# Sorting on the Cray T3D

*Brandon Dixon*, University of Alabama, Tuscaloosa, Alabama, and
*John Swallow*, Davidson College, Davidson, North Carolina

**ABSTRACT:** *In this paper we study the sorting performance of the CRAY T3D on a variety of sorting tasks. Our problems range from that of sorting one word per processor to sorting the entire memory of the machine, and we give efficient algorithms for each case. In addition, we give both algorithms that make assumptions about the distribution of the data and those that make no assumptions. The clear winner, if data can be assumed to be uniformly distributed, is a method that we call a hash-and-chain sort. The time for this algorithm to sort 1 million words per proccessor over 64 processors is less than two seconds, which compares favorably to about four seconds using a 4-processor CRAY C90 and about 17 seconds using a 64-processor Thinking Machines CM-5.*

## 1 INTRODUCTION

Sorting integers is an easily understood and well studied problem that is always a natural choice when examining the performance of a new computer system. We developed sorting algorithms particularly suited for the CRAY T3D parallel computer and evaluated the performance of the machine on several different sorting tasks. Our problems ranged from sorting one integer per processor to sorting all of the available memory of the machine, which is currently about 180 Million words. Our goal is both to be able to sort integers quickly and to gain insights into the efficient use of the T3D.

Sequential sorting algorithms have been extensively studied since the 1950's and 1960's. Our primary reference for sequential sorting algorithms is Knuth [K]. We will assume knowledge of some standard sequential sorts discussed in Knuth, such as Quicksort, Radix sorts, and Radix-Exchange sorts.

Parallel sorting algorithms have been of particular interest since 1968 when Batcher developed the bitonic sorting algorithm [K]. This sort requires $\Theta(\log^2 n)$ time using $n$ processors. His algorithm gives a fixed set of comparisons, called a sorting network, that can be easily implemented on a parallel computer. In 1983, Ajtai, Komlos, and Szemeredi gave a sorting network requiring $\Theta(\log n)$ time using $n$ processors [AKS]. This algorithm is theoretically optimal, but from a practical point of view the hidden constants in the $O$-notation are far too large to produce useful algorithms. Reif and Valiant proposed a more practical $O(\log n)$-time randomized algorithm for sorting, known as flashsort [RV]. Many other parallel sorting algorithms have appeared in the literature, including parallel versions of radix sort and quicksort [B], and parallel merge sort [C]. An extensive paper on implementing sorting algorithms on the Connection Machine CM-2 appears in [BLMPSZ].

## 2 MACHINE DESCRIPTION

The CRAY T3D is a parallel computer comprised of 128 DEC Alpha processors running at 151 Mhz with a network connecting them in a 3-D toroidal configuration. Each Alpha is a 64-bit RISC processor that has its own memory, and all processors have the capability to read from and write to the memory of any other processor. For this paper, the memory of each processor was 2 MW, about 1.6 MW of which was available to the user. Each processor of a CRAY T3D is equipped with a direct-mapped data cache of 1,024 words, and loads and stores are done in 4-word cache lines.

There are several other machine-specific capabilities on the T3D. Each Alpha processor can request several types of memory operations by setting certain high bits in the memory address. These capabilities include a read-ahead mode, which improves sequential local memory reads, and non-cached reads, to permit other memory locations to reside in the cache. For the purposes of this paper we took advantage of only the read-ahead mode for memory references. Several functions are supported by the interconnection network. One is that of shared memory reads and writes, which we will call, in accordance with the names of their C functions, "`shmem_get`" and "`shmem_put`". Another capability is to perform a hardware barrier among all processors. This barrier is implemented using a flag bit on each processor, such that when a processor executes a barrier command, it sets its barrier bit and waits until all processors' barrier bits are set before proceeding. Once all processors reach the barrier, the barrier bits are cleared. There is also a provision to wait until all remote memory writes have completed by waiting for all acknowledgments from remote processors to be received; this command will be called a **net_quiet**. Finally, the function "memory barrier" flushes the buffer which holds data to be stored or transmitted; this verifies that the data has been written to local memory or has entered the interconnection network. For the rest of this paper, a "barrier" will refer to the execution of these operations in the following sequence: memory barrier, **net_quiet**, hardware barrier; this barrier insures that all remote memory operations have been completed.

We fix some notation concerning programs on the T3D. A tick will denote one clock cycle. We will denote by $N$ the number of processors used in a program; $N$ is constrained to be a power of 2. The individual processors we will denote by $P_0 \dots P_{N\text{-}1}$, and, for pseudo-code, **mype_num** will denote the processor number, so **mype_num** on $P_i$ will be $i$.

We point out that our run times are given generally for a single piece of C code optimized for running on 64 processors with a very large data size. For many fewer or many more processors, or much smaller data sizes, other methods would undoubtedly do better. We give run times for a variety of data sizes and number of processors for comparison, and in each run the data is fairly random and well distributed. We made efforts to optimize our use of the data cache, but since our code was in C, we undoubtedly faced instruction and data cache misses which good assembly subroutines could avoid. We did, however, use a version of C developed by Bill Carlson at the Supercomputing Research Center, and we note that his AC compiler typically gained 30% over the released CRAY C compiler.

## 3 SORTING ONE INTEGER PER PROCESSOR

The problem begins with one integer stored at each of the $N$ processors of the machine, and the goal is to finish with the integer at $P_0$ smaller than that at $P_1$, the one at $P_1$ smaller than that at $P_2$, and so on.

We studied two different algorithms to solve this problem. The first method that we tried was chosen because of its ease of implementation. The algorithm has three basic steps:

1. From each $P_i$, send the data element to array location $i$ at $P_0$.
2. Sort the array at $P_0$.
3. From $P_0$, send array element $i$ to $P_i$ for all $i$.

While this method is far from asymptotically optimal, we had some hope that because this is such a small, special case it might be the fastest solution. Our preliminary results showed, however, that our second method was a clear winner, even with these small data sizes, and thus no more effort was put into optimizing this method.

Our second idea for the one word problem was to use a sorting network. A sorting network is a fixed comparison-exchange sequence that is guaranteed to sort any input permutation. Because the number of data elements for this problem is $N$, always a power of 2, a natural choice is to use Batcher's Odd-Even merge sorting network. The network consists of (log$N$) (1 + log$N$)/2 rounds, and in each round selected pairs of processors compare and exchange their data. Although the asymptotic complexity of Batcher's method is $0(N \log^2 N)$ comparisons, the number of rounds is close to optimal when sorting $2^k$ elements for small $k$. What follows is a pseudo-code description of Batcher's algorithm and a discussion of our implementation. For a complete description and analysis of sorting networks and Batcher's method see Knuth.

Let $N$ be the number of processors, which is the number of data elements, and $d$, $p$, $q$, and $r$ be integer variables. In pseudo-code Batcher's algorithm is the following.

```
let q = N / 2;
while (p > 0)
  let q = N / 2, d = p, and r = 0;
  repeat
    if mype_num<(N-d) and
        (mype-num & p) = r then
      send your data to
        processor mype_num+d
      wait for the data from
        processor mype_num+d
      keep the smaller of the data items;
    otherwise if mype_num >=d and
        (mype_num-d) & p = r then
      send your data to
        processor mype_num-d;
      wait for the data from
        processor mype_num-d;
      keep the larger of the data items;
    let d = q-p, q = q / 2, and r = p;
  until p > q;
  let p = p / 2;
end.
```

The major concern in implementing this algorithm is interprocessor synchronization; the program must insure that every processor waits for the data word from its partner on a particular round. One strategy is to synchronize by executing barriers. On a given round of the repeat...until loop, every processor which sends a word of data insures that its data has been received by executing a barrier after the send. Since all processors must execute a barrier, we have those that are idle also execute a barrier. Since the overhead to execute a barrier is large compared to the time to send one word to another processor, this method dramatically increases the run time.

Instead of using barriers to synchronize communication, we chose to implement our own method of local synchronization between pairs of processors. Each processor first initializes an (log $N$) (1~ log $N$)/2-long array, one entry for each round of the algorithm. On the $k^{th}$ round, let $P_i$ and $P_{i+d}$ be a pair of processors comparing data. We have $P_i$ send its data to position $k$ in

$P_{i+d}$ 's array, and then we have $P_{i+d}$ busy-wait until the value of element *k* of the array changes from the initialized value. In this manner the processors are not waiting for their sends to be received by their partners, only that they have received the data from their partners. This method is significantly faster than using barriers, and since some processors are not involved in the last few rounds they are now free to begin other tasks earlier than the other processors, which may be helpful in some applications. The sole difficulty in coding this method is the construction of a busy-wait loop on a memory location, since without explicit indications to the contrary, a compiler will assume that the memory location cannot change during the execution of the loop and will set up an infinite loop. We give the run times for different numbers of processors *N* in the table below. We find that the run times are approximately 1250 + 195 (log *N*)(1 + log *N*)/2 ticks, or 195 ticks per round with some overhead.

| One Integer Per Processor: Table of run times | | | | | | |
|---|---|---|---|---|---|---|
| N | 4 | 8 | 16 | 32 | 64 | 128 |
| Clock ticks | 1787 | 2432 | 3176 | 4208 | 5393 | 6680 |

## 4 OUT-OF-PLACE METHODS

For the remainder of this paper, the sorting problem begins with an array of **NUMWORDS** integers stored at each of the *N* processors of the machine and the goal is to finish with a sorted array at each processor such that no integer at $P_0$ is greater than any integer at $P_1$, no integer at $P_1$ is greater than any integer at $P_2$, and so on.

For this section we will restrict **NUMWORDS** so that the data fits in at most half of the memory available to the program. This permits out-of-place algorithms that store an extra copy of the data. We studied two algorithms that fall into this category: a Batcher mergesort and a radix sort. While the Batcher sort succeeds for all data sets, the radix sort makes the assumption that the log *N* high-order bits are uniformly distributed. The radix sort is more than twice as fast as the Batcher mergesort for large data sets. We note that algorithms for sorting more data, considered later in this paper, are competitive with these algorithms even for small data sizes.

### *4.1 The Batcher Mergesort.*

The Batcher mergesort is a merging network, i.e., a fixed merging sequence among *N* processors that is guaranteed to sort any input permutation. Each processor begins by sorting its data using a sequential sort. Then a sequence of 2-processor merges is executed, where the processors that merge at each round are those that compare and exchange in the l-word algorithm described above. Since the sequence is the same, we give pseudo-code only for the merge, which replaces the line "keep the smaller (larger) of the data items" in the algorithm for the one-word sort.

The 2-processor merge begins with each processor having a sorted array of **NUMWORDS** integers. When the merge of processor $P_i$ and $P_{i+d}$ is completed, the array at $P_i$ contains the **NUMWORDS** smallest integers from the two arrays and $P_{i+d}$ contains the largest. To achieve this, the processors must exchange data, and the simplest algorithm is to have $P_i$ send $P_{i+d}$ a complete copy of its data and $P_{i+d}$ send a copy to $P_i$ as well. This creates unnecessary communication, however. Since processor $P_i$ is keeping the small values, it can begin merging from the small ends of the arrays, and likewise $P_{i+d}$ can begin merging from the large ends. Both processors complete when they have merged a total of **NUMWORDS** integers.

Note that $P_i$ will use the same number data values from $P_{i+d}$ in its merged array as $P_{i+d}$ uses from $P_i$. This fact implies that the extra communication can be eliminated, as follows. Whenever processor $P_i$ uses a value from its partner, it sends its partner another value. $P_i$ does not need to inform $P_{i+d}$ that it sent the data; we are assured that $P_{i+d}$ will send another value, so $P_i$ simply waits to receive this next data element. Of course the overhead is quite large for sending only one data element, so a compromise of sending a fraction of data elements works best. We will denote the size of the communication block **BUCKET**.

Let **A[0..NUMWORDS-1]** be an array which contains the processor's sorted list, **B** another array of the same size, **C** an array of length **BUCKET**, and *i, j, k, l* integer variables. We constrain **BUCKET** to be a divisor of **NUMW0RDS**. All other variables are as in the pseudo-code for the one-word Batcher sort. The processor that collects the smaller of the data items executes the following pseudo-code.

```
send A[NUMWORDS-BUCKET..NUMWORDS-l]
     to array C at processor
   mype_num+d;
wait until array C receives
   A[0..BUCKET-l] from processor
   mype_num+d;
let i = 0, j = 0;
let k = NUMWORDS-BUCKET,
while (i < NUMWORDS) do
   if (B[j] < A[i]) then
      let C[l] = B[j];
      increment j and l;
   else
      let C[l] = A[i];
      increment l;
   increment i;
   if (j=BUCKET) then
      decrement k by BUCKET;
      send A[k..k+BUCKET-l] to array
         C at processor mype_num+d;
      let j = 0;
      wait until array C receives next
         BUCKET words from processor
         mype_num+d;
end.
```

The result of the merge now lies in array **B**. The processor which collects the larger of the data items simultaneously executes the following pseudo-code.

```
send A[0..BUCKET-l] to array C
  at processor mype_num-d;
wait until array C receives
  A[NUMWORDS-BUCKET..NUMWORDS-l]
  from processor mype_num+d;
let i = NUMWORDS - 1, j = BUCKET -1;
let k = 0, 1 = NUMWORDS;
while (i >= 0 ) do
  if (B[j] > A[i]) then
    let C[l] = B[j];
    decrement j and l;
  else
    let C[l] = A[i];
    decrement l;
  decrement i;
  if (j<0) then
    increment k by BUCKET;
    send A [k  k+BUCKET-l] to array C
      at processor mype_num-d;
  let j = BUCKET;
  wait until array C receives
    previous BUCKET words from
    processor mype_num+d;
end.
```

The result of the merge now lies in array **A**.

Our implementation alternates the roles of arrays **A** and **B** at every round, merging from one into the other, and therefore avoids having to copy the data from **A** to **B** or from **B** to **A**. The local synchronization issues are greater than those of the one-word sort. Since we cannot afford to have many extra copies of array **C**, we execute barriers between each round. However, we still need two copies of array **C**, one to use in the merge and the other to receive data, and we use busy-waits on flags to coordinate the alternation of the copies of array **C**.

The Batcher mergesort requires two buffers of size **NUMWORDS** and two buffers of size **BUCKET**, so it faces a data size limit of roughly one-half the memory of the machine. We found that the optimal **BUCKET** value for **NUMWORDS**= 703125 was approximately 1125 words. A larger size increases the possibility of sending too many words, since only a fraction will actually be needed for the merge, and for a smaller size the communication overhead begins to degrade performance. Because the mergesort is an out-of-place sort, we could have used an out-of-place sequential sort for the initial single-processor sorts, but we opted for a version of Quicksort.

In the tables that follow, by "wpp" we mean "words per processor" . All run times are in millions of ticks (megaticks MT); thus 150 MT is one second of CRAY T3D cpu time (1 pe).

Note that the sequential sort takes more than half the time when **NUMWORDS** is greater than 500 for $N$ = 4,10000 words for $N$ = 8, or 500000 for $N$ = 16. When $N$ = 128, Batcher sorting **NUMWORDS**=703125 words devotes 30% of the running time to the initial sequential sort.

Batcher Mergesort: Table of run times (MT)

| wpp $N$ | 4 | 8 | 16 | 32 | 64 | 128 |
|---|---|---|---|---|---|---|
| 500 | .119 | .175 | .246 | .378 | .491 | .730 |
| 1000 | .244 | .355 | .503 | .761 | .994 | 1.26 |
| 5000 | 1.45 | 1.92 | 2.50 | 3.42 | 4.30 | 5.33 |
| 10000 | 3.13 | 3.95 | 5.02 | 6.40 | 7.95 | 9.68 |
| 50000 | 17.9 | 22.0 | 27.3 | 33.9 | 41.8 | 50.5 |
| 100000 | 37.6 | 45.5 | 55.0 | 67.2 | 81.1 | 97.6 |
| 500000 | 211. | 252. | 305. | 371. | 453. | 540. |
| 703125 | 301. | 368. | 438. | 537. | 644. | 775. |

Batcher Mergesort: Table of run times excluding initial sequential (MT)

| wpp $N$ | 4 | 8 | 16 | 32 | 64 | 128 |
|---|---|---|---|---|---|---|
| 500 | .057 | .111 | .185 | .312 | .429 | .666 |
| 1000 | .109 | .217 | .365 | .624 | .858 | 1.13 |
| 5000 | .531 | 1.01 | 1.59 | 2.50 | 3.40 | 4.41 |
| 10000 | .958 | 1.89 | 2.96 | 4.30 | 5.91 | 7.49 |
| 50000 | 4.54 | 8.76 | 14.2 | 20.9 | 28.7 | 37.1 |
| 100000 | 8.74 | 16.9 | 26.5 | 37.8 | 51.8 | 69.1 |
| 500000 | 43.1 | 86.8 | 136. | 205. | 286. | 368. |
| 703125 | 61.6 | 126. | 192. | 292. | 398. | 537. |

## *4.2 The Radix Sort.*

The radix sort assumes a uniform distribution in the high log $N$ bits of the data because these bits determine the processor number where the data will be stored at the conclusion of the sort. The sort first separates the data into $N$ portions based on the high log $N$ bits. Then each processor sends its $i^{th}$ portion to processor $P_i$. Finally each processor separates its data into 1024 buckets (for large data sizes) based on the next ten bits and, within each bucket, begins a radix-exchange sort, which employs successive Quicksort-style partition passes where partitions are made according to bits. It is standard to use a specialized sort, such as an insertion sort or a sorting network, when the partition becomes sufficiently small.

We implemented the data movement by an interprocessor conference, as follows. First, every processor $P_i$ sends the length of its $j^{th}$ portion to an array on processor $P_j$, indexed by $i$. Then each processor $P_j$ computes the partial sums of the received lengths; this computation results in a $N$-long list of indices **I[0 . . N-l]** into the target array on $P_j$. Then each processor $P_j$ sends out **I [i]** to an array on processor $P_i$, indexed by $j$, for all $i$. At this point $P_i$ knows where on $P_j$ to send its $j^{th}$ portion, and does so using a **shmem-put**.

The pseudo-code for this radix sort is the following. We require that no more than **NUMWORDS+EXTRA** data elements have the same top log $N$ bits and allocate this much space in array **A** and that no more than **BUCKETSIZE** initial data words have the same top log $N$ bits on any processor. Let $i$, $j$, $k$, $l$ be integers, **A [0 . . NUMWORDS+EXTRA-l]** an array, the first **NUMWORDS** words of which contain the initial data, **temp[O . . N-l][O . . BUCKETSIZE-1]** an array to hold bucketed data, **cnt[O . . 1023], cnt2[0 . . 1023]**, and **off[0 . . 1023]** arrays of integers, and **tlen[0 . . N-l]** an array to hold the sizes of each bucket, initialized to zero.

```
let i = 0;
repeat
  let j = the top log N bits
    of A[i];
  let tempt[j][tlen[j]] = A[i];
  increment i, tlen[j];
until i=NUMWORDS;
hold conference with tlen[] of
  each processor to determine
  where in array A on processor
  j each processor should send
  temp[j][0..tlen[j]-1];
send temp [j][0..tlen[j]-1] to
  processor j at computed  location,
  for each j;
make 1024 counts cnt[0..1023] of
  number of elements in A with
  each 10-bit pattern (below the
  top log N bits);
initialize off[i] to the sum of
  cnt[0..i-1] for each i in 0..1023;
let cnt2[i] be the sum of cnt[0..i]
  for each i in 0..1023;
let i = 0;
repeat
  let j = A[off[i]];
  repeat
    let l = ten bits after top log
      N bits of j;
    let k = A[off[l]];
    let A[off[l]]=j;
    increment off[l];
    let j = k;
  until l=i;
  while (off[i]=cnt[i] and i<1024)
    increment i;
until i=1024;
for each i in 0 . . 1024, perform a
  radix-exchange sort on each
  bucket of A;
end.
```

This fairly simple sort is a good test case for several T3D architecture features such as the read-ahead capability and the use of barriers. When $N = 64$ and **NUMWORDS**=703125, using the read-ahead capability reduced the time for the initial separation phase by about 3 million ticks down to 39 million ticks. With the same values of $N$ and **NUMWORDS**, we used barriers between each round of **shmem_ puts** in the data movement phase to bring the time for the stage down 6 million ticks, to 17 million ticks.

Comparing this sort with other sorts in this paper, we find that the sequential sort is more dominant in this radix sort than in any other sort: for $N = 64$, the sequential sort comprised 76% of the total time. Still, the network contention for large $N$ becomes significant: the run times are very close given the same number of data words per processor, until we reach 128 processors. The time for the interprocessor conference grows but is negligible (less than 20,000 ticks for $N = 64$), and the time to separate the data into buckets at the beginning is slightly lower (8 million ticks) when the number of processors is small enough so that the counts can be kept in cache. Ignoring these small effects, most of the run-time penalty, is due to increased demands on the network. Consider that as $N$ grows, each processor sends the same number of words but does so in smaller packets with more network contention. The $N = 128$ case, where 127 rounds of 128 simultaneous messages of size about 5700 words were sent, was 10,000,000 ticks (10 MT) slower than the $N = 64$ case, in which 63 rounds of 64 simultaneous messages of size about 11700 words were sent.

Half-Memory Radix Sort: Table of run times (MT)

| wpp $N$ | 4 | 8 | 16 | 32 | 64 | 128 |
|---|---|---|---|---|---|---|
| 500 | .291 | .297 | .341 | .348 | .385 | .580 |
| 1000 | .426 | .432 | .460 | .493 | .529 | .669 |
| 5000 | 1.36 | 1.44 | 1.45 | 1.51 | 1.58 | 1.66 |
| 10000 | 2.69 | 2.85 | 2.85 | 2.99 | 3.04 | 3.18 |
| 50000 | 14.2 | 14.6 | 14.9 | 15.4 | 15.5 | 16.2 |
| 100000 | 29.0 | 30.0 | 30.7 | 31.4 | 31.7 | 33.0 |
| 500000 | 155. | 160. | 164. | 167. | 168. | 175. |
| 703125 | 221. | 228. | 233. | 238. | 239. | 246. |

Half-Memory Radix Sort: Table of run times excluding initial sequential sort (MT)

| wpp $N$ | 4 | 8 | 16 | 32 | 64 | 128 |
|---|---|---|---|---|---|---|
| 500 | .0403 | .0501 | .0607 | .0802 | .116 | .197 |
| 1000 | .0741 | .0858 | .0964 | .119 | .156 | .226 |
| 5000 | .309 | .359 | .400 | .447 | .480 | .578 |
| 10000 | .581 | .693 | .741 | .833 | .898 | 1.06 |
| 50000 | 2.83 | 3.25 | 3.61 | 3.98 | 4.15 | 4.82 |
| 100000 | 5.65 | 6.56 | 7.21 | 7.90 | 8.19 | 9.53 |
| 500000 | 28.0 | 32.7 | 36.2 | 39.3 | 40.5 | 47.2 |
| 703125 | 39.3 | 46.0 | 50.5 | 55.2 | 56.8 | 66.3 |

## 5 SORTING MORE THAN HALF THE MEMORY OF THE MACHINE

The sorting problem is the same as that of sorting less than half the memory, but we no longer allow an extra buffer of comparable size to the data. We consider several sorts: a modification of the radix sort from the previous section; a partition sort, which is reasonably distribution independent; and a hash-and-chain sort, which requires uniform distribution in more top bits for peak performance. The Batcher mergesort was not extended due to the large performance penalty of an in-place merge.

### *5.1 The Partition Sort.*

The partition sort is an attempt to create a distribution-independent in-place sort for the T3D. The difficulty in doing so lies in the interprocessor communication, because we have several sequential in-place sorts available such as Quicksort and Radix-Exchange sort. Our sort proceeds in much the same way as the radix sort, where first data is sent to its destination processor and afterwards the data is sorted sequentially on each processor.

The algorithm is comprised of three stages. First, all processors jointly determine the $N$-tiles of the data, i.e., they each divide their data into $N$ portions such that

(a) no element in partition $i$ is greater than any element in partition $j$ for $i < j$, and

(b) for all $i$, the sum of the lengths of all $i^{th}$ partitions across all $N$ processors is **NUMWORDS**.

By doing so, we separate the data so that after the second stage, where the $i^{th}$ portions are sent to processor $P_i$ for all $i$, we have achieved relative order among the processors. We finish by performing the third stage, a sequential sort.

The method to divide the data is to repeat Quicksortlike partitioning steps as follows. Choose a partition element for all processors; at each processor, partition the data with respect to this element; compute the sum across all of the processors of the number of data elements less than the partition value; and if the sum is not the desired value, then choose the next partition element, using a binary search. The data at each processor is now partitioned into $N$ blocks where the $i^{th}$ block is destined for $P_i$.

If data elements are identical, some additional work may be necessary, since we must insure that the total number of data elements destined for any processor is exactly **NUMWORDS.** If we find that the total number of data elements less than our partition element is less than **NUMWORDS** and that the same total for the next greater partition element is larger than **NUMWORDS**, then we face a situation where the partition element is repeated. We then execute a clean-up phase of the first stage, where we hold an interprocessor conference to decide which repeats of the partition element should be placed in either side of the partition.

We now give pseudo-code for the first stage of the algorithm, beginning with code for the simple partitioning of the data. Let **A** be the array of data and **l**,**r**, and **p** integers such that we would like to partition **A [l . . r]** with respect to **p**. The following code partitions the data and returns the index to the rightmost element strictly less than the partition element **p**.

```
while (A[i]<p) and (i<j)
  increment i;
while (A[j]>=p) and (j>i)
  decrement j;
 if (i= j) then
  exit and return i;
repeat
  switch the values of A[i], A[j];
  repeat
    increment i;
  until (A[i] >=p);
  repeat
    decrement j;
  until (A [j] <p);
until (i<=j);
return i;
end.
```

We next give pseudo-code to find the partition point of all of the data such that the sum of the number of elements in the left partition is $M$, assuming that no partition element repeats. Let **A** be the array of data, **lowptr** and **highptr** indices into **A** such that we are partitioning **A[lowptr . . highptr]**, **plow** and **phigh** integer elements which are lower and upper bounds for the data. The code returns the partition element and the index to the right-most element strictly less than the partition element p. Temporary variables are integers **newp**, **p**, **total**, and **size**.

```
let newp = the average of all of
  the first, middle, and last
  elements of A [lowptr..highptr]
  on every processor;
repeat
  let p = newp;
  partition A[lowptr..highptr]
    with respect to p;
  let size = index of rightmost
    element less than p in
    A [lowptr..highptr];
  let total = size across
    all processors;
  if (total>M) then
    let phigh = p;
    let newp = (plow+phigh)/2;
  else if (total<M) then
    let plow = p;
    let newp = (plow+phigh)/2;
  if (newp<p) then
    let highptr=size-1;
  else
    let lowptr=size;
until total=M;
return size and partition element p;
end.
```

In order to find the $N$ partitions recursively, we have the following pseudo-code. Variables are as before. We begin this code with depth equal to log $N$, $M$ = **NUMWORDS** • $N$/2, offset= **NUMWORDS** • $N$/2, **low** = 0, **high** = **NUMWORDS**—1, and upper and lower bounds for the data **phigh** and **plow**.

```
find partition element p such that
  sum of number of elements in left
  partition is M;
let size = index of rightmost element
  less than p in A[low..high];
decrement depth;
if (depth>0) then
  recurse with low = low,
    high = size-l, plow = plow,
    phigh = p, depth = depth,
    M = M - offset/2, and
    offset = offset/2;
  recurse with low = size,
    high = high, plow = p,
    phigh = phigh, depth = depth,
    M = M + offset/2, and
    offset = offset/2;
end.
```

While the implementation of stage one is relatively straightforward, and the execution of stage three using Quicksort is efficient, the data movement in stage two is of particular interest for several reasons. Designing an efficient scheme which succeeds on any data set. is a hard problem, since pathological cases may occur in which some processors must send all of their data to others while other processors must spread their data among many others. Also, determining the correct sequence of interprocessor communications to use when sending data is a serious implementation issue. If the sequence is broken up into many communications among few processors, the total time will be great; if one assumes *N*—1 rounds during which each processor sends a portion of its data to another, there will be significant contention on the T3D network and there exists the possibility that there will be insufficient space on destination processors to receive data. Empirical evidence indicates that some communication permutations among processors are significantly better than others, although the time to send from one processor to any other with no contention has very little variation. Employing simple methods, such as having each processor send on round **i** to processor (**mype_num+i**) mod *N*, makes coordinating data movement easier, but we have found that some of these permutations perform worse than random, precisely because these very ordered permutations are ordered poorly. Since a T3D node contains two processors but shares a network connection, it can be helpful to send to processors that are not sending or are sending very little data, although this does not help us in this particular case. Similarly, having processors send only along one of the three T3D network axes can improve bandwith. It is an open question how to design *N*—1 rounds which maximizes T3D network bandwidth.

We implemented the data movement stage as follows. We assume that the data buffer is larger than **NUMWORDS** by a small amount which is proportional to the data size, and that the extra space lies at the left end of the buffer. Recall that each processor begins this phase with a contiguous sequence of variable-sized buckets of data, such that bucket *j* must go to processor $P_j$. First, for all *i*, $P_i$ shifts some of its data left so that the beginning of bucket 0 is at the beginning of the buffer and the extra space, called the hole, lies between buckets *i* and *i* + 1. Then we execute *N*—1 rounds l. *N*—1 where on round *j* each processor $P_k$ sends its data destined for $P_{(j+k)modN}$ to the hole on processor $P_{(j+k)modN,}$ thereby shifting the holes on each processor to the right and possibly changing its size. The processors continue in succession, wrapping around the buffer when necessary, and after completion of the rounds data is moved so that the data is contiguous. As long as the hole on each processor does not overflow at any round, this method of data movement will succeed.

The chance of success of this method varies with the distribution of the data and the amount of extra space in the buffer. The extra space on each processor must be at least **NUMWORDS**/*N*, and if the data is reasonably well distributed, this space can be smaller than 3 • **NUMWORDS**/(2*N*), or about 2% of **NUMWORDS** when *N* = 64. The less distributed the data, the greater this space should be, and the space need be no larger than the size of the data buffer, in which case the algorithm is guaranteed to succeed.

In our implementation we compute whether or not the data movement method would succeed, and, if not, we apply the following randomization to the data. We first run through the data on each processor in groups of 12 words (three cache lines) and exchange these 12 words with a random contiguous preceding segment of 12 words. This randomly permutes the data at each processor. Then we divide the data into *N* segments of length **NUMWORDS**/N, such that segment *i* is sent to $P_i$ using a movement scheme much like the above. This movement, however, is guaranteed to succeed since the data blocks are of equal length. After this randomization, we repartition the data, using the partition elements from the previous trial, and now the data movement scheme will succeed with high probability since the distribution of the sizes of the partitions on each processor is close to uniform.

Partition Sort: Table of run times (MT)

| wpp     *N* | 4 | 8 | 16 | 32 | 64 | 128 |
|---|---|---|---|---|---|---|
| 1000 | .283 | .410 | .796 | 2.32 | 9.81 | 30.7 |
| 5000 | 1.57 | 1.91 | 2.49 | 4.46 | 13.6 | 87.3 |
| 10000 | 3.40 | 3.64 | 4.75 | 6.82 | 16.8 | 101. |
| 50000 | 19.5 | 21.1 | 22.7 | 28.0 | 39.0 | 128. |
| 100000 | 40.4 | 45.3 | 48.3 | 55.2 | 69.0 | 167. |
| 500000 | 240. | 253. | 264. | 290. | 319. | 442. |
| 703125 | 339. | 358. | 383. | 413. | 449. | 583. |
| 1000000 | 505. | 537. | 565. | 598. | 656. | 777. |
| 1250000 |  | 663. | 711. | 754. | 802. | 942. |
| 1406250 |  |  |  | 851. | 912. | 1,095. |

Partition Sort: Table of run times excluding initial sequential sort (MT)

| wpp     *N* | 4 | 8 | 16 | 32 | 64 | 128 |
|---|---|---|---|---|---|---|
| 1000 | .143 | .277 | .663 | 2.26 | 9.68 | 30.6 |
| 5000 | .587 | .920 | 1.52 | 3.50 | 12.7 | 86.3 |
| 10000 | 1.22 | 1.62 | 2.61 | 4.81 | 14.6 | 99.0 |
| 50000 | 5.49 | 8.30 | 10.1 | 15.0 | 26.3 | 115. |
| 100000 | 11.2 | 16.0 | 19.4 | 27.4 | 41.1 | 139. |
| 500000 | 61.1 | 84.4 | 98.7 | 124. | 156. | 279. |
| 703125 | 78.8 | 113. | 142. | 172. | 217. | 344. |
| 1000000 | 115. | 162. | 197. | 246. | 304. | 436. |
| 1250000 |  | 193. | 255. | 308. | 365. | 509. |
| 1406250 |  |  |  | 351. | 399. | 603. |

In all of the runs timed above, the randomization step was not needed; had it been, the running time would have increased by 30% to 40%. One should try to choose the size of the extra space so that this step is infrequent, and so for these runs we chose the extra space to be close to the 1.5/*N* factor for large data sizes and chose it larger for particularly small data sizes. The holes in the lower left corner of this table are due to the fact that extra space requirements grow larger as the number of processors decreases, and thus the memory of the machine was no longer large enough to hold both the data and the extra space. What is clear from the table is that a severe penalty is paid above 16 processors, and for small data sets the dominant factor in this penalty is the communication necessary to determine a partition value. At *N* = 128, it actually takes longer to determine the 128 partition values than to sort the data on the processor. As the number of processors grows, some improvement to the algorithm to find the *N*-tiles may be necessary. The partition-and-conference takes 41% of the total time (whereas Quicksort took 54%) for *N* = 64 with 1 million words/processor.

The times can be improved if the data is known to be of a particular type. For instance, by assuming relatively random data, we were able to modify our code so that we found $N$-tiles in 2/3 of the previous time. In the code which found the $N$-tile partition points, we replaced the binary search with a search which was biased more towards the initial guess at the outset and which used this bias less and less as the search progressed. The choice of methods at the beginning of such a search is of chief importance since the partition steps at the beginning, larger data sets, cost the most.

It is interesting to note that the proportion of repeats in the data can both increase and decrease the time to find the $N$-tiles. With a moderate number of repeats, the number of partitioning rounds is less than average, and very quickly the clean-up phase of stage one is entered, where the communication to determine exact $N$-tiles is relatively cheap compared to many more conference-and-partition steps. Of course, with a very large number of repeats performance degrades since several partitioning rounds are redundant (before flagging a repeat). Special code could certainly be written to take care of this case, but a different sort would be appropriate on such data.

### 5.2 *The Radix Sort.*

In order to permit the radix sort a larger data size, we incorporate two changes from the out-of-place radix sort. First, instead of separating the data into buckets at the beginning, we do a radix-exchange sort on the top log $N$ bits. This change releases us from the requirement of having extra buckets large enough to hold the data. The second change is to use the data movement phase from the preceding partition sort, modifying it to allow each processor to receive a different number of words.

The following tables show the run times of this version of the radix sort:

Radix Sort: Table of run times (MT)

| wpp $N$ | 4 | 8 | 16 | 32 | 64 | 128 |
|---|---|---|---|---|---|---|
| 1000 | .180 | .229 | .374 | .906 | 2.99 | 13.2 |
| 5000 | 1.30 | 1.42 | 1.65 | 2.28 | 4.56 | 15.1 |
| 10000 | 2.67 | 2.96 | 3.31 | 4.34 | 7.68 | 21.3 |
| 50000 | 14.3 | 15.4 | 16.5 | 18.6 | 22.9 | 36.9 |
| 100000 | 29.6 | 31.9 | 34.3 | 37.5 | 41.0 | 59.0 |
| 500000 | 157. | 168. | 180. | 191. | 205. | 236. |
| 703125 | 226. | 243. | 259. | 275. | 291. | 320. |
| 1000000 | 322. | 350. | 370. | 396. | 417. | 459. |
| 1250000 |  | 441. | 463. | 499. | 526. | 538. |
| 1406250 |  |  |  | 565. | 592. | 649. |

Radix Sort: Table of run times excluding initial sequential sort (MT)

| wpp $N$ | 4 | 8 | 16 | 32 | 64 | 128 |
|---|---|---|---|---|---|---|
| 1000 | .0696 | .115 | .254 | .789 | 2.87 | 13.1 |
| 5000 | .299 | .430 | .651 | 1.29 | 3.57 | 14.1 |
| 10000 | .615 | .859 | 1.26 | 2.19 | 5.53 | 19.2 |
| 50000 | 3.05 | 4.22 | 5.35 | 7.12 | 11.8 | 25.5 |
| 100000 | 6.27 | 8.68 | 10.9 | 13.99 | 17.7 | 35.7 |
| 500000 | 30.0 | 40.7 | 52.1 | 65.1 | 77.4 | 108. |
| 703125 | 43.2 | 60.3 | 75.3 | 92.4 | 108. | 136. |
| 1000000 | 58.3 | 84.3 | 106. | 132. | 154. | 198. |
| 1250000 |  | 105. | 131. | 162. | 190. | 239. |
| 1406250 |  |  |  | 183. | 214. | 269. |

Again, as in the out-of-place radix sort and the preceding partition sort, the cost to use $N = 128$ processors becomes relatively expensive as the interprocessor conference prior to the data movement phase involves more and more processors which experience significant network contention. Comparing these tables with those for the out-of-place radix sort, we notice that the out-of-place version performs better on every data size (for which it is capable) and number of processors save 10,000 words on four processors. The out-of-place version loses because it is separating the data on each processor into four buckets, thus reading and writing every data word, while the present version does only a 2-bit radix exchange, reading every word but having to write only a fraction. We could envision a special sort for small data sizes on a small number of processors, employing the 1- or 2-bit radix exchange from the present sort and the data movement scheme from the out-of-place, which would beat both, but the utility of such a sort seems limited.

### 5.3 *The Hash-and-Chain Sort.*

The hash-and-chain sort operates on the principle that if one knows the approximate position of a data element in the final sorted list, it should be placed there. On very uniform data this idea improves on the radix algorithms mentioned above, although for peak performance this algorithm requires a larger amount of extra space. This sort is about three times as fast as the Batcher sort and is faster than any other sort for virtually any data size.

The basic strategy of this algorithm is similar to that of the radix sort described above; we separate the data by destination processor, send it in several rounds, and finish with a sequential sort. The separation phase is slightly different than that of the radix sort. We first separate the data into buckets on the top log $N$ bits as follows. We declare the data buffer to be $N$ contiguous buckets of equal size. Then we run through the data, picking up each data word and deciding its target buffer. We exchange our element with an unbucketed element in the target buffer, and repeat.

After the data is bucketed, we then employ the data movement scheme used for the out-of-place radix sort. We have processor $P_i$ save its $i^{th}$ bucket to a separate array and then execute 1 . . $N$—1 rounds, where on round $k$ processor $P_i$ sends its $(i + k)^{th}$ bucket to the empty bucket on processor $P_{i+k}$. Then each processor collapses its data from the separate array and the $N$—1 buckets into a contiguous segment.

Now we are ready to describe the hash-and-chain sort. First we require some extra space at the end of the buffer. We run through the data, picking up each data word and calculating its approximate location in the sorted list by masking off the top log $N$ bits and performing a floating-point multiply. If this target location is empty, we place the hashed element there. If this target location contains a data element which has not yet been hashed, we exchange it with our hashed element. If this target location contains a data element which has already been hashed, we execute a forward insertion sort to place our element in order among a short segment of hashed elements.

There are several strategies to determine whether or not a location is empty, contains an unhashed data element, or contains a hashed data element. One is to use a flag value for the empty spaces, where a flag value is some value which no data element can take, and to use a spare bit in the data word to indicate that the data word has already been hashed, but we wished to allow more general data than 63-bit data with a flag value. We can eliminate these requirements for the multiprocessor sort by adopting a different strategy. Since this sequential sort occurs at the end of the sorting algorithm, we are guaranteed that the top log *N* bits of each data word on a given processor are the same, so we are free to use two of them as flags as long as *N* is required to be at least 4.

We give pseudo-code for the final sequential sort as follows. Let **A[O . . NUMWORDS-1+EXTRA]** be an array of integers, of which the first **NUM** words are data. (**NUM** might not be **NUMWORDS**, due to the fact that not every processor has the same amount of data at this stage.) **FLAGVAL** is some value which is not identical to any data element and which is recognized as not being a hashed value. For us, **FLAGVAL** can be chosen to be any data word with the second high-order bit changed, since we change the high-order bit indicate having been hashed. Let *i, j, k, 1, t* be integers.

```
initialize A[NUM..NUMWORDS-1+EXTRA
  to FLAGVAL;
let i = 0;
while i<NUM do
  if A[i] has not been hashed then
    let j = A[i], a[i] = FLAGVAL;
    repeat
      let 1 be the result of
        masking off the top log
        N bits of j and multiplying
        by 1/NUM;
      if A[l] = FLAGVAL then the
        location is empty, so
        do
          let t = FLAGVAL;
          let A[l] = j with the
            hashed bit set;
      else the location contains data,
        so do
          let k = j with the hashed
            bit set;
      while (A[l] has been hashed)
        and (k > A[l]), continue
        insertion-sorting:
          let t = A[l];
          increment l;
      let A[l] = k;
      increment l;
      if A[l] has been hashed then
        while A[l] has been hashed
          switch A[l] and t;
          increment l;
        switch A[l] and t;
      let j = t;
    until (t is not FLAGVAL) and
      (j does not have the hashed
       bit set);
  increment i;
```

```
let i = 0, j = 0;
while i<NUM do
  if A[j] is not FLAGVAL then
    let A[i] = A[j];
    increment i;
  increment j;
end.
```

We found that on the T3D the extra space at the end of the data buffer needed to be roughly **NUMWORDS**/4 if the method was to perform well on random data. Therefore this sort should only be considered a 4/5 memory sort, while the radix sort presented above can, for large N, approach the full memory size, needing only extra space of size 3 • **NUMWORDS**/(2*N*). If space is available, the extra space should be closer to one-half of the initial data size.

In the tables, we show times when the extra buffer size is chosen fairly carefully. When space allows, such as on data sizes up to 50,000 words, we use an extra buffer 1/2 as large; on higher data sizes, we are forced to cut this buffer down. For 1,250.000 words we were particularly constrained, using an extra buffer only 10% as large, and the performance certainly degrades. We do not give times in the lower left corner because either the date movement phase, requiring a separate buffer of size **NUMWORDS**/*N*, cannot be allocated, or the extra buffer for the sequential sort must be so small that the run time increases dramatically. For optimal buffer sizes, the time for a fixed number of processors is virtually linear. For a fixed data size, the initial bucketing increases only slightly with more processors; the chief increase as more processors are used is due to the network contention. More packets of smaller sizes are sent between more processors, and this degrades performance significantly from 64 to 128 processors.

Hash-and-Chain Sort: Table of run times (MT)

| wpp *N* | 4 | 8 | 16 | 32 | 64 | 128 |
|---|---|---|---|---|---|---|
| 1000 | .210 | .221 | .244 | .270 | .327 | .599 |
| 5000 | 1.22 | 1.23 | 1.29 | 1.35 | 1.42 | 1.95 |
| 10000 | 2.46 | 2.59 | 2.64 | 2.72 | 2.79 | 3.00 |
| 50000 | 12.7 | 13.2 | 13.4 | 13.7 | 14.0 | 14.8 |
| 100000 | 25.5 | 26.2 | 26.8 | 27.6 | 27.9 | 29.5 |
| 500000 | 127. | 131. | 134. | 137. | 139. | 147. |
| 703125 | 180. | 185. | 189. | 193. | 196. | 207. |
| 1000000 |  | 268. | 271. | 278. | 282. | 303. |
| 1250000 |  |  | 385. | 392. | 397. | 406. |

Hash-and-Chain Sort: Table of run times excluding initial sequential sort (MT)

| wpp *N* | 4 | 8 | 16 | 32 | 64 | 128 |
|---|---|---|---|---|---|---|
| 1000 | .0845 | .0963 | .112 | .139 | .179 | .291 |
| 5000 | .367 | .408 | .445 | .512 | .558 | .684 |
| 10000 | .706 | .820 | .894 | .976 | 1.05 | 1.29 |
| 50000 | 3.50 | 4.04 | 4.31 | 4.72 | 4.93 | 5.78 |
| 100000 | 7.00 | 7.96 | 8.65 | 9.41 | 9.74 | 11.3 |
| 500000 | 34.8 | 39.5 | 42.3 | 46.2 | 48.0 | 55.8 |
| 703125 | 49.5 | 55.6 | 60.0 | 65.6 | 68.2 | 79.2 |
| 1000000 |  | 79.5 | 86.0 | 93.3 | 97.4 | 117. |
| 1250000 |  |  | 108. | 118. | 125. | 150. |

We also implemented a special case version of this sort, where the size of the data buffer was a power of two and the data movement could be achieved out-of-place. At its best, with the data uniformly distributed among the positive integers less than a power of 2, this version sorted in 80% of the time of the sort above. For *N* = 64 and **NUMWORDS** = 400,000 the run time was about 90 million ticks, or about 0.6 seconds. Although this sort time is impressive, we did not pursue this sort further because of its lack of generality.

## 6 CONCLUSIONS

We begin by giving several tables for various sizes and types of data which indicate, for a given number of processors, the quickest of the sorts considered in this paper. For arbitrary data, the Batcher mergesort and the partition sort are the only options, and we note that for more than half of the memory, partition sorting is the only choice. For data which is uniform in the top log *N* bits, we may choose between Batcher mergesorting, partition sorting, out-of-place radix sorting, or almost in-place radix sorting. Finally, for data which is uniform in more than the top log *N* bits, we may choose among any of the methods discussed in this paper. Note, however, that the hash-and-chain sort cannot sort as much data as the radix sort.

The relative speeds of Batcher mergesorting, partition sorting, radix sorting, and hash-and chain sorting on a CRAY T3D are similar to that of other machines; we found that those sorts which take advantage of assumptions about the uniformity and distribution of the data perform better. Given the parallel nature of the ÇRAY T3D, we expected the efficiency of sorting data in parallel also to depend significantly on the degree to which processors communicate or send data using network connections. While this dependence was certainly apparent, the penalty for network communication was not as great as expected. For instance, we compared the proportion of time spend sequentially sorting for each of the sorts, given 64 processors sorting roughly half the memory. While for the Batcher mergesort the initial sequential sort comprised 40% of the total time, the bulk of the remainder was spent merging, not waiting for data; the partition sort devoted 50% of the time to sequential sorting and only 4% to large block data moves. Similarly, the almost in-place radix sort devoted 63% of the time to a sequential radix sort and 28% to partitioning; the hash-and-chain sort used 66% of the time for sequential sort; and the out-of-place radix sort spend 72% of the time sequentially sorting. The penalty for network contention occurred primarily in the partition sort, which required frequent interprocessor conferencing, and in simultaneous data movement among more than 64 processors. We conclude that the memory operations on the DEC Alpha chips are sufficiently slow, compared to vector processors, that the network contention does not pose a significant burden.

The CRAY T3D can currently be equipped with as many as 1024 processors, with up to 8 MW of memory per processor. While we believe that increased memory will not significantly change the relative performance of these sorts, we suspect that those sorts which have extensive interprocessor communication will have dramatically increased total time per word sorted. Indeed, on large data sizes the behavior of the partition sort already degrades between *N* = 64 and *N* = 128. In contrast, the radix sorts should perform well even for large *N*.

| Sort choices between mergesorting (M) and partition-sorting (P) for almost all data sets | | | | | | |
|---|---|---|---|---|---|---|
| Table of run times excluding initial sequential sort (MT) | | | | | | |
| wpp *N* | 4 | 8 | 16 | 32 | 64 | 128 |
| 1000 | M | M | M | M | M | M |
| 5000 | M | P | P | M | M | M |
| 10000 | M | P | P | M | M | M |
| 50000 | M | P | P | P | P | M |
| 100000 | M | P | P | P | P | M |
| 500000 | M | M | P | P | P | P |
| 703125 | M | P | P | P | P | P |
| 1000000 | P | P | P | P | P | P |
| 1250000 | none | P | P | P | P | P |
| 1406250 | none | none | none | P | P | P |

| Sort choices between mergesorting (M), partition-sorting (P), and almost in-place radix sorting (R2), for data uniform in the top log *N* bits | | | | | | |
|---|---|---|---|---|---|---|
| Table of run times excluding initial sequential sort (MT) | | | | | | |
| wpp *N* | 4 | 8 | 16 | 32 | 64 | 128 |
| 1000 | M | P | R1 | R1 | R1 | R1 |
| 5000 | R1 | R1 | R1 | R1 | R1 | R1 |
| 10000 | R2 | R1 | R1 | R1 | R1 | R1 |
| 50000 | R1 | R1 | R1 | R1 | R1 | R1 |
| 100000 | R1 | R1 | R1 | R1 | R1 | R1 |
| 500000 | R1 | R1 | R1 | R1 | R1 | R1 |
| 703125 | R1 | R1 | R1 | R1 | R1 | R1 |
| 1000000 | R2 | R2 | R2 | R2 | R2 | R2 |
| 1250000 | none | R2 | R2 | R2 | R2 | R2 |
| 1406250 | none | none | R2 | R2 | R2 | R2 |

| Sort choices between mergesorting (M), partition-sorting (P), out-of-place radix sorting (R1), almost in-place radix sorting (R2), and hash-and-chain sorting (H), for very uniform data | | | | | | |
|---|---|---|---|---|---|---|
| Table of run times excluding initial sequential sort (MT) | | | | | | |
| wpp *N* | 4 | 8 | 16 | 32 | 64 | 128 |
| 1000 | H | H | H | H | H | H |
| 5000 | H | H | H | H | H | H |
| 10000 | H | H | H | H | H | H |
| 50000 | H | H | H | H | H | H |
| 100000 | H | H | H | H | H | H |
| 500000 | H | H | H | H | H | H |
| 703125 | H | H | H | H | H | H |
| 1000000 | R2 | H | H | H | H | H |
| 1250000 | none | R2 | H | H | H | H |
| 1406250 | none | none | R2 | R2 | R2 | R2 |

The paper [TS] gives some sorting benchmarks for a Thinking Machines CM-5. They use a 1024 processor machine and can sort 1 billion 32 bit keys (1 million keys per processor) using a radix sort in about 17 seconds. We did not have such a large machine to use for our experiments, unfortunately, because a direct comparison would certainly be interesting. Their results are virtually the same for 64 processors, where their sort still takes about 17 seconds. This compares to less than two seconds for our hash-and-chain method.

We finish this paper with a rough comparison of the of the sorting performance of a T3D and a CRAY C90. For uniform data, one C90 head performs as well as between 8 and 16 processors of a T3D, while for general data one C90 head performs as well as between four and eight processors of a T3D. When the C90 is multitasked, four heads are comparable to 32 processors of a T3D.

Comparison of C90 radix sorting versus T3D radix (R) and hash-and-chain (H) sorting, for general data, in seconds

| Data size | C90 | | T3D | | | |
|---|---|---|---|---|---|---|
| | 1 head | 4 heads | 8 procs | 16 procs | 32 procs | 64 procs |
| 1,000,000 | .162 | .059 | R: .263 | .124 | .0621 | .0314 |
| | | | H: .217 | .111 | .0563 | .0288 |
| 10,000,000 | 1.26 | .583 | R: 2.92 | 1.35 | .648 | .327 |
| | | | H: | 1.10 | .570 | .287 |

Comparison of C90 Quicksorting versus T3D Batcher mergesorting (M) and partition sorting (P), for general data, in seconds

| Data size | C90 | T3D | |
|---|---|---|---|
| | | 8 procs | 16 procs |
| 1,000,000 | .454 | M: .420 | .226 |
| | | P: .419 | .199 |
| 10,000,000 | 4.8 | M: | 2.58 |
| | | P: 4.39 | 2.27 |

## REFERENCES

[AKS] M. Ajtai, J Komlos and E. Szmeredi, *An O* (n log n) *sorting network,* Proceedings of the Fifteenth Annual ACM Symposium on Theory of Computing (April 1983), 1-9.

[B] G. E. Blelloch, *Vector Models for Data-Parallel Computing* (1990), The MIT press.

[BLMPSZ] G. Blelloch, C. Leiserson, B. Maggs, G. Plaxton, S. Smith, and M. Zagha, *A Comparison of Sorting Algorithms for thc Connection Machine CM-2,* Proceedings of the 3rd Annual ACM Symposium on Parallel Algorithms and Architectures (July 1991).

[C] R. Cole, *Parallel merge sort,* SIAM Journal on Computing (1988), 770–785.

[K] D. Knuth, *The Art of Computer Programming,*vol. 3, Searching and Sorting, Addison-Wesley, Reading, MA, 1973.

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

[TS] K. Thearling and S. Smith, *An Improved Supercomputer Sorting Benchmark,* Proceedings of Supercomputing '92 (November 1992), ACM press, 14–19.