A Comparison of Parallel Sorting Algorithms on Different Architectures*

NANCY M. AMATO amato@cs.tamu.edu
RAVISHANKAR IYER ravi@cs.tamu.edu
SHARAD SUNDARESAN sharad@cs.tamu.edu
YAN WU yanwu@cs.tamu.edu

Department of Computer Science, Texas A&M University, College Station, TX 77843-3112

Technical Report 98-029
Department of Computer Science
Texas A&M University
January 1996

Abstract

In this paper, we present a comparative performance evaluation of three different parallel sorting algorithms: bitonic sort, sample sort, and parallel radix sort. In order to study the interaction between the algorithms and the architecture, we implemented all the algorithms on three different architectures: a MasPar MP1202, a mesh-connected computer with 2048 processing elements; an nCUBE 2, a message-passing hypercube with 32 processors; and a Sequent Balance, a distributed shared-memory machine with 10 processors. For each machine, we found that the choice of algorithm depends upon the number of elements to be sorted. In addition, as expected, our results show that the relative performance of the algorithms differed on the various machines. It is our hope that our results can be extrapolated to help select appropriate candidates for implementation on machines with architectures similar to those that we have studied. As evidence for this, our findings on the nCUBE 2, a 32 node hypercube, are in accordance with the results obtained by Blelloch *et al*. [5] on the CM-2, a hypercube with 1024 processors. In addition, preliminary results we have obtained on the SGI Power Challenge, a distributed shared-memory machine, are in accordance with our findings on the Sequent Balance.

*This research supported in part by NCSA.

1 Introduction

Sorting is one of the fundamental problems of computer science, and parallel algorithms for sorting have been studied since the beginning of parallel computing. Batcher's $\Theta(\log^2 n)$-depth bitonic sorting network [2] was one of the first methods proposed. Since then many different parallel sorting algorithms have been proposed (see, e.g., [4, 11, 17]), of which we mention only a few here. The first $\Theta(\log n)$-depth sorting circuit was discovered by Ajtai, Komlos, and Szemeredi [1]; this result is mainly of theoretical importance however, since the constants in their construction are quite large. Around that same time, Reif and Valiant proposed a more practical $O(\log n)$ time randomized algorithm called flashsort [16]. Some other notable parallel sorting algorithms are parallel versions of radix sort and quicksort [4, 17], column sort [10], and Cole's parallel merge sort [6].

Given the large number of parallel sorting algorithms and the wide variety of parallel architectures, it is a difficult task to select the best algorithm for a particular machine and problem instance. The main reason that the choice is more difficult than in sequential machines is because there is no known theoretical model that can be applied to accurately predict an algorithm's performance on different architectures. Thus, experimental studies take on an increased importance for the evaluation and selection of appropriate algorithms for multiprocessors. There have been a number of implementation studies reported in the literature in the past few years (see, e.g., [5, 12]). However, more studies are needed before we can approach the point where a certain algorithm can be recommended for a particular machine with any degree of confidence.

In this paper, we present an experimental study of three different parallel sorting algorithms: bitonic sort, sample sort, and parallel radix sort. Since parallel algorithms are generally only beneficial when the problem size is large enough, we concentrated on the case when n, the number of elements to be sorted, is much larger than p, the number of processors. In order to study the interaction between the algorithms and the architecture, we implemented all the algorithms on three different architectures: a MasPar MP1202, a mesh-connected computer with 2048 processing elements; an nCUBE 2, a message-passing hypercube with 32 processors; and a Sequent Balance, a distributed shared-memory machine with 10 processors. For each machine, we found that the choice of algorithm depends upon the number of elements to be sorted. In addition, as expected, our results show that the relative performance of the algorithms differed on the various machines. It is our hope that our results can be extrapolated to help select appropriate candidates for implementation on machines with architectures similar to those that we have studied. As evidence for this, our findings on the nCUBE 2, a 32 node hypercube, are in accordance with the results obtained by Blelloch *et al*. [5] on the CM-2, a hypercube with 1024 processors. In addition, preliminary results we have obtained on the SGI Power Challenge, a distributed shared-memory machine, are in accordance with our findings on the Sequent Balance.[1]

2 Machine Descriptions

The MasPar MP1202. The MasPar [13] is a Single Instruction Multiple Data (SIMD) machine. The system used in this study consists of a front-end DECstation 5000, a MP1202 Data Parallel Unit (DPU) with 2,048 processing elements (PEs), and a MasPar Parallel Disk Array (MPDA). The basic architecture is mesh-like, but an (inefficient) global router is available for general point-to-point communication. All parallel processing is performed by the DPU which consists of two major hardware components: the PE Array and the Array Control Unit (ACU). The PE Array performs the parallel computing while the ACU controls the interactions between the front-end and the PE Array. The DPU contains the mechanisms for PE to PE and PE to ACU communications. The MP1202 DPU has 2,048 PEs arranged in a 32×64 matrix. Each PE is a 1.8 MIPS 4-bit load/store arithmetic processor with 40 dedicated 32-bit registers and 16 Kbytes RAM. One PE can communicate with another using the X-Net construction or the Global Router. X-Net

[1] Experimental results on the SGI Power Challenge will be included in the final version of this paper.

communications allow a PE to communicate with its eight neighbors (north, northeast, east, southeast, south, southwest, west, and northwest), while the Global Router allows a PE to communicate with any other PE. The average communication cost using the X-Net ranges from 39 to 53 clock cycles, while the global router takes an average of 5,000 clock cycles. The parallel programming languages available on MasPar are the C-like MasPar Programming Language (MPL) and MasPar Fortran (MPFortran). A feature of the MasPar which we found very useful was the MasPar Programming Environment (MPPE), an integrated graphical environment for developing and debugging MasPar programs.

The nCUBE 2. The nCUBE [15] is a Multiple Instruction Multiple Data (MIMD) machine. The nCUBE used in this study uses a Sun workstation as its front-end and consists of 64 processors, each rated at 7.5 MIPS. The processors are connected in a hypercube interconnection pattern, and communicate with each other by message-passing. Processors 0-31 are configured with 1Mbyte of RAM, while the remaining 32 are configured with 4 bytes of RAM. Thus, to obtain a homogeneous system, only the first 32 processors were used in our experiments. Each nCUBE 2 processor consists of a general-purpose 64-bit central purpose unit (CPU), an error-correcting memory management unit (MMU), and a network communication port (NCU). The nCUBE 2 system provides communication primitives for point-to-point message passing and for broadcasting from one processor to a set of specified processors. Communication synchronization, error reporting, program exceptions and software facilities are handled using vectored interrupts. Programs on the nCUBE are written in an extended version of C, and can be executed on a sub-cube of a specified dimension.

The Sequent Balance. The Sequent Balance [18] is Multiple Instruction Multiple Data (MIMD) distributed shared-memory multiprocessor. It is a tightly-coupled shared-memory system in which all processors are connected to a common set of memory modules by a single high-speed bus. Hardware support is provided for synchronization (mutual exclusion, barrier synchronization, etc.), and the DYNIX operating system (a version of UNIX 4.2BSD) supports microtasking with the m_fork parallel programming library. The system used in our studies had ten identical general-purpose 32-bit microprocessors, each with 8 KB of cache RAM, and programs were written in a extended version of C.

3 Parallel Sorting Algorithms

In this section we give brief, machine-independent descriptions of the three parallel sorting algorithms selected for implementation. To make the analysis as general as possible, the complexity of each algorithm is reported in terms of *local computation* L, and *interprocessor communication* C. In order to facilitate comparison to previous work, we selected the same algorithms used in the study on the CM-2 by Blelloch *et al*. [5]; refer to their paper for background on the selection of these algorithms.

3.1 Bitonic Sort

Batcher's Bitonic sort [2] is a parallel sorting algorithm whose main operation is a technique for merging two bitonic sequences. A bitonic sequence is the concatenation of an ascending and a descending sequence of numbers. For example, $2, 4, 6, 8, 9, 24, 6, 3, 2, 0$ is a bitonic sequence. Various adaptations of Batcher's original algorithm have been proposed, e.g., [3, 14].

To sort a sequence of n numbers, the Batcher's algorithm proceeds as follows. The first step is to convert the n numbers into a bitonic sequence with $n/2$ numbers in an increasing subsequence and $n/2$ numbers in a decreasing subsequence. This is done recursively, as illustrated in Figure 1. After the bitonic sequence with n numbers is obtained, it is merged into an ordered sequence (either increasing or decreasing, depending upon which is needed). The merging technique consists of $\log n$ stages, and each stage consists of three operations: shuffle, compare, and unshuffle. Since this algorithm is well-known, we do not explain these operations in detail here; however, an example is shown in Figure 2. We next examine the complexity of the

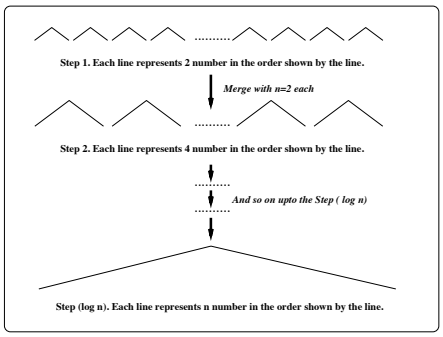

Figure 1: Recursive construction of bitonic sequence.

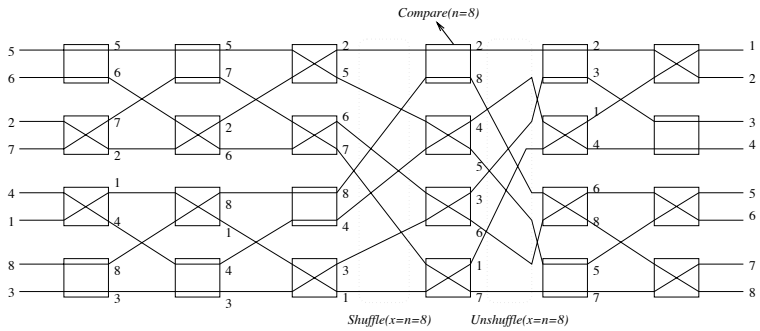

Figure 2: A bitonic sorter for $n = 8$.

algorithm. First, we consider Batcher's original algorithm, and then an adaptation which uses parallel radix sort to form the n element bitonic sequence from the n input elements.

Batcher's original bitonic sort. Each of the p processors works on $(\frac{n}{p})$ numbers in every stage. At each stage, the compare operation takes $O((\frac{n}{p})L)$ local computation and the shuffle and unshuffle operations have communication costs with other processors of $O((\frac{n}{p})C)$. Therefore the time complexity is a factor of the number of stages and the term $O((\frac{n}{p})L + (\frac{n}{p})C)$. Given n numbers, we have $\log n$ phases. Phase 1 consists of 1 stage, phase 2 consists of 2 stages, and so on. Thus, the total number of stages is $O(\log^2 n)$, and the overall time complexity of the Batcher's bitonic sort is

$$T_{bs_1} = O(((\frac{n}{p})L + (\frac{n}{p})C)\log^2 n). \tag{1}$$

Bitonic sorting with radix sort. To convert n input elements into a bitonic sequence with $n/2$ elements in the increasing and decreasing sequences we use parallel radix sort. This takes $T_{rs}(\frac{n}{2})$ (by Equation 5). We then use Batcher's Bitonic Merge to convert n element bitonic sequence into an increasing sequence of n numbers. This is done in $\log n$ stages and takes time $O(((\frac{n}{p})L + (\frac{n}{p})C)logn)$. Thus the total time complexity for this modification of bitonic sort is

$$T_{bs_2} = O(T_{rs}(\frac{n}{2}) + (\frac{n}{p})L + (\frac{n}{p})C)\log n \tag{2}$$

3.2 Sample Sort

There have been many sorting algorithms proposed that use a random sample of the input elements to partition the input into distinct subproblems that can be sorted independently (see, e.g., [8, 9, 16, 19]). The basic version we implemented consists of the following three phases. First, a random sample of $p-1$ *splitter* elements $S = \{s_1, s_2, \ldots, s_{p-1}\}$ is selected from the n input elements. Once sorted, these splitter elements define a set of p *buckets*, e.g., bucket i covers the range $[p_i, p_{i+1})$. Second, each input element is placed in (sent to) the appropriate bucket, and finally, the elements in each bucket are sorted. Typically, one bucket is created for each processor, and the sorting of the elements in each bucket is performed in parallel. The random selection of the splitter elements is performed in parallel; if the size of the random sample is large enough, then it may be beneficial to perform the sort of the splitter elements in parallel as well. Note that the running time of this algorithm does not depend on the distribution of the keys since the splitters are selected from the input elements.

The running time of this algorithm is dependent on the maximum bucket size, i.e., the maximum number of elements contained in any bucket. Ideally, one would like all buckets to contain an equal number of

elements. Oversampling is a technique that has been used to try to reduce the deviation in bucket size: in Step 1, one selects a set of ps input elements, sorts them, and uses the elements with ranks $s, 2s, \ldots, (p-1)s$ as the splitter elements. The integer parameter s is called the *oversampling ratio*. Clearly, the choice of s is crucial to the expected running time of the algorithm. In particular, s should be chosen large enough to achieve a small deviation in bucket sizes and small enough so that the time required to sort the sample in Step 1 does not become too large; the ratio of the largest bucket size to the average bucket size, n/p, is called the *bucket expansion*. Thus, the appropriate choice for s depends on both the value of n and the number of processors in the system.

The sample sort algorithm is summarized below.

SAMPLESORT(A)

Step 1 *Selecting the splitters*. Each of the p processors randomly selects a set of s elements from its $\lceil n/p \rceil$ elements of A. These ps keys are sorted by parallel radix sort, and then the $p-1$ splitter elements are selected. Duplicates can be distinguished by tagging the elements before sorting.

Step 2 *Distributing the keys to the buckets*. Each processor determines the bucket to which each of its $\lceil n/p \rceil$ elements of A belongs by performing a binary search of the array of splitters (which may be stored separately on each processor). The keys are then distributed to their respective buckets.

Step 3 *Sorting the keys in each bucket*. Each processor sorts the keys locally within its bucket using a sequential radix sort.

Assuming that the input elements are initially distributed to the processors, Step 1 uses $O(s)$ time in local computation to select the ps elements, $T_{rs}(ps)$ time to sort them (by Equation 5), and $O(p)$ local computation time and communication to select and collect the splitter elements. In Step 2, the local computation costs are $O((n/p)\log p)$ to determine the buckets for all elements and $O(n/p)$ in communication to place the elements in the buckets. Step 3 uses expected time $T_{srs}(n/p)$ (by Equation 4) in local computation and $O(n/p)$ time in communication. Therefore, the expected time of the parallel sample sort is

$$T_{ss}(n) = O(T_{rs}(ps) + T_{srs}(n/p) + (p + n/p)C + (p + (n/p)\log p)L). \tag{3}$$

3.3 Parallel Radix Sort

Radix sort [7] is a relatively simple algorithm that is quite easy to parallelize. Since it is not a comparison sort (i.e., it is not based only on comparisons), radix sort is not subject to the $\Omega(n \log n)$ lower bound for comparison-based sorting. We first describe sequential radix sort, and then discuss how it can be parallelized.

Suppose each element is represented by b bits. In its simplest version, radix sort sorts the elements b times. Each sorting pass considers only one bit of each element, with the ith pass sorting according to the ith least significant bit. The number of passes can be reduced from b to $\lceil b/r \rceil$ by considering blocks of r bits instead of a single bit in each pass. The sorting algorithm used by radix sort must be *stable*, i.e., the relative order of two items with the same key must be the same in both the input and output orderings. *Counting sort* [7] is often used as the stable sorting algorithm in radix sort. In counting sort, the input elements must be integers in a fixed range; for radix sort the elements are integers in $[0, 2^r)$. Counting sort finds the *rank* of each input element, i.e., the number of elements smaller than that element. First, we count the number of elements of each value present in A; these counts are stored in an auxiliary count array $R[0..2^r - 1]$. We next compute the prefix sums of the counts in R; the prefix sum in $R[i-1]$ identifies the first position in the output array occupied by an element with value i. Finally, the elements are copied to their positions in the

output array B. This can be done by scanning A and placing elements in the position in B indicated by $R[]$, and then updating $R[]$ to show where the next element with this value should be placed.

The sequential complexity of radix sort on n elements is $T_{srs}(n) = O((b/r)f(n,r))$, where $f(n,r)$ is the time required by the intermediate stable sort when considering only r bits of the input elements. Since a standard implementation of counting sort gives $f(n,r) = O(n+2^r)$ [7], the complexity of sequential radix sort is

$$T_{srs}(n) = O((b/r)(n+2^r)). \tag{4}$$

The simplest way to parallelize radix sort is to use a parallel version of counting sort (also done in [5]). Let p be the number of processors available. In the parallel counting sort, each processor is responsible for $\lceil n/p \rceil$ input elements. First, each processor counts the number of its elements with each possible value (in $[0, 2^r)$), and then computes the prefix sums of these counts. Next, the processors' prefix sums are 'combined' by computing the prefix sums of the processor-wise prefix sums. Finally, each processor places its elements in the output array. More details are given in the pseudo-code below.

PARALLELCOUNTINGSORT(A, i, r) /* consider ith block of r bits */

Step 1 In parallel, each processor counts the number of its (approximately) $\lceil n/p \rceil$ assigned elements with value v, for $0 \leq v \leq 2^r - 1$. Processor i places the counts in count array $R_i[0..2^r - 1]$, and then computes the prefix sums for R_i, for $0 \leq i \leq p - 1$.

Step 2 Processors cooperatively compute prefix sums of the processor-wise prefix sums.

(a) Each processor is assigned (approximately) $\lceil 2^r/p \rceil$ values of $[0, 2^r)$ and computes the prefix sums of all the count arrays for those values. For example, the processor assigned value v will set $R_i[v] = \sum_{j=0}^{i} R_j[v]$, for each $0 \leq i \leq p - 1$.

(b) Each processor now adds an offset to each element in its count array. For example, processor i adds $R_{p-1}[v]$ to $R_i[v+1]$, for each $v \in [0, 2^r)$.

Step 3 In parallel, the processors copy their assigned elements to the output array B. This is done just like sequential counting sort, except processor i uses its count array R_i.

Assuming that the input elements are initially distributed among the processors, Step 1 uses $O(n/p+2^r)$ time in local computation and requires no interprocessor communication. Step 2 requires $O(2^r)$ time in local computation and $O(2^r)$ remote data accesses for each processor. Step 3 takes $O(n/p)$ time in local computation and perhaps as many as $O(n/p)$ remote accesses for each processor. Thus, the complexity of parallel radix sort using this version of parallel counting sort is

$$T_{rs}(n) = O((b/r)((n/p+2^r)C + (n/p+2^r)L)). \tag{5}$$

From the expression above we see that the parameter r should be chosen so that $2^r = \Theta(n/p)$, e.g., $r = \log(n/p)$. In this way we choose r as large as possible (minimizing b/r, the number of counting sorts performed), but still small enough so that the factor of 2^r does not overtake the factor of n/p.

4 Experimental Results

In this section we discuss implementation details and present our experimental results. For each machine/algorithm pair, we first describe any machine-specific modifications made to the algorithms and then discuss the results. In order to make our results as generalizable as possible to machines with similar architectures, all code was written in a high level language (a variant C) and we did not employ any low level

machine-specific optimizations. In all cases, the keys to be sorted were randomly generated 32 bit integers. All experiments were repeated 25 times in single-user mode, and the results reported are averaged over the 25 runs.

4.1 The MasPar MP1202

Bitonic sort. Our implementation of bitonic sort on the MasPar was based on the mesh sort of Nassimi and Sahni [14], which is an adaptation of bitonic sort suitable for mesh-connected machines. The mesh sort of Nassimi and Sahni could be implemented using only X-Net communication (to the 8 local neighbors). Recall that on the MasPar, communication using the X-net is much faster than the global router. Radix sort with $r = 4$ was the local sort used in the bitonic merge.

The results obtained for bitonic sort on the MasPar are shown in Figures 3 and 4. Figure 3 shows that good speed-ups were obtained for all input sizes. The breakdown in the execution time for each phase of the algorithm is shown in Figure 4 for $n = 64,000$; the execution time of each phase is shown as the addition to the phases shown below it in the graph. We can see that the bitonic merge takes much more time than the sequential sort. Since the main operation of bitonic merge is PE communication, we conclude that the performance of the algorithm is mainly dependent on the bandwidth of X-net connection.

Sample sort. The sample sort we implemented on MasPar is very similar to the algorithm described in Section 3.2. The splitter candidates were picked randomly by each PE and were sorted by sequential radix sort inside each PE. Figures 5 and 6 show how the oversampling ratio affects the execution time and the bucket expansion bound. Although the graph shows the bucket expansion always decreases when the oversampling ratio increases, the total execution time does not always decrease. In particular, after some point, between 16 and 32, the benefit from better load balancing is overshadowed by the cost of sorting a larger set of splitter candidates and the total time begins to increase. In our experiments we used an oversampling ratio of 32; note that as long as the oversampling ratio is larger than 16, the average bucket expansion bound is less than 2.

The results obtained for sample sort on the MasPar are shown in Figures 7 and 8. Figure 7 shows speedups for various values of n and p. Note that better speedups are obtained for larger values of n. This is because the overhead of selecting the splitters becomes more prominent for small values of n. Figure 8 shows the breakdown in execution time for the phases of the algorithm. The graph shows that distributing the elements to the buckets (PEs) dominates the execution time; this phase is accomplished using the very inefficient global router. Thus, again, interprocessor communication is the bottle neck.

Parallel radix sort. The radix sort implemented on MasPar is the same as the algorithm described in Section 3.3. The global router was used to implement the parallel counting sort. The results obtained for parallel radix sort on the MasPar are shown in Figures 9 through 11. Figure 9 shows the execution time for different values of n/p and r. As noted in Section 3.3, the graph shows that the optimal value of r depends on both n and p; the larger n/p the larger the optimal value of r. We used $r = 8$ in our experiments. Figure 10 shows that similar speedups were obtained for all values of n. There exists a step after $p = 256$, which is likely caused by additional communication costs needed by the increased number of PEs. Figure 11 shows the breakdown in the execution time for the phases of the algorithm. The local sort includes the local computation and the interprocessor scan for the counting sort and the communication is the time needed to move the elements among the PEs at the conclusion of each counting sort phase. Clearly, the communication costs dominate the execution time. It is interesting to note that the local sort component remains almost constant as the number of PEs increases, implying that the scan operation used in the counting sort is not sensitive to n/p.

Comparison of the three algorithms on the MasPar MP1202. The comparison of three algorithms on MasPar is shown in Figures 12 and 13. Figure 12 shows the speedups for algorithms for $n = 64,000$. Radix sort is always the fastest, sample sort is the next for $p \leq 512$, but is passed by bitonic sort for $p \geq 1024$. In

terms of speedup, bitonic sort is the best, followed by radix sort, and then sample sort. The results shown in Figure 13 for $p = 2048$ (the maximum), are different from those shown in Figure 12. In particular, bitonic sort is the fastest for $n < 64000$, and then it is passed by radix sort. Initially, sample sort is the slowest, but it passes bitonic sort to claim second place for $n \geq 512,000$.

As noted above, the communication costs of the algorithms are the dominant components of the execution time. Thus, we would expect sample sort to have the best performance. We believe that our result did not show this because we have a limitation on n/p, the number of elements allocated to each PE. In particular, since our MP1202 has only 16KB local memory for each PE, n/p is limited to around 1024. Thus, given more local memory and n large enough, we would expect sample sort to out perform the other algorithms.

4.2 The nCUBE 2

Since the nCUBE is a message passing computer, interprocessor communication and data movement are very expensive operations. Thus, whenever possible, we modified the algorithms to reduce the number of messages passed. In particular, rather than sending multiple separate messages, we collected all messages to be sent to a particular destination processor and sent them all in one message.

Bitonic sort. Subject to the modification mentioned above, the bitonic sort implemented was similar to the algorithm described in Section 3.1. We implemented both the original bitonic sort, and the version where the sequence was first transformed into a bitonic sequence using radix sort, and then a bitonic merge was performed. The results shown in the figures are from the latter version which always was superior to the original bitonic sort.

The results obtained for bitonic sort on the nCUBE are shown in Figures 14 and 15. Figure 14 shows that the behavior of the execution time versus the number of processors is similar for different values of n. In addition, we see that the communication time is more dependent on the number of elements processed than it is on the number of processors used, i.e., as the number of processors increases the communication time increases by a factor of two and the execution time decreases by a factor that is less than two. Figure 15 shows the breakdown in execution time for each phase of the algorithm; the execution time of each phase is shown as the addition to the phases shown below it in the graph. The graph shows the difference between communication cost (shuffle and unshuffle) and computation cost (sequential merge). For example, with 2 processors the communication time is very low as compared to the execution time. Conversely, for 32 processors the computation time is very low when compared to the communication time because the number of elements at each processor is greatly reduced. Similarly, it can be seen that the execution time for the shuffle procedure decreases as time increases since the number of elements decreases as the number of processors increases.

Sample sort. The implementation of sample sort on the nCUBE involves some deviations from the algorithm described in Section 3.2 to reduce interprocessor communication costs. Namely, using an oversampling ratio $s = 10$, each processor randomly selected s of its n/p keys and then sent them to a single processor which sorted them, selected the splitter elements, and then broadcast the $p-1$ splitters to all other processors. This choice of s was judged sufficient since it yielded bucket expansion bounds of less than 2. As with bitonic sort, communication overhead was reduced by grouping the keys according to destination processor (bucket) and sending all keys in a single message.

The results obtained on the nCUBE 2 for sample sort are shown in Figures 16 and 17. Figure 16 shows that, like bitonic sort, the execution time of sample sort depends on the number of elements as well as on the number of processors involved in the sorting. In particular, the execution time increases by a factor of approximately 2.1 to 1.33 as the number of elements is increased by a factor of 2 and the number of processors is held constant. As the number of processors is varied, we note that although good speedups are obtained for the range of 2 to 8 processors, the execution time does not continue to decrease

at the same rate for larger numbers of processors. This is because of increased communication costs due to longer interprocessor distances. Fig. 17 shows the breakdown in execution time for each phase in the algorithm. The selection of the sample keys is not seen on the graph because its time is very negligible when compared to the other phases. The high execution time for 2 processors shows that the computation time (the local sort) is higher than the communication time (the distribution phases). As expected, when the number of processors is increased, the computation costs decrease and the communication costs increase. The abnormal change between 4 and 8 processors for the local sort indicates that the oversampling ratio may not have been large enough to obtain good load balancing for 8 processors.

Parallel radix sort. The implementation of parallel radix sort on the nCUBE differs somewhat from the algorithm described in Section 3.3, and in particular, in the internal parallel counting sort. The number of messages to be sent and received was reduced by broadcasting the processors individual count arrays and the final count array. Although each processor does not need all this information, the overall communication costs were reduced by sending fewer larger messages rather than many small messages. After the final count array is received, each processor determines the destination processor for each of its n/p elements, and then sends a single message to each destination processor.

The results for the parallel radix sort algorithm on the nCUBE are shown in Figures 18 through 20. The curves shown in Figure 18 show the execution time as r, the number of bits per pass, is increased from 1 to 8. As noted in Section 3.3, the graph shows that the optimal value of r depends on both n and p. However, we see that a value of 4 or 5 is suitable in many cases; our experiments used a fixed value of $r = 4$. Fig. 19 shows that the execution time varies by a factor of 1.95 to 1.1 as the number of elements is increased by a factor of 2 and the number of processors is kept constant. It can also be seen that the execution time increases for low values of n as the number of processors increases beyond 8 processors; this is due to the large amount of communication and the fact, as seen in sample sort, that communication costs in the nCUBE increase beyond 8 processors due to larger ranges. Since parallel radix consists of several iterations of parallel counting sort, each of which requires 3 phases of communication, it is difficult to draw conclusions from the total execution time of the algorithm. Figure 20 shows the execution time for the different phases of the algorithm. Note that in all cases the most time is taken in the final phase of the algorithm. In particular, when the number of processors is high the actual communication time is high, and when the number of processors is low the communication time is high due to the large number of messages sent (since each processor has many elements). The least amount of time is spent calculating the offsets since this is just local computation and no communication is involved. The time taken for broadcasting the initial count array is also high because it involves both computation and communication. Finally, broadcasting the elements costs a small amount of time which increases as the number of processors increases, showing that the communication time is dependent on the number of processors.

Comparison of the three algorithms on the nCUBE 2. Above, the communication cost of the nCUBE was seen to have a very high impact on performance. The comparative performance of the nCUBE implementations of the three algorithms is shown in Figures 21 and 22.

Fig. 21 shows the execution time for the three algorithms for different values of p for $n = 32768$; in all cases, sample sort is the fastest, followed by parallel radix sort, and then bitonic sort. The relative performance of the algorithms can be explained by their communication patterns. First, parallel radix sort requires more communication than sample sort since it has 3 communication phases in each iteration of the parallel counting sort. Thus in a message passing multiprocessor such as the nCUBE, parallel radix sort can be expected to perform worse than sample sort. Finally, the bitonic sorting algorithm has the highest execution time for this value of n since its communication is very high. Figure 22, where $p = 32$, shows a similar performance between the three algorithms except that bitonic sort performs better than parallel radix sort for $n < 8192$. For smaller values of n and $p = 32$, parallel radix sort has higher communication costs since communication can be over the entire range of 32 processors but in bitonic sort the messages do not interfere too much. In particular, in bitonic sort the communication is strictly defined from one processor to

another while in parallel radix sort a processor may be sending values to all 32 processors or to just to one processor. However, as the number of elements increases, parallel radix sort improves in relative time since the uniformity improves, while bitonic sort becomes less competitive since the number of stages increases along with the point-to-point communication costs.

4.3 The Sequent Balance

Since the Sequent Balance is a shared-memory machine, interprocessor communication is reduced to reading from and writing to shared memory locations. Moreover, since Sequent processors do not have local memory (only caches), there is really no difference between a 'local' computation in which a processor works on its 'own' data and a shared memory access. An important fact to note is that the Sequent was the easiest machine to program of the three, and that very few, if any, changes were made to the algorithms as presented in Section 3. The initial cost of setting up parallel tasks is very expensive on the Sequent Balance, and moreover, as shown in Figure 23, this start-up time increases linearly with the number of tasks (processors) initiated. Since in many cases these start-up costs were larger than the rest of the computation of the algorithm, we omitted them from the timings shown below. This was necessary so that we could compare the algorithms, and is reasonable since in a machine like the Sequent one would want to initiate all tasks at the beginning of a program and reuse them as needed (as we did in our implementations).

Bitonic sort. Since bitonic sort is optimized when using 2^k processors, we present results for only $2, 4$, and 8 processors. We implemented both the original bitonic sort, and the version where the sequence was first transformed into a bitonic sequence using parallel radix sort, and then a bitonic merge was performed. The results shown in the figures are from the latter version which always was superior to the original bitonic sort. During the shuffle and unshuffle procedures, the implementation had to be careful to prevent a processor from writing a key to a new index before the original key in that index was accessed. This was done using two versions of the array, and copying between the versions at the end of each phase.

The results for bitonic sort on the Sequent Balance are shown in Figures 24 and 25. Figure 24 shows that the execution time required increases proportionately to n, and also that there is a direct relationship between the number of elements sorted and the processors used. From observing the graph, it is apparent that the curves are very close to the behavior predicted in Section 3.1 (with $C = 0$), and that the algorithm appears to achieve scalable speedups. In particular, as n increases by a factor of 2, so does the running time; similarly, as p increases by a factor of 2, the time decreases by this amount as well. Figure 25 shows the breakdown of the execution time for each phase of the algorithm. Note that most of the time is spent in the shuffle and unshuffle phases.

Sample sort. The implementation of sample sort followed the algorithm in Section 3.2. In shared memory system such as the Sequent, it is convenient to store all buckets consecutively in one shared array; in this case the offsets into this array were computed by counting the number of each processor's elements in each bucket and then performing a prefix sums computation. The results for sample sort on the Sequent Balance are shown in Figures 26 and 27. In our experiments we used an oversampling ratio of $p = 10$, which was seen yielded bucket expansion bounds of less than 2. In Figure 26 we see that the performance is close to the theoretically predicted values, and the implementation achieves good speedups for a wide range input sizes. Figure 27 shows the breakdown in the execution time among the phases of the algorithm. The majority of the time is spent in the local sorting of the buckets, and as expected, this decreases as the number of processors increases. The selection and sorting of the sample requires very little time, and as expected, increases slightly as the number of processors is increased. The time spent distributing the keys to the buckets remains almost constant and does not decrease as much as we would have expected over the range of processors. This difference may be due to contention at the memory modules which increases with the number of processors.

Parallel radix sort. The implementation of parallel radix sort followed very closely the algorithm in Sec-

tion 3.3. A difference from the MasPar and nCUBE implementations is that it was not necessary to move the data structures from one processor to another during the parallel counting sort.

The results for radix sort on the Sequent Balance are shown in Figures 28 and 29. As expected, Figure 28 shows that the optimal value of r depends on both n and p; for simplicity, we used $r = 6$ for our experiments. In Figure 29 we see that the performance is close to the theoretically predicted values, and the implementation achieves good speedups for a wide range input sizes. Recall that we used a fixed value of r; better results for larger numbers of processors or data sizes would likely have been obtained with a larger value of r. The breakdown in execution time for the different phases of the algorithm is not very interesting for the Sequent since, unlike the MasPar and nCUBE implementations, all phases require similar types of operations because local computation and communication are not distinguished.

Comparison of the three algorithms on the Sequent Balance. The comparative performance of the Sequent Balance implementations of the three algorithms is shown in Figures 30 and 31.

Fig. 30 shows the execution time for the three algorithms for different values of p for $n = 2048$; in all cases, parallel radix sort is the fastest, followed by sample sort, and then bitonic sort. Figure 31, where $p = 8$, shows a similar performance between the three algorithms except that sample sort performs better than parallel radix sort for $n < 1024$. The probable reason that bitonic sort is slower than the others is that it requires more data movement, i.e., it has many phases, and in each phase elements are copied between arrays. It is less clear why radix sort beats sample sort. In particular, sample sort has few data movements (only one), and thus we might conjecture that it would be faster. Since the majority of the time for sample sort is spent sorting the buckets, it is possible that imperfect load balancing is the cause. However, in all cases the bucket expansion ratio was less than 2 which is very good, so it does not seem likely that this problem can be corrected simply by increasing the oversampling ratio. Even so, the graph shows that both parallel radix sort and sample sort produce good speedups and that they are fairly competitive with each other.

5 Comparisons across Machines and Recommendations

In this section, we summarize our findings and attempt to draw some broader conclusions about which algorithms are good candidates for certain classes of architectures. Of course, more studies will be needed to further substantiate the algorithm/architecture recommendations we make below.

On the MasPar MP1202, essentially a mesh-connected computer, we found that the version of bitonic sort due to Nassimi and Sahni [14] was the fastest for smaller input sizes and that parallel radix sort was the fastest for larger input sizes. As mentioned above, we would expect sample sort to become more competitive with parallel radix sort for larger values of n if the processing elements had larger memories (our MasPar had only 16KB of memory for each processing element). However, in terms of recommending a single sorting algorithm for a mesh-connected computer, our results indicate that parallel radix sort might be a good choice since its performance was consistently good when both n and p were varied.

On the nCUBE 2, a hypercube multiprocessor, we found that sample sort was the algorithm of choice. Our results are in agreement with those found by Blelloch *et al.* for larger values of n on the CM-2, a hypercube with 1024 processors [5]. The advantage of sample sort over the other algorithms is that it minimized data movement and interprocessor communication, which is an important consideration for message passing computers. Thus, we believe that sample sort is an attractive choice for message-passing machines, and in general, for multiprocessors and distributed systems in which interprocessor communication and data movement are expensive operations.

On the Sequent Balance, a distribed shared-memory machine, we found that sample sort was the fastest for smaller values of n and parallel radix sort was the fastest for larger problem sizes. As with the MasPar, however, parallel radix sort was very competitive even for smaller values of n. The disadvantage of sample sort is that the size of subproblems is determined by a random sample, and thus there will be some degree

of load imbalance. Thus, the penalty incurred by this imbalance, even if it is very small, can overcome the benefit of reducing the amount of interprocessor communication and data movement which is fairly cheap on a shared-memory machine such as the Sequent Balance. Therefore, we believe that parallel radix sort is an attractive choice for shared-memory machines. As further evidence for this conjecture, our preliminary results on the SGI Power Challenge, a machine with an architecture similar to the Sequent Balance, are in accordance with this recommendation.

Acknowledgment

This work began as a course project in CPCS 629 at Texas A&M University. We would like to thank Thomas Pham for coding the original version of sample sort on the nCube as part of his course project.

References

[1] M. Ajtai, J. Komlos, and E. Szemeredi. Sorting in $c \log n$ parallel steps. *Combinatorica*, 3:1–19, 1983.

[2] K. E. Batcher. Sorting networks and their applications. In *AFIPS Springer Joing Computer Conference*, pages 307–314, Arlington,VA, April 1968.

[3] G. Bilardi and A. Nicolau. Adaptive bitonic sorting: An optimal parallel algorithm for shared-memory machines. *SIAM J. Comput.*, 18(2):216–228, April 1989.

[4] G. E. Blelloch. *Vector Models for Data-Parallel Computing*. The MIT Press, 1990.

[5] G. E. Blelloch, C. E. Leiserson, B. M. Maggs, C. G. Plaxton, S. J. Smith, and M. Zagha. A comparison of sorting algorithms for the Connection Machine CM-2. In *Annual ACM Symp. Paral. Algor. Arch.*, pages 3–16, 1991.

[6] R. Cole. Parallel merge sort. *SIAM J. Comput.*, 17(4), August 1988.

[7] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. MIT Press, 1990.

[8] W. D. Frazer and A. C. McKellar. Samplesort: A sampling approach to minimal storage tree sorting. *J. ACM*, 17(3):496–507, 1970.

[9] J. S. Huang and Y. C. Chow. Parallel sorting and data partitioning by sampling. In *Proceedings of the IEEE Computer Society's Seventh International Computer Software and Applications Conference*, pages 627–631, 1983.

[10] F. T. Leighton. Tight bounds on the complexity of parallel sorting. *IEEE Trans. Comput.*, 34(4):344–354, 1985.

[11] F. T. Leighton. *Introduction to Parallel Algorithms and Architectures: Arrays, Trees, Hypercubes*. Morgan Kaufmann, 1992.

[12] H. Li and K. C. Sevcik. Parallel sorting by overpartitioning. In *Annual ACM Symp. Paral. Algor. Arch.*, pages 46–56, 1994.

[13] MasPar Computer Corporation. *MPPE User Guide, PN9305-0000*, 1992.

[14] D. Nassimi and S. Sahni. Bitonic sort on a mesh-connected parallel computer. *IEEE Trans. Comput.*, c-27(1), January 1979.

[15] nCUBE Corporation 1990. *nCUBE2 Processor Manual*, 1990.

[16] J. H. Reif and L. G. Valiant. A logarithmic time sort for linear size networks. *Journal of the ACM*, 34(1):60–76, 1987.

[17] John H. Reif, editor. *Synthesis of Parallel Algorithms*. Morgan Kaufman, San Mateo, Ca, 1993.

[18] Sequent Systems. *Guide to Parallel Programming*, 1987.

[19] Y. Won and S. Sahni. A balanced bin sort for hypercube multiprocessors. *Journal of Supercomputing*, 2:435–448, 1988.

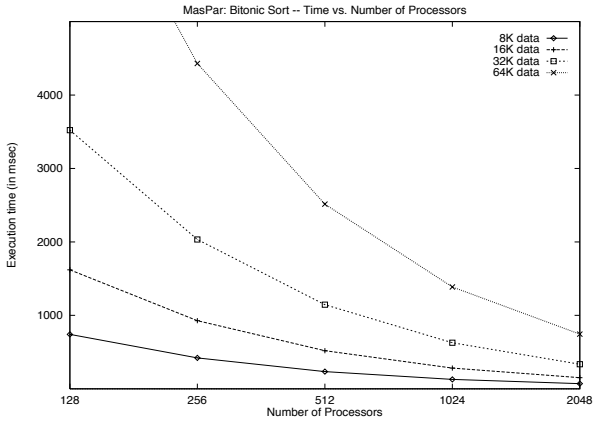

Figure 3: MasPar: Bitonic Sort, speed-up

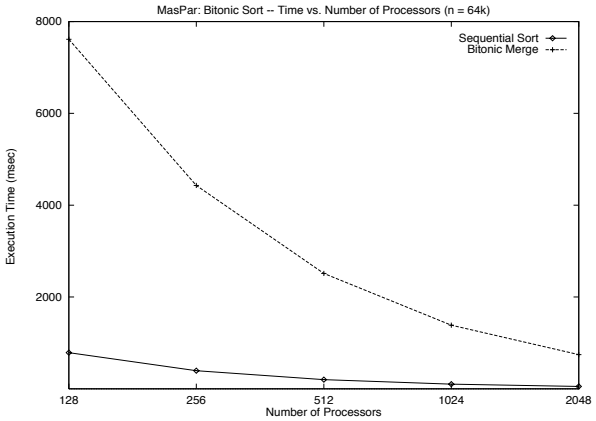

Figure 4: MasPar: Bitonic Sort, time division

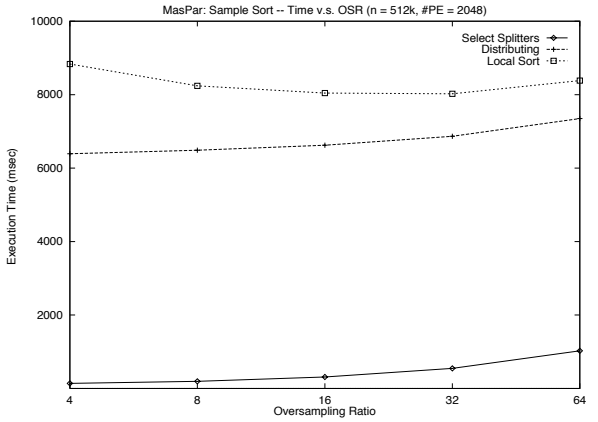

Figure 5: MasPar: Sample Sort, oversampling ratio

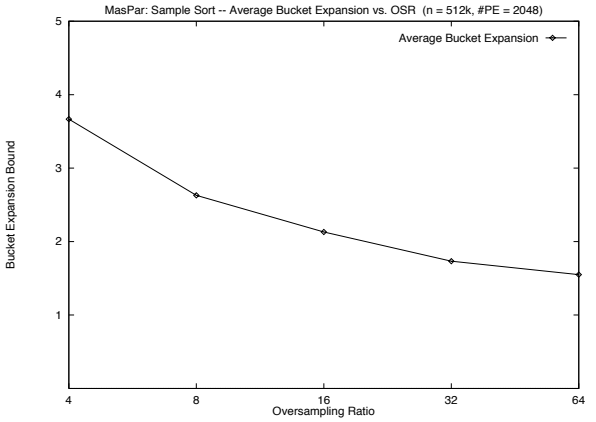

Figure 6: MasPar: Sample Sort, bucket expansion

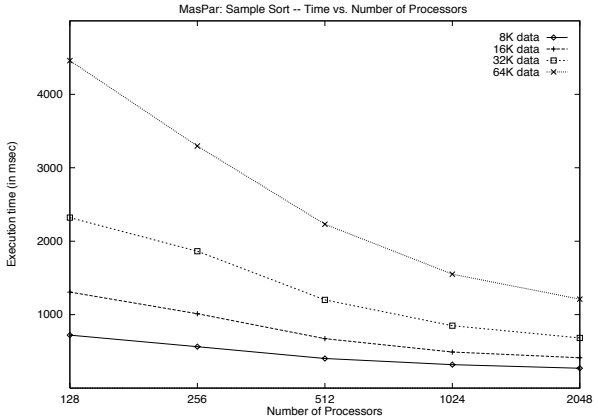

Figure 7: MasPar: Sample Sort, speed-up

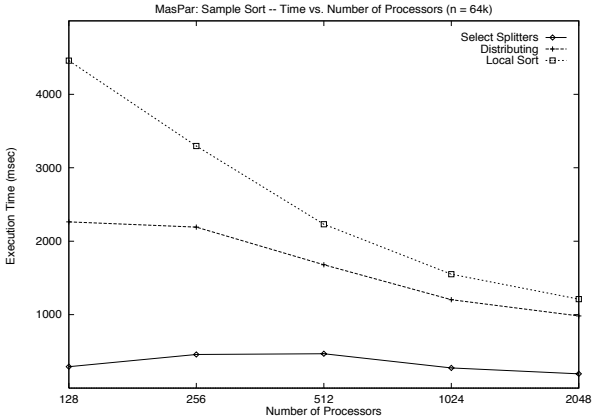

Figure 8: MasPar: Sample Sort, time division

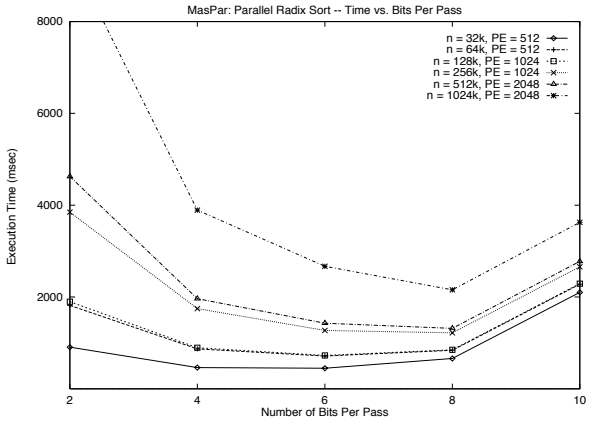

Figure 9: MasPar: Parallel Radix Sort, selecting r

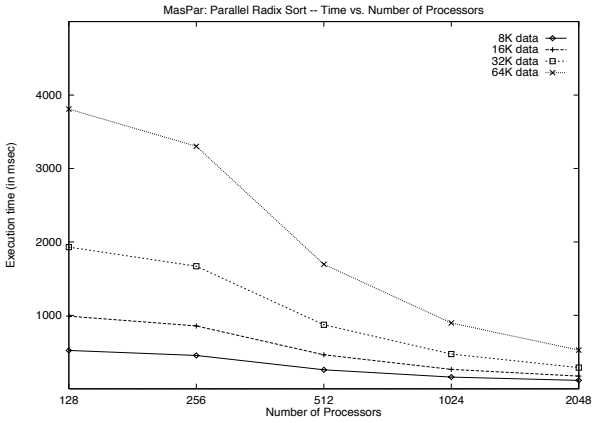

Figure 10: MasPar: Parallel Radix Sort, speed-up

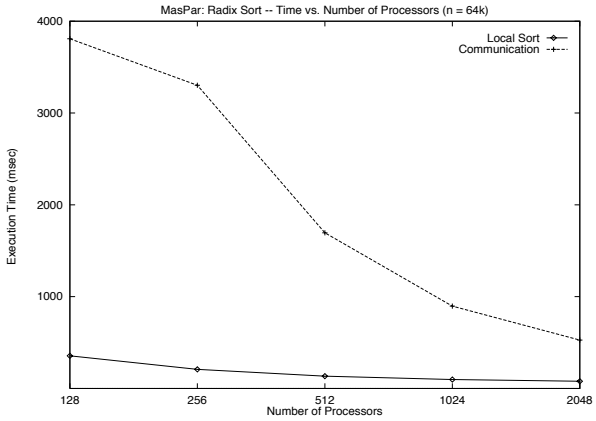

Figure 11: MasPar: Parallel Radix Sort, time division

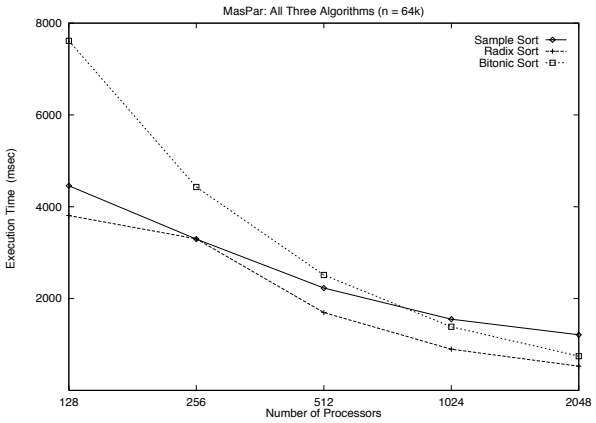

Figure 12: MasPar: All algorithms, speed-up, $n = 64K$

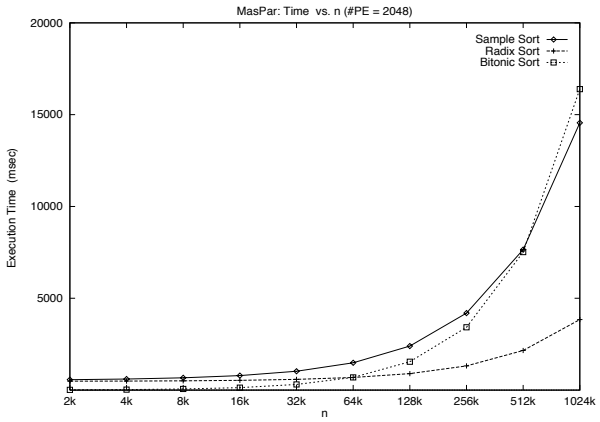

Figure 13: MasPar: All algorithms, $p = 2048$

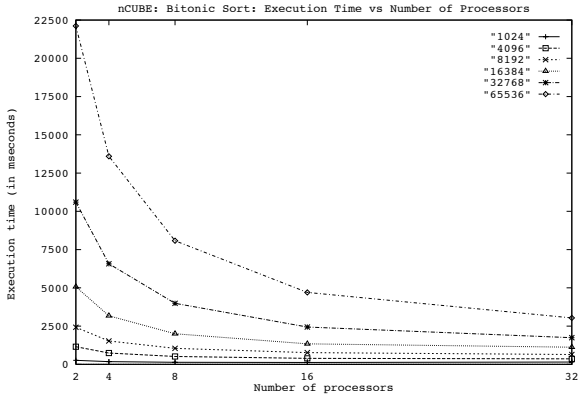

Figure 14: nCUBE: Bitonic Sort, speed-up

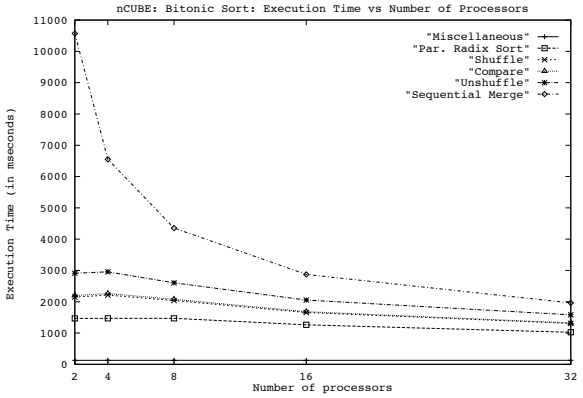

Figure 15: nCUBE: Bitonic Sort, time division

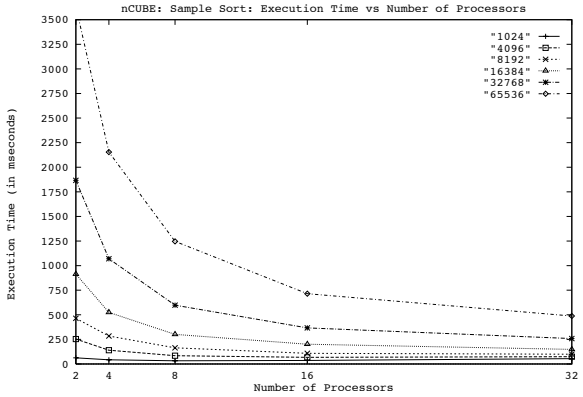

Figure 16: nCUBE: Sample Sort, speed-up

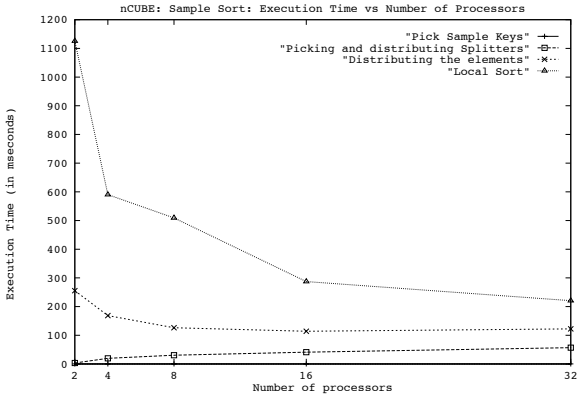

Figure 17: nCUBE: Sample Sort, time division

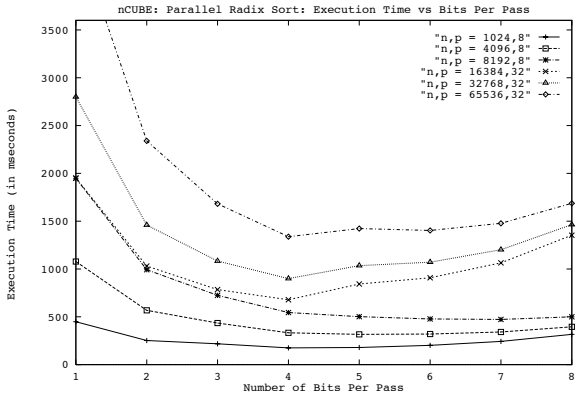

Figure 18: nCUBE: Parallel Radix Sort, selecting r

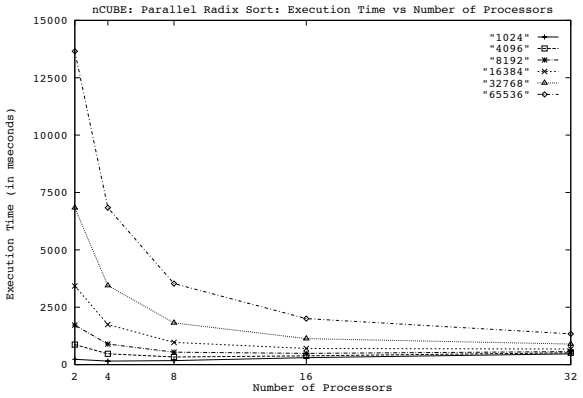

Figure 19: nCUBE: Parallel Radix Sort, speed-up

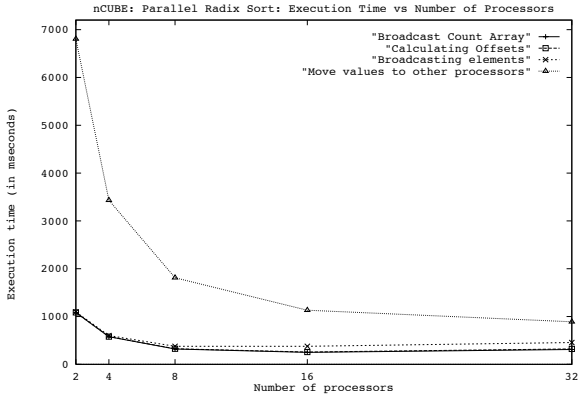

Figure 20: nCUBE: Parallel Radix Sort, time division

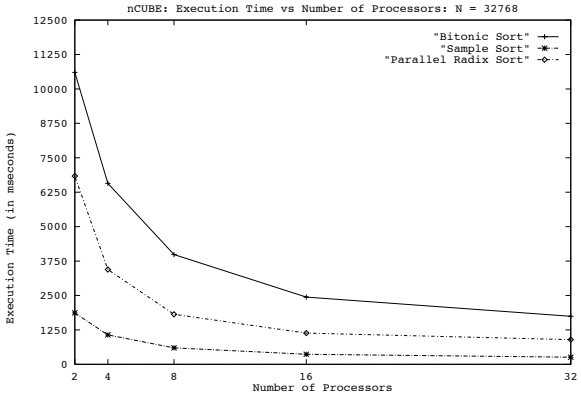

Figure 21: nCUBE: All algorithms, speed-up

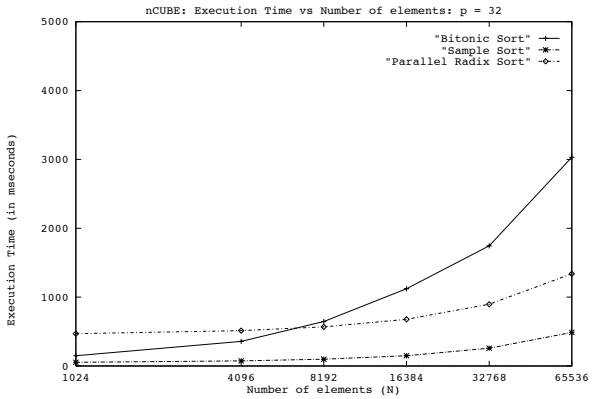

Figure 22: nCUBE: All algorithms, $p = 32$

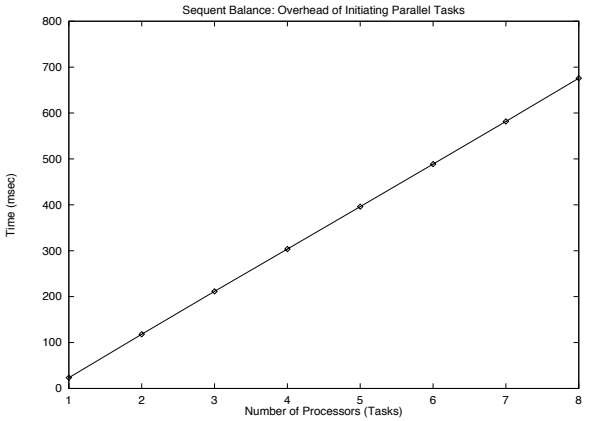

Figure 23: Sequent: task startup costs

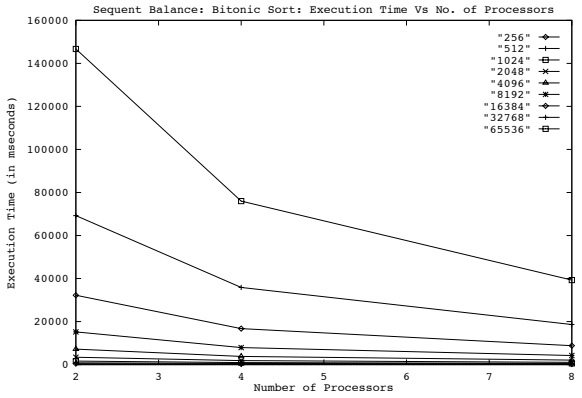

Figure 24: Sequent: Bitonic Sort, speed-up

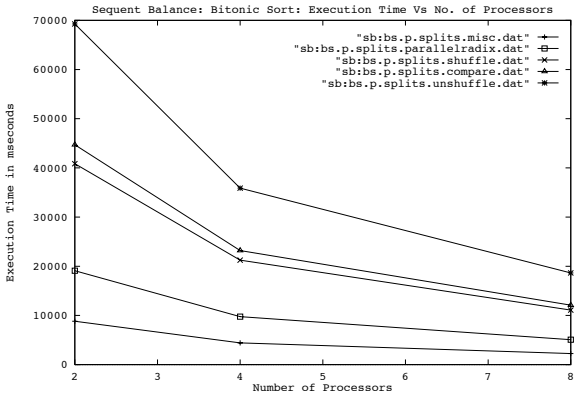

Figure 25: Sequent: Bitonic Sort, time division

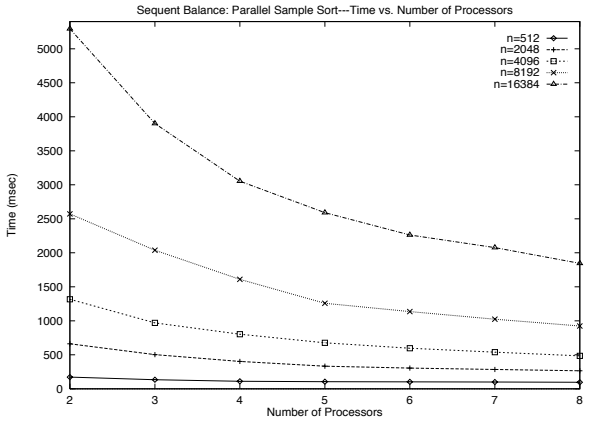

Figure 26: Sequent: Sample Sort, speed-up

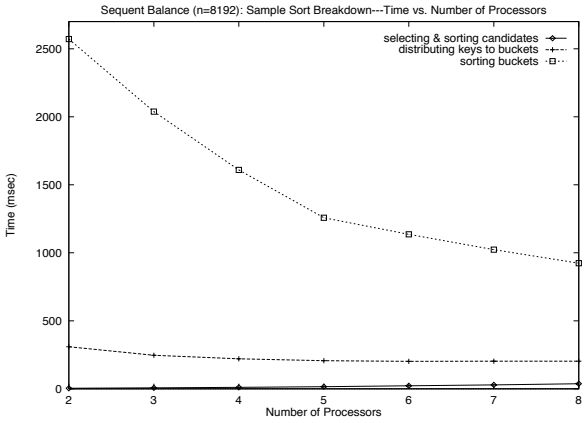

Figure 27: Sequent: Sample Sort, time division

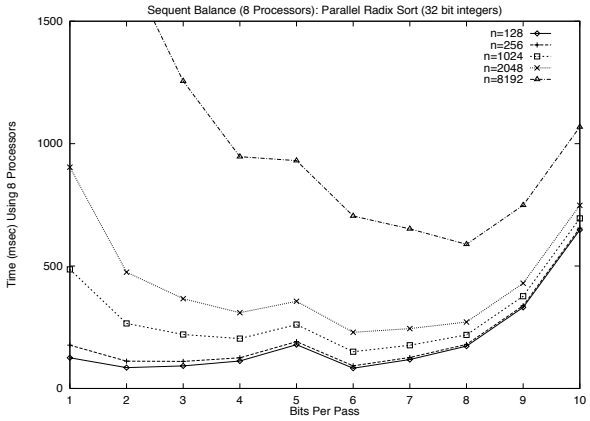

Figure 28: Sequent: Parallel Radix Sort, selecting r

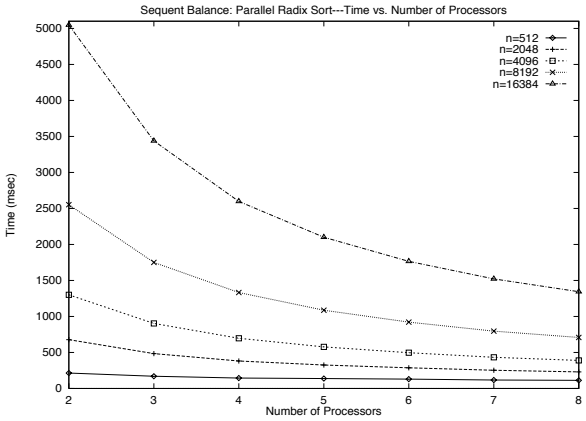

Figure 29: Sequent: Parallel Radix Sort, speed-up

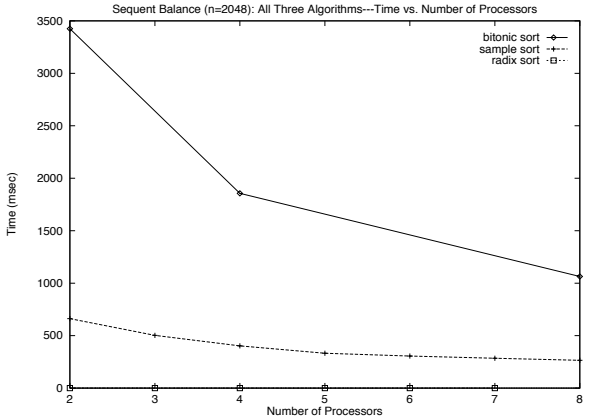

Figure 30: Sequent: All algorithms, speed-up

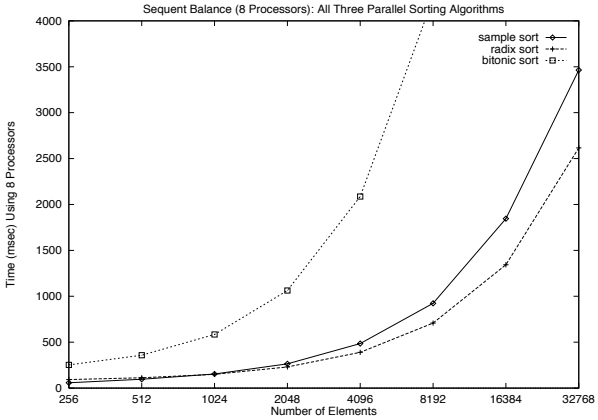

Figure 31: Sequent: All algorithms, $p = 8$