Computing 29, 31—49 (1982)

Computing
© by Springer-Verlag 1982

Parallel Algorithms for the Single Source Shortest Path Problem*

P. Mateti, Cleveland, Ohio, and **N. Deo,** Pullman, Washington

Received November 23, 1981

Abstract — Zusammenfassung

Parallel Algorithms for the Single Source Shortest Path Problem. We present several parallel algorithms for the problem of finding shortest paths from a specified vertex (called the source) to all others in a weighted directed graph. The algorithms are for machines ranging from array processors, multiple-instruction multiple-data stream machines to a special network of processors. These algorithms have been designed by "parallelizing" two classic sequential algorithms – one due to Dijkstra (1959), the other due to Moore (1957). Our interest is not only in obtaining speeded-up parallel versions of the algorithms but also in exploring the design principles, the commonality of correctness proofs of the different versions, and the subjective complexity of explaining and understanding these versions.

AMS Subject Classification: 68C05, 68C25, 05C38, 68B10, 68E10.

Key words: Graph algorithms, shortest paths, parallel algorithms, parallelization, networks of processors, MIMD machines.

Parallele Algorithmen für die kürzesten Wege von einer einzelnen Quelle. Wir geben parallele Algorithmen an, die in einem gerichteten, bewerteten Graphen die kürzesten Wege von einem Knoten (der Quelle) zu allen anderen Knoten liefern. Die Algorithmen sind für verschiedene Maschinentypen entworfen, die von Feldrechnern (Array-Prozessoren) über Vielfach-Befehle, Vielfach-Daten-Strom-Maschinen (MIMD) bis zu einem speziellen Netz von Prozessoren reichen. Die Algorithmen sind durch „Parallelisierung" von zwei klassischen sequentiellen Algorithmen entstanden – dem von Dijkstra (1959) und dem von Moore (1957). Unser Interesse besteht nicht nur in der Konstruktion schnell laufender paralleler Versionen. Wir untersuchen auch die Entwurfsprinzipien, die Gemeinsamkeiten der Korrektheitsbeweise in den verschiedenen Versionen und die subjektive Komplexität, die verschiedenen Fassungen zu verstehen und zu erklären.

1. Introduction

Recent changes in the hardware technology have made it attractive to use a number of processors in parallel that cooperate and coordinate their efforts so that the elapsed execution time is reduced significantly. However, the art of designing parallel algorithms is far behind that of sequential algorithms. In this context one might either attempt to design a parallel algorithm from scratch, or "parallelize" known sequential algorithms.

* This work was supported in part by US DOT contract DTRS-5681-C-00033 and by NSF grant MCS 78-25851.

In this paper we explore the latter approach and consider the problem of producing all shortest paths from a single source to all other vertices in a weighted directed graph. We present several parallel algorithms for a wide variety of parallel machines. These algorithms are interesting at least for the following reasons.

(1) None of the algorithms were designed from scratch. All are essentially "parallelizations" of two sequential algorithms. This raises such questions as: When can we say that two algorithms are the same, except that one is sequential and the other parallel? Is it possible that two sequential algorithms considered to be different heretofore yield same parallelized algorithms? What are the principles involved in this sequential to parallel program translation? We discuss some of these issues in this paper.

(2) The elegance of our parallel algorithms, or the lack of it, can be traced to the (parallel) architecture of the machines. This becomes more apparent when we look at a set of algorithms that are equivalent in a strong sense but are designed for a range of machines. For example, our experience in this case study indicates that MIMD machines with a fixed number of processors cause considerable overhead both in synchronizing the processors and in assigning tasks to them. Other architectures provide natural parallel algorithms but require a number of processors proportional to the size of the graph. It is not clear if it would ever become technologically and economically feasible to construct machines with so many processors, in spite of the extremely simple nature of the computations performed by each.

(3) The parallel algorithms are interesting in their own right because of their efficiency and the amount of parallelism achieved. Most of the algorithms are also elegant. Much of the elegance and efficiency can be attributed to the fact that parallel computations take place on disjoint locations of memory most of the time, only occasionally requiring communication and/or mutual exclusion and synchronization.

We concentrate in this paper on presenting a variety of parallel algorithms. As a result, we do not devote enough space to correctness proofs of the algorithms. The need for formal correctness proof is undoubtedly greater in the case of parallel algorithms because of the possibility of subtle errors, starvation and deadlocks. Unfortunately the theory required [see e.g. Keller 1976, Lamport 1977 and 1980, Owicki and Gries 1976, and Pnueli 1981] for such proofs is still in its infancy. Correctness proofs, using these theories, for our algorithms will be too long and tedious. And, we are afraid, also uninteresting to the intended audience. Consequently we include, only where we felt necessary, informal arguments that go toward strengthening our intuition about the correctness of the algorithms.

Although a number of theoretical studies have been reported on parallel processing of graphs [Arjomandi 1975, Bentley 1980, Hirschberg et al. 1979, Reghbati and Corneil 1978, and Savage 1977] very few of them have considered the single source shortest path problem. To the best of our knowledge, only one study [Deo et al. 1980] has actually designed, coded and executed a parallel shortest-path algorithm on a real MIMD computer.

The paper is organized as follows. In Section 2, some of the relevant definitions are given. In Section 3, we describe the sequential versions of the two single source shortest path algorithms that are to be parallelized. Section 4, the main body of this paper, contains various parallelized versions of the two shortest path algorithms; and finally, in Section 5 we make some concluding remarks.

2. Definitions and Notations

2.1 Graphs

A *directed graph* $G=(V,E)$ is an ordered pair of a set V of N *vertices*, and the set $E \subseteq V \times V$ of *edges*. An edge (u,v) of E is said to be *directed* from vertex u to v. There is a function L that maps pairs of vertices into real numbers. If (u,v) is an edge, we consider $L(u,v)$ to be the *distance* from u to v. If (u,v) is not on edge, $L(u,v)=\infty$. We also define $L(u,u)$ to be zero, for all u. The *length* of a sequence S of vertices $v_1, v_2, \ldots, v_k$ is $\mathrm{len}(S)=\mathrm{sum}$ of $L(v_i, v_{i+1})$, for $1 \leq i < k$. The sequence S is a *path* from v_1 to v_k if (v_i, v_{i+1}) are all edges of G.

The *shortest distance* $\mathrm{sd}(u,v)$ from vertex u to v, is given by the minimum $\mathrm{len}(S)$ for all sequences S of vertices beginning with u and ending with v. A *shortest path* from u to v is a path P such that $\mathrm{len}(P)=\mathrm{sd}(u,v)$. The *single source shortest path problem* is that of finding all the shortest paths from a distinguished vertex, called *source*, to all other vertices.

For any vertex u, $A(u)=\{v \mid (u,v) \text{ is an edge}\}$ is the set of *adjacent* vertices of u.

Without any loss of generality we take V to be $\{1,2,\ldots,N\}$. The function L is often represented as an $N \times N$ matrix. Here, we only assume that $L(u,v)$ is computable in a constant time. The set $A(u)$ is often represented as a linear list. As noted in Section 3.1, the ordering of vertices in $A(u)$ can have a significant impact on the speed of an algorithm.

2.2 Syntax of Algorithms

For any two statements $s1$ and $s2$, by $s1; s2$ we denote the execution of $s1$ first, and that of $s2$ after $s1$ has terminated. By $s1 \| s2$ we denote, as in [22], that the executions of both $s1$ and $s2$ may begin simultaneously. We consider $(s1 \| s2)$ to have terminated only after both $s1$ and $s2$ have. We use the **if-then-else**, **while**, **for** and **repeat** statements with their traditional semantics. The statement **loop** s **end** executes s repeatedly. If s has an **exit-loop** clause, it is possible for the looping to terminate, otherwise it will not. We show grouping of statements both by explicitly bracketing them in pairs of words such as **do-od, if-fi** and by indentation to improve readability.

We prefix the word "**process**" to an algorithm-segment to indicate that it is a significant but independent chunk of parallel computation. By **start** P, we indicate that the execution of process P should be triggered. Thus (s; **start** P) executes s, starts P and then terminates.

We indicate all necessary synchronization among processes explicitly. Often this is accomplished by means of a global variable. The primitive **lock** x can only be completed by exactly one process. Until it is **unlocked** by that process, all other processes attempting to lock the same variable x can be conceptualized as waiting at the entrance to x.

We require that $\infty + c = \infty$, for any c.

3. The Sequential Algorithms

The two sequential algorithms that we choose to parallelize are perhaps the best known shortest path algorithms [Deo and Pang, 1980]. Since we need a standard reference to compare our parallel algorithms to, we describe here the sequential algorithms mentioning some not as well-known facts about them. A discussion of our first sequential algorithm $S/1$ can be found in [Deo, 1974] and that of the second algorithm in both [Deo, 1974] and [Horowitz and Sahni, 1978].

3.1 Algorithm S/1

This algorithm is originally due to Moore [1957]. The version given here is that of Pape [1980]. The algorithm is quite an efficient one in practice; however, in its worst case it can take an exponential amount of time on certain graphs when the adjacency list is ordered in a certain way [Deo and Krishnamoothy, 1981]. Note that the algorithm does not terminate if negative cycles exist in the given graph.

Algorithm S/1 (Moore)

```
for all vertices $u$ do
    $D[u] := \infty$;
od;
$D$ [source] $:= 0$;
initialize $Q$ to contain source only;
while $Q$ is not empty do
    Delete $Q$'s head vertex $u$;
    for each edge $(u, v)$ do
        if $D[v] > D[u] + L[u, v]$ then
            $P[v] := u$;
            $D[v] := D[u] + L[u, v]$;
            if $v$ was never in $Q$ then
                insert $v$ at the tail of $Q$;
            fi;
            if $v$ was in $Q$ but is not in currently then
                insert $v$ at the head of $Q$;
            fi;
        fi;
    od;
od;
```

Note that we can replace the queue Q of the algorithm by a set R. Line 5 would then be replaced by $R := \{\text{source}\}$, lines 12 to 17 by $R := R \cup \{v\}$, the test in line 6 by"R is not empty" and line 7 by "u any element of R; $R := R - \{u\}$". We use this variant of algorithm $S/1$ in some of our parallel algorithms. Experimental studies [Pape, 1980] indicate that in the sequential case the double-ended queue reduces execution time. We do not know if this is true in the parallel versions of the algorithm also.

3.2 Algorithm S/2

This algorithm is originally due to Dijkstra [1959]. It assumes that $L(u, v) \geq 0$ for all u, v. The time taken by this algorithm is $O(N^2)$. When the algorithm is modified to account for negative edge-lengths, the worst case becomes exponential [Johnson, 1973].

Algorithm S/2 (Dijkstra)

```
 1   for $i := 1$ to $N$ do
 2       $S[i] := 0$;
 3       $D[i] := L(\text{source}, i)$;
 4   od;
 5   $S[\text{source}] := 1$;
 6   for $j := 2$ to $N-1$ do
 7       choose $u$ such that $D[u] = \min\{D[v] : S[v] = 0\}$,
 8       $S[u] := 1$;
 9       for all $v$ with $S[v] = 0$ do
10           $dv := D[u] + L[u, v]$;
11           if $D[v] > dv$ then
12               $D[v] := dv$;
13               $P[v] := u$;
14           fi;
15       od;
16   od;
```

A modification that we will use later on is to replace line 9 by simply "for all vertices v do". This change preserves correctness because if $S[v] = 1$ for a vertex v, its shortest distance sd (source, v) is already computed and is found as $D[v]$.

4. Parallel Algorithms

We now present ten algorithms for a variety of parallel machines. Our algorithms should be considered versions of $S/1$ and $S/2$ where a program transformation has been applied that relaxes the control flow into parallelism that satisfies only the required precedence constraints. Communication issues [Lint and Agerwala, 1981] do not dominate in our algorithms, except in those for the tree machine.

The correctness of all the (sequential and parallel) algorithms in this paper follows from essentially the three properties listed below.

$F1$: For any vertex u, either $D[u]=\infty$, or there exists a path P from source to u such that $\text{len}(P)=D[u]$.

$F2$: Let P be a path from source to a vertex u. Then $D[u]\leq \text{len}(P)$ will be true at some point in time during the execution of the algorithm.

$F3$: The temporal sequence of successive values assigned to $D[u]$ is non-increasing, for any u.

Properties $F1$ and $F2$ together imply that eventually $D[u]=\text{sd}(\text{source}, u)$, for all u. Property $F3$ guarantees that the algorithms do not zig-zag toward this goal but progress in a uniform way. Note that these properties do not guarantee termination of an algorithm. This part of the proof is accomplished for each algorithm by placing a time-bound on when $F2$ will become true for all shortest paths.

4.1 A Highly Parallel Algorithm

Let us begin by first observing which of the many precedence relationships present in a sequential computation are necessary for the shortest path problem. An algorithm can then be designed to "maximize" the amount of parallelism by observing only the necessary precedence constraints. Such an algorithm can further be moulded to fit the architecture of a given parallel computer. Also, such an algorithm might be a partial answer to the important question of what the minimum time is for the shortest path problem we are considering assuming that no parallelization constraints (such as the present state of technology) exist. The following two algorithms are the results of such an attempt to "maximize" the parallelism.

4.1.1 Algorithm $P/1$

Consider the sequential algorithm $S/1$. Clearly, the initializations as indicated by lines 1 to 5 have to be performed before the rest of the computation. The Q contains only those vertices whose distances from source have decreased and consequently the distances to their adjacent vertices should be re-computed. Certainly, we can compute these for each vertex of Q. We only need to be careful as to how the various distances computed by many parallel computations to a given vertex v will be coordinated into yielding at all times the least possible value. For this purpose, let us postulate the existence of a mechanism, which surrounds each $D[u]$, such that the value retained in $D[u]$ is always the least of the current value of $D[u]$ and all the attempted update values appearing on the right hand side of assignments. Thus, e.g., if we surround the variable x, whose current value is, say a, with this mechanism,

$$(x:=2 \,||\, x:=1 \,||\, x:=3)$$

would leave the least of a, 1, 2, and 3 as the value of x. (This assumption is analogous to one that [Lamport, 1980] makes. We do not use this mechanism in other algorithms.)

Algorithm P/1

```
 1  initialize
 2     ($D[source]:=0$ ||
 3     for all $u \neq$ source do ||
 4        $D[u]:=\infty$;
 5     od);
 6     for all $u$ do ||
 7        start $P1(u)$;
 8     od;
 9  where
10  process $P1(u)$ is
11     loop
12        for each $v$ adjacent to $u$ do ||
13           $D[v]:=\min(D[v], D[u]+L(u,v))$;
14        od;
15     end;
```

It is easy to see that properties $F1$, $F2$ and $F3$ are valid for the algorithm $P/1$. But, obviously the algorithm is nonterminating. In spite of this, the values of $D[.]$ would have converged to sd (.) in $O(N)$ steps, assuming that the operation specified in lines 12 to 14 can be done in a unit of time.

If we now parallelize the algorithm $S/2$ along similar lines, it turns out to be the same as $P/1$. This curious phenomenon strengthens our belief that some sequential algorithms are simply the same basic computations but packaged differently in control structures.

4.2 An Algorithm in Hardware

Suppose we have N processors each capable of adding a scalar d to each element of a vector A of size N. For a network of such processors, we can adapt algorithm *P/1* by

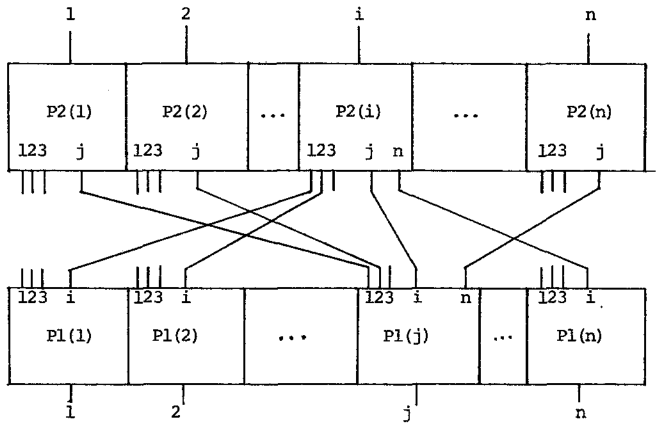

Fig. 1. Network version of $P/2$. (Only connections to $P2(i)$ and $P1(j)$ are shown)

(a) simply observing that if a vertex v is not adjacent to u, $L(u, v) = \infty$ and therefore letting the for-loop range over all vertices (whether adjacent or not) would cause no harm, and (b) implementing the mechanism whose existence we postulated in $P/1$.

The resulting network is shown in Fig. 1. It can be algorithmically expressed as $P/2$. The assignments in lines 4, 10 and 15 are array (vector) assignments and the operations involved are vector operations. The notation $tD[y, *]$ stands for the y-th row vector sliced from a two-dimensional array tD; similarly $tD[*, x]$ gives the x-th column vector. The min operation of line 10 finds the minimum value of the set of values in the vector. The plus operation of line 15 adds to each element $L[u, i]$ the value of $D[u]$ giving a new vector which is then assigned to the u-th row vector of tD.

The element $tD[i,j]$ of the $N \times N$ array records the length of the shortest path found so far from source to vertex j whose last but one vertex is i. That $F2$ holds for $P/2$ follows immediately once we observe that the value, say d, of $D[u]$ prior to $D[u] := \min(tD[*, u])$ was (perhaps still is) a member in the u-th column vector of tD.

Algorithm P/2

```
 1   **initialize**
 2      ($D[u] := \infty$, for all $u \neq$ source ||
 3      $D$[source] := 0);
 4      $tD[u, *] := D[u] + L[u, *]$, for all $u$;
 5      (**start** $P1(u)$, for all $u$ ||
 6      **start** $P2(u)$, for all $u$)
 7   **where**
 8   **process** $P1(u)$ **is**
 9      **loop**
10         $D[u] := \min(tD[*, u])$;
11      **end**
12   **and**
13   **process** $P2(u)$ **is**
14      **loop**
15         $tD[u, *] := D[u] + L[u, *]$;
16      **end**
```

Note that $P/2$ has no conflicting assignments to a location. Only the process $P1(u)$ updates $D[u]$; only $P2(u)$ accesses its value. Similarly the row- and column-vectors of tD are updated and accessed respectively by $P1$- and $P2$-processes.

The most elegant aspect of this algorithm is that the responsibility of computing the shortest distance to the vertex u is left to $P1(u)$ alone.

As it is in $P/1$, the algorithm is non-terminating. Also, it is written for asynchronous MIMD operation. But, an elegant synchronous network (see Fig. 1) can be derived from it.

The $P1$'s are now minimum-finding processors that accept n inputs and produce one output. The $P2$'s each produce n outputs. The j-th output value of $P2(i)$ is equal to the input value x received plus $L[i,j]$. The connections between $P1$'s and $P2$'s are as follows. The single output of $P1(j)$ is connected to the input of $P2(j)$; thus the input value x received by $P2(i)$ is the shortest distance found so far to the vertex i.

Initially, the values $\infty, \infty, \ldots, 0, \ldots, \infty$ are fed to the inputs of $P2$'s the $P2$(source) receiving the zero. Outputs are "collected" from the $P1$'s after $O(N \log N)$ time units. This estimate is based on the following. The $P2$'s can complete their operation in one time unit, and the $P1$'s in $\log N$ time units. Assuming one unit of time delay for the outputs of $P2$ to reach the $P1$'s and vice versa, we see that after every $2+\log N$ time units one of the vertices will receive its ultimate shortest distance ("permanent labeling" in the sense of Dijkstra's algorithm $S/2$).

4.3 Algorithm for Array Processors

Algorithm $P/3$ is very similar to $P/2$ except that termination is provided by a very simple condition and is written as a sequential algorithm for array processors. The following operations involve vectors and, except for indexnz, yield a vector as their result as defined below.

$$\begin{aligned}
&\mathrm{elmin}(A,B)[i] &&= \min(A[i], B[i])\\
&(d+A)[i] &&= d+A[i]\\
&(A<B)[i] &&= 1, \text{ if } A[i]<B[i]\\
& &&= 0, \text{ otherwise}\\
&(A\cup B)[i] &&= A[i] \textbf{ or } B[i]\\
&\mathrm{indexnz}(A) &&= i, \text{ if } A[i]=0 \text{ for some } i\\
& &&= 0, \text{ if no such } i \text{ exists}
\end{aligned}$$

Note that indexnz as defined is nondeterministic; what value it should return is of not much interest. We assume that the indices range from one to N.

Algorithm P/3

```
S := [0, 0, ..., 0, 0, 0, ..., 0];                  /* all zero */
D := [∞, ∞, ..., 0, ..., ∞, ∞, ..., ∞];             /* D[source] = 0 */
u := source;
while u ≠ 0 do
    S[u] := 0;
    DA := D[u] + L[u, *];
    D := elmin(D, DA);
    S := S ∪ (DA < D);
    u := indexnz(S)
od;
```

Toward the end of Section 3.1, we discussed a variant of algorithm $S/1$ in which the queue was replaced by a set R. In the above algorithm, the $(0,1)$-vector S serves the same purpose as that of R. After line 8 is executed, S has 1's in those index positions corresponding to vertices whose distance from source has decreased and which have not since been examined. The execution time of this algorithm is not lower than that of the sequential $S/1$ (if we assume that the graph is not dense, i.e., the out-degree of vertices is a very slowly increasing function of N) because the number of times the loop of lines 4 to 10 is executed equals the sum of the number of times each vertex is entered into S.

However, a minor change in the algorithm not only makes it a parallel version of $S/2$, but also reduces its execution time to $O(N \log N)$ as in the network of Fig. 1. Consider the operation

$$\begin{aligned} x\min(D,S) &= i, \quad \text{if } D[i] = \min\{D[j] \mid S[j]=0\}, \\ &= 0, \quad \text{if the set } \{D[j] \mid S[j]=0\} \text{ is empty.} \end{aligned}$$

The S in $x\min(D,S)$ is a selector of elements of D and thus we find the index of the smallest one. Algorithm $P/4$ uses this operation in line 7. It loops exactly N times. Lines 5 and 6 of $P/4$ take a constant amount of time while $x\min$ takes $O(\log N)$.

Algorithm P/4

```
1   $S := [0, 0, \ldots, 0, \ldots, 0, 0, \ldots, 0]$;
2   $D := [\infty, \infty, \ldots, 0, \infty, \infty, \ldots \infty]$;
3   $u :=$ source;
4   while $u \neq 0$ do
5       $S[u] := 1$;
6       $D :=$ elmin $(D, D[u] + L[u, *])$;
7       $u := x\min(D, S)$;
8   od
```

4.4 Algorithms for a Tree Machine

The tree machine of Bentley and Kung [1979] consists of a large number ($O(N)$) of three kinds of processors (shown as circles, squares and triangles in Fig. 2). The circles are capable of broadcasting data they receive. The triangles can combine their inputs in an elementary way. The tree machine as defined by Bentley and Kung is a synchronous machine. Our version of the tree machine makes slightly different assumptions. In particular, we assume that the squares are small von Neumann machines that can store a program and data of size $O(N)$. Since we do not yet have a good notation to write programs for synchronous machines in a compact manner, we write ours as though the machine were asynchronous. The required synchronization is explicitly shown (by the primitives **wait**, **receive** and **send**), or is explained in the text.

The **send** operation is not completed until the intended receiver receives the data sent. Similarly, a **receive** operation is not completed until the intended sender has sent the data. However, a process may test if input is ready for it to receive. the intended receiver/sender is clearly identified by the connections shown in Fig. 2.

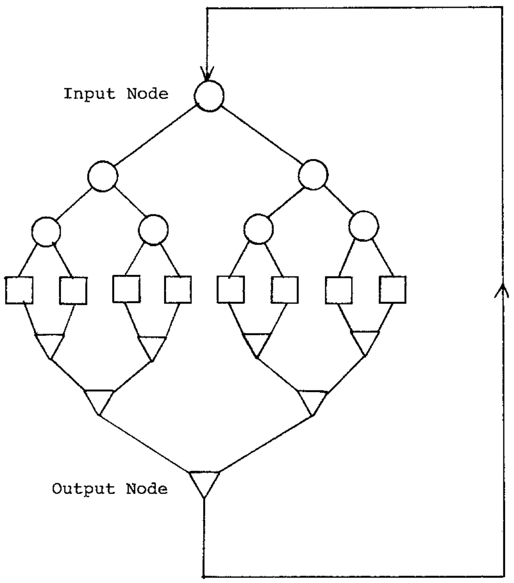

Fig. 2. The tree machine

Both sequential algorithms $S/2$ and $S/1$ are parallelized for the tree machine. We present the parallel algorithms as consisting of processes each designated to execute either on a square processor or a triangle processor. The circle processors only broadcast the input they receive to their descendants, so we don't show processes for them.

Both algorithms given here use the adjacency lists $A[u, *]$ for each vertex u. We assume that following the last adjacent vertex is a 0 indicating the end of the list. Except for the arrays A and L all other variables are local to each processor.

4.4.1 Algorithm $P/5$

This is a parallelization of $S/2$. The task of recomputing the distance to a vertex u is that of square processor u. It is always the case in $P/5$ that if square processor x receives input value pair (x, dx) then $dx = \text{sd}$ (source, x). Other square processors receive this pair, and recompute the distances to the vertices they are in-charge of, as in lines 9 to 15 of $S/2$. Note the conspicuous absence of the set S of $S/2$; the same information is now found in the fact that a certain square processor is executing outside the first loop, repeatedly sending $(0, \infty)$ for the dummy vertex 0. The algorithm can be terminated when the bottom most triangle sends out $(0, \infty)$.

Algorithm P/5

```
1   initialize
2       start all processes;
3       send (source, 0) to topmost circle;
```

```
where
process square ($u$) is
    local $x, dx$;  $D[u] := \infty$;
    loop receive ($x$, $dx$);
        if $x = u$ then $D[u] := dx$; exit loop fi;
        $D[u] := \min(D[u], dx + L(x, u))$;
        send ($u$, $D[u]$);
    end;
    loop send (0, $\infty$) end;
and
process triangle ($t$) is
    loop
        (receive ($x$, $dx$) from left ||
        receive ($y$, $dy$) from right);
        if $dx < dy$ then send ($x, dx$) else send ($y, dy$) fi;
    end
```

It is easy to see that $P/5$ is an $O(N \log N)$ algorithm since the square processors receive inputs at intervals of $\log N$ units of time as a result of the minimum value percolating through the triangles.

4.4.2 Algorithm $P/6$

In this parallelization of $S/1$, we let each square processor compute the distance to a vertex i whose value is synchronously incremented modulo N by all squares. The square processors work in step in each iteration of their loops as a result of the **receive** operation. The triangle processors are now "pipelined" to compute the minimum among the distances to vertex i. After an initial delay of $\log N$, the bottom-most triangle is ready to send its output to the topmost circle at every unit of time.

Algorithm P/6

```
initialize
    start all processes;
    send (source, 0) $\log N$ times;
where
process square ($u$) is
    local $x$, $dx$, $i$;
    $i := 0$;
    $D[u] := \infty$;
    loop
        receive ($x$, $dx$);
        if $x = u$ & $dx < D[u]$ then $D[u] := dx$ fi;
        $i := (i \bmod N) + 1$;
        send ($i, D[u] + L(u, i)$);
    end
```

4.4.3 Algorithm $P/7$

Dijkstra [1959] presents two algorithms; one we have been considering so far for the shortest path problem, the other for finding minimal spanning trees. The two algorithms have similar syntactic structure. Consequently, a parallelization of $S/2$ can be obtained by following the line of development of Bentley [3] who parallelizes the minimal spanning tree algorithm.

4.5 Algorithms for MIMD Machines

The multiple-instruction, multiple-data stream machines consist of several, say K, processors. Each executes its own program in its own memory but with a strong coupling to other processes via a common memory. All necessary synchronization among processors is via global variables stored in the common memory. In the interest of clarity, we use the following abbreviations:

$$\textbf{increment}\, x = (\textbf{lock}\, x;\ x := x + 1;\ \textbf{unlock}\, x)$$

$$\textbf{decrement}\, x = (\textbf{lock}\, x;\ x := x - 1;\ \textbf{unlock}\, x)$$

We now give several algorithms for MIMD machines. So far we have quite successfully avoided explicit synchronization among the processes. Here we find this almost unavoidable, and for this reason these algorithms are not as elegant as the previous ones.

4.5.1 Algorithm $P/8$

This algorithm is a restructured version of algorithm PPDM of [Deo et al., 1980]. As can be seen, the algorithm is a complex one. We describe it briefly here referring the reader to [Deo et al., 1980] for more details about its correctness.

The algorithm uses three global variables – WAIT, DONE and MSYN – for synchronization among the $K-1$ "worker" processes and the "master" process. The variable MSYN is updated only by the master process. The master process sets MSYN to "no" when it has successfully obtained a vertex to reach out from the Q. Note that just before it tries to retreive an element from Q, MSYN is always equal to "yes". The value of MSYN remains a "yes" as long as it finds the Q to be empty while looping at lines 15 to 27. Meanwhile, let x be the number of worker processes looping at lines 37 to 39. Then, the remaining workers must be executing reachout of line 43. If one of them inserts an element into Q then the master will retrieve it and execute the lines 23 – 24 before resuming the above protocol. If Q remains empty after all workers complete reachout, they will all eventually be looping at lines 37 to 39. Clearly then WAIT $= K-1$ and the master initializes DONE to 1 and workers follow suit and increment DONE as they terminate.

```
1    initialize
2        start master;
3    where
4    process master is
5        MSYN := yes; WAIT := 0; DONE := 0;
```

```
 6          for $i := 2$ to $K$ do
 7              start worker ($i$);
 8          od;
 9          for $u := 1$ to $N$ step $K$ do
10              $D[u] := \infty$;
11          od;
12          while WAIT $< K-1$ do od;
13          $D[source] := 0$;
14          initialize $Q$ to contain source only;
15          loop
16              get $-u$;
17              if $u = 0$ then
18                 if WAIT $= K-1$ then
19                    DONE $:= 1$;
20                    exit loop
21                 fi;
22              else
23                 MSYN := no;
24                 reachout ($u$);
25                 MSYN := yes;
26              fi;
27          end;
28          while DONE $< K$ do od;
29      and
30      process worker ($i$) is
31          for $u := i$ to $N$ step $K$ do
32              $D[u] := \infty$;
33          od;
34          loop
35              while MSYN = yes do
36                 increment WAIT;
37                 repeat
38                    if DONE $> 0$ then goto 45 fi;
39                 until MSYN = no;
40                 decrement WAIT;
41              od;
42              get $-u$;
43              if $u > 0$ then reachout ($u$) fi;
44          end;
45          increment DONE;
```

The procedures get $-u$, reachout and insert of the above algorithm are also used in the next algorithm. These are given below.

```
procedure get $-u$ is
    lock $Q$;
    if  $Q$ is not empty then
        delete head $u$ of the $Q$;
    **else** $u := 0$
    **fi**;
    **unlock** $Q$;

**procedure** reachout ($u$) **is**
    **for** each arc $(u, v)$ **do**
        newdv: $= D[u] + L[u, v]$;
        **lock** $D[v]$;
        **if** $D[v] \leq$ newdv **then unlock** $D[v]$
        **else**
            $P[v] := u$;
            $D[v] :=$ newdv;
            **unlock** $D[v]$;
            insert ($v$);
        **fi**;
    **od**;

**procedure** insert ($v$) **is**
    **lock** $Q$;
    **if** $v$ was never in $Q$ **then**
        insert $v$ at the tail **of** $Q$;
    **fi**;
    **if** $v$ was in $Q$ but is not currently **then**
        insert $v$ at the head **of** $Q$;
    **fi**;
    **unlock** $Q$;
```

4.5.2 Algorithm $P/9$

This algorithm is another MIMD parallelization of $S/1$. However, it is a "symmetric" algorithm, where no master-worker distinction is made unlike the preceding algorithm. The "master" process starts $K-1$ worker processes and then behaves like a worker. The global variable INIT is used to make sure that the initialization of the D array is complete before beginning to compute the distance from source. The variables dqempty, local to each process, are true if the process has tested Q and discovered that it is empty. After it is set to true, if a subsequent test shows that Q has become nonempty (because another process has inserted an element) dqempty is set to false again. This boolean's only purpose is to correctly increment/decrement EMQ, the number of processes which now find Q to be empty.

Algorithm P/9

```
**initialize**
    INIT: $= 0$; EMQ: $= 0$;
    initialize $Q$ to contain source only;
    **for** $i := 1$ **to** $K$ **do**
        **start** worker ($i$);
        od;
    where
    process worker ($i$) is
        local $u$, dqempty;
        for $u := i$ to $N$ step $K$ do
            $D[u] := \infty$;
        od;
        $D$[source] $:= 0$;
        dqempty := false;
        increment INIT;
        while INIT $\neq K$ do od;
        repeat
            get $- u$;
            cases
                $u = 0$ & dqempty:
                    /* do nothing */
                $u = 0$ & not dqempty:
                    increment EMQ; dqempty := true;
                $u \neq 0$ & dqempty:
                    decrement EMQ; dqempty := false;
                    reachout ($u$);
                $u \neq 0$ & not dqempty:
                    reachout ($u$);
            end cases;
        until EMQ $= K$
```

4.5.3 Algorithm *P*/10

This algorithm is a parallelization of *S*/2. The master process starts off, as usual, $K-1$ workers who after performing the initialization of D, will propose a candidate in $C[p]$ to be the next vertex to become permanently labelled. The master process picks the finalist from among these. We use the dummy vertex 0 to indicate that a certain worker has no candidate vertex to propose.

The global counter INIT is used to signal the completion of the initialization of D and S arrays. The counter nu is used for communication between the master and the workers. When nu $= 0$, the next vertex to become permanently labeled has been chosen. When nu $= K-1$, the workers have all updated the C array that contains their candidate vertices, so that the master can begin its selection procedure.

Algorithm P/10

```
    initialize
        start master;
    where
    process master is
        local $i, j, d$;
        $D[source] := 0$;
```

```
 7         $u$:=source;
 8         nu:=0;
 9         INIT:=0;
10         for $i:=1$ to $K-1$ do
11             start worker ($i$);
12         od;
13         while INIT$\neq K-1$ do od;
14         while $u \neq 0$ do
15             $S[u]:=1$;
16             while nu$\neq K-1$ do od;
17             $u:=0$;
18             $d:=\infty$;
19             for $i:=1$ to $K-1$ do
20                 $j:=C[i]$;
21                 if $j\neq 0$ then
22                     if $D[j]<d$ then $d:=D[j]$; $u:=j$; fi;
23                 fi;
24             od;
25             nu:=0;
26         od;
27     and
28     process worker ($w$) is
29         local $i, j, d$;
30         for $i:=w$ to $N$ step $K-1$ do
31             $D[i]:=\infty$;
32             $S[i]:=0$;
33         od;
34         increment INIT;
35         while INIT$\neq K-1$ do od;
36         while $u\neq 0$ do
37             $C[w]:=0$;
38             d$:=\infty$;
39             for $i:=w$ to $N$ step $K-1$ do
40                 if  $S[i]=0$ then
41                     $D[i]:=\min(D[i],\ D[u]+L(u,i))$;
42                     if $d>D[i]$ then $d:=D[i]$; $C[w]:=i$ fi;
43                 fi;
44             od;
45             increment nu;
46             while nu$\neq 0$ do od;
47         od;
```

Note that while the master process is selecting the finalist, the workers are all waiting. Also, if the graph is sparse, the computation of line 42 of worker (w) doesn't change $D[i]$ often (because $L(u,i)=\infty$). In spite of this, the asymptotic time bound for the algorithm is $O(N*(N/K+K))$ which can be simplified to $O(N/K)$ assuming $K \ll N$.

5. Concluding Remarks

Below we comment on some of the transformational issues and issues arising from the problem domain of graph algorithms.

Algorithm Structure

Parallelization of known sequential algorithms can give us surprising insights into the nature of computation as we found in this case study. Algorithms that hitherto were considered "very different" become identical/similar under this program transformation when inessential control and data structure is absorbed into the abstraction. No doubt these details have a significant impact on the efficiency of the sequential algorithms. They do not appear to carry over to parallel versions.

Speed-up

The observation made in [Deo et al., 1980] that exploiting parallelism is difficult when the graph is sparse is corroborated by our experience in this and other graph algorithms. One wonders if the question can be formulated and answered precisely.

Computer Architecture

It is becoming increasingly clear that certain parallel architectures are hard to program. More case studies of the kind done here are needed before we can evaluate this difficulty.

Parallelization

As the word suggests, this is a program transformation. It is generally agreed that writing programs for parallel machines is several orders of magnitude more difficult than writing corresponding sequential programs. Here in lies the need for discovering proven techniques that parallelize sequential algorithms known to be correct. Such principles and techniques have value even if they always result in a loss of speed-up that might otherwise be possible. In addition there is a potential that such techniques might be (semi-) automatic and can be incorporated into system programs.

While we have concentrated on the parallelization of two specific algorithms, we were motivated by the possibility of discovering general principles of this rather unexplored area of program transformation.

References

[1] Arjomandi, E.: A study of parallelism in graph theory. Doctoral thesis, TR 86, Dept. of Computer Science, Univ. of Toronto, 1975.
[2] Baer, J. L.: Computer systems architecture, ch. 10. Potomac, Md.: Computer Science Press 1980.
[3] Bentley, J. L.: A parallel algorithm for constructing minimum spanning trees. J. Algorithms *1*, 51 – 59 (1980).

[4] Bentley, J. L., Kung, H. T.: A tree machine for searching problems. Proc. International Conf. on Parallel Processing, August 1979, pp. 257–266.

[5] Browning, S. A.: The tree machine: a highly concurrent computing environment. Ph. D. Thesis, Computer Science Department, California Institute of Technology, Pasadena, Calif., 1980.

[6] Deo, N.: Graph theory with applications to engineering and computer science. Englewood Cliffs, N. J.: Prentice-Hall 1974.

[7] Deo, N., Krishnamoorthy, M. S.: A note on worst case complexity of Ford-Bellman-Moore algorithm. Computer Science Department, Washington State University, Pullman, Wash., Technical Report CS-81-076, October 1981.

[8] Deo, N., Pang, C. Y.: Shortest path algorithms: taxonomy and annotation. Computer Science Department, Washington State University, Pullman, Wash., Technical Report No. CS-80-057, March 1980.

[9] Deo, N., Pang, C. Y., Lord, R. E.: Two parallel algorithms for shortest path problems. Proc. International Conf. on Parallel Processing, August 26–29, 1980, pp. 244–253.

[10] Dijkstra, E.: A note on two problems in connexion with graphs. Numerische Mathematik *1*, 269–271 (1959).

[11] Eckstein, D. M.: Parallel processing using depth-first and breadth-first search. Doctoral thesis, Department of Computer Science, University of Iowa City, Iowa, July 1977.

[12] Hirschberg, D. S., Chandra, A. K., Sarwate, D. V.: Computing connected components on parallel computers. Comm. ACM *22*, 461–464 (1979).

[13] Horowitz, E., Sahni, S.: Fundamentals of computer algorithms. Potomac, Md.: Computer Science Press 1978.

[14] Johnson, D. B.: A note on Dijkstra's shortest path algorithm. J. ACM *20*, 385–388 (1973).

[15] Keller, R. M.: Formal verification of parallel programs. Comm. ACM *19*, 371–384 (1976).

[16] Kung, H. T.: The structure of parallel algorithms. Advances in Computers *19*, 65–112 (1980).

[17] Lamport, L.: A new technique for proving the correctness of multiprocess programs. ACM TOPLAS, 1980.

[18] Lamport, L.: Proving the correctness of multiprocess programs. IEEE Trans. Software Engg. *3*, 125–143 (1977).

[19] Lawler, E. L.: Combinatorial optimization: networks and matroids. New York: Holt, Rinehart and Winston 1976.

[20] Lint, B., Agerwala, T.: Communication issues in the design of parallel algorithms. IEEE Trans. Software Engg. *SE-7*, 174–188 (1981).

[21] Moore, E. G.: The shortest paths through a maze. Proc. Internat. Symp. on Theory of Switching, 1957, pp. 285–292.

[22] Owicki, S., Gries, D.: Verifying properties of parallel programs: an axiomatic approach. Comm. ACM *19*, 279–284 (1976).

[23] Pape, U.: Algorithm 562: shortest path lengths. ACM Trans. on Math. Software *6*, 450–455 (1980).

[24] Pneuli, A.: The temporal semantics of concurrent programs. Theoretical Computer Science *13*, 45–60 (1981).

[25] Reghbati, E. (Arjomandi), Corneil, D. G.: Parallel computations in graph theory. SIAM J. Computing *7*, 230–236 (1978).

[26] Savage, C.: Parallel algorithms for graph theoretical problems. Doctoral Thesis, Mathematics Department, University of Illinois at Urbana-Champaign, August 1977, Report ACT-4, Coordinated Science Laboratory, University of Illinois.

P. Mateti
Department of
Computer Engineering and Science
Case Western Reserve University
Cleveland, OH 44106, U.S.A.

N. Deo
Computer Science Department
Washington State University
Pullman, WA 99164, U.S.A.