In *Proceedings of the First International Symposium on Shared Memory Multiprocessing*, pages 13-24, April 1991.

Experimental Evaluation of Algorithmic Performance on Two Shared Memory Multiprocessors*

Anand Sivasubramaniam *Gautam Shah* *Joonwon Lee*
Umakishore Ramachandran *H. Venkateswaran*
College of Computing
Georgia Institute of Technology
Atlanta, GA 30332-0280. USA.

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 BBN 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; the bus is not a bottleneck 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.

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.

*This work has been funded in part by NSF grants CCR-8711749, CCR-8619886, and MIPS-8809268.

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 are necessary to implement them on the parallel machine. And lastly, the hardware capabilities like synchronization primitives, memory access times and caching strategies may introduce further slow-down.

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. Therefore, we have chosen to perform our experiments on two parallel machines - the *Sequent Symmetry* and the *BBN Butterfly* - with entirely different architectures. The Sequent is a bus based shared memory multiprocessor machine. All processors are identical (in our system, there are 10 processors) 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, 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.

We have chosen three algorithms - *List Ranking*, *Parallel Prefix* and *Optimal Binary Search Tree* - each of which we feel identify a class of problems. 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.

The 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.

2 Programming Paradigms And Performance Metrics

There are several ways to implement the above algorithms on these machines. Both machines provide support for parallel programming such as the parallel programming library on the Sequent [11] and the Uniform System [2] on the Butterfly. In the description that follows, we use the term *task* to mean a unit of work and the term *process* to mean a virtual processor. 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). The parent process forks child processes that execute the parallel algorithm (works on the data partition assigned to it). During the course of execution of the child processes, they may need to synchronize. We use a barrier for this purpose.

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]. 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.

2.2 Task Granularity and Synchronization

The processes need to synchronize between executions of successive tasks. If the tasks are too fine grain, then this synchronization overhead may affect performance. To study this effect, we decided to vary the task granularity and study its impact on performance. 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. 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 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

[1] Recently, several researchers [9, 13, 15] investigate the relative merits of dynamic and static scheduling policies at the operating system and application level for multiprocessors.

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

Figure 1: List Ranking Algorithm

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.

In the discussions that follow, the completion time is used as a measure of the performance of parallel algorithms. In the Butterfly implementations, the times for process creation and termination are included in the completion time, whereas in the Sequent implementations, they are not. 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.

3 Related Work

To our knowledge, there are very few experimental studies that investigate the impact of architectural features on algorithmic performance. Anderson [10] 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. [3], 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 [8] 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

4.1 List Ranking

A parallel algorithm for the list ranking problem is discussed in [14, 4]. Figure 1 shows a pseudo-code for this algorithm.

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 (32K on the Sequent and 8K on the Butterfly) is generated using a standard random number generator.

4.1.1 Sequent Implementation

The results for static scheduling is shown in Figure 2 which plots the completion time of the algorithm versus the number of processors. There is almost a linear improvement in performance as we increase the number of 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 8 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. Increasing the computation granularity increases the completion time as can be expected but the shape of the speedup curve remains the same for a given computation granularity. This trend further supports our earlier remark that the bus may not be a bottleneck for static scheduling. 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.

Figure 3 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 [7], wherein they show that lock contention leads to poor performance in bus-based shared memory multiprocessors. It is also the reason why the 8 processor performance is much poorer than the 4 or 2 processor case for extremely 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 4 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 perfor-

mance of the 8 processor case improves. 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 [12]. Note that this effect is less significant for 2 or 4 processors, which explains the better performance for 4 processors as compared to the 8 processor case for small data 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 list could be allocated in any of the processors' memories on the Butterfly. Thus data partitioning is another dimension that makes an interesting 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 2.3).

Figure 5 shows the performance for static scheduling for each of these data partitioning schemes. Local allocation seems to have the best performance of the three. 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. These results seem to indicate that the way in which we partition data has a significant effect on performance. Of the other two schemes, random allocation seems to show better performance than the skewed allocation. For both these schemes, the 2 processor performance seems to be worse than the 1 processor case. 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. Experiments performed with 1 processor with the data distributed among several memory modules results in much poorer performance compared to the case where the data is strictly in local memory. This result leads us to believe that increased latency is the more dominant factor affecting the performance. In the 2 processor case, the negative effects dominate. In general, the positive effect becomes more prominent for larger number of processors. But in the skewed allocation scheme, the performance worsens as we go from 8 to 16 processors. This is due to the increased likelihood of non-local memory accesses thus allowing the negative effects to dominate.

[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.

Figures 6 and 7 show the effect of increasing the computation granularity on the performance for random and skewed allocation schemes respectively. It can be seen that the poorer performance observed in going from 1 to 2 processors (see Figure 5) disappears as the computation granularity is increased. This result shows that increasing the computation granularity exploits the additional compute power available with increased number of processors, offsetting the negative effect due to the network traffic. The retarding factors that affect the performance in these two distributions do not have a significant role in the local allocation scheme (see Figure 5) even with no added computation granularity. As can be expected, increasing the computation granularity merely reinforces the positive effect.

The effect of varying the data granularity for dynamic scheduling is shown in Figure 8. Since in this scheduling strategy processors are assigned to tasks at run-time, the local allocation scheme may not have much meaning. Therefore only the random and skewed allocation schemes are considered. Since observed behavior for these two allocation schemes are similar, only the results for the random allocation scheme are presented in Figures 8 and 9. The performance at low chunk sizes is much worse than that observed in the static case (see Figure 8). As the chunk size is increased, the performance approaches that of static scheduling. This behavior is similar to the observed results for dynamic scheduling on the Sequent. An interesting observation that shows the effect of network traffic is the poorer performance for higher chunk sizes for 2 processors as compared to 1 processor. Further, the performance uniformly improves with increasing the number of processors showing no anomalous behavior as in static scheduling.

Figure 9 shows the effect of computation granularity on performance. We only present the results for low chunk sizes 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 chunk sizes) on the performance on the Sequent with larger number of processors (see Figure 3). On the other hand, this contention is not as pronounced on the Butterfly (see Figure 8). 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 [6], and Figure 10 gives 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. In

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

Figure 10: Parallel Prefix Algorithm

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). Experiments include data sizes of 4K, 16K and 32K elements in the array; and for the dynamic scheduling case, the data granularity ranges from 1 to 1K elements. In both cases, computation granularity is varied as in list ranking (see Section 4.1.1). Since the experimentation is over a range of data sizes, only an interesting selection of results is presented in the subsequent sections.

4.2.1 Sequent Implementation

Figure 11 shows the completion time as a function of the number of processors for a data size of 32K on the sequent. It is observed that an almost linear speedup is achieved. In fact it is interesting to note that results for static scheduling are remarkably similar to the corresponding results for list ranking (see Figure 2). This confirms our earlier observation regarding the negligible effect of synchronization cost and bus overhead on the performance of these algorithms.

Figures 12 and 13 present the results for dynamic scheduling on the Sequent. Figure 13 shows the effect of increasing computation granularity on the completion time for fixed low data granularity (chunk size = 1). It is seen that at low computation granularity the performance with 8 processors is poorer than with lesser number of processors. This loss in performance may be attributed to the overhead of accessing the global counter, i.e., as the number of processors increase, the contention on the lock outweighs the added compute power. There is another interesting similarity between the observed results of list ranking and parallel prefix. With 8 processors, the completion time decreases with increase in computation granularity, a result that seems counter-intuitive at first glance. However, the reason for this phenomenon is exactly similar to the list ranking case (see Section 4.1.1). Figure 12 shows the effect of varying the data granularity on the completion time as a function of the number of processors. It is observed that the performance with 8 processors is poorer than lesser number of processors for data granularity less than 256. With reference to Figure 12, it is also observed that as the data granularity is increased the performance with dynamic scheduling tends to equal that with static scheduling, but it never does better.

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. The experiments included data sizes of 8K and 16K, and Figures 14 through 18 present the results for data size of 8K. The results for data size of 16K are similar. Figure 14 shows the completion time versus the number of processors for the three data distributions in the case of static scheduling. It is seen that local data distribution scheme performs the best. The performance is almost linear with the number of processors. This result is to be expected, since in this problem each processor works on its local data for the most part. However, note that we do not see a linear speedup with the number of processors for the other two distributions. This result is primarily due to the latency for non-local accesses. Further, there could be some contribution due to switch contention. The performance for random distribution is better than that for skewed distribution, since in the latter case, every local access is always followed by a non-local one. It is also noted that as in list ranking, the 2 processor performance is worse than the 1 processor case, for these two distributions. With reference to Figures 15 and 16, a behavior similar to list ranking (see Figures 6 and 7) is observed when computation granularity is varied for random and skewed distributions.

The results for dynamic scheduling on the Butterfly are shown in Figures 18 and 17. Figure 18 shows the effect of varying the data granularity on the performance. The observations are very similar to the corresponding results obtained for list ranking (see Figure 8). Since the measured times for parallel prefix are quite small, at larger chunk sizes the trend shown by the shape of the curves is more important than the absolute numbers.

It is observed that the 16 processor case is consistently worse than the 8 processor case. This behavior can be attributed to the detrimental factors (non-local memory access and contention for the shared queue) that start to dominate with increased number of processors. With respect to Figure 17, it is seen that as the computation granularity is increased, the positive factor (increased compute power) starts to overshadow the detrimental effects. For example, with a chunk size of 1 and for computation granularity beyond 40, our experiments indicate that the 16 processor performance is better than the 8 processor performance.

4.3 Optimal Binary Search Tree

A standard dynamic programming algorithm for this problem computes a 2-dimensional cost matrix as shown in Figure 19 [1].

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

```
For i := 1 to n do
    For j := i+1 to n do
```

$$C_{i,j} \leftarrow \min_{i \le k < j} (C_{i,k} + C_{k+1,j}) + \sum_{k=i}^{j} w_k$$

Figure 19: Optimal Binary Search Tree Algorithm

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 20 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 21 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. However, with respect to Figure 20, it is observed that for a given chunk size, dynamic scheduling does result in speedup with increase in number of processors.

4.3.2 Butterfly Implementation

Figure 22 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 23 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 21), and is due to the work imbalance. However, there is a distinct difference between the two results as can be seen by comparing figures 21 and 23. In the case of the Butterfly, for a given chunk size, the performance worsens beyond a certain number of processors. This effect is more pronounced at higher chunk sizes (see the curve for chunk size = 16), since the added computing power with additional processors is offset by the detrimental factors.

5 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.

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. For example, in an 8 processor system with 4K data size and 1K chunk size, 4 processors would remain idle.

6 Conclusions

There were several hypotheses regarding the effects of architectural features on algorithmic performance that motivated 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 the bus is not a hindrance on the Sequent, the network latency on the Butterfly is a hindrance in achieving linear speedup.

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.

The last 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.

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.

References

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

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

[3] D.Chen, H.Su, P.Yew. The Impact of Synchronization and Granularity on Parallel Systems. *International Symposium on Computer Architecture.* 1990.

[4] R.M.Karp, V.Ramachandran. A Survey of Parallel Algorithms for Shared-Memory Machines. *Technical Report UCB-CSD-88-408.* UC Berkeley. March 1988.

[5] H.T.Kung. The Structure of Parallel Algorithms. *Advances In Computers.* Vol. 19, 1980.

[6] R.E.Ladner, M.J.Fischer. Parallel Prefix Computation. *Journal of Association of Computing Machinery.* Vol 27, No. 4, October 1980.

[7] J.Lee, U.Ramachandran. Synchronization with Multiprocessor Caches. *International Symposium on Computer Architecture.* 1990.

[8] C.Lin, L.Snyder. A Comparison of Programming Models for Shared Memory Multiprocessors. *International Conference on Parallel Processing.* 1990.

[9] S.T.Leutenegger, M.K.Vernon. The Performance of Multiprogrammed Multiprocessor Scheduling Algorithms. *Conference on Measurement and Modeling of Computer Systems.* 1990.

[10] R.J.Anderson. An Experimental Study of Parallel Merge Sort. *Preliminary Version, 0.1.* University of Washington. 1990.

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

[12] T.E.Anderson. The Performance of Spin Lock Alternatives for Shared-Memory Multiprocessors. *IEEE Transactions On Parallel And Distributed Systems.* January, 1990.

[13] A.Tucker, A.Gupta. Process Control and Scheduling Issues for Multiprogrammed Shared Memory Multiprocessors. *12th ACM Symposium on Operating System Principles.* 1989.

[14] J.C.Wyllie. The Complexity of Parallel Computations. *PhD Dissertation.* Computer Science Dept., Cornell Univ., Ithaca, NY,1981.

[15] J.Zahorjan, C.McCann. Processor Scheduling in Shared Memory Multiprocessors. *Conference on Measurement and Modeling of Computer Systems.* 1990.

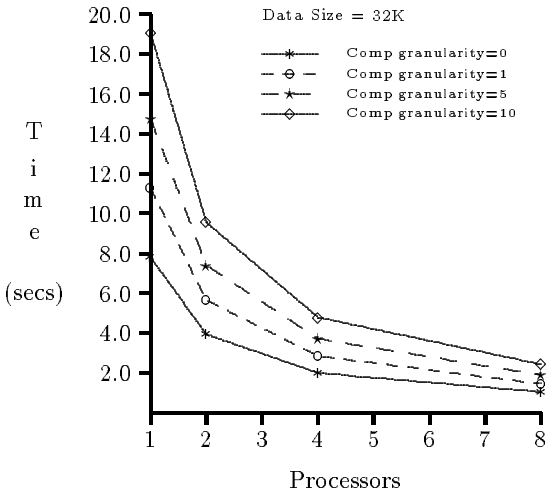

Figure 2: List Ranking – Static Scheduling On Sequent

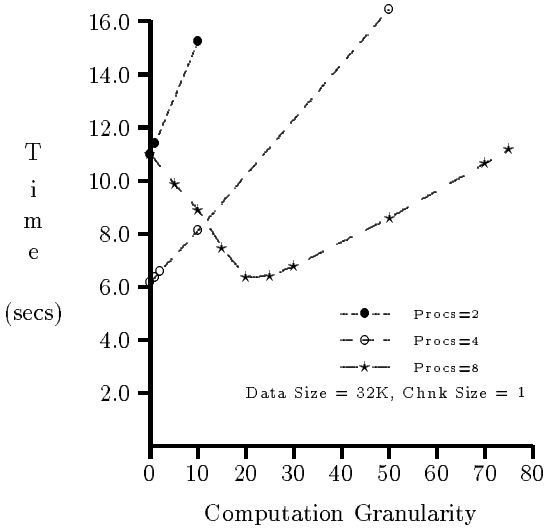

Figure 4: List Ranking – Dynamic Scheduling On Sequent (Computation Granularity)

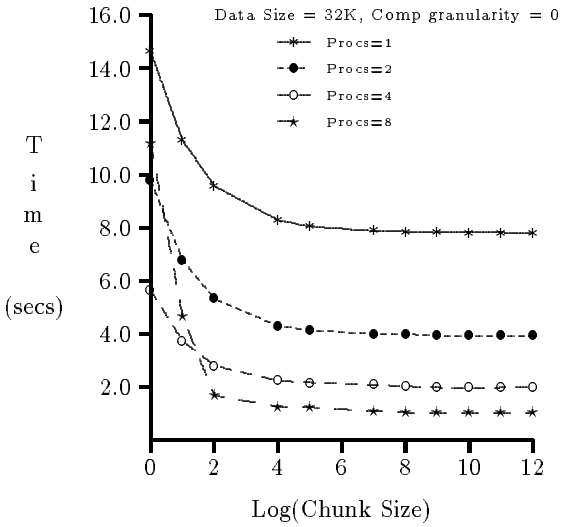

Figure 3: List Ranking – Dynamic Scheduling On Sequent (Data Granularity)

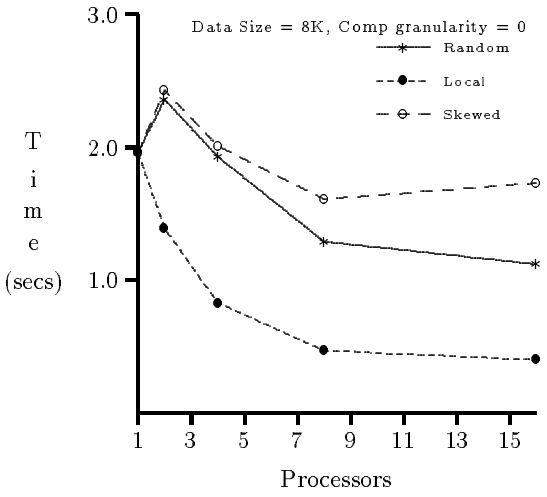

Figure 5: List Ranking – Static Scheduling On Butterfly (Data Distribution)

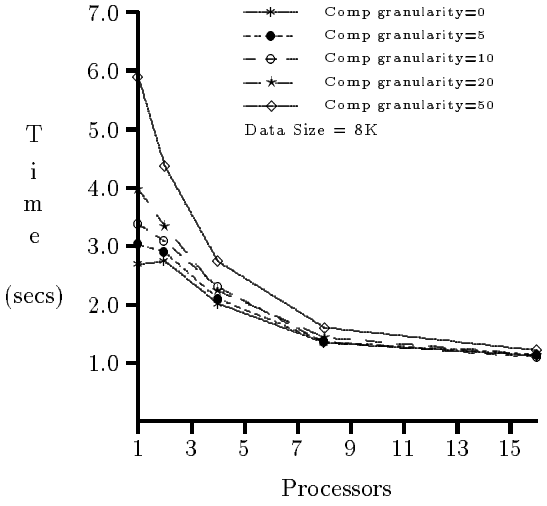

Figure 6: List Ranking – Static Scheduling On Butterfly (Random Distribution)

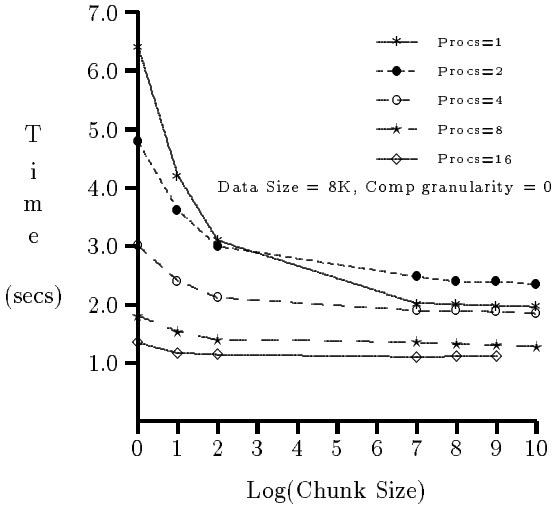

Figure 8: List Ranking – Dynamic Scheduling On Butterfly (Random Distribution, Data Granularity)

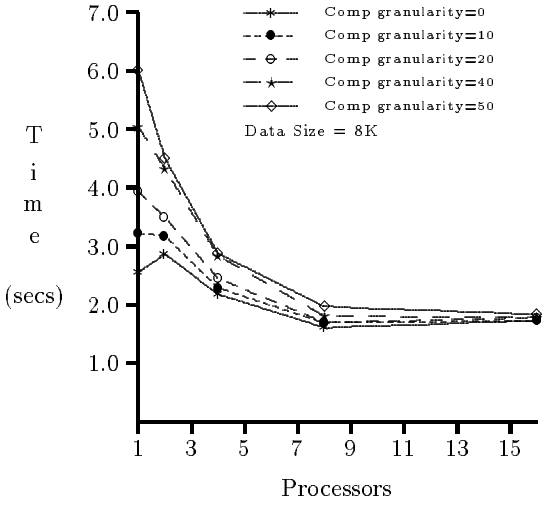

Figure 7: List Ranking – Static Scheduling On Butterfly (Skewed Distribution)

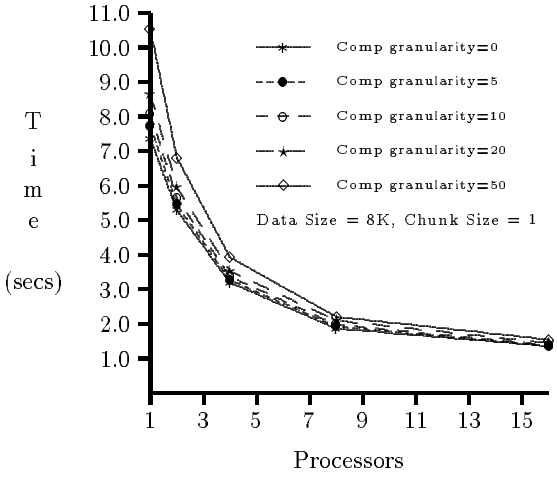

Figure 9: List Ranking – Dynamic Scheduling On Butterfly (Random Distribution, Computation Granularity)

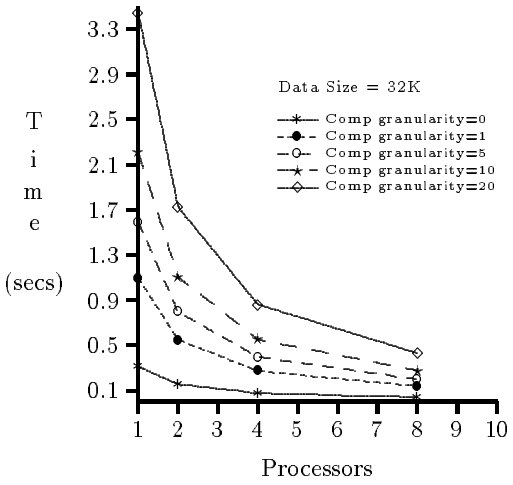

Figure 11: Parallel Prefix – Static Scheduling On Sequent

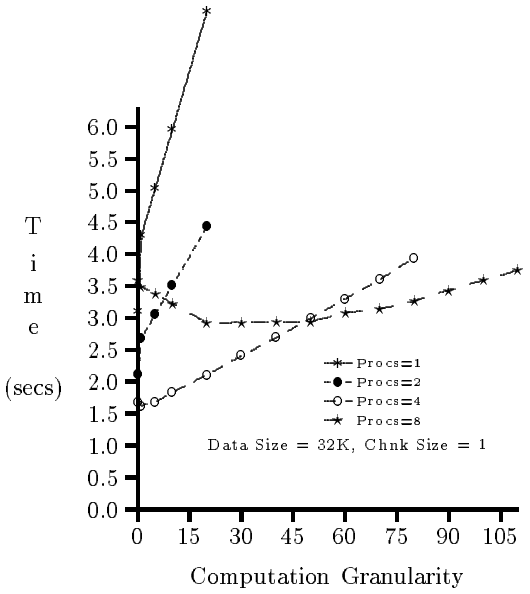

Figure 13: Parallel Prefix – Dynamic Scheduling On Sequent (Computation Granularity)

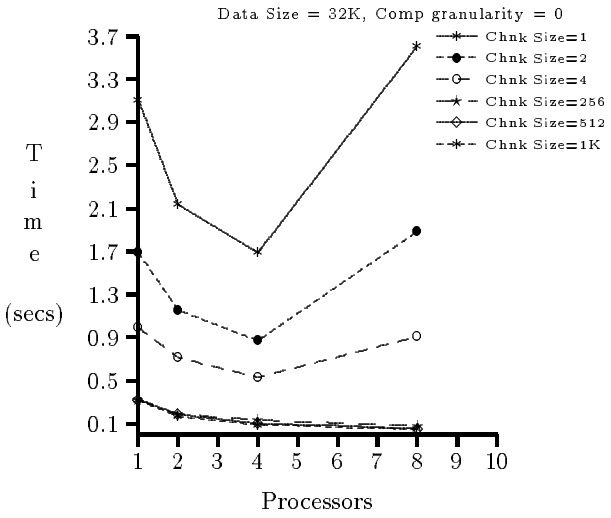

Figure 12: Parallel Prefix – Dynamic Scheduling On Sequent (Data Granularity)

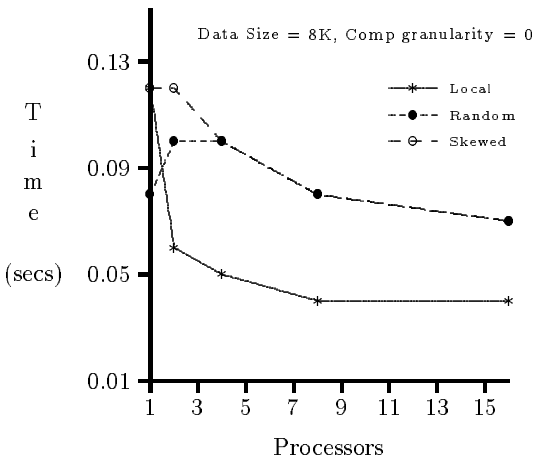

Figure 14: Parallel Prefix – Static Scheduling On Butterfly (Data Distribution)

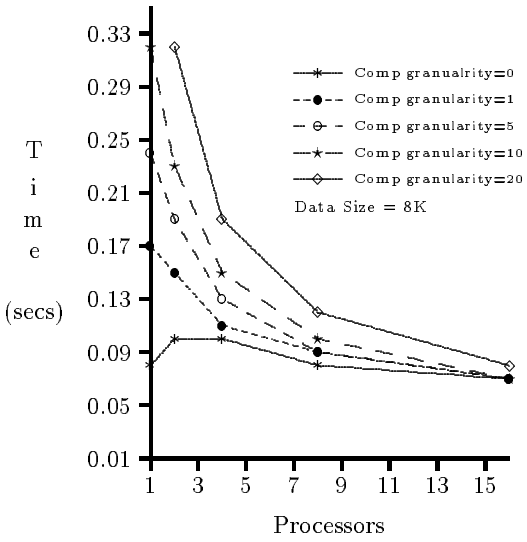

Figure 15: Parallel Prefix – Static Scheduling On Butterfly (Random Distribution)

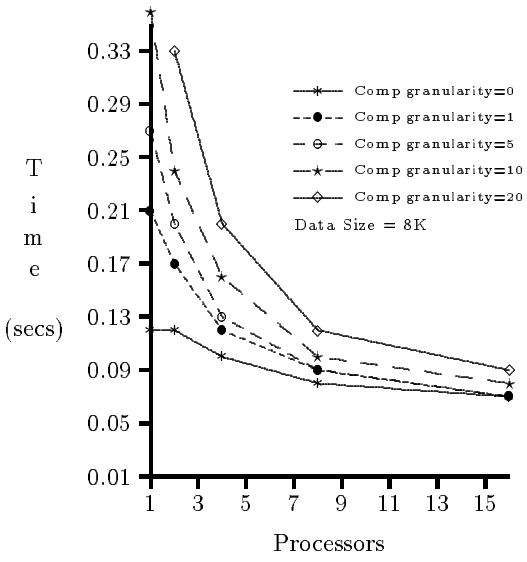

Figure 16: Parallel Prefix – Static Scheduling On Butterfly (Skewed Distribution)

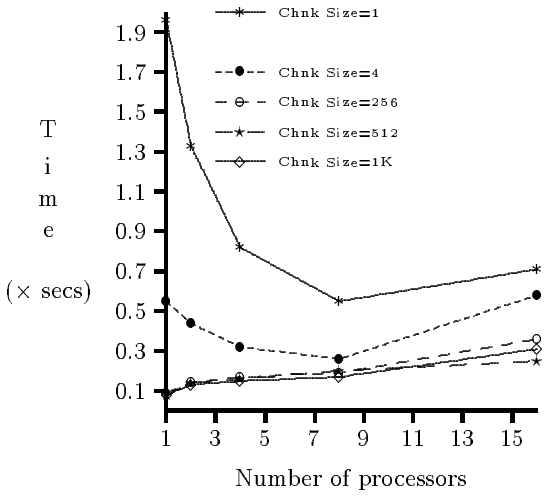

Figure 17: Dynamic Scheduling On Butterfly (Random Distribution, Data Size = 8K, No Artificial Work)

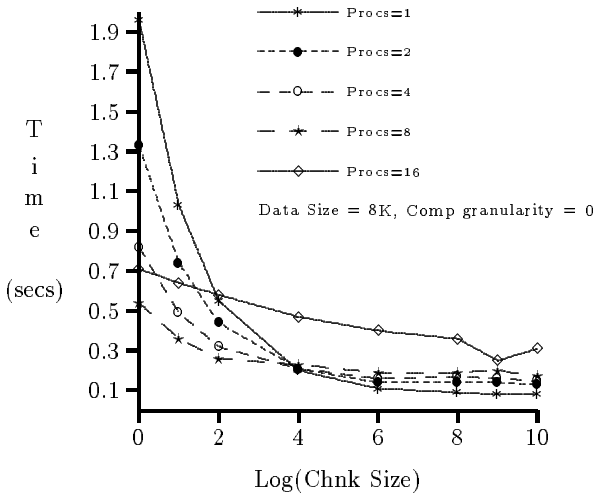

Figure 18: Parallel Prefix – Dynamic Scheduling On Butterfly (Random distribution, Data Granularity)

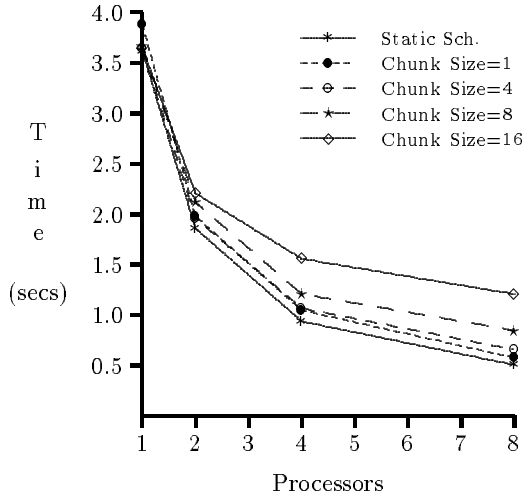

Figure 20: Optimal Binary Search Tree – Sequent Implementation

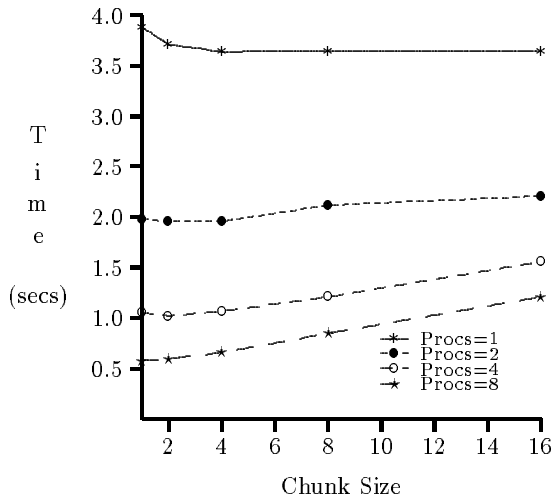

Figure 21: Optimal Binary Search Tree – Dynamic Scheduling On Sequent

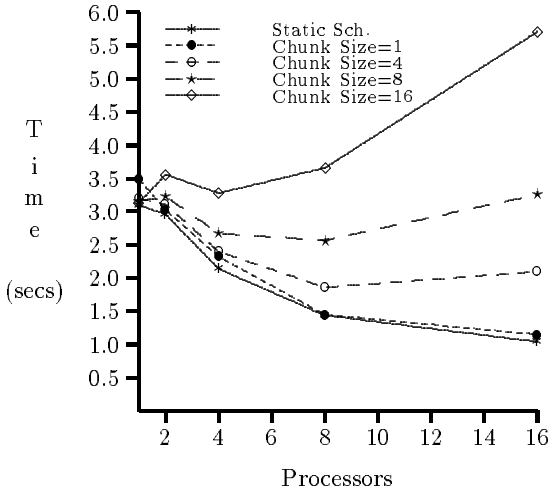

Figure 22: Optimal Binary Search Tree – Butterfly Implementation

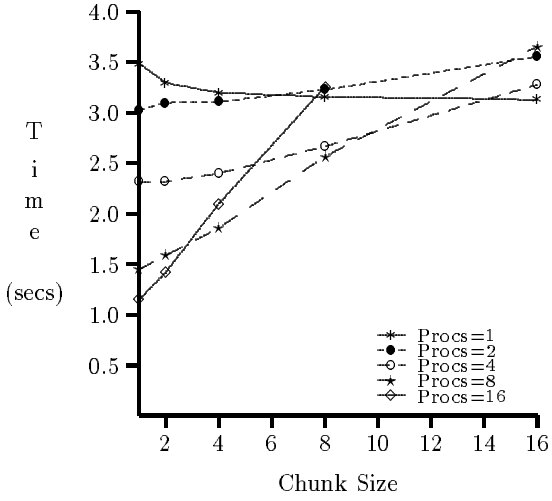

Figure 23: Optimal Binary Search Tree – Dynamic Scheduling On Butterfly