1

An Experimental Analysis of Parallel Sorting Algorithms

Guy E. Blelloch
Carnegie Mellon University
Pittsburgh, PA 15213

Charles E. Leiserson
MIT
Cambridge, MA 02139

Bruce M. Maggs
Carnegie Mellon University
Pittsburgh, PA 15213

C. Greg Plaxton
University of Texas
Austin, TX 78712

Stephen J. Smith
Pilot Software
Cambridge, MA 02141

Marco Zagha
Silicon Graphics
Mountain View, CA 94043

Abstract

We have developed a methodology for predicting the performance of parallel algorithms on real parallel machines. The methodology consists of two steps. First, we characterize a machine by enumerating the primitive operations that it is capable of performing along with the cost of each operation. Next, we analyze an algorithm by making a precise count of the number of times the algorithm performs each type of operation. We have used this methodology to evaluate many of the parallel sorting algorithms proposed in the literature. Of these, we selected the three most promising, Batcher's bitonic sort, a parallel radix sort, and a sample sort similar to Reif and Valiant's flashsort, and implemented them on the Connection Machine model CM-2. This paper analyzes the three algorithms in detail and discusses the issues that led us to our particular implementations. On the CM-2 the predicted performance of the algorithms closely matches the observed performance, and hence our methodology can be used to tune the algorithms for optimal performance. Although our programs were designed for the CM-2, our conclusions about the merits of the three algorithms apply to other parallel machines as well.

1 Introduction

Sorting is arguably the most studied problem in computer science, both because it is used as a substep in many applications and because it is a simple combinatorial problem with many interesting and diverse solutions. Sorting is also an important benchmark for parallel supercomputers. It requires significant communication bandwidth among processors, unlike many other supercomputer benchmarks, and the most efficient sorting algorithms communicate data in irregular patterns.

Parallel algorithms for sorting have been studied since at least the 1960's. An early advance in parallel sorting came in 1968 when Batcher discovered the elegant $\Theta(\lg^2 n)$-depth *bitonic sorting network* [3]. (Throughout this paper $\lg n$ denotes $\log_2 n$.) For certain families of fixed interconnection networks, such as the hypercube and shuffle-exchange, Batcher's bitonic

*This research was supported in part by Thinking Machines Corporation, and in part by the Defense Advanced Research Projects Agency under Contracts N00014-87-0825 and F33615-90-C-1465.

CM-2, Connection Machine, CM, *Lisp, Paris, and CMIS are trademarks of Thinking Machines Corporation.

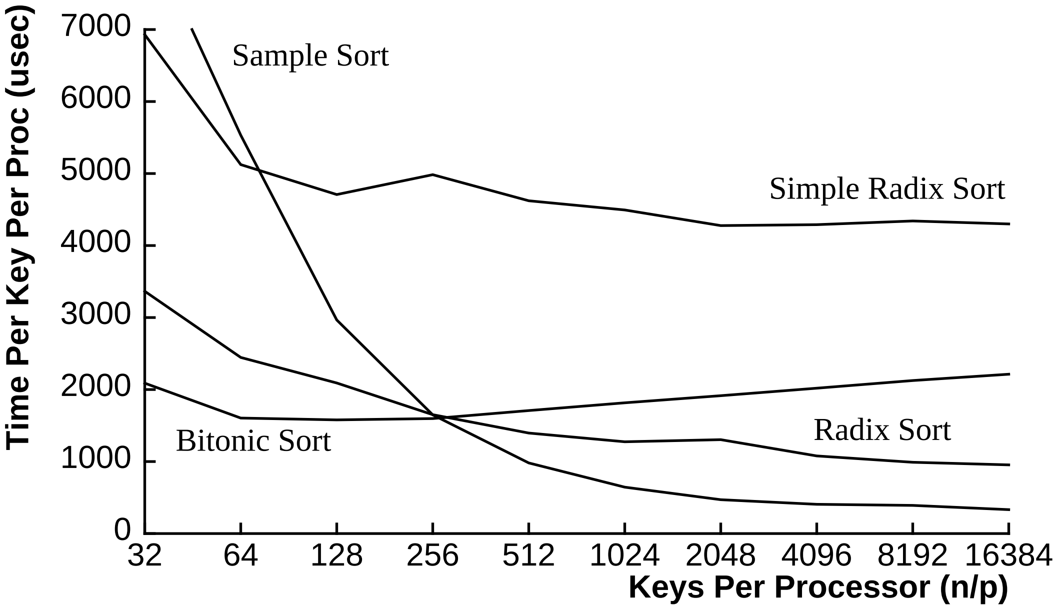


Figure 1: Actual running times for sorting 64-bit keys on a 32K Connection Machine CM-2. In the figure, the running times are divided by the number of keys per processor to permit extrapolation to machines with different numbers of processors. The term *processor*, as used in this paper, is a 32-bit wide so-called *Sprint* node, of which there are $p = 1024$ in a 32K CM-2. To determine the total running time of a sort involving n keys, multiply the time per key per processor in the figure by n/p.

sorting technique provides a parallel algorithm for sorting n numbers in $\Theta(\lg^2 n)$ time with n processors. The question of the existence of a $o(\lg^2 n)$-depth sorting network remained open until 1983, when Ajtai, Komlós, and Szemerédi [1] provided an optimal $\Theta(\lg n)$-depth sorting network, but unfortunately, their construction leads to larger networks than those given by bitonic sort for all "practical" values of n. Leighton [15] has shown that any $\Theta(\lg n)$-depth family of sorting networks can be used to sort n numbers in $\Theta(\lg n)$ time in the n-node bounded-degree fixed-connection network domain. Not surprisingly, the optimal $\Theta(\lg n)$-time n-node fixed-connection sorting networks implied by the AKS construction are also impractical.

In 1983, Reif and Valiant proposed a more practical $O(\lg n)$-time randomized algorithm for sorting [19], called *flashsort*. Many other parallel sorting algorithms have been proposed in the literature, including parallel versions of *radix sort* and *quicksort* [5], a variant of quicksort called hyperquicksort [23], smoothsort [18], column sort [15], Nassimi and Sahni's sort [17], and parallel merge sort [6].

This paper reports the findings of a project undertaken at Thinking Machines Corporation to develop a fast sorting algorithm for the Connection Machine Supercomputer model CM-2. The primary goals of this project were

1. to implement as fast a sorting algorithm as possible for integers and floating-point numbers on the CM-2,

2. to generate a library sort for the CM-2 (here we were concerned with memory use, stability, and performance over a wide range of problem and key sizes in addition to

running time),

3. and to gain insight into practical sorting algorithms in general.

Our first step towards achieving these goals was to analyze and evaluate many of the parallel sorting algorithms that have been proposed in the literature. After analyzing many algorithms, we selected the three most promising alternatives for implementation: bitonic sort, radix sort, and sample sort. Figure 1 compares the running times of these three algorithms (two versions of radix sort are shown). As is apparent from the figure, when the number of keys per processor (n/p) is large, sample sort is the fastest sorting algorithm. On the other hand, radix sort performs reasonably well over the entire range of n/p, and it is deterministic, much simpler to code, stable, and faster with small keys. Although bitonic sort is the slowest of the three sorts when n/p is large, it is more space-efficient than the other two algorithms, and represents the fastest alternative when n/p is small. Based on various pragmatic issues, the radix sort was selected to be used as the library sort for Fortran now available on the CM-2.

We have modeled the running times of our sorts using equations based on problem size, number of processors, and a set of machine parameters (e.g., time for point-to-point communication, hypercube communication, scans, and local operations). These equations serve several purposes. First, they make it easy to analyze how much time is spent in various parts of the algorithms. For example one can quickly determine the ratio of computation to communication for each algorithm and see how this is affected by problem and machine size. Second, they make it easy to generate good estimates of running times on variations of the algorithms without having to implement them. Third, one can determine how various improvements in the architecture would improve the the running times of the algorithms. For example, the equations make it easy to determine the effect of doubling the performance of message routing. Fourth, in the case of radix sort we are able to use the equations to analytically determine the best radix size as a function of the problem size. Finally, the equations allow anyone to make reasonable estimates of the running times of the algorithms on other machines. For example, the radix sort has been implemented and analyzed on the Cray Y-MP [25], and the CM-5 [22], which differ significantly from the CM-2. In both cases, when appropriate values for the machine parameters are used, our equations accurately predicted the running times. Similar equations are used by Stricker [21] to analyze the running time of bitonic sort on the iWarp, and by Hightower, Prins and Reif [12] to analyze the running time of flashsort on the Maspar MP-1.

The remainder of this paper studies the implementations of bitonic sort, radix sort and sample sort. In each case, it describes and analyzes the basic algorithm, as well as any enhancements and/or minor modifications that we introduced to optimize performance. After describing the primitive operations that our algorithms use in Section 2, Sections 3, 4, and 5 present our studies of bitonic sort, radix sort, and sample sort, respectively. In Section 6, we compare the relative performance of these three sorts, not only in terms of running time, but also with respect to such criteria as stability and space. Appendix A presents a brief analysis of other algorithms that we considered for implementation. Appendix B presents a probabilistic analysis of the sampling procedure used in our sample sort algorithm.

2 Primitive Operations

This section describes a set of primitive parallel operations that can be used to implement a parallel sorting algorithm. These operations are likely to be found on any parallel computer, with the possible exception of the Cube Swap operation, which applies only to hypercube-based machines. All of the algorithms in this paper are described in terms of these operations. Although the costs of the operations are given only for the CM-2, our analysis can be applied to other machines by substituting the appropriate costs.

There are four classes of operations:

- *Arithmetic*: A local arithmetic or logical operation on each processor. Also included are global operations involving a front end machine and the processors, such as broadcasting a word from the front end to all processors.

- *Cube Swap*: Each processor sends and receives one message across each of the dimensions of the hypercube.

- *Send*: Each processor sends one message to any other processor through a routing network. In this paper we use two types of sends: a *single-destination send*, and a *send-to-queue*. In the single-destination send (used in our radix-sort) messages are sent to a particular address within a particular processor, and no messages may have the same destination. In the send-to-queue (used in our sample sort) messages are sent to a particular processor and are placed in a queue in the order they are received.

- *Scan*: A parallel-prefix (or suffix) computation on integers, one per processor. Scans operate on a vector of input values using an associative binary operator such as integer addition. (The only operator employed by our algorithms is addition.) As output, the scan returns a vector in which each position has the "sum," according to the operator, of those input values in lesser positions. For example, a plus-scan (integer addition as the operator) of the vector

  $$[\,4 \quad 7 \quad 1 \quad 0 \quad 5 \quad 2 \quad 6 \quad 4 \quad 8 \quad 1 \quad 9 \quad 5\,]$$

  yields

  $$[\,0 \quad 4 \quad 11 \quad 12 \quad 12 \quad 17 \quad 19 \quad 25 \quad 29 \quad 37 \quad 38 \quad 47\,]$$

  as the result of the scan.

In this paper, we describe our algorithms in English, and, where more precision is required, in a parallel vector pseudocode. We generally assume that the variable n refers to the number of keys to be sorted, and that p is the number of processors in the machine.

In the parallel vector pseudocode, we assume that data is stored in two kinds of variables: n-element vectors and scalars. Vectors are identified by capitalized variable names, whereas scalar variable names are uncapitalized. Parallel arithmetic operations on vectors are performed in an elementwise fashion. The special vector *Self* refers to the vector of coordinate addresses ($\mathit{Self}[i] = i$). Cube swaps along one dimension of the hypercube are performed on a vector V

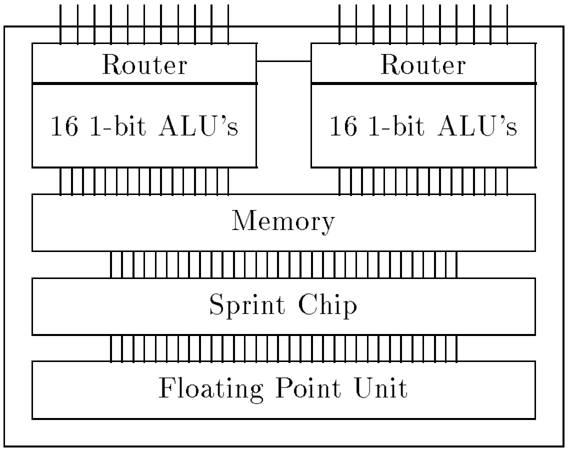


Figure 2: The organization of a CM-2 Sprint node.

by an operation CUBE-SWAP(V, j), which returns a vector whose ith coordinate is $V[i + 2^j]$ if the jth bit of i (in binary) is 0, and $V[i - 2^j]$ if the jth bit of i is 1. Cube swaps along all dimensions simultaneously are described in English. A single-destination send is accomplished by the operation SEND(V, Dest), which returns the vector whose ith coordinate is that $V[j]$ such that $\mathit{Dest}[j] = i$. Scan operations are performed by a procedure SCAN(V) that returns the plus-scan of the vector.

2.1 Primitive Operations on the CM-2

The CM-2 is a single-instruction multiple-data (SIMD) computer. In its full 64K-processor configuration, it can be viewed as 2048 (2^{11}) *Sprint* nodes configured as an 11-dimensional hypercube. (A d-dimensional hypercube is a network with 2^d nodes in which each node has a distinct d-bit label, and two nodes are neighbors if their labels differ in precisely on bit position.) The Sprint nodes have *multiport* capability: all dimensions of the hypercube can be used at the same time. The Sprint nodes are controlled by a *front-end processor* (typically a Sun4 or Vax). Figure 2 illustrates the organization of a Sprint node, which consists of the following chips:

- Two processor chips, each containing 16 1-bit processors, a 1-bit bidirectional wire to each of up to 11 neighboring nodes in the hypercube, and hardware for routing support.
- Ten DRAM chips, containing a total of between 256K bytes and 4M bytes of error-corrected memory, depending on the configuration. All recent machines contain at least 1M bytes of memory per node.
- A floating-point chip (FPU) capable of 32-bit and 64-bit floating-point arithmetic, as well as 32-bit integer arithmetic.
- A *Sprint* chip that serves as an interface between the memory and the floating-point chip. The Sprint chip contains 128 32-bit registers and has the capability to convert data from

Operation	Symbolic Time	Actual Time
Arithmetic	$A \cdot (n/p)$	$1 \cdot (n/1024)$
Cube Swap	$Q \cdot (n/p)$	$40 \cdot (n/1024)$
Send (routing)	$R \cdot (n/p)$	$130 \cdot (n/1024)$
Scan (parallel prefix)	$3A \cdot (n/p) + S$	$3 \cdot (n/1024) + 50$

Figure 3: The time required for operations on a 32K Connection Machine CM-2. The value p is the number of processors (Sprint nodes), and n is the total number of elements being operated on. All operations are on 64-bit words, except for scans which are on 32-bit words. All times are in microseconds.

the bit-serial format used by the 1-bit processors to the 32-bit word format used by the floating-point chip.

In this paper, we view each Sprint node as a single processor, rather than considering each of the 64K 1-bit processors on a fully configured CM-2 as separate processors. This point of view makes it easier to extrapolate our results on the CM-2 to other hypercube machines, which typically have 32 or 64-bit processors. Furthermore, it is closer to the way in which we viewed the machine when implementing the sorting algorithms. Our programs for the CM-2 were written in Connection Machine assembly language (Paris) and high-level microcode (CMIS).

Figure 3 gives estimated running times for each of the four classes of primitives on a 32K CM-2. We assume that each of $p = 1024$ processors contains n/p elements, for a total of n elements. Times are given for 64-bit data, except for scans, which operate on 32-bit data. With respect to the operation times, we have generally simplified our expressions by ignoring fixed overheads whenever they are small, concentrating instead on throughput. (For scans, the fixed overhead is substantial, so we have included it explicitly.) Because of these simplifications, our analyses do not accurately model performance when the number of elements per processor is small. When n/p is large, however, they are accurate to within approximately 10 percent. Since most data on the CM-2 originates in the 1-bit processors, n/p is typically at least 32. As a practical matter, most sorting applications involve $n/p \geq 128$, and often, $n/p = 2048$ or much larger.

We now discuss in somewhat more detail the time estimates for each of the classes of operations.

The time A for arithmetic operations is nominally chosen to be 1 microsecond. For example, the cost of summing two integer values, including the costs of loading and storing the data into local memory and of incrementing a counter (assuming the operation is in a loop) is about $1.4A$. An indirect access in which different processors access potentially different memory locations requires about $3A$ time. Also, computing the maximum (or minimum) of two values require about $3A$ time, since these operations involve a compare followed by a conditional memory move. Throughout the paper, the coefficients of A were obtained empirically. For radix sort and sample sort, we instrumented the code with calls to a real-time clock which provides accurate and deterministic timings. The constant coefficient reported for bitonic merge was determined by fitting a curve to the timing data.

On the CM-2, the time Q for cube swapping is the same whether a processor sends one message across just one dimension of the hypercube or 11 messages each across one of the 11 dimensions of the hypercube. To fully exploit the communication bandwidth provided by the hypercube, it is desirable, of course, to use all dimensions simultaneously.

The time R given for a send is based on routing messages randomly, where each message is equally likely to go to any other processor. The time for the two types of sends (i.e., single-destination and send-to-queue) is approximately the same as long as in the send-to-queue the number of messages received at each processor is approximately balanced. Some variation in the time for a send occurs because some routing patterns take longer than others. As long as there is no congestion at the receiving processor, however, no known pattern takes longer than $2R$. If each processor is receiving approximately the same number of messages, congestion can be avoided by injecting messages into the router in a psuedorandom order. The CM-2 also supports combining sends.

Consider the cost of a single scan operation on the CM-2 when the number of elements per processor is large. In this case, the running time is only about $3A \cdot (n/p)$, since the fixed overhead S can be safely ignored. In the case of multiple independent scans (each on one element per processor), however, the fixed overhead S must be taken into consideration. The other operations (Cube Swap and Send) have fixed overheads as well, but they are negligible by comparison.

3 Batcher's Bitonic Sort

Batcher's bitonic sort [3] is a parallel merge sort that is based upon an efficient technique for merging so-called "bitonic" sequences. A bitonic sequence is one that increases monotonically and then decreases monotonically, or can be circularly shifted to become so. One of the earliest sorts, bitonic sort was considered to be the most practical parallel sorting algorithm for many years. The theoretical running time of the sort is $\Theta(\lg^2 n)$, where the constant hidden by Θ is small. Moreover, bitonic sort makes use of a simple fixed communication pattern that maps directly to the edges of the hypercube; a general routing primitive need not be invoked when bitonic sort is implemented on the hypercube.

In this section, we discuss our implementation of bitonic sort. The basic algorithm runs efficiently on a hypercube architecture, but uses only one dimension of the hypercube wires at a time. The CM-2 hypercube has multiport capability, however, and by pipelining the algorithm, it is possible to make efficient use of all hypercube wires at once. This optimization results in a 5-fold speedup of the communication and over a 2-fold speedup in the total running time of the algorithm. Even with this optimization, the other two algorithms that we implemented outperform bitonic sort when the number n/p of keys per processor is large. When n/p is small, however, bitonic sort is the fastest of the three, and uses considerably less space than the other two.

Figure 4 illustrates the bitonic sort algorithm. The key step is an operation called a *bitonic merge*. The inputs to this operation are a pair of sequences that are sorted in opposite directions, one in ascending order and the other in descending order, so that together they form a bitonic sequence. Bitonic merge takes this bitonic sequence and from it forms a single sorted sequence. (In fact, bitonic merge will form a sorted sequence from any bitonic sequence.) For the moment, let us assume that we have n input keys to be sorted and that we have $p = n$ processors, each with 1 key. For each integer $d = 1, \ldots, \lg n$, the algorithm performs $n/2^d$

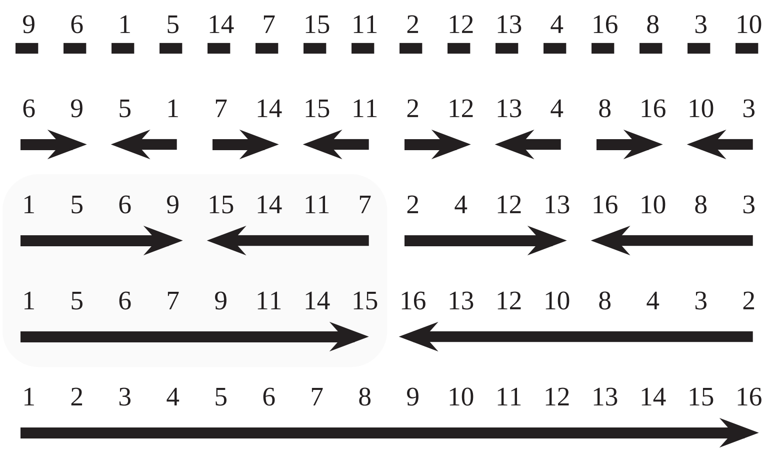


Figure 4: An illustration of the BITONIC-SORT procedure. Each arrow represents a sequence of keys sorted in the direction of the arrow. The unsorted input sequence of n keys is shown at the top, and the sorted output sequence is shown at the bottom. A bitonic merge operation is shaded. During the dth step of the algorithm (where $1 \le d \le \lg n$, $n/2^d$ merges are performed, each producing a sorted sequence of length 2^d from two sorted sequences of length 2^{d-1}.

merges, where each merge produces a sorted sequence of length 2^d from two sorted sequences of length 2^{d-1}.

The key step of bitonic sort is the merge operation, which is described by the following pseudocode.

BITONIC-MERGE(*Key*, d)
1 **for** $j \leftarrow d-1$ **downto** 0
2   **do** *Opposite* $\leftarrow$ CUBE-SWAP(*Key*, j)
3   **if** $Self\langle j\rangle \oplus Self\langle d\rangle$
4   **then** *Key* $\leftarrow$ min(*Key*, *Opposite*)
5   **else** *Key* $\leftarrow$ max(*Key*, *Opposite*)

In line 3, the operator $\oplus$ denotes the exclusive-or function, and the expression "$Self\langle j\rangle$" means the jth bit of the integer representing the position of the key in the input vector. $Self\langle 0\rangle$ is the least significant bit.

The operation of this algorithm can be understood with the help of Figure 5, which shows how two sorted sequences (i.e., a single bitonic sequence) are merged into a single ascending sequence. Each vertical line of the figure represents a processor in the hypercube, each of which initially contains one of the input keys. Time moves downward in the diagram, with the two sorted input sequences at the top, and the final single sorted sequence at the bottom. During a single step of the algorithm, all keys are communicated across a single dimension of the hypercube. After keys have been communicated across all the dimensions of the hypercube, the hypercube processors contain the output sorted in ascending order.

Each iteration of the loop in BITONIC-MERGE is represented by the collection of horizontal line segments in a shaded region of the figure. Each horizontal line segment represents the communication of keys between two processors along a hypercube wire, which corresponds to the CUBE-SWAP in line 2. In the algorithm, $Self\langle d\rangle$ tells whether we are producing an

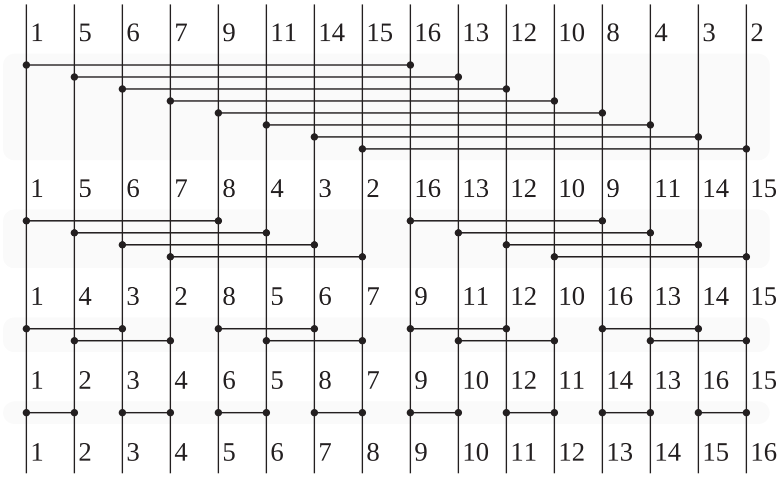


Figure 5: Viewing the BITONIC-MERGE procedure as a hypercube algorithm. Each vertical line represents a hypercube processor. Each horizontal line segment represents the communication of keys along the hypercube wire between two processors.

ascending (0) or descending (1) order, and $Self\langle j\rangle$ tells whether the processor is on the left (0) or right (1) side of a wire. For the example in the figure, we are sorting into ascending order ($Self\langle d\rangle = 0$), and thus for each pair of keys that are swapped, the smaller replaces the key in the processor on the left and the larger is kept on the right.

We shall not prove the correctness of this well-known algorithm; the interested reader is referred to [2, 7].

To this point, we have assumed that the number n of input keys is equal to the number p of processors. In practice, it is important for a sorting algorithm to be able to cope with unequal values of n and p. As it happens, the best hypercube algorithms to date use substantially different techniques for the cases $n \ll p$ and $n \gg p$. This project focuses entirely on the development of sorting algorithms for the more frequently occurring case when $n \geq p$.

To handle multiple keys per processor, we view each key address as being composed of a processor address (high-order bits corresponding to "physical" hypercube dimensions) and an index within the processor (low-order bits corresponding to "virtual" hypercube dimensions). In a bitonic merge, communication occurs across successive dimensions, in descending order. Across any physical dimension, this communication is realized by a set of n/p cube swaps. After processing the physical dimensions, what remains to be performed amounts to a bitonic merge within each processor. Given n keys and p processors, the CM-2 time for the bitonic merge becomes

$$T_{\text{merge}} = \begin{cases} (n/p)\cdot 5A\cdot d & \text{if} \quad d \leq \lg(n/p) \ , \\ (n/p)\cdot(Q\cdot(d-\lg(n/p)) + 5A\cdot d) & \text{if} \quad d > \lg(n/p) \ ; \end{cases}$$

where the coefficient 5 was determined empirically by fitting to the data. (If $d \leq \lg(n/p)$, then the bitonic merges occur entirely within processors, and so the coefficient of Q is 0.)

The bitonic sort algorithm calls the BITONIC-MERGE subroutine once for each dimension.

Bitonic-Sort(Key, n)

for $d \leftarrow 1$ **to** $\lg n$
    do Bitonic-Merge(Key, d)

The time taken by the algorithm is

$$
\begin{aligned}
T_{\text{bitonic}} &= \sum_{d=1}^{\lg n} T_{\text{merge}} \\
&= Q \cdot (n/p)(\lg p)(\lg p + 1)/2 + 5A \cdot (n/p)(\lg n)(\lg n + 1)/2 \\
&\approx 0.5Q \cdot (n/p) \lg^2 p + 2.5A \cdot (n/p) \lg^2 n \ . \qquad (1)
\end{aligned}
$$

Let us examine this formula more closely. The times in Figure 3 indicate that Q is 40 times larger than A, and $\lg n$ is at most 2 or 3 times larger than $\lg p$ for all but enormous volumes of data. Thus, the first term in formula (1), corresponding to communication time, dominates the arithmetic time for practical values of n and p.

The problem with this naive implementation is that it is a single-port algorithm: communication occurs across only one dimension of the hypercube at a time. By using all of the dimensions virtually all of the time, we can improve the algorithm's performance significantly. The idea is to use a multiport version of Bitonic-Merge that pipelines the keys across all dimensions of the hypercube. In the multiport version, a call of the form Bitonic-Merge(Key, d) is implemented as follows. On the first step, all processors cube swap their first keys across dimension d. On the second step, they cube swap their first keys across dimension $d-1$, while simultaneously cube swapping their second keys across dimension d. Continuing the pipelining in this manner, the total number of steps to move all the keys through $d - \lg(n/p)$ physical dimensions is $n/p + d - \lg(n/p) - 1$. This algorithm is essentially equivalent to a pipelined bitonic merge on a butterfly network.

Thus, pipelining improves the time for bitonic merging to

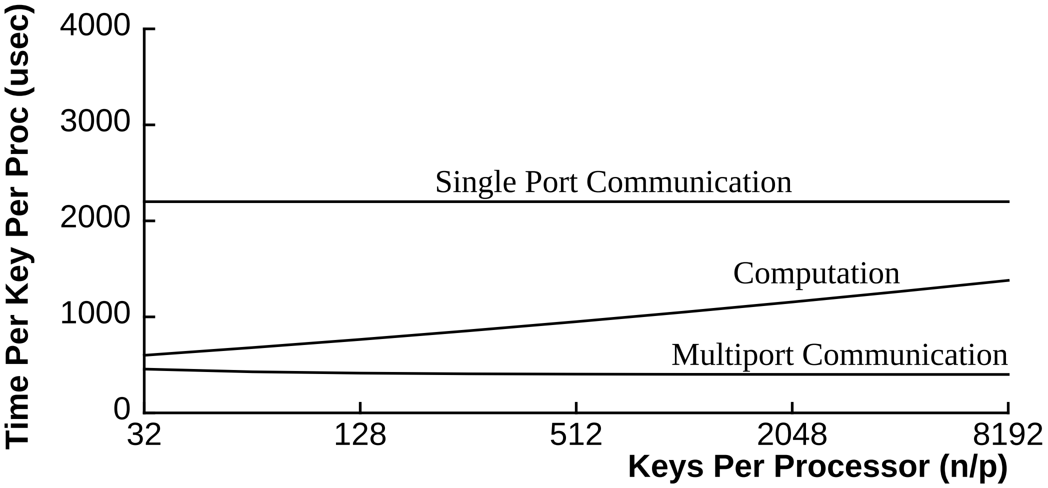
$$
T_{\text{multiport-merge}} = \begin{cases} (n/p) \cdot 5A \cdot d & \text{if} \quad d \le \lg(n/p) \ , \\ Q \cdot (n/p + d - \lg(n/p) - 1) + 5A \cdot (n/p)d & \text{if} \quad d > \lg(n/p) \ . \end{cases}
$$

By summing from $d = 1$ to $\lg n$, the time for the entire multiport bitonic sort, therefore, becomes

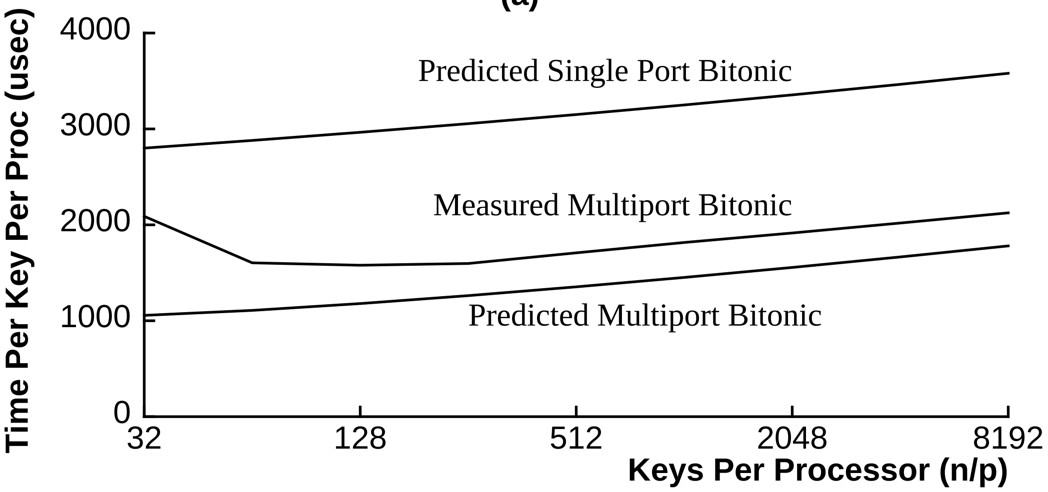
$$
\begin{aligned}
T_{\text{multiport-bitonic}} &= Q \cdot (\lg p)(n/p + (\lg p)/2 - 1/2) + 5A \cdot (n/p)(\lg n)(\lg n + 1)/2 \\
&\approx Q \cdot ((n/p) \lg p + 0.5 \lg^2 p) + 2.5A \cdot (n/p) \lg^2 n \ . \qquad (2)
\end{aligned}
$$

Let us compare this formula with the single-port result of equation (1). For $n = O(p)$, the two running times do not differ by more than a constant factor. For $n = \Omega(p \log p)$, however, the coefficient of Q is $\Theta(\log p)$ times smaller in the multiport case. Thus, total communication time is considerably reduced by pipelining when n/p is large. The number of arithmetic operations is not affected by pipelining.

Figure 6a shows the communication and computation components of the running time for both the single-port and multiport versions of bitonic sort. These times are generated from



(a)



(b)

Figure 6: Bitonic sorting 64-bit keys on a 32K CM-2 ($p = 1024$). **(a)** The predicted single port communication is approximately 5 times the predicted multiport communication time. **(b)** The measured performance of multiport bitonic sort closely matches the predicted performance, but contains a fixed overhead.

formulas (1) and (2). The computation component is equivalent for both algorithms. Figure 6b shows the predicted total time for the single-port and multiport bitonic, and the measured performance of our implementation of the multiport algorithm. The difference between predicted and measured times for small values of n/p is mostly due to the fact that our equations ignore constant overhead. The difference at high n/p is due to some overhead in our implementation caused by additional memory moves, effectively increasing the cost Q of the cube swap. This overhead could be eliminated by an improved implementation, but the resulting algorithm would still not be competitive with sample sort for large values of n/p.

Multiport bitonic sort can be further improved by using a linear-time serial merge instead of a bitonic merge in order to execute the merges that occur entirely within a processor [4]. We estimated that the time for a processor to merge two sorted sequences of length $(n/2p)$ to form a single sorted sequence of length (n/p) is approximately $(n/p) \cdot 10A$. The constant is large because of the indirect addressing that would be required by the implementation. In this case, the time for multiport-merge becomes

$$T_{\text{multiport-merge}} = \begin{cases} (n/p) \cdot 10A & \text{if} \quad d \leq \lg(n/p) \ , \\ Q \cdot (n/p + d - \lg(n/p) - 1) + 5A \cdot (n/p)d & \text{if} \quad d > \lg(n/p) \ . \end{cases}$$

This variation yields a final running time of

$$T_{\text{multiport-bitonic}} \approx Q \cdot ((n/p)\lg p + 0.5\lg^2 p) + A \cdot (n/p)(2.5\lg^2 p - 5\lg p + 10\lg n) \ .$$

For large n/p, this formula reduces the A term by a factor of 2 or more relative to equation (2). Once again, this improvement would not yield an algorithm that is close to the performance of the sample sort, and thus we decided not to implement it. Furthermore, the local merges could not be executed in place, so that the algorithm would lose one of its major advantages: it would no longer only require a fixed amount of additional memory.

4 Radix Sort

The second algorithm that we implemented is a parallel version of a counting-based radix sort [7, Section 9.3]. In contrast with bitonic sort, radix sort is not a *comparison sort*: it does not use comparisons alone to determine the relative ordering of keys. Instead, it relies on the representation of keys as b-bit integers. (Floating-point numbers can also be sorted using radix sort. With a few simple bit manipulations, floating point keys can be converted to integer keys with the same ordering and key size. For example, IEEE double precision floating point numbers can be sorted by inverting the mantissa and exponent bits if the sign bit is 1, and then inverting the sign bit. The keys are then sorted as if they were integers.) Our optimized version of radix sort is quite fast, and it was the simplest to code of the three sorting algorithms that we implemented.

The basic radix sort algorithm (whether serial or parallel) examines the keys to be sorted r bits at a time, starting with the least significant block of r bits in each key. Each time through the loop, it sorts the keys according to the r-bit block currently being considered in each key. Of fundamental importance is that this intermediate radix-2^r sort be *stable*: the output ordering must preserve the input order of any two keys whose r-bit blocks have equal values.

The most common implementation of the intermediate radix-2^r sort is as a counting sort. We first count to determine the *rank* of each key—its position in the output order—and then we permute the keys to their respective locations. The following pseudocode describes the implementation.

```
RADIX-SORT(Key)
1  for i ← 0 to b − 1 by r
2      do Rank ← COUNTING-RANK(r, Key⟨i, ..., i + r − 1⟩)
3         Key ← SEND(Key, Rank)
```

Since the algorithm requires b/r passes, the total time for a parallel sort is:

$$T_{\text{radix}} = (b/r) \cdot (R \cdot (n/p) + T_{\text{rank}})$$

where T_{rank} is the time taken by COUNTING-RANK.

The most interesting part of radix sort is the subroutine for computing ranks called in line 2. We first consider the simple algorithm underlying the original Connection Machine library sort [5], which was programmed by one of us several years ago. In the following implementation of COUNTING-RANK, the vector *Block* holds the r-bit values on which we are sorting.

```
SIMPLE-COUNTING-RANK(r, Block)
offset ← 0
for k ← 0 to 2^r − 1
    do Flag ← 0
       where Block = k do Flag ← 1
       Index ← SCAN(Flag)
       where Flag do Rank ← offset + Index
       offset ← offset + SUM(Flag)
return Rank
```

In this pseudocode, the **where** statement executes its body only in those processors for which the condition evaluates to TRUE.

The SIMPLE-COUNTING-RANK procedure operates as follows. Consider the ith key, and assume that $Block[i] = k$. The rank of the ith key is the number $offset_k$ of keys j for which $Block[j] < k$, plus the number $Index[i]$ of keys for which $Block[j] = k$ and $j < i$. (Here, $offset_k$ is the value of *offset* at the begining of the kth iteration of the **for** loop.) The code iterates over each of the 2^r possible values that can be taken on by the r-bit block on which we are sorting. For each value of k, the algorithm uses a scan to generate the vector *Index* and updates the value of *offset* to reflect the total number of keys whose *Block* value is less than or equal to k.

To compute the running time of SIMPLE-COUNTING-RANK, we refer to the running times of the CM-2 operations in Figure 3. On the CM-2, the SUM function can be computed as a by-product of the SCAN function, and thus no additional time is required to compute it. Assuming that we have p processors and n keys, the total time is

$$\begin{aligned} T_{\text{simple-rank}} &= 2^r \cdot (3A \cdot (n/p) + S) + 2^r (2A)(n/p) \\ &= A \cdot (5 \cdot 2^r (n/p)) + S \cdot 2^r , \end{aligned} \tag{3}$$

where the coefficient 2 of A in the last term of the first line of this equation was determined empirically by instrumenting the code. Here the term $2^r \cdot (3A \cdot (n/p) + S)$ represents the time to perform 2^r scans, and the term $2^r(2A)(n/p)$ represents the cost of the remaining operations, such as initializing the vector *Flag* and computing the vector *Rank*.

The total time for this version of radix sort—call it SIMPLE-RADIX-SORT—which uses SIMPLE-COUNTING-RANK on r-bit blocks of b-bit keys, is therefore

$$\begin{aligned} T_{\text{simple-radix}} &= (b/r)(R \cdot (n/p) + T_{\text{simple-rank}}) \\ &= (b/r)(R \cdot (n/p) + 5A \cdot 2^r(n/p) + S \cdot 2^r) \ . \end{aligned} \tag{4}$$

(The library sort actually runs somewhat slower for small values of n/p, because of a large fixed overhead.) Notice from this formula that increasing r reduces the number of routings proportionally, but it increases the arithmetic and scans exponentially.

We can determine the value of r that minimizes $T_{\text{simple-radix}}$ by differentiating the right-hand side of equation (4) with respect to r and setting the result equal to 0, which yields

$$r = \lg\left(\frac{(n/p)R}{(n/p)5A + S}\right) - \lg(r \ln 2 - 1) \ .$$

For large n/p (i.e., $n/p \gg (S/5A)$), the optimal value of r is

$$\begin{aligned} r &\approx \lg(R/5A) - \lg(r \ln 2 - 1) \\ &\approx 3.9. \end{aligned}$$

This analysis is borne out in practice by the CM-2 library sort, which runs the fastest for large n/p when $r = 4$.

We now consider an improved version of parallel radix sort. The idea behind this algorithm was used by Johnsson [14]. We shall describe the new algorithm for counting ranks in terms of the physical processors, rather than in terms of the keys themselves. Thus, we view the length-n input vector *Block* as a length-p vector, each element of which is a length-(n/p) array stored in a single processor. We also maintain a length-p vector *Index*, each element of which is a length-2^r array stored in a single processor. We shall describe the operation of the algorithm after giving the pseudocode:

```
COUNTING-RANK(r, Block)
for j ← 0 to 2^r − 1
    do Index[j] ← 0
for j ← 0 to n/p
    do increment Index[Block[j]]
offset ← 0
for k ← 0 to 2^r − 1
    do count ← SUM(Index[k])
       Index[k] ← SCAN(Index[k]) + offset
       offset ← offset + count
for j ← 0 to n/p − 1
    do Rank[j] ← Index[Block[j]]
       increment Index[Block[j]]
return Rank
```

The basic idea of the algorithm is as follows. For all *Block* values $k = 0, 1, \ldots, 2^r - 1$, lines 1–4 determine how many times each value k appears in each processor. Now, consider the ith processor and a particular value k. Lines 5–9 determine the final rank of the first key, if any, in processor i that has *Block* value k. The algorithm calculates this rank by computing the number offset_k of keys with *Block* values less than k to which it adds the number of keys with *Block* value equal to k that are in processors 0 through $i-1$. These values are placed in the vector $\mathit{Index}[k]$. Having computed the overall rank of the first key in each processor (for each *Block* value k), the final phase of the algorithm (lines 10–12) computes the overall rank of every key. This algorithm requires indirect addressing, since the processors must index their local arrays independently.

The total time for Counting-Rank is

$$T_{\text{rank}} = A \cdot (2 \cdot 2^r + 10(n/p)) + S \cdot 2^r ,$$

where the constants 2 and 10 were determined empirically. Note that $\mathrm{SUM}(\mathit{Index}[k])$ in line 7 is actually a by-product of $\mathrm{SCAN}(\mathit{Index}[k])$ in line 8, and hence a single SCAN suffices. Comparing with the result obtained for Simple-Counting-Rank, we find that the n/p and 2^r terms are now additive rather than multiplicative.

The time for Radix-Sort is

$$\begin{aligned} T_{\text{radix}} &= (b/r)(R \cdot (n/p) + T_{\text{rank}}) \\ &= (b/r)(R \cdot (n/p) + S \cdot 2^r + A \cdot (2 \cdot 2^r + 10(n/p))) \\ &= (b/r)((n/p) \cdot (R + 10A) + 2^r(S + 2A)) \ . \qquad (5) \end{aligned}$$

Figure 7 breaks down the running time of radix sort as a function of r for $n/p = 4096$. As can be seen from the figure, as r increases, the send time diminishes and the scan time grows. We can determine the value for r that minimizes the total time of the algorithm by differentiating the right-hand side of equation (5) with respect to r and setting the result equal to 0. For large numbers of keys per processor, the value for r that we obtain satisfies

$$\begin{aligned} r &= \lg((n/p)(R + 10A)/(S + 2A)) - \lg(r \ln 2 - 1) \qquad (6) \\ &\approx \lg(n/p) - \lg\lg(n/p) + 2 \ . \qquad (7) \end{aligned}$$

For $n/p = 4096$, as in Figure 7, equation (7) suggests that we set $r \approx 10$, which indeed comes very close to minimizing the total running time. The marginally better value of $r = 11$ can be obtained by solving equation (6) numerically.

Unlike the choice of r dictated by the analysis of Simple-Radix-Sort, the optimal choice of r for Radix-Sort grows with n/p. Consequently, for large numbers of keys per processor, the number of passes of Radix-Sort is smaller than that of Simple-Radix-Sort. When we substitute our choice of r back into equation (5), we obtain

$$T_{\text{radix}} \approx (n/p)\left(\frac{b}{\lg(n/p) - \lg\lg(n/p) + 1.5}\right)\left(R + 10A + \frac{3}{\lg(n/p)}(S + 2A)\right) \ . \qquad (8)$$

In our implementation of Radix-Sort, the optimal values of r have been determined empirically. Figure 8 compares the performance predicted by equation (8) with the actual running time of our implementation.

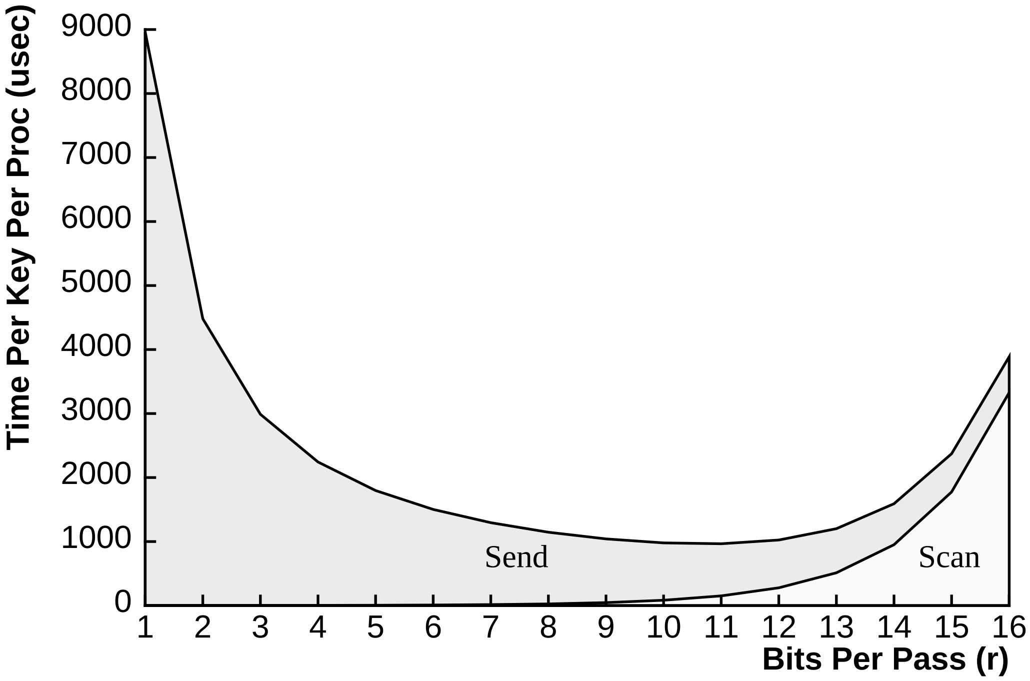


Figure 7: A breakdown of the total running time of radix sort into send time and scan time for sorting 64-bit keys ($b = 64$) with $n/p = 4096$. The total running time is indicated by the top curve. The two shaded areas represent the scan time and send time. As r is increased, the scan time increases and the send time decreases. (The arithmetic time is negligible.) For the parameters chosen, the optimal value of r is 11.

5 Sample Sort

The third sort that we implemented is a sample sort [10, 13, 19, 20, 24]. This sorting algorithm was the fastest for large sets of input keys, beating radix sort by more than a factor of 2. It also was the most complicated to implement. The sort is a randomized sort: it uses a random number generator. The running time is almost independent of the input distribution of keys and, with very high probability, the algorithm runs quickly.

Assuming n input keys are to be sorted on a machine with p processors, the algorithm proceeds in three phases:

1. A set of $p-1$ "splitter" keys are picked that partition the linear order of key values into p "buckets."

2. Based on their values, the keys are sent to the appropriate bucket, where the ith bucket is stored in the ith processor.

3. The keys are sorted within each bucket.

If necessary, a fourth phase can be added to load balance the keys, since the buckets do not typically have exactly equal size.

Sample sort gets its name from the way the $p-1$ splitters are selected in the first phase. From the n input keys, a sample of $ps \leq n$ keys are chosen at random, where s is a parameter

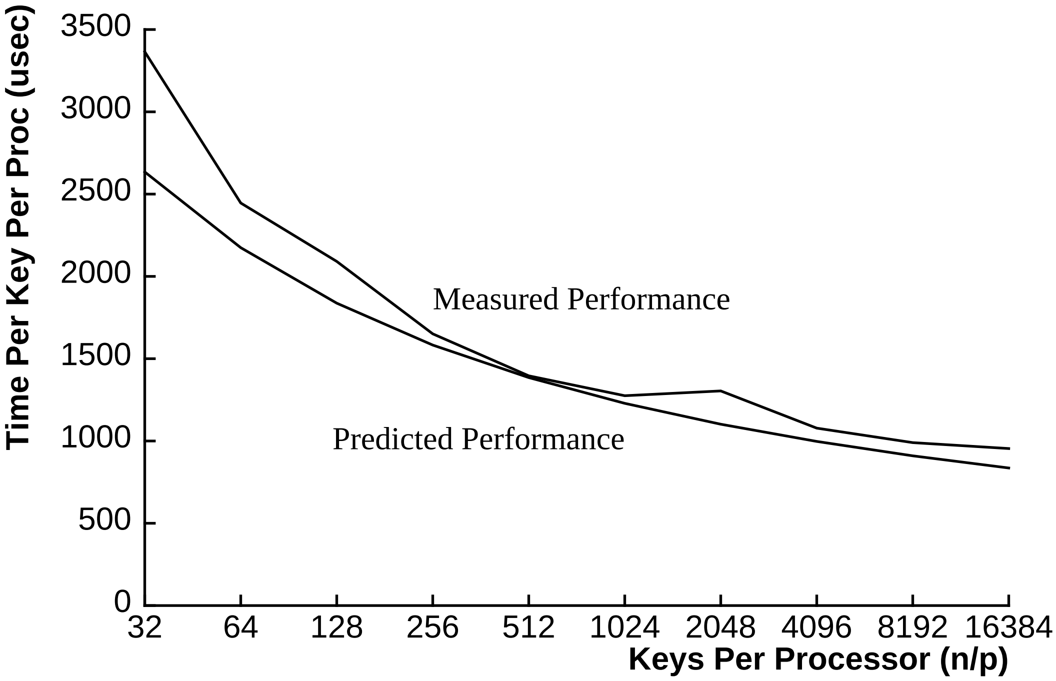


Figure 8: Predicted and measured performance of radix sort with 64-bit keys. Measured performance is on a 32K CM-2 and uses the empirically determined optimal values for r. Predicted performance was calculated using equation (8).

called the *oversampling ratio*. This sample is sorted, and then the $p-1$ splitters are selected by taking those keys in the sample that have ranks $s, 2s, 3s, \ldots, (p-1)s$.

Some sample sort algorithms [10, 20, 24] choose an oversampling ratio of $s = 1$, but this choice results in a relatively large deviation in the bucket sizes. By choosing a larger value, as suggested by Reif and Valiant [19] and by Huang and Chow [13], we can guarantee with high probability that no bucket contains many more keys than the average. (The Reif-Valiant flashsort algorithm differs in that it uses buckets corresponding to $O(\lg^7 p)$-processor subcubes of the hypercube.)

The time for Phase 3 of the algorithm depends on the maximum number, call it L, of keys in a single bucket. Since the average bucket size is n/p, the efficiency by which a given oversampling ratio s maintains small bucket sizes can be measured as the ratio $L/(n/p)$, which will be referred to as the *bucket expansion*. The bucket expansion gives the ratio of the maximum bucket size to the average bucket size. The expected value of the bucket expansion depends on the oversampling ratio s and on the total number n of keys, and will be denoted by $\beta(s,n)$.

It is extremely unlikely that the bucket expansion will be significantly greater than its expected value. If the oversampling ratio is s, then the probability that the bucket expansion is greater than some factor $\alpha \geq 1 + 1/s$ is

$$\Pr[\beta(s,n) > \alpha] \quad \leq \quad n e^{-(1-1/\alpha)^2 \alpha s/2} \; . \tag{9}$$

This bound, which is proved in Appendix B, is graphed in Figure 9. As an example, with an

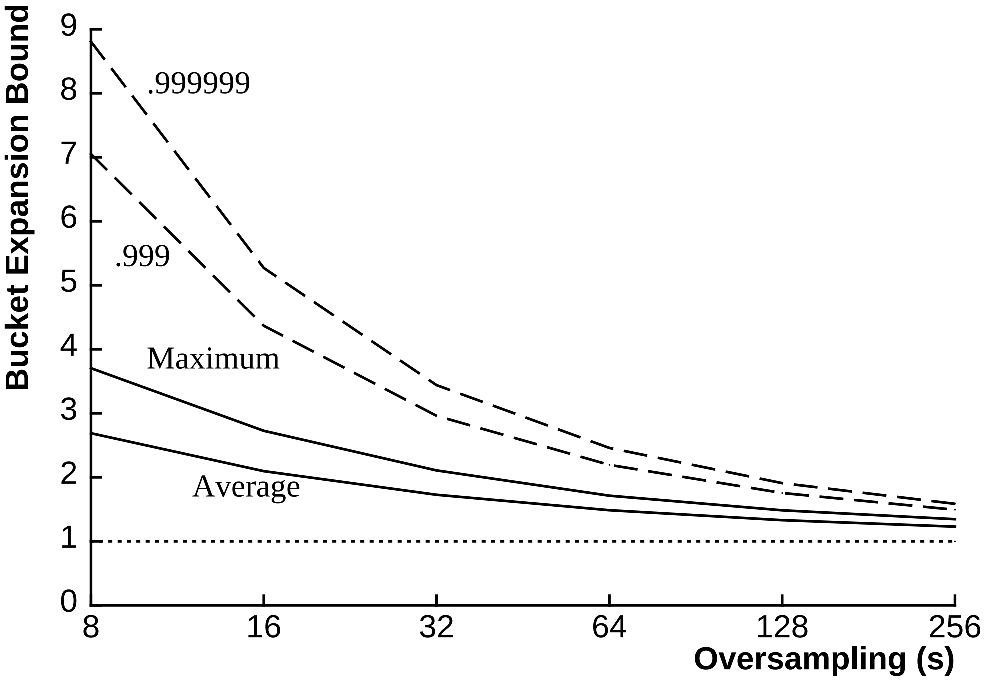


Figure 9: Bucket expansion for sample sorting $n = 10^6$ keys, as a function of oversampling ratio s ($p = 1024$). The dashed curves are theoretical upper bounds given by inequality (9) when setting the probability of being within the bound to $1 - 10^{-3}$ (the lower dashed curve) and $1 - 10^{-6}$ (the upper dashed curve). The solid curves are experimental values for bucket expansion. The upper solid curve shows the maximum bucket expansion found over 10^3 trials, and the lower solid curve shows the average bucket expansion over 10^3 trials. In practice, oversampling ratios of $s = 32$ or $s = 64$ yield bucket expansions of less than 2.

oversampling ratio of $s = 64$ and $n = 10^6$ keys, the probability that the largest bucket is more than 2.5 times as large as the average bucket is less than 10^{-6}.

We shall see shortly that the running time of sample sort depends linearly on both the oversampling ratio and the bucket expansion. As is apparent from Figure 9, as the oversampling ratio s increases, the bucket expansion decreases. Thus, the oversampling ratio s must be carefully adjusted in order to obtain optimal performance.

We are now ready to discuss our implementation of the sample sort algorithm. Before executing Phase 1, however, the algorithm must do a little preprocessing. The reason is that the basic sample sort algorithm assumes that all input keys are distinct. If many keys happen to have the same value, failure to break ties consistently between them can result in an uneven distribution of keys to buckets. Consequently, before the first phase of the sample sort begins, we tag each key with its address, thereby guaranteeing that the tagged keys all have distinct values.

Phase 1: Selecting the splitters

The first phase of sample sort begins with each processor randomly selecting a set of s tagged keys from among those stored in its local memory. We call these keys the *candidates*. We implement this method by partitioning each processor's n/p keys into s blocks of n/ps keys, and then we choose one key at random from each block. This selection process differs from that where each processor selects s tagged keys randomly from the entire set, as is done in both the Reif-Valiant [19] and Huang-Chow [13] algorithms. All of these methods yield small bucket expansions. Since the CM-2 is a distributed-memory machine, however, the local-choice method has an advantage in performance over global-choice methods: no global communication is required to select the candidates. In our implementation, we typically pick $s = 32$ or $s = 64$, depending on the number of keys per processor in the input.

Once the candidates have been selected, they are sorted across the machine using the simple version of radix sort described in Section 4. (Since radix sort is stable, the tags need not be sorted.) Since the sample contains many fewer keys than does the input, this step runs significantly faster than sorting all of the keys with radix sort. The splitters are now chosen as the keys with ranks $s, 2s, 3s, \ldots, (p-1)s$. The actual extraction of the splitters from the sample is implemented as part of Phase 2.

The dominant time required by Phase 1 is the time for sorting the candidates:

$$T_{\text{candidates}} = RS(ps, p) , \tag{10}$$

where $RS(ps, p)$ is the time required to radix sort ps keys on p processors. Using the radix sort from the original CM-2 Paris library, we have $T_{\text{candidates}} \approx 7000A \cdot s$.

Notice that the time for Phase 1 is independent of the total number n of keys, since during the selection process, a processor need not look at all of its n/p keys in order to randomly select from them. Notice also that if we had implemented a global-choice sampling strategy, we would have had a term containing $R \cdot s$ in the expression.

Phase 2: Distributing the keys to buckets

Except for our local-choice method of picking a sample and the choice of algorithm used to sort the oversampled keys, Phase 1 follows both the Reif-Valiant and Huang-Chow algorithms. In Phase 2, however, we follow Huang-Chow more closely.

Each key can determine the bucket to which it belongs by performing a binary search of the sorted array of splitters. We implemented this part of the phase in a straightforward fashion: the front end reads the splitters one by one and broadcasts them to each processor. Then, each processor determines the bucket for each of its keys by performing a binary search of the array of splitters stored separately in each processor. Once we have determined to which bucket a key belongs, we throw away the tagging information used to make each key unique and route the keys directly to their appropriate buckets. We allocate enough memory for the buckets to guarantee a very high probability of accommodating the maximum bucket size. In the unlikely event of a bucket overflow, excess keys are discarded during the route and the algorithm is restarted with a new random seed.

The time required by Phase 2 can be separated into the time for the broadcast, the time for the binary search, and the time for the send:

$$\begin{aligned} T_{\text{broadcast}} &= 50A \cdot p , \\ T_{\text{bin-search}} &= 6.5A \cdot (n/p) \lg p , \\ T_{\text{send}} &= R \cdot (n/p) ; \end{aligned}$$

where the constants 50 and 6.5 were determined empirically by instrumenting the code.

As is evident by our description and also by inspection of the formula for $T_{\text{broadcast}}$, the reading and broadcasting of splitters by the front end is a serial bottleneck for the algorithm. Our sample sort is really only a reasonable sort when n/p is large, however. In particular, the costs due to binary search outweigh the costs due to reading and broadcasting the splitters when $6.5(n/p)\lg p > 50p$, or equivalently, when $n/p > (50/6.5)p/\lg p$. For a 64K CM-2, we have $p = 2048$, and the preceding inequality holds when the number n/p of input keys per processor is at least 1432. This number is not particularly large, since each processor on the CM-2 has a full megabyte of memory even when the machine is configured with only 1-megabit DRAM's.

Phase 3: Sorting keys within processors

The third phase sorts the keys locally within each bucket. The time taken by this phase is equal to the time taken by the processor with the most keys in its bucket. If the expected bucket expansion is $\beta(s,n)$, the expected size of the largest bucket is $(n/p)\beta(s,n)$.

We use a standard serial radix sort in which each pass is implemented using several passes of a counting sort (see, for example, [7, Section 9.3]). Radix sort was used because it is significantly faster than comparison sorts such as quicksort. The serial radix sort requires time

$$T_{\text{local-sort}} = (b/r)A \cdot ((1.3)2^r + 10(n/p)\beta(s,n)) , \tag{11}$$

where b is the number of bits in a key and 2^r is the radix of the sort. The first term in the coefficient of A corresponds to the b/r (serial) scan computations on a histogram of key values, and the second term corresponds to the work needed to count the number of keys with each r-bit value and to put the keys in their final destinations.

We can determine the value of r that minimizes $T_{\text{local-sort}}$ by differentiating the right-hand side of equation (11) with respect to r and setting the result equal to 0. This yields $r \approx \lg(n/p) - 1$ for large n/p. With this selection of r, the cost of the first term in the equation is small relative to the second term. Typically, $b/r \approx 6$, and $\beta(s,n) \approx 2$, which yields

$$\begin{aligned} T_{\text{local-sort}} &\approx 6A \cdot 10(n/p) \cdot 2 \\ &= 120A \cdot (n/p) . \end{aligned} \tag{12}$$

Discussion

The main parameter to choose in the algorithm is the oversampling ratio s. A larger s distributes the keys more evenly within the buckets, thereby speeding up Phase 3 of the algorithm. A larger s also means a larger sample to be sorted, however, thereby causing Phase 1 of the algorithm to take longer. Figure 10 shows the tradeoff we obtained experimentally, for $n/p = 16384$. As can be seen from the figure, choosing $s = 64$ is optimal in this case.

To obtain the arithmetic expression that describes the total running time of the sample sort, we sum the formulas for the phases, which results in the expression

$$\begin{aligned} T_{\text{sample}} &= T_{\text{candidates}} + T_{\text{broadcast}} + T_{\text{bin-search}} + T_{\text{send}} + T_{\text{local-sort}} \\ &\approx 7000A \cdot s + 50A \cdot p + 6.5A \cdot (n/p)\lg p + R \cdot (n/p) + 120A \cdot (n/p) , \end{aligned}$$

where we have dropped inessential terms. Figure 11 shows the experimental breakdown of times for the various tasks accomplished by the algorithm, which closely match this equation.

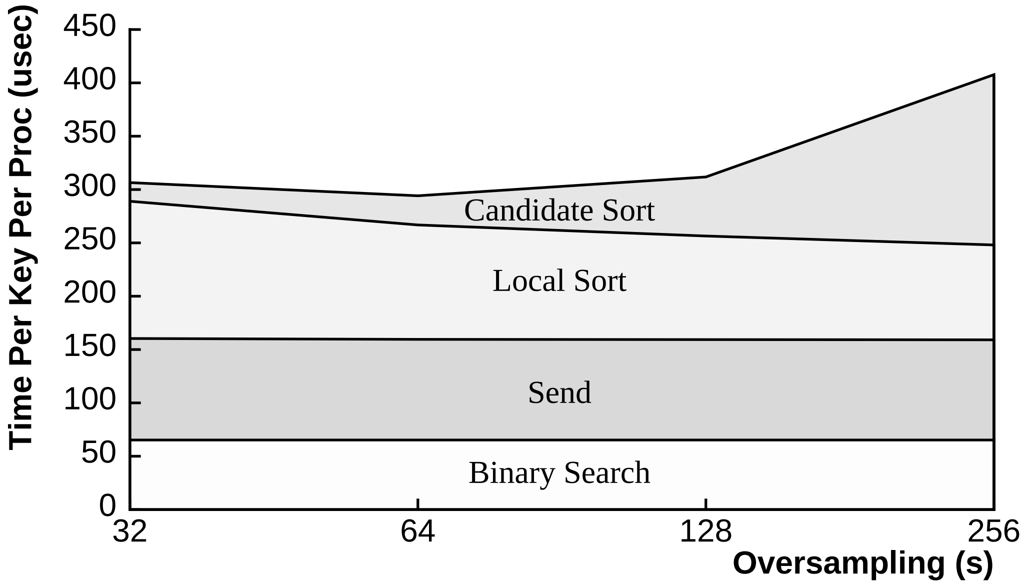


Figure 10: Breakdown of sample sort for various choices of the oversampling ratio s. The graph shows the measured running time for sorting 64-bit keys ($b = 64$) with $n/p = 16384$ keys per processor on a 32K CM-2 ($p = 1024$), where the height of each labeled region indicates the time for the corresponding component of the sort. As the oversampling ratio is increased, two effects may be observed: (i) the time for the candidate sort increases because there are more candidates to sort, and (ii) the time for the local sort decreases because the maximum bucket expansion diminishes. For these parameters, the total time is minimized at $s = 64$. ($T_{\text{broadcast}}$ is not shown, since it is negligible.)

Let us look more closely at the formula for T_{sample} as n/p becomes large. The first two terms, which correspond to the sorting of candidates and the broadcasting of splitters, become insignificant. On a 64K CM-2, the other three terms grow proportionally. Specifically, the last two terms, which correspond to the send and local sort, take about the same time, and the third term, binary searching, takes about half that time. Thus, as n/p becomes large, the entire algorithm runs in less than 3 times the cost of a single send. Since T_{sample} is so close to the time of a single send, it is unlikely that any other sorting algorithm on the CM-2 can outperform it by much.

There were many variations of the sample sort algorithm that we considered implementing. We now discuss a few.

Splitter-directed routing. Rather than broadcasting the splitters, binary searching, and sending in Phase 2, we might have used the "splitter-directed routing" method from Reif and Valiant's flashsort. The idea is to send each key through the hypercube to its destination bucket. At each hypercube node, the key chooses either to leave the node or not based on a comparison of the key to a splitter value stored at the node. Each key therefore follows a path through the network to its bucket based on $\lg p$ comparisons, one at each level of the network. On the CM-2, the algorithm can be pipelined across the cube wires in a way that is similar to the pipelined version of bitonic sort. The local processing required at each step of the routing is quite involved, however. It requires managing queues since a variable number of messages

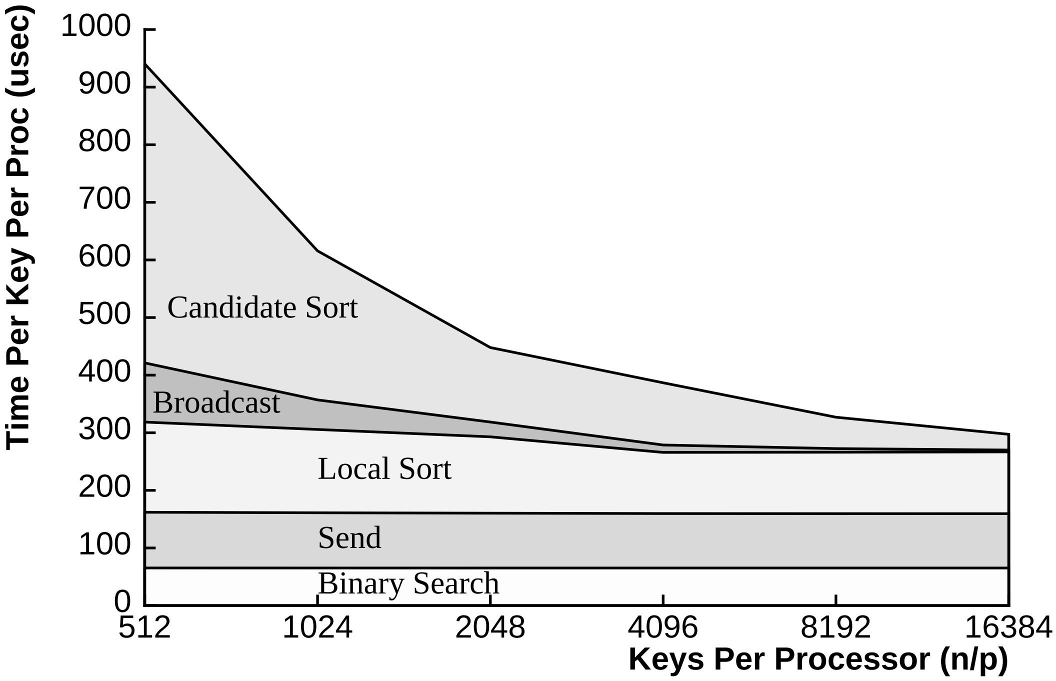


Figure 11: A breakdown of the actual running time of sample sort, as a function of the input size n. The graph shows actual running times for 64-bit keys on a 32K CM-2 ($p = 1024$). The per-key cost of broadcasting splitters decreases as n/p increases, since the total cost of the broadcast is independent of n. The per-key cost of the candidate sort decreases until there are 4K keys per processor; at this point, we increase the oversampling ratio from $s = 32$ to $s = 64$ in order to reduce the time for local sorting. The local sort improves slightly at higher values of n/p, because the bucket expansion decreases while the per-key times for send and binary search remain constant.

can arrive at each node. Our analysis indicated that although communication costs for splitter-directed routing might take less time than the communication costs required to simply route messages through the network, this advantage could not be exploited, because bookkeeping (arithmetic) costs would dominate. Our conclusion was that splitter-directed routing may be a reasonable option if it is supported by special-purpose hardware. Lacking that on the CM-2, the scheme that we chose to implement was faster and much simpler to code. Since our original work, Hightower, Prins and Reif have implemented a version of splitter-directed routing [12] on a toroidal mesh and found that when the number of keys per processor is not large, splitter directed routing can outperform samplesort on the Maspar MP-1.

Smaller sets of keys. As implemented, sample sort is suitable only when the number n/p of keys per processor is large, especially since we broadcast the $p - 1$ splitters from the front end. One way to improve the algorithm when each processor contains relatively few input keys would be to execute two passes of phases 1 and 2. In the first pass, we can generate $\sqrt{p} - 1$ splitters and assign a group of $\sqrt{p}$ processors to each bucket. Each key can then be sent to a random processor within the processor group corresponding to its bucket. In the second pass, each group generates $\sqrt{p}$ splitters which are locally broadcast within the subcubes, and then keys are sent to their final destinations. With this algorithm, many fewer splitters need to be

Alg.	Stable	Load Bal.	Time/(n/p)		Mem.	Rank
			$n/p = 64$	$n/p = 16K$		
Bitonic	no	yes	1600μsec	2200μsec	1.0	1.5
Radix	yes	yes	2400μsec	950μsec	2.1	1.0
Sample	no	no	5500μsec	330μsec	3.2	1.5

Figure 12: Summary of the three sorting algorithms assuming 64-bit keys. The *load balanced* column specifies whether the final result is balanced across the processors. The *time* column is the time to sort on a 1024-processor machine (32K CM-2). The *memory* column is the ratio, for large n/p, of the space taken by the algorithm to the space taken by the original data. The rank column is an approximate ratio of the time of a rank to the time of a sort. The *rank* operation returns to each key the rank it would attain if the vector were sorted.

distributed to each processor, but twice the number of sends are required. This variation was not implemented, because we felt that it would not outperform bitonic sort for small values of n/p.

Load balancing. When the three phases of the algorithm are complete, not all processors have the same number of keys. Although some applications of sorting—such as implementing a combining send or heuristic clustering—do not require that the processor loads be exactly balanced, many do. Load balancing can be performed by first scanning to determine the destination of each sorted key and then routing the keys to their final destinations. The dominant cost in load balancing is the extra send. We implemented a version of sample sort with load balancing. With large numbers of keys per processor, the additional cost was only 30 percent, and the algorithm still outperforms the other sorts.

Key distribution. The randomized sample sort algorithm is insensitive to the distribution of keys, but unfortunately, the CM-2 message router is not, as was mentioned in Section 2. In fact, for certain patterns, routing can take up to two and a half times longer than normally expected. This difficulty can be overcome, however, by randomizing the location of buckets. For algorithms that require the output keys in the canonical order of processors, an extra send is required, as well as a small amount of additional routing so that the scan for load balancing is performed in the canonical order. This same send can also be used for load balancing.

6 Conclusions

Our goal in this project was to develop a system sort for the Connection Machine. Because of this goal, raw speed was not our only concern. Other issues included, space, stability, portability, and simplicity. Radix sort has several notable advantages with respect to these criteria. Radix sort is stable, easy to code and maintain, performs reasonably well over the entire range of n/p, requires less memory than sample sort, and performs well on short keys. Although the other two sorts have domains of applicability, we concluded that the radix sort was most suitable as a system sort.

Figure 12 compares the three sorting algorithms. In the following paragraphs, we examine some of the quantitative differences between the algorithms.

Running Time. A graph of the actual running times of all three sorts along with the time of the original system sort was given in Figure 1. With many keys per processor, the sample sort is approximately three times faster than the other two sorts and therefore, based on pure performance, sample sort is the clear winner.

More informative than the raw running times are the equations for the running times, since they show how the running time is affected by the number of keys, the number of processors, and various machine parameters. If we assume that n/p is large, we can approximate the equations for the three algorithms as

$$\begin{aligned} T_{\text{bitonic}} &\approx (n/p)(Q \cdot (\lg p) \ + \ A \cdot 2.5(\lg^2 n)) \\ T_{\text{radix}} &\approx (n/p)(R \cdot 6 \ + \ A \cdot 80) \\ T_{\text{sample}} &\approx (n/p)(R \ + \ A \cdot (5 \lg p + 120)) \ . \end{aligned}$$

If Q, R and A are known, these equations can be used to give rough estimates of running times for the algorithms on other machines. We caution, however, that running times predicted in this fashion could err by as much as a factor of 2. The A terms in the equations are likely to be the least accurate since the constants were all derived empirically for the CM-2, and they depend highly on the local capabilities of the processors.

The equations can also give an idea of how much would be gained in each sorting algorithm by improving various aspects of the CM-2. For example, we could analyze the effect of improving the time for a send. Based on the equations, we see that radix sort would benefit the most, since its running time is dominated by the send (currently on the CM-2, $R = 130A$).

Space. A second important concern is the space required by each sorting algorithm. Bitonic sort executes in place and therefore requires only a small constant amount of additional memory for storing certain temporary variables. Our radix sort, using n keys, each consisting of w 32-bit words, requires $2w(n/p) + 2^r$ 32-bit words of space per processor. The first term is needed for storing the keys before and after the send (the send cannot be executed in place), and the second term is needed for holding the bucket sums. Because of the first term, the space required by the sort is at least twice that required by the original data. The number in Figure 12 corresponds to the case $w = 2$ (64-bits) and $r = \lg(n/p) - 2$ (set to minimize the running time). Our sample sort requires a maximum of $2w(n/p)\beta(s,n) + 2^r + (w+1)(p-1)$ 32-bit words of space in any processor. The first and second terms are needed for local radix sorting, and the third term is needed for storing the splitters within each processor. The number in Figure 12 corresponds to the case $w = 2$, $r = \lg(n/p) - 1$ (set to minimize the running time), and $\beta(s,n) \approx 1.5$ (determined from experimental values).

Ranking. Often, in practice, a "rank" is a more useful operation than a sort. For a vector of keys, the *rank* operation returns to each key the rank it would attain if the vector were sorted. This operation allows the user to rank the keys and then send a much larger block of auxiliary information associated with each key to the final sorted position. For each of the three algorithms that we implemented, we also implemented a version that generates the ranks instead of the final sorted order. To implement a rank operation in terms of a sort, the original index in the vector is tagged onto each key and is then carried around during the sort. Once sorted, the final index is sent back to the location specified by the tag (the key's original position). In a radix-sort-based implementation of the rank operation, the cost of the additional send can be avoided by omitting the last send of radix sort, and sending the rank directly back to the index specified by the tag. Furthermore, as each block of the key is used by radix sort, that block can be thrown away, thereby shortening the message length of subsequent sends. Because of this, the time of "radix-rank" is only marginally more expensive than that of radix sort. For sample sort and bitonic sort, carrying the tag around slows down the algorithm by a factor of between 1.3 and 1.5.

Stability. Radix sort is stable, but the other two sorts are not. Bitonic sort and sample sort can be made stable by tagging each key with its initial index, as is done for the rank. In this case, however, not only must the tag be carried around during the sends, it must also be used in the comparisons. Sorting the extra tag can cause a slowdown of up to a factor of 1.5.

Key Length. Another issue is sorting short keys—keys with perhaps 10, 16, or 24 significant bits. Sorting short keys is a problem that arises reasonably often in CM-2 applications. For short keys, the time required by bitonic sort is not at all improved over the 32-bit time. The time required by the sample sort is marginally improved, since the cost of the local radix sort is reduced. The time required by radix sort, however, is essentially proportional to the key length. Since r is typically in the range $10 \leq r < 16$, sorting 20 bits requires 2 passes instead of 3 to 4 for 32 bits and 5 to 7 for 64 bits.

A Other Sorts of Sorts

Many algorithms have been developed for sorting on the hypercube and related networks such as the butterfly, cube-connected cycles, and shuffle-exchange. We considered a number of these algorithms before deciding to implement bitonic sort, radix sort, and sample sort. The purpose of this section is to discuss some of the other sorting algorithms considered and, in particular, to indicate why these alternatives were not selected for implementation.

Quicksort. It is relatively easy to implement a parallel version of quicksort on the CM-2 using segmented scans. First, a pivot is chosen at random and broadcast using scans. The pivot partitions the keys into *small* keys and *large* keys. Next, using scans, each small key is labeled with the number of small keys that precede it in the linear order, and each large key is labeled with the number of large keys that precede it, plus the total number of small keys. The keys are then routed to the locations specified by their labels. The new linear order is broken into two segments, the small keys and the large keys, and the algorithm is recursively applied to each segment. The expected number of levels of recursion is close to $\lg n$, and, at each level, the algorithm performs 1 route and approximately 7 scans. This algorithm has been implemented in a high level language (*Lisp) and runs about 2 times slower than the original system sort. We believed that we could not speed it up significantly, since the scan and route operations are already performed in hardware.

Hyperquicksort. The hyperquicksort algorithm [23] can be outlined as follows. First, each hypercube node sorts its n/p keys locally. Then, one of the hypercube nodes broadcasts its median key, m, to all of the other nodes. This key is used as a pivot. Each node partitions its keys into those smaller than m, and those larger. Next, the hypercube nodes exchange keys along the dimension-0 edges of the hypercube. A node whose address begins with 0 sends all of its keys that are larger than m to its neighbor whose address begins with 1. The neighbor sends back all of its keys that are smaller than m. As keys arrive at a node, they are merged into the sorted sequence of keys that were not sent by that node. Finally, the algorithm is recursively applied to the $p/2$-node subcubes whose addresses begin with 0 and 1, respectively.

The communication cost of hyperquicksort is comparable to that of the fully-pipelined version of bitonic sort. The expected cost is at least $Qn \lg p/2p$ since the algorithm uses the $\lg p$ dimensions one at a time and, for each dimension, every node expects to send half of its n/p keys to its neighbor. The cost of bitonic sort is always $Q \cdot (\lg p)(n/p + (\lg p)/2 - 1/2)$ (see Section 3).

The main advantage of bitonic sort over hyperquicksort is that its performance is not affected by the initial distribution of the keys to be sorted. Hyperquicksort relies on a random initial distribution to ensure that the work each processor has to do is reasonably balanced. Although hyperquicksort may perform less arithmetic than bitonic sort in the best case, it uses indirect addressing, which is relatively expensive on the CM-2.

Sparse enumeration sort. The Nassimi-Sahni sorting algorithm [17], which will be referred to as *sparse enumeration sort*, is used when the number n of items to be sorted is smaller than the number p of processors. In the special case $n = \sqrt{p}$, sparse enumeration sort is a very simple algorithm indeed. The n records are initially stored one-per-processor in the n lowest-numbered processors; viewing the processors of the hypercube as forming a two-dimensional $n \times n$ array, the input records occupy the first row of the array. Sparse enumeration sort proceeds by performing a set of n parallel column broadcasts (from the topmost entry in each column) followed by n parallel row broadcasts (from the diagonal positions), so that the processor at row i and column j of the array contains a copy of the ith and jth items. At this point, all pairs of items can be simultaneously compared in constant time, and prefix operations over the rows can be used to compute the overall rank of each item. The ith row is then used to route a copy of item i to the column corresponding to its output rank. Finally, a set of n parallel column routes is used to move each item to its sorted output position in the first row. For values of n strictly less than $\sqrt{p}$, sparse enumeration sort proceeds in exactly the same fashion: n^2 processors are used, and the remaining $p - n^2$ processors are idle. Thus, sparse enumeration sort runs in $O(\lg n)$ time when $n \leq \sqrt{p}$.

Sparse enumeration sort generalizes the preceding algorithm in an elegant manner to obtain a smooth tradeoff between $O(\lg n)$ performance at $n = \sqrt{p}$ and $O(\lg^2 n)$ performance at $n = p$ (the performance of bitonic sort). In this range, sparse enumeration sort is structured as a (p/n)-way merge sort. After the ith set of parallel merges, the n items are organized into $n(n/p)^i$ sorted lists of length $(p/n)^i$. (initially there are n lists of length 1.) The ith set of merges is performed in $O(i \lg(p/n))$ time using a constant number of bitonic merges, prefix operations, and monotone routes. Monotone routes are a special class of routing operations that can be performed deterministically, on-line in a collision-free manner. On the CM-2, monotone routes would be implemented using cube swaps; the entire implementation of sparse enumeration sort would not make use of the CM-2 router. A straightforward computation shows that the overall time complexity of sparse enumeration sort is $O(\lg^2 n/\lg(p/n))$ time.

For sufficiently large values of the ratio $p/n > 1$, one would expect sparse enumeration sort would perform better than the other sorts we looked at. It is unclear, however, that one would need a parallel computer to solve such small problems and one might get better times by solving the problem on a single processor, or by reducing p.

Column sort. Leighton's column sort [15] is an elegant parallel sorting technique that has found many theoretical applications. Column sort sorts n keys using two primitive operations. The first primitive operation is to sort $n^{1/3}$ separate sets (called columns) of $n^{2/3}$ keys each. Depending on the particular application, this sorting primitive may either be accomplished by a recursive call or, more typically, by some other sorting algorithm. The second primitive operation is to route all n keys according to a fixed permutation. Alternating between sorts and routes 4 times suffices to sort all n elements.

If $n \geq p^3$, then column sort runs quite efficiently. The sorting primitive is executed as a local sort, and all of the fixed permutations required by column sort are straightforward to implement in a greedy, collision-free manner. In terms of the CM-2, they can be implemented with a simple sequence of cube swaps rather than by invoking the router. As another implementation optimization, we remark that the "standard" column sort algorithm is not pipelined and would only make use of a $1/\lg p$ fraction of the CM-2 wires at any given time. A $\Theta(\lg p)$ speedup can be achieved by pipelining, and there are at least two approaches worthy of consideration. The first approach is to partition the data at each processor into $\lg p$ equal-sized sets, interleave $\lg p$ column sorts, and then merge the resulting $\lg p$ sorted lists. The second approach is to pipeline each of the routing operations in a single application of column sort.

The main drawback of column sort is that, for $n < p^3$, some degree (depending on the ratio n/p) of recursion is necessary in order to perform the sorting primitive; sets of $n^2/3$ items occupy more than a single processor. We chose not to implement column sort because it appeared that the condition $n \geq p^3$ would not be satisfied in many cases of interest, and a close analysis of critical sections of the potential code indicated that a recursive version of column sort would provide little, if any, improvement over either radix sort or sample sort. Furthermore, the relative performance of column sort would tend to degrade quite severely for small values of the ratio n/p.

The asymptotic performance of column sort is best understood by considering arithmetic and communication costs separately. Let us assume that $n \geq p^{1+\epsilon}$, where ϵ denotes an arbitrary positive constant, which implies a bounded depth of recursion. Under this assumption, the total arithmetic cost of column sort is $\Theta((n/p)\lg n)$, which is optimal for any comparison-based sort. With pipelining, the communication cost of column sort is $\Theta(n/p)$, which is optimal for *any* sorting algorithm.

To summarize, although we felt that column sort might turn out to be competitive at unusually high loads ($n \geq p^3$), its mediocre performance at high loads ($p^2 \leq n < p^3$), and poor performance at low to moderate loads ($p \leq n < p^2$) made other alternatives more attractive. Column sort might well be a useful component of a hybrid sorting scheme that automatically selects an appropriate algorithm depending upon the values of n and p.

Cubesort. Like column sort, the cubesort algorithm of Cypher and Sanz [9] gives a scheme for sorting n items in a number of "rounds", where in each round the data is partitioned into

n/s sets of size s (for some s, $2 \leq s \leq n$), and each set is sorted. (Successive partitions of the data are determined by simple fixed permutations that can be routed just as efficiently as those used by column sort.) The main advantage of cubesort over column sort is that, for a wide range of values of s, cubesort requires asymptotically fewer rounds than column sort. In particular, for $2 \leq s \leq n$, column sort (applied recursively) uses $\Theta((\lg n/\lg s)^{\beta})$ rounds for $\beta = 2/(\lg 3 - 1) \approx 3.419$, whereas cubesort uses only $O((25)^{\lg^* n - \lg^* s}(\lg n/\lg s)^2)$ rounds. (The cost of implementing a round is essentially the same in each case.) For $n \geq p^3$, cubesort can be implemented without recursion, but requires 7 rounds as opposed to 4 for column sort. For $n < p^3$, both cubesort and column sort are applied recursively. For n sufficiently smaller than p^3 (and p sufficiently large), the aforementioned asymptotic bounds imply that cubesort will eventually outperform column sort. However, for practical values of n and p, if such a crossover in performance ever occurs, it appears likely to occur at a point where both cubesort and column sort have poor performance relative to other algorithms (e.g., at low to moderate loads).

Nonadaptive smoothsort. There are several variants of the smoothsort algorithm, all of which are described in [18]. The most practical variant, and the one of interest to us here, is the nonadaptive version of the smoothsort algorithm. The structure of this algorithm, hereinafter referred to simply as "smoothsort," is similar to that of column sort. Both algorithms make progress by ensuring that under a certain partitioning of the data into subcubes, the distribution of ranks of the items within each subcube is similar. The benefit of performing such a "balancing" operation is that after the subcubes have been recursively sorted, all of the items can immediately be routed close to their correct position in the final sorted order (i.e., the subcubes can be approximately merged in an oblivious fashion). The effectiveness of the algorithm is determined by how close (in terms of number of processors) every item is guaranteed to come to its correct sorted position. It turns out that for both column sort as well as smoothsort, the amount of error decreases as n/p, the load per processor, is increased.

As noted in the preceding section, for $n \geq p^3$, column sort can be applied without recursion. This is due to the fact that after merging the balanced subcubes, every item has either been routed to the correct processor i, or it has been routed to one of processors $i-1$ and $i+1$. Thus, the sort can be completed by performing local sorts followed by merge-and-split operations between odd and even pairs of adjacent processors. As a simple optimization, it is more efficient to sort the ith largest set of n/p items to the processor with the ith largest standard Gray code instead of processor i. This permits the merge-and-split operations to be performed between adjacent processors.

The main difference between column sort and smoothsort is that the "balancing" operation performed by smoothsort (the cost of which is related to that of column sort by a small constant factor) guarantees an asymptotically smaller degree of error. For this reason, smoothsort can be applied without recursion over a larger range of values of n and p, namely, for $n \geq p^2 \lg p$. Interestingly, the balancing operation of smoothsort is based upon a simple variant of merge-and-split: the "merge-and-unshuffle" operation. Essentially, the best way to guarantee similarity between the distribution of ranks of the items at a given pair A and B of adjacent processors is to merge the two sets of items, assign the odd-ranked items in the resulting sorted list to processor A (say), and the even-ranked items to processor B. This effect is precisely that of a merge-and-unshuffle operation. The balancing operation of smoothsort amounts to performing $\lg p$ sets of such merge-and-unshuffle operations, one over each of the hypercube dimensions. As in the case of column sort, there are at least two ways to pipeline the balancing operation in order to take advantage of the CM-2's ability to communicate across all of the hypercube wires at once.

At high loads ($p^2 \leq n < p^3$), we felt that smoothsort might turn out to be competitive with sample-sort. Like column sort, however, the performance of smoothsort degrades (relative to that of other algorithms) at low to moderate loads ($p \leq n < p^2$), which was the overriding factor in our decision not to implement smoothsort. For unusually high loads ($n \geq p^3$), it is likely that column sort would slightly outperform smoothsort because of a small constant factor advantage in the running time of its balancing operation on the CM-2. It should be mentioned that for $n \geq p^{1+\epsilon}$, the asymptotic performance of smoothsort is the same as that of column sort, both in terms of arithmetic as well as communication. Smoothsort outperforms column sort for smaller values of n/p, however. For a detailed analysis of the running time of smoothsort, the reader is referred to [18].

Theoretical results. This subsection summarizes several "theoretical" sorting results—algorithms with optimal or near-optimal asymptotic performance but which remain impractical due to large constant factors and/or nonconstant costs that are not accounted for by the model of computation. In certain instances, a significant additional penalty must be paid in order to "port" the algorithm to the particular architecture provided by the CM-2.

Many algorithms have been developed for sorting on Parallel Random Access Machines (PRAMs). The fastest comparison-based sort is Cole's parallel merge sort [6]. This algorithm requires optimal $O(\lg n)$ time to sort n items on an n-node exclusive-read exclusive-write (EREW) PRAM. Another way to sort in $O(\lg n)$ time is to emulate the AKS sorting circuit [1]. In this case, however, the constants hidden by the O-notation are large.

If one is interested in emulating a PRAM algorithm on a fixed-connection network such as the hypercube or butterfly, the cost of the emulation must be taken into account. Most emulation schemes are based on routing. Since the cost of sample sort is only about three times the cost of a single routing operation, it seems unlikely that any direct emulation of a PRAM sorting algorithm will lead to a competitive solution.

For the hypercube and related networks such as the butterfly, cube-connected cycles, and shuffle-exchange, there have been recent asymptotic improvements in both the deterministic and randomized settings. A deterministic, $O(\lg n (\lg \lg n)^2)$-time algorithm for the case $n = p$ is described in [8]. An $O(\lg n)$-time algorithm that admits an efficient bit-serial implementation and also improves upon the asymptotic failure probability of the Reif-Valiant flashsort algorithm is presented in [16]. Unfortunately, both of these algorithms are quite impractical. The reader interested in theoretical bounds should consult the aforementioned papers for further references to previous work.

B Probabilistic Analysis of Sample Sort

This appendix analyzes the sizes of the buckets created by the sample sort algorithm from Section 5. Recall how buckets are created, a method we'll call *Method P*. First, each of the p processors partitions its n/p keys into s groups of n/ps and selects one candidate at random from each group. Thus, there are a total of exactly ps candidates. Next, the candidates are sorted, and every sth candidate in the sorted order is chosen to be a splitter. The keys lying between two successive splitters form a bucket. Theorem B.4 will show that it is unlikely that this method assigns many more keys than average to any one bucket.

The proof of Theorem B.4 uses three lemmas, the first two of which are well known in the literature. The first is due to Hoeffding.

Lemma B.1 *Let X_i be a random variable that is equal to 1 with probability q_i and to 0 with probability $1 - q_i$, for $i = 1, 2, \ldots, n$. Let $W = \sum_{i=1}^{n} X_i$, which implies that $E[W] = \sum_{i=1}^{n} q_i$. Let $q = E[W]/n$, and let Z be the sum of n random variables, each equal to 1 with probability q and to 0 with probability $1 - q$. (Note that $E[W] = E[Z] = qn$.) If $k \leq qn - 1$ is an integer, then*

$$\Pr[W \leq k] \leq \Pr[Z \leq k] \ .$$

□

Our second lemma is a "Chernoff" bound due to Angluin and Valiant [11].

Lemma B.2 *Consider a sequence of r Bernoulli trials, where success occurs in each trial with probability q. Let Y be the random variable denoting the total number of successes. Then for $0 \leq \gamma \leq 1$, we have*

$$\Pr[Y \leq \gamma r q] \leq e^{-(1-\gamma)^2 r q/2} \ .$$

Our third lemma shows that Method P can be analyzed in terms of another simpler method we call Method I. In Method I, each key of the n keys independently chooses to be a candidate with probability ps/n. For this method, the expected number of candidates is ps. The following lemma shows that upper bounds for Method I apply to Method P.

Lemma B.3 *Let S be a set of n keys, and let T denote an arbitrary subset of S. Let Y_P and Y_I denote the number of candidates chosen from T by Methods P and I, respectively. Then for any integer $k \leq (|T|ps/n) - 1$, we have*

$$\Pr[Y_P \leq k] \leq \Pr[Y_I \leq k] \ .$$

Proof: Let $\{S_i\}$ be the partition of keys used by Method P, that is, $S = \bigcup_{i=1}^{ps} S_i$ and $|S_i| = n/ps$. Define $T_i = S_i \cap T$, for $i = 1, 2, \ldots, ps$. Since $|T_i| \leq |S_i| = n/ps$, each set T_i contributes 1 candidate with probability $|T_i|ps/n$, and 0 candidates otherwise.

Now, define $|T|$ 0-1 random variables as follows. For each nonempty T_i, define $|T_i|$ random variables, where the first random variable is equal to 1 with probability $|T_i|ps/n$ and 0 otherwise, and the remaining $|T_i| - 1$ random variables are always 0.

Call the resulting set of $|T|$ random variables $X_1, \ldots, X_{|T|}$ (order is unimportant), and let Y_P be the random variable defined by $Y_P = \sum_{i=1}^{|T|} X_i$. Consequently,

$$E[Y_P] = \sum_{i=1}^{|T|} E[X_i] = \sum_{i=1}^{ps} |T_i|ps/n = |T|ps/n \ ,$$

and thus, Y_P is the random variable corresponding to the number of candidates chosen from the set T by Method P.

Define Y_I to be the sum of $|T|$ ps/n-biased Bernoulli trials. Note that Y_I is the random variable corresponding to the number of candidates chosen from the set T by Method I. Hence, by substituting $W = Y_P$ and $Z = Y_I$ into Hoeffding's inequality, we have

$$\Pr[Y_P \leq k] \leq \Pr[Y_I \leq k]$$

for $k \leq E[Y_P] - 1 = E[Y_I] - 1 = (|T|ps/n) - 1$. □

With Lemmas B.1 and B.3 in hand, we are prepared to prove the bound given by inequality 9.

Theorem B.4 *Let n be the number of keys in a sample sort algorithm, let p be the number of processors, and let s be the oversampling ratio. Then for any $\alpha \geq 1 + 1/s$, the probability that Method P causes any bucket to contain more than $\alpha n/p$ keys is at most $ne^{-(1-1/\alpha)^2 \alpha s/2}$.*

Proof: To prove that no bucket receives more than $\alpha n/p$ keys, it suffices to show that the distance l from any key to the next splitter in the sorted order is at most $\alpha n/p$. We begin by looking at a single key. We have $l > \alpha n/p$ only if fewer than s of the next $\alpha n/p$ keys in the sorted order are candidates. Let T denote this set of $\alpha n/p$ keys. Let Y_P denote the number of candidates in T, which are chosen according to Method P. Thus, $\Pr[l > \alpha n/p] \leq \Pr[Y_P < s]$.

We can obtain an upper bound on $\Pr[Y_P < s]$ by analyzing Method I instead of Method P, since by Lemma B.3, any upper bound derived for $\Pr[Y_I \leq s]$ also applies to $\Pr[Y_P \leq s]$, as long as $s \leq (|T|ps/n) - 1$, which holds for $\alpha \geq 1 + 1/s$. If the candidates are chosen according to Method I, then the number of candidates in the set T of $\alpha n/p$ keys has a binomial distribution, that is,

$$\Pr[Y_I = k] = \binom{r}{k} q^k (1-q)^{r-k} ,$$

where $r = \alpha n/p$ is the number of independent Bernoulli trials, $q = ps/n$ is the probability of success in each trial, and Y_I is the number of successes. The probability that fewer successes occur than expected can be bounded using the "Chernoff" bound

$$\Pr[Y_I \leq \gamma rq] \leq e^{-(1-\gamma)^2 rq/2} ,$$

which holds for $0 \leq \gamma \leq 1$. Substituting $r = \alpha n/p$, $q = ps/n$, and $\gamma = 1/\alpha$, we have

$$\begin{aligned} \Pr[l > \alpha n/p] &\leq \Pr[Y_P \leq s] \\ &\leq \Pr[Y_I \leq s] \\ &\leq e^{-(1-1/\alpha)^2 \alpha s/2}. \end{aligned}$$

The probability that the distance from *any* of the n keys to the next splitter is more than $\alpha n/p$ is at most the sum of the individual probabilities, each of which is bounded by $e^{-(1-1/\alpha)^2 \alpha s/2}$. Since there are are n keys, the probability that the distance from any key to the next splitter is greater than $\alpha n/p$ is therefore at most $ne^{-(1-1/\alpha)^2 \alpha s/2}$, which proves the theorem. □

Acknowledgments

Many people were involved in this project, and we would like to thank all of them. We would particularly like to thank Mark Bromley, Steve Heller, Bradley Kuszmaul and Kevin Oliveau of Thinking Machines Corporation for helping with various aspects of the CM-2, including the implementation of the send that was needed for the sample sort. We would also like to thank John Mucci of Thinking Machines and Rita of Rita's for his support and her inspiration.

References

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

[2] S. G. Akl. *Parallel Sorting Algorithms*. Academic Press, Toronto, 1985.

[3] K. Batcher. Sorting networks and their applications. In *Proceedings of the AFIPS Spring Joint Computing Conference*, volume 32, pages 307–314, 1968.

[4] G. Baudet and D. Stevenson. Optimal sorting algorithms for parallel computers. *IEEE Transactions on Computers*, C–27:84–87, 1978.

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

[6] R. Cole. Parallel merge sort. *SIAM Journal on Computing*, pages 770–785, 1988.

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

[8] R. E. Cypher and C. G. Plaxton. Deterministic sorting in nearly logarithmic time on the hypercube and related computers. In *Proceedings of the 22nd Annual ACM Symposium on Theory of Computing*, pages 193–203, May 1990.

[9] R. E. Cypher and J. L. C. Sanz. Cubesort: A parallel algorithm for sorting N data items with S-sorters. *Journal of Algorithms*, 13:211–234, 1992.

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

[11] T. Hagerup and C. Rub. A guided tour of Chernoff bounds. *Information Processing Letters*, 33(6):305–308, February 1990.

[12] William L. Hightower, Jan F. Prins, and John. H. Reif. Implementations of randomized sorting on large parallel machines. In *Proceedings of the Symposium on Parallel Algorithms and Architectures*, June 1992.

[13] 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, November 1983.

[14] Lennart Johnsson. Combining parallel and sequential sorting on a boolean n-cube. In *Proceedings of the International Conference on Parallel Processing*, pages 444–448, 1984.

[15] F. T. Leighton. Tight bounds on the complexity of parallel sorting. *IEEE Transactions on Computers*, C–34(4):344–354, April 1985.

[16] T. Leighton and G. Plaxton. A (fairly) simple circuit that (usually) sorts. In *Proceedings of the 31st Annual Symposium on Foundations of Computer Science*, pages 264–274, October 1990.

[17] D. Nassimi and S. Sahni. Parallel permutation and sorting algorithms and a new generalized connection network. *Journal of the ACM*, 29(3):642–667, July 1982.

[18] C. G. Plaxton. Efficient computation on sparse interconnection networks. Technical Report STAN-CS-89-1283, Stanford University, Department of Computer Science, September 1989.

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

[20] S. R. Seidel and W. L. George. Binsorting on hypercubes with d-port communication. In *Proceedings of the Third Conference on Hypercube Concurrent Computers*, pages 1455–1461, January 1988.

[21] Thomas M. Stricker. Supporting the hypercube programming model on mesh architectures (a fast sorter for iWarp tori). In *Proceedings of the Symposium on Parallel Algorithms and Architectures*, June 1992.

[22] K. Thearling and S. Smith. An improved supercomputer sorting benchmark. In *Proceedings Supercomputing '92*, pages 14–19, November 1992.

[23] B. A. Wagar. Hyperquicksort: A fast sorting algorithm for hypercubes. In M. T. Heath, editor, *Hypercube Multiprocessors 1987 (Proceedings of the Second Conference on Hypercube Multiprocessors)*, pages 292–299, Philadelphia, PA, 1987. SIAM.

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

[25] Marco Zagha and Guy E. Blelloch. Radix sort for vector multiprocessors. In *Proceedings Supercomputing '91*, pages 712–721, November 1991.