Parallel Computing 13 (1990) 245–251
North-Holland

Short Communication

Graph search algorithms and maximum bipartite matching algorithm on the hypercube network model *

Jang-Ping SHEU
Department of Electrical Engineering, National Central University, Chungli, Taiwan, Republic of China

Nan-Ling KUO
Institute of Information Engineering, Tatung Institute of Technology, Taipei, Taiwan, Republic of China

Gen-Huey CHEN
Department of Computer Science and Information Engineering, National Taiwan University, Taipei, Taiwan, Republic of China

Received October 1988
Revised May 1989

Abstract. In this paper, we propose parallel algorithms for breadth-first search and depth-first search on the hypercube network model. In addition, a parallel algorithm based on the same model for finding maximum matching in bipartite graphs is proposed.

Keywords. Maximum matching, breadth-first search, depth-first search, bipartite graph, hypercube, parallel algorithm.

1. Introduction

The parallel algorithms for graph search techniques such as breadth-first search (BFS) and depth-first search (DFS) have been studied by several researchers [2,3,5,7]. In this paper, two parallel search algorithms, one for BFS and the other for DFS, based on the hypercube network model are proposed. In addition, we devise a parallel algorithm to find the maximum matching in bipartite graphs. Both search techniques have the lower bound of time complexity $\Omega(n^2)$ for sequential algorithms in the worst case, and the fast known sequential maximum bipartite matching algorithm needs $O(n^{2.5})$ time [4], where n is the number of vertices in the graph. Our parallel algorithms take $O(n \log n)$ time for the two graph search techniques and $O(n^{1.5} \log n)$

* This work was partially supported by the National Science Council under Grant NSC76-0408-E008-06.

for the maximum bipartite matching problem. Since the number of processors used is only $n/\log n$, all our algorithms achieve the optimal speedup.

In this paper, processors in hypercubes operate asynchronously, the only means of communication and synchronization is message passing. In our algorithm, some of the processors perform just a few simple computations in parallel after receiving a message sent from anyone of their neighbors. Thus, it is sufficient to account only for the number of messages passed between near neighbors as the measure of time complexity.

2. Parallel BFS and DFS algorithms

In this section, we consider the search problems of a connected undirected graph $G = (V, E)$ on the hypercube network model. By adopting the 'near neighbor' method of the Prim–Dijkstra algorithm for finding the minimum spanning tree [1,6], we develop a parallel BFS algorithm and a parallel DFS one.

2.1. Breadth-first search algorithm

Given a root r, say vertex 0 for simplicity, we want to find a BFS tree rooted at r and determine the order, denoted as BFS_i, in which vertex i is visited by BFS. Without loss of generality, let node (processor) i store a vertex labelled i. Additionally, the adjacent vector $ADJACENCY_i$ of vertex i is also stored in node i. If (i, j) is an edge in E, $ADJACENCY_i[j] = 1$; otherwise, $ADJACENCY_i[j] = \infty$. Associated with each vertex i are two labels, $DIST_i$ and $NEAR_i$, which hold the length of the shortest path from the root vertex to vertex i and the nearest neighbor of vertex i which has been brought in the BFS tree, respectively.

For the sake of convenience, vertex 0 is selected as the first vertex of the partial-formed spanning tree T. Within each iteration, an isolated vertex v is found and sent to node 0 if its $DIST_v$ has the minimum value. Ties are resolved in favor of the vertex with the smallest vertex number. After receiving such vertex v, node 0 joins the edge $(v, NEAR_v)$ to T, and sends not only the vertex number and $DIST_v$ value of vertex v but also the current BFS order to all other nodes to adjust their $DIST_i$, BFS_i, and $NEAR_i$ values if required, where $i \neq v$. Only when the first one of its neighbors, say vertex j, is selected to be included in T, an isolated vertex i changes the values of its $DIST_i$ and $NEAR_i$, to $DIST_j + 1$ and j, respectively. After being joined to T, the BFS order of vertex i is the 'current BFS order' receiving from node 0. Repeat this process $n - 1$ times, and we finish the BFS algorithm. The formal algorithm is given in algorithm PBFS_CUBE.

Algorithm PBFS_CUBE

/* For each hypercube node i do in parallel. */

Step 1: Initially, node 0 set $NEAR_0 = \infty$, $DIST_0 = 0$, BFS tree $T = \phi$, and BFS order = 1, while other nodes set $NEAR_i$ = vertex 0, $DIST_i = ADJACENCY_i[0]$, and $BFS_i = 0$.

Step 2: Find a vertex v with the minimum $DIST_v$ value ($\neq \infty$) among all remaining isolated vertices; transmit it to node 0.

Step 3: /* For node 0 only. */
Join $(v, NEAR_v)$ to T, increase BFS order by 1, and send v, current BFS order, and $DIST_v$ to all other nodes.

Step 4: /* Action executed in node 1 to node $n - 1$. */
If resident vertex $i = v$, then set BFS_i = current BFS order else if $ADJACENCY_i[v] = 1$ and $DIST_i = \infty$, then set $NEAR_i = v$ and $DIST_i = DIST_v + 1$.

Step 5: If $|T| \leqslant n - 1$ goto Step 2.

In Algorithm PBFS_CUBE, the most time consuming operations are Step 2 and Step 3 within one iteration. At Step 2, all processors cooperate in finding an extremum among n values each of which is stored in a different processor. This step can be implemented by applying the recursive operations for finding extrema of Saad's algorithm [8], which takes O(log n) time. On the other hand, the operation of broadcasting a message at Step 3 is done by using Saad's Hypercube Broadcast algorithm in O(log n) time [9]. There are $n-1$ iterations. Thus, the P[illegible]_CUBE algorithm requires O(n log n) time.

Let us consider the situation when there are only $p < n$ processors available. In such a situation, n/p vertices with their adjacent vectors are assigned to a single processor and each processor simulates the processes of n/p processors in the original algorithm PBFS_CUBE. To find out a vertex i with the minimum DIST_i value per iteration, each processor requires O(n/p) time to locally compute the minimal value in its resident n/p vertices in parallel, and then it takes O(log p) time to find the minimum value among these p minimal values. A broadcasting operation also needs O(log p) time on a p-processor hypercube. Thus, the total time complexity $T(n)$ becomes

$$T(n) = (n-1)\text{O}(n/p + \log p + \log p)$$
$$= \text{O}(n^2/p + n \log p).$$

If $p = n/\log n$, then

$$T(n) = \text{O}(n \log n).$$

We see that the time complexity of PBFS_CUBE remains O(n log n) even when using $n/\log n$ processors.

2.2. Depth-first search algorithm

In the following, a parallel algorithm for determining the DFS tree of $G = (V, E)$ and deciding the order, denoted as DFS_i, in which we scan vertex i by DFS is presented. The DFS tree is a spanning tree in which the vertices are visited on a last-reached first-scanned basis. That is, we always scan the most recently reached vertex i by searching for an unvisited vertex j adjacent to i. If no such j is available, the search returns to vertex u, from which i is reached. Because of the 'last-reached first-scanned' property of the DFS problem, a work stack named WSTACK is used in every node to hold the vertices during the searching process. Whenever a vertex is visited, it is pushed into WSTACK of every node. A vertex which has not been searched yet can be searched if it is adjacent to the top vertex in WSTACK. There may be many isolated vertices satisfying this condition, but the one with the smallest vertex number is selected. Every node pops its WSTACK when it can find no vertex to be search next. Node 0 will broadcast a 'POP' message to all other nodes if it does not receive a vertex to be joined to the partial-formed DFS tree after it waited O(log n) time. Otherwise, it broadcasts the selected vertex and the current DFS order. The formal algorithm is shown in algorithm PDFS_CUBE.

Algorithm PDFS_CUBE

/* For each hypercube node i do in parallel. */

Step 1: Initially, node 0 set $\text{NEAR}_0 = \infty$, DFS tree $T = \phi$, and DFS order = 1. While other nodes set $\text{DFS}_i = 0$ and if $\text{ADJACENCY}_i[0] = 1$, then set $\text{NEAR}_i = 0$ else set $\text{NEAR}_i = \infty$. All nodes push 0 into the working stack WSTACK.

Step 2: Find a vertex v adjacent to the top element in WSTACK, i.e., NEAR_v = ID number of the top vertex in WSTACK, among all isolated vertices and transmit it to node 0.

Step 3: /* For node 0 only. */
3.1 A vertex v is not received after waiting O(log n) time: Pop top element off WSTACK and broadcast ("POP").
3.2 A vertex v is received in time: Push v into WSTACK, joint (v, NEAR$_v$) to T, increase DFS order by 1, and broadcast message (v, current DFS order, $|T|$).
Step 4: After receiving the broadcast message, each node $i \neq 0$ executes one of the following operations:
4.1 If receive ("POP"), then pop top element off WSTACK.
4.2 Otherwise, push v into WSTACK, and if $i = v$, then set DFS$_i$ = current DFS order else if ADJACENCY$_i[v] = 1$ and DFS$_i = 0$, then set NEAR$_i = v$.
Step 5: If $|T| \leqslant n - 1$, then goto step 2.

The most time-consuming operations in Algorithm PDFS_CUBE are Step 2 and Step 3 within one iteration. Step 2 takes O(log n) time to find the isolated vertex which is adjacent to the top element of WSTACK. At Step 3, an "POP" message occurs when there is a need for backtracking. Therefore, the complexity of Algorithm PDFS_CUBE is also O(n log n) time using n/log n processors.

3. Parallel maximum bipartite matching algorithm

Let $G = (X \cup Y, E)$ be a bipartite graph, where X, Y are two disjoint vertex set and E is the set of edges, each of which has one end in X and the other in Y. The number of vertices in G is n. That is $|X| + |Y| = n$. We refer to the vertices in X as boys and those in Y as girls in the following discussion. A matching M in G is a subset of E such that no two edges in M share a common vertex. A matching with the maximum cardinality is called maximum matching. A vertex v is matched relative to a matching M if it is incident to one edge in M and is unmatched otherwise. An alternating path with respect to M is a simple path whose edges are alternatively in M and $E - M$, and an augmenting path is defined as an alternating path between two unmatched vertices. An augmenting path P is called a shortest path relative to M if P has the smallest length among all the augmenting paths relative to M. Whenever we find an augmenting path P relative to M, then by interchanging matched edges with unmatched ones, we can get a new matching M' with one more edge than M. Hopcroft and Karp [4] showed that if the matching-augmentation is done through a maximal set of vertex-disjoint shortest augmenting paths, a maximum bipartite matching can be obtained in O($n^{2.5}$) time. Their algorithm is outlined as follows.

Algorithm Maximum_Matching
Initial: Start with a null matching M.
Phase 1: Construct a layered graph G^* relative to M.
Phase 2: Find a maximal set of paths $S = \{P_1, P_2, \ldots, P_t\}$ in G^* with the following two properties:
1. Paths P_i, $1 \leqslant i \leqslant t$, are all the shortest augmenting paths relative to M.
2. Any two paths P_i and P_j, $i \neq j$, are vertex-disjoint. If $S \neq \phi$, then set $M = M \oplus P_1 \oplus P_2 \oplus \cdots \oplus P_t$ and go to Phase 1; otherwise, halt.

Hopcroft and Karp claimed that the shortest augmenting paths in G relative to M were in one-to-one correspondence with the directed paths in G^*, which started from unmatched boys and ended at unmatched girls. In the following, we show how to implement algorithm Maximum_Matching on the hypercube network model.

With each vertex, there exist two adjacent vectors, R_ADJACENCY and L_ADJACENCY. Consider an edge (b_i, g_j) in E, where $b_i \in X$ and $g_j \in Y$. If this edge is matched, then L_ADJACENCY[j] of b_i and R_ADJACENCY[i] of g_j are set to be 1; otherwise, R_ADJACENCY[j] of b_i and L_ADJACENCY[i] of g_j are set to be 1. All other entries of both vectors of b_i and g_j are set to 0. The sizes of the two adjacent vectors of $b_i \in X$ are equal to the number of vertices in Y, and those of $g_j \in Y$ are equal to the number of vertices in X. Assume that each vertex with its own adjacent vectors is initially assigned to a different processor. Every node stores the vertex number of the assigned vertex in VERTEX, and there exists an extra indication bit in VERTEX to point out whether the resident vertex is matched or not.

Phase 1 can be parallelized by applying the parallel BFS algorithm in Section 2. At the beginning, any vertex i is put in the first layer of G^* and initialized with $\text{DIST}_i = 1$ and $\text{BFS}_i = 0$ if it is an unmatched boy; otherwise, it is set with $\text{DIST}_i = \infty$ and $\text{BFS}_i = 0$. During the construction of G^*, vertex i is said to be in layer j of G^* if its $\text{DIST}_i = j$. Each node i runs in cooperation with other nodes to find the DIST_i value of its resident vertex i by applying the BFS method. Note that the adjacent vector used in this phase is the L_ADJACENCY of every vertex. An extra flag FIND is used in node 0 to denote whether an unmatched girl is reached. If an unmatched girl is traversed and sent to node 0, node 0 set its flag FIND to be TRUE and sends the relative BFS order to the unmatched girl node. But any other nodes do not alter the states of their labels. When no vertex is available to be traversed during the execution of PBFS_CUBE, node 0 issues a message to all other nodes to inform them of entering Phase 2 in case that its flag FIND is TRUE. It means that the layered graph G^* is already constructed. However, if FIND is still FALSE, node 0 halts all processors. Since there is no unmatched girl or unmatched boy, we cannot augment the matching M. Algorithm G*Constructing is to implement this phase.

Algorithm G*constructing /* Applying BFS method to generate a layered graph G^*. */
/* For each processor i do in parallel. */

Step 1: If resident vertex = unmatched boy, then set $\text{DIST}_i = 1$ and $\text{BFS}_i = 0$; otherwise, set $\text{DIST}_i = \infty$ and $\text{BFS}_i = 0$.

Step 2: Find a vertex v with the minimum DIST_v value among all isolated vertices, and transmit vertex v to node 0.

Step 3: /* Action for node 0 only. */
Node 0 executes one of the following operations:

3.1 A vertex v is not received after waiting O(log n) time:
If FIND = TRUE, then broadcast message ("enter Phase 2") else halt all processors.

3.2 A vertex v is received in time: If v is an unmatched girl, then set FIND = TRUE, send BFS order increased by 1 to node v, and goto Step 2. Otherwise, increase BFS order by 1, broadcast message (v, v's corresponding DIST_v value, BFS order), and goto Step 2.

Step 4: /* Action executed in node 1 to node $n-1$. */
After receiving message from node 0, each node $i \neq 0$ executes one of the following operations:

4.1 If receive message ("enter Phase 2"), then enter Phase 2.

4.2 If resident vertex $i = v$, then BFS_i = current BFS order and goto Step 2.

4.3 If resident vertex $i \neq v$ and $\text{L\_ADJACENCY}_i[v] = 1$ and $\text{DIST}_i = \infty$, then $\text{DIST}_i = \text{DIST}_v + 1$ and goto Step 2.

The time needed to parallelize Phase 1 is dependent on the parallel BFS algorithm used, so it requires O(n log n) time using $n/\log n$ processors. As for Phase 2, using the parallel DFS algorithm described in the last section, we build a maximal set of vertex-disjoint augmenting

paths from G^* in parallel. After finding such set of paths, we go to augment M. The formal algorithm to implement Phase 2 is shown in Algorithm Max_Path_Set. Assume that there exists a pseudo vertex named S with an vertex number of $n+1$. Initially, every node put number $n+1$ into its WSTACK. If its resident vertex is an unmatched girl in the last layer of G^*, then node i set its NEAR_i to the vertex number of vertex S; otherwise, set its NEAR_i equal to ∞. All nodes whose BFS order values are not 0 run Algorithm Max_Path_Set concurrently and cooperatively. In Algorithm Max_Path_Set, the R_ADJACENCY of each vertex is used at its adjacent vector. Assume that all the final results are recorded in node 0 to augment the matching M. When a vertex v is found and sent to node 0, node 0 adds it to an array $\mathscr{P}$ according to the following rules:

(1) v is an unmatched girl: add it to $\mathscr{P}$ initialized to be empty.

(2) v is an unmatched boy: if $\mathscr{P}$ is empty, then discard it; otherwise, after adding it to $\mathscr{P}$, execute $M = M \oplus \mathscr{P}$.

(3) v is a matched boy or girl: if $\mathscr{P}$ is empty, then discard it; otherwise, add it to $\mathscr{P}$.

If the vertex traversed is an unmatched boy, node 0 applies the operations stated in rule (2), clears $\mathscr{P}$ to be empty to store another path, and then broadcasts a message to let all nodes whose resident vertices are not scanned yet return to their initial states. When node 0 does not accept a vertex available to be searched after waiting $O(\log n)$ time, it broadcasts a 'pop' message as did in Algorithm PDFS_CUBE. If the current top element of its WSTACK is $n+1$, it knows that all augmenting paths have been found. It broadcasts all matched edges to every node to adjust its adjacent vectors and the state of its resident vertex as the matching M has been augmented. Then each node goes back to Phase 1, and repeats the process of augmenting M. However, if the matching M has not been changed, the whole process is stopped and M is the maximum matching we want. The formal algorithm is shown as follows.

Algorithm Max_Path_Set

/* For all nodes whose resident vertices are in G^*, i.e., those nodes whose BFS order values are not 0, do in parallel. */

Step 1: Push $n+1$ into WSTACK. If resident vertex i = unmatched girl, then set $\text{NEAR}_i = n+1$; otherwise, set $\text{NEAR}_i = \infty$.

Step 2: Find a vertex v adjacent to top element in WSTACK of all isolated vertices and transmit it to node 0.

Step 3: /* Action for node 0 only. */

Node 0 executes one of the two operations:

3.1 A vertex v is not received after waiting $O(\log n)$ time: Pop top vertex off WSTACK and broadcast ("POP") in case of WSTACK $\neq n+1$. Otherwise, if M has been augmented, then broadcast (matched edges) else stop the whole process of finding maximum bipartite matching.

3.2 A vertex v is received in time: Push v into WSTACK, add v to $\mathscr{P}$ according to the three rules stated above, increases DFS order by 1, and then broadcast (v, current DFS order).

Step 4: /* Action executed in node 1 to node $n-1$. */

After receiving the message from node 0, each node $i \neq 0$ does one of the following operations:

4.1 If receive ("POP"), then pop the top element off WSTACK and goto Step 2.

4.2 If receive the matched edges, then adjust its adjacent vectors and return to execute algorithm G*Constructing.

4.3 If v received is an unmatched boy and $\text{DFS}_i = 0$, then go to Step 1.

4.4 Push v received into WSTACK, and if resident vertex $i = v$, then set DFS_i = current DFS order else if $\text{R\_ADJACENCY}_i[v] = 1$ and $\text{DFS}_i = 0$, then $\text{NEAR}_i = v$. And goto Step 2.

There are two main operations in Algorithm Max_Path_Set: DFS, in parallel, and the action of broadcasting all matched edges. We know that Algorithm PDFS_CUBE takes $O(n \log n)$ time using $n/\log n$ processors. On the other hand, there are at most $n/2$ matched edges in an n-vertex graph, so, in the worst case, the broadcasting action takes $O(n \log p)$ time on a p-processor hypercube. Thus, it needs $O(n \log n)$ time to perform Phase 2 while using $n/\log n$ processors.

As we know, there are at most $O(n^{0.5})$ iterations in Algorithm Maximum_Matching. Within each iteration, it contains two phases of operation. These two phases both take $O(n \log n)$ time using $n/\log n$ processors on the hypercube network model by applying the parallel search algorithms in Section 2. Thus, it takes $O(n \log n)$ time using $n/\log n$ processors per iteration. Therefore, the time complexity of Algorithm Maximum_Matching on the hypercube network model is $O(n^{1.5} \log n)$ time using $n/\log n$ processors. Note that, we are not counting the time it would take to initially load the adjacent vectors of all vertices. It would need $O(m)$ time, where m is the number of edges in a graph, to download the adjacent vectors to the processors in the hypercube.

References

[1] E.W. Dijkstra, A note on two problems in connection with graphs, *Numer. Math.* **1** (1959) 269–271.

[2] R.K. Ghosh and G.P. Bhattacharjee, A parallel search algorithm for directed acyclic graphs, *Bit* **24** (1984) 134–150.

[3] R.K. Ghosh and G.P. Bhattacharjee, Parallel breadth first search algorithms for trees and graphs, *Internat. J. Comput. Math.* **15** (1984) 255–268.

[4] J.E. Hopcroft and R.M. Karp, An $O(n^{2.5})$ algorithm for maximum matching in bipartite graphs, *SIAM J. Comput.* **2** (4) (1973) 354–364.

[5] T. Kim and K.-Y. Chwa, Parallel algorithms for a depth first search and a breadth first search, *Internat. J. Comput. Math.* **19** (1986) 39–54.

[6] R.C. Prim, Shortest connection networks and some generalizations, *Bell System Tech. J.* **36** (1957)

[7] E. Reghbati and D.G. Corneil, Parallel computations in graph theory, *SIAM J. Comput.* **2** (2) (1978) 230–237.

[8] Y. Saad and M.H. Schultz, Topological properties of hypercubes, Research Report YALEU/DCS/RR-389 Yale University, 1985.

[9] Y. Saad and M.H. Schultz, Data communication in hypercubes, Research Report YALEU/DCS/RR-428 Yale University, 1985.