CONGRESSUS NUMERANTIUM

VOLUME 47

DECEMBER, 1985

WINNIPEG, CANADA

A FAST PARALLEL

CLOSURE ALGORITHM

Tom Altman

Department of Computer Science
University of Kentucky
Lexington, Ky. 40506

In this paper, we present parallel algorithms that compute the transitive closure of a directed graph and detect nontrivial cycles in parallel time O(log n). Both algorithms require n^3 processors, where n is the number of nodes in the directed graph. We also show that the detection of k-cycles (cycles of length $1 \leq k \leq n$) can be achieved in parallel time O(log k) using n^3 processors.

Keywords: transitive closure, cycle detection, parallel computation.

1. Introduction

Our model of parallel computation is somewhat similar to that used by other researchers (e.g., see 4). For unbounded parallelism, the model consists of sufficiently many numbered processors with unrestricted access to global memory. The processors (SIMD) work together synchronously and communicate via common random access memory in which both read and write conflicts are allowed. In case of a write conflict, it is assumed that the lowest numbered processor succeeds, although for the algorithms described below, that condition is not a necessary one. One reason for favoring these conventions is that the circuitry needed to implement them seems not to be substantially more complicated than that which disallows read and write conflicts (see 2). In other words, if one can build a machine where a datum can flow simultaneously to an arbitrary number of (reading) processors in one time step, then one should also be able to reverse the flow (i.e., allow for a simultaneous write operation) from processors to memory.

Each processor is capable of performing a Boolean operation or a comparison in one time step and the communication delays are ignored. The parallel time complexity of a computation in our model is the least number of steps needed by the processor that finishes last.

Congressus Numerantium, **47** (1985), pp.29-32

2. Transitive Closure

Let G be a directed graph of n nodes represented by a Boolean adjacency matrix A. First, save the major diagonal of A in a vector s. Set the elements of the major diagonal of A to be all 1's and call the resulting matrix A^1. Next observe that, given n^3 processors, we can multiply two boolean matrices $D = BC$ in parallel time O(1).

This can be accomplished by computing each element $d_{i,j}$, of the resulting matrix D, independently in time O(1) with only n processors. Processor k, (of the n processors labeled 1,...,n, which are assigned to compute some element $d_{i,j}$), reads in the k-th position of the row vector i' of B and the k-th position of the column vector j' of C. If both are 1, then processor k writes a 1 in $d_{i,j}$. Because write conflicts are allowed (and once set to 1, the entry $d_{i,j}$ can never be reset to 0) the above computation of an inner product of two boolean vectors has parallel time complexity O(1). Since there are n^2 elements in D, the number of processors required to perform the multiplication in O(1) time is n^3.

We now find the transitive closure in the following manner: Compute the product $A^2 = A^1A^1$. This will give us all paths of length 2 or less. Repeat the process, multiplying $A^4 = A^2A^2$, $A^8 = A^4A^4$, $A^{16} = A^8A^8$,... until we obtain A^n. As the last step, we zero out the major diagonal of A^n and look at all pairs (i,j) and (j,i) such that i<j. If both are 1, then nodes i and j are part of some cycle and hence, they are reachable from themselves. We therefore insert 1's in the entries (i,i) and (j,j) of A^n. This last operation can be accomplished in time O(1) with $\frac{n^2}{2}$ processors. Finally we "OR" the original diagonal of A (which was saved in s) with that of A^n to take care of nodes that have loops on them.

Clearly, the number of matrix multiplications required is O(log n), which is also the parallel time complexity of our algorithm. Moreover, the above multiplications can be performed "in-place", (i.e., no additional memory is needed to store the intermediate results).

Now consider graphs where n is not a "convenient" power of two. We have $A^i = A^{i+1}$ for all $i \geq n$, because in directed graphs the length of the longest possible simple path is at most

$n-1$. Consequently we can use for A^n a matrix A^m obtained by the matrix squaring process described above, where $n \leq m \leq 2n$.

3. Cycle Detection

The cycle detection algorithm is a simple modification of the transitive closure algorithm described above. Compute A^n as above. Next, make use of the following observation: A directed graph G is cyclic iff there exists at least one index pair (i,j), such that both (i,j) and (j,i) are 1 in the adjacency matrix representing the transitive closure of G.

With $\frac{n^2}{2}$ processors, the above condition can be checked in time O(1), where each processor p_{ij} reads $A^n(i,j)$ and $A^n(j,i)$, and outputs a "yes" if they are both 1. If after one time step "yes" has not been written out, then the graph is acyclic.

4. K-cycle Detection

Suppose we are not interested whether a given directed graph contains any (nontrivial) cycles, but whether it contains cycles of length at most k. If k is a power of two then we may use the first algorithm to obtain $A^{\frac{k}{2}}$ and use its (i,j) - (j,i) method for the k-cycle answer. It should be clear that a graph contains a k-cycle iff there exists at least one pair of nodes (i,j) such that i is reachable from j in $\frac{k}{2}$ steps or less **and** j is reachable from i via a path whose length is at most $\frac{k}{2}$. Unfortunately, there is a problem if k is not a power of two. The first algorithm would fail because the "extra" matrix multiplication could introduce cycles of length greater than k. Below, we present an algorithm that will correctly detect k-cycles in time $O(\log\ k)$ with n^3 processors.

Given the original adjacency matrix A, we set its major diagonal to be all 1's and compute $A^1, A^2, A^4, A^8, \ldots, A^{\lceil \frac{k}{2} \rceil}$ (note: these matrices must be stored separately in the main memory). This can be done in time $O(\log\ k)$ with n^3 processors.

We then compute $A^{(\lceil \frac{k}{2} \rceil)^2}$ multiplying the appropriate A^{i}'s such that their exponents add

up to $\lceil \frac{k}{2} \rceil$. With n^3 processors, we can clearly do this in time $O(log\ k)$. Assuming k is even, we can again use the (i,j) - (j,i) method to detect possible k-cycles. If k is odd, we must also compute $A^{(\lfloor \frac{k}{2} \rfloor)^*}$ and compare $a_{ij}^{(\lceil \frac{k}{2} \rceil)^*}$ to $a_{ji}^{(\lfloor \frac{k}{2} \rfloor)^*}$ in order to detect the odd length cycles.

References

1. Arjomandi, E.R. and Corneil, D.G., "Parallel Computations in Graph Theory," *SIAM J.Computing*, vol. 7, pp. 230-237, 1978.

2. Cook, S.A., "The Classification of Problems which have Fast Parallel Algorithms," *Lecture Notes in Computer Science # 158: Foundations of Computation Theory*, pp. 78-93, M. Karpinski ed., 1983.

3. Kucera, L., "Parallel computation and conflicts in memory access," *Inf. Process. Lett.*, vol. 14, pp. 93-96, 1982.

4. Quinn, M.J. and Deo, N., "Parallel Graph Algorithms," *ACM computing surveys*, vol. 16, pp. 319-348, 1984.