorithm for implementation on a vector multiprocessor architecture, we develop our algorithm for implementation on the symmetric multiprocessor architecture (SMP). The advantage of vector multiprocessors is the high

global communication bandwidth and the pipelined memory access. Indeed, as long as there are no memory bank conflicts, the network can service one memory request per clock cycle for each memory pipe. However, despite this advantage, recent trends in multiprocessor architecture have placed in question the future of these vector machines. By contrast, symmetric multiprocessors dominate the high-end server market and are currently the primary candidate for constructing large scale multiprocessor systems.

Our prefix computation algorithm was 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 Silicon Graphics Power Challenge, and the DEC AlphaServer. We ran our code using a variety of benchmarks which we identified to examine the dependence of our algorithm on memory access patterns. For some problems, our algorithm actually matched or exceeded the performance of the sequential solution using only a single thread. Moreover, in spite of the fact that the processors must compete for access to main memory, our algorithm still yielded scalable performance with up to 16 processors, which was the largest platform available to us.

The organization of our paper is as follows. **Section 2** presents our computational model for analyzing algorithms for this class of problems on symmetric multiprocessors. **Section 3** describes in detail our prefix computation algorithm for this platform. Finally, **Section 4** describes the experimental performance of our prefix computation algorithm.

2 Computational Model

For our purposes, the cost of an algorithm needs to include a measure that reflects the number and type of memory accesses. A number of models have already been proposed which focus on the cost of accessing different levels of memory, including the D-disk model of Vitter and Shriver [9], the Hierarchical Memory with Block Transfer Model of Aggarwal et al. [1], and the Uniform Memory Hierarchy Model of Alpern et al. [2]. However, we believe that these models are unnecessarily complicated to describe the behavior of existing symmetric multiprocessors. Other models have been proposed which focus instead on the contention caused by multiple processors competing to access the same location in main memory, including the (d,x)-BSP model of Blelloch et al. [4] and the Queue-Read Queue-Write (QRQW) PRAM model of Gibbons et al. [5]. The difficulty with these models is that, while they address an issue which has an important impact on performance, the contention they describes depends on specific implementation details such as the memory map which may be entirely beyond the control of the algorithm designer.

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 accounting 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. For the class of problems considered in this paper, we measure the overall complexity of an algorithm by the pair of values $\langle T_M, T_C \rangle$, where T_M is the maximum number of non-contiguous main memory accesses required by any processor and T_C is an upper bound on the local computational complexity of any of the processors. Note that in our model each non-contiguous main memory access may involve an arbitrary sized contiguous block of data, and, hence, accessing a block of z contiguous words will contribute only a unit cost to T_M. Further, since our model is concerned only with the cost of main memory access, once the values are stored in cache they may be accessed in any pattern at no cost. An algorithm is considered optimal in our model if it requires the minimum number of non-contiguous memory accesses consistent with an optimal computational complexity.

3 Prefix Computation Algorithm

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:

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

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 at most $(2n + 1)$ 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 $(n+1)$ non-contiguous memory accesses while running in $O(n)$ computation time, it is optimal up to an additive constant of 1 in our model.

The first fast parallel algorithm for prefix computations was probably the list ranking algorithm of Wyllie [10], which requires at least $n \log n$ non-contiguous accesses. Other parallel algorithms which improved upon this result include those of Vishkin [8] ($5n$ non-contiguous accesses), Anderson and Miller [3] ($4n$ non-contiguous accesses), and Reid-Miller and Blelloch [7] ($2n$ non-contiguous accesses - see [6] for details of this analysis). Clearly, however, none of these approach the memory requirement of our optimal sequential algorithm, which seems necessary to be practically significant on the relatively small number of processors available on the SMP.

3.1 A New Algorithm for Prefix Computations

A high-level description of our prefix computation algorithm proceeds as follows. We 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* $\otimes$ *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* $\otimes$ *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* $\otimes$ *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 [6].

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 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 access and $O(p)$ computation time. In Step (3), each processor randomly chooses $\left(\frac{s}{p}\right)$ 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 $\left(\frac{s}{p} + 1\right)$ non-contiguous memory accesses and $O(\ln n)$ computation time. In Step (4), each processor traverses the sublist associated with each of

its $\left(\frac{s}{p}\right)$ 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 the record in *Sublists* which corresponds to the current sublist and the record in *Sublists* which corresponds to the succeeding sublist can always lie in non-contiguous memory locations, this step requires at most $\left(\alpha(s)\frac{n}{p} + \frac{s}{p} + 1\right)$ non-contiguous memory accesses 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 [6]). 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 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 four 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). Accessing the list array will of course require only a single non-contiguous memory access. Hence, overall, this step will require only $(s+1)$ non-contiguous memory accesses and $O\left(\frac{n}{p}\right)$ computation time. Thus, *with high probability*, the overall complexity of our prefix computation algorithm is given by

$$\begin{aligned} T(n,p) &= \langle T_M(n,p); T_C(n,p)\rangle \\ &= \left\langle \left(\alpha(s)\frac{n}{p} + 2s + 2\frac{s}{p} + 7\right); O\left(\frac{n}{p}\right)\right\rangle \end{aligned} \quad (1)$$

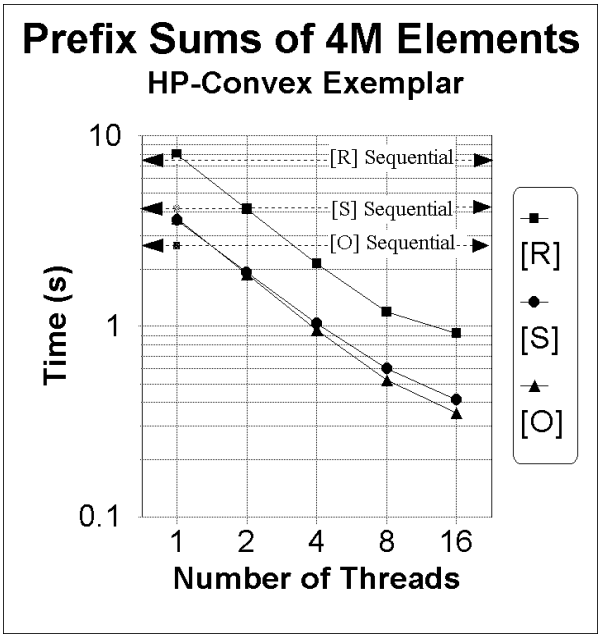
for $\alpha(s) \geq 2.62$, $s \geq (p \ln n + 1)$, and $n > p^2 \ln n$. Recalling that *on average* each processor traverses only about $\frac{n}{p}$ elements in Step (4), we would expect that in practice the complexity of our algorithm could be characterized as:

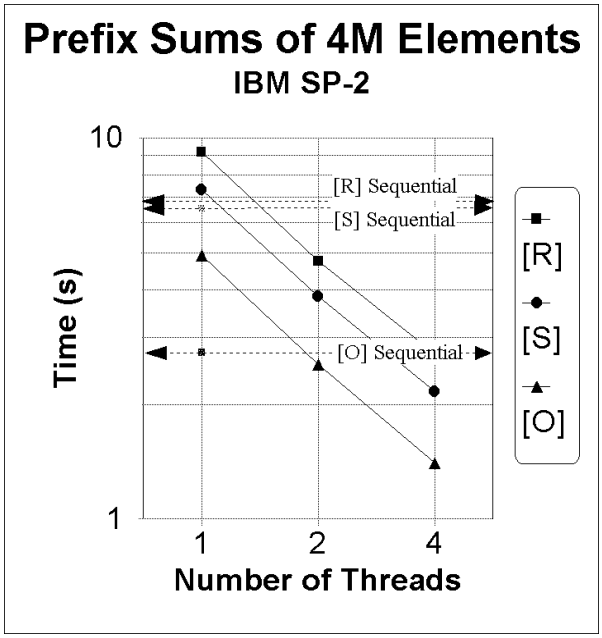
$$\begin{aligned} T(n,p) &= \langle T_M(n,p); T_C(n,p)\rangle \\ &= \left\langle \left(\frac{n}{p} + 2s + 2\frac{s}{p} + 7\right); O\left(\frac{n}{p}\right)\right\rangle . \end{aligned} \quad (2)$$

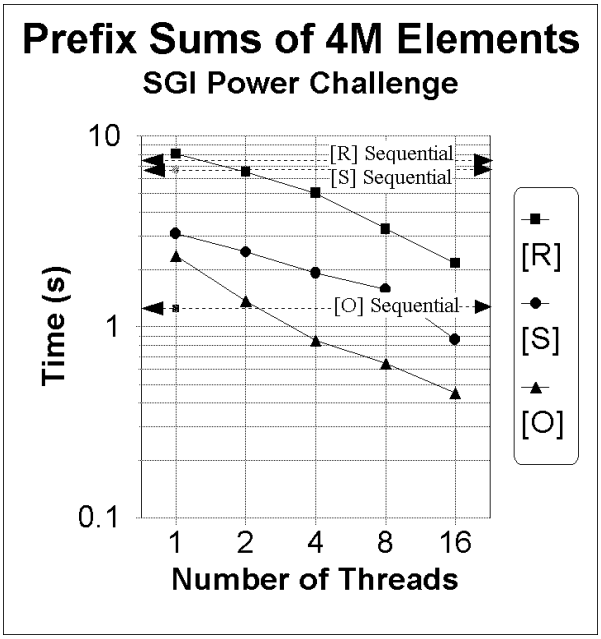
3.2 Performance Evaluation

Both our parallel algorithm and the optimal sequential algorithm were implemented in C using POSIX threads and run on an HP-Convex Exemplar (S-Class), an IBM SP-2 (High Node), an SGI Power Challenge, and a DEC AlphaServer 2100A system. To evaluate these algorithms, we examined the prefix operation 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 placed in the array according to its rank. See [6] for a more detailed description and justification of these benchmarks.

The graphs in **Figures 1** and **2** 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 on the SGI Power Challenge and the DEC AlphaServer, where the successor of an element is always the next location in memory. Given that some degree of overhead is usually unavoidable when parallelizing a solution, the relative success of our parallel algorithm is encouraging. 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 on 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] (Random) 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 fortuitously 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 accesses 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 in this problem on minimizing the number of non-







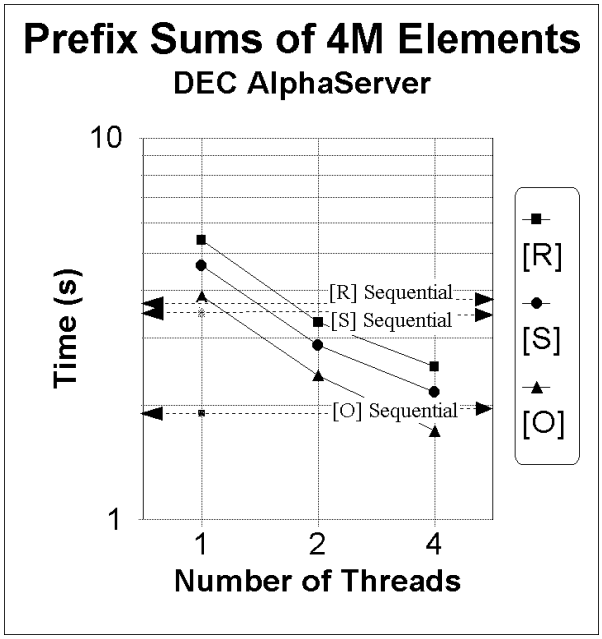


Figure 1. Comparison between the performance of our parallel algorithm and our optimal sequential algorithm using two 16-processor platforms and three different benchmarks.

Figure 2. Comparison between the performance of our parallel algorithm and our optimal sequential algorithm using two 4-processor platforms and three different benchmarks.

p	Bench.	Step(s)				
		(1)-(3)	(4)	(5)	(6)	Total
1	[R]	0.59	6.69	0.01	0.69	7.97
	[S]	0.87	1.86	0.12	0.75	3.60
	[O]	0.66	2.33	0.01	0.69	3.68
2	[R]	0.34	3.40	0.01	0.37	4.12
	[S]	0.40	1.08	0.04	0.38	1.91
	[O]	0.34	1.17	0.01	0.35	1.87
4	[R]	0.18	1.75	0.01	0.21	2.14
	[S]	0.21	0.57	0.05	0.20	1.03
	[O]	0.18	0.59	0.01	0.19	0.97
8	[R]	0.10	0.96	0.01	0.11	1.19
	[S]	0.12	0.31	0.06	0.1	0.60
	[O]	0.10	0.30	0.01	0.11	0.52
16	[R]	0.08	0.74	0.01	0.09	0.92
	[S]	0.08	0.22	0.02	0.12	0.41
	[O]	0.08	0.18	0.01	0.08	0.35

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 p.

contiguous memory accesses.

The graphs in **Figures 3** and **4** examine the scalability of our prefix computation algorithm as a function of the number of threads. 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, 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

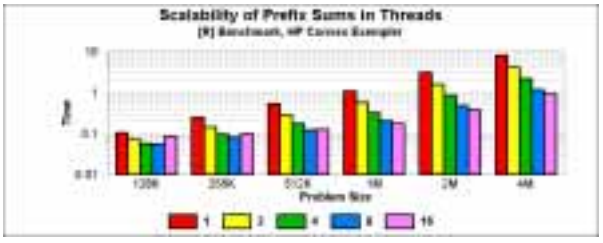


Figure 3. Scalability of our prefix computation algorithm on the HP-Convex Exemplar with respect to the number of threads, for differing problem sizes.

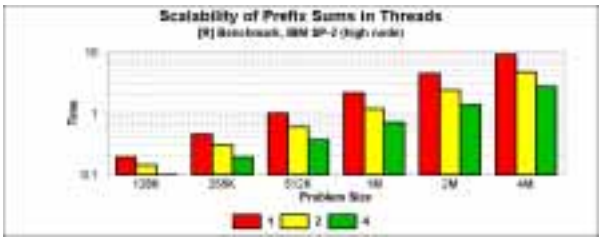


Figure 4. Scalability of our prefix computation algorithm on the IBM SP-2 with respect to the number of threads, for differing problem sizes.

the problem. This introduces a number of issues which are beyond the intended scope of our algorithm.

References

[1] 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.

[2] B. Alpern, L. Carter, E. Feig, and T. Selker. The Uniform Memory Hierarchy Model of Compuatation. *Algorithmica*, 12:72–109, 1994.

[3] 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.

[4] G. Blelloch, P. 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.

[5] P. 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.

[6] D. 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.

[7] M. Reid-Miller. List Ranking and List Scan on the Cray C90. *Journal of the Computer and System Sciences*, 53:344–356, 1996.

[8] U. 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.

[9] J. Vitter and E. Shriver. Algorithms for Parallel Memory I: Two-Level Memories. *Algorithmica*, 12:110–147, 1994.

[10] J. Wyllie. *The Complexity of Parallel Computations*. PhD thesis, Department of Computer Science, Cornell University, Ithica, NY, 1979.