An $O(n(n^2/p+\log p))$ Parallel Algorithm to Compute the All Pairs Shortest Paths and the Transitive Closure

MA JUN* and TADAO TAKAOKA*

Multiprocessors with shared memory structured as a complete binary tree are considered for use with a parallel algorithm to compute the all pairs shortest paths and the reflexive transitive closure in a weighted directed graph. The time complexity of the parallel algorithm is $O(n(n^2/p+\log p))$, where n is the number of vertices in the graph and $p(\leq n^2)$ is the number of processors used. The Ada language is used to implement the algorithm.

1. Basics

A weighted directed graph $G=(V, E, \text{COST})$ is an ordered triple of the set V of n vertices numbered from 0 to $n-1$, the set E of edges and a function COST that maps into real numbers. Edge (i, j) of E is said to be directed from vertex i to j. The function COST is usually given by a matrix COST $(0 \ldots n-1, 0 \ldots n-1)$, where COST (i, j) is the weight of the edge from vertex i to j. Here we let COST$(i, i)=0$. for $i=0, \ldots, n-1$, and COST $(i, j)=\infty$ if there is no edge from i to j. We define the cost of the shortest path from i to j in G as the minimum of the sums of the weights of the edges over the paths from i to j.

If A is the adjacency matrix of a graph G, the matrix A^* with the property that $A^*(i, j)=1$ if there is a path of length ≥ 0 from i to j and 0 otherwise is the reflexive transitive closure matrix of G.

Let R be an equivalence relation of a set A and, for each $a \in A$, let $[a]$, called an equivalence class of A, be the set of elements to which a is related:

$$[a]=\{x: (a, x) \in R\}$$

The collection of equivalence classes of A, denoted by A/R, forms a partition on A and is called the quotient of A by R.

2. Introduction

Recent progresses in hardware technology have caused the appearance of parallel computers with a large number of processors. Whether such machines are general or special purpose, a natural way is needed to map problems onto them. Only in this way will it be possible for applications to rapidly find their way into execution in this new computing environment.

In this paper we discuss two problems, the problem to compute shortest paths between all pairs of vertices in a directed weighted graph and the problem to calculate the reflexive transitive closure matrix of a directed graph.

Let us first discuss the problem to compute all pairs shortest paths. This problem is important in graph theory and has many practical applications. In a computer communication network a knowledge of the shortest paths between every two processing nodes is essential to determine dynamically the optimal feasible route from one processor to the other in order to minimize the communication delay. The input to this problem is a weighted directed graph $G=(V, E, \text{COST})$. The output is an array $A(i, j)$ containing the cost of the shortest path from i to j, for $i, j=0, \ldots, n-1$.

The parallel algorithms to compute all pairs shortest paths have been previously investigated [2], [10]. These algorithms were designed for VLSI implementation. Although the time complexity of these algorithms is very good, one with time $O(n \log n)$ and the other with time $O(n)$, n^2 processors are needed in each of the two algorithms. We can also use the parallel algorithms for solving the single source shortest path problem [4], [7] to compute all pairs shortest paths in $O(n \log n)$ time, but in this approach n^2 processors are necessary. The assumption that there are n^2 processors is too strong to implement these algorithms in practice and the parallel architectures are too special. So we want to find parallel algorithms which are independent of the size n of input graph and have high efficiency. We use the Ada language to implement our algorithms and measure time complexity assuming that a unit operation in Ada is done in $O(1)$ time, in very much the same way as sequential algorithms are analyzed using PASCAL. The architecture on which the algorithms are executed is a general purpose parallel machine described in the

*Department of Computer Science, Ibaraki University, Hitachi, Ibaraki 316, Japan.

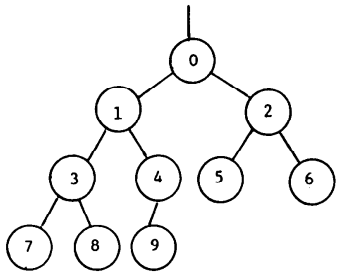

Fig. 1 A parallel architecture with a binary tree communication network (p=10).

following.

Shared memory multiprocessor systems are becoming increasingly popular as general purpose machines [3]. The advantages offered by shared memory multiprocessors include ease of programming and high performance which result from the tight coupling between processors and memories. In a shared memory multiprocessor system the processors share a number of global variables stored in the common memory. These variables also enable the processors to communicate efficiently through the shared memory. In the connection of processors the complete binary tree is considered as one of general networks [8]. A possible layout of such a binary processor tree with 10 processors is shown in Fig. 1. For it we have three assumptions:

1. Each processor in the binary processor tree can operate independently, has the same computing power and is labelled with a number shown in Fig. 1.
2. These processors are sharing a common memory.
3. Any processor in the binary tree can send and collect messages to and from its children processors (if it has) via the complete binary tree network.

On this parallel architecture let us restudy the problem of computing the all pairs shortest paths in a weighted directed graph. Because the multitasking constructs of the Ada programming language [1] now provide source-level facilities which can be used to implement algorithms incorporating parallelism and thus will provide a means of porting parallel algorithm over various parallel architectures, let us use it to discribe our parallel algorithm, and to execute our algorithms on the parallel machine mentioned above. The basic assumption on the architecture is that the same location of the shared memory can be accessed by any number of processors concurrently for reading and by one processor for writing at a time. Also, along the paths of the tree architecture, two child tasks can be generated in $O(1)$time. Let us first study a sequential algorithm to compute all pairs shortest paths.

3. The Sequential Algorithm

The sequential algorithm that we choose to parallelize is the best known all pairs shortest path algorithm [Floyd, 1962]. Let us restate it briefly. Floyd's sequential algorithm to solve the all pairs shortest path problem is as follows.

Define $A^k(i,j)$, $0\le i,j,k\le n-1$, to be the cost of the shortest path from i to j going through no intermediate vertex of index greater than k. Then $A^{n-1}(i,j)$ will be the cost of the shortest path from i to j in G since G contains no vertex with the index greater than $n-1$. $A^{-1}(i,j)$ is just COST(i,j). The basic idea of Floyd is to successively generate the matrices $A^0, A^1, \ldots, A^{n-1}$. If we have already generated A^{k-1}, then we can generate A^k by the formulas 3.1 or 3.2. We suppose G has no cycle with negative length containing vertex k here.

$$A^k(i,j)=\min\{A^{k-1}(i,j), A^{k-1}(i,k)+A^{k-1}(k,j)\}, \quad 0\le k\le n-1 \qquad 3.1$$

$$A^{-1}(i,j)=\text{COST}(i,j) \qquad 3.2$$

Now let us give Floyd's algorithm in an Ada procedure.

```
type adjacencymatrix is array (0 . . . n−1, 0 . . . n−1) of integer;
1  Procedure allpairs (cost: in adjacencymatrix;
                       A: out adjacencymatrix;
                       n: integer);
2    begin
3      copy A from COST;
4      for k in 0 . . . n−1 loop
5        for i in 0 . . . n−1 loop
6          for j in 0 . . . n−1 loop
7            A(i,j):=minimum
                     {A(i,j), A(i,k)+A(k,j)};
8          end loop;
9        end loop;
10     end loop;
11   end;
```

The time complexity of Floyd's algorithm is clearly $O(n^3)$.

4. The Parallel Algorithm

Let us develop a parallel algorithm based on Floyd's algorithm on the parallel machine shown in Fig. 1. We use $p(p\le n^2)$ to express the number of processors used in the computer system. In Floyd's algorithm it seems the operations between line 5 to line 9 can be implemented in parallel by multiprocessors. But if this is done by many processors working on the shared array A concurrently and asynchronously, A^k is not generated by the formula

$$A^k(i,j)=\min\{A^{k-1}(i,j), A^{k-1}(i,k)+A^{k-1}(k,j)\} \quad 0\le k\le n-1.$$

In fact, A^k is generated by one of the following four randomly chosen formulas

$$A^k(i,j)=\min\{A^{k-1}(i,j), A^{k-1}(i,k)+A^{k-1}(k,j)\} \quad 0\le k\le n-1, \quad 1.$$

$$A^k(i,j)=\min\{A^{k-1}(i,j),\ A^{k-1}(i,k)+A^k(k,j)\} \quad 0\le k\le n-1, \quad 2.$$

$$A^k(i,j)=\min\{A^{k-1}(i,j),\ A^k(i,k)+A^{k-1}(k,j)\} \quad 0\le k\le n-1, \quad 3.$$

$$A^k(i,j)=\min\{A^{k-1}(i,j),\ A^k(i,k)+A^k(k,j)\} \quad 0\le k\le n-1, \quad 4,$$

since $A^k(i,j)$ $(i,j=1,\ldots,n)$ are not updated all at once.

Now let us first prove an important fact that the generation A^k from A^{k-1} are independent of which operation of the above four is used.

Theorem 1. In the process of generating A^k from A^{k-1} of Floyd's algorithm, whichever operation of the above four is used the result is the same.

Proof. We can simply prove this theorem by proving $A^k(i,k)=A^{k-1}(i,k)$, and $A^k(k,j)=A^{k-1}(k,j)$.

Because we have supposed that there is no cycle with negative length at the beginning and $A^{-1}(k,k)=0$, for any k, $0\le k\le n-1$, $A^k(k,k)=0$ is not changed in the generation A^k from A^{k-1}. So

$$\begin{aligned}A^k(i,k)&=\min\{A^{k-1}(i,k),\ A^{k-1}(i,k)+A^{k-1}(k,k)\}\\&=\min\{A^{k-1}(i,k),\ A^{k-1}(i,k)+0\}\\&=A^{k-1}(i,k).\end{aligned}$$

For the same reason we have $A^k(k,j)=A^{k-1}(k,j)$.

According to theorem 1 in the generation of A^k from A^{k-1} we can calculate the elements of A in any order. So we want to find a mutual disjoint partition $\{A_i\}$ of the elements of A, $i=1,\ldots,p$ and find a method to let p processors work on the p subsets separately.

Let us map an array index (i,j) to a number x by the formula 4.1.

$$x=in+j. \quad 0\le i,j\le n-1. \qquad 4.1$$

It is clear that this is a one to one mapping from n^2 array indices to the set $S=\{0,\ldots,n^2-1\}$. It is also very easy to decompose a number belonging to S into an array index (i,j).

It is well known fact that $a\equiv b \pmod p$ is a equivalence relation on set $\{0,\ldots,n^2-1\}$ and $[0],\ldots,[p-1]$ are the quotient set of $\{0,\ldots,n^2-1\}$ [11]. Now we let the processor with label i, $(0\le i\le p-1)$, compute the elements of A whose array indices are mapped by formula 4.1 into $[i]$. Our algorithm has two steps. In step 1, the main program creates a process on the processor at the position of the root of the complete binary tree and sends 1 to the process. When a process on a processor in the binary tree receives a number x expressing indices (i,j) of array A, if it has a left child processor it creates a process on its left child processor and sends $2x$ to the process and if it has a right child processor it creates a process on its right child processor and sends $2x+1$ to the process. It is clear that the processor with label i $(0\le i\le p-1)$ receives a number of $i+1$. Hereafter we shall not distinguish between process and processor.

In step 2, the main program sends k to the process at the root of the binary tree. Each process waits to receive k from the binary tree. When a process receives k, if it has a left child process it sends k to its left child processor and if it has a right child process it sends k to its right child process. After that it calculates the elements of array A whose indices are mapped into $[x-1]$ by procedure "compute". This is like a process with label x undertake the additional calculations at the hypothetical nodes $x+p$, $x+2p,\ldots$. After n iterations of the loop of the main program like the sequential algorithm of Floyd, A^{n-1} is generated in A and $A(i,j)$ is the cost of the shortest path from vertex i to vertex j. Because the principle of our algorithm is the same as Floyd's, the correctness of our algorithm is obvious. Our algorithm is given as follows in an abstract form:

```
main program
begin
—STEP 1
1  start p processes in form of binary tree and send 1 to
   the process at the root of binary tree;
2  wait until the complete binary tree is formed and
   each processor receives its number as labels;
—STEP 2
3  for k in 0 . . . n−1 loop
4     send k to the process at the root of binary tree;
5     wait until k-th iteration is completed;
6  end loop;
end main;

process: (to be implemented by task in Ada)
  x, i, j, t: integer;
  bound: integer:=n^2;
procedure compute is
begin
1  t:=x−1
2  while t<bound loop
3     decompose t to an array index (i,j);
4     A(i,j):=min {A(i,j), A(i,k)+A(k,j)};
5     t:=t+p;
   end loop;
end
begin
—STEP 1
 1  wait to get a number x expressing an array index
    from binary tree network;
 2  if 2x+1≤p then {has two children}
 3     create a left child process and send 2x to it;
 4     create a right child process and send 2x+1 to it;
 5     wait until left and right subtrees are formed;
 6  elsif 2x≤p then {has left child only}
 7     create a left child process and send 2x to it;
 8     wait until left subtree is fromed;
 9  end if;
```

```
10  report to parent that this job is done;
—STEP 2
11  repeat loop n times
12    wait to get k in binary tree network;
13    if 2x+1≤p then {has two children}
14      send k to its left child process;
15      send k to its right child process;
16      compute;
17      wait until jobs are done in left and right sub-
        trees;
18    elsif 2x≤p then {has left child}
19      send k to its left child process;
20      compute;
21      wait until jobs are done in left subtree;
22    else compute;
23    end if;
24    report to parent that this job is done;
25  end loop;
end process;
```

Let us analyze the complexity of our algorithm.

Definition. The time complexity of a parallel algorithm is the time that passes from the first processor starts operating until the last one ends. Elementary operations are assumed to take one time unit, including process creation.

Theorem 2. On the parallel architecture shown in Fig. 1, if the number of processors used is $p(p \le n^2)$, the time complexity of our algorithm is $O(n(n^2/p+\log p))$.

Proof. Since the height of the binary tree is $\lceil \log p \rceil + 1$[6], the time for step 1 to create processes in the binary tree form is $O(\log p)$ in step 1.

Since the number of iterations of the loop between line 3 and line 6 in step 2 of main program is n, the number of iterations of the loop between line 11 and line 25 in the step 2 of processor is chosen to be n. It is clear that for any $i(0 \le i \le p-1)$, $|[i]| \le \lceil n^2/p \rceil$. So the number of iterations of the loop in procedure "compute" is less than or equal to $\lceil n^2/p \rceil$. Adding the time for passing k to the processes at the leaves of the binary tree, the time complexity of our algorithm is $O(n(n^2/p+\log p))$.

Corollary. In the multiprocessor system shown in Fig. 1, if the number of processors used is $O(n)$, the time complexity of our algorithm is $O(n^2)$. If the number of processors used is $O(n^2/\log n)$, the time complexity of our algorithm is $O(n \log n)$.

Note 1. In the analysis of the time complexity of our algorithm, we ignore the time of the input of COST, copying A from COST and the output of array A.

Note 2. The $O(\log p)$ time is necessary for synchronizing p processors for each iteration with each k.

The second problem to calculate the reflexive transitive closure matrix is that of determining for every pair of vertices i, j in G the existence of a path from i to j. For this problem an algorithm was given by Warshall [12]. The input to Warshall's algorithm is a boolean array A with the property $A(i,j)$=true if pair (i,j) is an edge of G and $A(i,i)$=true, for $i=0, \ldots, n-1$. The idea of Warshall was also to successively generate the matrices $A^0, A^1, \ldots, A^{n-1}$. If we change the line 7 of Floyd's algorithm as:

$$A^k(i,j) := A^{k-1}(i,j) \quad \text{or} \quad (A^{k-1}(i,k) \quad \text{and} \quad A^{k-1}(k,j)), \tag{4.2}$$

and A^{-1} is the input array, the changed Floyd's algorithm becomes Warshall's algorithm. Since the structure of Warshall's algorithm is much the same as Floyd's, we have a theorem similar to theorem 1 to parallelize Warshall's algorithm and the resulting parallel algorithm has the same time complexity of $O(n(n^2/p+\log p))$ with p processors. The details are left with the reader.

5. Conclusion

In this paper multiprocessors with shared memory structured as a complete binary tree are considered for use with a parallel algorithm to compute the all pairs shortest paths and the reflexive transitive closure of a weighted directed graph. Comparing our algorithms with the algorithms in [2], [10], the parallel architecture used in our algorithm is more general and our parallel algorithm is simpler conceptually. Because the number of processors used in our algorithm is independent of n, the size of input graph, our algorithm can be used to compute the all pairs shortest paths and reflexive transitive closure matrix on large graphs with an available number of processors.

There are very many parallel computational models proposed in the world. Depending on models on which a parallel algorithm is implemented, its time complexity varies. Based on theorem 1 it is easy to implement Floyd's algorithm on an SIMD parallel computer with CREW shared memory with n^3/p time where SIMD and CREW stand for "single instruction stream multiple data stream" and "concurrent read and exclusive write". However, most of these models are not available easily on real computers. On the other hand, Ada is easily available and becoming important as a language for describing parallel algorithms. Thus we adopted Ada as a language for our algorithm and also as an environment in which we measure time complexity. We stress that the tree topology is needed to realize the time complexity of our Ada program. Note that a similar technique of synchronization can be used for higher-dimensional arrays. In appendix A we give an implementation of our algorithm to solve the two problems over the binary tree parallel machine in Ada.

Acknowledgement

The authors wish to thank the anonymous refrees for their comments which enabled the improvement of this paper.

References

1. Ada programming language, Department of Defense, Washington, DC, ANSI/MIL-STD 1815 A, 1983.
2. AINHA, BHABANI P., BHATTACHARYA, BHARGAB B., GHOSE, SURANJAN and SRIMANI, PRADIP K. A Parallel Algorithm to Compute the Shortest Paths and Diameter of a Graph and its VLSI Implementation, *IEEE Trans. Comput.* C-35, 11 (1986) 1000-1004.
3. MEAD, C. and CONWAY, L. Introduction to VLSI systems, Addison-Wesley Publishing Company.
4. HOROWITZ, E. and SAHNI, S. Data Structures in Pascal, *Computer Science Press*, 1984.
5. FLOYD, R. W. Algorithm 97. Shortest Path, *CACM* 5, 6 (1962) 345.
6. CARLISLE, H., CRAWFORD, A. and SHEPPARD, S. Ada multitasking and the Single Source Shortest Path Problem, *Parallel Computing* 4 (1987), 75-91.
7. BENTLEY, J. L. and KUNG, H. T. A tree machine for searching problems, *Proc. International Conference on Parallel Processing* (1979).
8. MATETI, P. and PARALLEL, N. DEO, Algorithms for the Single Source Shortest Path Problem, *Computing* 29 (1982) 31-49.
9. RETTBERG, R. and THOMAS, R. Contention is no Obstacle to Shared Memory Multiprocessing, *Comm. ACM* 29 (1986) 1202-1212.
10. KUNG, SUN-YUAN, LO, SHENG-CHEN and LEWIS, P. S. Optimal Systolic Design for the Transitive Closure and the Shortest Path Problems, *IEEE Trans. Comput.* C-36, 5 (May 1987).
11. LIPSCHUTZ, S. Schaums's outline of Theory and problems of Discrete Mathematics, McGraw-Hill (1976).
12. WARSHALL, S. A Theorem on Boolean matrices, *J. ACM.* 9, 1 (1962) 11-12.

(Received November 27, 1987; revised August 6, 1988)

Appendix A

```
with text_io; use text_io;
procedure Floyd_Warshall is
  n, p: integer;—the size of graph and processors
  transitive: boolean;—transitive closure
  package int_io is new integer_io (integer); use
    int_io;
  package bool_io is new enumeration_io (boolean);
    use bool_io;
begin
  put ("input p and n please"); new_line;
  put ("if for transitive closure input true else false
    please");
  get (p); get (n); get (transitive);
declare
  A: array (0 . . . n-1, 0 . . . n-1) of integer;
  bound: integer:=n*n;
  procedure output is
  begin
    new_line;
    for i in 0 . . . n-1 loop
      for j in 0 . . . n-1 loop
        put (A(i, j));
      end loop;
      new_line;
    end loop;
  end;
  begin
    put ("input A please");
    for i in 0 . . . n-1 loop
      forj in 0 . . . n-1 loop
        get (A(i, j));
      end loop;
    end loop;
declare—declaration of process and related objects
  task type process is—specification of process
    entry receive (num: in integer);
    entry wait;
  end;
  type point is access process;
  subtype sub_process is process;
  process_point: point;—pointer to process
  task body process is—definition of process
  i, x, j, k, t, count: integer;
  left_point, right_point: point;
  procedure compute is
  begin
    t:=x-1;
    while t<bound loop
      j:=t mod n; i:=t/n;
      if transitive then
        if A(i, j)=0 then A(i, j):=A(i, k)*A(k, j);
      else
        if A(i, j)>A(i, k)+A(k, j) then
          A(i, j):=A(i, k)+A(k, j);
        end if;
      end if;
      t:=t+p;
    end loop;
  end compute;
  begin
  —step 1
    accept receive (num: in integer) do
      x:=num;
    end receive;
    if 2*x+1<=p then
      left_point:=new sub_process;
      left_point. receive (2*x);
      right_point:=new sub_process;
      right_point. receive (2*x+1);
      left_point. wait; right_point. wait;
    elsif 2*x<=p then
      left_point:=new sub_process;
      left_point.receive (2*x);
      left_point.wait;
    end if;
    accept wait;
  —step 2
    for count in 0 . . . n-1 loop
      accept receive (num: in integer) do
        k:=num;
      end receive;
      if 2*x+1<=p then
        left_point. receive (k);
```

```
        right_point. receive (k);
        compute;
        left_point. wait; right_point. wait;
      elsif 2*x<=p then
        left_point. receive (k);
        compute;
        left_point. wait;
      else compute;
      end if;
        accept wait;
      end loop;
    end process;
  begin—main program
    —create binary tree
    process_point:=new process;
    process_point. receive (1);
    process_point. wait;
      —iteration by k
      for k in 0 . . . n−1 loop
        put ("iteration"); put (k); new_line;
        process_point. receive (k);
        process_point. wait;
      end loop;
    end;
      if transitive then
        put ("THE REFLEXIVE TRANSITIVE
CLOSURE IS");
      else
        put ("THE COST OF SHORTEST PATH
FROM I TO J IS");
      end if;
      output;
    end;
  end Floyd_Warshall;
```