# An Experimental Approach to the Performance Evaluation of Parallel Algorithms[*]

*Anand Sivasubramaniam*
*Gautam Shah*
*Umakishore Ramachandran*
*H. Venkateswaran*

Technical Report GIT-CC-92/10
February 1992
College of Computing
Georgia Institute of Technology
Atlanta, Georgia 30332-0280

**Abstract**

The results of experimenting with three parallel algorithms on the Sequent Symmetry architecture and the BBN Butterfly architecture are reported. The main objective of this study is to understand the impediments to the efficient implementation of parallel algorithms, developed for theoretical models of parallel computation, on realistic parallel architectures. Scheduling, task granularity, and synchronization are the issues that are explored in implementing these algorithms on the two architectures. In the case of theBBN Butterfly, which is a distributed shared memory architecture,data distribution in the distributed memories is also studied. The key findings are that synchronization is not a significant cost for the algorithms we studied on the two architectures; static scheduling outperforms dynamic scheduling for the algorithms studied; the bus is not a bottleneck at higher task granularities for the configuration of the Sequent machine that we experimented with; and a fairly simple minded data distribution may be as good as any other on the BBN Butterfly. These studies also suggest a top-down approach to understanding the scalability of shared memory multiprocessors.

[*]This paper represents a substantially reworked version of an earlier paper that appeared in the International Symposium on Shared Memory Multiprocessing, Tokyo, Japan [SSL$^+$91].

## 1 Introduction

Parallel Computation provides some of the most challenging problems in computer science. The term parallel computation covers a broad spectrum of research ranging from purely theoretical models for complexity analysis of parallel algorithms, to detailed system performance issues of large problems that lend themselves to parallel implementations. Both ends of the spectrum have one thing in common, namely, to understand the performance potential of parallel computation. A computation is expressed as a task graph and the objective is to determine the speedup that is realizable for the computation. While the theoretical models are concerned with the asymptotic limits of computing in parallel, the system-oriented studies are concerned with determining the best heuristic mapping for a computation on a target architecture that would result in the best (average case) performance. Each study has its merits and de-merits. The asymptotic limits give a ceiling for maximum achievable performance for a given algorithm based on an abstract model of parallel architectures. The average case results are useful for determining what is achievable in reality.

Understanding the performance of parallel computation requires a knowledge of the capabilities of the underlying parallel architecture. Further, the performance limits depend on the mapping of the problem on to the parallel architecture. Theoretical models abstract away real life limits such as the number of processors, synchronization requirements, scheduling and data distribution to derive the asymptotic limits. On the other hand, system-oriented studies are so concerned with mapping the algorithm to real architectures that it is difficult to know from the results of such studies where the parallelism inherent in the algorithm has been lost. The aim of this study is to address some of the issues in the interface between theory and architecture, from the point of view of algorithmic performance.

Parallel algorithms for certain problems theoretically guarantee a certain amount of speedup. But when these algorithms are implemented on existing architectures, the results may not agree with the expected theoretical speedup. Some inherent features in the algorithm, its implementation, and the hardware capabilities of the machine together contribute to the slow-down. The parallel algorithms usually assume a certain minimum number of processors to be available with an underlying interconnection topology between them. To implement such algorithms, we may have to make do with a limited number of processors and simulate the assumed interconnection. The language run time and the operating system may further introduce synchronization costs not inherent in the algorithm but necessary to implement them on the parallel machine. And lastly, the hardware capabilities that may introduce slow-down are: synchronization primitives, network latency for memory accesses, network contention generated by memory accesses, and the organization of the memory hierarchy.

One straightforward way to understand the architectural impact on parallel computation is to implement algorithms with intrinsic parallelism on parallel architectures and interpret the results with respect to the above factors. However, it is extremely difficult to accomplish this goal since the design space of parallel algorithms and parallel architectures is very huge. The first difficulty is in selecting a set of parallel algorithms that are non-trivial and occur as kernels in several applications. Secondly these kernels should be distinct in that they capture different algorithmic characteristics. Thirdly, the approach of the experimentation should be such that it does not make the results machine dependent. Developing a set of parallel kernels representative of several applications is an important objective which we hope to fulfill as a result of our ongoing research. As a first step in this direction we have chosen three kernels to be described shortly. To make the results of the experimentation meaningful beyond the architectures on which these are carried out, it is important to design the experiments carefully, and ensure that the measurements capture the high level architectural features alluded to earlier. To validate the hypotheses that we may have from the experimentation, we need to be able to extrapolate the experimental results to new architectures via simulation. However, the focus of this paper is to report on experimenting with kernels on specific parallel architectures.

We have chosen to perform our experiments on two parallel machines - the *Sequent Symmetry* and the *BBN Butterfly TC 2000* - with entirely different architectures. The Sequent is a bus based shared memory multiprocessor machine. All processors are identical (there are 26 processors in the system experimented with) each having a 64K cache memory and connected to a global shared memory through a single system bus. Sequent falls into the category of Uniform Memory Access (UMA) machines in that a user has no *a priori* knowledge of the access time for a given memory location. The Butterfly on the other hand is a collection of processors (there are 45 processors in the system experimented with), each with its own local memory. However each processor can access non-local memories via a multistage interconnection network. Such non-local accesses are more expensive than local ones. Thus Butterfly falls into the category of Non-Uniform Memory Access (NUMA) machines in that a user has *a priori* knowledge of access times to a given memory location based on whether it is local or non-local.

Since programming of parallel machines has tended to be the outgrowth of programming sequential machines, parallel algorithm development as well as models of parallel computation have predominantly used a shared memory abstraction. Therefore we have chosen three shared memory parallel algorithms - *List Ranking*, *Parallel Prefix* and *Optimal Binary Search Tree* - each of which occurs as a kernel in several applications. List ranking is a fundamental operation on lists. The problem is: given a linked list, compute the distance of each cell from the end of the list. This algorithm is data dependent and thus the memory access patterns depend on the data placement. For such a problem, it is expected that data partitioning would play a vital role on performance, especially when implemented on a NUMA machine.

Parallel prefix on the other hand is a data oblivious algorithm implying that the running time of the algorithm is independent of the input data. The prefix problem takes as input $n$ elements and gives an $n$ element output, where the *i-th* output element is the product of the first $i$ input elements. If the input data is partitioned appropriately for such an algorithm, then it is possible to reduce the amount of accesses to non-local data by a processor.

Optimal binary search tree is another data oblivious algorithm that takes a set of weights (leaf nodes) as input and constructs a tree of minimal weight. The interesting variation from the other two algorithms is the work imbalance that is inherent during different phases of the algorithm.

After defining an execution and implementation model for the shared memory paradigm (Section 2), the results from the experimentation are presented (Section 4). Though the results presented are with respect to the experimentation of the three algorithms on the two machines, the conclusions drawn have general applicability from the point of view of the performance implications of parallel algorithms (see Section 5).

## 2 An Execution Model for the Shared Memory Paradigm

The model of execution used is single-program-multiple-data wherein each process independently executes the same code on a different portion of the data (*data partitioning*). We use the term *task* to mean a unit of work and the term *process* to mean a virtual processor. The input data is partitioned into units of work (tasks). Each process performs the same set of operations on the task assigned to it. There are four important issues to be considered in mapping this execution model onto shared memory architectures.

### 2.1 Scheduling

For implementing these algorithms, we define a set of tasks and let processes work on these tasks. The assignment of a task to a process is called scheduling. This level of scheduling is from the point of view of algorithms as opposed to an operating system scheduler that does resource management[1]. Each process participating in the parallel algorithm is bound to a distinct physical processor, and thus there is no multiprogramming. This ensures that the operating system scheduler does not interfere with the scheduling that is happening in the parallel algorithm. Our experimental work considers two types of scheduling. *Static* scheduling pre-assigns tasks to processes at compile time. Even though static scheduling may be easy to program, it may not always be the most efficient in terms of processor utilization. *Dynamic* scheduling assigns the tasks to processes at run time and thus has the potential for better processor utilization. However dynamic scheduling may result in extra synchronization costs because it normally involves accesses to globally shared queues (critical sections).

For static scheduling, the data partition that each process works on is pre-determined. For dynamic scheduling on the other hand, the chunk of data that a process works on is determined at run time. The chunks of data that are ready for execution are in a global queue. When each process finishes with its chunk, it gets the next piece of work from this global queue. For the algorithms discussed in this paper, we simulate the global queue with an atomic counter. To exploit spatial locality which may be important in certain algorithms such as parallel prefix, the chunk of data assigned to a process statically or dynamically is always contiguous.

[1]Recently, several researchers [LV90, TG89, ZM90] investigate the relative merits of dynamic and static scheduling policies at the operating system and application level for multiprocessors.

## 2.2 Task Granularity

Task granularity has two dimensions : *computation granularity* and *data granularity.* The former deals with the amount of computation that a process needs to do for the particular task while the latter involves the size of the data partition in the task. The notion of computation granularity is to capture the amount of computation a task does on each input. In other words, it is a measure of the work done in a task for a given memory reference pattern as determined by the data granularity. These are important input parameters that need to be considered to determine the effect of task granularity and synchronization on performance. The data granularity is varied by changing the number of data elements in the chunk that is allotted to a process while the computation granularity is varied by introducing some artificial work in the place where the process solves the problem for the chunk allotted to it. The artificial work is an idle loop and the loop count is used as a measure of the computation granularity.

## 2.3 Synchronization

During the course of execution a process may need to synchronize with other processes. A special form of synchronization is the *barrier synchronization* where all the processes involved need to arrive at a common point in the course of execution before any of them may proceed. In implementing this execution model barrier synchronization is used.

## 2.4 Data Distribution

The above description has applicability to both the Sequent and the Butterfly. But on the Butterfly, there is an added dimension that is worth studying : *data distribution* (because of non-uniform memory access). The data distribution patterns that are included in this study are : *random distribution* (a processor may need to access any memory module for its data chunk), *skewed distribution* (in which successive data elements are allocated in successive virtual processors), and *local allocation* (where a processor's data chunk is locally allocated). Each of these data distributions generates differing network traffic. Such a study is expected to give us a feel for the effect of the switch contention and remote memory accesses on the performance. The corresponding effect for the Sequent is simulated by varying the computation granularity. If the computation granularity is made small, then there is more synchronization overhead which in turn generates more traffic on the bus.

## 3 Related Work

To our knowledge, there are very few experimental studies that investigate the impact of architectural features on algorithmic performance. Anderson [And90a] reports results of an experimental and analytical study of parallel merge sort. In this study, implementation of this algorithm on the Sequent is used to verify the speedup with different number of processors with respect to the analytical model. Yew et al. [CSY90], analyze specific parallel programs to identify the appropriate grain size of parallelism that exists in these programs. Further they present a simulation study to measure the impact of synchronization overhead on the execution of these programs. Lin and Snyder [LS90] compare message passing and shared memory paradigms for implementing specific parallel algorithms on shared memory multiprocessors. Our work is more general in that we experiment with algorithms that represent classes of problems and study synchronization, scheduling and task granularity issues in implementing these algorithms.

## 4 Observed Results

The implementations of these algorithms use the parallel programming library [Seq87] on the Sequent, and the uniform system [BBN86] on the Butterfly. In the experiments, *completion time* of the program is used as a measure of the performance of the parallel algorithm. The results for the uniprocessor cases are obtained by running the multiprocessor parallel algorithm on a single processor. Hence the speedup using $n$ processors refers to the ratio of the completion time of the parallel algorithm on *1* processor to that on $n$ processors. Alternatively, speedup could be defined as the ratio of the execution time of a sequential program on a single processor to that of a parallel program on a parallel processor. However, the focus of this study is to understand the performance potential of parallel algorithms and hence the earlier definition is used in the rest of the paper.

### 4.1 List Ranking

A parallel algorithm for the list ranking problem is discussed in [Wyl79, KR90]. Figure 1 shows the algorithm and Figure 4 shows the corresponding pseudo code. The algorithm is data dependent.

The randomness of access of the list elements does not favor data partitioning. Therefore, it is not known *a priori* the best way to partition the data and assign it to the processors. This feature of the algorithm makes it interesting to study its performance on architectures with different organization and memory access capabilities, like the Sequent and the Butterfly. The input list (of size 32K) is generated using a standard random number generator.

### 4.1.1 Sequent Implementation

The results for static scheduling (see Section 2.1) are shown in Figures 6 and 7. Figure 6 plots the speedup of the algorithm versus the number of processors for different computation granularities, while Figure 7 plots the completion times versus the number of processors. At low computation granularities, there is almost a linear speedup (Figure 6) as we increase the number of processors upto 12. The speedup curve gradually tapers off giving no further improvement in performance after 20 processors. This result shows that: (a) the processors are almost always allocated an equal amount of work, (b) there is negligible amount of overhead in synchronization (for instance, of a completion time of 1 second in the 16 processor case, the total amount of time spent in synchronization is around 10 milliseconds), and (c) the bus may not be a bottleneck for small number of processors but clearly saturates beyond 20 processors. Now, increasing the computation granularity (Figure 7) increases the completion time as can be expected. It is observed that at higher computation granularities we get close to linear speedup. This trend further supports our earlier remark that the bus may not be a bottleneck for static scheduling at sufficiently high computation granularities. Since the processors are almost always allocated an equal amount of work between any two successive synchronization points, there is negligible time spent waiting at the barrier. However at low computation granularities, contention for the bus does become an issue affecting the speedup curve.

Figure 8 shows the effect of data granularity on performance when dynamic scheduling is used on the Sequent. When it is extremely fine grain, the dynamic scheme performs very poorly compared to the static scheme. Extensive contention for the globally shared queue is the reason for this behavior. The bus does become a bottleneck in this case and hence this poor performance is quite understandable. This experimental result corroborates the simulation result reported in [LR90], wherein they show that lock contention leads to poor performance in bus-based shared memory multiprocessors. It is also the reason why the performance for higher number of processors (8 and 16 in the Figure) is much poorer than lower number of processors for fine grain data granularity. As the data granularity increases, the performance improves until it becomes comparable to the static case. Increasing the data granularity further does not result in improved performance. In fact, when the data granularity is increased beyond that of the static case, the completion time increases which is quite understandable since one or more processors may be idle between any two barrier synchronization points. For a wide range of chunk sizes, dynamic scheduling seems to fare as well as static scheduling. Over these grains, the contention for the global queue is almost negligible. It is interesting to note that the dynamic scheme does not fare better than the static scheme at any point. Thus we may conclude that there is equal load balancing at all points of execution in the static case.

Figure 9 shows the effect of increasing the computation granularity. The results, as in the static case, are quite predictable for larger chunk sizes. The interesting observations are for smaller chunk sizes. It is reasonable to expect that increasing the computation granularity may result in poorer performance. However it is seen that as the computation granularity is increased, the performance of the 8 and 16 processor cases improve. This improvement can be explained with reference to the contention for the queue (bus traffic). By increasing the computation granularity, the time between any two successive bus accesses increases, thus reducing the contention for the shared queue thereby improving the performance. As the computation granularity is increased further, the effect of contention becomes less significant which explains the observed result. The contention effect is quite similar to what is explained as quiesce[2] time in [And90b]. Note that this effect is less significant for lower number of processors. which explains the better performance for 4 processor case as compared to the 8 or 16 processor cases for small computation granularity.

### 4.1.2 Butterfly Implementation

On the Sequent, there is no choice for data placement since it is a uniform memory access machine. On the other hand, the input list could be allocated in any of the processors' memories on the

[2]Quiesce time is the time it takes for spurious bus contention to settle down in a bus-based shared memory multiprocessor upon the release of a lock.

Butterfly, making data partitioning another interesting dimension to study. Data partitioning is expected to affect performance because of the non-uniform memory access times. We study this problem with the three types of data distributions that we enumerated earlier (see Section 2.4).

Figure 10 shows the performance for static scheduling for each of these data partitioning schemes. Local allocation seems to have the best performance of the three although the algorithm is data-dependent. In the local allocation scheme, a processor writes only to the local memory even though it may read from non-local memory modules. In the other two schemes, the accesses (both reads and writes) are purely random and they seem to perform equally worse. These results seem to indicate that the way in which we partition data has a significant effect on performance.

Figure 11 shows the effect of increasing the computation granularity on the speedup for the local allocation scheme. The results show a remarkable similarity to the results obtained on the Sequent (Figure 6): A linear speedup is observed for small number of processors regardless of computation granularity. The speedup diminishes beyond 12 processors at low computation granularity, while it remains almost linear for higher computation granularity.

Increasing the number of processors results in an interplay of increased computation power, higher probability of non-local memory accesses (thus increasing network traffic) and the increased cost of synchronization. The first is a positive factor while the latter two have negative effects. Non-local memory access affects performance in two ways: increased latency and possible switch contention. For larger number of processors (greater than 12 in the Figure), these retarding effects tend to dominate. However, we do continue to get an improvement in performance upto 32 processors on the Butterfly, which is unlike the result obtained on the Sequent. This leads us to conclude that the bus on the Sequent is a far more shared resource than the switch on the Butterfly. Increasing the computation granularity exploits the additional compute power available with increased number of processors, thus offsetting the negative effects even further. Experimentation with the other two data distribution schemes (random and skewed) also yielded similar results.

The effect of varying the data granularity for dynamic scheduling is shown in Figure 12. Since in this scheduling strategy processors are assigned to tasks at run-time, the local and skewed allocation schemes may not have much meaning. Therefore only the random allocation scheme is considered. The performance at low chunk sizes is much worse than that observed in the static case. As the chunk size is increased, the performance approaches that of static scheduling. This behavior is quite similar to the observed results for dynamic scheduling on the Sequent (Figure 8. The only difference is that the anomalous behavior observed at low chunk sizes on the Sequent is non-existent on the Butterfly.

Figure 13 shows the effect of computation granularity on performance for dynamic scheduling. We only present the results for low chunk sizes (1 in Figure 13) since at higher chunk sizes, the behavior is expected to be similar to static scheduling. Increasing computation granularity does not affect network traffic and thus the performance improves with the increase in the number of processors for a given computation granularity. We noted that in dynamic scheduling, the processors have to access a global shared queue of tasks. The contention that results from this queue has a detrimental effect (at low computation granularity) on the performance for Sequent with larger number of processors (see Figure 9). On the other hand, this contention is not as pronounced on the Butterfly as can be seen in Figure 13. This observation reiterates the fact that the bus on the Sequent is a much more shared resource than the switch on the Butterfly.

### 4.2 Parallel Prefix

An algorithm for the parallel prefix problem is discussed in [LF80], and Figure 2 gives the algorithm and Figure 5 shows the pseudo code for this algorithm. The algorithm is data oblivious. Each phase of the parallel part can be performed only after all processors in the previous phase have completed their task. The recognition of the end of a phase is by waiting on a barrier. As kernels, parallel prefix and list ranking have the same loop structure; and as algorithms they use the same recursive doubling paradigm. Yet, the locality of reference due to the data oblivious nature of the parallel prefix algorithm leads to a significantly different memory reference pattern. In the static scheduling case, each processor knows the part of the data it has to work on, and all the work is equally distributed. In the dynamic scheduling case, each processor identifies the chunk of data it has to work on, by using a global counter (see Section 2.1). In both cases, computation granularity is varied as in list ranking (see Section 4.1.1). The data size used in these experiments is 32K. Given the difference in the memory reference patterns for the two algorithms (list ranking and parallel prefix) due to the data dependence and the amount of work done in each step, it is reasonable to expect that the observed results for the two to be quite different. However, it is observed that there are no significant differences between the performance patterns of the two algorithms on the two architectures.

### 4.2.1 Sequent Implementation

Figures 14 and 15 show the performance of the algorithm using static scheduling on the Sequent. The results for static scheduling are remarkably similar to the corresponding results for list ranking (see Figures 6 and 7). An almost linear speedup (Figure 14) is achieved for all computation granularities. The observed speedup tends to diminish at lower computation granularities beyond 20 processors. This result confirms our earlier observation regarding the negligible effect of synchronization cost and bus overhead on the performance of these algorithms for smaller number of processors (12 for list ranking and 20 for parallel prefix). The flattening of the speedup curve at lower computation granularity occurs at higher number of processors for parallel prefix owing to the locality of accesses exhibited by the parallel prefix algorithm as compared to the list ranking algorithm.

Figure 16 shows the effect of varying the data granularity on the completion time. Figure 17 shows the effect of increasing computation granularity on the completion time for fixed low data granularity (chunk size = 1). As in static scheduling, the results in dynamic scheduling are also quite similar to the corresponding results for list ranking (see Figures 8 and 9). The minor differences between the two sets of results may be attributed to the differences in the nature (such as the data dependence and the amount of work done in each step) of the two algorithms. For example, in parallel prefix the crossover point where the 16 processor performance fares better than that of the 8 processor case occurs at higher computation granularity compared to list ranking.

### 4.2.2 Butterfly Implementation

We experimented with three data distribution schemes in the case of the parallel prefix problem as in the list ranking case. Figure 18 shows the completion time versus the number of processors for the three data distributions in the case of static scheduling. As in the case of list ranking (see Figure 10), local data distribution scheme performs the best and is almost linear with the number of processors, while the other two data distributions fare equally worse. The effect of computation granularity on speedup shown in Figure 19 follows the same pattern as in list ranking (see Figure 11).

The results for dynamic scheduling on the Butterfly are shown in Figures 20 and 21. Figure 20 shows the effect of varying the data granularity on the performance, and Figure 21 shows the effect of varying the computation granularity on the performance. The results are very similar to the corresponding ones obtained for list ranking (see Figures 12 and 13).

## 4.3 Optimal Binary Search Tree

A standard dynamic programming algorithm for this problem computes a 2-dimensional cost matrix as shown in Figure 3 [AHU74]. This algorithm is data oblivious and traverses one diagonal after another (the values for the elements in the current diagonal depend on the values in the previous diagonal). The number of diagonal elements to be computed in each phase decreases by 1 as we step through the diagonals. Therefore, in the static scheduling case we divide the number of diagonal elements in each phase by the number of available processors and determine the elements that each processor has to work on. In the dynamic scheduling case, a global counter is used. Note that, for this algorithm, the unequal workload at each phase is an additional detrimental factor when compared to the other two algorithms.

### 4.3.1 Sequent Implementation

Figure 22 shows the performance of this algorithm on the Sequent using static and dynamic scheduling. An almost linear speedup is observed for static scheduling showing that the unequal workload is not a significant detrimental factor.

Figure 23 shows the performance using dynamic scheduling with respect to chunk size. Unlike the other two algorithms, the performance worsens as the chunk size increases with multiple processors. This result can be explained as follows. In the dynamic scheduling case, the number of units of work depends on the chunk size and the number of diagonal elements to be completed. For example, there are 2 units of work when the chunk size is 16 and the number of diagonal elements is 31. Thus for the 4 processor case, 2 processors remain idle, creating work imbalance. This work imbalance is not significant for lower chunk sizes, but as the chunk size increases the imbalance does affect performance, particularly for larger number of processors (see the curve for 16 processors in

Figure 23).

### 4.3.2 Butterfly Implementation

Figure 24 shows the performance on the Butterfly using static and dynamic scheduling with skewed data allocation scheme. Since the data structure in this algorithm is 2-dimensional, only the skewed allocation scheme is used. It can be seen from the results for static scheduling that the speedup is not linear in the number of processors. The increase in the non-local memory references with the number of processors is the reason for this behavior.

Figure 25 shows the effect of chunk size on the performance for dynamic scheduling. For a given number of processors, increasing the chunk size results in poorer performance. This behavior is very similar to that observed on the sequent (see Figure 23), and is due to the work imbalance. However, the detrimental effects seem to have a more pronounced influence on the Butterfly than on the Sequent (the performance of 8 and 16 processors at higher chunk sizes is much worse than with lower number of processors).

## 4.4 Discussion

It is observed that there is almost a linear improvement in performance with the number of processors on the Sequent for all the three algorithms. Adding more processors while increasing the computing power also increases the synchronization overhead. But the results indicate that for the algorithms studied the overhead is not very significant on the Sequent. Further these results also show that the bus is not a bottleneck when there is sufficient computation granularity.

However a linear speedup is not always realized on the Butterfly for the three algorithms. Data distribution in the memories of the processors is another dimension on the Butterfly that seems to have a significant impact on the performance. For example, the speedup curve is almost linear for the parallel prefix algorithm with local data distribution. The reason for this result is twofold. The algorithm is data oblivious and the local data distribution provides a similar effect to having a private cache on the Sequent for this algorithm. For the other two data distributions considered, we do not observe a linear speedup for any of the three algorithms. Thus, we may conclude that even on the Butterfly synchronization overhead is not a dominant cost in limiting algorithmic performance; on the other hand network latency is a dominant factor. In comparing the data distribution schemes, local allocation seems to be the best followed by random allocation for both list ranking and parallel prefix algorithms.

It is intuitive that static scheduling should perform the best when the workload is known in advance. Our observations confirm this intuition. However, for data dependent algorithms such as list ranking the workload may be uneven. Furthermore, even for some data oblivious algorithms (such as optimal binary search tree) there could be uneven workload between phases, indicating that static scheduling may be inefficient. It is reasonable to expect dynamic scheduling to perform better under such circumstances. In fact our observations are contrary to this intuition, which can be explained by the fact that there is an inherent overhead in dynamic scheduling. This overhead is significant at low computation granularity. As the computation granularity is increased, the performance of dynamic scheduling tends towards static scheduling and may even become better when there is a significant imbalance in the workload.

An artifact of implementing dynamic scheduling is the choice of data granularity. When the data size is very small there is more overhead in dynamic scheduling. Our results show that this overhead can be overcome only by increasing the computation granularity. Note, that for a given data size, very large data granularity (chunk size) generates uneven workload, thus having a detrimental effect on the performance.

## 5 Concluding Remarks

The work presented in this paper represents the first step in understanding the implications of architectural features on the performance of parallel algorithms. The approach taken is an experimental one, and there have been very few experimental studies of this nature. The algorithms used in this study were chosen since they are representative of kernels that occur in several applications, and each of them has a distinct set of characteristics. While the results (presented in Section 4) themselves do not provide sufficient basis for generalization in terms of the types of parallelism that can be exploited in algorithms and their match with the architectural features, they do provide the basis for developing general hypotheses regarding the match between algorithmic requirements and architectural features. This study has also been fruitful in confirming some of these hypotheses through experimentation. Our ongoing research extends this preliminary work to identify a suite of parallel kernels which would be representative of several large applications, and explore the match between their requirements and the architectural features.

There are several hypotheses regarding the effects of architectural features on algorithmic performance that motivate this research. One has to do with memory organization. For example, a multistage interconnection network can provide higher throughput, at the cost of increased latency for each individual request. On the other hand, bus-based systems offer lower latency for each individual request, but provide lower overall throughput. Our results show that while in general the bus is not a hindrance on the Sequent for a small number of processors, the network latency on the Butterfly is a hindrance in achieving linear speedup. An explanation of this result is that the network in the butterfly is more of a shared resource than the bus in the Sequent, since the latter has private (coherent) caches associated with each processor while the former does not.

The second hypothesis has to do with task granularity. While smaller granularity allows increased parallelism, it would also engender more synchronization overhead. A larger granularity while reducing the synchronization overhead limits the exploitation of the available parallelism especially when there are a large number of processors. Our results confirm this hypothesis.

The third hypothesis deals with the scheduling overhead. This overhead could be a limiting factor in exploiting the available parallelism in a computation. This factor is related to the granularity of tasks that we mentioned earlier. The specific strategy used could affect the performance of the algorithm. For example, with the dynamic scheduling scheme, there is high contention for the global queue especially if all the processors are looking for work after a barrier synchronization. This hypothesis is also borne out by our results since static scheduling outperforms dynamic scheduling for the most part. It is natural to wonder if a dynamic scheduler that uses a distributed queue would have resulted in dynamic out-performing static scheduling. However, there are two costs associated with the explicit synchronization that is generated as a result of dynamic scheduling: wait time at such synchronization points, and the intrinsic cost of the synchronization operation. While a distributed queue may help reduce the amount of wait time, it may actually increase the number of synchronization operations. Of course, we cannot generalize that static scheduling is better than dynamic for arbitrary kernels. However, it is reasonable to expect that this hypothesis may hold for other kernels that exhibit similar memory reference patterns. Alternative strategies for dynamic scheduling such as *guided self-scheduling*, that rely on some form of compiler support, may help alleviate some of the inherent overhead in dynamic scheduling [PK87]. However, such strategies would require careful data dependence analysis at compile time. This is clearly a fruitful direction for extending our work in the scheduling dimension from the point of view of our bigger goals for the whole project.

The fourth hypothesis concerns synchronization primitives on the available parallel machines. It can be expected that synchronization would be crucial for both dynamic scheduling and algorithm implementation. Although the Sequent and the Butterfly provide different types of synchronization primitives, these differences do not significantly affect the algorithmic performance. This is due to the fact that the amount of synchronization required in these algorithms is not significant compared to the amount of computation involved. On the other hand, the absence of efficient high-level synchronization primitives such as queue manipulation primitives on both these architectures limits the performance of dynamic scheduling compared to static scheduling.

The last hypothesis is of a general nature and deals with the structure of parallel algorithms. One expects that PRAM algorithms (such as parallel prefix and list ranking) with different characteristics (such as data dependence and amount of work done in each step) to perform differently on a given real shared memory architecture, even though they may have similar running times on the PRAM. For example, both list ranking and parallel prefix algorithms have a running time of $O(\log n)$. However, for shared memory architectures that we experimented with the performance patterns of the two algorithms are quite similar which seem to support some of the inherent assumptions in the PRAM model. This result is also significant since the Sequent and the Butterfly have considerably different memory organizations.

This study has generated several interesting research directions: a more realistic theoretical framework for analyzing algorithms and architectures, a classification of the architecture primitives that would allow the realization of the parallelism promised by the algorithms, an exploration of novel architectures suggested by the structure of the algorithms, and a set of metrics for evaluating the performance of parallel algorithms. Another important research direction suggested by this study is a novel top-down approach to the design of scalable shared memory multiprocessors, which quantifies the reliance of the algorithmic performance on architectural features such as synchronization, latency, and network contention to help synthesize the right set of primitives. Many of the current studies approach the problem starting from low level system issues. They try to address the problems of synchronization, latency and network contention without devoting much attention to the influence of such issues on the performance of real parallel algorithms. Our top-down approach can quantify the effect of each architectural artifact on the performance of parallel algorithms which is really an indication of the scalability of a parallel system.

## 6 Acknowledgments

This work has been funded in part by NSF grants CCR-8711749, CCR-8619886, MIPS-8809268, an NSF PYI Award MIPS-9058430, and an equipment grant from Digital Equipment Corporation. We are also thankful to Argonne National Laboratory for making the Sequent Symmetry and BBN Butterfly TC 2000 available to us for our experimental work. We would like to thank Joonwon Lee who participated in the earlier phases of this research.

## References

[AHU74] A. V. Aho, J. E. Hopcroft, and J. D. Ullman. *The Design and Analysis of Algorithms.* Addison-Wesley, 1974.

[And90a] Richard J. Anderson. An Experimental Study of Parallel Merge Sort. Preliminary Version 0.1 . University of Washington, Seattle, 1990.

[And90b] Thomas E. Anderson. The Performance of Spin Lock Alternatives for Shared-Memory Multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 1(1):6–16, January 1990.

[BBN86] BBN Advanced Computers Inc., Massachusetts. *The Uniform System Approach to Programming the Butterfly Parallel Processor*, 1986.

[CSY90] D. Chen, H. Su, and P. Yew. The Impact of Synchronization and Granularity on Parallel Systems. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 239–248, 1990.

[KR90] Richard M. Karp and Vijaya Ramachandran. A Survey of Parallel Algorithms for Shared-Memory Machines. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, pages 869–942. North Holland, Amsterdam, 1990.

[LF80] R. E. Ladner and M. J. Fisher. Parallel Prefix Computation. *Journal of Association of Computing Machinery*, 27(4):831–838, October 1980.

[LR90] Joonwon Lee and Umakishore Ramachandran. Synchronization with Multiprocessor Caches. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 27–37, 1990.

[LS90] Calvin Lin and Lawrence Snyder. A Comparison of Programming Models for Shared Memory Multiprocessors. In *Proceedings of the 1990 International Conference on Parallel Processing*, pages II 163–170, 1990.

[LV90] Scott T. Leutenegger and Mary K. Vernon. The Performance of Multiprogrammed Multiprocessor Scheduling Policies. In *Proceedings of the ACM SIGMETRICS 1990 Conference on Measurement and Modeling of Computer Systems*, pages 226–236, 1990.

[PK87] Constantine D. Polychronopoulos and David J. Kuck. Guided self-scheduling : A practical scheduling scheme for parallel supercomputers. *IEEE Transactions on Computers*, C-36(12):1425–1439, December 1987.

[Seq87] Sequent Computer Systems Inc., Oregon. *Sequent Guide to Parallel Programming*, 1987.

[SSL$^+$91] Anand Sivasubramaniam, Gautam Shah, Joonwon Lee, Umakishore Ramachandran, and H. Venkateswaran. Experimental Evaluation of Algorithmic Performance on Two Shared Memory Multiprocessors. In *Proceedings of the First International Symposium on Shared Memory Multiprocessing*, pages 13–24, Tokyo, Japan, April 1991.

[TG89] A. Tucker and A. Gupta. Process Control and Scheduling Issues for Multiprogrammed Shared Memory Multiprocessors. In *Proceedings of the 12th ACM Symposium on Operating System Principles*, pages 159–166, 1989.

[Wyl79] J. C. Wyllie. *The Complexity of Parallel Computations.* PhD thesis, Department of Computer Science, Cornell University, 1979.

[ZM90] John Zahorjan and Cathy McCann. Processor Scheduling in Shared Memory Multiprocessors. In *Proceedings of the ACM SIGMETRICS 1990 Conference on Measurement and Modeling of Computer Systems*, pages 214–225, 1990.

```
For $log\ n$ iterations repeat
    In parallel, for $i$ := $1,\ldots,n$ do
        list[$i$].rank $\leftarrow$ list[$i$].rank + list[list[$i$].successor].rank;
        list[$i$].successor $\leftarrow$ list[list[$i$].successor].successor;
```

Figure 1: List Ranking Algorithm

```
Prefix($x$,$n$,$s$)
    If $n$=$1$ then $s_1 \leftarrow x_1$
    else
        in parallel, for $i$:=$1$ to $n/2$ do
            $y_i \leftarrow x_{2i-1}$ * $x_{2i}$
        prefix($y$,$n/2$,$ss$)
        $ss_0 \leftarrow$ identity
        in parallel, for $i$:=$1$ to $n$ do
            if $i$ even then $s_i \leftarrow ss_{i/2}$
            else $s_i \leftarrow ss_{(i-1)/2}$ * $x_i$
```

Figure 2: Parallel Prefix Algorithm

```
For $i$ := $1$ to $n$ do
    For $j$ := $i+1$ to $n$ do
        $C_{i,j} \leftarrow \min_{i \leq k < j}(C_{i,k} + C_{k+1,j}) + \sum_{k=i}^{j} w_k$
```

Figure 3: Optimal Binary Search Tree Algorithm

```
In parallel
   chunk = (DATA_SIZE / N_PROCS);
   start_index = chunk * MYID;
   end_index = chunk * (MYID + 1) - 1;

   for log(DATA_SIZE) steps {
      for ($i$ = start_index; $i$ <= end_index; $i$++) {
         B[$i$].rank = A[$i$].rank + A[A[$i$].successor].rank;
         B[$i$].successor = A[A[$i$].successor].successor;
      }
      Barrier_Synchronize();
      Switch_Pointers(A,B);
   }
```

Figure 4: Pseudo Code for List Ranking

```
In parallel
   chunk = DATA_SIZE  N_PROCS;
   start_index = chunk * MYID;
   end_index = chunk * (MYID + 1) - 1;

   for (i = start_index+1; i <= end_index; i++)
      input[$i$] = input[$i$] + input[$i$-1];

   pdata[MYID] = input[end_index];
   Barrier_Synchronize();

   for ($i$ = 2; $i$ <= N_PROCS; i = $i * 2$) {
      if (MYID mod $i$ == 0)
         pdata[MYID+$i/2$] = pdata[MYID+$i/2$] + pdata[MYID];
      Barrier_Synchronize();
   }

   pdata[MYID] = pdata[MYID] - input[end_index];
   for ($i$ = start_index; $i$ <= end_index; $i$++)
      input[$i$] = input[$i$] + pdata[MYID];
```

Figure 5: Pseudo Code for Parallel Prefix

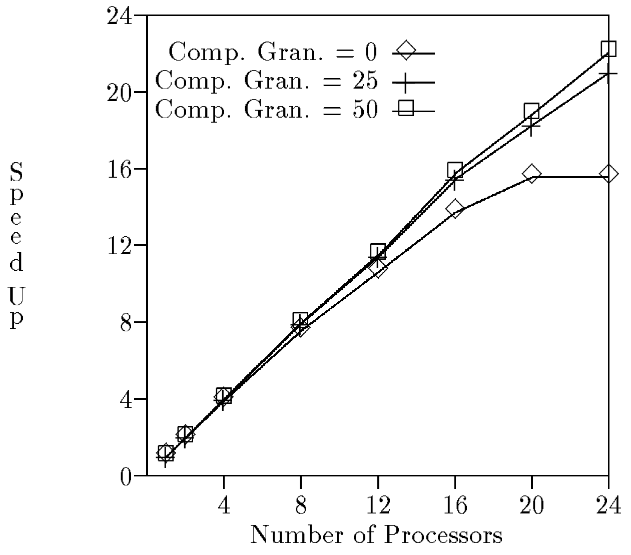

Figure 6: List Ranking - Static Scheduling on Sequent (Speedup)

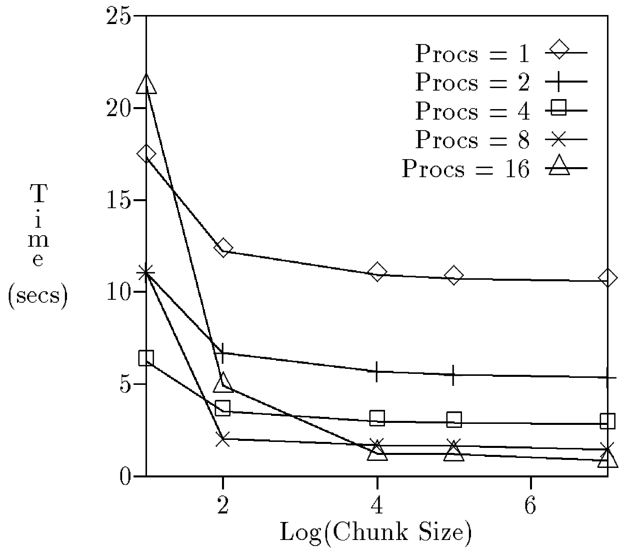

Figure 8: List Ranking - Dynamic Scheduling on Sequent (Data Granularity)

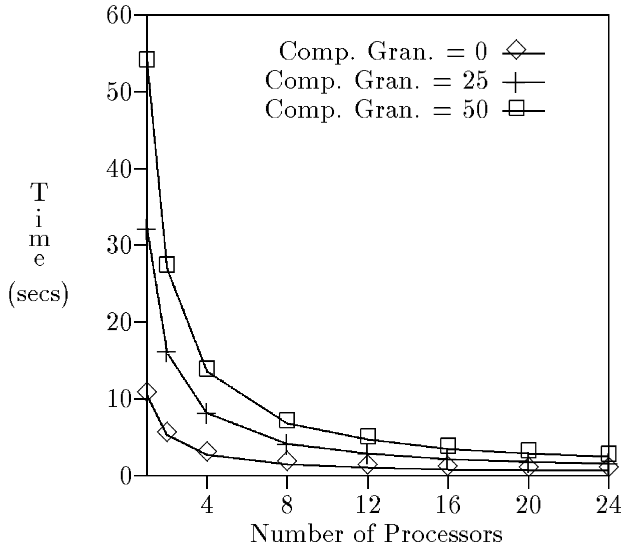

Figure 7: List Ranking - Static Scheduling on Sequent (Completion Time)

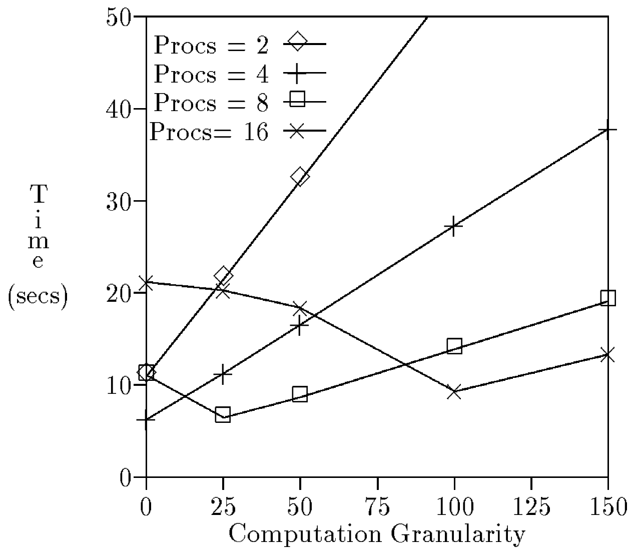

Figure 9: List Ranking - Dynamic Scheduling on Sequent, Chunk Size = 1 (Computation Granularity)

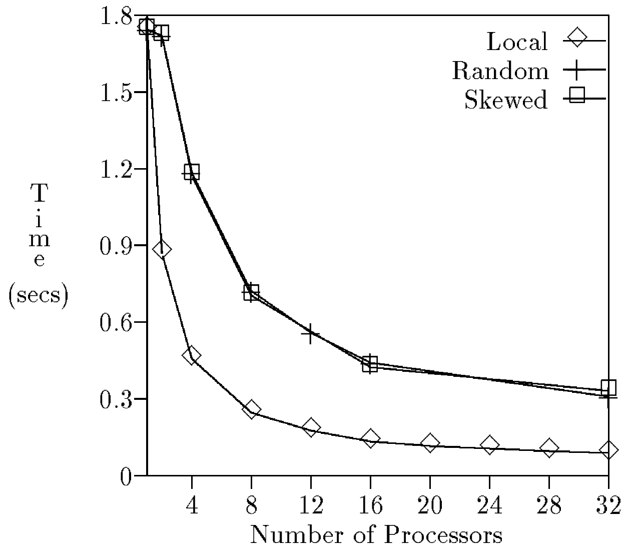

Figure 10: List Ranking - Static Scheduling on Butterfly (Data Distribution)

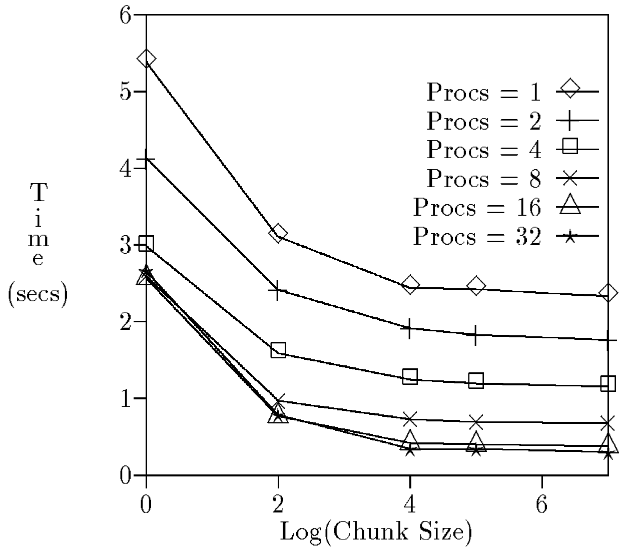

Figure 12: List Ranking - Dynamic Scheduling on Butterfly (Random Distribution, Data Granularity)

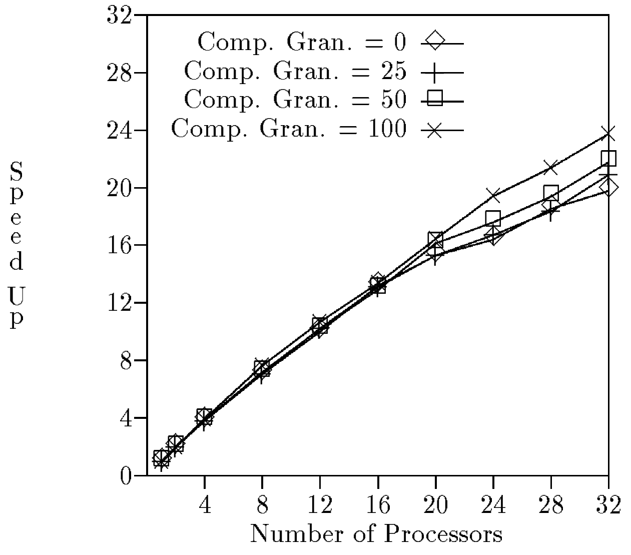

Figure 11: List Ranking - Static Scheduling on Butterfly (Local Distribution)

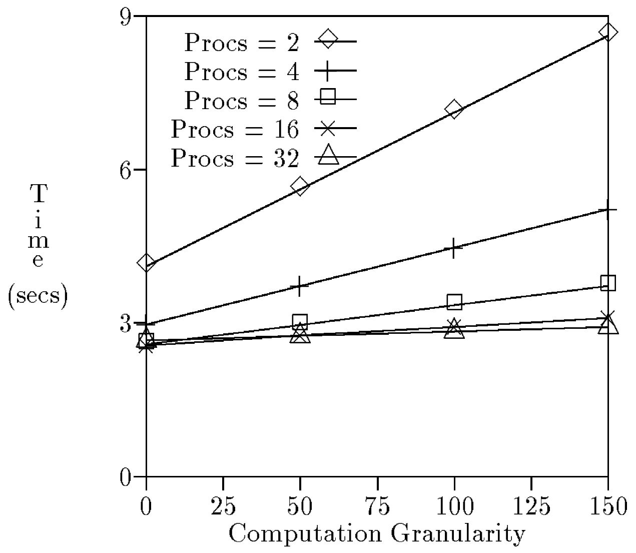

Figure 13: List Ranking - Dynamic Scheduling on Butterfly, Chunk Size = 1 (Random Distribution, Computation Granularity)

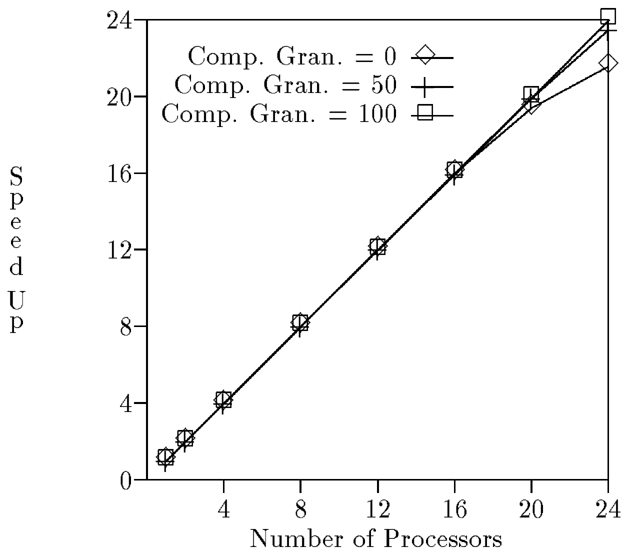

Figure 14: Parallel Prefix - Static Scheduling on Sequent (Speedup)

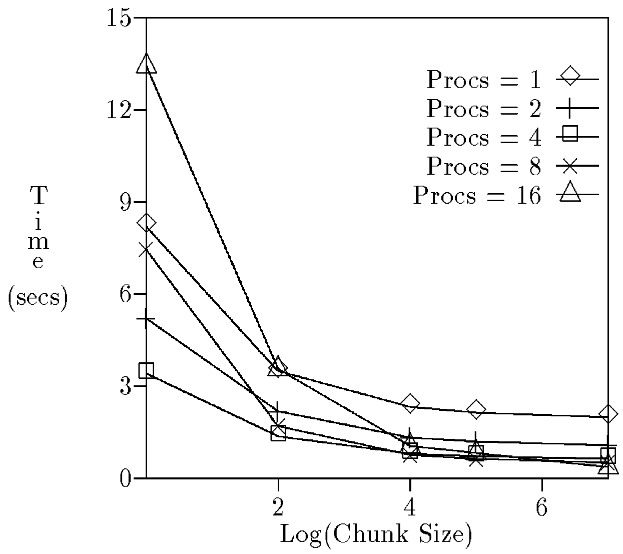

Figure 16: Parallel Prefix - Dynamic Scheduling on Sequent (Data Granularity)

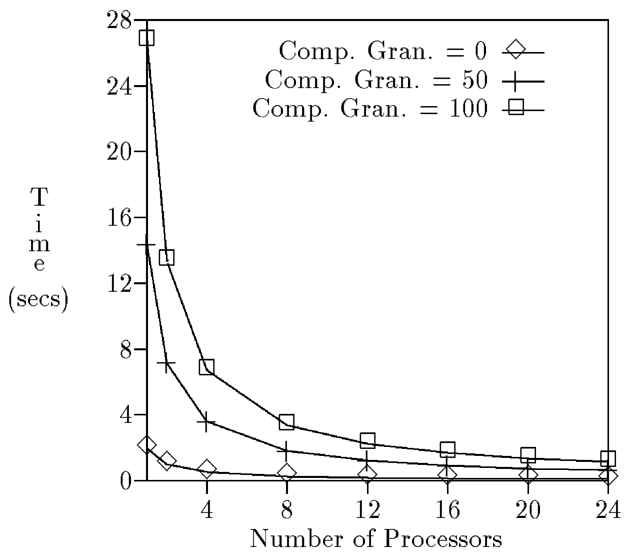

Figure 15: Parallel Prefix - Static Scheduling on Sequent (Completion Time)

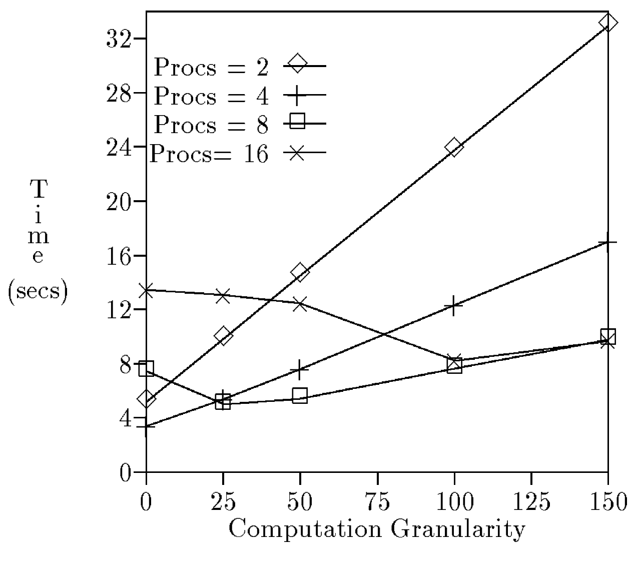

Figure 17: Parallel Prefix - Dynamic Scheduling on Sequent, Chunk Size = 1 (Computation Granularity)

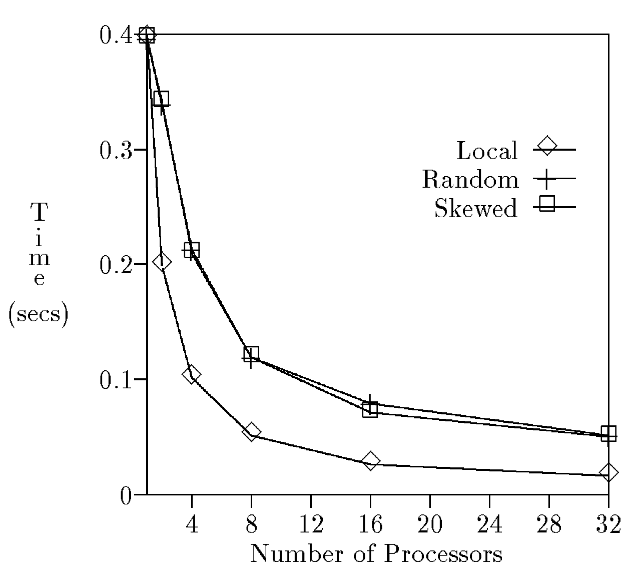

Figure 18: Parallel Prefix - Static Scheduling on Butterfly (Data Distribution)

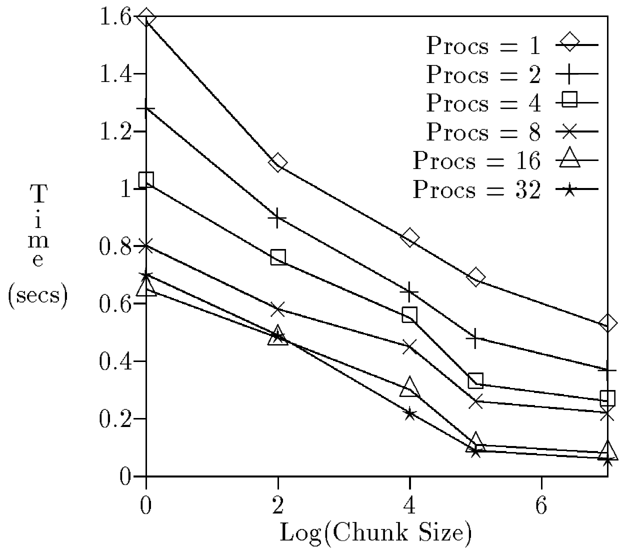

Figure 20: Parallel Prefix - Dynamic Scheduling on Butterfly (Random Distribution, Data Granularity)

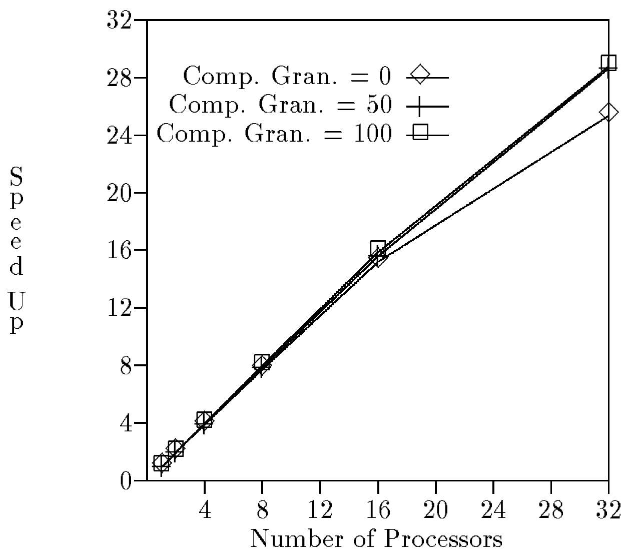

Figure 19: Parallel Prefix - Static Scheduling on Butterfly (Local Distribution)

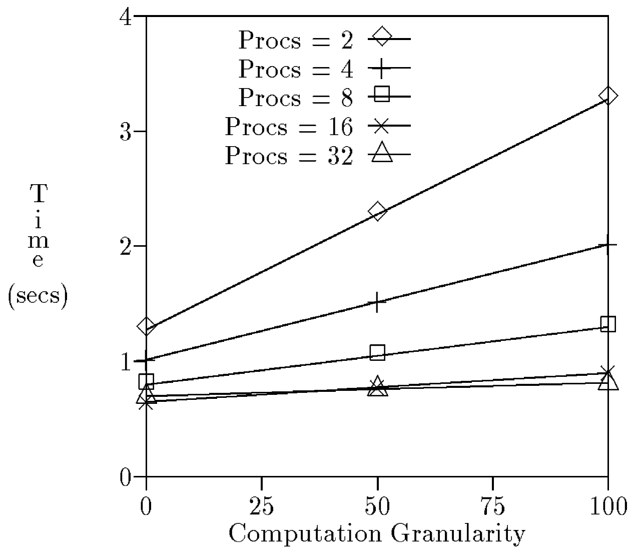

Figure 21: Parallel Prefix - Dynamic Scheduling on Butterfly, Chunk Size = 1 (Random Distribution, Computation Granularity)

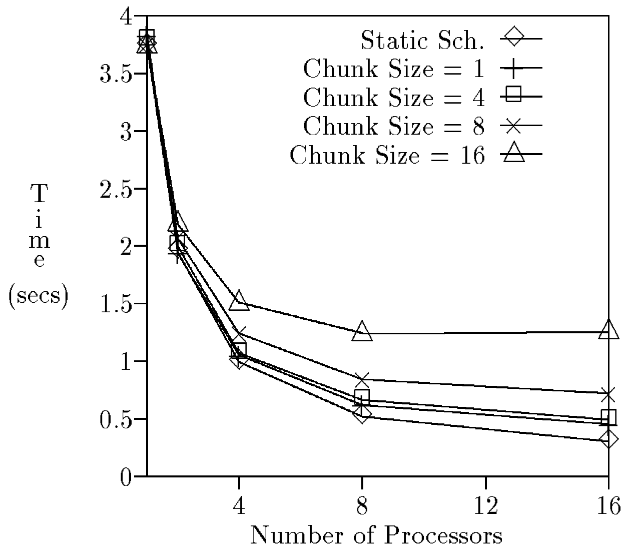

Figure 22: Optimal Binary Search Tree - Sequent Implementation

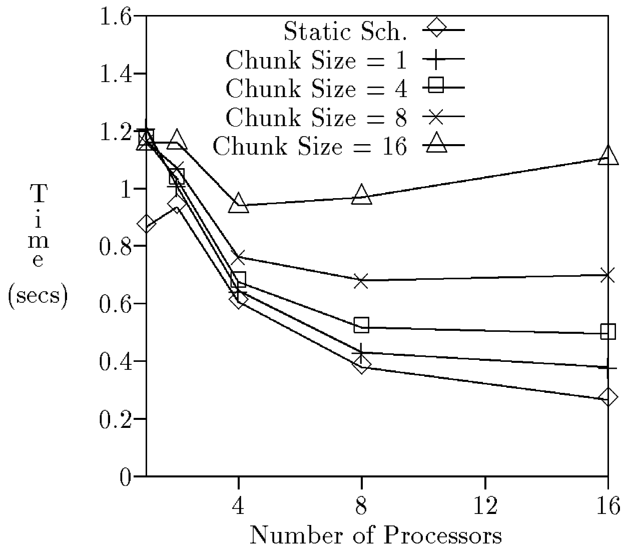

Figure 24: Optimal Binary Search Tree - Butterfly Implementation

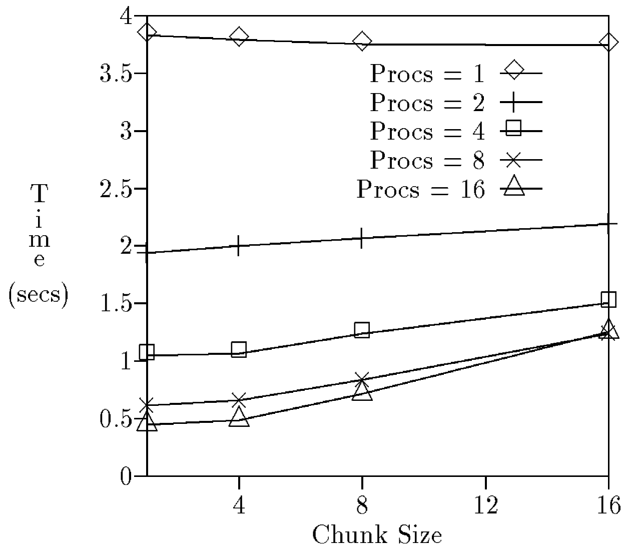

Figure 23: Optimal Binary Search Tree - Dynamic Scheduling on Sequent

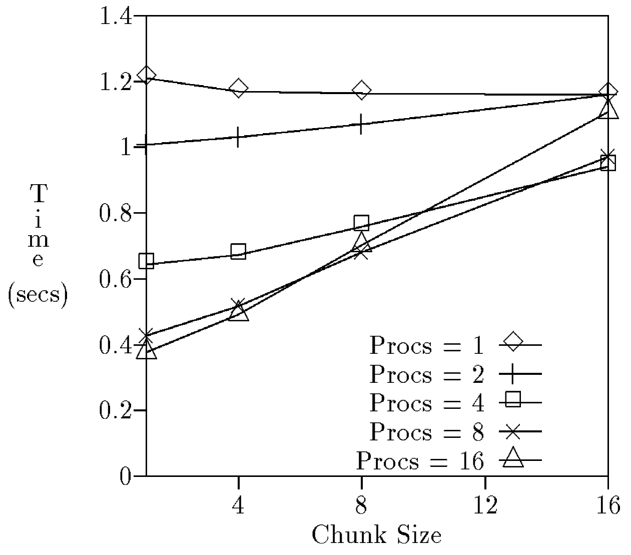

Figure 25: Optimal Binary Search Tree - Dynamic Scheduling on Butterfly