Prefix Computations on Symmetric Multiprocessors* (Preliminary Draft)

David R. Helman Joseph JáJá

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

July 28, 1998

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 three symmetric multiprocessors - the DEC AlphaServer, the Silicon Graphics Power Challenge, and the HP-Convex Exemplar. 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 optimal 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 resulted in scalable performance up to 16 processors, which was the largest platform available to us.

Keywords: Parallel Algorithms, List Ranking, Prefix Computations, Symmetric Multiprocessors, Parallel Performance.

*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

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 nodes except by essentially solving the overall problem. Thus, there is no way to divide the nodes amongst the various processors that will avoid substantial and irregular communication. To make matters worse, there is very little computation with which to mask these communication 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 [12, 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 $n > p^2 \ln n$. 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 three symmetric multiprocessors - the DEC AlphaServer, the Silicon Graphics Power Challenge, and the HP-Convex Exemplar. 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 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. 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 [15] 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 transfered 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. [8]. Known as the Queuing Shared Memory (QSM) model, it describes the execution time in terms of the maximum time required by any processor for computation, the maximum 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 QSM 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. Finally, neither model considers the effects of memory hierarchy.

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:

$$X[i].prefix = \begin{cases} X[i].data & \text{if } X[i] \text{ is the head of the list.} \\ X[i].data \otimes X[pre].prefix & \text{otherwise.} \end{cases}, \qquad (1)$$

where pre is the index of the predecessor of X_i. The last element in the list is distinguished by a negative array 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 memory 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. Indeed, as we will show latter, our experimental results confirm the superiority of this algorithm over the obvious sequential algorithm. Since only a relatively small number of processors are available on a symmetric multiprocessor, any parallel algorithm must be competitive even on a single processor in order to be practically significant.

The first fast parallel algorithm for prefix computations was probably the list ranking algorithm of Wyllie [16] (note that his algorithm was originally developed for the suffix computation problem, and our presentation here makes appropriate modifications for our prefix computation problem). After initially traversing the list to establish the predecessor links, the algorithm consists of iteratively shrinking the list by replacing the predecessor of each element with the predecessor of its predecessor. Before each iteration, the prefix computation of each element is replaced with the result of applying the prefix operation between the current prefix value and that of its predecessor. Since this whole process must be repeated for $O(\log n)$ iterations and we can not assume that the predecessor of a node is in a contiguous memory location, it will require a total of at least $n \log n$ non-contiguous accesses, which is far from optimal. Note that throughout this discussion, when we refer to the *total* number of non-contiguous accesses, we are referring to the sum of the non-contiguous accesses made by all p processors.

Another class of of known algorithms repeatedly contract the list of n elements by identifying and removing an independent set of nodes until $\frac{n}{p}$ elements remain [7, 11, 5]. As each element is removed, the prefix computation of its successor is replaced with the result of applying the prefix operation between the prefix value of the element being removed and the value of its successor. When $\frac{n}{p}$ elements remain, the prefix computation of the reduced list is done using a sequential algorithm, after which the list is restored by reinserting the independent sets in the opposite order in which they were removed. As each element is reinserted, the value of its prefix computation is obtained by applying the prefix operation between its current prefix value and that of its predecessor.

The efficiency of this approach obviously depends on the cost of identifying and removing the independent set. An independent set can be found in a deterministic fashion using the 3-coloring algorithm of Cole and Vishkin [7]. Since this involves substantial constants, we consider the randomized approach of Miller and Reif [11], in which each node in the list is randomly labeled as either active or inactive. If an inactive node is succeeded by an active node, than that inactive node is a member of the independent set. This method for removing the independent set results in an algorithm which requires only $O\left(\frac{n}{p}\right)$ computation time. However, the algorithm requires *on average* at least a total of $5n$ non-contiguous memory accesses - n accesses to establish a doubly linked list from a singly linked list, $3n$ accesses to determine whether the successors of inactive nodes are active and, if so, to properly remove them, and n accesses to restore the list. To see why the list contraction step requires at least $3n$ non-contiguous memory accesses, note that *on average* for any iteration half of the nodes are labeled inactive and half of their successors are active (this analysis is actually more generous than that of its authors in [11]). If iteration i begins with n_i elements, then for each of the approximately $\frac{1}{2}n_i$ inactive elements a check must be made of its successor to see if it is active. Approximately $\frac{1}{4}n_i$ of the nodes will be removed, in which case the successor pointers of their predecessors will have to updated as well. Hence, each iteration requires $\frac{3}{4}n_i$ non-contiguous memory accesses, and the overall result follows. In contrast to the $5n$ total non-contiguous memory accesses required by this algorithm, our optimal sequential algorithm only requires at most $(n+1)$ non-contiguous memory accesses. Moreover, this parallel algorithm seems to require a significant number (at least $4\log n$) of synchronization barriers to run correctly, each of which is a fairly expensive operation.

Anderson and Miller [5] modified the randomized algorithm so that the list does not have to be compacted after each independent set is removed. In this variation, the list is decomposed into $\log n$ queues, where membership in a particular queue implies nothing about the relative proximity of those members in the list. With the exception of the first element in each queue, which is randomly labeled as either active or inactive, every other element is labeled to be active. Then, if the element at the top of a queue is inactive and its predecessor in active, the element at the top of the queue is deleted from the list and the process is repeated with the next element in that queue. Anderson and Miller showed that with *high probability* all $\log n$ queues would be depleted in $O(\log n)$ iterations, resulting in an algorithm which requires only $O\left(\frac{n}{p}\right)$ computation time. However, this algorithm still requires at least a total of $4n$ non-contiguous memory accesses - n accesses to establish a doubly linked list from a singly linked list, $2n$ accesses to contract the list (one access to check the successor of an inactive node and one to update its predecessor), and n accesses to restore the list. Additionally, this algorithm seems to require at least $3\log n$ expensive synchronization barriers to run correctly.

Clearly, one way to improve on the contract and expand approach typified by these algorithms is to accomplish the contraction (and expansion) in a single step. This could be done by choosing relatively few elements to be active and then deleting (and inserting) the entire sublist of inactive elements which precede each active element in a single step. The challenge would be to demonstrate that, in spite of the relatively few active elements, the work of these sublist deletions could be evenly distributed amongst the available processors.

Indeed, Reid-Miller and Blelloch [13, 12] proposed this very modification in a novel, efficient, parallel algorithm which achieved significantly reduced costs. While they originally developed this algorithm for a vector multiprocessor machine, Sibeyn et. al [14] have subsequently applied the idea to distributed memory architectures as well. In this algorithm, the input is randomly divided into $(m+1)$ sublists, where $(n >> m >> p)$. Each of these sublists is traversed to compute the sublist prefix value of the last element in the sublist, and the result is used to form a new linked list of size $(m+1)$ that links these values in the order that their sublists appear in the original list. After that, the prefix sum of the new list is computed using either the sequential algorithm or Wyllie's pointer jumping algorithm. Note that the prefix value of a particular node in the reduced list corresponds to the prefix value of the node at the tail of the corresponding sublist. Finally, each sublist is traversed again using the prefix value computed for the tail of the preceding sublist, and the prefix value of each element is computed in a straightforward manner. Clearly, this algorithm only requires a total of about $2n$ non-contiguous memory accesses (the authors assume the head of the list is already known) - n accesses for the initial traversal and n accesses for the final traversal. Further, on a shared memory machine, the algorithm would require only a constant number of synchronization barriers to run correctly. Finally, Reid-Miller and Blelloch established that *on average* no single processor will require more than about $2\frac{n}{p}$ memory accesses and $O\left(\frac{n}{p}\right)$ computation time for $(n > p^2 \log m)$.

Reid-Miller and Blelloch acknowledge in their paper that a difficulty exists with their algorithm in the way in which the heads of the sublists are chosen. Specifically, each processor chooses $\frac{m}{p}$ positions at random, leaving open the possibility that two processors could randomly choose the same position. Depending on how the algorithm is implemented, this can simply result in some wasteful duplication of effort or it can actually cause problems with correctness. Reid-Miller and Blelloch suggest removing duplicate random numbers by having a competition amongst the processors, in which each processor writes its index to the locations it has randomly chosen. When the process is completed, each processor checks its locations to see if its index has been overwritten. If it has, then it can drop that sublist from those it must traverse.

In this paper, we introduce a new optimal prefix computation algorithm which builds upon the sparse ruling set approach of Reid-Miller and Blelloch. In addition to avoiding the problem of choosing the same location as a sublist head twice, our algorithm requires only about half the total number of non-contiguous memory accesses while achieving a stronger computational complexity bound. Our algorithm only requires a total of about n non-contiguous memory accesses because we are able to entirely eliminate the second set of sublist traversals. Yet, whereas Reid-Miller and Blelloch claim an *expected* computational complexity of $O\left(\frac{n}{p}\right)$ for $n > p^2 \log m$, we claim a computational complexity *with high probability* of $O\left(\frac{n}{p}\right)$ for $n > p^2 \ln n$. Nevertheless, like their algorithm, we require only a constant number (five) of barrier synchronizations. Note that our requirement of about n total non-contiguous memory accesses beats even the requirements of the obvious sequential algorithm and compares closely with the requirements of the optimal sequential algorithm.

A high-level description of our 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 to establish the following Lemma.

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 $n > p^2 \ln n$.

Proof: The number of elements T traversed by a processor can only exceed $\alpha\frac{n}{p}$ if less than $\frac{s}{p}$ of the first $\alpha\frac{n}{p}$ list elements traversed have been selected as splitters. Recall that, in Step (3) of our algorithm, splitters are chosen by first partitioning the input list into a set $\{S_0, S_1, ..., S_{(s-2)}\}$ of s blocks, each of size $\frac{n}{(s-1)}$, after which a splitter is chosen at random from each block. Clearly, if we define T_i as the subset of elements in the traversal T which also belong to the block S_i (i.e $T_i = T \cap S_i$), then the probability that a splitter selected from the block S_i appears in the traversal T is simply $\frac{|T_i|}{|S_i|} = |T_i|\frac{(s-1)}{n}$.

Consider a set of independent but not identically distributed Bernoulli trials $\{X_0, X_1, ..., X_{(s-2)}\}$, where W_i is one with probability $|T_i|\frac{(s-1)}{n}$ and zero otherwise. Clearly, W_i describes the probability that a splitter selected from the block S_i appears in the traversal T, and $\mathrm{Sum}_X = \sum_{(i=0)}^{s-1} X_i$ describes the total number of splitters encountered in the traversal T. Next, consider another set of independent but not identically distributed Bernoulli trials $\{Y_0, Y_1, ..., Y_{(\alpha\frac{n}{p}-1)}\}$, where:

$$Y_i = \begin{cases} X_i & \text{for } (0 \le i \le (s-2)). \\ 0 \text{ with probability one} & \text{for } \left((s-1) \le i \le \left(\alpha\frac{n}{p} - 1\right)\right). \end{cases} \tag{2}$$

Obviously, the sum $\mathrm{Sum}_Y = \sum_{(i=0)}^{s-1} Y_i$ will be equal to Sum_X and will therefore describe as well the total number of splitters encountered in the traversal T.

We can now bound the probability that less than $\frac{s}{p}$ of the first $\alpha\frac{n}{p}$ elements encountered in traversal T were splitters by taking advantage of the well known fact that the variance of a sum of independent Bernoulli trials is maximized when they are independent and identically distributed [10]. More formally, let A be the number of successes in m independent Bernoulli trials, and let p_i be the probability of success in the i^{th} trial. If we define q as being equal to $\frac{\mathrm{E}[A]}{m} = \sum_{(i=0)}^{(m-1)} \frac{p_i}{m}$, then Hoeffding [10] showed that:

$$\Pr(A \le c) \le \sum_{k=0}^{c} \binom{m}{k} q^k (1-q)^{(m-k)} \tag{3}$$

for any integer c such that $(0 \le c \le (\mathrm{E}[A] - 1))$. Applying this result to our problem, we first note that:

$$\mathrm{E}[\mathrm{Sum}_Y] = \mathrm{E}[\mathrm{Sum}_X] = \sum_{(i=0)}^{(s-2)} \mathrm{E}[X_i] = \sum_{(i=0)}^{(s-2)} |T_i|\frac{(s-1)}{n} = \alpha\frac{n}{p}\frac{(s-1)}{n} = \alpha\frac{(s-1)}{p}. \tag{4}$$

Note that if we assume that $\alpha \le \frac{s}{s-1}$, then it follows that:

$$\mathrm{E}[\mathrm{Sum}_Y] - 1 = \alpha\frac{s-1}{p} - 1 \ge \frac{s}{p} - 1. \tag{5}$$

We can then use Hoeffding's theorem to establish that

$$\Pr\left(\mathrm{Sum}_Y \le \left(\frac{s}{p} - 1\right)\right) = \Pr\left(\mathrm{Sum}_X \le \left(\frac{s}{p} - 1\right)\right) \le \sum_{k=0}^{\frac{s}{p}-1} \binom{r}{k} q^k (1-q)^{r-k}, \tag{6}$$

where $r = \alpha\frac{n}{p}$, $q = \frac{\mathrm{E}[\mathrm{Sum}_X]}{\alpha\frac{n}{p}} = \frac{(s-1)}{n}$, and $\alpha \geq \frac{s}{(s-1)}$. Using the following "Chernoff" type bound [9] for estimating the head of a binomial distribution

$$\sum_{k=0}^{\epsilon r q} \begin{pmatrix} r \\ k \end{pmatrix} q^k (1-q)^{r-k} \leq e^{-(1-\epsilon)^2 \frac{rq}{2}}, \tag{7}$$

and noting that $\left(\frac{s}{p} - 1\right) \leq \frac{(s-1)}{p}$ for $p \geq 1$, it follows that the probability that a particular processor will traverse at least $\alpha\frac{n}{p}$ elements can be bounded by

$$e^{-\left(1-\frac{1}{\alpha}\right)^2 \frac{\alpha(s-1)}{2p}}. \tag{8}$$

Hence, the probability that any processor will traverse at least $\alpha\frac{n}{p}$ elements without encountering at least $\frac{s}{p}$ splitters can be bounded by

$$\sum_{i=0}^{p-1} e^{-\left(1-\frac{1}{\alpha}\right)^2 \frac{\alpha(s-1)}{2p}}. \tag{9}$$

If we then assume that $\frac{(s-1)}{p} \geq \ln n$ and $n > p^2 \ln n$, it is easy to show that the above sum can be bounded by $n^{-\lambda}$ for some $\lambda > 0$ and $\alpha(s) \geq 2.62$. It is trivial to verify that for these values of n, s, and $\alpha(s)$, it will always be true that $\alpha(s) \geq \frac{s}{(s-1)}$ and Lemma 1 follows. $\square$

With the results of Lemma 1, we can now establish the following theorem:

Theorem 1: For our prefix computation algorithm, which uses a total of about n non-contiguous memory accesses, no processor will require more than $\left(\alpha(s)\frac{n}{p} + 2s + 2\frac{s}{p} + 7\right)$ non-contiguous memory accesses and $O\left(\frac{n}{p}\right)$ computation time *with high probability* for $\alpha(s) \geq 2.62$, $s \geq (p \ln n + 1)$, and $n > p^2 \ln n$. Moreover, *on average* no single processor will require more than $\left(\frac{n}{p} + 2s + 2\frac{s}{p} + 7\right)$ non-contiguous memory accesses and $O\left(\frac{n}{p}\right)$ computation time.

Proof: The analysis of our algorithm on a symmetric multiprocessor 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 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 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). 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

$$\begin{aligned} T(n,p) &= \langle T_M(n,p); T_C(n,p)\rangle & (10)\\ &= \langle\left(\alpha(s)\frac{n}{p}+2s+2\frac{s}{p}+7\right); O\left(\frac{n}{p}\right)\rangle & (11)\end{aligned}$$

for $\alpha(s)\geq 2.62$, $s\geq(p\ln n+1)$, and $n>p^2\ln n$. Moreover, *on average*, the overall complexity of our algorithm is given by

$$\begin{aligned} T(n,p) &= \langle T_M(n,p); T_C(n,p)\rangle & (12)\\ &= \langle\left(\frac{n}{p}+2s+2\frac{s}{p}+7\right); O\left(\frac{n}{p}\right)\rangle, & (13)\end{aligned}$$

and Theorem 1 follows. □

Note that our experimental discussion will verify that in practice the burden of these accesses is very evenly distributed across the available processors.

4 Performance Evaluation

Both the sequential algorithm and our parallel algorithm were implemented in C using POSIX threads and run on a DEC AlphaServer 21000A system, an SGI Power Challenge, and an HP-Convex Exemplar. The DEC AlphaSever consists of four Alpha 21064A processors, each running at 275 MHz. Each Alpha 21064A processor has a 16KB primary data cache and a 4MB secondary data cache. The SGI Power Challenge consists of sixteen MIPS R10000 processors, each running at 195MHz. Each MIPS R10000 processor has a 32KB primary data cache and a 2MB secondary unified (data and instructions) cache. The HP-Convex Exemplar is an S-Class machine consisting of sixteen PA-8000 processors each running at 180 MHZ. Each PA-8000 processor has a single level 1MB data cache.

The evaluation of our algorithm's performance is organized as follows. **Subsection 4.1** describes the benchmarks used and their justification, while **Subsection 4.2** examines the performance of our algorithm as a function of input distribution. **Subsection 4.3** compares the performance of our algorithm with the optimal sequential algorithm, and **Subsection 4.4** examines the scalability of our algorithm in both the number of threads and the problem size. Finally, **Subsection 4.5** examines the performance of our algorithm as a function of the number of splitters.

4.1 Benchmarks

For our experimental evaluation, we examine the prefix operation of floating point addition on three different benchmarks. The input for prefix sums is an array of n records. Each record corresponds to a node in the list and consists of two fields. The first field is a four byte integer which specifies the array index of its successor and the other is an eight byte double precision which is the input for the prefix operation. The output of the prefix computation is simply the input array with the properly computed prefix sums in the place of the prefix input values.

The successor fields for our three benchmarks are created as follows:

1. **Random [R]** - in which each successor is randomly chosen , is initialize by placing the elements in the first two positions of the list in a doubly linked circular list (note that predecessor links are removed when the list is complete). Subsequent elements are added as follows. Assume the doubly linked circular list contains the first k elements in the list array, and we wish to add the $(k+1)^{th}$ element. We call the C library random number generator random(), which was originally seeded the value 1001. Then, if the current predecessor of the $t = ((\text{random}() \bmod (k+1))^{th}$ element in the array has the index s, the new successor of the s^{th} element will be the $(k+1)^{\text{th}}$ element and the successor of the $(k+1)^{\text{th}}$ element will be the t^{th} element. When this circular list is complete, it is broken by setting the successor field of the $((\text{random}() \bmod n)^{th}$ element to -1.

2. **Stride [S]** - in which each successor is (wherever possible) some stride S away, is generated as follows. Assume the head of the list is at index $\frac{n}{2}$ in the array, and a default stride of $S = 1001$. Then, the successor of this element is the location $((\frac{n}{2} + 1001) \mod n)$. The exercise is repeated $(n-2)$ more times until the list is complete. In the event that a location has already been

selected for a previous element, a vacant position in the array is found by linear probing.

3. **Ordered [O]** - in which each element is placed in the array according to its rank, is created by placing the first element in the list at the first position in the array, the second element in the list at the second portion in the array, and so forth.

For each benchmark, the prefix input value for an element is determined by first calling the function random(), which returns a value between 0 and $(2^{31} - 1)$, and then subtracting 2^{30} from the value returned to yield a random value in the interval $(-2^{30}, 2^{30} - 1)$.

These benchmarks were chosen to compare the impact of various memory access patterns. Accessing the successor of a list element in the **Random [R]** benchmark will usually produce a cache miss for a list of size larger than the cache. By contrast, even though the successor of a list element in the **Stride [S]** benchmark will usually also be in a non-contiguous memory location, the constant stride makes pre-fetching a possibility and hence a cache miss may or may not result. At the other extreme is the **Ordered [O]** benchmark, in which the successor of a list element is almost always going to be in cache, either as part of the same cache line or because of elementary pre-fetching.

4.2 Performance as a Function of Input Distribution

The graphs in **Figure 1** compare the performance of our prefix computation algorithm on each of our three platforms as a function of the input distribution. Results are shown for an input size of 4M elements for the full range of processors available on each of the SMPs. Almost without exception, the [O] benchmark always outperformed the [S] benchmark, which in turn always outperformed the [R] benchmark. To see why, consider **Table I**, which shows the step by step breakdown of the execution time in Figure 1 for the HP-Convex Exemplar. Note first that for any particular number of threads, Steps (1)-(3), Step (5), and Step (6) require approximately the same amount of time, irrespective of which benchmark we consider. This agrees well with our theoretical expectations, since these steps essentially involve only contiguous memory accesses, regardless of the benchmark. Instead, the dependence of the overall execution time on the input distribution reflects the relative differences in the time required to complete Step (4). Recall that in this step, each of the sublists is traversed by following the successor pointers, and so the nature of the memory access pattern is entirely determined by the input. In the Random Benchmark [R], the memory 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. Hence, as we would expect, the [O] benchmark always outperforms the [R] benchmark. Perhaps more surprising is the fact that [S] benchmark always outperforms the [R] benchmark, even though the benchmark is designed so that where possible the successor is always a constant stride away. Since we chose the stride to be 1001, we would expect that every step in the traversal would involve accessing a non-contiguous location in memory. However, what distinguishes the [S] benchmark from the [R] benchmark is the the constant stride, which can take advantage of cache pre-fetching.

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

Table I: 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.

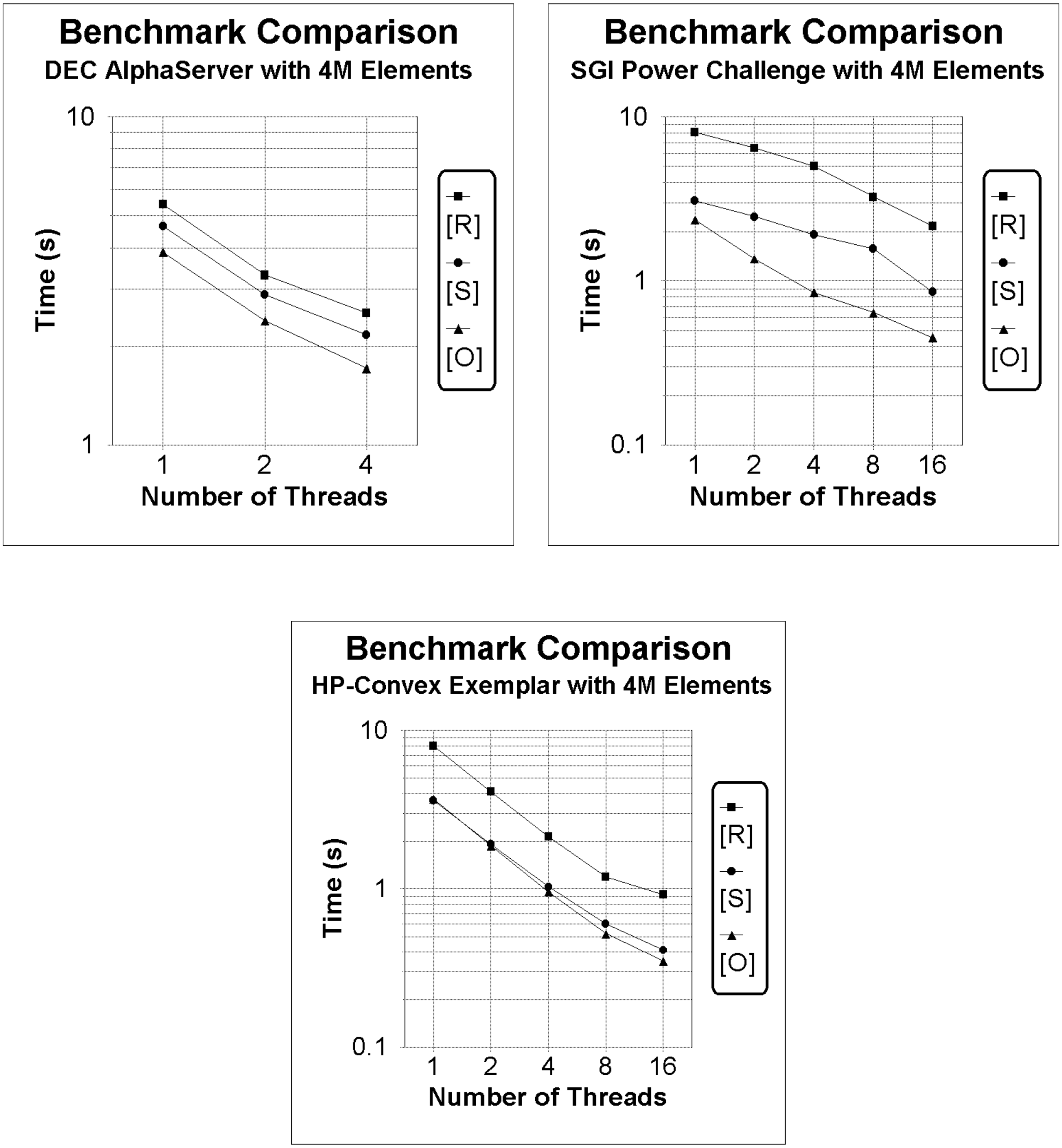


Figure 1: Performance (in seconds) of our algorithm on each of the three platforms as a function of the input distribution, using an input of 4M elements.

4.3 Comparison with the Optimal Sequential Algorithm

To start with, we verify the efficiency of our optimal sequential algorithm by comparing its performance with that of the obvious sequential algorithm. The graphs in **Figure 2** compare the execution time of the two sequential algorithms, showing results for each of our three platforms using different size inputs generated with the [R] benchmark. In every instance, the performance of the optimal sequential algorithm surpassed the performance of the obvious sequential algorithm. This is the expectation of our model, as the optimal algorithm calls for only half the non-contiguous memory accesses as the obvious one. A similar relationship was observed for the other benchmarks.

The graphs in **Figures 3** through **5** compare the performance of our optimal parallel prefix computation algorithm with that of our optimal sequential algorithm. Results are shown for each of our three platforms on inputs of 4M elements generated using both the [R] and [S] benchmarks. For 4M elements, our parallel algorithm always outperforms the optimal sequential algorithm with only one or two threads. This is also true for [O] benchmark (not shown), with the single exception of the DEC AlphaServer. At the other extreme, for an input of 128K elements (not shown), the sequential algorithm outperforms our parallel algorithm on all but the HP-Convex Exemplar, irrespective of the number of threads. However, it is important to qualify this last result by pointing out that a list of 128K twelve byte records can fit reasonably well in the 2MB secondary cache of the Power Challenge and the 4MB secondary cache of the AlphaServer. Hence, once the list is brought in from main memory in a contiguous fashion for initialization, most subsequent memory requests would not be expected to miss to main memory. Thus, our algorithm, which accept a measure of additional overhead when compared to the sequential algorithm to reduce costly main memory accesses, is not competitive. By contrast, 128K twelve byte records do not entirely fit into the 1MB single-level cache of the HP-Convex Exemplar. Hence, cache misses are still an issue, and our algorithm remains competitive. Clearly, the success of our algorithm on problems which exceed the capacity of the cache strongly supports the attention our model attaches to the number of non-contiguous memory accesses.

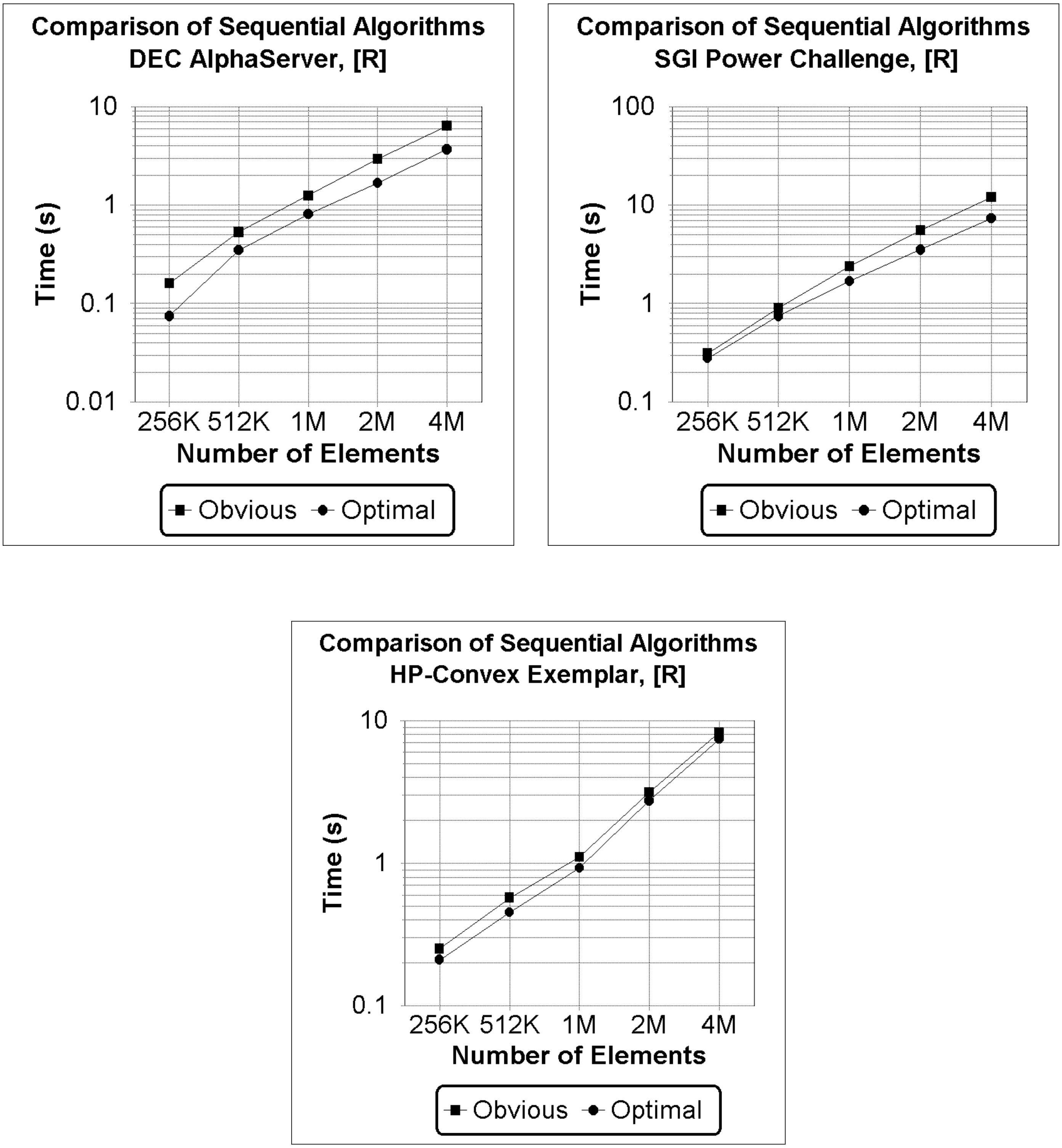


Figure 2: Comparison between the performance of the obvious sequential algorithm and our optimal sequential algorithm.

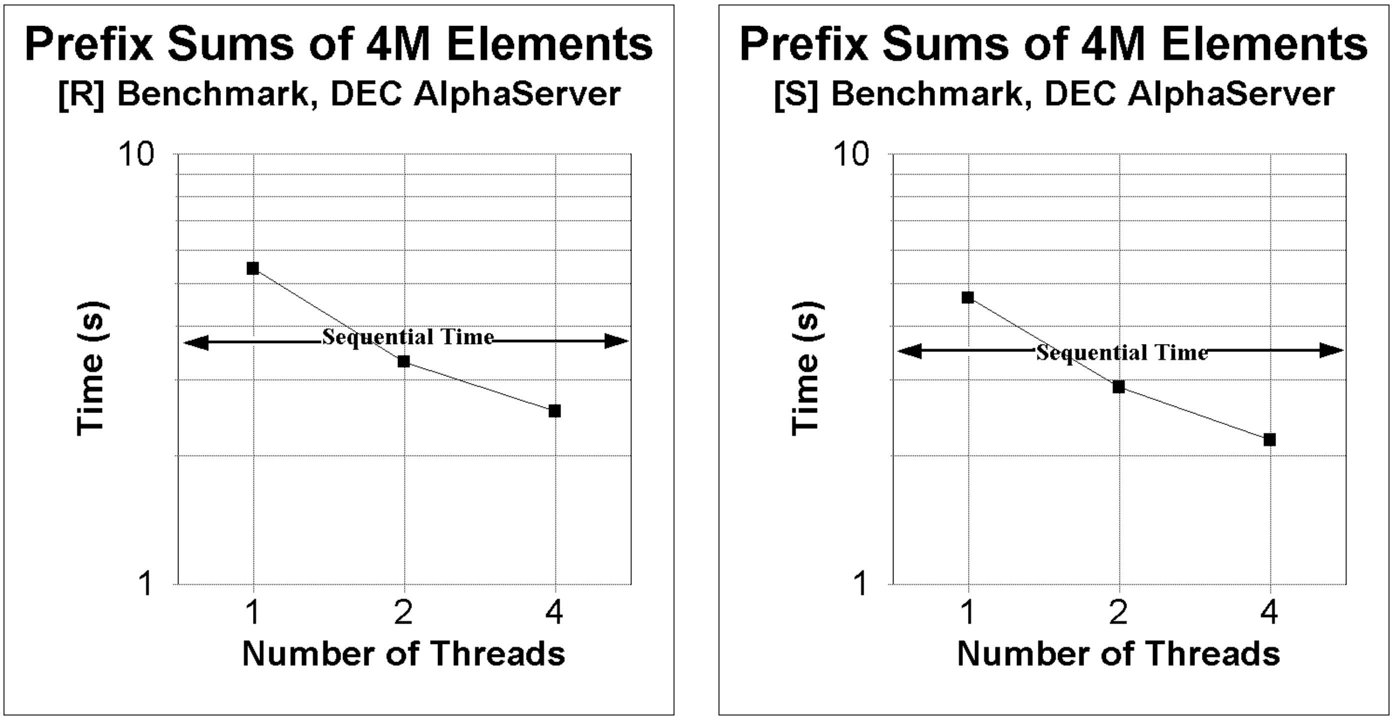


Figure 3: Comparison between the performance of our parallel algorithm and our optimal sequential algorithm on the DEC AlphaServer.

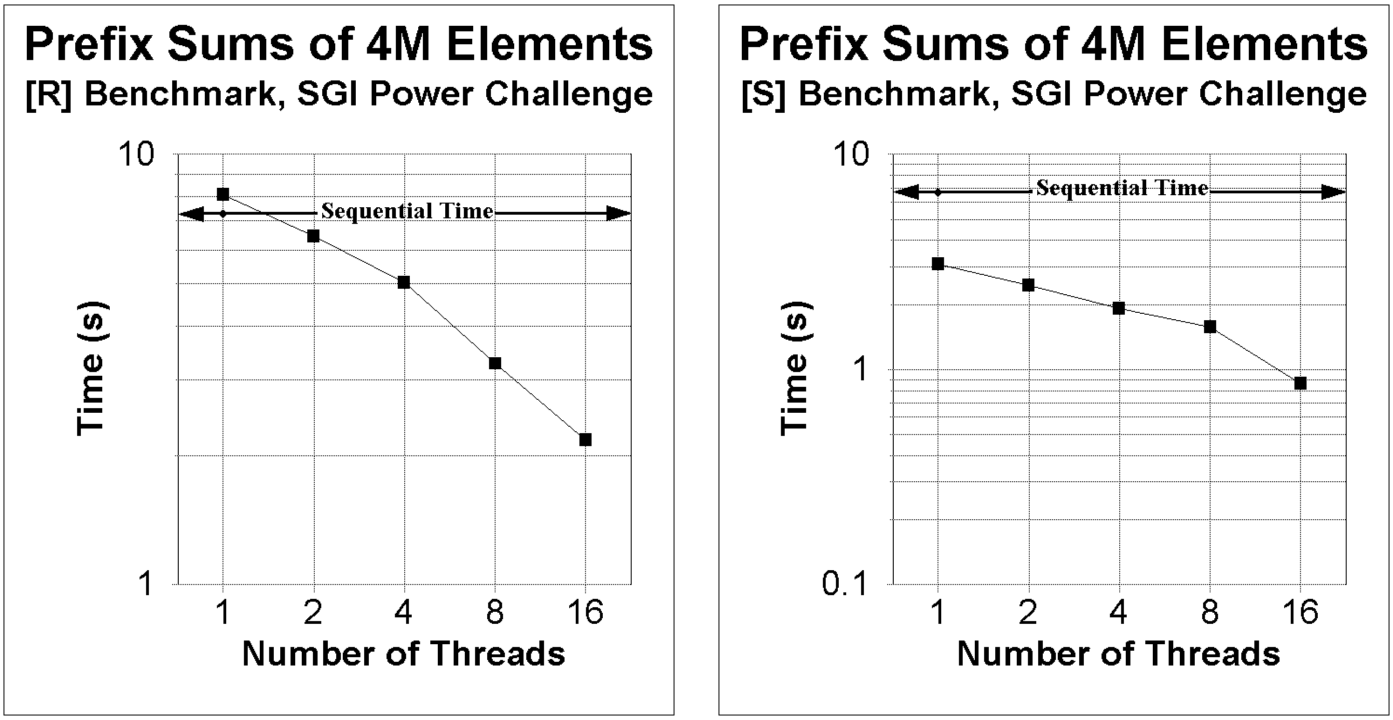
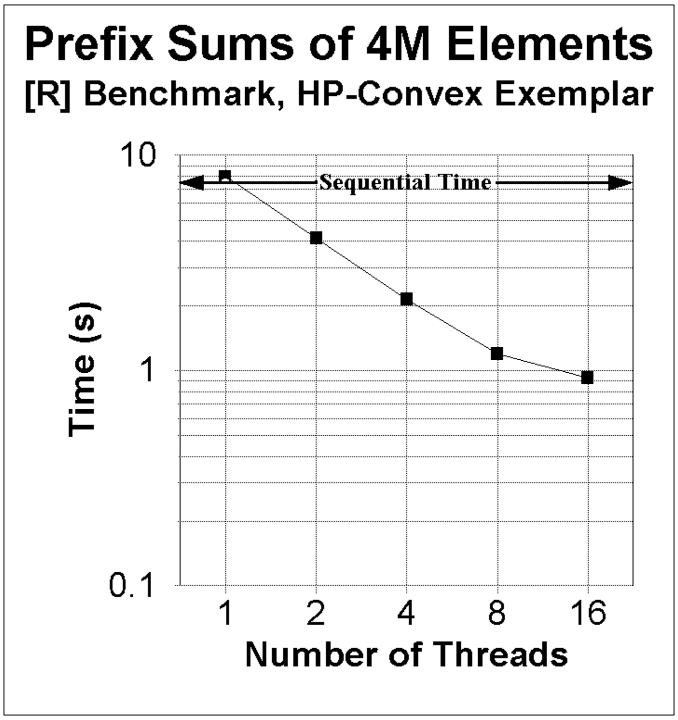


Figure 4: Comparison between the performance of our parallel algorithm and our optimal sequential algorithm on the SGI Power Challenge.



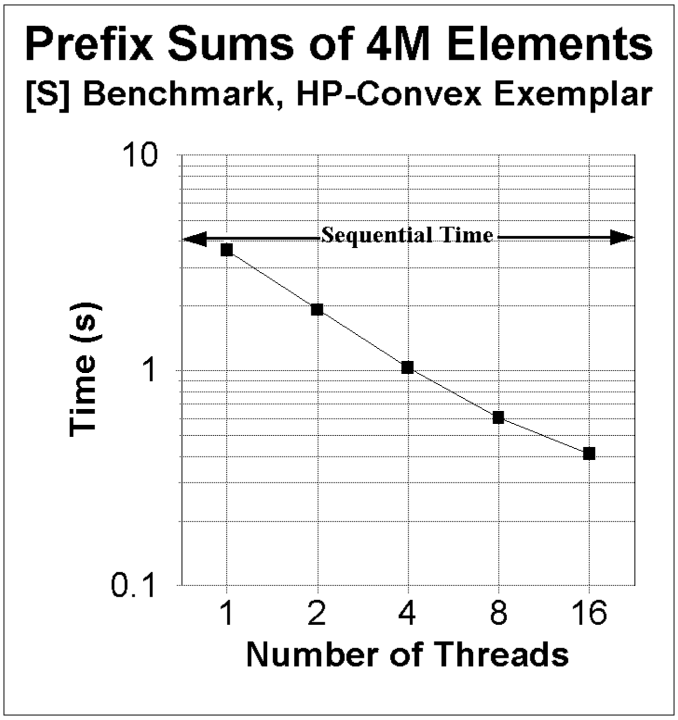


Figure 5: Comparison between the performance of our parallel algorithm and our optimal sequential algorithm on the HP-Convex Exemplar.

4.4 Scalability in Number of Threads and Problem Size

The graphs in **Figure 6** 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 each of our three platforms 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. The step-by-step breakdown in **Table II** verifies that this decrease occurs constantly at every step. 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 the problem. This introduces a number of issues which are beyond the intended scope of our algorithm.

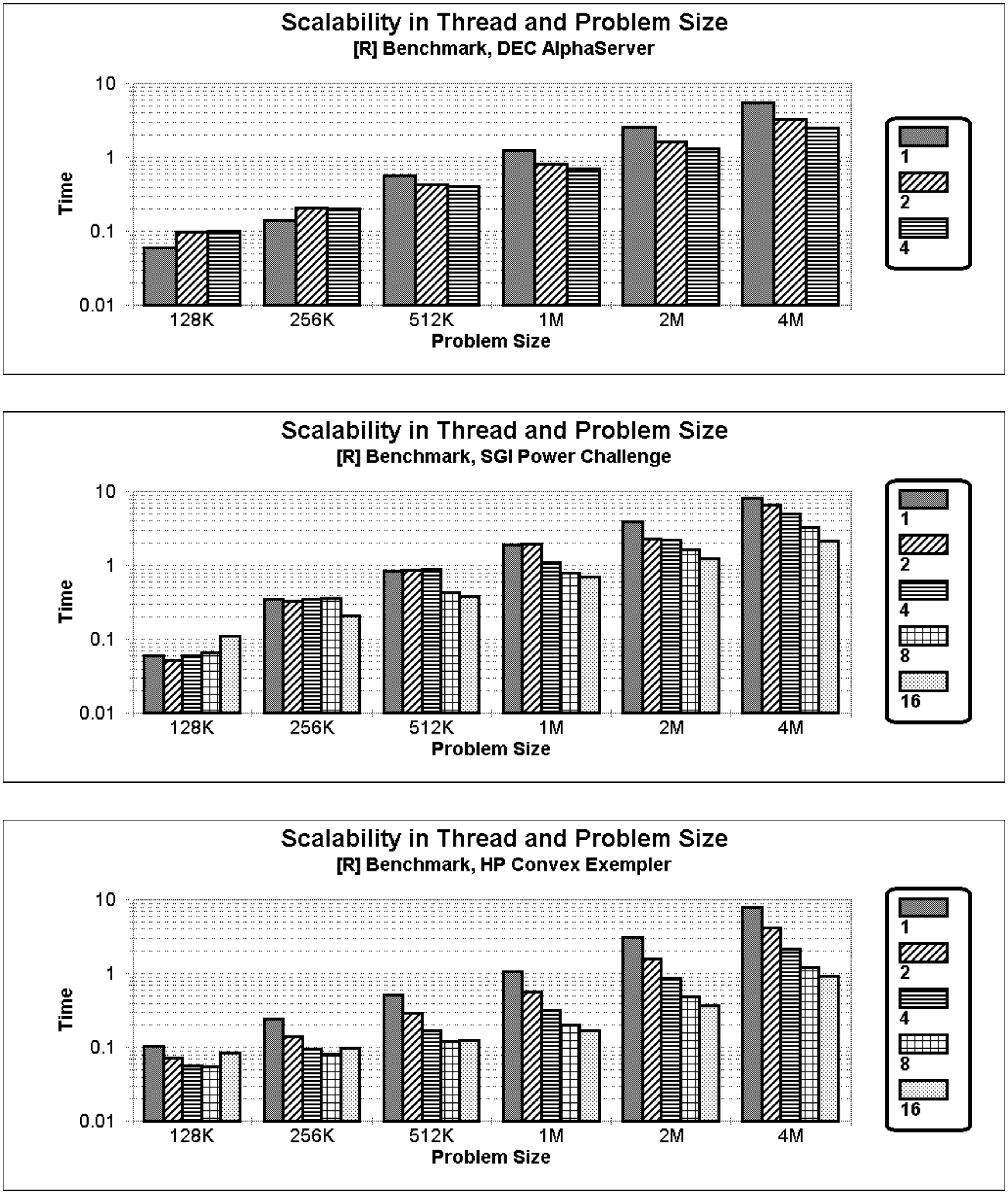


Figure 6: Scalability of our algorithm with respect to the number of threads, for differing benchmarks and problem sizes.

Step:	Platform & Number of Threads												
	DEC			SGI					HP-Convex				
	1	2	4	1	2	4	8	16	1	2	4	8	16
(1)-(3):	1.20	0.78	0.66	0.76	0.65	0.81	0.39	0.48	0.59	0.34	0.18	0.10	0.08
(4):	2.94	1.72	1.25	6.62	5.33	3.94	2.67	1.50	6.69	3.40	1.75	0.96	0.74
(5):	0.01	0.01	0.01	0.01	0.01	0.01	0.05	0.01	0.01	0.01	0.01	0.01	0.01
(6):	1.27	0.80	0.61	0.67	0.49	0.27	0.15	0.18	0.69	0.37	0.21	0.11	0.09
Total:	5.41	3.29	2.52	8.05	6.48	5.02	3.26	2.16	7.97	4.12	2.14	1.19	0.92

Table II: Time (in seconds) required for each step of computing the prefix sums of 4M list elements generated with the [R] benchmark using a variety of threads.

4.5 Performance as a Function of the Number of Splitters

The first graph in **Figure 7** examines the performance of our prefix computation algorithm on the [R] benchmark as a function of the number of splitters s. Results are shown for a list of 4M elements on the SGI Power Challenge using differing numbers of threads - results obtained on other platforms were similar. The graph clearly indicates that, for each number of threads, there is a range of values of s that result in the best performance. To see why, consider the step-by-step breakdown of the execution time for four threads in **Table III**. Notice that, for small values of s, increasing the value of s decreases the execution time primarily by decreasing the time required for Step 4. This is due to to an improvement in the load distribution amongst the threads, which agrees with out theoretical expectation that as the value of s increases, the value of $\alpha(s)$ drops. More interesting is the fact that, for $s > 32K$, the performance begins to deteriorate sharply. Table III shows that this corresponds to an increase in the time required for all but Step 4. The gradual increase in the time required for Steps (1) through (3) is reasonable and reflects the increased cost of selecting more splitters in Step (3). More surprising is the time required in the time required for Step 5, which suddenly began to increase after remaining relatively constant for $s \leq 32768$. However, this behavior makes sense if we recall that this step involves traversing the linked list in the *Sublists* array, in which each element consists of a 20 byte record in our implementation. As long as the size of this list remains comfortably less than the capacity of the 2MB combined (data/instruction) cache of the RS1000 processor, then the non-contiguous memory accesses of the *Sublists* list traversal will not result in a cache miss. However, as s increases beyond 32768, this becomes less and less likely, and the hence the cost of Step (5) begins to rapidly escalate. Similarly, Step 6 involves looking up prefix computation values in the *Sublists* array $\frac{n}{p}$ times, and so might be expected to be independent of the value s. However, as noted in our complexity discussion, each successive lookup might be to a non-contiguous memory location. As long as the *Sublists* array fits easily into the 2MB cache, than each successive lookup will not result in a cache miss. However, as s increases beyond 32768, this becomes less and less likely, and therefore the cost of this step begins to increase.

More surprising was the relationship that existed between the execution time and the number of splitters for the [S] benchmark. The second graph in **Figure 7** examines the performance of our algorithm on the the SGI Power Challenge using a list of 4M elements generated using a stride of 1001. It shows that as the number of splitters is increased from 2048 to 262144, the execution time is reduced by more than a half, irrespective of the number of threads. The third graph in **Figure 7** shows similar results on the same platform using the **[S]** benchmark generated using a stride of 256, suggesting that this phenomenon is not an artifact of a single choice of benchmark stride. The fourth graph in **Figure 7** examines the performance of our algorithm on the HP-Convex Exemplar using the the [S] benchmark generated using a stride of 1001. The results show that the efficiency of increasing the number of splitters, though less pronounced on the Exemplar, is not limited to a single platform. **Tables IV** and **V** display a step-by-step breakdown of the execution times using a single thread on the Power Challenge and the Exemplar, respectively. The data in these tables show the reduction in the execution time is entirely due to a reduction in the time required for the sublist traversals of Step (4). Increasing the number of splitters on the Power Challenge reduced the cost of this step by 81%, whereas increasing the number of splitters on the Exemplar reduced the cost of this step by a more modest 47%. Finally, although not shown, increasing the number of splitters on the AlphaServer reduced the cost of this step by 30%.

	Number of Splitters										
Step:	**512**	**1K**	**2K**	**4K**	**8K**	**16K**	**32K**	**64K**	**128K**	**256K**	**512K**
(1)-(3):	0.78	0.81	0.78	0.81	0.81	0.81	0.81	0.93	0.99	1.17	1.44
(4):	4.12	3.99	4.03	4.01	3.94	3.95	3.97	3.88	3.87	3.88	3.88
(5):	0.01	0.01	0.01	0.01	0.01	0.01	0.01	0.02	0.04	0.23	0.59
(6):	0.26	0.26	0.27	0.27	0.27	0.27	0.28	0.30	0.52	0.87	1.05
Total:	5.16	5.06	5.07	5.09	5.02	5.03	5.07	5.12	5.42	6.15	6.96

Table III: Time (in seconds) required for each step of computing the prefix sums of 4M list elements on an SGI Power Challenge as a function of the number splitters. Results are shown for four threads using an input generated with the [R] benchmark.

Finally, **Table VI** shows the experimentally derived expected value (E) of the coefficient $\alpha(s)$ used to describe the complexity of our algorithm in Section 3 as a function of the number of splitter s. The values shown were obtained by analyzing data collected while computing the prefix sums of a 1M list generated using the [R] benchmark. The data was obtained from a total of 200 trials, where, for every trial, a different seed was used for the random number generator, both to generate the benchmark and to choose the splitters as part of Step (1). In each trial, the value recorded was the largest occurrence of α on any of the four processors. Note that the experimentally derived expected values for α are far less our theoretically derived value of 2.62 for $s \geq (p \ln n + 1) = 56$, supporting our contention

Step:	Number of Splitters										
	512	**1K**	**2K**	**4K**	**8K**	**16K**	**32K**	**64K**	**128K**	**256K**	**512K**
(1)-(3):	0.72	0.72	0.71	0.72	0.72	0.73	0.76	0.79	0.88	1.04	1.32
(4):	6.57	6.56	6.42	6.28	6.02	5.41	4.35	2.69	1.50	1.23	1.26
(5):	0.01	0.01	0.01	0.01	0.01	0.01	0.01	0.02	0.04	0.23	0.59
(6):	0.69	0.67	0.66	0.66	0.66	0.66	0.66	0.66	0.67	0.69	0.71
Total:	7.98	7.94	7.79	7.65	7.40	6.81	5.78	4.16	3.09	3.19	3.89

Table IV: Time (in seconds) required for each step of computing the prefix sums of 4M list elements generated on an SGI Power Challenge as a function of the number splitters. Results are shown for a single thread using an input generated with the [S] benchmark and a stride of 1001.

Step:	Number of Splitters										
	512	**1K**	**2K**	**4K**	**8K**	**16K**	**32K**	**64K**	**128K**	**256K**	**512K**
(1)-(3):	0.70	0.67	0.67	0.67	0.67	0.68	0.70	0.72	0.78	0.87	1.04
(4):	3.37	3.36	3.32	3.25	3.11	2.96	2.79	2.51	2.12	1.86	1.81
(5):	0.01	0.01	0.01	0.01	0.01	0.01	0.01	0.01	0.06	0.12	0.32
(6):	0.70	0.71	0.71	0.71	0.72	0.72	0.73	0.74	0.75	0.75	0.81
Total:	4.74	4.74	4.71	4.64	4.50	4.37	4.22	4.00	3.70	3.60	3.98

Table V: Time (in seconds) required for each step of computing the prefix sums of 4M list elements generated on an HP-Convex Exemplar as a function of the number splitters. Results are shown for a single thread using an input generated with the [S] benchmark and a stride of 1001.

that in practice no processor in Step (4) is required to do much more than $\frac{n}{p}$ non-contiguous memory accesses.

Value	Total Number of Splitters							
	16	**32**	**64**	**128**	**256**	**512**	**1024**	**2048**
E(α):	1.44	1.37	1.24	1.18	1.12	1.09	1.06	1.04

Table VI: Statistical evaluation of the experimentally observed value of the algorithm coefficient α.

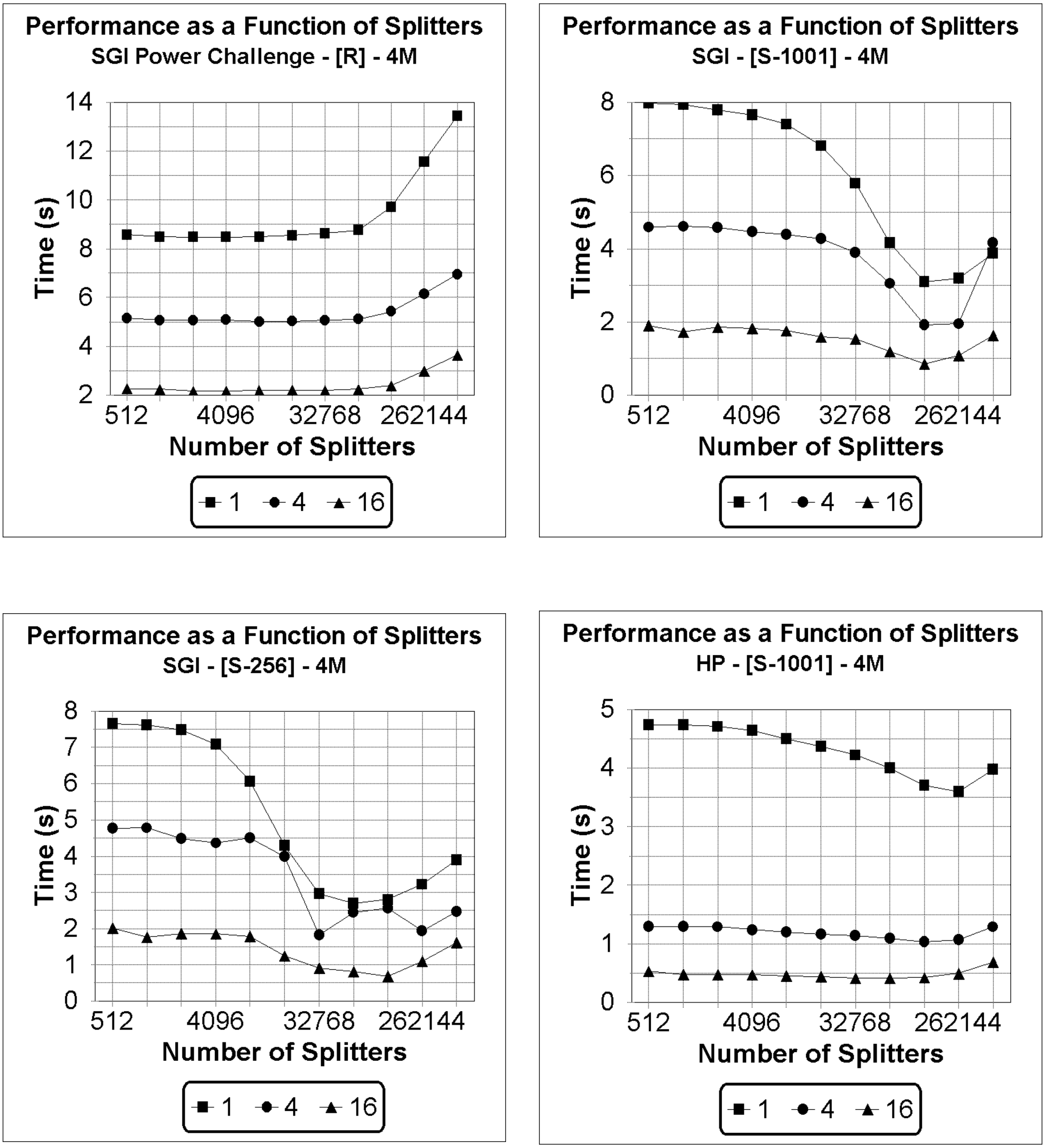


Figure 7: Performance as a function of the total number of splitters, for differing numbers of threads. Results are shown for various platforms using inputs generated with the various benchmarks.

5 Conclusion

We have introduced 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 algorithm's complexity *with high probability* instead of merely *on average*. Finally, whereas the previous algorithm was intended for implementation on vector multiprocessors, our algorithm is intended for efficient implementation on symmetric multiprocessors.

Our algorithm was implemented and tested on several platforms using widely different benchmarks. For problems which exceeded the size of the cache, our algorithm scaled as predicted both in number of threads and in problem size. For some problems, our algorithm actually matched or exceeded the optimal sequential solution using only a single thread, which again was the expectation of our computational model based on the similar number of non-contiguous memory accesses. Together, these experimental results clearly confirm the value of our algorithm for computing prefix computations on symmetric multiprocessors. Equally important, the results verify the utility of our computational model, and in particular its emphasis on the precise number of non-contiguous memory accesses, as a guide for constructing efficient algorithms for symmetric multiprocessors.

Please see `http://www.umiacs.umd.edu/research/EXPAR` for related work by the authors.

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] R. Cole and U. Vishkin. Deterministic Coin Tossing with Applications to Optimal Parallel List Ranking. *Information and Control*, 70:32–53, January 1986.

[8] P.B. Gibbons, Y. Matias, and V. Ramachandran. Can a Shared-Memory Model Serve as a Bridging-Model for Parallel Compatation? In *Proceedings of the 9th ACM Symposium on Parallel Algorithms and Architectures*, pages 72–83, June 1997.

[9] T. Hagerup and C. Rüb. A Guided Tour of Chernoff Bounds. *Information Processing Letters*, 33:305–308, 1990.

[10] W. Heffding. On the Distribution of the Number of Successes in Independent Trials. *Annals of Mathematical Statistics*, 27:713–721, 1956.

[11] G. L. Miller and J. H. Reif. Parallel Tree Contraction and its Application. In *Proceedings Twenty-Sixth Annual IEEE Symposium on Foundations of Computer Science*, pages 478–489, Portland. OR, October 1985.

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

[13] M. Reid-Miller and G. Blelloch. List Ranking and List Scan on the Cray C90. Technical Report CMU-CS-94-101, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, March 1994.

[14] J.F. Sibeyn, F. Guillaume, and T. Seidel. Practical Parallel List Ranking. In *Proceedings of the 4th Symposium on Solving Irregularly Structured Problems in Parallel*, pages 25–36. Springer-Verlag, 1997.

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

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