Proceedings of the 1986 ACM Fourteenth Annual Computer Science Conference
Cincinnati, Ohio, February 4-6, 1986 ISBN: 0-89791-117-6

PARALLEL MERGE-SORT ALGORITHMS ON THE HEP

Paul Hartono Singgih, The University of Tulsa
Howard B. Demuth, University of Idaho
Martin T. Hagan, The University of Tulsa
Roger L. Wainwright, The University of Tulsa

ABSTRACT

In this paper we describe four parallel merge-sort algorithms: (1) Parallel merging; (2) Bubble/merge; (3) Batcher's odd-even merge; and (4) Quicksort/merge. In each algorithm we divide a sequence of numbers of length n into k subsequences of equal length. Using k processors we sort each subsequence using a serial algorithm, either Merge-sort, Bubble, Batcher's or Quicksort. Finally the k sorted subsequences are merged in parallel using a parallel implementation of the tree sort. Each algorithm was run on the Denelcor HEP computer. The HEP is the first commercially available MIMD multiprocessor system. Each algorithm was executed using k = 1, 2, 4, 8, and 16 processors over dataset sizes ranging from 64 to 8192 items. The serial and parallel CPU times and the speedups for each algorithm are presented.

1. INTRODUCTION

According to the multiplicity of instruction and data streams, digital computers can be classified into four categories as folows [9]: single instruction stream - single data stream (SISD), single instruction stream - multiple data stream (SIMD), multiple instruction steam - single data stream (MISD), multiple instruction stream - multiple data stream (MIMD). The research reported here was done on the Denelcor HEP, an MIMD computer.

Performance of parallel algorithms on MIMD machines is often measured by the speedup ratio. Ideally, the speedup that can be achieved by a parallel computer with n processors working concurrently on a single problem is at most n (times faster than a single processor). In practice, the speedup is less than n because some processors may be idle at a given instant. The speedup ratio is defined to be the ratio of the time required for the problem using the best uniprocessor algorithm to the time required by a given parallel algorithm. Thus, the speedup ratio is $S^{(k)} = T^{(1)} / T^{(k)}$. Here $S^{(k)}$ is the speedup ratio by using k processors, and $T^{(1)}$ and $T^{(k)}$ are the times required for solving the problem using one and k processors, respectively. $S^{(k)} = k$ is often called the optimal speedup ratio.

2. THE HEP MULTIPROCESSOR SYSTEM

The Denelcor Heterogenous Element Processor (HEP) is a large scale scientific multiprocessor system which can execute a number of sequential (SISD) or parallel (MIMD) programs simultaneously. The system contains up to 16 Process Execution Modules (PEM) and up to 128 Data Memory Modules(DMM). The PEM and DMM are connected with the I/O and the control subsystem. The HEP is the first commercially available MIMD multiprocessor system. It is manufactured by Denelcor, Inc. Aurora, Colorado. For details on the architecture of the HEP see [7, 9, 10, 15].

The PEM is designed to execute multiple independent instruction streams on multiple data stream simultaneously. This is accomplished by pipelining each PEM with multiple functional units. Here, maximum parallelism can be achieved by providing multiple independent instruction streams executing multiple data streams in a pipelined fashion. Because the multiple instructions executed concurrently by an MIMD machine are independent of each other, the execution of one instruction does not influence the execution of the other instructions and true parallelism in processing may be achieved.

Extensions were made to Fortran 77 and to C on the HEP to provide language support for parallel processing. A special data type called an asynchronous variable was introduced to enable synchronization between cooperating and competing processes. Such a variable may be written into only when its location is "empty" and may be fetched only when it is "full". Either operation on an asynchronous variable that does not meet these requirements waits under hardware control until the proper access state is set by a parallel process. A statement called CREATE is used to create a parallel process. It is similar to a Fortran CALL, but it causes the created subroutine to run in parallel with its creator. The created Process is eliminated when it reaches the RETURN statement.

3. PARALLEL SORTING AND MERGING ALGORITHMS.

In this section we describe four parallel merge-sort algorithms: (1) Parallel merging; (2)

Bubble/merge; (3) Batcher's odd-even merge; and (4) Quicksort/merge. All of these algorithms can be classified as "easy split and hard join" sorting algorithms [3, 13, 14]. The basic idea behind each algorithm is the same and is described as follows:

1. Divide the original sequence of numbers of length n into k subsequences each of length N. Using k processors, sort these subsequences in parallel using a serial sorting algorithm.
2. Merge the k sorted subsequences in parallel using a parallel implementation of the tree (or tournament sort) algorithm.

Each of the four algorithms was run using k = 1, 2, 4, 8 and 16 processors on datasets of sizes n = 64, 128, 256, 512, 1024, 2048, 4096 and 8192. Tables reporting results of CPU times and speedup only show results for n = 128, 512, 2048 and 8192 to conserve space. All datasets consist of randomly generated values.

3.1 Parallel Merging

The concept of merging involves a multipass process, in which consecutive passes create longer and longer partitions until a final pass develops a partition containing all elements of the list to be sorted. The number of passes required depends on the initial number of partitions and the order of the merge. Consider a merge sort on a list of 16 elements. The merge process is entered with 16 partitions of size 1. The first pass will merge the initial 16 partitions into 8 partitions of two elements each, using a two way merge. The process of merging is repeated again and again until a sorted sequence of 16 items is formed. There are 8 independent processes in pass 1 that can be done simultaneously. These independent processes will decrease by a factor of 2 in every pass until only one process can be activated to merge the final sequence. Parallelism is introduced by having 8 processes work on the initial subsequences, 4 processes active at the next merge stage, etc, until only one process is used to merge the final sequence. However, due to the limited number of processes that can be used simultaneously on the HEP, this concept fails to work when the number of items exceeds twice the number of processors available.

To implement an algorithm that can be run on the HEP, we partition the initial unsorted sequence into k subsequences (1 < k < 16) and sort the subsequences in parallel using k processors and the sequential merge sort algorithm. Next, we merge the k sorted subsequences in parallel using the merge sort until a sorted sequence is formed. Hence we call this algorithm Parallel merging. Tables 1 and 2 show the CPU times and speedup for the Parallel merge, respectively. Figure 1 graphically illustrates the speedup of the Parallel merge. Here, the number of comparisons required is the same on the average, independent of k. Thus, the speedup obtained is due solely to the use of multiple processes.

n= k	128	512	2048	8192
1	0.045	0.223	1.068	4.977
2	0.025	0.123	0.580	2.673
4	0.017	0.079	0.358	1.611
8	0.014	0.060	0.262	1.153
16	0.014	0.057	0.240	1.068

Table 1. Parallel merge times (sec) for n items, k partitions and k processors

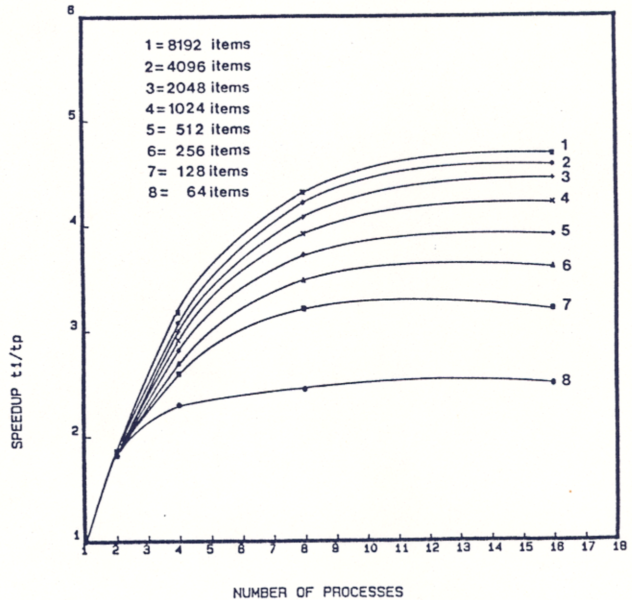

Figure 1. The Speedup of Parallel merge

3.2 The Bubble/Merge Sort

This algoritm is similar to the Parallel merge just described. It combines the bubble sort with the merge sort and runs both in parallel. Here, we divide the original unsorted sequence into k subsequences to be executed on k processors. These subsequences are first sorted using the bubble sort. Then the merge sort is used to merge two of the sorted subsequences.

The choice of the number of partitions (k) will affect the operation of the system in several ways. First the number of comparisons required to sort (or the actual work to be done) varies with k. Second, the number of processes that can be assigned to do the work also varies with k. Finally, the ability of the HEP to handle multiple concurrent processes varies with the number of such processes [8]. Table 3 shows the serial times (sec) for the Bubble/merge sort using k partitions, but only one processor. As may be seen from this table, the serial sorting times decrease as the number of partitions increase. This is because the number of comparisons (the actual work to be done) decreases

n= k	128	512	2048	8192
1	1.00	1.00	1.00	1.00
2	1.80	1.81	1.84	1.86
4	2.65	2.82	3.05	3.09
8	3.21	3.72	4.08	4.32
16	3.21	3.91	4.45	4.66

Table 2. The Speedup for Parallel merge

as the number of partitions increase. The number of comparisons for the bubble sort is $O(n^2)$, whereas the number of comparisons for the merging process is $O(n)$, where n is the number of items. Thus, the speedup for the serial version is obtained from reducing the number of items to be sorted using the bubble sort. Figure 2 illustrates the speedup of the serial Bubble/merge sort due to multiple partitions not multiple processors. Table 4 shows the parallel sorting times (sec) for the Bubble/ merge sort. The parallel sorting time is obtained when k processes are used to bubble sort the k original unsorted subsequences in parallel, then merge the k sorted subsequences simultaneously using a parallel implementation of the tree sort. This speedup may exceed the number of processes used because the speedup is due not only to the use of parallel processes but is due also to the use of multiple partitions (that is, the algorithm changes with k). Figure 3 illustrates the speedup for parallel Bubble/merge sort. From this Figure we see that the speedup (t_1/t_p) is as high as 50 for 4096 items and 8 partitions (processes), even though the maximum linear speedup is 8 if 8 processes are used. Figure 4 illustrates the normalized speedup. This normalized speedup, due only to parallel computations, is obtained by dividing the time required to sort using a serial algorithm by the time to sort using a parallel algorithm with both algorithms utilizing the same number of partitions. The normalized speedups do conform to theoretical limits. When more than 8 processes are used the pipeline becomes full and only a small amount of additional speedup can be achieved. The speedup of the algorithm increases somewhat as the number of items to be sorted increases. This is because each process has more computations to perform and a correspondingly smaller percentage of time is spent on synchronization.

3.3 Batcher's Odd-Even Parallel Merging

Historically, Batcher's Odd-Even merging is one of the earliest parallel sorting networks to appear in the literature [1]. It is a parallel sorting algorithm based on the principle of iterated merging. This scheme can be described as a multiple-input muliple-output switching network, in which all input data are available at once and the output sorted data are

n= k	128	512	2048	8192
1	0.117	1.885	30.20	477.
2	0.065	0.970	15.10	239.
4	0.041	0.520	7.6	120.
8	0.032	0.306	4.0	61.
16	0.031	0.210	2.2	31.

Table 3. Bubble/merge sort serial sorting times (sec) for n items and k partitions on one processor

n = k	128	512	2048	8192
1	0.120	1.915	31.0	495.
2	0.036	0.516	7.9	124.
4	0.069	0.160	2.2	32.
8	0.013	0.072	0.7	9.
16	0.013	0.055	0.4	4.

Table 4. Parallel Bubble/Merge Sort Time (sec) for n items, k partitions and k processors

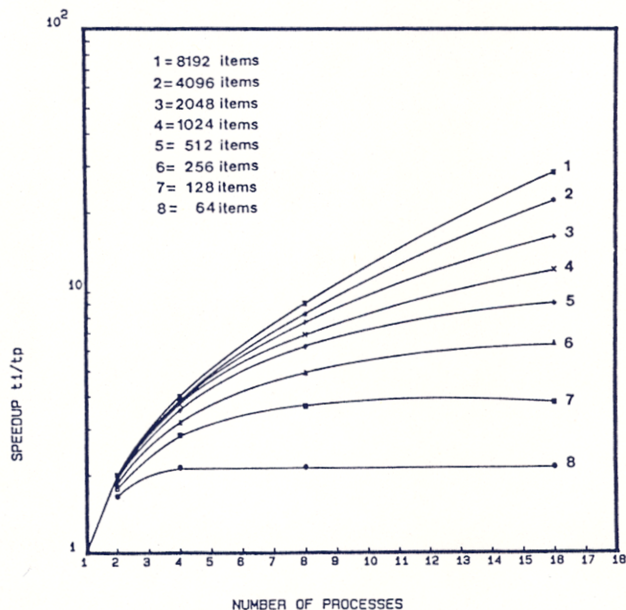

Figure 2. Speedup of Bubble/Merge Sort due to Multiple Partitions

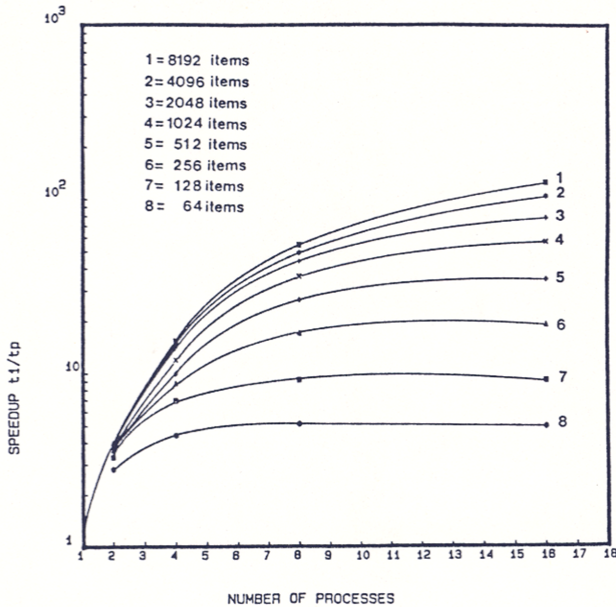

Figure 3. The Speedup of Parallel Bubble/merge sort

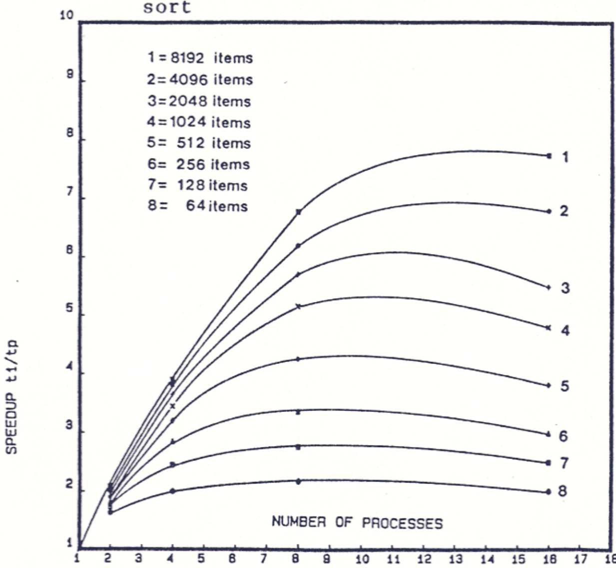

Figure 4. Normalized Speedup of Bubble/Merge sort due to Parallel Computations

produced simultaneously after all input data are compared and exchanged iteratively. The basic element of a sorting network is a comparator and switch module. Here numbers entering at the left are compared and sent out at the right. This module causes an interchange of its data, if necessary, so that the larger number appears on the lower line after passing through the module. When the initial sequence of 2^k items is to be sorted, iterated merging creates sorted sequences of length 2, 4, 8 ..., 2^k during successive stages of the algorithm. Consider Batcher's Odd-Even parallel merging algorithm. This algorithm is described as follows:

1. Let $a_1 < a_2 < a_3 < \ldots < a_n$ and $b_1 < b_2 < b_3 < \ldots < b_n$ be two sorted sequences to be merged, where for simplicity, we assume $n = 2^k$ and $k > 0$. If $k = 0$ the network is a single comparator module.
2. For any $k > 1$ generate the odd sequences $(a_1, a_3 \ldots , a_{n-1})$ and $(b_1\ b_3, \ldots , b_{n-1})$ and merge them using Odd-Even merge to obtain a sorted sequence $d_1 < d_2 < \ldots < d_n$. Likewise generate the even sequences $(a_2, a_4, \ldots , a_n)$ and $(b_2, b_4 \ldots , b_n)$ and merge to obtain a sorted sequence $e_1 < e_2 < \ldots < e_n$.
3. Define $c_1 = d_1$, the lowest value of the odd merge
   $c_{2i} = \min\ [d_{i+1}, e_i]$ $c_{2i+1} = \max\ [d_{i+1}, e_i]$; $i = 1, 2, \ldots, n-1$
   and
   $c_{2n} = e_n$ the largest value of even merge.

It can be shown that $c_1 < c_2 < c_3 < \ldots . < c_{2n}$ is the sorted sequence [13]. It may be noted here that to sort N items, the Odd-Even sorting network requires $O((\log_2 N)^2)$ steps and $O(N(\log_2 N)^2)$ comparators. Figure 5 illustrates a 4 by 4 Odd-Even merge for sorting 8 numbers of arbitrary order. From this Figure we see that when we are to sort an unsorted sequence of 8 items, the first stage generates 4 sorted subsequences of 2 items each, in which one step of 4 parallel comparisons is performed. In the second stage, two steps of 4 and 2 parallel comparisons are needed to merge the 2 item sorted subsequences into 4 item sorted subsequences. Finally, 3 steps of 4, 2, and 3 parallel comparisons are needed to merge the two sorted sequences into 1 sorted sequence of 8 items. If it is assumed that a single comparator exchange module can be replaced with one process, there are a maximum of 4 processes that can be used simultaneously. Again, this algorithm fails to work as the number of items exceeds 100, (twice the maximum available processes on one PEM in the HEP). Our implementation of Batcher's odd-even sort on the HEP is described as follows:

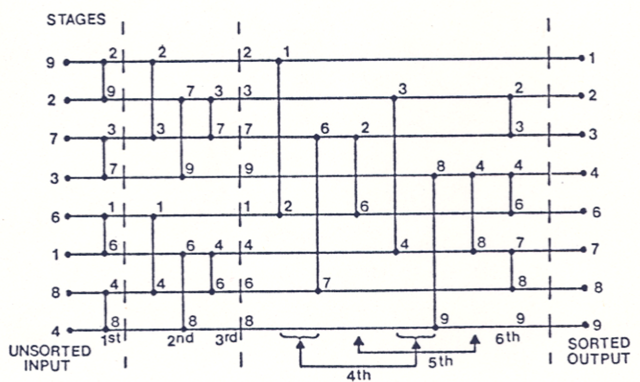

Figure 5. 4 by 4 Odd-Even Merging Network.

Once again, the original unsorted sequence is divided into k subsequences. These subsequences are first sorted using the Batcher's Odd-Even merging, then using the same algorithm, two of those sorted subsequences are merged. However, in this case, since non-overlaping comparison-

exchanges can be performed in every stage, all processors can be used to merge the sorted subsequences until a sorted sequence containing all items is obtained! We implemented a modification of Batcher's odd-even algorithm developed by Sedgewick [16]. Figure 6 illustrates the modification of Batcher's Odd-Even merging for sorting 8 arbitrary items. Here, an additional step is needed to shuffle the input after each step of comparison-exchange.

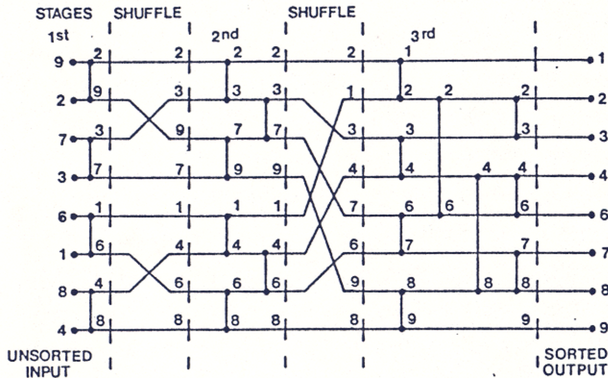

Figure 6. Modification of Batcher Odd-Even Merging

In the shuffle process, the first sorted input array is moved into the odd positions, A[1], A[3], ... , A[2N-1] of the output array, and the second sorted input array is moved into the even positions A[2], A[4], ... , A[2N] of the output array. Then Batcher's method may be implemented as follows:

```
for j = 1, 2, ... , N
  if A[2j-1] > A[2j] then swap
next j
for d = 2^(log2N-1), 2^(log2N-2), ... , 1
  for j = 1, 2, ..., N-d
    if A[2j] > A[2j+2d-1] then swap
  next j
next d
```

In this program, we see that the only statements which actually operate on the data are the compare-exchange statements of the form:

if A[2j-1] > A[2j] then swap

and

if A[2j] > A[2j+2d-1] then swap;

and these are performed in the same order regardless of the input. To execute this program in parallel, a slight modification is performed as described below:

```
    DO 500 J=1,NPROC
    KK (J) = J
    CREATE CPEX1(A,ISTART,IEND, KK(J))
500 CONTINUE
    .

    SUBROUTINE CPEX1(A, ISTART, IEND, NPROC)
    .
    DO 700 K=ISTART-1+NPROC, IEND/2, NPROC
     IF ( A(2*J-1) .GT. A(2*K)) THEN
       SWAP
    ENDIF
700 CONTINUE
    RETURN
    END
```

From the algorithm described above, we see that one process will not only perform a single comparison-exchange, but the compare exchange tasks are more evenly distributed among the processes used. Tables 5 and 6 show the serial and parallel times (sec) for Batcher's Odd-Even merge. The serial times increase as the number of partitions increase. This is due to the overhead created from the call statement of the program. Table 7 and Figure 7 show the speedup of parallel Batcher's Odd-Even merge. This speedup is obtained by dividing the time required to sort the items for the serial version with the time required to sort for the parallel version with the same partitions.

n= k	128	512	2048	8192
1	0.068	0.359	1.859	9.401
2	0.071	0.370	1.914	9.659
4	0.079	0.412	2.120	10.61
8	0.094	0.479	2.436	12.13
16	0.117	0.573	2.882	14.17

Table 5. Serial Sorting Times (sec) of Batcher's Odd-Even Merging for n items, k partitions on one processor

n= k	128	512	2048	8192
1	0.068	0.359	1.859	9.401
2	0.043	0.218	1.121	5.560
4	0.032	0.154	0.769	3.745
8	0.028	0.129	0.630	2.998
16	0.032	0.137	0.666	3.156

Table 6. Parallel Sorting Times (sec) of Batcher's Odd-Even Merging for n items, k partitions and k processors

n= k	128	512	2048	8192
1	1.00	1.00	1.00	1.00
2	1.65	1.70	1.71	1.74
4	2.47	2.67	2.76	2.83
8	3.36	3.71	3.87	4.05
16	3.65	4.18	4.33	4.49

Table 7. Speedup of Batcher's Odd-Even Merge due to Parallel Computations

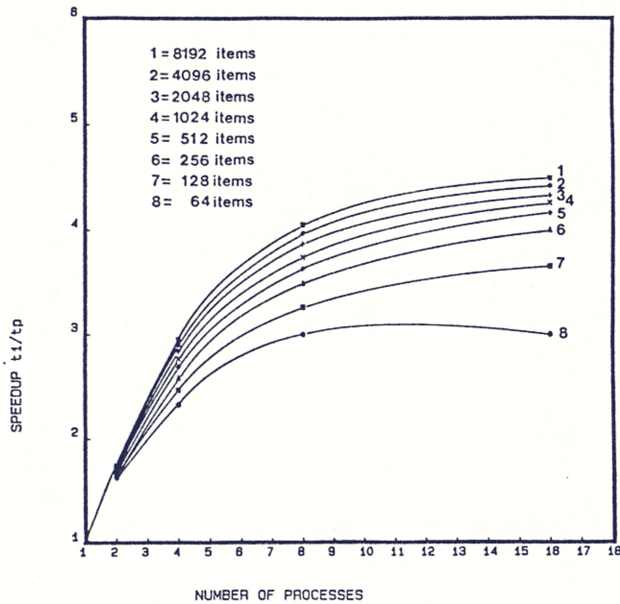

Figure 7. The Speed-up of Batcher's Odd-Even Merge Due to Parallel Computations

3.4 Quicksort/Merge Parallel Sorting Alorithm

The original unsorted sequence is divided into k subsequences. These subsequences are first sorted using quicksort. Then a merge sort is used to merge the sorted subsequences two at a time until the final sorted sequence containing all items is obtained. Here, once again the number of partitions (k) affects the system operation. However, k has a greater effect on the bubble/merge sort. The number of comparisons for the quicksort is in the order of ($C*N*\log_2 N$), where N is the number of items and C is some constant value between 1.1 and 1.4 [17]. The number of comparisons for the merging process is O(C*N) and involves $\log_2 k$ passes. Thus, we see that there are only small variations of time obtained when this algorithm is executed on one processor (serial). Tables 8 and 9 show the serial and parallel times (sec) for the Quicksort/merge algorithm. Table 10 shows the normalized speedup. Figure 8 illustrates the speedup due only to parallel computations for the Quicksort/merge algorithm. From Tables 8, 9, and 10 we see that even though there is a factor of about 6 speedup obtained (for sorting 8192 items with 16 partitions), the time required to sort them in parallel is only 0.322 seconds. This shows that an excellent serial algorithm greatly influences the performance of a parallel system.

n= k	128	512	2048	8192
1	0.020	0.104	0.502	2.404
2	0.024	0.118	0.552	2.566
4	0.027	0.130	0.601	2.752
8	0.023	0.108	0.515	2.398
16	0.019	0.092	0.449	2.139

Table 8. Serial Times (sec) of Quicksort/Merge algorithm for n items and k partitions on one processor

n= k	128	512	2048	8192
1	0.021	0.106	0.516	2.434
2	0.015	0.073	0.332	1.494
4	0.013	0.057	0.248	1.083
8	0.007	0.028	0.123	0.535
16	0.005	0.017	0.072	0.322

Table 9. Parallel Times (sec) of Quicksort/Merge due only to Parallel computations.

n= k	128	512	2048	8192
1	1.00	1.00	1.00	1.00
2	1.60	1.62	1.66	1.72
4	2.08	2.28	2.42	2.54
8	3.28	3.86	4.19	4.48
16	3.80	5.41	6.24	6.64

Table 10. Speedup of Parallel Quicksort/Merge Algorithm

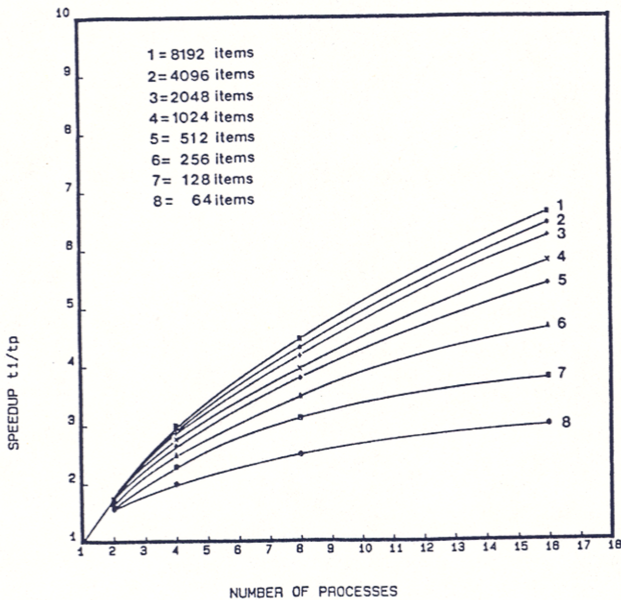

Figure 8. The Speedup of Quicksort/Merge algorithm due to Parallel Computations

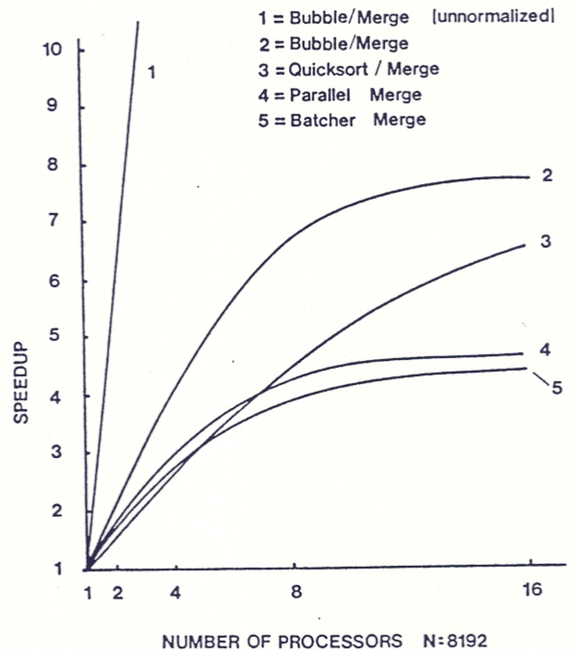

Figure 9. Composite graph showing normalized speedups for all four algorithms and unnormalized speedup for Bubble/merge sort with N=8192

4. RESULTS AND CONCLUSIONS

Several parallel merge sorting algorithms were developed and implemented on the HEP. These algorithms were all based on the concept of dividing the original unsorted sequence into a number of subsequences and sorting them separately in parallel. Then, the sorted subsequences are merged in parallel. Consider the following summary graphs showing CPU tmes and speedups via algorithm (Figures 9 and 10). From Figure 9, curve 1, we see that the speedup for the Bubble/merge sort exceeds the number of processors used because in this case the speedup is due not only to the use of parallel processes, but is due also to the use of multiple partitions (that is, the algorithm actually changed with k). For this reason we have plotted normalized speedup (due only to the use of parallel processes) for all four algorithms in curves 2 through 5. Among the other three alorithms, Batcher's merge and Parallel merge yielded about the same speedup ratios. Quicksort/merge yielded the best speedup performance. When more than 8 processors are used the pipeline becomes full and only a small amount of speedup can be achieved. Among the algorithms evaluated, the combination of the Quicksort/Merge was the best in terms of parallel CPU times.

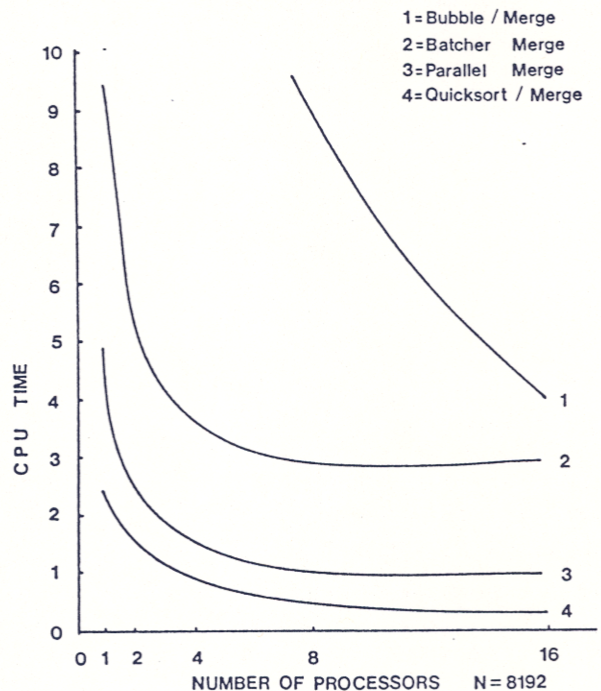

Figure 10. Composite graph of CPU times for all algorithms and N=8096

The Parallel merge also yielded excellent CPU times (see Figure 10). The Bubble/merge algorithm required the greatest CPU times despite its inflated speedup performance. It is interesting to note that the Parallel merge and Batcher's odd-even merge had almost identical speedups. However, the Parallel merge performed

significantly better than Batcher's merge in CPU times. Batcher's Odd-Even Merge is a network sorting algorithm designed for large numbers of processors. Sixteen to thirty-two processors is far short of the 512 to 1024 or more processor environment that it was designed for. Parallel merge, Bubble/merge and Quicksort/merge are divide and conquer algorithms. Thus, the better performing algorithms in serial are the better performing algorithms in parallel in our circumstances, where the number of processors is considerably fewer than the number of items to be sorted.

Over the last decade, parallel sorting has been a topic of active research. There are many parallel sorting algorithms currently known, ranging from network sorting algorithms to algorithms for hypothetical shared memory parallel computers, or VLSI chips. Typically, algorithms have been developed for hypothetical computers that utilize unlimited parallelism and space for solving the sorting problem in a hypothetical minimal time. Further research is needed to determine whether these algorithms can be adapted to realistic models of parallel computation.

Finally, the implementation of parallel algorithms on the HEP is a natural and straightforward process. It is concluded that the HEP can be applied to advantage in the effective processing of data sorting algorithms.

ACKNOWLEDGEMENT

The authors wish to acknowledge the technical support and encouragement given by Los Alamos National Laboratory and various staff members including Ann Hayes, Olaf Lubeck, and Bob Hiromoto. Dale Carstenson and Larry Wolf of the Denelcor staff at Los Alamos have been helpful in solving various problems and in facilitating our remote operation.

REFERENCES

1. Batcher, K. E., "Sorting Networks and Their Applications", Proc. AFIPS 1968 SJCC, vol. 32, Montvale, NJ: AFIPS Press, pp. 307-314.

2. Bitton, Dina; DeWitt, David J; Hsiao, David K; and Menon, Jaishandar; "A Taxonomy of Parallel Sorting", Computer Surveys, vol. 16, no. 8, September 1984, pp. 287-318

3. Cheung, J; Dhall, S.; Lakshmivarahan; Miller, L.; and Walker B.; "A New Class of Two stage Parallel Sorting Schemes", Proc. Natl. Assoc. Comput. Mach. Conf., 1982, pp. 26-29.

4. Cook, Curtis R.; and Kim, Do Jin; "Best Sorting Algorithm for Nearly sorted Lists", Communication of the ACM, vol. 23, no. 11, November 1980, pp. 620-624

5. Dongarr; J. J.; and Eisenstat; "Squeezing the most out of Algorithms in Cray Fortran", Argone National Laboratory, May 1983.

6. Dongarra, J. J.; and Hiromoo, Robert E.; "A Collection of Parallel Linear Equation Routines for the Denelcor HEP", Parallel Computing, vol. 1, no. 2, December 1984.

7. Fortran 77 Refrence Manual Release 1.0, Denelcor Inc., Aurora, Colorado, Publication Number 9008020-000, June 1984

8. Hagan, Martin T.; Demuth, Howard B.; and Singgih, Paul H.; "Parallel Signal Processing Research on the HEP", Proceedings of the 1985 International Conference on Parallel Processing. St Charles, Ill., August 20-23, 1985, pp. 599-606

9. Hwang, Kai; and Brigg, Faye A.; Computer Architecture and Parallel Processing, McGraw Hill, 1984.

10. Jordan, Harry F.; Performance Measurements on the HEP a Pipelined MIMD Computer, Electrical and Computer Engineering Department Univ. of Colorado, Boulder, Colorado, 1981.

11. Knuth, Donald E.; The Art of Computer Programming, vol. 3, Sorting and Searching, Addison Wesley, Reading MA., 1973.

12. Kung, H. T.; The Structure of Parallel Algorithms, Advances in Computers, vol. 19, Academic Press Inc., 1980.

13. Lakshmivarahan, S.; Dhall, Sudarshan, K.; and Miller, Leslie L.; Parallel Sorting Algorithms, Advances in Computers, vol. 23, Academic Press Inc., 1984.

14. Merritt, Susan M.; "An Inverted Taxonomy of Sorting Algorithms"; Communication of the ACM, vol. 28, no. 1, January 1985, pp. 96-99

15. Moore, James W.; The HEP Parallel Processor, Los Alamos Science, Fall 1983.

16. Sedgewick, Robert; "Data Movement in Odd-Even Merging", Journal Comput. SIAM, vol. 7 no. 3, 1978, pp. 239-272

17. Wirth, Niklaus; Algorithms + Data Structures = Programs, Prentice Hall, 1976