PROCEEDINGS

SUPERCOMPUTING '89:
SUPERCOMPUTING STRUCTURES & COMPUTATIONS

Editors
Prof. Lana P. Kartashev
and
Dr. Steven I. Kartashev

VOLUME I

International Supercomputing Institute, Inc.
1989
Algorithms for Shortest Paths Problems on an Array Processor

J. Yadegar
S. El-Horbaty
AMT,INC,IRVINE,CA,USA
UAE UNIV,AL AIN,

D. Parkinson
A.M. Frieze
UC,LONDON UNIV,UK
CMU,PITTSBURGH,USA

ABSTRACT

The innovation of parallel computers has added a new dimension to the design of algorithms. Parallel programming is not a simple extension of serial programming. We describe and implement parallel algorithms for the well known problem of finding shortest paths in a network. We present our computational experience using the massively parallel processor DAP.

Keywords: Shortest paths problems, network, grid graph, parallel algorithms, distributed array of processors, SIMD computer.

The longest path problem is the central component of critical path analysis which is applied for the economical scheduling, planning and controlling of large and complex projects. It is often known by acronyms such as CPS, CPM and PERT. It is noteworthy that the longest path problem is mathematically identical to a shortest path problem and thus, the algorithms described in this paper apply to all such problems.

There has, therefore, been a large number of papers and reports published on this subject. To mention but a few, the surveys by Dreyfus [10], the bibliographies by Golden and Magnanti [19] and Pierce [27], and the taxonomy and annotation by Deo and Pang [6]. Many papers have also been written to examine the relative merits of various shortest path algorithms, such as Dial et al. [8], Gilsinn and Witzgall [16], Glover and Klingman [17], Golden [18] and Hulme and Wisniewski [20].

The innovation of parallel computers has added a new dimension to the design of algorithms. Parallel programming is not a simple extension of serial programming.

Our motivation is this research has been to develop parallel algorithms for shortest path problems; namely, the single-source problem and all-pairs problem.

Although most networks arising in practical problems (e.g. road and project networks) have only non-negative arc lengths, there are, however, networks which have both negative and positive arc lengths (e.g. a negative arc length may represent cost and a positive arc length may show income). Our algorithm deals with such networks (the most general case), and a specialized version of it is used for grid networks, modeling "perfect" cities and VLSI systems. We have also described a parallel algorithm for the all-pairs problem.

1. INTRODUCTION

One of the most fundamental problems in network theory is to find the shortest and/or longest path(s) in a network. These problems are major components of numerous quantitative transportation and communication models which seek to improve efficiency by reducing travel time, minimizing congestion, increasing capacity, lowering the cost of transportation service, or improving vehicle routing. Such models usually use a network to represent the transportation system where it is required to find the minimum time, cost, distance, or maximum capacity between several pairs of points in the network. The former are often called shortest path problems and the latter are called longest path problems.

167
This research differs from most other works in this area - see, for example, Chandy and Misra [5], Deo, Pang and Lord [7], Frieze [15], Mateti and Deo [23] and Resta [29] - in that all the algorithms described in this paper have been implemented and executed on a real SIMD computer, called the Distributed Array of Processors (DAP). With this in mind, the emphasis of the paper is on implementation rather than the well known complexity results on the shortest path problem - see, for example, Lawler [22].

The rest of the paper is organized as follows. In Section 2, we outline very briefly the DAP. Section 3 contains the relevant definitions and notations. In Section 4, we describe a parallel algorithm for finding shortest paths for the single-source problem and present our computational results in Section 4.2. The parallel algorithm to deal with the rectangular grid network is described in Section 5, followed by some results in Section 5.2. In Section 6, we present a parallel algorithm for the all-pairs problem and give computational results in Section 6.2. We conclude the paper in Section 7.

2. THE DAP ARCHITECTURE

The general principle of the DAP - see Parkinson [25, 26] and Quinn [28] - is that of a SIMD machine as defined by Flynn [12]. On a p x p DAP, one can perform up to p^2 operations (of the same type) simultaneously. This parallel processing capability of the DAP is achieved by a p x p matrix of processors, called Processing Elements, each of which may operate independently on its own local store. Thus, it is convenient to think of the DAP as a square array of processors placed on a 2-dimensional grid in which each processor can communicate directly with its four neighbors. Specifically, we have an array of p^2 (p=32 or 64) processors P_{i,j}, 1 \leq i, j \leq p, where P_{i,j} can communicate directly with its four neighbors P_{i-1,j}, P_{i,j-1}, P_{i+1,j} and P_{i,j+1}, with 1 \leq i,j \leq p and P_{p+1,j} = 1. In addition, processors are connected via row and column highways to a set of edge registers in such a way that in unit time data can be selected from any set of processors, one per row (or column), into the corresponding register; or data can be broadcast from a register to some or all processors in the same row (or column). The row and column highways coupled with the bit serial nature of the processors, allows the DAP to exhibit many properties of associative or content addressable processors [14]. For further details on the DAP and its programming language, Fortran-Plus, we refer the reader to [1, 2].

We measure the complexity of the algorithms in terms of number of "DAP operations", each one is executable very "efficiently" on the DAP. Some of these are:

(i) Compute the resultant matrix (a_{ij} \circ b_{ij}) where \circ = +, -, ** or /\). The p x p matrices (a_{ij}) and (b_{ij}) are stored off element per processor.

(ii) Overwrite a row or column of a p x p matrix with a given p-vector.

(iii) Given a p-vector V, construct a p x p matrix MATR(V), where each row is identical to V. Similarly, MATC(V) has each column identical to V.

(iv) Compute the position(s) of the minimum value of a p x p matrix or a p-vector.

On the DAP, one is able to execute these operations in O(1) time. To get a feeling of the actual computation times for these operations on the existing DAPs with p=32 and p=64, Table 2-1 gives the ratio of the time for an operation compared to that for a 32 bit floating point matrix (element by element) multiplication as defined in (i) above. The time for matrix (element by element) multiplication is 155.7 micro seconds, in which time 1024 multiplications are performed when p=32 and 4096 multiplications are performed when p=64.

<table>
<thead>
<tr>
<th>Function</th>
<th>Operation Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>*</td>
<td>1.0</td>
</tr>
<tr>
<td>+</td>
<td>0.67</td>
</tr>
<tr>
<td>/</td>
<td>1.36</td>
</tr>
<tr>
<td>A(i)=V or A(j)=V</td>
<td>0.13</td>
</tr>
<tr>
<td>MATR(V) or MATC(V)</td>
<td>0.11</td>
</tr>
<tr>
<td>MINP(A)</td>
<td>0.14</td>
</tr>
<tr>
<td>MINP(V)</td>
<td>0.14</td>
</tr>
</tbody>
</table>

TABLE 2.1: Ratio of the DAP computation time for an operation compared to that of a 32 bit floating point matrix (element by element) multiplication.
In Table 2.1, A is a p x p matrix and V is a p-vector. The assignment \( A(i,j) = V \) overwrites the \( i \)th row (\( j \)th column) of A by V. Functions MATR and MATCH have already been defined in (iii) above. Function MINP takes a matrix (or vector) argument and returns a logical matrix (or vector) with TRUE value(s) corresponding to the position(s) of the minimum value of its argument. For more detail on these functions and other Fortran-Plus functions we refer the reader to [3].

3. DEFINITIONS AND NOTATIONS

Before describing our algorithms, it is appropriate to summarize some of the conventions and terms we shall use in this paper.

We shall deal with a directed graph or digraph \( G = (V, A) \) consisting of a finite set \( V = \{1, 2, ..., n\} \) of elements called nodes or vertices and a finite set \( A \subseteq V \times V \) of \( m \leq n \) ordered pairs of nodes called arcs or edges. An edge \((u, v)\) of \( A \) is said to be directed from \( u \) to \( v \), where \( v \) is called a successor of \( u \), and \( u \) is called a predecessor of \( v \).

A directed path or path from node \( s \) to node \( t \) (called an \( s-t \) path) is a finite sequence of arcs \((s, v_1), (v_1, v_2), ..., (v_{k-1}, t)\), sometimes written as \((s, v_1, ..., t)\), in which all the vertices are distinct. Note that such a path has no loops. An \( s-t \) path is a cycle if \( s = t \).

By a network, we mean a digraph \( G = (V, A) \) together with a real valued function \( l: A \rightarrow R \) defined as follows:

\[
\begin{align*}
    l_{ij} &= \text{the (finite) length of arc } (i, j) \\
    l_{ii} &= 0 \\
    \text{otherwise}
\end{align*}
\]

As mentioned earlier, in a (general) network, \( G \), the arc lengths may be either positive or negative. However, we assume that \( G \) contains no directed cycle with a strictly negative length.

It is noteworthy that an undirected shortest path problem can be reduced to a directed one, if all arc lengths are non-negative, by replacing each undirected arc \((i, j)\) by a symmetric pair of directed arcs \((i, j)\) and \((j, i)\) with \( l_{ij} = l_{ji} \). However, if the length of arc \((i, j)\) is negative, then such a transformation would introduce a negative directed cycle into the network, and this approach is invalid. In such cases, the problem can be reduced to that of a weighted non-bipartite matching problem [21].

3.1. Computer Representation of a Network

A network may be represented in a computer in several ways and the manner in which it is represented directly affects the performance of algorithms applied to the network. There is no single overall best data structure for networks. The choice depends on the size of the network, its density and on the computer as well as the language being used.

The length-matrix \( l \) in which the function \( l \) is represented as an \( n \times n \) matrix, and the arc-lists in which for each node \( i \), there is a list of those arcs \((i, j)\) which are incident to node \( i \), together with their lengths \( l_{ij} \), are perhaps the most used data structures for networks - see Sislo, Deo and Kowalik [30].

Below in Figure 3.1, we give an example of a network with its length-matrix as well as arc-lists representations.

![Network Diagram](image)

\[
\begin{bmatrix}
0 & 8 & 5 & 10 & 0 & 0 \\
4 & 0 & 6 & 11 & 0 & 0 \\
5 & 3 & 0 & 4 & 0 & 0 \\
10 & 0 & 2 & 0 & 6 & 8 \\
0 & 5 & 0 & 3 & 0 & 0 \\
0 & 11 & 3 & 2 & 0 & 0 \\
\end{bmatrix}
\]

Tails = (1 2 2 3 3 3 4 4 4 5 5 6)
Heads = (7 3 4 5 1 4 6 2 3 5 6 4 6 5)
Arc lengths = (8 5 6 4 11 3 4 2 2 3 6 8 5 3 2)
Algorithm SP solves the problem by successive approximations. That is, given an approximation of a shortest path from the source to node \( j \), we try to improve this approximation by considering paths via predecessor nodes of \( j \). Formally, let:

\[
d_{d}(t) = \text{the length of a shortest path from the source (say node 1) to node } j \text{, such that the path contains no more than } r \text{ arcs}
\]

There are two cases to be considered. One for problems smaller than or equal DAP size, and one for problems larger than DAP size.

**CASE 1:** \( n^2 \leq \text{number of processors } p^2 \)

Each processing element, \( PE_{t+1} \), keeps the value of \( l_j \) and also the value of \( d_{d}(t) \). Let \( d(t) \) be the vector holding the values \( d_1(t), d_2(t), \ldots, d_n(t) \). Initially, we let:

\[
d(1) = (l_{11}, l_{12}, \ldots, l_{1r})
\]

Then, we compute \( (r+1)^{st} \) order approximations from the \( r^{th} \) order for all \( j \in V \) simultaneously as follows:

\[
d(t+1) = d(t) O L
\]

Where \( L = (l_{ij}) \) is the arc length matrix, and \( O \) is defined by:

\[
d_j(t+1) = \min_{r \in \{d_k(t) + l_{kj}\}} \text{ for all } j \in V
\]

Here, we are assuming that \( l_{jj} = 0 \) for all \( j \in V \).

Pictorially, we have:

4. SINGLE-SOURCE SHORTEST PATH PROBLEM

The most commonly encountered shortest path problem is to find the shortest paths (or the lengths of shortest paths) from a specified node, called the source, to all other nodes in a network. This problem is usually known as the "single-source problem." In this section we describe a parallel algorithm for (the most general case) of this problem. That is, when the arc lengths can be positive or negative.

4.1. Algorithm SP

This algorithm finds the length of shortest paths from the source to every other node in a directed network in which the arc lengths can be positive or negative. Of course, as mentioned earlier, we do, however, assume that there are no negative cycles.
Now, the ability of DAP to broadcast using the row and/or column high ways means that in $O(1)$ time operation, all the processors $PE_k$ will have simultaneously the corresponding value of $d_k^{(r)} + l_{kj}$, for all $k, j$. It only remains to find the minimum across each column, and this can be done on the DAP simultaneously very efficiently.

It is perhaps interesting to observe a certain similarity between (4.2) and algebraic multiplication of a row vector $d_1^{(r)}$ by a matrix $L_1$ that is, $d_k^{(r+1)} = \sum_{j=1}^{p} d_j^{(r)} \cdot l_{kj}$ for all $j$, when replacing the summation and multiplication by taking minima and addition in (4.2) respectively.

It is not difficult to see, given the definition of $d_k^{(r)}$ in (4.1), that equation (4.2) does converge to the correct value of the shortest path from the source to node $j$. Indeed, we can be assured that $d_k^{(r+1)} = d_k^{(r)}$ for all $j \in V_2$, where $d_k$ is the length of a shortest path from the source to node $j$.

It is important to notice that the computation may be terminated whenever $d_k^{(r+1)} = d_k^{(r)}$; a test which can be executed on the DAP extremely fast. This stopping criterion is usually reached long before $r = n_a$, and this is what makes the method so efficient on average.

**CASE 2:** $n^2 > \text{number of processors } p^2$

In this case we need to find a way to fit the problem into the DAP store. One of the most common mappings is called **slicing**. To demonstrate this, let us consider, for example, the task of dealing with a network having $n = 2p$ nodes (and a maximum of $4p^2$ arcs), where $p^2$ is the number of processors on the DAP. The strategy is to simply cut the length matrix $L = (l_{ij})$ into four blocks of DAP matrices, and store them in a matrix array as follows:

![Diagram of a DAP Matrix Array having 4 DAP frames]

The DAP declaration for $L$ is a real (or integer) matrix array ARC-LENGTH ($\mathbf{NB}$, $\mathbf{NB}$), where $\mathbf{NB} = [n/p]^4$ (for our assumption, $\mathbf{NB} = 2$). The first comma in the declaration, defines a frame of the DAP data holding $p^2$ values which can be processed simultaneously.

Note: The slicing mapping can be considered similar to the following Fortran declaration:

$$\text{REAL L}(64, 64, 2, 2)$$

or

$$\text{REAL L}(64, 64, 4)$$

when $p = 64$ say.

Let us now define $d_1^{(r)}$ and $d_2^{(r)}$ to be the corresponding $r^{th}$ approximation of the lengths of shortest paths from the source to the set of nodes in $V_1 = \{1, 2, ..., p\}$ and $V_2 = \{p+1, p+2, ..., 2p\}$ respectively. Then:

$$(d_1, d_2^{(r+1)} = (d_1, d_2^{(r)} \cdot 4, 2) \equiv \begin{bmatrix} L_{11} & L_{12} \\
L_{21} & L_{22} \end{bmatrix}$$

where $L_{ij}$ is the frame of DAP data stored in ARC-LENGTH ($\mathbf{NB}$), and $@$ is defined by:

$$d_1^{(r+1)} = \min (d_1^{(r)} @ L_{11}, d_2^{(r)} @ L_{21})$$

$$d_2^{(r+1)} = \min (d_1^{(r+1)} @ L_{12}, d_2^{(r)} @ L_{22})$$

(4.3)

*$[x]$ is the ceiling function; that is, $[x] = x$ if $x$ is integer

$$\text{int}(x+1) \text{ otherwise}$$

Here, we notice that once we have computed $d_1^{(r+1)}$, this was subsequently used in calculating $d_2^{(r+1)}$. This is because, to calculate $d_1^{(r+1)}$, one needs only the blocks $L_{11}$ and $L_{21}$ which represent the length of the arcs in $V_1 \times V_1$ and $V_2 \times V_1$ respectively. Similarly, to compute $d_2^{(r+1)}$, we only need the data in blocks $L_{22}$ and $L_{12}$.

Computationally, (4.3) was implemented more efficiently in the sense that the minimum of the two operands, which are DAP matrices, was carried out first (and this is very efficient on the DAP) and then the minimum across each column was computed. Also, there are two ways in storing the length of the paths on the DAP. One, as a vector array, say PATH-LENGTH ($\mathbf{NB}$) or as a matrix, say PATH-LENGTH ($\mathbf{NB}$). Storing it as a matrix means that we can treat all its components simultaneously and this becomes more efficient when $\mathbf{NB}$ is large.
The idea behind Algorithm SP is very close to the dynamic programming type algorithm attributed to Bellman and Ford [4, 13], having complexity of $O(n^2)$ on an n x n DAP as compared to the $O(n^2)$ of the serial algorithm.

We implemented Algorithm SP on the DAP 610 with 4096 processors, using the length-matrix representation to store the network. For dense and semi-dense networks, the algorithm is very efficient (see Table 4.1). However, for very sparse graphs, the results are not as encouraging as for dense and semi-dense networks. Note that the improvements suggested by Yen - see [32] - are not applicable in a parallel environment.

### 4.2. Computational Results

All of the test problems were generated in a random fashion. A random (connected) network with n nodes and m arcs were generated as follows:

First, a spanning tree rooted at the source node (node 1) is created by randomly joining a node in the current (non-spanning) tree to another node not in the tree yet. Then, the remaining arcs were added by joining two distinct nodes (not yet connected) chosen randomly from {1, 2, ..., n}, until a total of m arcs have been generated. We note that, the resulting network has no multiple arcs and loops.

The above procedure was used to generate random networks having from 20 to 640 nodes, with arc densities, defined as $\delta = m/(n^2 - n)$, ranging from 1.0 to 0.0025. The arc-lengths were randomly generated values from a uniform distribution on the interval [1,10000]. Table 4.1 contains the computation times of Algorithm SP. These timing become more attractive for more dense networks, confirming that the runtime of Algorithm SP is proportional to the maximum number of arcs in the shortest paths in the network.

The results given in Table 4.1 compare very favorably with those of serial algorithms as reported in the literature (See for example, Hulme and Wisniewski [20] and Dial et al. [8]).

It is worth mentioning that when we implemented a parallel version of Dijkstra algorithm [9], using both the length-matrix and arc-lists representations, no substantial speed up was obtained. This was due to the fact that such an algorithm will process every node sequentially and evaluate the successor nodes in parallel.

### Table 4.1: Computation Times for Algorithm SP in Milliseconds on the AMT DAP 610.

<table>
<thead>
<tr>
<th>$\delta$</th>
<th>1.0</th>
<th>0.9</th>
<th>0.8</th>
<th>0.7</th>
<th>0.6</th>
<th>0.5</th>
<th>0.4</th>
<th>0.3</th>
<th>0.2</th>
<th>0.1</th>
</tr>
</thead>
<tbody>
<tr>
<td>20</td>
<td>6.05</td>
<td>5.75</td>
<td>5.45</td>
<td>5.20</td>
<td>4.95</td>
<td>4.70</td>
<td>4.45</td>
<td>4.20</td>
<td>3.95</td>
<td>3.70</td>
</tr>
<tr>
<td>32</td>
<td>8.25</td>
<td>7.95</td>
<td>7.65</td>
<td>7.40</td>
<td>7.15</td>
<td>6.90</td>
<td>6.65</td>
<td>6.40</td>
<td>6.15</td>
<td>5.90</td>
</tr>
<tr>
<td>64</td>
<td>10.45</td>
<td>10.15</td>
<td>9.85</td>
<td>9.60</td>
<td>9.35</td>
<td>9.10</td>
<td>8.85</td>
<td>8.60</td>
<td>8.35</td>
<td>8.10</td>
</tr>
<tr>
<td>100</td>
<td>12.65</td>
<td>12.35</td>
<td>12.05</td>
<td>11.80</td>
<td>11.55</td>
<td>11.30</td>
<td>11.05</td>
<td>10.80</td>
<td>10.55</td>
<td>10.30</td>
</tr>
<tr>
<td>200</td>
<td>24.95</td>
<td>24.65</td>
<td>24.35</td>
<td>24.10</td>
<td>23.85</td>
<td>23.60</td>
<td>23.35</td>
<td>23.10</td>
<td>22.85</td>
<td>22.60</td>
</tr>
<tr>
<td>300</td>
<td>34.70</td>
<td>34.40</td>
<td>34.10</td>
<td>33.80</td>
<td>33.50</td>
<td>33.20</td>
<td>32.90</td>
<td>32.60</td>
<td>32.30</td>
<td>32.00</td>
</tr>
<tr>
<td>400</td>
<td>44.50</td>
<td>44.20</td>
<td>43.90</td>
<td>43.60</td>
<td>43.30</td>
<td>43.00</td>
<td>42.70</td>
<td>42.40</td>
<td>42.10</td>
<td>41.80</td>
</tr>
<tr>
<td>600</td>
<td>64.70</td>
<td>64.40</td>
<td>64.10</td>
<td>63.80</td>
<td>63.50</td>
<td>63.20</td>
<td>62.90</td>
<td>62.60</td>
<td>62.30</td>
<td>62.00</td>
</tr>
</tbody>
</table>

Upper rows are the times, under that are number of arcs in graph.

5. GRID NETWORKS

A grid network is one which can be embedded in a rectangular plane grid. Vertices are situated at lattice points and each vertex is directly connected to at most four neighbors in the N, S, E, W direction. Therefore, a pxq grid network has pq nodes and 4pq - 2p - 2q arcs. It is important to note, however, that the arc lengths are randomly generated. Thus, arc lengths are not necessarily symmetric and the triangle inequality may not hold. Interest in these networks stems from models of "perfect" cities and VLSI. Because the structure of grid networks closely resembles that of the DAP, by using the nearest neighbors connections, we are able to process these type of networks very rapidly.

5.1. Algorithm G

In this section we describe a parallel algorithm for solving the single-source problem in rectangular grid networks. There is no restriction on the arc lengths, they can be positive or negative. However, there are no negative cycles.
The algorithm is based on successive approximation, as described in Section 4.1 for Algorithm SP. The $(r+1)^{st}$ order approximation of shortest distance from the source, say node (1,1), to any node $(i,j)$ is only affected by the $r^{th}$ order approximation of shortest distance to its four neighbors, plus the arc lengths directed from these neighbors (N, S, E, W) nodes into node $(i,j)$, as shown in Figure 5.1.

![Figure 5.1](image)

To see how Algorithm G works more clearly, let us, without loss of generality, assume we have a pxq grid network and a pxq processing element DAP (e.g. in DAP 610, $p = q = 64$). Each processing element PE$_{ij}$ handles the following information:

(i) **The current $(r^{th}$ order) shortest distance from node (1,1) to node $(i,j)$** -- $d(r)(i,j)$.

(ii) **The length of the incoming-arc from North to node $(i,j)$** -- $h_N(i,j)$.

(iii) **The length of the incoming-arc from West to node $(i,j)$** -- $h_W(i,j)$.

(iv) **The length of the incoming-arc from South to node $(i,j)$** -- $h_S(i,j)$.

(v) **The length of the incoming-arc from East to node $(i,j)$** -- $h_E(i,j)$.

Now, if we only consider arcs coming from, say, north; that is, arcs of the form ((i-1,j), (i,j)) for all i, then we can compute the $(r+1)^{st}$ order approximation of shortest distances from node (1,1) to all nodes in parallel as follows:

$$d(r+1)(i,j) = \min\{d(r)(i,j), d(r)(i-1,j) + h_N(i,j)\}$$

for all i, j

(5.1)

In a similar fashion we consider in turn each of $h_W(i,j)$, $h_S(i,j)$ and $h_E(i,j)$. The advantage of this algorithm is that at the second, third and fourth turn of computing (5.1) with the appropriate arc lengths, we are (always) using the improved $r^{th}$ order approximation of the shortest distances. The algorithm terminates whenever $d(r+1) = d(1)$, a very fast check (one bit operation) on the DAP.

We like to point out that to do (5.1) on the DAP is extremely simple and fast, using the built-in shift functions which employ the nearest neighborhood connections. In fact, the piece of the Fortran-Plus to compute (5.1) is as below:

$$\text{WM} = \text{SHSP(D)} + \text{LN}$$

$$\text{D(WM, LT.D)} = \text{WM}$$

Where D and LN are DAP matrix objects, holding the pq values of $d(1)(i,j)$ and $h_N(i,j)$, respectively, for $i = 1, \ldots, p$ and $j = 1, \ldots, q$. WM is a temporary DAP matrix. SHSP is one of the built-in shift functions, which shifts every element in D south by one place using Planar geometry. This means values are dropped out from the south edge and zeros are fed in from the north edge. The expression (WM, LT.D) produces a logical mask, causing inhibit write.

5.2. Computational Results

It is clear that in a pxq grid network, node $(p,q)$ is reachable from the source node (node (1,1)) after performing two shifts, say in the direction of south and east, at least $(k-1)$ times, where $k = \max(p, q)$. So, it will be a waste of time to check for termination in Algorithm G prior to the $(k-1)^{th}$ iteration. Also, some of the four shifts tend to have no effect in earlier iterations.

Our experiments show that, it is better first to perform the two shifts for about $1/2k'$ iterations ($k' = \min(p, q)$) and then do the four shift for a further of $(k - 1/2k')$ iterations without any check for termination. After that, at each iteration the four shifts are performed with a check for termination.
Table 5.1 describes some of the grid network test problems with number of nodes ranging from 1024 to 4096. These problems are all randomly generated using a uniform distribution with arc lengths belonging to the interval [1,10000].

The results show that the parallel Algorithm G implemented on DAP 610 is much superior to the best serial code - see Dial et al. [8] - and on average is about 25 times faster than the best serial code ran on CDC 6600 machines.

Table 5.1: Computation Times for Algorithm G in Millisecs on the AMT DAP 610.

<table>
<thead>
<tr>
<th>Rectangularity</th>
<th>Number of Nodes</th>
<th>Number of Arcs</th>
<th>Run Times in Millisecs</th>
</tr>
</thead>
<tbody>
<tr>
<td>32 x 32</td>
<td>1024</td>
<td>3968</td>
<td>9.502</td>
</tr>
<tr>
<td>32 x 64</td>
<td>2048</td>
<td>8000</td>
<td>11.824</td>
</tr>
<tr>
<td>64 x 32</td>
<td>2048</td>
<td>8000</td>
<td>11.905</td>
</tr>
<tr>
<td>40 x 40</td>
<td>1600</td>
<td>6240</td>
<td>8.496</td>
</tr>
<tr>
<td>40 x 64</td>
<td>2560</td>
<td>10032</td>
<td>11.879</td>
</tr>
<tr>
<td>64 x 40</td>
<td>2560</td>
<td>10032</td>
<td>12.013</td>
</tr>
<tr>
<td>50 x 50</td>
<td>2500</td>
<td>9800</td>
<td>10.83</td>
</tr>
<tr>
<td>50 x 64</td>
<td>3200</td>
<td>12572</td>
<td>12.343</td>
</tr>
<tr>
<td>64 x 50</td>
<td>3200</td>
<td>12572</td>
<td>13.203</td>
</tr>
<tr>
<td>64 x 64</td>
<td>4096</td>
<td>16128</td>
<td>13.874</td>
</tr>
</tbody>
</table>

6. ALL-PAIRS SHORTEST PATH PROBLEM

In this section we consider the problem of finding the length of the shortest path between every pair of nodes in a directed network. This problem is often called "all-pairs problem". Instead of solving this problem by repeated application of, say, Algorithm SP for solving single-source problem, choosing n separate origins, we will develop a single integrated parallel procedure.

6.1. Algorithm AP

No restrictions are placed on the arc lengths except that there are no negative cycles. Let us suppose we have an n node network, stored in an n x n matrix L using the length-matrix representation, and an n x n processor DAP. Each processing element PEi keeps the value of Lij (for all i,j) and also the current rth length of the shortest path from node i to node j, dij(r).

The algorithm works by inserting one or more nodes into paths whenever it is advantageous to do so. After initializing D(1) = L, we construct a sequence of n matrices D(2), D(3), ..., D(n+1), where the (i,j)th entry of D(r) gives the length of a shortest path from node i to node j, subject to the condition that the intermediate nodes of the path are taken from the set of nodes {1,2,...,r-1}.

More formally, we need to compute the following:

\[ d_{ij}(r) = \begin{cases} l_{ij} & \text{for all } i,j \leq r \\ \min\{d_{ij}(r-1), d_{ik}(r-1) + d_{kj}(r-1)\} & \text{for all } i,j \leq r \text{ and for } r=1,2,...,n \end{cases} \]

Note that all the entries of D(r) are computed simultaneously on the DAP.

Thus, at iteration (r+1), the algorithm tries to improve paths by first checking if \( d_{ij}(r) + d_{ik}(r) < d_{kj}(r) \) and if so then the known path from node i to node j is replaced by the known path from node i to node r followed by the known path from node r to node j, see Figure 6.1.

Clearly, we must have \( d_{ij}(n+1) \) as the length of the shortest path from node i to node j.

These are the entries of D(n+1) which are computed simultaneously.

Algorithm AP runs in \( O(n^3) \) time as compared to \( O(n^3) \) of the serial algorithm of Floyd and Washall - see [11] and [31] - which is the basis of our algorithm.

It is worth giving the main piece of the Fortran Plus code which will execute Algorithm AP:

```
D = L
DO 10 R = 1,N
   WM = MATC(D(R)) + MATR(D(R))
   D(WM,LT,D) = WM
10 CONTINUE
```

For problems larger than DAP size, one can use the sliced mapping and the expansion is straightforward.

6.2. Computational Results

The results of running our implementation on DAP 610 for a number of test problems are summarized in Table 6.1. The test problems are randomly generated using a uniform distribution with arc lengths belonging to the interval [1,10000]. Note that the computation time is independent of sparsity of the network.

Table 6.1: Computation Times for Algorithm AP in Millisecs on the AMT DAP 610

<table>
<thead>
<tr>
<th>Number of Nodes</th>
<th>Run Time in Millisecs</th>
</tr>
</thead>
<tbody>
<tr>
<td>20</td>
<td>1.912</td>
</tr>
<tr>
<td>32</td>
<td>3.118</td>
</tr>
<tr>
<td>40</td>
<td>3.897</td>
</tr>
<tr>
<td>50</td>
<td>4.871</td>
</tr>
<tr>
<td>60</td>
<td>5.845</td>
</tr>
<tr>
<td>64</td>
<td>6.233</td>
</tr>
</tbody>
</table>
7. CONCLUSION

We have demonstrated that the DAP is a powerful machine for solving shortest path problems. The performance is at its peak for more dense networks, as all the processors are active. For extremely sparse networks, which usually have some type of structure, one needs to use that structure in order to achieve worthwhile speedup.

Contrary to the serial case, the Bellman and Ford type parallel algorithms is faster by an order of magnitude than the Dijkstra's algorithm; a phenomenon which usually is common with other algorithms.

The current work and other similar works on the DAP indicate that the idea of applying parallel programming techniques to solve combinatorial optimization problems is promising and deserves much more attention, and will likely lead to significant performance improvements.

REFERENCES


