1

a system of thousands of processors could be connected in this manner. A more reasonable approach would be to have a single large memory area which can be used simultaneously by all the processors. Processors could communicate with each other by manipulating the shared memory. Another strategy is to arrange the processors in some geometrically regular pattern and allow processors to communicate directly only with their neighbours. Some of the favoured patterns are 2 and 3-dimensional lattices, hypercubes and hexagonal arrays. These designs suffer in performance compared to the shared memory design because of the time required for non-adjacent processors to communicate. There are many other possible configurations. Rather than study them all we shall select a few and discuss algorithms for them in detail.

Another trend is toward large computer networks. That is, independent computer systems which have the ability to communicate with each other. An example would be a system of computerized libraries in which a user may request information from any library in the system. Again, we will not normally have every processor directly connected to every other processor. Such a system is called a distributed computation network. The configuration of these networks may change frequently so problems related to

The instruction set includes the data-routing commands `move-left`, `move-right`, `move-up`, and `move-down`. Using these, $2(n-1)$ data routing steps are required to send a message from a processor in one corner of the mesh to the processor in the opposite corner.

Each processor is assigned an index. The indexing scheme is dependent on the algorithm to be used. The most natural scheme is **row-major indexing** illustrated above. Another (less natural) is **shuffled row-major indexing**. If a processor has index i in row-major indexing then it has index $\text{shuf}(i)$ in shuffled row-major indexing. The shuffle function, $\text{shuf}(i)$, is obtained by interleaving the high-order bits with the low-order bits. More precisely,

$$\text{shuf}(i) = (i_{2k-1} \cdot i_{k-1} \cdot i_{2k-2} \cdot i_{k-2} \cdots i_{k+1} \cdot i_1 \cdot i_k \cdot i_0)_2,$$

where $i = (i_{2k-1} \cdot i_{2k-2} \cdots i_1 \cdot i_0)_2$ is the binary representation of i . It is assumed that n is a power of 2 and let $k = \log_2 n$. For $n=4$ the shuffled row-major indexing is illustrated below.

MULTIPROCESSOR ALGORITHMS FOR SOME
COMBINATORIAL PROBLEMS

BY

Michael Joseph McCloskey

A thesis submitted to the Faculty of Graduate
Studies and Research in partial fulfillment
of the requirements for the degree of
Master of Science

Department of Mathematics and Statistics
Carleton University
Ottawa, Ontario
November 1983

• Michael Joseph McCloskey

The undersigned recommend to the Faculty of Graduate Studies

and Research acceptance of the thesis

MULTIPROCESSOR ALGORITHMS FOR SOME COMBINATORIAL PROBLEMS

submitted by Michael McCloskey, B.Sc.

in partial fulfillment of the requirements for

the degree of Master of Science.

Thesis Supervisor

Chairman, Department of Mathematics and Statistics

Carleton University
November 30, 1983

Multiprocessor Algorithms For Some Combinatorial Problems

Michael Joseph McCloskey, M.Sc. Thesis.

Abstract

Multiprocessor algorithms are discussed in detail for the problems of sorting, and computing the connected components and minimum weight spanning trees of a graph. Several different models of computation are used.

Algorithms for the shared memory and cube-connected computers are given to sort N records with $N^{1+1/k}$ processors in time $O(k \log N)$ for any positive integer k . The mesh-connected computer will require N processors and $O(N^{1/2})$ time.

For the shared memory computer an algorithm is presented to find the connected components of a graph using $\max\{|E|, |V|\}$ processors in time $O(\log^2 |V|)$. A new algorithm for the shared memory computer is described to find the minimum weight spanning tree with $\max\{2|E|, |V|^{1+1/k}\}$ processors in time $O(k \log^2 |V|)$, k a positive integer.

A distributed computation network algorithm is presented to find the minimum weight spanning tree in $3|V| \log |V|$ communication steps.

Acknowledgements

I would like to thank my advisor, Dr. Brian C. Mortimer, for his help, guidance, and understanding; and the Carleton University Department of Mathematics and Statistics for the generous use of their word processing facilities.

Table of Contents

<u>Acceptance Sheet</u>	ii.
<u>Abstract</u>	iii.
<u>Acknowledgements</u>	iv.
<u>Table of Contents</u>	v.
 <u>Chapter 0 Introduction</u>	
§0.0	1.
§0.1 Preliminary Definitions and Models of Computation.	5.
§0.2 Miscellaneous	11.
 <u>Chapter 1 Sorting</u>	
§1.0 Introduction	15.
§1.1 Sorting on a Shared Memory Computer	17.
§1.2 Sorting on a Cube-Connected Computer	35.
§1.3 Sorting on a Mesh-Connected Computer	52.
 <u>Chapter 2 Connected Component Algorithms For a Shared Memory Computer</u>	
§2.1	58.
§2.2	65.
 <u>Chapter 3 Minimum Spanning Tree Algorithms For a Shared Memory Computer</u>	
§3.1	70.
§3.2	75.
 <u>Chapter 4 An Algorithm For the Minimum Weight Spanning Tree of a Distributed Computation Network</u>	
§4.1 Introduction	83.
§4.2	89.
 <u>References</u>	119.

CHAPTER 0: INTRODUCTION

SECTION 0.0

The history of computer hardware is dominated by two major trends: decreasing cost and increasing speed. While the trend of decreasing cost is likely to continue, some writers [ASS] are suggesting theoretical reasons why transistors cannot be made very much faster than the fastest experimental ones already in existence. For the "increasing speed trend" to continue computers of the future must incorporate large scale parallelism into their designs. Decreasing hardware costs will facilitate this change.

Indeed, some parallel processor systems have been built. For example, the Illiac IV system contains 64 processors and the MPP system contains 16,384 processors [HLS]. In this thesis we will be concerned with algorithms for multiple processor systems with many rudimentary processing elements. The number of processors required will be a function of the size of the input problem.

At the present time it is not known how to best arrange communication between the processors. The ideal would be to allow each pair of processors to communicate freely through a direct interprocessor link. However, it is doubtful that

a system of thousands of processors could be connected in this manner. A more reasonable approach would be to have a single large memory area which can be used simultaneously by all the processors. Processors could communicate with each other by manipulating the shared memory. Another strategy is to arrange the processors in some geometrically regular pattern and allow processors to communicate directly only with their neighbours. Some of the favoured patterns are 2 and 3-dimensional lattices, hypercubes and hexagonal arrays. These designs suffer in performance compared to the shared memory design because of the time required for non-adjacent processors to communicate. There are many other possible configurations. Rather than study them all we shall select a few and discuss algorithms for them in detail.

Another trend is toward large computer networks. That is, independent computer systems which have the ability to communicate with each other. An example would be a system of computerized libraries in which a user may request information from any library in the system. Again, we will not normally have every processor directly connected to every other processor. Such a system is called a distributed computation network. The configuration of these networks may change frequently so problems related to

minimizing communication costs arise. This is the topic of chapter 4.

In chapter 1 the sorting problem will be discussed. A single algorithmic approach will be adapted to three multiprocessor designs: the shared memory computer, the cube-connected computer, and the mesh-connected computer. For the shared memory computer we discuss Hirschberg's [HIR] family of stable sorting algorithms which use $N^{1+1/k}$ processors to sort N records in time $O(k \log N)$, where k is an arbitrary positive integer. Next we describe Nassimi and Sahni's [NAS] adaptation of these algorithms to the cube-connected computer. By using Batcher's bitonic merge [BAT] Nassimi and Sahni's algorithms obtain the same time-processor complexity as Hirschberg's algorithms. The restrictive structure of the mesh-connected computer decreases the efficiency drastically. The algorithm, due to Thompson and Kung [THK], requires n^2 processors in a $n \times n$ array to sort n^2 records in time $O(n)$.

Chapter 2 deals with shared memory computer algorithms for finding the connected components of a graph. First, the algorithm of Hirschberg, et al. [HCS] is given. For a N -vertex graph $N \lceil N/\log N \rceil$ processors and $O(\log^2 N)$ time

suffice. This algorithm uses an adjacency matrix to represent the graph. By using an adjacency list we show the processor needs can be reduced to $\max(2M, N)$, where M is the number of edges in the graph, and yet retain the same time complexity. The method is based on that of Savage and Ja'Ja' [SAJ], but our data structure is more space efficient and in some cases uses fewer processors.

Chapter 3 deals with two minimum weight spanning tree algorithms for shared memory computers. The first algorithm is that of Savage and Ja'Ja' [SAJ]. It uses N^2 processors and $O(\log^2 N)$ time. This algorithm is similar to the algorithms of chapter 2. Next, we present a new algorithm which uses only M processors and has time complexity $O((\log^3 N)/\log(2M/N))$. The sorting algorithms of chapter 1 are used as subroutines.

As mentioned, chapter 4 is concerned with distributed computation networks. Section 4.1 gives an introduction and section 4.2 describes a minimum weight spanning tree algorithm. This algorithm is an improved version of that of Gallagher, Humblet, and Spira [GHS]. Our algorithm requires at most $3N \log N$ communication steps whereas Gallagher, Humblet, and Spira's algorithm requires $5N \log N$ communication steps.

SECTION 0.1: PRELIMINARY DEFINITIONS AND

MODELS OF COMPUTATION

The multiprocessor designs to be used will now be defined precisely.

(1) Shared Memory Computer (SMC)

A SMC has a large shared memory which may be used by every processor. Each processor is capable of elementary logic and arithmetic operations. All the processors are identical. As in all the designs to be discussed, each processor is identified by a unique index. The i^{th} processor will be referred to as $P[i]$.

The SMC is **synchronous** meaning that all the processors execute the same instruction simultaneously. Usually the operands of the instruction are a function of the index of the processor. For example, an algorithm to add 1 to each element of the $N \times 1$ array X is for each processor $P[i]$, $0 \leq i < N$, to execute the instruction

$$X[i] := X[i] + 1.$$

Unless otherwise indicated the index "i" will refer to the processor index

Few SMC systems have been built and these have a relatively small number of processors. The ICL DAP system has 4096 processors. The SMC design is popular among theorists but technological advances are necessary before one with a large number of processors can be constructed.

Usually we will allow more than one processor to read a single memory location simultaneously. This is called a memory-read conflict. A memory-write conflict is when more than one processor attempts to simultaneously write into a single memory location. Memory-write conflicts are not normally allowed.

(2) Mesh-Connected Computer (MCC)

The MCC is a much more restricted model than the SMC. It consists of an $n \times n$ array of processors with each processor connected to its nearest neighbours.

The MCC is synchronous.

The instruction set includes the data-routing commands move-left, move-right, move-up, and move-down. Using these, $2(n-1)$ data routing steps are required to send a message from a processor in one corner of the mesh to the processor in the opposite corner.

Each processor is assigned an index. The indexing scheme is dependent on the algorithm to be used. The most natural scheme is **row-major indexing** illustrated above. Another (less natural) is **shuffled row-major indexing**. If a processor has index i in row-major indexing then it has index $\text{shuf}(i)$ in shuffled row-major indexing. The shuffle function, $\text{shuf}(i)$, is obtained by interleaving the high-order bits with the low-order bits. More precisely,

$$\text{shuf}(i) = (i_{2k-1} \cdot i_{k-1} \cdot i_{2k-2} \cdot i_{k-2} \cdots i_{k+1} \cdot i_1 \cdot i_k \cdot i_0)_2,$$

where $i = (i_{2k-1} \cdot i_{2k-2} \cdots i_1 \cdot i_0)_2$ is the binary representation of i . It is assumed that n is a power of 2 and let $k = \log_2 n$. For $n=4$ the shuffled row-major indexing is illustrated below.

The significance of the different indexing schemes will be discussed in section 1.3.

The MPP multiprocessor system [HLS] is a mesh-connected computer. It uses 16,384 microprocessors connected in a 128 by 128 array. The MPP was designed for satellite image processing and is extremely fast. Over 1.8 billion 32-bit floating-point multiplications can be performed per second.

(3) Cube-Connected Computer (CCC)

A CCC has 2^k processors, for some positive integer k . $P[i]$ is connected to $P[j]$, $i \neq j$, if and only if the binary representations of i and j differ in only one bit. Thus each processor is connected to exactly k others. The processors may be viewed as being located at the vertices of a k -dimensional cube.

The CCC is synchronous. It is a generalization of the MCC, but is much more powerful. The maximum number of data routing steps to send a message between any two processors is only k .

In a 3-dimensional universe, building a large cube-connected computer is difficult. Recent work [LOC] [HLS] suggests that a $2^{16}=65,536$ processor system is possible.

(4) Distributed Computation Network

This model of computation is very different from the previous models. Let $G=(V,E)$ be a connected graph. The distributed computation network will be identified with G . At each vertex $i \in V$ there is the processor $P[i]$. The processor $P[i]$ will be identified with its vertex i . For vertices $i, j \in V$, $P[i]$ is connected to $P[j]$ if and only if edge $(i,j) \in E$. The distributed computation network may be used to solve graph problems related to the graph G .

A distributed computation network is asynchronous. Each processor executes the same algorithm.

Figure An eight vertex distributed computation network.

A DCN corresponds to a group of computers which have facilities to communicate with one another. To each edge we may assign a communication cost. In a constantly changing

network it may be necessary to compute the minimum cost spanning tree of the DCN in order to minimize communication costs.

SECTION 0.2: MISCELLANEOUS

To write pseudo-code for a multiprocessor will require some special constructs. Let $B[i]$, $0 \leq i < N$, be a boolean expression and $A[i]$ be some operation which may be performed by processor $P[i]$. Suppose that for each processor $P[i]$ we wish to perform operation $A[i]$ on $P[i]$ if and only if $B[i]$. This may be written as

```
cobegin B[i]
    A
coend
```

or alternately

```
A[i] : (B[i]).
```

If $B[i]$ is omitted then $A[i]$ is performed for all i , $0 \leq i < N$. These constructs are appropriate only for synchronous machines.

Often we will have to compute an expression such as

```
a[0] # a[1] # ... # a[N-1]
```

where $\#$ is a associative operator which can be analysed in constant time by a single processor. For example, if $b \# c = \max(b, c)$ then

```
a[0] # a[1] # ... # a[N-1] = max(a[0], a[1], ..., a[N-1]).
```

Other such associative operators are addition and multiplication on the real line.

Given a single processor such an expression can be computed in $O(N)$ time. We will show a simple way that $O(N)$ processors of a shared memory computer can compute the expression in $O(\log N)$ time ($\log N := \lceil \log_2 N \rceil$).

Assume $\#$ is addition and let $n = \log N$. The array $a[0], a[1], \dots, a[N-1]$ is stored in the shared memory. The processors are $P[0], P[1], \dots, P[N-1]$. (Actually only the $\lceil N/2 \rceil$ even-indexed processors are required.)

For $i \in Z_N$ ($Z_N := \{0, 1, \dots, N-1\}$) let

$$i^{(b)} := (i_{n-1} \dots i_{b+1} \cdot \bar{i}_b \cdot i_{b-1} \dots i_0)_2$$

where $\bar{i}_b := 1$, if $i_b = 0$

0, if $i_b = 1$,

$0 \leq b < n$. (That is, $i^{(b)}$ has the same binary

representation is i except the b^{th} bit, i_b , is changed.

Generalizing, $i^{(b)(c)}$ is i with i_b changed and then i_c

changed. The operation is involutory: $i^{(b)(b)} = i$.) The

algorithm may be described as follows. (Assume $a[i] = 0$ for $i \geq N$.)

```

begin <<Compute  $\sum a[i] : 0 \leq i < N$ . Answer is in  $a[0]$ >>
  for b := 0 to n-1 do
    cobegin  $i_b = 0$ ,  $i \in V$ 
       $a[i] := a[i] + a[i^{(b)}]$ 
    coend
  endfor
end
```

The algorithm may be visualized as a binary tree (below).

Figure: Case of $N=11$.

It can be shown that such expressions cannot be computed faster than $O(\log N)$ time with any number of processors [KUC].

Suppose we have a N processor cube-connected computer. In each processor $P[i]$ we store $a[i]$. Since processors $P[i]$ and $P[i^{(b)}]$, $0 \leq i < N$, are connected the above algorithm may be used on a CCC as well.

A mesh-connected computer requires $O(N^{\frac{1}{2}})$ time to sum N numbers.

When speaking of time complexity it is customary to use the "big O " notation. A similar notation will be used for processor complexity for shared memory computers.

Definition Let $f, g: \mathbb{Z}^+ \rightarrow \mathbb{Z}^+$ be functions. Then

$g \leq_p O(f)$ if and only if there exist constants K and L such that

$$g(n) \leq f(n) + K$$

for all $n > L$.

(Recall that $g=O(f)$ if and only if there exist constants J , K , and L such that

$$g(n) \leq Jf(n) + K$$

for $n > L$.)

The " O_p " notation is **not** symmetric. That is, $g=O_p(f)$ does not imply $f=O_p(g)$.

With the " O " notation constant factors are not important but with the " O_p " notation they are important:

$$g=O(f) \Rightarrow Jg=O(f), \quad g=O(Jf)$$

for any $J > 0$. But if $g \leq O_p(f)$ then $Jg \leq O_p(f)$ is guaranteed only if $J \leq 1$.

In the previous example of computing the sum of N elements on a SMC we stated that $N/2$ processors were required. However, we could perform the same algorithm with $N/4$ processors in twice the time by "time sharing". In contrast using N or $2N$ processors would not improve the algorithm's speed. Thus it is appropriate to say $O_p(N/2)$ processors are required.

C[s]		...	C[i]	...		...	C[a-1]
C[a]		...		...	C[x]	...	C[b-1]

The binary search will compute the minimal value of x ,

$a \leq x < b$, such that

$$C[i] \leq C[x].$$

If $C[b-1] < C[i]$ then $x := b$. Now

$$\text{rank}(i) := i + (x-a) = i - a + x.$$

Differentiating between the two cases above will ensure that SORT_LOGN is stable.

A bucket sort is applied:

$$C[\text{rank}(i)] := C[i]$$

for each $i \in Z_N$. This is the end of the q^{th} phase.

$O(\log^2 N)$ with $N^{1+1/\log_2 N} = 2N$ processors. The total number of steps used by all the processors is bounded by the product of the number of processors and the time required:

$$2N O(\log^2 N) = O(N \log^2 N).$$

Thus these parallel algorithms are less efficient in this global sense than their serial counterparts by a factor of $\log N$.

For each of the computer types to be discussed there are many possible sorting algorithms. We have chosen ours so that they all have the same "framework".

SECTION 1.1: SORTING ON A SHARED MEMORY COMPUTER

Recall that we wish to demonstrate that for each positive integer k there is an algorithm to sort N records using $O_p(N^{1+1/k})$ processors and time $O(k \log N)$. This will be shown by induction on k . The method is based on that of Hirschberg [HIR].

To begin, consider the case of $k=1$. We have $O_p(N^{1+1/1}) = O_p(N^2)$ processors and the desired time complexity is $O(\log N)$. Assume there are exactly N^2 processors: N processors for each record. Let $C = \langle C[0], C[1], \dots, C[N-1] \rangle$ be the list of records. The simple algorithm below will sort them in ascending order. For each record $C[i]$, $i \in \mathbb{Z}_N$, SORT_1 simply counts the number of records, $\text{rank}(i)$, which should appear before $C[i]$ in the sorted list.

Procedure $\text{SORT}_1[C]$

$\ll$ Algorithm to sort $C = \langle C[0], C[1], \dots, C[N-1] \rangle \gg$

begin

1. **for** $i \in \mathbb{Z}_N$ **do**

2. **begin**

3. $\text{rank}(i) := |\{ j : j \in \mathbb{Z}_N, C[j] < C[i] \}|$
 $+ |\{ j < i : j \in \mathbb{Z}_N, C[j] = C[i] \}|$

4. $C[\text{rank}(i)] := C[i]$

5. **end**

end SORT_1 .

The "for" loop is performed in parallel with N processors assigned to each record $C[i]$, $i \in Z_N$. To compute $\text{rank}(i)$ involves finding the cardinality of two subsets of Z_N . This may be done using an adaptation of the algorithm of section 0.2 (page 10) for computing the sum of N elements in time $O(\log N)$. After it is computed, rank is a permutation of Z_N . Line 4 performs the corresponding permutation of the elements of C . Because rank is a permutation there are no memory-write conflicts, however, numerous memory-read conflicts will occur.

A sorting algorithm is said to be **stable** if it does not reorder elements of equal value. That is, if $C[s]=C[t]$, $s < t$, before sorting, and if s' and t' are the "after-sorting" indices of the records corresponding to $C[s]$ and $C[t]$ respectively, then $s' < t'$.

SORT_1 is a stable sort because in the case above

$$\begin{aligned} t' - s' &= \text{rank}(t) - \text{rank}(s) \\ &= |\{i < t : C[i] = C[t]\}| - |\{i < s : C[i] = C[s]\}| \\ &= |\{s \leq i < t : C[i] = C[s] = C[t]\}| \\ &\geq |(\{s\})| = 1. \end{aligned}$$

Thus $s' < t'$.

For other values of k we will obtain a general method for defining a stable sorting algorithm, SORT_k , which uses

$\text{SORT}_{(k-1)}$ as a subroutine. Before proceeding with the induction, the case $k=2$ will be examined.

When $k=2$ there are $N^{1+1/k} = N^{3/2}$ processors and the desired time complexity is $O(k \log N) = O(\log N)$. Assume that N is a perfect square and let $n = N^{1/2}$. SORT_2 will arrange the records in the $n \times n$ array D and then sort each row of D using SORT_1 . For every $i, j, i' \in Z_n$, we compute $\text{rank}(i, j, i')$ which is the number of records in row i' which should appear before the record $D[i, j]$ ($D[i, j]$ is in the i^{th} row and j^{th} column) in the final sorted list. Next we sum

$$\text{rank}(i, j) := \sum_{i'} \{ \text{rank}(i, j, i') : i' \in Z_n \}.$$

Now $\text{rank}(i, j)$ is the final position of $D[i, j]$ in the sorted list, and so a bucket sort is applied.

Procedure SORT_2 [C,N]

Begin

```

1.  $n^2 = N$ . Arrange C as a nxn array, D, with
    $D[i,j] := C[in + j]$ ,
    $i,j \in \mathbb{Z}_n$ . Row i of D will be denoted  $D[i]$ ,  $i \in \mathbb{Z}_n$ .
   Consider each row of D as a sublist.
   Append first and last elements to each sublist:
    $D[i,-1] := -\infty$ , and  $D[i,n] := +\infty$ ,  $i \in \mathbb{Z}_n$ .

2. Sort each sublist using SORT_1.

3. for each record  $D[i,j]$ ,  $i,j \in \mathbb{Z}_n$  do
   for each sublist  $D[i']$ ,  $i' \in \mathbb{Z}_n$  do
     case
4.        $i' < i$  : find  $t \in \mathbb{Z}_n$  such that
                $D[i',t-1] \leq D[i,j] < D[i',t]$ 
5.        $i' = i$  :  $t := j$ 
6.        $i' > i$  : find  $t \in \mathbb{Z}_n$  such that
                $D[i',t-1] < D[i,j] \leq D[i',t]$ 
     endcase
7.        $\text{rank}(i,j,i') := t$ 
   endfor
8.    $\text{rank}(i,j) := \sum \{ \text{rank}(i,j,i') : i' \in \mathbb{Z}_n \}$ 
9.    $C[\text{rank}(i,j)] := C[in + j]$ 
10. end
end SORT_2.
```

At the end of line 7 $\text{rank}(i,j,i')$ will be equal to the number of records in $D[i']$ which should appear ahead of $D[i,j]$ in the sorted list. The "case" clause ensures that

the sort is stable. Line 8 sums all the rank entries related to $D[i,j]$ and line 9 performs the bucket sort.

We will show that the resource requirements are as claimed. Recall there are $N^{3/2}$ processors available.

Line 2 applies SORT_1 to n lists of length n . This requires $n^2 \cdot n = N^{3/2}$ processors and $O(\log n) = O(\log N)$ time. The

interior of the inner "for" loop must be executed

$n^3 = N^{3/2}$ times. Therefore one processor may be assigned

to each iteration so that all the iterations may be

performed in parallel. In the case where $i=i'$ the loop may

be performed in constant time. Otherwise we must execute

lines 4 or 6. These are easily done with a binary search

subroutine in $O(\log n) = O(\log N)$ time. For each iteration of

line 8, in which a sum of n elements must be computed, n

processors will be available. This requires $O(\log N)$ time.

Finally, line 9 may be executed in constant time with N

processors. Thus SORT_2 requires $O_p(N^{3/2})$ processors

and $O(\log N)$ time.

A general strategy for constructing SORT_k from SORT_{(k-1)}, $k > 1$, is as follows.

Procedure SORT_k [C,N]

Begin

1. <<Divide>> Divide C into $N^{1/k}$ sublists of $N^{(k-1)/k}$ elements each.

2. <<SORT_(k-1)>> Sort each sublist using SORT_(k-1).

3. <<Search>> Compute rank(i,j,i') for each i, j, and i';
 $0 \leq i, i' < N^{1/k}$, $0 \leq j < N^{(k-1)/k}$. (As in SORT_2).

4. <<Sum>> $\text{rank}(i,j) := \sum \{ \text{rank}(i,j,i') : 0 \leq i' < N^{1/k} \}$
 for all $i, j \in Z_n$.

5. <<Bucket Sort>> $C[\text{rank}(i,j)] := C[in + j]$
 for all $i, j \in Z_n$.
end SORT_k.

The assumed induction hypothesis is that SORT_(k-1) can stably sort a list of M elements in time $O((k-1)\log M)$ using $M^{1+1/(k-1)}$ processors.

Line 2 applies SORT_(k-1) to lists of length $N^{(k-1)/k}$. This requires

$$(N^{(k-1)/k})^{1+1/(k-1)} = N^{(k-1)/k \cdot (k/k-1)} = N$$

processors and time $O((k-1)\log[N^{(k-1)/k}]) \leq O((k-1)\log N)$.

To sort $N^{1/k}$ such lists in parallel will require $N^{1+1/k}$ processors and $O((k-1)\log N)$ time.

A total of $N^{1+1/k}$ values of rank(i,j,i') must be computed. Therefore one processor is assigned to each

computation. The time required will be $O(\log N^{(k-1)/k})$.
 (the time required for a binary search on all sublists).

In line 4 there are N sums, each of $N^{1/k}$ elements,
 which must be computed. Using $N^{1+1/k}$ processors this
 requires $O(\log N^{1/k}) = O((1/k)\log N)$ time.

Finally the bucket-sort requires constant time and N
 processors.

Hence using $N^{1+1/k}$ processors we obtain a time bound
 of $O((k-1)\log N + ((k-1)/k)\log N + (1/k)\log N)$
 $= O((k-1 + 1 - 1/k + 1/k)\log N)$
 $= O(k\log N)$

as required. The validity of the algorithm and the fact
 that it is stable is easily verified.

In the preceding algorithm descriptions there has been
 considerable vagueness on certain aspects of implementation.
 For instance, which processor is assigned to which process
 is not specified and the synchronicity of the processors is
 not discussed. The algorithms above are sufficiently simple
 that such questions can be answered easily by the reader.
 These questions will be addressed explicitly when the
 answers are necessary or particularly interesting.

```

    << Move C-values to proper columns. >>
    for b := q downto 0 do
        if C[i] ≠ null and (rank[i])b ≠ ib
            then
                C[i(b)] := C[i]
                rank[i(b)] := rank[i]
            endif
        endfor

        r := q
        q := q + m
    end << Main Loop >>

    << Move the sorted list to the first row. >>
    for b := n to n+m-1 do
        C[i(b)] := C[i] : ( C[i] ≠ null )
    endfor
coend CUBE_SORT.

```

The procedure COUNT(r) operates on each 2^r -block independently and in parallel. Each 2^r -block contains two sorted lists (of length 2^r), one in the S-registers, the other in the T-registers. COUNT(r) will find the rank of the elements in the S-list with respect to the T-list.

The 2^r -block containing P[i] is a right-block if

$$\lfloor i/2^n \rfloor < \lfloor i/2^r \rfloor \bmod 2^m,$$

and a left-block if

$$\lfloor i/2^n \rfloor > \lfloor i/2^r \rfloor \bmod 2^m.$$


```

Procedure RECURSIVE_SORT_LOGN (C,N)
Begin
    if N>2
    then
        RECURSIVE_SORT_LOGN( <C[0],C[1],...,C[(N/2)-1]>, N/2 )
        RECURSIVE_SORT_LOGN( <C[N/2],...,C[N-1]>, N/2 )
        merge <C[0],...,C[(N/2)-1]> and <C[N/2],...,C[N-1]>
    else << N=2, apply SORT_1 >>
        C[0] := min( C[0], C[1] )
        C[1] := max( C[0], C[1] )
    endif
end RECURSIVE_SORT_LOGN.

```

To make the algorithm efficient it must be made non-recursive. The non-recursive version, SORT_LOGN, will have $\log N$ phases. At the beginning of the q^{th} phase, $0 < q \leq \log N$, there will be $N/2^{q-1}$ sorted sublists of 2^{q-1} elements each. Each sublist will then be merged with one other sublist. Records $C[i]$ and $C[j]$, $i, j \in \mathbb{Z}_N$, will be in the same 2^{q-1} -sublist if and only if the binary representations of i and j differ only in the $q-1$ least significant bits. Thus the first record in the sublist containing record $C[i]$ is $C[s]$ where

$$s := (i_{k-1} \cdot i_{k-2} \cdot \dots \cdot i_q \cdot i_{q-1} \cdot 0 \cdot 0 \cdot \dots \cdot 0)_2$$

and the last record is $C[t-1]$, where

$$t := s + 2^{q-1}.$$

This list will be merged with the list whose indices run from a to $b-1$, where

$$a := s^{(q-1)}$$

and

$$b := a + 2^{q-1}.$$

(Recall that $s^{(q-1)}$ is the number whose binary representation is the same as s except the $(q-1)$ -bit has been changed.)

The processor $P[i]$ will be assigned to the record $C[i]$. In order to merge the lists, $P[i]$ will compute $\text{rank}(i)$, the new position of $C[i]$. This will be done by performing a binary search on the list $\langle C[a], \dots, C[b-1] \rangle$.

Suppose $i_{q-1} = 1$. Then

$$a < b-1 < b = s \leq i < t.$$

$C[a]$	...	$C[x]$	...		...	$C[b-1]$
$C[b]$	...		...	$C[i]$	...	$C[t-1]$

The binary search will compute the minimal value of x , $a \leq x < b$, such that

$$C[i] < C[x].$$

If $C[b-1] \leq C[i]$ then $x = b$.

Now

$$\text{rank}(i) := i - b + x.$$

If $i_{q-1} = 0$ then

$$s \leq i < t = a < b$$

$C[s]$		...	$C[i]$	...		...	$C[a-1]$
$C[a]$	-	...		...	$C[x]$	...	$C[b-1]$

The binary search will compute the minimal value of x ,

$a \leq x < b$, such that

$$C[i] \leq C[x].$$

If $C[b-1] > C[i]$ then $x := b$. Now

$$\text{rank}(i) := i + (x-a) = i - a + x.$$

Differentiating between the two cases above will ensure that SORT_LOGN is stable.

A bucket sort is applied:

$$C[\text{rank}(i)] := C[i]$$

for each $i \in Z_N$. This is the end of the q^{th} phase.

Procedure SORT_LOGN [C,N]

<< Sort the N-element list C using N processors and $O(\log^2 N)$ time.>>

Shared Memory Variables C, N, rank; << All other variables are >>

Begin << local to the processor. >>

cobegin $i \in \mathbb{Z}_N$, i even

Sort the list $\langle C[i], C[i+1] \rangle$.

coend

cobegin $i \in \mathbb{Z}_N$

k := logN

for q := 2 to k do

<< C[i] is in the list of records whose indices differ>>

<< from i only in the q-1 least significant bits >>

a := $(i_{k-1} \cdot i_{k-2} \cdot \dots \cdot i_q \cdot \sim i_{q-1} \cdot 0 \cdot 0 \cdot \dots \cdot 0)_2$

a := min(a, N)

b := min($a + 2^{q-1}$, N)

if $i_{q-1} = 0$ then rank(i) := i - a

else rank(i) := i - b;

<< find the rank of C[i] in the >>

<< list $\langle C[a], \dots, C[b-1] \rangle$ >>

for q iterations do << binary search >>

x := $\lfloor \frac{1}{2}(b+a) \rfloor$

if $C[x] < C[i]$ or

($C[x] = C[i]$ and $i_{q-1} = 1$)

then a := x

else b := x;

endfor

rank(i) := rank(i) + x

$C[\text{rank}(i)] := C[i]$

endfor

coend

end SORT_LOGN.

Note that this algorithm works even when N is not a power of 2.

The reader can verify that $O(\log^2 N)$ time is required.

A more careful analysis shows that at most

$$\frac{1}{2} \log N (\log N + 1) + 2$$

main memory read cycles and

$$2 \log N - 1$$

main memory write cycles are required.

For other values of k the recursion may be unfolded in a similar manner. Assume N to be power of 2 and let $n = \log N$. In addition assume k to be a divisor of n . Let $m = n/k$. We will require

$$N^{1+1/k} = 2^{n(1+m/n)} = 2^{n+m}$$

processors and $O(k \log N) = O(kn)$ time.

When SORT_k is applied, C is broken into $N^{1/k} = 2^m$ sublists, each containing 2^{n-m} elements. $\text{SORT}_{(k-1)}$ will now be applied. Each sublist will be split into

$$\begin{aligned} (2^{n-m})^{\frac{1}{(k-1)}} &= 2^{\frac{n-m}{(n/m)-1}} \\ &= 2^{(n-m) \left(\frac{m}{n-m} \right)} \\ &= 2^m \end{aligned}$$

sublists. Thus there are $2^m \cdot 2^m = 2^{2m}$ sublists in

total, each with 2^{n-2m} elements. In general, $\text{SORT}_{(k-t)}$

will split C into $2^{(t+1)m}$ sublists, each with $2^{n-(t+1)m}$ elements. $\text{SORT}_{(k-(t+1))}$ is then called to sort each sublist. SORT_2 will operate on

$$2^{((k-2)+1)m} = 2^{(k-1)m} = 2^{n-1} = \frac{1}{2}N$$

sublists each with only two elements. Thus the algorithm proceeds in k phases. At the beginning of the s^{th} phase, $0 \leq s < k$, there are 2^{n-sm} sorted sublists of 2^{sm} elements. During this phase the sublists will be merged in groups of 2^m yielding $2^{(n-sm)-m} = 2^{n-(s+1)m}$ sorted sublists.

Let $M = 2^{n+m}$. The processors are indexed from 0 to $M-1$. Each list element is assigned 2^m processors. Processor $P[i']$ is assigned to record $C[i]$ if and only if

$$i = i' \bmod 2^n,$$

$$i \in \mathbb{Z}_N, i' \in \mathbb{Z}_M.$$

It will be useful to represent numbers in base 2^m . With i and i' as above, if

$$i = (i_k i_{k-1} \dots i_0)_{2^m}$$

then

$$i' = (0 i_{k-1} \dots i_0)_{2^m}.$$

For the remainder of this section all number representations are base 2^m .

Each sublist is identified by an index. At the beginning of phase s , $0 \leq s < k$, with i and i' as above, record $C[i]$ is in the sublist with index

$$\begin{aligned} \text{list}(i) &:= \lfloor i/2^{sm} \rfloor \\ &= (i_{k-1} \cdot i_{k-2} \cdot \dots \cdot i_{s+1} \cdot i_s). \end{aligned}$$

This sublist contains the records indexed from

$$\text{list}(i) \cdot 2^{sm} = (i_{k-1} \cdot i_{k-2} \cdot \dots \cdot i_s \cdot 0 \cdot 0 \cdot \dots \cdot 0)$$

to

$$\text{list}(i) \cdot 2^{sm} + 2^{sm-1} = (i_{k-1} \cdot \dots \cdot i_s \cdot 2^{m-1} \cdot 2^{m-1} \cdot \dots \cdot 2^{m-1}).$$

Sublist $\text{list}(i)$ will be merged with the 2^{m-1} other sublists having indices which differ only in the least significant 2^m -digit. These 2^m sublists constitute a block.

As mentioned above, there are 2^m processors assigned to each record. Record $C[i]$ is assigned the processors indexed as $(r \cdot i_{k-1} \cdot \dots \cdot i_0)$, for each r , $0 \leq r < 2^m$. Thus $C[i]$ has one processor for each sublist in its block. To merge the sublists we again must compute the function "rank". For each r , $0 \leq r < 2^m$, let

$$\text{rank_list}(r) := (i_{k-1} \cdot \dots \cdot i_{s+1} \cdot r)$$

(rank_list is dependent on i). The sublist whose index is $\text{rank_list}(r)$ is one of those with which sublist $\text{list}(i)$ is to be merged. There are three cases.

- (1) If $\text{rank_list}(r) < \text{list}(i)$, then $\text{rank}(i, \text{rank_list}(r))$
will be the number of records, $C[j]$, in sublist
 $\text{rank_list}(r)$ such that $C[j] \leq C[i]$.
- (2) If $\text{rank_list}(r) = \text{list}(i)$, then $\text{rank}(i, \text{rank_list}(r))$ will
be set to $(i_{s-1} \dots i_0) = i \bmod 2^{sm}$.
- (3) If $\text{rank_list}(r) > \text{list}(i)$, then $\text{rank}(i, \text{rank_list}(r))$
will be the number of records, $C[j]$, in sublist
 $\text{rank_list}(r)$ such that $C[j] < C[i]$.

Processor $P[(r.i_{k-1} \dots i_0)]$, $0 \leq i < 2^m$, will compute
 $\text{rank}(i, \text{rank_list}(r))$ using a binary search

Next, the rank values must be summed:

$$\text{rank}(i, \text{rank_list}(0)) := \sum \{ \text{rank}(k, \text{rank_list}(r)) : 0 \leq r < 2^m \}.$$

Finally, a bucket sort is applied:

$$C[\text{rank}(i, \text{rank_list}(0))] := C[i], \quad i \in \mathbb{Z}_N.$$

The block has now been merged and sorted. Phase s is
complete.

Procedure SORTK [C,N,k]

<< Algorithm to sort the N-element list C >>

<< N is a power of two and k divides logN. >>

Shared Memory Variables C, N, k, rank

begin

n := logN

m := n/k

M := 2^{n+m}

for s := 0 to k-1 do

cobegin i' ΘZ_M

<< All numbers are base 2^m . >>

i := (0.i' i'_{k-1} .i' i'_{k-2}i' i'_0) << = (i i_{k-1}i i_0) >>

<< P[i'] is assigned to C[i]. >>

<< C[i] is in the sublist identified by the >>

<< index (i i_{k-1} .i i_{k-2}i i_s). It will be merged >>

<< with the 2^m sublists whose indices differ only >>

<< in the least significant 2^m -digit. >>

rank_list := (i i_{k-1}i i_{s+1} .i i_k)

<< Find the rank of C[i] w.r.t. list "rank_list" >>

compute rank(i, rank_list) << as in SORT_k >>

<< rank(i,rank_list) is the number of records in. >>

<< sublist rank_list which are smaller than C[i]. >>

<< See pages 31-32. >>

```

    << Sum all the rank-valuesi for C[i]. >>
    << Store the sum in rank(i, (ik-1 ... is+1.0)). >>
    for t := 0 to m-1 do
        if the tth binary bit of rank_list is 1
        then
            rank(i, rank_list - 2t) :=
                rank(i, rank_list - 2t) + rank(i, rank_list)
            exit loop
        endif;
    endfor
    << rank(i) is now in P[i]. >>
    << Bucket sort. >>
    if ik = 0
    then << rank list = (ik-1 ... is+1.0) >>
        C[ rank(i, ranklist) ] := C[i];
    coend
end SORTK.

```

SECTION 1.2: SORTING ON A CUBE-CONNECTED COMPUTER

The cube-connected computer has a more restrictive design than the shared memory computer, yet for the sorting problem the same complexity may be achieved. The algorithm CUBE_SORT to be given, due to Nassimi and Sahni [NAS], is an adaptation of Hirschberg's algorithm SORTK of the previous section. Although Hirschberg's work is mentioned in Nassimi and Sahni's article, it would appear that they were not aware of the similarity between the algorithms. This may be because Hirschberg does not describe his algorithms explicitly - he only shows the existence of such algorithms in a recursive form.

As in the previous section there are N records to be sorted: $\langle C[0], \dots, C[N-1] \rangle$. N is assumed to be a power of 2. Let $n = \log N$. For simplicity we assume that k divides n . (Other values for k are possible but the analysis obscures some of the essential points.) Let $m = n/k$. Like SORTK, CUBE_SORT will sort C with 2^{m+n} processors in $O(k \log N) = O(n^2/m)$ time. Let $M = 2^{m+n}$.

For $i, j \in \mathbb{Z}_M$, there is a direct communication link between $P[i]$ and $P[j]$ if and only if the binary representations of i and j differ in exactly one bit. Thus each processor is connected to $m+n$ other processors. For the sorting problem, $C[i]$ will initially be in $P[i]$,

$i \in \mathbb{Z}_N$. The processors may be considered to be in a $2^m \times 2^n$ rectangular array in row-major indexing. For any i , $0 \leq i < 2^{m+n}$, $P[i]$ is in row $\lfloor i/2^n \rfloor$ and column $i \bmod 2^n$. So if we write i in binary, then the row of $P[i]$ is $(i_{m+n-1} \dots i_n)_2$ and its column is $(i_{n-1} \dots i_0)_2$. Initially C is in the first row.

Define a 2^r -block as a sequence of 2^r consecutive processors of the form $P[j2^r+0], P[j2^r+1], \dots, P[j2^r+2^r-1]$, where $0 \leq j < 2^{n+m-r}$. For our purposes r will always be less than n , so that a 2^r -block will be contained in a single row. Each row is partitioned into 2^{n-r} different 2^r -blocks.

A 2^r -column-block is a sequence of 2^r adjacent columns of processors: columns $j2^r+0, j2^r+1, \dots, j2^r+2^r-1$, for some $j, 0 \leq j < 2^{n-r}$.

The indices of the processors in a 2^r -block vary only in the r least significant bits. The indices of the processors in a 2^r -column-block vary only in the m most significant bits and the r least significant bits.

If the processors are 2^r -blocked then $P[i]$ is diagonal if and only if

$$\lfloor i/2^n \rfloor \equiv \lfloor i/2^r \rfloor \pmod{2^m}.$$

If $P[i]$ is diagonal then so is every processor in its 2^r -block. This block is then called a **diagonal** 2^r -block.

Figure Example showing the diagonal 2^r -blocks when $n=3$, $m=2$, $r=1$.

Define the boolean function $\text{DIAG}[r,i]$ to be true if and only if $P[i]$ is diagonal under the 2^r -blocking.

The major obstacles to implementing SORTK on a CCC are computing "rank" and the bucket sort. Both these tasks would seem to require a central memory. Note that the procedure of summing the "rank" values may be implemented on a CCC as is.

As in SORTK, there are k phases. During phase s , $0 \leq s < k$, the processors will be in the 2^{sm} -blocking.

Associated with each column q , $q \in Z_N$, is a record from the list C . Each processor $P[i]$, $i \in Z_M$, has a register $C[i]$. At the beginning of phase s , $0 \leq s < k$, the record associated with column q will be contained in exactly one of the C -registers, $C[q^*]$ say, of that column (q^* will change during the algorithm). The other C -registers will be equal to "null". In this way each 2^{sm} -column-block contains a sublist of 2^{sm} records (that is, all the non-null C -registers in the column-block). The sublist is sorted. The task of phase s is to merge together the 2^m different sorted sublists of each $2^{(s+1)m}$ -column-block.

Each processor $P[i]$, $i \in Z_M$, also has two other registers $S[i]$ and $T[i]$. Phase s will begin with each column q , $q \in Z_N$, copying its record, $C[q^*]$, over all of its S -registers. Since processors $P[i]$ and $P[i^{(b)}]$ are connected, $0 \leq b < m+n$, this may be done by executing the following simple routine.

```

cobegin i ∈ ZM
  S[i] := C[i];
  for b := n to n+m-1 do
    S[i(b)] := S[i] : ( S[i] ≠ null )
  endfor
coend

```

(The statement $S[i^{(b)}] := S[i]$ is actually a compound statement involving instructions for two different processors, $P[i^{(b)}]$ and $P[i]$. Processor $P[i]$ will send the value of $S[i]$ along the direct interprocessor link between $P[i]$ and $P[i^{(b)}]$. Processor $P[i^{(b)}]$ will receive this value and store it in $S[i^{(b)}]$.) Thus in the S-registers of each 2^{sm} -block is contained the sublist corresponding to that 2^{sm} -column-block.

Under the 2^{sm} -blocking, each $2^{(s+1)m}$ -block contains exactly one diagonal 2^{sm} -block. Each diagonal 2^{sm} -block will copy the contents of its S-registers over the T-registers of every 2^{sm} -block contained in the same $2^{(s+1)m}$ -block. Within the $2^{(s+1)m}$ -block containing processor $P[i]$, $i \in Z_M$, we may determine which 2^{sm} -block contains $P[i]$ by the value of bits $i_{(s+1)m-1}$ down to i_{sm} . If $P[i]$ is in a diagonal 2^{sm} -block then we wish to write $S[i]$ over $T[j]$ for each $j \in Z_M$ whose binary representation is the same as that of i except (perhaps) in bits $j_{(s+1)m-1}$ through j_{sm} .

The subroutine $COUNT[sm]$ is now called. Each non-diagonal 2^{sm} -block contains two sorted lists: one in the S-registers and one in the T-registers. For each $S[i]$, $i \in Z_M$, of a given 2^{sm} -block, $COUNT[sm]$ will use the processors of that block to find the number of elements of

the T-list which should appear before S[i] in the final sorted list. This number will be stored as rank[i]. If P[i] is in row p and column q, then rank[i] corresponds to rank(q, rank_list(p)) of SORTK. Subroutine COUNT will be discussed in detail later.

Next the rank values are summed by columns. For each $i \in \mathbb{Z}_M$,

$$\text{rank}[i] := \sum \{ \text{rank}[j] : i \equiv j \pmod{N} \}.$$

By now all of the non-null C-values have been moved to the diagonal blocks. The C-registers of the non-diagonal blocks have been set to null.

Finally, the C-values are moved horizontally from the diagonal blocks to their proper columns using a variation of the routine above. Record C[i] is moved to column rank[i] of its $2^{(s+1)m}$ -block.

Phase s is now completed. In each column of every $2^{(s+1)m}$ -column-block is a non-null C-register. Together these form a sorted list of $2^{(s+1)m}$ records.

Start of phase 1, with $n=3$ $m=1$, and $k=1$

0	1	2	3	4	5	6	7
6			5	1			4
8	7	2			3	0	

Shaded processors are diagonal.

Each 2-column-block contains a sorted list of C-values.

A typical processor is shown below.

i	S
C	T

Copy C-values vertically over S-values

6	7	2	5	1	3	0	4
6			5	1			4
6	7	2	5	1	3	0	4
	7	2			3	0	

Copy S-values over T-values horizontally from diagonal blocks

6	7	2	5	1	3	0	4
6	6	2	2	1	1	0	0
6	7	2	5	1	3	0	4
7	7	5	5	3	3	4	4

Execute COUNT and move elements to their proper columns

2		6		0	1		
	5		7			3	4

Ready to start next phase.

```

Procedure CUBE_SORT (C,n,m)
cobegin  $0 \leq i < 2^{m+n}$ 
    r := 0; q := m;  << r=sm and q=(s+1)m always >>
    for s := 0 to (n/m)-1 do
        begin << M a i n    L o o p >>
            << Merge sorted sublists of length  $2^r$  together >>
            << in groups of  $2^m$ . The processor index is i. >>
            S[i] := C[i]
            << Copy S vertically >>
            for b := n to n+m-1 do
                S[i(b)] := S[i] : (S[i] ≠ null)
            endfor
            << Prepare to copy S onto T horizontally >>
            T[i] := S[i]      : (DIAG[r,i] )
            T[i] := null      : (not DIAG[r,i] )
            C[i] := T[i]  << Now only diagonal processors >>
                        << have non-null C-registers. >>
            << Copy horizontally blockwise. >>
            for b := r to q-1 do
                T[i(b)] := T[i] : ( T[i] ≠ null )
            endfor

            call COUNT(r)

            << Compute column sums of rank. >>
            for b := n to n+m do
                rank[i(b)] := rank[i(b)] + rank[i]
            endfor
        end
    endfor

```

```

    << Move C-values to proper columns. >>
    for b := q downto 0 do
        if C[i] ≠ null and (rank[i])b ≠ ib
            then
                C[i(b)] := C[i]
                rank[i(b)] := rank[i]
            endif
        endfor

        r := q
        q := q + m
    end << Main Loop >>

    << Move the sorted list to the first row. >>
    for b := n to n+m-1 do
        C[i(b)] := C[i]      : ( C[i] ≠ null )
    endfor
coend CUBE_SORT.

```

The procedure COUNT(r) operates on each 2^r -block independently and in parallel. Each 2^r -block contains two sorted lists (of length 2^r), one in the S-registers, the other in the T-registers. COUNT(r) will find the rank of the elements in the S-list with respect to the T-list.

The 2^r -block containing P[i] is a right-block if

$$\lfloor i/2^n \rfloor < \lfloor i/2^r \rfloor \bmod 2^m,$$

and a left-block if

$$\lfloor i/2^n \rfloor > \lfloor i/2^r \rfloor \bmod 2^m.$$

In a 2^{r+m} -column-block the right-blocks appear above and to the right of the diagonal blocks. Conversely, the left-blocks appear below and to the left of the diagonal blocks. A 2^r -block must be either a left-block, a right-block, or a diagonal block.

COUNT[r] will perform the following on the 2^r -block containing P[i].

- 1) If the 2^r -block is a right-block,

$$\text{rank}[i] := |\{ T[j] : T[j] < S[i] \}|.$$
- 2) If the 2^r -block is diagonal,

$$\text{rank}[i] := i \bmod 2^r.$$
- 3) If the 2^r -block is a left-block,

$$\text{rank}[i] := |\{ T[j] : T[j] \leq S[i] \}|.$$

The ranking is performed by merging the S and T list together using an adaptation of Batcher's bitonic merge [BAT]. Given two sorted lists $L[0], \dots, L[2^r-1]$ and $L[2^r], \dots, L[2^{r+1}-1]$, the first list in ascending order and the second in descending order, Batcher's bitonic merge is as follows.

```

for t := r downto 0 do
  for each i,  $0 \leq i < 2^{r+1}$ ,  $i_t = 0$  do
    if  $L[i] > L[i^{(t)}]$ 
      then
         $L[i] := L[i^{(t)}]$ ;
      endfor
    endfor
  endfor
endfor

```

COUNT uses a clever implementation of the bitonic merge so that $L[i]$ and $L[i^{(t)}]$ are in the same processor when they are to be compared. Only 2^r processors will be needed.

To reduce notational clutter we take the indices modulo 2^r (or 2^{r+1} for L) in the discussion to follow. This is possible since COUNT considers each 2^r -block independently.

COUNT must first reverse the T-sequence so that it is in descending order. To do this, we must move $T[i]$, $0 \leq i < 2^r$, to $T[2^r - i]$. But $2^r - 1 - i = \text{NOT}(i)$, that is, the number obtained by changing each bit of i . The T-sequence is reversed by executing

```

cobegin  $0 \leq i < 2^r$ 
  for b := 0 to r-1 do
     $T[i^{(b)}] := T[i]$ 
  endfor
coend.

```

Note the assignment statement $T[i^{(b)}] := T[i]$ is in fact an exchange since while processor $P[i]$ is sending $T[i]$ to processor $P[i^{(b)}]$, $P[i^{(b)}]$ is sending $T[i^{(b)}]$ to $P[i^{(b)(b)}] = P[i]$.

In the notation above, $L[i]$ corresponds to $S[i]$ if $0 \leq i < 2^r$. If $2^r \leq i < 2^{r+1}$, then $L[i]$ corresponds to $T[i - 2^r] = T[i^{(r)}]$. In the first iteration of the bitonic merge loop, we must compare $L[i]$ and $L[i^{(r)}]$ for each i , $0 \leq i < 2^{r+1}$, $i_r = 0$. Since $i_r = 0$ implies $i < 2^r$, we will compare $S[i]$ with $T[i]$. Both $S[i]$ and $T[i]$ reside in the same processor.

In the second iteration of the bitonic merge we must compare $L[i]$ and $L[i^{(r-1)}]$ for each i , $0 \leq i < 2^{r+1}$, $i_{r-1} = 0$. If $0 \leq i < 2^r$ and $i_{r-1} = 0$ then

$$i = (0.0.i_{r-2} \dots i_0),$$

and

$$i^{(r-1)} = (0.1.i_{r-2} \dots i_0) < 2^r.$$

So $L[i]$ corresponds to $S[i]$ and $L[i^{(r-1)}]$ corresponds to $S[i^{(r-1)}]$. Conversely, if $2^r \leq i < 2^{r+1}$ and $i_{r-1} = 0$, then

$$i = (1.0.i_{r-2} \dots i_0) < 2^r + 2^{r-1},$$

and

$$i^{(r-1)} = (1.1.i_{r-2} \dots i_0) \geq 2^r + 2^{r-1}.$$

Therefore $L[i]$ corresponds to $T[i]$ and $L[i^{(r-1)}]$ corresponds to $T[i^{(r-1)}]$. Pictorially we have:

S	0		i		2^{r-1}		$i^{(r-1)}$	
L	0		i		2^{r-1}		$i^{(r-1)}$	

T	0		i		2^{r-1}		$i^{(r-1)}$	
L	2^r		$i^{(r)}$		2^r		$i^{(r-1)(r)}$	

We wish to compare $S[i]$ with $S[i^{(r-1)}]$ and $T[i]$ with $T[i^{(r-1)}]$, $0 \leq i < 2^r$. There are two 2^{r-1} -blocks of S-values and two 2^{r-1} -blocks of T-values. In order to facilitate the comparisons we will switch the second S-block with the first T-block. That is, for $0 \leq i < 2^r$,

$$T[i^{(r-1)}] := S[i] \quad (i_{r-1} = 1).$$

Now instead of comparing $T[i]$ with $T[i^{(r-1)}]$, $2^{r-1} \leq i < 2^r$, $i_{r-1} = 1$, we compare $T[i]$ with $S[i]$. Similarly, instead of comparing $S[i]$ with $S[i^{(r-1)}]$, $0 \leq i < 2^r$, $i_{r-1} = 0$, we compare $S[i]$ with $T[i^{(r-1)(r-1)}] = T[i]$.

Thus all comparisons are made within the processors without additional interprocessor data transfers.

The bitonic merge now continues in the same manner, considering each 2^{r-1} -block independently.

After $r+1$ parallel comparison steps the merge is complete. The final sorted list is

$$S[0], T[0], S[1], T[1], \dots, S[2^r-1], T[2^r-1].$$

While performing the merge, the variables $\text{Origin_S}[i]$ and $\text{Origin_T}[i]$, $0 \leq i < 2^r$, will accompany $S[i]$ and $T[i]$ respectively whenever a data transfer occurs. This is so we can compare the position of the records before and after merging. Initially, we set

$\text{Origin_S}[i] := i$, and

$\text{Origin_T}[i] := \begin{cases} -1, & \text{if } \text{RIGHT}[r, i] \\ 2^r, & \text{if } \text{LEFT}[r, i]. \end{cases}$

(Differentiating between right and left blocks is necessary so that the sort will be stable. When comparing $S[i]$ and $T[i]$ we may find these records to have equal values. The switch will then take place if and only if $\text{Origin_S}[i] > \text{Origin_T}[i]$.)

Record $S[i]$ is the $(2i)^{\text{th}}$ largest record in the merged list while $T[i]$ is the $(2i+1)^{\text{th}}$ largest record, $0 \leq i < 2^r$. (The smallest record, $S[0]$, is considered to be the 0^{th} largest record.) If $\text{Origin_S}[i]$ is not -1 or 2^r then $S[i]$ came originally from the S-list. Initially it was $S[\text{Origin_S}[i]]$. Therefore the number of T-records which come before the original $S[i]$ in the merged list is

$\text{rank}[\text{Origin_S}[i]] := 2i - \text{Origin_S}[i]$.

Similarly if $\text{Origin_T}[i]$ is not -1 or 2^r then

$\text{rank}[\text{Origin_T}[i]] := 2i+1 - \text{Origin_T}[i]$...

However, $\text{Origin_S}[i]$ and $\text{Origin_T}[i]$, $0 \leq i < 2^r$, are located in processor $P[i]$, not in processor $P[\text{Origin_S}[i]]$ or $P[\text{Origin_T}[i]]$ respectively. COUNT's job is to compute $\text{rank}[i]$ and store it in processor $P[i]$. Therefore we will overwrite S and T with the final positions:

$$S[i] := \begin{cases} 2i, & \text{if } \text{Origin_S}[i] \neq -1 \text{ or } 2^r \\ \text{null}, & \text{otherwise,} \end{cases}$$

and

$$T[i] := \begin{cases} 2i+1, & \text{if } \text{Origin_T}[i] \neq -1 \text{ or } 2^r \\ \text{null}, & \text{otherwise,} \end{cases}$$

Now $S[i]$ will be moved to $S[\text{Origin_S}[i]]$ (if $S[i] \neq \text{null}$) and $T[i]$ will be moved to $S[\text{Origin_T}[i]]$ (if $T[i] \neq \text{null}$), $0 \leq i < 2^r$. This is done by reversing the bitonic merge.

Procedure COUNT[r]

<< Determine the ranking for each 2^r -block. >>

Cobegin $0 \leq i < 2^{m+n}$

<< Reverse the T-sequence >>

for $b := 0$ to $r-1$ do

$T[i^{(b)}] := T[i]$

endfor

<< Save original positions of the S-lists >>

$\text{Origin_S}[i] := i \bmod 2^r$

<< The following is needed to handle equal elements >>

$\text{Origin_T}[i] := 2^r$; (LEFT[r,i])

$\text{Origin_T}[i] := -1$; (RIGHT[r,i])

```
<< BITONIC MERGE >>
```

```
b := r-1
```

```
loop
```

```
  if (S[i]>T[i]) or ( S[i]=T[i] and Origin_S[i]>Origin_T[i] )
```

```
  then << Interchange within the processor >>
```

```
    S[i] := T[i]
```

```
    Origin_S[i] := Origin_T[i]
```

```
  endif
```

```
  if b<0 then exit loop endif
```

```
  << Move list elements to prepare for next iteration >>
```

```
  T[i(b)] := S[i] : (ib=1)
```

```
  Origin_T[i(b)] := Origin_S[i] : (ib=1)
```

```
  b := b-1
```

```
endloop
```

```
<< Overwrite S and T with the corresponding final positions >>
```

```
S[i] := 2(i mod 2r)
```

```
T[i] := 2(i mod 2r) + 1
```

```
<< Destroy the final positions corresponding >>
```

```
<< to elements originally from the T-sequences >>
```

```
S[i] := null : ( Origin_S[i] = -1 or 2r )
```

```
T[i] := null : ( Origin_T[i] = -1 or 2r )
```

```

<< Route final positions back to their origin >>
<< by reversing the bitonic merge. >>
for b := 0 to r-1 do
  if ( S[i] ≠ null and (Origin_S[i])b = 1)
    or ( T[i] ≠ null and (Origin_T[i])b = 0)
  then
    S[i] := T[i]
    Origin_S[i] := Origin_T[i]
  endif
  T[i](b) := S[i] : (ib = 1)
  Origin_T[i](b) := Origin_S[i] : (ib = 1)
end

(S[i], Origin_S[i]) := (T[i], Origin_T[i]) : (T[i] ≠ null)

<< Compute the rank >>
rank[i] := S[i] - Origin_S[i]
rank[i] := i mod 2r : (DIAG[r, i])
coend COUNT.

```

The main loops of COUNT will be iterated r times for each call. Each iteration requires a constant amount of time. Thus COUNT requires $O(r) \leq O(n) = O(\log N)$ time for each call.

The complete algorithm CUBE_SORT requires $O(n^2/m) = O(k \log N)$ time.

SECTION 1.3: SORTING ON A MESH-CONNECTED COMPUTER

Compared with the shared memory and cube-connected computers the mesh-connected computer is not nearly as efficient with respect to the sorting problem. To sort a list of N records our algorithm (due to Thompson and Kung [THK]) will require N processors and $O(N^{\frac{1}{2}})$ time. We are willing to forgive the MCC for its lack of speed because it is a relatively easy machine to build compared to the SMC and CCC. Indeed several such systems already exist (ILLIAC IV, MPP). It is easy to imagine a very large scale MCC being constructed using present-day technology. The same cannot be said of the SMC or CCC.

The algorithm to be used follows the outline of SORT_LOGN of section 1.1. The sublists will be merged using a bitonic merge. Essentially this is an adaptation of Batcher's bitonic merge algorithm [BAT].

Assume N is an even power of two. Let $n=N^{\frac{1}{2}}$ and let $k=\log n$. Both n and k are integers.

Initially $C[i]$ is in $P[i]$ for each $i \in \mathbb{Z}_N$.

Let $B[r,j]$ be the 2^r -block of list elements

$$\langle C[j2^r+0], C[j2^r+1], \dots, C[j2^r+2^r-1] \rangle,$$

for $0 \leq r \leq 2k$, and $0 \leq j < 2^{2k-r}$. The block $B[0,j]$,

$0 \leq j < 2^{2k-0} = N$, is the single record list $\langle C[j] \rangle$. Block

$B[2k,0]$ is the complete list of records, C .

There are $2k$ phases. At the beginning of the r^{th} phase, $0 \leq r < 2k$, each $B[r, j]$ will be sorted in ascending order if j is even and descending order if j is odd. This is because the bitonic merge requires two lists which are ordered in opposite directions.

A simplified outline of the algorithm is as follows.

Procedure MESH_SORT

Begin

for $r := 0$ to $2k-1$ do

for each even j , $0 \leq j < 2^{2k-r}$ << $2^{2k-r} = N/2^r$ >> do

Bitonically merge $B[r, j]$ and $B[r, j+1]$

in ascending order if $j/2$ is even and

in descending order if $j/2$ is odd.

<< $B[r+1, j/2]$ is now sorted >>

endfor

endfor

end MESH_SORT.

The bitonic merge of $B[r, j]$ and $B[r, j+1]$; $0 \leq r < 2k$, $0 \leq j < 2^{2k-r}$, j even; will require that for $b := 0, 1, \dots, r-1$ we move record $C[i]$ from processor $P[i]$ to processor $P[i^{(b)}]$. After comparing $C[i]$ with $C[i^{(b)}]$ the new value for $C[i]$ must be returned to $P[i]$. The speed of the algorithm will be determined by the amount of time required for these data transfers. The "distance" between $P[i]$ and $P[i^{(b)}]$ is of prime importance. This distance is highly dependent on the scheme used for indexing the processors.

First we will consider row-major indexing. Let $d(b)$ be the number of data routing steps required for the MCC to move $C[i]$ from $P[i]$ to $P[i^{(b)}]$ for each i , $0 \leq i < N$, such that $i_b = 0$, $0 \leq b < 2k$.

If $0 \leq b < k$ and $i_b = 0$ then $P[i]$ and $P[i^{(b)}]$ are in the same row but $P[i^{(b)}]$ is 2^b columns to the right of $P[i]$. Thus by executing 2^b "shift-right" instructions the data transfer will be complete. In this case $d(b) = 2^b$.

If $k \leq b < 2k$ then $P[i]$ and $P[i^{(b)}]$ are in the same column but $P[i^{(b)}]$ is 2^{b-k} rows below $P[i]$. So 2^{b-k} shift-down instructions will complete the data movement, therefore $d(b) = 2^{b-k}$.

In conclusion, $d(b) = 2^{b \bmod k}$ for $0 \leq b < 2k$.

Let $M(r)$ be the number of data routing steps required to merge two 2^r -blocks. Then

$$M(r) = \sum_{b=0}^r 2d(b).$$

Let $S(k)$ be the number of data routing steps to sort a 2^k by 2^k array. We have

$$\begin{aligned} S(k) &= \sum_{r=0}^{2k-1} M(r) \\ &= \sum_{r=0}^{2k-1} \sum_{b=0}^r 2d(b) \\ &= 2 \sum_{b=0}^{2k-1} d(b) \sum_{r=b}^{2k-1} 1 \\ &= 2 \sum_{b=0}^{2k-1} 2^{b \bmod k} (2k-b) \\ &= 2 \sum_{c=0}^{k-1} 2^c \{ (2k-c) + (2k-(c+k)) \} \end{aligned}$$

$$\begin{aligned}
&= 2 \sum_{c=0}^{k-1} 2^c (3k-2c) \\
&= 6k \left(\sum_{c=0}^{k-1} 2^c \right) - 4 \left(\sum_{c=0}^{k-1} c 2^c \right) \\
&= 6k(2^k - 1) - 4 * 2 \{ (k-2) 2^{k-1} + 1 \} \\
&= 2k 2^k + 8 * 2^k - 6k - 8 \\
&= O(k 2^k) = O(n \log n) = O(N \frac{1}{2} \log N).
\end{aligned}$$

Next we consider shuffled row-major indexing.

For each $i \in \mathbb{Z}_N$, processor $P[i]$ is in row

$$(i_{2k-2} \cdot i_{2k-4} \cdot \dots \cdot i_4 \cdot i_2 \cdot i_0)_2$$

and column

$$(i_{2k-1} \cdot i_{2k-3} \cdot \dots \cdot i_5 \cdot i_3 \cdot i_1)_2.$$

If b is even, $0 \leq b < 2k$, then $P[i^{(b)}]$ is in the same column as $P[i]$. Assume $i_b = 0$. Then the row index of $P[i^{(b)}]$ is

$$(i_{2k-2} \cdot \dots \cdot i_{b+2} \cdot 1 \cdot i_{b-2} \cdot \dots \cdot i_2 \cdot i_0)_2.$$

Thus $P[i^{(b)}]$ is $2^{b/2}$ rows below $P[i]$.

If b is odd then $P[i^{(b)}]$ is in the same row as $P[i]$.

Again assume $i_b = 0$. Then the column index of $P[i^{(b)}]$ is

$$(i_{2k-1} \cdot \dots \cdot i_{b+2} \cdot 1 \cdot i_{b-2} \cdot \dots \cdot i_3 \cdot i_1)_2.$$

Thus $P[i^{(b)}]$ is $2^{(b-1)/2}$ columns to the right of $P[i]$.

Let d_s and S_s be the counterparts of d and S respectively. Then

$$d_s(b) = 2^{\lfloor b/2 \rfloor}, \quad 0 \leq b < 2k$$

Figure: Model of a Mesh-connected Computer

The shuffled row-major indexing is shown in boldface (decimal and binary) and the row-major indexing is shown in standard typeface. N=64.

0 000	1 000	4 000	5 000	16 010	17 010	20 010	21 010
000	001	100	101	000	001	100	101
0 000	1 000	2 000	3 000	4 000	5 000	6 000	7 000
000	001	010	011	100	101	110	111
2 000	3 000	6 000	7 000	18 010	19 010	22 010	23 010
010	011	110	111	010	011	110	111
8 001	9 001	10 001	11 001	12 001	13 001	14 001	15 001
000	001	010	011	100	101	110	111
8 001	9 001	12 001	13 001	24 011	25 011	28 011	29 011
000	001	100	101	000	001	100	101
16 010	17 010	18 010	19 010	20 010	21 010	22 010	23 010
000	001	010	011	100	101	110	111
10 001	11 001	14 001	15 001	26 011	27 011	30 011	31 011
010	011	110	111	010	011	110	111
24 011	25 011	26 011	27 011	28 011	29 011	30 011	31 011
000	001	010	011	100	101	110	111
32 100	33 100	36 100	37 100	48 110	49 110	52 110	53 110
000	001	100	101	000	001	100	101
32 100	33 100	34 100	35 100	36 100	37 100	38 100	39 100
000	001	010	011	100	101	110	111
34 100	35 100	38 100	39 100	50 110	51 110	54 110	55 110
010	011	110	111	010	011	110	111
40 101	41 101	42 101	43 101	44 101	45 101	46 101	47 101
000	001	010	011	100	101	110	111
40 101	41 101	44 101	45 101	56 111	57 111	60 111	61 111
000	0001	100	101	000	001	100	101
48 110	49 110	50 110	51 110	52 110	53 110	54 110	55 110
000	001	010	011	100	101	110	111
42 101	43 101	46 101	47 101	58 111	59 111	62 111	63 111
010	011	110	111	010	011	110	111
56 111	57 111	58 111	59 111	60 111	61 111	62 111	63 111
000	001	010	011	100	101	110	111

As before,

$$\begin{aligned}
 S_g(k) &= \sum_{r=0}^{2k-1} \sum_{b=0}^r 2d_g(b) \\
 &= 2 \sum_{r=0}^{2k-1} (2k-b) * 2 * 2^{\lfloor b/2 \rfloor} \\
 &= 2 \sum_{r=0}^{2k-1} 2^c \{ (2k-2c) + [2k-(2c+1)] \} \\
 &= 2 \sum_{r=0}^{2k-1} 2^c \{ (4k-1) - 4c \} \\
 &= \{ 2(4k-1) \sum_{c=0}^{k-1} 2^c \} - 16 \sum_{c=0}^{k-1} c 2^{c-1} \\
 &= (8k-2)(2^k-1) - 16 \{ (k-2)2^{k-1} + 1 \} \\
 &= 8k2^k - 8k - 2*2^k - 14 - 8k2^k + 16*0^k \\
 &= 14*2^k - 8k - 14 \\
 &= 14n - 8\log n - 14 \\
 &= O(N^{\frac{1}{2}}).
 \end{aligned}$$

We conclude that the shuffled row-major indexing yields a more efficient sorting algorithm than the row-major indexing.

In the discussion above we have not considered the time required by the processors to perform tasks such as comparisons and updating local variables. The reader can verify that these tasks require a number of steps which are at most proportional to the number of data routing steps.

CHAPTER 2: CONNECTED COMPONENT ALGORITHMS FOR A
SHARED-MEMORY COMPUTER

SECTION 2.1

Like sorting, computing the connected components of a graph is a very fundamental computer science problem. In this section an algorithm due to Hirschberg, et al. [HCS], will be given which computes the connected components of a N -vertex graph on a SMC using NQ processors with time complexity $O(\log^2 N)$ where

$$Q = \lceil N/\log N \rceil$$

This chapter deals exclusively with algorithms for shared memory computers.

Hirschberg's algorithm uses an adjacency matrix to represent the graph. In section 2.2 other graph representations will be used. These lead to more efficient algorithms.

Let $G=(V,E)$ be an undirected graph with $V=\{0,1,\dots,N-1\}$. Let $A_{N \times N}$ be the adjacency matrix of G : for all $i,j \in V$,

$$A[i,j] = \begin{cases} 1, & \text{if } (i,j) \in E, \\ 0, & \text{otherwise.} \end{cases}$$

The input to the algorithm CONNECT will be the matrix A.

The output will be the N-vector Comp, where

$\text{Comp}[i] = \min\{j \in V : j \text{ is in the same component as } i\},$
 $i \in V$. Then $\text{Comp}[i] = \text{Comp}[j]$ if and only if i and j are in
 the same component, $i, j \in V$.

In each component there will be a unique vertex, i say,
 such that $\text{Comp}[i] = i$, $i \in V$. This i will be called the
 root of its component.

The algorithm CONNECT operates by collapsing vertices
 which are found to be in the same component into a
 pseudocomponent. Each pseudocomponent has a root which is
 its vertex of minimum index. The pseudocomponent will be
 identified by its root.

Initially each vertex forms its own pseudocomponent.
 At each general step each pseudocomponent searches for
 another pseudocomponent to which it is connected. These are
 then merged. The merging may involve many pseudocomponents.
 There are $\log N$ general steps.

Procedure CONNECT

Begin

```

1. For all i ∈ V do Comp[i] := i;
2. for logN iterations do
    for all i ∈ V do
        begin
            << find edges from one pseudocomponent to another >>
3.      C[i] := min{Comp[j]: (i,j) ∈ E and Comp[i] ≠ Comp[j]}
            if none then Comp[i];
            << each pseudocomponent finds the pseudocomponent >>
            << of minimal index to which it is joined >>
4.      C[i] := min{C[j]: Comp[j]=i and C[j] ≠ i }
            if none then Comp[i]
            << save C >>
5.      D[i] := C[i]
            << trace the path formed by the trail of C-values >>
6.      for logN iterations do
          C[i] := C[C[i]]
        endfor
            << merge pseudocomponents along each path of >>
            << C-values into the one of minimal index >>
7.      Comp[i] := min{ C[i], D[C[i]] }
        end
    end
end CONNECT.

```

Some observations. Consider the q^{th} iteration, $1 \leq q \leq \log N$, of the main loop. After line 4 is executed we will have: for any $i \in V$ $C[i] = \text{Comp}[i]$ if and only if

- (1) i is not the root of a pseudocomponent

or

- (2) i is the root of a pseudocomponent which is not

connected to any other pseudocomponent (which means this pseudocomponent is a component).

Otherwise, $C[i] \neq \text{Comp}[i]$ and i is the root of a pseudocomponent, $\text{Comp}[i]$, which is connected to pseudocomponent $C[i]$. Thus $C[i]$ does not satisfy (1) or (2) so $C[C[i]] \neq C[i]$. Therefore the C -values induce a path between pseudocomponents: $(i, C[i], C[C[i]], \dots)$. Since there are finitely many vertices and $C[j] \neq j$ for any j on the path we must have the path ending in a non-empty simple cycle.

Let $Z = (i_0, i_1, \dots, i_k = i_0)$, $2 \leq k \leq N$, be this cycle, $C[i_j] = i_{j+1}$ for $j = 0, 1, \dots, k-1$. Without loss of generality assume i_0 is the pseudocomponent of minimum index. By assumption there is an edge connecting pseudocomponents i_0 and i_1 as well as another edge between i_1 and i_2 .

But because of the way C is constructed (line 4) we must have $i_2 \leq i_0$. But i_0 is the minimum index. Therefore $i_2 = i_0$ and $k = 2$. Thus the C -path ends in a cycle of length 2.

The loop of line 6 will trace the C-paths to their respective 2-cycles. This 2-cycle will be $(C[i], D[C[i]])$ for any vertex i contained in a pseudocomponent on the C-path.

Line 7 selects the pseudocomponent of minimum index on the 2-cycle. All other connected pseudocomponents will be merged into this one.

Proposition 1 After q iterations of the main loop of CONNECT at least one of the following hold for each $i \in V$:

- (i) $\text{Comp}^{-1}[i] = \emptyset$
- (ii) $|\text{Comp}^{-1}[i]| \geq 2^q$
- (iii) $\text{Comp}^{-1}[i]$ is the vertex set of a component of G .

Corollary After $\log N$ iterations $\text{Comp}^{-1}[i] = \emptyset$ or $\text{Comp}^{-1}[i]$ is a component, $i \in V$.

Proposition 1 is easily verified by induction. The corollary shows the validity of CONNECT.

It is clear that using N^2 processors CONNECT would have a time complexity of $O(\log^2 N)$. We shall show that the number of processors may be reduced to N^q without affecting the time complexity.

The best sequential algorithms employ a simple depth first search and run in time $O(|E|+N)$. [RND]

Assume there are NQ processors. For each $i \in V$ we must be able to execute each line of the main loop in time $O(\log N)$ with Q processors. Lines 5, 6, and 7 present no problems.

Consider line 3. To each $i \in V$ we assign Q processors. Let

$$C_i[j] := \begin{cases} \text{Comp}[j], & \text{if } A[i,j]=1 \text{ and } \text{Comp}[i] \neq \text{Comp}[j] \\ \infty, & \text{otherwise,} \end{cases}$$

$i, j \in V$. Using Q processors, the vector C_i may be calculated in time $O(N)/Q = O(\log N)$. Now to find $C[i]$ we find the minimum element of the N -vector C_i . There are two steps:

- (1) Divide C_i into Q groups of $\log N$ elements. Each processor will now find the minimum element of a single group. This can be done in $O(\log N)$ time.
- (2) There is now a vector of length Q . The minimum can be found in $O(\log Q) \leq O(\log N)$ time.

Thus for a given vertex line 3 may be executed in $O(\log N)$ with Q processors. Line 4 may be executed in a similar manner with the same time-processor complexity. In conclusion, only NQ processors are required to execute CONNECT in time $O(\log^2 N)$.

Chin, Lam, and Chen [CLC] give an improved algorithm requiring $N \lceil N / \log^2 N \rceil$ processors which runs in time $O(\log^2 N)$.

SECTION 2.2

Hirschberg's implementation of CONNECT can only be considered efficient if the graph is very dense. This is typical of algorithms which use an adjacency matrix to represent the graph. Savage and Ja'Ja' [SAJ] show that using an adjacency list brings the processor needs down to $O_p(\max\{2M, N\})$, where M is the cardinality of E , and yet the time complexity remains $O(\log^2 N)$.

The adjacency list will be the $2M$ -vector Adj . For each node $i \in V$, $first[i]$ will be a pointer to the beginning of the entries related to i . The set of nodes adjacent to i will be

$$\{ Adj[s] : first[i] \leq s < first[i+1] \}.$$

Define $first[N] := 2M+1$.

Associated with each entry $Adj[s]$ of the adjacency list will be $list_id[s]$ to indicate which node's adjacency list contains $Adj[s]$. So $list_id[s] = i$ if and only if $first[i] \leq s < first[i+1]$.

This data structure requires $4M+N+1$ memory locations. The data structure used by Savage and Ja'Ja' requires N^2 memory locations. Our data structure will give the same processor-time complexity as that of Savage and Ja'Ja'.

As before lines 5, 6, and 7 are easily done in $O(\log N)$ time with $O_p(N)$ processors. The problem is to implement lines 3 and 4.

Assume we have $K = \max(2M, N)$ processors available.

First we consider line 3. For each $i \in V$ we will calculate

$$C''[i] := \min\{\text{Comp}[j] : (i, j) \in E, \text{Comp}[j] \neq \text{Comp}[i]\} \\ = \min\{\text{Comp}[\text{Adj}[s]] : \text{list_id}[s] = i, \text{Comp}[\text{Adj}[s]] \neq \text{Comp}[i]\}.$$

(C'' is the same as C on line 3 except the "if none" clause is omitted.) Let the processors be indexed as $P[0], P[1], \dots, P[K-1]$. To calculate C'' we assign $P[s]$ to $\text{Adj}[s]$, $0 \leq s < 2M$. To calculate $C''[i]$ the processors $P[\text{first}[i]]$ to $P[\text{first}[i+1]-1]$ inclusive will be used. Each processor $P[s]$ will calculate

$X[s] = \text{Comp}[\text{Adj}[s]], \text{ if } \text{Comp}[\text{Adj}[s]] \neq \text{Comp}[\text{list_id}[s]]$
 ∞ , otherwise.

X can be computed in constant time.

Now

$C''[i] = \min\{X[s] : \text{first}[i] \leq s < \text{first}[i+1]\}$

for $i \in V$. C'' can now be found in $O(\log N)$ time since $\text{first}[i+1] - \text{first}[i] \leq N$.

Line 4 requires a different approach. We must compute

$C^*[i] = \min\{C[j] : \text{Comp}[j] = i, C[j] \neq i\}$

for each $i \in V$. Let S be a $N \times 2$ array. Define the first column to be

$S[j,1] := \text{Comp}[j], j \in V$.

The second column will be

$S[j,2] := C[j], \text{ if } \text{Comp}[j] \neq C[j]$

∞ , otherwise,

$j \in V$. Now sort the rows of S using the first column as

the primary key and the second column as the secondary key.

There are many sorting algorithms available. Which one is

used will depend on the density of the graph and the

availability of storage space in the shared memory. This

will be discussed later. To find $C^*[i]$ we find the first

occurrence of i in the first column. The corresponding entry

in the second column will be $C^*[i]$. (If there is no

occurrence of i in the first column or if $C^*[i] = \infty$ then

the "if none" clause of line 4 applies.) Using N processors we perform this calculation as follows: each processor $P[j]$, $0 \leq j < N$, will execute

```

    if  $S[j-1,1] \neq S[j,1]$  or  $j=0$ 
    then
         $C[S[j,1]] := S[j,2]$ 
    endif.

```

This can be done in constant time.

Excluding the work required for sorting S , CONNECT now requires $O_p(\max\{2M, N\})$ processors to run in time $O(\log^2 N)$ using $O(M)$ space.

In each of the $\log N$ iterations the N rows of S must be sorted using both a primary and a secondary key. First we will apply a sorting algorithm using the primary key and then we will sort again using the secondary key. The algorithm used for the second sort must be stable.

All of the entries of S are elements from Z_N . Hirschberg's bucket-sort algorithm [HIR] will sort N records from Z_N using N processors and $O(\log N)$ time but requires the use of a N by N boolean array which has been initialized by putting every entry to 0. Before terminating, the algorithm will restore the boolean array to its initial state. Using this algorithm, CONNECT will require $O_p(\max\{2M, N\})$ processors, $O(\log^2 N)$ time, and

$O(N^2)$ space. To reduce storage requirements, we may use other sorting algorithms, but at the cost of increasing the processor-time complexity. Preparata's sorting algorithm [PRE] requires $O_p(N \log N)$ processors and $O(\log N)$ time while SORT_K of section 1.1 requires $O_p(N^{1+1/k})$ processors and $O(k \log N)$ time, $0 < k \leq \log N$. Both of these algorithms require $O(N)$ space but are less efficient because they do not take advantage of the fact that the records are from Z_N . Below is a table summarizing the processor, time, and space complexities of CONNECT when different sorting algorithms are used.

Sorting Algorithm	Processor Complexity	Time Complexity	Space Complexity
Hirschberg's Bucket Sort	$\max\{2M, N\}$	$\log^2 N$	N^2
SORT_K	$\max\{2M, N^{1+1/k}\}$	$k \log^2 N$	M
SORT_LOGN	$\max\{2M, N\}$	$\log^3 N$	M
Preparata	$\max\{2M, N \log N\}$	$\log^2 N$	M

CHAPTER 3: MINIMUM SPANNING TREE ALGORITHMS FOR

SHARED MEMORY COMPUTERS

SECTION 3.1

Let $G=(V,E)$ be an undirected connected graph with $V=\{0,1,\dots,N-1\}$ and $M:=|E|$. Each edge $e=(i,j)\in E$ will have a real-valued weight $w(i,j)=w(e)$. If $G'=(V',E')$ is a subgraph of G then define the weight of G' to be

$$w(G') = w(E') := \sum \{w(e) : e \in E'\}.$$

A spanning tree T of G is a minimum spanning tree (MST) if its weight, $w(T)$, is minimal.

In this section we present an algorithm due to Savage and Ja'Ja' [SAJ] which finds the MST of a graph using $O_p(N^2)$ processors and $O(\log^2 N)$ time on a SMC. In section 3.2 we present a new algorithm which uses only $O_p(M)$ processors and $O(\log^3 N / \log(2M/N))$ time.

Assume the weights are distinct. The framework of the algorithm (due to Sollin [SAJ]) is as follows.

```
begin  T := (V, ∅)
      while T is not connected do
        for each component K of T do
          find the edge  $e \in E$  of minimal weight which has
            exactly one end in K.
          T := T ∪ {e}
        endfor;
      end
```

Each iteration of the main loop will result in each component being merged with at least one other. Initially each component is a single vertex. Therefore the main loop will run at most $\log N$ times.

An iteration of the main loop starts with each vertex $i \in V$ finding the edge $(i, C[i]) \in E$, $C[i] \in V$, of minimal weight which leaves the component containing i . Now each component, K , finds the edge of minimum weight which has only one end in K . The edge is added to the tree and the components are merged accordingly. The merges are performed by the mechanisms used in Hirschberg's CONNECT (section 2.1).

A proof of the validity of the algorithm is deferred until chapter 4.

Procedure MST

```

1. for all  $i \in V$  do  $\text{Comp}[i] := i$  endfor
2.  $T := (V, \emptyset)$ 
3. while  $T$  is not connected do
4.   for each  $i \in V$  do
5.     find a vertex  $C[i]$  such that  $\text{Comp}[C[i]] \neq \text{Comp}[i]$ ,
        $(i, C[i]) \in E$ , and  $w(i, C[i])$  is minimal.
       if none then  $C[i] := \text{nil}$ 
6.   if  $i$  is the root of a component (i.e.  $\text{Comp}[i] = i$ )
7.   then
8.     find  $j \in V$  such that  $\text{Comp}[j] = i$  and
        $w(C[j], j)$  is minimal
       if none then  $G$  is not connected stop
9.    $T := T \cup \{(j, C[j])\}$ 
10.   $C[i] := D[i] := \text{Comp}[C[j]]$ 
11.  else
12.     $C[i] := D[i] := \text{Comp}[i]$ ;
13.  for  $\log N$  iterations do
14.     $C[i] := C[C[i]]$ ;
15.     $\text{Comp}[i] := \min\{C[i], D[C[i]]\}$ 
16.  endfor;
end MST.
```

Consider the q^{th} iteration of the main "while" loop of MST. Assume that T is not connected when this iteration begins. For each vertex $i \in V$ let $K_i := \text{Comp}^{-1}[i]$. The components of T are those K_i such that $K_i \neq \emptyset$.

Define the graph T_q whose vertices are the components of T and whose edges are the new edges added during the q^{th} iteration, $q=1, 2, \dots$. More precisely,

$$V(T_q) = \{K_i : K_i \neq \emptyset, i \in V\}$$

and

$$E(T_q) = \{(K_i, K_{C[i]}) : K_i \in V(T_q)\}$$

where C above is evaluated at the end of line 10.

Let C° be the value of C at the end of line 5.

Let $K_i \in V(T_q)$. The root of K_i is i . We claim that lines 5 to 9 select the edge of minimal weight leaving K_i . This edge is $e_i = (j, C^\circ[j])$, where j is defined in line 8. It corresponds to $(K_i, K_{C[i]})$ of T_q . There is exactly one edge of T_q which leaves K_i .

Let $i' = C[i] (=Comp[j])$. Let $e_{i'}$ be the edge of minimal weight leaving $K_{i'}$. There are two cases: (1) $e_i \neq e_{i'}$, and (2) $e_i = e_{i'}$. In the first case we must have $w(e_i) > w(e_{i'})$. We conclude T_q contains no cycles of length greater than two. The situation is as in CONNECT: the path of new edges ends in a 2-cycle.

Each component of T_q will be merged as in CONNECT. The minimum weight edge of each component (of T_q) will be in the 2-cycle. If K_u and K_v are the vertices (of T_q) on the 2-cycle then the entire component will assume the index $\min\{u, v\}$.

The complexity analysis of MST is as with CONNECT. $O_p(N^2)$ processors execute MST in time $O(\log^2 N)$ using the naive approach of Savage and Ja'Ja'. Chin, Lam, and Chen [CLC] show that only $O_p(N \lceil N / \log^2 N \rceil)$ processors are necessary.

SECTION 3.2

Next we present a new algorithm for the MST problem which uses $O_p(\max(2M, N^{1+1/k}))$ processors and has time complexity $O(k \log^2 N)$ where k is an arbitrary fixed integer, $k > 1$. The algorithm relies heavily on the sorting algorithms SORT_LOGN and SORT_K of chapter 1.

Assume $M > N$ (if $M < N$ then either G is already a MST or G is not connected). In order to minimize processor and time complexity we might set $2M = N^{1+1/k}$ and solve for k .

This gives

$$k_0 = \frac{\log_2 N}{\log_2(2M/N)}.$$

($\log_2 x$ means no rounding up). Since k must be an integer, k_0 will have to be rounded up or down.

Since

$$2N \leq 2M \leq 2(N(N-1)/2)$$

then

$$2 \leq 2M/N \leq N-1$$

so

$$1 \leq \log_2(2M/N) \leq \log_2(N-1) < \log_2 N \leq \log N.$$

Dividing by $\log_2 N$ gives

$$\frac{1}{\log_2 N} \leq \frac{\log_2(2M/N)}{\log_2 N} = \frac{1}{k_0} < 1.$$

Taking reciprocals we find

$$1 < k_0 \leq \log_2 N.$$

In summary, with $O_p(2M)$ processors we obtain an algorithm with time complexity

$$O_p\left(\frac{(\log_2 N)^3}{\log_2(2M/N)}\right).$$

If the graph is sparse, $M=O(N)$ say, then only $O(k \log^2 N) = O(\log^3 N)$ time and $O(N)$ processors are required.

If the graph is dense, $M=(N-1)N/2$ for example, then $O_p(N(N-1)) \leq O_p(N^2)$ processors and $O(\log^2 N)$ time is required.

Thus our algorithm is more efficient than other algorithms when the graph is sparse. Otherwise, it is comparable to that of Savage and Ja'Ja'. The algorithms have the same complexity if $M=O(N^2)$. The best sequential algorithms have complexity $O(N^2)$ [PRI] or $O(M \log \log N)$ [YAO].

The algorithm, EMST, will use a linked adjacency list Adj together with the pointer array First to represent the graph. (The adjacency list of section 2.2 was not linked.) Adj is a sequence of lists. Initially, Adj[First[i]], Adj[first[i]+1], ..., Adj[first[i+1]-1] is a linked list of the vertices adjacent to vertex i, $i \in V$. The adjacency list for each vertex will be sorted according to edge-weight. If an edge is found to have both of its ends

```

8.  cobegin e ∈ E'
9.    Deleted[e] := false;
10.   Succ[e] := e+1
11.   if e+1 = First[Invald[e]+1]
       then << e is at the end of its adjacency list >>
           Succ[e] := nil
12. coend
    << Assume Succ[nil]=nil. >>
    << end of INITIALIZATION >>

    << M A I N   L O O P >>
13. while not all vertices are in the same component do begin
    << delete edges within components >>
    << At this time, C[i] is the component of i, i ∈ V. >>
14.  cobegin e ∈ E'
15.    if C[Adj[e]] = C[Invald[e]]
        then Deleted[e] := true
16.  coend
    << remove deleted edges from Adjlist >>
17.  for logN iterations do begin
18.    cobegin e ∈ E'
19.      if Deleted[Adj[Succ[e]]]
          then Succ[e] := Succ[Succ[e]]
20.    coend
21.    cobegin i ∈ V << special case for first element in list >>
22.      if Deleted[First[i]]
          then First[i] := Succ[First[i]]
23.    coend
24.  endfor
    << for i ∈ V the minimum weight inter-component edge >>
    << is (i, Adj[First[i]]) unless the adjacency list is empty >>

```

Data Structures

A typical element of the linked adjacency list:

Vertex First

Invadj = Inverse of Adj. Indicates which vertex's adjacency list we are in.

Deleted = $\begin{cases} 0, & \text{if the edge is not deleted} \\ 1, & \text{if the edge is deleted.} \end{cases}$

Succ = Pointer to the next element of the linked list.

Suppose each vertex has a local variable x ($=x[i]$). We present an algorithm, FIND_MAX, to find

$$\text{Max}_x := \max\{x[i] : i \in V\}.$$

Initially every node is sleeping. This is indicated by the local variable "state" (that is, state=sleeping). The algorithm begins with the spontaneous awakening of the designated vertex "root". Only the root will be allowed to spontaneously awaken. The root will proceed to waken the other nodes "manually". Each node has a local variable index which contains the index of that node. (index[i]=i, $i \in V$.)

The algorithm below is the one which would be given to each node.

```

8.  cobegin e ∈ E'
9.    Deleted[e] := false;
10.   Succ[e] := e+1
11.   if e+1 = First[Invadj[e]+1]
      then << e is at the end of its adjacency list >>
          Succ[e] := nil
12. coend

    << Assume Succ[nil]=nil. >>
    << end of INITIALIZATION >>

    << M A I N   L O O P >>
13. while not all vertices are in the same component do begin
      << delete edges within components >>
      << At this time, C[i] is the component of i, i ∈ V. >>
14.  cobegin e ∈ E'
15.    if C[Adj[e]] = C[Invadj[e]]
      then Deleted[e] := true
16.  coend
      << remove deleted edges from Adjlist >>
17.  for logN iterations do begin
18.    cobegin e ∈ E'
19.      if Deleted[Adj[Succ[e]]]
      then Succ[e] := Succ[Succ[e]]
20.    coend
21.    cobegin i ∈ V << special case for first element in list >>
22.      if Deleted[First[i]]
      then First[i] := Succ[First[i]]
23.    coend
24.  endfor
      << for i ∈ V the minimum weight inter-component edge >>
      << is (i, Adj[First[i]]) unless the adjacency list is empty >>

```



```

    << M A I N    L O O P (continued) >>
    << find the mininum weight edge out of each component >>
25.  cobegin i@V
    .  X[i] := i
      coend
26.  sort X using primary key = C[X]
      secondary key= Weight[First[X]]
      << assume Weight[nil]=∞ >>
27.  cobegin i@V
28.    if C[X[i]]≠C[X[i-1]] and First[X[i]]≠nil
29.    then << X[i] is the first vertex from >>
        << component C[X[i]] in X >>
30.    add the edge (X[i],Adj[First[X[i]]]) to T
        << this is the minimum weight edge leaving >>
        << component C[X[i]] >>
31.    D[X[i]] := C[Adj[X[i]]]
32.    else
        D[X[i]] := C[X[i]]
33.    endif

34.    C[i] := D[i] << temporary storage >>
35.    for logN iterations do D[i] := D[D[i]] endfor
36.    C[i] := min{ D[i], C[D[i]] }
37.  coend
38. endwhile
end EMST.

```

The initialization phase (lines 1 to 12) may be done with $O_p(2M)$ processors and $O(\log^2 N)$ time using SORT_LOGN of section 1.1 to execute line 7.

As before, the main loop will be executed at most $\log N$ times. We check that not all vertices are in the same component by computing the boolean expression

$$\min\{C[i]: i \in V\} = \max\{C[i]: i \in V\}$$

with $O_p(N)$ processors and $O(\log N)$ time. The implementation of lines 14 to 25 is straightforward; $O_p(2M)$ processors and $O(\log N)$ time is needed. On line 26 a list of length N is sorted. This may be done using $O_p(N^{1+1/k})$ processors and $O(k \log N)$ time using `SORT_K` (section 1.1). The rest of the main loop (lines 27 to 37) is an adaptation of `CONNECT`. It requires $O_p(N)$ processors and $O(\log N)$ time.

Therefore the algorithm requires $O_p(\max\{2M, N^{1+1/k}\})$ processors and $O(k \log^2 N)$ time.

The storage requirements of EMST are $O(M + N^{1+1/k})$. If $k=k_0$, then the storage requirements are $O(M)$. Savage and Ja'Ja's MST algorithm uses $O(N^2)$ space.

CHAPTER 4: AN ALGORITHM FOR THE MINIMUM WEIGHT SPANNING

TREE OF A DISTRIBUTED COMPUTATION NETWORK

SECTION 4.1: INTRODUCTION

In the previous chapters we studied multiprocessor designs which had a high level of symmetry: the processors were identical, arranged in regular patterns, and were synchronous. In contrast a distributed computation network (DCN) has little symmetry - the processors need not be identical or synchronous and any fixed interprocessor connection design is possible. The distributed computation network is the most general multiprocessor design in that it incorporates the others as special cases.

Let $G=(V,E)$ be a connected graph with N vertices. At each vertex $i \in V$ there is the processor $P[i]$. $P[i]$ is identified with i . For $i,j \in V$, $P[i]$ is connected to $P[j]$ by a direct interprocessor communications link if and only if $(i,j) \in E$. At the beginning of a DCN algorithm, all the processors are "sleeping". The algorithm begins when one or several processors spontaneously "wake-up". The awakened processors will send "wake-up" messages to adjacent processors.

Dijkstra and Scholten [DJS] have determined a number of conditions which will guarantee termination after a finite amount of time. These rules provide a guide for DCN work, however, our algorithms will not always adhere to them due to efficiency considerations.

In this section we will give a simple, but instructive, algorithm to find the vertex containing the highest value of the variable "X". In section 4.2 we will give an $O(N \log N)$ time algorithm for finding the minimum spanning tree (of G). The MST algorithm is mainly due to Gallager, Humblet, and Spira [GHS]. We have added some improvements which reduce the time requirements by a constant factor.

Each processor will perform the same algorithm, but not synchronously. The algorithm consists of a series of responses to received messages. The response may include the sending of additional messages.

A typical scenario in a DCN is as follows: A node, i , receives a message, M , on an edge, e . In response, node i sends the message M' out on each of its incident edges except e . Node i then waits for a response to each of the messages M' . When all responses have been received node i returns a response to M along e .

Suppose each vertex has a local variable $x (=x[i])$. We present an algorithm, FIND_MAX, to find

$$\text{Max}_x := \max\{x[i] : i \in V\}.$$

Initially every node is sleeping. This is indicated by the local variable "state" (that is, state=sleeping). The algorithm begins with the spontaneous awakening of the designated vertex "root". Only the root will be allowed to spontaneously awaken. The root will proceed to waken the other nodes "manually". Each node has a local variable index which contains the index of that node. (index[i]=i, $i \in V$.)

The algorithm below is the one which would be given to each node.

Procedure FIND_MAX begin

```
<< The answer (max(x[i]:iθV)) will be the value of >>
<< max_x[root] when root has entered state "finished". >>
<< All variables are local >>
```

In response to spontaneous awakening do

begin

if index=root

then execute subroutine Initialize;

for all incident edges e do

send START on e

count := count + 1

endfor

else << this is not the root. Go back to sleep >> endif

end

subroutine Initialize

begin

state := awake

count := 0

max_x := x << max_x is the best x-value so far >>

end

In response to START on edge f do

```

begin
  if state=sleeping
  then
    execute subroutine Initialize
    inedge := f
    for all incident edges e except f do
      send START on e
      count := count + 1
    endfor
    if count=0 then execute subroutine RETURN_MESSAGE;
  else << node has already received a START message >>
    send ALREADY_AWAKE on f
  endif
end

```

In response to ALREADY_AWAKE on edge e do

```

begin
  count := count - 1
  if count=0 then execute subroutine RETURN_MESSAGE endif
end

```

subroutine RETURN_MESSAGE

```

begin << count is zero >>
  if index≠root
  then send BEST_X[max_x] on inedge endif
  state := finished
end

```

```

In response to BEST_X[y] on edge e do
  begin
    if y > max_x then max_x := y endif
    if count = 0 then execute subroutine RETURN_MESSAGE endif
  end
end FIND_MAX.

```

The action at a typical node i (root is as follows:

Initially asleep, i will receive a START message on an edge it will call $inedge$. In turn i will send START messages on all other incident edges. The local variable count will be equal to

$$\begin{aligned}
 & (\# \text{ of START messages sent}) - \\
 & (\# \text{ of ALREADY_AWAKE and BEST_X messages received}).
 \end{aligned}$$

Once i is awake, it will return an ALREADY_AWAKE message for each START message received. Node i may receive BEST_X[y] messages on the edges it sent START messages. If $y > \max_x$ then $\max_x$ will be updated ($\max_x := y$). Eventually i will have received a response (either ALREADY_AWAKE or BEST_X) for each START message it sent. At this point count will be zero. Now the message BEST_X[max_x] will be sent on $inedge$.

The algorithm may be considered to be a parallel breadth-first search:

The principles involved in FIND_MAX will be used repeatedly in the MST algorithm.

SECTION 4.2

Some theory must be developed before the algorithm may be given.

Let $G=(V,E)$ be a simple undirected connected graph. Each edge $e \in E$ has a real weight $w(e)$.

Define a fragment to be a non-empty subtree of a MST. An outedge of a subgraph is an edge with exactly one end in the subgraph. The minimum-weight outedge will simply be called the minimum outedge.

Definition Let the tree T be a subgraph of G . Let $e=(u,v) \in E$ be an edge not in T with $u,v \in V(T)$. Let P be the (unique) simple path in T from v to u . Then define the fundamental cycle of e with respect to T to be $P \cup \{e\}$

Proposition 1

[GHS] Let F be a fragment and e be a minimum outedge of F . Then $F \cup \{e\}$ is also a fragment.

Proof

Let T be a MST which contains F as a subtree. Suppose T does not contain e . Let C be the fundamental cycle of e

with respect to T . Since C contains vertices in F and not in F , it must contain an outedge, f say, of F different from e .

Since e is a minimum outedge of F , then $w[e] \leq w[f]$.

Let

$$T' := (T \setminus \{f\}) \cup \{e\}.$$

Now, T' is a spanning tree of G , but

$$w[T'] = w[T] - w[f] + w[e] \leq w[T].$$

By the minimality of $w[T]$ we must have $w[T'] = w[T]$, so T' is also a MST. Since $F \cup \{e\}$ is a subtree of T' then $F \cup \{e\}$ is a fragment. $\square$.

Proposition 1 shows the validity of the MST algorithms of chapter 3.

Proposition 2

If the edge weights are all distinct then the MST is unique.

Proof

Assume the edge weights are distinct.

Suppose there are two distinct MST's, T and T' . Let e be the edge of minimum weight which is in one MST but not in

22
OF / DE

the other. Without loss of generality assume $e \in T$. Let C be the fundamental cycle of e with respect to T' . Since T has no cycles there must be an edge, f say, contained in circuit C but not in T , furthermore, $f \notin e$.

Then $w[f] > w[e]$ since $w[e]$ is minimal for edges in the symmetric difference of $E(T)$ and $E(T')$.

Let $T'' := (T' \setminus \{f\}) \cup \{e\}$. Then T'' is a spanning tree and

$$w[T''] = w[T'] - w[f] + w[e] < w[T'].$$

This is a contradiction since T' is a MST. Hence the MST is unique. $\square$.

Henceforth we will assume that the edge-weights are distinct.

Proposition 3

Let F and F' be disjoint fragments and e be an edge with one end in F , the other in F' . If e is a minimum outedge of F or F' then

$$F'' := F \cup F' \cup \{e\}$$

is a fragment.

Proof

By proposition 1, $F'' := F \cup \{e\}$ is a fragment. The edge-weights are distinct so the MST is unique. Thus F'' and F' are fragments of the same MST and so is their union F'' . $\square$.

The algorithm will operate by performing repeated applications of proposition 3. In the above notation fragments F and F' are said to **merge** to form F'' . The new edge e is called the **core** of F'' .

The simplest fragment is a single vertex. The algorithm will start by considering each vertex as a fragment. The general strategy will be for the vertices in a fragment to cooperate in order to find the minimum outedge and then attempt to merge with the fragment at the other end of this outedge.

We cannot allow merging to proceed indiscriminately. Each time a merger takes place the entire fragment must be informed. In some fragments this could require $O(N)$ time (we have not yet rigorously defined "time" in an asynchronous multiprocessor). Furthermore if a fragment

merges successively with "small" fragments then many ($O(N)$ time) merges may be required. For example, in the following graph if vertex 1 were to merge with vertex 2 then to merge

with vertex 3 and so on, then $O(N^2)$ time would be required.

The rule to be used is that fragments which are "larger" in a certain sense will have to wait for the "smaller" ones. The index of size will be the level of the fragment. The single-vertex fragment will be level 0. Inductively, if $t \geq 0$ then a level t fragment can raise its level by two methods:

- M1. By merging with another level t fragment to form a level $t+1$ fragment.
- M2. By merging with a level t' fragment where $t' > t$. The new fragment will be level t' .

A simple induction argument prove the next proposition.

Proposition 4

Let F be a level t fragment, $t \leq 0$.

- 1. F has at least $\min(N, 2^t)$ vertices.
- 2. If $t > 0$ then F was formed by first merging two level $t-1$ fragments and then possibly merging with some

other fragments whose level is less than t . (That is, an application of M1 followed by zero or more applications of M2.)

3. $t \leq \log N$. $\square$

From (2) above we see that any fragment F with level $t > 0$ has a unique edge which was used as the core of a merge of two $t-1$ fragments. This edge will be called the initial core of F . The algorithm will use the weight of the initial core as the identity of the fragment. The identity of a level 0 fragment will not be required by the algorithm.

figure: A typical level t fragment.

Let F and F' be the identities of disjoint fragments of level t and t' respectively. Let e be the minimum outedge of F and suppose e has one end in F' . Three cases arise:

- (1) If $t < t'$ then F and F' will merge immediately. The new fragment will have the same identity as F' and level t' .
- (2) If $t = t'$ and e is also the minimum outedge of F' then F and F' will merge immediately. The new fragment will have identity $w[e]$ and level $t+1$.
- (3) Otherwise, the merge will not take place immediately. F will have to wait. Eventually the level of F' will be raised enough so that (1) or (2) hold. (This will be proven later.)

We now describe how the nodes cooperate to efficiently implement the above concepts.

Before the algorithm begins each node will be sleeping. The algorithm begins when one or more nodes spontaneously awakens. Depending on the structure of the graph it may be advantageous to manually awaken all the nodes before the algorithm begins. For simplicity we normally will assume all nodes are spontaneously awakened at the same time.

Consider a typical phase of the algorithm. Let F be the identity of a level t fragment, $t > 0$, with initial core $e = (x, y)$ ($w[e] = F$), $x, y \in V$.

(2) If $t=t'$ then the merger will take place only if g is also the minimum outedge of F' . If this is so then s will return an $\text{INITIATE}[t+1, g, \text{find}]$ message on g .

In general, when a node receives an $\text{INITIATE}[\text{new_level}, \text{new_frag_id}, \text{new_state}]$ message on an edge j it will respond by assuming the new level, fragment identity and state given by the message. The variable "inbranch" will be set to j . The node will then send duplicate messages on all incident branches except j . Thus the tree is redirected.

As mentioned earlier it will be possible for a node to send REPORT messages before it has found its minimum outedge or exhausted its supply of basic edges. This corresponds to case (iii) of page 98 and will be called pruning the tree. Without pruning, it is possible during the TEST subphase to wait a considerable amount of time for a response on an edge which ultimately does not get chosen as the minimum outedge of the fragment. During the TEST subphase a node receiving an ACCEPT message which causes best_weight to change will send the message $\text{BEST_WT_SO_FAR}[\text{best_weight}]$ on all incident branches. Each node will maintain a variable "pruning_threshold" equal to the minimum argument of all BEST_WT_SO_FAR messages received. When a node receives a BEST_WT_SO_FAR message, on branch j say, which causes

(5) **CONNECT-INITIATE:** Perform the merge and send INITIATE messages throughout the fragment. This will raise the level and initiate the TEST sub-phase again.

A node will respond to TEST messages at any time (that is subphase (2)). The other subphases are executed in linear order. The nodes of a fragment are not synchronized with respect to the phase they are executing.

The TEST subphase will start for v , $v \in V(F)$, when it receives an INITIATE $[t, w[e], \text{find}]$ message on its inbranch. The node v will relay this message on its outbranches. Edges which are not branches and have not been deleted will be called basic edges. Each node will maintain a heap [AHU] of the edge-weights of incident basic edges. The incident edge of minimum weight will be at the top of the heap. The message TEST $[t, w[e]]$ will be sent out on every edge in the heap. Let f be one of the edges on v 's heap and let u be the vertex at the other end of f . There are three cases.

- (1) If u is in the same fragment as v then u labels f as "rejected". Node u will also send a TEST message of its own across f to v . Node v will recognize

that u is in the same fragment so v will also label f as rejected. Rejected edges are deleted from the heap

- (2) If u is not in the same fragment as v and $\text{level}(u) \geq t$ then u returns an ACCEPT[fragment identity of u] message over f .
- (3) Otherwise, we have $\text{level}(u) < t$. No response from u is given. The message is placed at the end of a queue and will be considered again later when (hopefully) $\text{level}(u) \geq t$.

The reason for waiting in (3) above is that a merger would be forbidden at this time if f were selected as the minimum outedge of F .

Eventually one of three events will occur:

- (i) Vertex v receives an ACCEPT message from the edge at the top of v 's heap.
- (ii) All edges incident to v are rejected (except for branches).
- (iii) Vertex v receives outside information (to be discussed later) that none of its outedges could become the minimum outedge of the fragment.

The node v will now enter the REPORT subphase.

Each node, v , will maintain a variable best_wt which will contain the weight of the best candidate for minimum

outedge which v has knowledge of. This knowledge will come in the form of a message on an edge, j say. When this occurs the local variable `best_edge` will be set to j . This is so that a path may be traced to the edge having weight `best_wt`.

Suppose v is a leaf of the tree and one of (i), (ii), or (iii) occurs. Node v will enter the REPORT subphase and send the message `REPORT[best_wt]` on its inbranch.

Suppose v is an interior node. Then v will enter the REPORT subphase, but it will wait until it has received REPORT messages on all its outbranches before sending its own REPORT message. Node v will update `best_wt` based on incoming REPORT messages.

The REPORT subphase ends when the nodes at the ends of the initial core edge exchange REPORT messages. This is the beginning of the CHANGE_ROOT subphase. Now the weight of the minimum outedge, g say, of the entire fragment is known as well as which side of the initial core g lies on. The trail of `best_edges` tells how to find g . The algorithm will trace this path until it reaches the node, r say, which is incident to g but is still in fragment F .

The CONNECT subphase will now start with r sending a $\text{CONNECT}[t]$ message to the node, s say, at the other end of g . Let F' be the identity of the fragment containing s and t' be its level. From the TEST subphase we know $t \leq t'$. There are two cases.

(1) If $t < t'$ then s will return the message

$\text{INITIATE}[t', F', \text{state}[s]]$ where $\text{state}[s]$ is either "find" or "found". If $\text{state}[s]$ is "find" then s is in the TEST subphase. Since F is now being merged with F' then F should proceed to the TEST subphase. If $\text{state}[s]$ is "found" then s has finished the TEST subphase. The result of this TEST subphase would have been to find the minimum outedge of F' . Since g was one of the edges considered in this search there could be no lower-weighted edge emanating from F . Therefore F will be considered to have completed the TEST and REPORT subphases.

(2) If $t=t'$ then the merger will take place only if g is also the minimum outedge of F' . If this is so then s will return an $INITIATE[t+1,g,find]$ message on g .

In general, when a node receives an $INITIATE[new_level, new_frag_id, new_state]$ message on an edge j it will respond by assuming the new level, fragment identity and state given by the message. The variable "inbranch" will be set to j . The node will then send duplicate messages on all incident branches except j . Thus the tree is redirected.

As mentioned earlier it will be possible for a node to send REPORT messages before it has found its minimum outedge or exhausted its supply of basic edges. This corresponds to case (iii) of page 98 and will be called pruning the tree. Without pruning, it is possible during the TEST subphase to wait a considerable amount of time for a response on an edge which ultimately does not get chosen as the minimum outedge of the fragment. During the TEST subphase a node receiving an ACCEPT message which causes $best_weight$ to change will send the message $BEST_WT_SO_FAR[best_weight]$ on all incident branches. Each node will maintain a variable "pruning_threshold" equal to the minimum argument of all $BEST_WT_SO_FAR$ messages received. When a node receives a $BEST_WT_SO_FAR$ message, on branch j say, which causes

pruning_threshold to change it will send a BEST_WT_SO_FAR[pruning_threshold] message on all incident branches except j. If a node finds that it is waiting for a response from an edge whose weight is greater than the pruning_threshold then the node will abort the TEST subphase. This node will be considered to have been "pruned".

We must show that deadlock does not occur, that is, we must show that we cannot have all fragments simultaneously waiting for responses so that processing is suspended throughout the graph. There are two sets of conditions under which a node $v \in V$ in fragment F will be waiting for responses from other fragments:

- (1) All of the basic edges incident to v have their other end in a fragment with a level lower than the level of F . In addition v must have at least one incident basic edge. No responses to v 's TEST messages will be given.
- (2) The minimum outedge, f say, of the fragment F is incident to v and v has sent a CONNECT[level] message across f . Let F' be the fragment at the other end of f and suppose that F' is of the same level as F . If either

(a) F' has not yet determined its minimum outedge
or (b) the minimum outedge of F' is not f
then v will have to wait to receive an INITIATE
message.

Suppose the current lowest level of a fragment is t . Let F
be a level t fragment whose minimum outedge is minimal.
Condition (1) cannot hold since no fragment has a level
which is less than t . Condition (2b) cannot hold since the
minimum outedge of F' would have to be f . If condition (2a)
holds then fragment F' is active (since F' is looking for
its minimum outedge) so trivially we do not have a deadlock.
Thus deadlock does not occur.

A more formal description of the algorithm may now be
given.

Procedure DIST_MST**Variables**

<< The names of local variables are completely in lower >>
 << case letters. The names of subroutines have their >>
 << first letter capitalized. Messages are completely in >>
 << upper case letters. >>

<< Each node contains the following local variables. >>

<< notation: $I :=$ set of incident edges >>

state : {basic, branch, rejected}

edge_type[I] : {basic, branch, rejected}

<< It is possible for edge_type to have >>

<< different values at the two ends of an edge. >>

best_edge, level, best_weight, pruning_threshold,

in_branch, find_count, w[I], accepted[I],

fragment_id, min_test_edge

pruned : Boolean

<< Received messages are placed at the end of a queue>>

<< The message at the front queue will be processed >>

<< first. Each node will have a heap of the >>

<< edge-weights of incident basic edges. min_test_edge >>

<< will contain the index of of the edge whose weight is >>

<< at the top of the heap. Edges are assumed to be >>

<< automatically deleted from the heap if they are no >>

<< longer basic. >>

Initialize

```

begin
  state := sleeping
  for each edge  $j \in I$  do
    edge_type[j] := basic
  endfor
end

```

In response to spontaneous awakening do

```

begin execute Wake-up end

```

subroutine Wake-up

```

begin << merge with node(s) at other end of min-test-edge >>
  form the heap
  edge_type[min_test_edge] := branch
  level := 0
  state := found
  find_count := 0
  send CONNECT[0] on min_test_edge
end

```

```

In response to CONNECT[ t ] on edge j do
begin
  if state=sleeping then execute Wake-up endif
  if level>t
  then edge_type[j] := branch
    send INITIATE[ level, fragment_id, state ] on j
    if state=find then find_count := find_count + 1;
  else if edge_type[j]=basic
    then place message CONNECT[t] at end of queue
    else send INITIATE[ level+1, w[j], find ] on j;
  endif
end
end

```

```

In response to INITIATE[ t, F, S ] on j do
begin
  level:=t; fragment_id:=F; state:=S;
  inbranch:=j; best_weight:=∞; best_edge:=nil;
  pruned:=false; pruning_threshold:=∞;
  for all incident branches g, g≠j do
    begin
      send INITIATE[ t, F, S ] on g
      if state=find then find_count:=find_count+1
    end
    if state=find then execute Test endif
  end
end
end

```

subroutine Test

begin

if the heap is empty

then execute Report

else

for all edges g on the heap do

begin

accepted[g] := false

send TEST[level, fragment_id] on g

end

end

In response to TEST[t, F] on edge j do

begin

if state=sleeping then execute Wake-up endif

if level < t

then place message TEST[t,F] at end of queue

else

if F ≠ fragment_id

then

send ACCEPT[F] on j

else

edge_type[j] := rejected

execute Pruning_check

execute Report

endif

endif

end

In response to ACCEPT[F] on j do

begin

if F#fragment_id and state=find

then

accepted[j] := true

if w[j] < best_wt

then

best_wt := w[j]; best_edge := j;

send BEST_WT_SO_FAR[w[j]] on all incident branches

endif

if j=min_test_edge then execute Report endif

endif

end

Subroutine Report

begin

if (accepted[min_test_edge] or pruned or heap is empty)

and find_count=0 and state=find

then

state := found

send REPORT[best_wt] on inbranch

endif

end

```

In response to BEST_WT_SO_FAR [ w ] on j do
  begin
    if not accept[min_test_edge] and not pruned )
      and w < pruning_threshold
    then
      pruning_threshold := w
      send BEST_WT_SO_FAR[w] on all incident branches except j
      | execute Pruning_check
    endif
  end
end

```

Subroutine Pruning_check

```

begin
  if not( pruned) and not( accept[min_test_edge] )
    and w[min_test_edge]>pruning_threshold
  then
    pruned := true
    execute Report
  endif
end

```

In response to REPORT[W] on j do

begin

if j#inbranch

then << we are not at the initial core >>

find_count := find_count - 1

if W<best_wt then best_wt := W; best_edge := j endif

execute Report

else << we are at the initial core >>

if state=find

then place message at end of queue

else if W>best_wt

then << we are at the correct side of initial core >>

execute Change_root

else if W=best_wt=∞

then send TERMINATE on all outbranches

halt, we're done

endif

endif

endif

endif

end

In response to TERMINATE do

begin

send TERMINATE on all outbranches

halt

end

```

subroutine Change_root
  if edge_type[best_edge]=branch
  then << continue along path >>
    send CHANGE_ROOT on best_edge
  else << we are at the minimum outedge >>
    send CONNECT[ level ] on best_edge
    edge_type[ best_edge ] := branch
  endif
end

In response to CHANGE_ROOT do
  begin
    execute Change_root
  end

end DIST_MST.

```

Our algorithm differs from that of Gallager, Humblet, and Spira [GHS] in the following ways:

- (1) [GHS] do not utilize a heap to maintain the edge-weights so that the minimum may always be found in $O(\log N)$ time.
- (2) [GHS] send only one TEST message at a time. This will always be sent on the minimum-weight incident basic edge. If this edge is rejected then a TEST message is sent on the next best basic edge. This can be time consuming in worst-case conditions. Our algorithm sends all of its TEST messages at once.

(3) [GHS] do not prune the tree. Fragments can wait for long periods for a response from an edge which is too "heavy" to ever become the minimum outedge.

We now consider the time complexity of the algorithm. When analysing algorithms for DCN's the common practice is to assume that the processors send messages synchronously and messages take a unit time interval, t_c , to travel from one node to another. The unit time interval for one processor computation will be t_p . We will show that the time required for the algorithm is $O(N \log N)(t_c + t_p)$. The algorithm is said to have communication complexity $O(N \log N)$ and processor complexity $O(N \log N)$.

Proposition 5

DIST_MST has communication complexity $O(N \log N)$ when all nodes are awakened simultaneously.

Proof

We proceed by induction. The induction hypothesis will be that after

$$T(t) := 3tN - 2N + 2t - 2^{t+2} + 6$$

communication time units each node has achieved at least level t , $1 \leq t \leq \log N$, or there is only one fragment.

Consider $t=1$. $T(1)=3N-2N+2-8+6=N$. In the first communication step each node sends a CONNECT[0] message on its minimum outedge. The node now waits for an INITIATE

message to be returned on this edge. The INITIATE message will raise the level of the node from 0.

For each $v \in V$ let $s(v)$ be the node at the other end of the minimum outedge of v . As above, during the first communication step each v sends a CONNECT[0] message to $s(v)$. Now $s(v)$ will also be a level 0 fragment but it will return an INITIATE message immediately if and only if $s(s(v))=v$, that is, if v and $s(v)$ both have the same minimum outedge. Otherwise v must wait until the level of $s(v)$ is raised.

Define the digraph $H=(V,S)$ where

$$S := \{ (v, s(v)) : v \in V \}.$$

A cycle of length 2 in H corresponds to the case where two nodes have the same minimum outedge. Suppose H had a cycle, C say, of length greater than 2. Let e be the edge of minimum weight on C . Let $e=(x,y)$. Then $s(x)=y$. But $s(y) \neq x$ implies $w(y, s(y)) < w(e)$, a contradiction. Thus H contains no cycles of length greater than two.

Thus for each node $v_1 \in V$ we can define the simple path

$$P(v_1) := (v_1, v_2, \dots, v_k), \quad 2 \leq k \leq N,$$

where $s(v_i) = v_{i+1}$, $1 \leq i < N$, and $s(v_k) = v_{k-1}$. After 2 communication steps v_k and v_{k-1} will have sent and received an INITIATE[1,*,*] message. If $k > 2$, v_{k-1} will now send an INITIATE message to v_{k-2} , making v_{k-2} a level 1 node after 3 steps. Inductively, v_1 will be level 1 after k steps. Since $k \leq N$ we conclude that all nodes have achieved level 1 after N communication steps.

A graph in which the $N-1$ edges of highest weight are arranged in ascending weight along a simple path would take the full N communication steps for every node to reach level 1.

Assume that $T_0 \leq T(t)$ communication time units have passed and all nodes have reached level t , $t \leq 1$. Let F be a level t fragment.

(TEST) After 2 steps all the nodes of F will have finished the TEST subphase (lowest level nodes never have to wait for responses from TEST messages sent).

(REPORT) If F is the only fragment in the graph then we are done. Otherwise there is at least one other level t (or higher) fragment. This fragment has at least 2^t vertices. Thus F has at most $N - 2^t$ vertices. So the two

sides of F (one on either side of the initial core) each have at most $N-2^{t-2^{t-1}}$ vertices.

The REPORT subphase requires tracing a path from a vertex to the vertex at the opposite side of the initial core. This requires at most $N-2^{t-2^{t-1}}$ communication steps.

(CHANGE_ROOT-CONNECT) Next we must trace a path from an end of the initial core to the opposite side of the minimum outedge of F . (The CHANGE-ROOT subphase and then the CONNECT message is sent.)

(INITIATE) Now F must wait for a response on its minimum outedge, e say. This response will be an INITIATE message. Let F' be the fragment at the other end of the minimum outedge. If the level of F' is greater than t the response will be immediate. Assume F' is of level t . F' must have also already sent its own CONNECT message on its minimum outedge, e' say. The situation is familiar. We have a path of level t fragments connected by a decreasing-weight

sequence of minimum outedges. This path must end in either (a) a 2-cycle at which the fragments have the same outedge or (b) a level t^* fragment with $t^* > t$. When F' receives an INITIATE message on e' it will relay that message to all of its nodes. When the node in F' at the end of e receives an INITIATE message it will send an INITIATE message across e into F . This INITIATE message will be relayed throughout F . The entire process can take at most $N - 2^t$ communication steps. Now all vertices are at least at level $t+1$

In review, the contributions above are:

(TEST)	2
(REPORT)	$N - 2^t - 2^{t-1}$
(CHANGE_ROOT-CONNECT)	$N - 2^t - 2^{t-1}$
(INITIATE)	$N - 2^t$

In total $3N - 4 \cdot 2^t + 2$ communication steps are required to raise the level of all the nodes to at least $t+1$.

Now

$$T(t) + 3N - 4 \cdot 2^t + 2$$

$$\begin{aligned}
&= 3tN - 2N + 2t - 2^{t+2} + 6 + 3N - 4 \cdot 2^t + 2 \\
&= 3(t+1)N - 2N + 2(t+1) - 2^{t+2} - 2^{t+2} + 6 \\
&= 3(t+1)N - 2N + 2(t+1) - 2^{(t+1)+2} + 6 \\
&= T(t+1)
\end{aligned}$$

as required.

Once the graph becomes a single fragment a level t fragment requires an additional 1 communication step for sending TEST messages and then $N-2^t$ steps to send REPORT messages to the initial core and finally $N-2^{t-1}$ steps to send TERMINATE messages to all nodes. This is a total of $N-2^t$ steps.

Since $t \leq \log N$, we conclude that at most

$$\begin{aligned}
&T(\log N) + N - 2^{\log N} \\
&= 3N \log N - N + 2 \log N - 5 \cdot 2^{\log N} + 6 \\
&\leq 3N \log N - N + 2 \log N - 5N + 6 \\
&\leq 3N \log N \leq O(N \log N)
\end{aligned}$$

communication steps are required by DIST_MST. $\square$.

Gallager, Humblet, and Spira give a communication step bound of $5N \log N$ for their algorithm when all the nodes are wakened simultaneously. However, the [GHS] algorithm can require $O(N^2)$ steps if the nodes are not wakened simultaneously. Our algorithm sends out all of its TEST messages at once so that any node can waken all the others

within $N-1$ communication steps. Thus our algorithm requires at most

$$\begin{aligned} & \{ 3N\log N - 6N + 2\log N + 6 \} + \{ N-1 \} \\ &= 3N\log N - 5N + 2\log N + 5 \\ &\leq 3N\log N \end{aligned}$$

communication steps for any pattern of node awakening.

Proposition 5

DIST_MST has a processor complexity of $O(N\log N)$.

Proof

Each node is incident to at most $N-1$ edges. The total time required to form the heap for a given node and then delete its contents is $O(N\log N)$.

Excluding the work done with the heap, the processor time is clearly proportional to the communication time, $O(N\log N)$.

Therefore the processor complexity is also $O(N\log N)$. $\square$

References

- [ASS] Abraham, E., Seaton, E. T., and Smith, S. D. **The Optical Computer.** Scientific American, February, 1983, Vol. 248, No. 2, pp 85-93.
- [AHU] Aho, A. V., Hopcroft, J. E., and Ullman, J. D. **The Design and Analysis of Computer Algorithms.** Addison-Wesley, Reading, Mass., 1974.
- [BAT] Batcher, K. E. **Sorting Networks and Their Applications.** Proc. AFIPS 1968 SJCC, Vol. 32, AFIPS Press, Arlington, Va, pp 307-314.
- [CHD] Chandy, K. M., and Misra, J. **Distributed Computation on Graphs: Shortest Path Algorithms.** Comm. of the A.C.M., Vol 25, No. 11, Nov. 1982, pp 833-837.
- [CHG] Chang, E. J. H. **Echo Algorithms: Depth Parallel Operations on General Graphs.** IEEE Trans. on Software Engineering, Vol. SE-8, No. 4, July 1982, pp 391-401.
- [CLC] Chin, F. Y., Lam, J., and Chen, I. N. **Efficient Parallel Algorithms For Some Graph Problems.** Comm. ACM, Vol. 25, September 1982, pp 659-665.
- [DJS] Dijkstra, E. W., Scholten, C. S. **Termination Detection For Diffusing Computations.** Information Processing Letters, Vol. 11, No. 1, August 1980, pp 1-4.
- [GHS] Gallager, R. G., Humblet, P. A., and Spira, P. M. **A Distributed Algorithm For Minimum-Weight Spanning Trees.** ACM Transactions on Programming Languages and Systems, Vol. 5, No. 1, January 1983, pp 66-77.
- [HCS] Hirschberg, D. S., Chandra, A. K., and Sarwate, D. V. **Computing Connected Components on Parallel Computers.** Communications of the ACM, Vol. 22, No. 8, August 1979, pp 461-464.
- [HIR] Hirschberg, D. S. **Fast Parallel Sorting Algorithms.** Communications of the ACM, Vol. 21, No. 8, August 1978, pp 657-661.

- [HLS] Haynes, L. S., Lau, R. L., Siewiorek, D. P., and Mizell, D. W. **A Survey of Highly Parallel Computing.** Computer, January 1982, pp 9-24.
- [KUC] Kuck, D. J. **A Survey of Parallel Machine Organization and Programming.** Computing Surveys, Vol. 9, No. 1, March 1977, pp 29-59.
- [NAS] Nassimi, D., and Sahni, S. **Parallel Permutation and Sorting Algorithms and a New Generalized Connection Network.** Journal of the A.C.M., Vol. 29, No. 3, July 1982, pp 642-667.
- [PRE] Preparata, F.P. **Parallelism In Sorting.** Presented at the International Conference on Parallel Processing, Bellair, Michigan, August, 1977.
- [PRI] Prim, R. C. **Shortest Connection Networks and Some Generalizations.** Bell Systems Technical Journal, Vol. 26, 1957, pp 1389-1401.
- [RND] Reingold, E. M., Nievergelt, J., and Deo, N. **Combinatorial Algorithms.** Prentice-Hall, New Jersey, 1977.
- [SAJ] Savage, C., and Ja'Ja', J. **Fast, Efficient Parallel Algorithms For Some Graph Problems.** SIAM Journal on Computing, Vol. 10, No. 4, November 1981, pp 682-691.
- [THK] Thompson, C. D., and Kung, H. T. **Sorting on a Mesh-Connected Parallel Computer.** Communications of the ACM, Vol. 20, No. 4, April, 1977, pp 263-271.
- [YAO] Yao, A. C. **An $O(|E| \log \log |V|)$ Algorithm For Finding Minimum Spanning Trees.** Inform. Proc. Let., Vol. 4, 1975, pp 21-23.

END

10-06-186

FIN