PARALLEL COMPLEXITY OF SINGLE SOURCE SHORTEST PATH ALGORITHMS

Mishra, P. K.
Department of Applied Mathematics
Birla Institute of Technology, Mesra
Ranchi-835215 (India)
&
Dept. of Electronics & Electrical Communication Engineering
Computer Vision Lab
Indian Institute of Technology, Kharagpur
West Bengal-721302
India
Email: pkmishra@ieee.org, pkmisra@gmail.com

Abstract
In this paper we have presented efficient parallel algorithms for all pair shortest paths problem and discussed the complexity of algorithms. The parallel complexity of algorithm achieves $O(\log^2 n)$ time when using $O(n)$ processors in CRCW PRAM model. We achieve overall complexity of SSSP is $O(n \log^2 n)$.

Keywords: All pair shortest paths, Complexity analysis, Parallel algorithms.

1. Introduction

Dijkstra's linear breadth first search algorithm is the best known sequential algorithm for finding the shortest path between any given pair of vertices. However, Dijkstra's algorithm cannot be parallelized easily to run in less time than linear time and has a limited speed. ASSSP problem is that of binding the minimum weight path to every vertex of a graph starting from a single source vertex. It is fundamental problem in graph theory with numerous applications. For example if the vertices of a graph represent cities and the edge weights represent the cost of transportation between the cities they connect the SSSP from a source to vertex A give the minimum transportation costs from city A to every other city.

Graphs are abstract data structure consisting of vertices (nodes) and edges that connect pairs of them. In a directed graph the connection is in one direction only. Adjacency matrices [1,4] and adjacency lists are two popular computer representation of graphs. For a graph of n vertices the adjacency matrix [2, 8] is an $n \times n$ array of bits with entry (i, j) set to i if vertices i and j are connected by an edge. Another $n \times n$ array stores the weights associated with the edges. The adjacency list for the same graph consists of n lists [3,5,6,7]. Listing the vertices each vertex is connected to in some say ascending order of their labels. The lengths of lists sum to $2\,e$, were e is no of vertex. The storage requirement of adjacency lists typically is less than that of adjacency matrices. From adjacency list, connectivity can be determined only in $O(n)$ time in the worst case [1,5] where as the degree of a vertex can be compute in $O(1)$ time, so adjacency list is preferred over adjacency matrix. We represent graphs on processor array using adjacency lists. The list of neighbors of each vertex is stored in PE (Processing Elements). To access a particular vertex, the list must be accessed undependably in each PE for instance searching a neighbor vertex and finding minimum cost edge searching and sorting should be done independently on the lists present on PE 's. So addressing memory helps in graph operation. In parallel algorithm two variable are used, waiting is used an array that keeps tracks of processors waiting for work halt is set to true when all the processors waiting and queue is

empty. The queue must be locked before enqueuing. Before a process compares a new distance to distance(v). Variable distance(v) should be locked. Finally if a process finds a queue empty, it sets waiting true, and checks other processors if finds waiting sets halt true. The queue should be locked while process checks other processes [8,1].

2. Sequential Single Source Shortest Path :

Again we present a sequential solution based on divide and conquer which we shall parallelize. The sequential algorithm we shall be denoting by SSSP*(G)*. Before describing the algorithm we introduce some notations. The source vertex is denoted by *s*. We shall be referring to the same divide and conquer tree [6] for this problem. The tree is defined in the same fashion as it was defined earlier. We shall be denoting its level by l, starting from zero at the topmost level. Informally the invariant of our divide and conquer is the maintenance of the single source shortest path information in each sub problem from those vertices acting as the source, which had been a separator vertex on the path starting form the parent of that sub problem to the root of the divide and conquer tree or if it is *s*. More precisely for we maintain the shortest path information from each source (defined below) to every vertex $v_n \in V_{i,j}$ where

$$v_m = \left\{\left(\bigcup_{l=1}^{i} S_{(i-l),f^{(i-l)}(j)}\right) \cup \{s\}\right\} \cap V_{i,j} \ , \qquad (1)$$

where the function f is defined as follows,

$$f^{(i-1)}(j) = \lceil j/2 \rceil; \quad \text{for } l > 1 \quad . \qquad (2)$$
$$f^{(i-1)}(j) = \lceil f^{(i-l+1)}(j)/2 \rceil$$

For a more pictorial description of what the algorithm is doing please refer to the correctness of the algorithm. In our algorithm we shall assume that every vertex is associated with a number, which we shall be denoting this as its mask. Masking operation on a vertex will imply setting this number to one which denotes the lowest level in the divide and conquer tree at which that vertex first became a separator vertex. It is introduced to help in marinating the invariant described above and its use will be clear as we go into the details of the algorithm. We may assume that unmasking operation is equivalent to setting the number of the vertex to a negative in tag. Initially all vertices are unmasked, besides *s* whose mask is set to zero. Now we give the details of the algorithm.

Algorithm *SSSP* (*G, l*)

Comments :

l stands for the depth of recursion or the level in our divide and conquer tree.

k is the maximum clique size.

l will be initialized to one while we invoke this procedure.

Step I. *If* $c \log n$ (c is any constant) *then* apply *AP* to get the transitive closure and *exit.*

Step 2. This step is exactly similar to *step I* of AP i.e., identify the good separator *S*. Mask all unmasks vertices of *S* with *l*. If any vertex of *S* is already masked they are left unchanged. Then build up two induced sub graphs $G_1 = (V_1 \cup S, E_1)$ and of

G where $G_2 = (V_2 \cup S, E_2)$ of G where V_1 and V_2 are two sets of vertices of the two components after the decomposition of G with S.

Step 3. Recursively solve SSSP *(G_1, l+1).*

Step 4. Recursively solve SSSP *(G_2, l+1).*

Step 5. This step is the merging step to maintain the invariant .

(a) Build up a distance matrix S where each entry

$$d(v_i, v_j) = \min\{sd_{G_1}(v_i, v_j), sd_{G_2}(v_i, v_j)\} \quad (3)$$

for all pairs $v_i, v_j \in S$.

To obtain where $sd_G(v_i, v_j)$ perform a transitive closure on that matrix.

(b) For every vertex $v \in V_1$ in G_1 find the shortest path to every separator vertex by the following operation

$$d(v,u) = \min_{t \in S}\{sd_{G_1}(v,t) + sd_{G_2}(t,u)\}. \quad (4)$$

In the above operation $sd_G(t, u)$ is already identified in *step 5*(*a*) and $sd_{G1}(v, t)$ was maintained as our invariant. The same operation is done on each vertex of V_2 in G_2 also.

(c) Unmask all vertices of S which has number l. For each of the masked vertices taken as the source compute the shortest distance to every other vertex. If a masked vertex $v_i \in S$ then the shortest distance from it to every other vertex is already computed in *step5*(*b*). If $v_i \in V_1$ is a masked vertex then the shortest distance of any vertex $v_j \in V_2$ can be computed as

$$sd_G(v_i, v_j) = \min_{u \in S}\{sd_G(v_i, u) + sd_G(u, v_j)\}. \quad (5)$$

Here $sd_G(v_i, u)$ and $sd_G(u, v_j)$ were already identified in *step 5(b).* If $v_j \in V_1$ is a masked vertex and is any other vertex then the shortest distance can be computed as

$$sd_G(v_i, v_j) = \min\{\min_{u \in S}\{sd_G(v_i, u) + sd_G(u, v_j)\}, sd_{G_1}(v_i, v_j)\}. \quad (6)$$

The same operation will be done on masked vertices of V_2 .

Note: From now on in subsequent, discussions the word node will be for the divide and conquer tree and the word vertex will be used for the graph *G*.

Lemma 1

The algorithm SSSP correctly determines the single source shortest distance from s to every other vertex.

Proof : Here we shall elaborate what the algorithm *SSSP* is doing. We shall argue on the fact that this divide and conquer maintains the invariant which is the maintenance of the shortest path information from those source vertices at a node of the divide and conquer tree which are present in that node and had been separator vertices at levels less than or equal to the current level. Let us consider a part of the divide and conquer tree starting from the root. The source vertex is *s* for which we want to build up the shortest path information. Now *B* is one of the child of *A* obtained through splitting steps as described in *step* 1 and 2. At the node *B* only *s* is masked with 0. All other vertices are unmasked. Then in *C*

vertices s_1 and s_3 are masked with 1 and *s* has retained its mask 0. Similarly at *D*, s_2 is masked with 1. At E, s_4 and s_6 is masked with 2 and s_3 retains its mask 1. At F, *s* is masked with 0, s_1 and s_3 are masked with 1 and s_5 is masked with 2. Now we assume that the invariant is preserved at *E* and *F*. This means at *E* we know the shortest distances from s_3, s_4, s_5, s_6 and s_7 to every other vertex in *E* through paths which uses vertices of *E* in *G*. Similarly at *F* we know the shortest distances of s_1 , s_3, s_5, s_7 , and *s* to every other vertex in *F*. To prove the correctness we have to show that this invariant will be preserved at *C* also.

Now since every separator vertex (here s_3, s_7) at *E* or *F* the shortest distance to every other vertex is known implies distance between them (i.e. here distance between s_3 and s_7) will be known. Thus we can clearly see that *step*5(*a*) correctly computes the all pair shortest distance among separator vertices at the node *C*. In our invariant we know that the shortest distance of every vertex in *E* and *F* from s_3 and s_7 is known through paths which are solely contained in sub graphs induced by vertices of *E* and *F* respectively in *F*. So after the computation of the all pair shortest distances in *step*5(*a*) among separator vertices it is quite obvious to see that *step*5(*b*) computes the shortest distance of every vertex to each of separator vertices which are here s_3 and s_7. Lastly in *step* 5(*c*) only s_7 is unmasked but s_3 retains its mask since it is masked with 1. Then the shortest distance of s_1 , s_4, s_5 , s_6 to every other vertices in *C* are determined since they are only masked vertices besides s_3 for which the shortest distance to every other vertices in *C* were already determined in *step*5(*b*). Thus we see that the invariant is preserved at the node *C* of the divide and conquer tree.

3. Parallel Complexity of Single Source Shortest Path :

Now we shall be showing how to parallelize *SSSP*. We shall be denoting it by *PSSSP*. For *PSSSP* we shall be using only $O(n)$ processors. Here follows the details of the algorithm.

Description of PSSSP (*G*) :

Here we shall be taking the same approach which we took while we parallelized AP. The building up of the elimination tree will be done only once. This would take $O(n)$ processors and $O(\log^2 n)$ time. The algorithm consists of two phases.

The first phase is the divide phase in which the divide and conquer tree displayed will be built up in the same fashion as described in PAP and will take $O(n)$ processors and $O(\log^2 n)$ time.

In the second phase at the maximum level of the divide and conquer tree we perform *step1*. Here we have at most $O\left(\frac{n}{\log n}\right)$ sub problems each of size $O(\log n)$. We solve each of this sub problem in parallel allocating $O(\log n)$ processors to each. So according to *Theorem*1, this level can be completed in $O(\log n)$ time and with $O(n)$ processors. Now in general for level i where each sub problem corresponding to is of size $O\left(\frac{n}{2^i}\right)$ and we have $O\left(\frac{n}{2^i}\right)$ processors available for it. Clearly *step*5(*a*) will take $O(1)$ time since $\left|S_{i,j}\right| \le k$

where k is a constant. *Step* 5 (b) will take $O(1)$ time by assigning one processor to each vertex and parallelly executing this step for each of them. In $G_{i,j}$ number of masked vertices can be at most k_{i+1} (including s). So it will take $O(i)$ time to execute *step5*(c) for $G_{i,j}$. This follows from the observation that for each of the masked vertices of $G_{i,j}$ to build up single source shortest path information to all other vertices of $G_{i,j}$ as described in *step5*(c) amounts to $O\left(\frac{k_{in}}{2^i}\right)$ work. Hence applying *Theorem* 1 knowing the fact that total number of processors available for $G_{i,j}$ is $O\left(\frac{n}{2^i}\right)$ we establish our claim.

Since all sub problems at level i are executed in parallel, the time required to execute level i is also $O(i)$ with $O(n)$ processors. The overall time complexity of the entire problem is

$$\sum_{i=1}^{\lceil \log n \rceil} O(i) \in O(\log^2 n). \qquad (7)$$

Thus we have established the following Theorem .

Theorem 1

Single source shortest path problem on chordal graphs with maximum clique size bounded by constant can be solved in parallel with $O(n)$ processors in $O(\log^2 n)$ time on CRCW PRAM model of computation.

7. Conclusion

In this paper we have presented efficient parallel algorithms and complexity for shortest path problems in graphs. The algorithm for the single source problem, achieves near optimal processor time product in contrast to general graphs where we face the transitive closure bottleneck.

References:

1. Akl. S. G., *The Design and Analysis of Parallel Algorithms,* Prentice Hall, New Jersey, 1989.
2. Berkman, O. and Vishkin, U., Finding level-ancestors in trees, Tech. Report. UMIACS -TR – 91-9, University of Maryland, 1991.
3. Bertossi, A. A and Bonuccelli, M.A., Some parallel algorithms on interval graph, Discr. Appl. Math. 16 (1987) 101-111.
4. Booth, S. K. and Colbourn, C. J., Problems polynomially equivalent to graph isomorphism, Technical Report CS-77-04, Computer Science Dept. Univ. of Waterloo, Canada, 1979.
5. Chen, C.Y. and Das, S. K., Parallel algorithms for level order traversals of general trees, Journal of Combinatorics, Vol.14. No. 2-3 1989, pp. 135- 162.
6. Chen, L., Parallel complexity of discrete problems, Ph D thesis, Ohio State Univ. 1990.
7. Hoffmann, C. M., Group-theoretic algorithms and graph isomorphism, Vol. 136. Lecture Notes in Computer Science, Springer –Verlag.
8. Horowitz, E. and Sahni, S., Fundamentals of Computer Algorithms, Computer Science Press, 1978.