International Journal of Parallel Programming, Vol. 17, No. 5, 1988

Practical Parallel Union-Find Algorithms for Transitive Closure and Clustering

G. Cybenko,[1] T. G. Allen,[2] and J. E. Polito[3]

Received December 1988; Revised April 1989

Practical parallel algorithms, based on classical sequential Union-Find algorithms for computing transitive closures of binary relations, are described and implemented for both shared memory and distributed memory parallel computers. By practical algorithms, we mean algorithms that are efficient for parallel systems with bounded numbers of processors as opposed to algorithms where the number of processors grows with the problem size. Transitive closures are useful for decomposing many applications problems into independent subproblems. The implementations were on an ENCORE Multimax shared memory machine and an NCUBE hypercube. Our implementations indicate that transitive closure computations are intrinsically difficult for distributed memory parallel machines because of the need for global information. By contrast, our results for shared memory machines exhibited excellent speedups.

KEY WORDS: Parallel algorithms; clustering; transitive closure.

1. INTRODUCTION

This paper studies parallel algorithms for computing the transitive closure of a reflexive binary relation which we believe is a fundamental problem in

[1] Center for Supercomputing Research and Development, University of Illinois, Urbana, Illinois 61801. Supported in part by NSF Grant DCR-8619103, ONR contract N000-86-G-0202 and DOE Grant DE-FG02-85ER25001.

[2] ALPHATECH, INC, 111 Middlesex Turnpike, Burlington, Massachusetts 01803. Supported in part by RADC contract F30602-85-C-0303.

[3] ALPHATECH, INC, 111 Middlesex Turnpike, Burlington, Massachusetts 01803. Current address: Department of Computer Science, Duke University, Durham, North Carolina 27706. Supported in part by RADC contract F30602-85-C-0303.

0885-7458/88/1000-0403$06.00/0

many large scale applications. The algorithms we develop are based on parallelizations of Union-Find algorithms which are simple to implement and extremely efficient in the sequential case. The problem we study is simple to state and its utility in many applications should be immediately evident. Suppose that an application problem, called the *global* problem, involves some number of objects that may or may not be *related* to one another. Whenever two objects are *related* they determine an element of a binary relation and they belong to the same subproblem of the gobal problem. The computation of the transitive closure identifies the maximal number of independent subproblems as determined by the global problem data. We will use the term *clustering* to refer to the general problem of computing the transitive closure of a binary relation. The resulting equivalence classes are called *clusters*.

Clustering is an important general technique for problem decomposition. The basic goal of clustering is to decompose a computational problem into subproblems (clusters) that can be independently solved. The benefits of this are apparent for both sequential and parallel implementations of many problems. The overall computational complexity of an algorithm is typically reduced by such a decomposition while the independence of subproblems has the extra benefit that different clusters can be handled by different processors in a parallel implementation. A particular feature of the clustering problem is that clusters are typically only determined during runtime, so that clustering represents a form of dynamic problem decomposition.

This paper studies practical parallel algorithms for computing clusters on parallel computers. In our opinion, this work represents a significant departure from the numerous other works on parallel algorithms for computing connected components of graphs or transitive closures of binary relations. Our approach involves the parallelization of the most efficient sequential algorithm, namely a well-known version of the *Union-Find* algorithm whereas all other research on this problem typically deals with algorithms that require at least $O(n^2/\log^2 n)$ processors to solve a problem of size n. Such approaches are not practical for systems with a bounded number of processors because simulation attempts do not lead to efficient parallel algorithms. We describe parallel algorithms for both shared memory and distributed memory machines. Computational experiments have been performed on machines of these two distinct types—a shared memory architecture (the ENCORE MultiMax computer) and a distributed memory, message passing machine (NCUBE hypercubes).

Our experience suggests that distributed memory parallel architectures are not well suited for general clustering computations because of intrinsic reasons. This is due to the fact that the problem involves global informa-

tion that appears to be difficult to compress or encode and so severe communications penalties are encountered—our hypercube algorithms actually experienced slowdowns instead of speedups as the number of processors increased while keeping the problem size and algorithmic strategy fixed. Moreover, our clustering algorithms have linear complexity as functions of problem size so that area/perimeter effects[1] never play a role. It turns out that scaling to increasingly larger problems has an effect if the scaling is with respect to the ratio between the number of relation pairs and total number of objects. In that case, very high speedups can be observed. The hypercube performance should not be interpreted as a negative result about distributed memory machines in general since clustering is a preprocessing step to some subsequent algorithm (which presumably will benefit significantly from the decomposition of the large problem into smaller independent subproblems).

By contrast, our parallel algorithm for a shared memory system did exhibit some excellent speedups over a range of machine sizes. The shared memory version of our algorithm used a single global shared data structure for accumulating information about clusters. Moreover, we used busy-wait locks to implement exclusive write access during critical sections of the algorithms. The role of using busy-wait locks and cache refreshing appears to be a second order influence on algorithm performance.

Section 2 of this paper formulates the basic problem precisely and surveys a few applications where this problem arises. Section 3 is devoted to a review of the well-known *Union-Find* algorithm for the sequential computation of the transitive closure of a reflexive binary relation. The Union-Find algorithm turns out to be a basic building block of all our parallel algorithms so the review is justified as background. Section 4 describes a hypercube algorithm while Section 5 is devoted to a shared memory parallel algorithm based closely on the sequential Union-Find algorithm. Section 6 is a discussion of our experimental results.

2. PROBLEM FORMULATION AND APPLICATIONS

Suppose that in a particular problem instance, a total of m objects are present and these objects are indexed by integers j for $1 \leqslant j \leqslant m$. The underlying problem determines a reflexive binary relation on the set of objects—two objects are related if and only if they are related via the underlying problem. We will simply write (i, j) to indicate that object i is related to object j. We call an element, (i, j) of the relation, R, a *relation pair* of R instead of an element of R in order to avoid confusion with array elements which we will introduce shortly. The transitive closure of this relation is an equivalence relation and the resulting equivalence classes are

precisely what we call clusters. For the purpose of completeness, recall that the transitive closure of the relation $R = \{(i, j)\}$ is the smallest relation satisfying the recursive definition:

$$T = \{(i, j) | \text{ there exists a } k \text{ for which } (i, k) \in R \text{ and } (k, j) \in T\} \quad (1)$$

and

$$R \subseteq T \quad (2)$$

A simple graph theoretic analogy of transitive closure is obtained by thinking of the objects and relations of the original relation, R, as forming the vertices and edges of a undirected graph respectively. The resulting equivalence classes in the transitive closure are precisely the connected components of that graph.

At this point it is appropriate to give some examples of applications where this problem arises.

- *Linear system of equations*[2] Associate the nonzero entries of the coefficient matrix with edges in a graph in which the nodes are labeled by rows and columns. The connected components of this graph then determine permutations of the rows and columns so that the permuted matrix is block diagonal. The associated linear system decouples into independent systems with one subsystem for each component.
- *Data association problems* The authors of this paper were originally motivated to study the clustering problem because of their work on tracking and data fusion problems.[3–5] The way in which clustering arises naturally in a data association problem can be easily illustrated. Suppose that some number of objects are being observed by a collection of sensors. Each sensor reports spatial and possibly other types of measurements about objects but with errors. Loosely speaking, the data association problem is to decide which measurements from each sensor correspond to which measurements from other sensors. A standard approach to such problems is to first make gross pairwise correlations of the form: measurements i and j are possibly measurements for the same object—these correlations are the relation pairs of a reflexive binary relation. The equivalence classes of the transitive closure of this relation identify independent subproblems.
- *Pattern recognition*[6] The use of clustering in pattern analysis and recognition problems is well documented in the literature. A common situation for example is to group patterns according to

their proximity in some pattern feature space. Two patterns belong to the same cluster if their features are within a threshold distance according to some metric. The resulting equivalence classes or clusters are taken to be the number of distinct, identifiable patterns in the recognition problem.

A few words about the abstract PRAM-model (see Refs. 7 and 8) of parallel complexity of this problem are appropriate. Suppose that we have a PRAM with concurrent reads and exclusive writes (the CREW model). Then there is a simple $O(\log^2 m)$ algorithm for computing transitive closures of binary reflexive relations so that the problem belongs to the class NC (see Ref. 9). Here m is the number of objects given in the problem. We assume that the initial data of the problem consists of distinct pairs, (i, j). We form the node-adjacency matrix, M, of a graph determined by this relation in $O(1)$ PRAM steps. Next we compute the mth Boolean power of this matrix, M^m. The matrix M^m has a 1 in position (i, j) if and only if i and j are in the same connected component of the graph since the shortest path from i to j must be less than m in length. By using the iteration

$$M_0 = M \tag{3}$$

$$M_j = M_{j-1}^2 \qquad \text{for} \quad j = 1,\ldots,\lceil \log_2 m \rceil \tag{4}$$

and noting that a PRAM can compute the product of two m by m matrices in $O(\lceil \log_2 m \rceil)$ steps, we have a PRAM algorithm that uses $O(\log^2 m)$ steps and no more than m^3 processors.

A more complete discussion of PRAM-model and other algorithms for computing transitive closures and other path algebra problems can be found in Refs. 10–13. Additionally, recent work on parallel algorithms for transitive closure of database relations can be found in Refs. 14 and 15. By contrast, our interest is in deriving parallel algorithm for realistic parallel machines where the number of processors is fixed and independent of the problem size. The PRAM algorithms for these problems typically require unrealistically many processors in order to achieve their stated performance and they are not efficient in the sense that using fewer processors to simulate the algorithm does not result in efficient algorithms.

3. THE BASIC UNION-FIND ALGORITHM

The Union-Find algorithm takes a number of set relations of the form

$$i \text{ and } j \text{ belong to the same set}$$

and maintains a data structure that stores the resulting set memberships.

The way in which we use the Union-Find algorithm in our algorithms is quite straightforward—a relation pair of the form $(i, j) \in R$ means that i and j belong to the same set (equivalence class) in the transitive closure. The Union-Find algorithm builds these sets by processing a list of such relation pairs. We review the basic ideas in this section and refer the reader to standard references for a more thorough treatment of this method (for examples see Refs. 16–18).

A set is represented by a tree data structure. The particular way that a tree represents a set is that all vertices in the tree belong to the same set. The root of the tree stores a nonpositive number that is 1 minus the number of vertices in the tree. The collection of all trees (a forest) is stored in an array as follows.

- Objects are labeled and henceforth identified with integers, i, $1 \leqslant i \leqslant m$.
- Each nonroot vertex in a tree is represented by the corresponding array element and points to its parent by storing the array index for its parent vertex.
- A tree's root vertex, as outlined above, stores the tree size in the form $1 -$ (size of the tree). The size of the tree is the number of vertices in the tree.
- The array is initialized to have all entries 0 which corresponds to every vertex being the root of a singleton tree.

The basic idea is to use the root of a tree as a unique representative element of the set. Thus two objects belong to the same set if the trees that they belong to have the same root. This can be checked by following pointers to the roots of the trees and comparing for equality. If we find that two elements are not in the same set and we want them to be in the same set we union the two sets. In the forest data structure used, this is done by having one root point to the other. This merging of trees requires performing the appropriate update on the representation of the tree size. The demoted root now points to the surviving root and the number stored in the surviving root is the sum of the two previous root numbers minus 1.

Depicted in Figure 1 is a forest. Figure 2 illustrates the array representation of this forest. Note that nonpositive values indicate a root node and the nonpositive value is 1 minus the size of the tree.

We introduce two functions that will help subsequent discussions. The specific forest being manipulated by the Union-Find algorithm is a parameter of the function. Thus,

$$root(F, i)$$

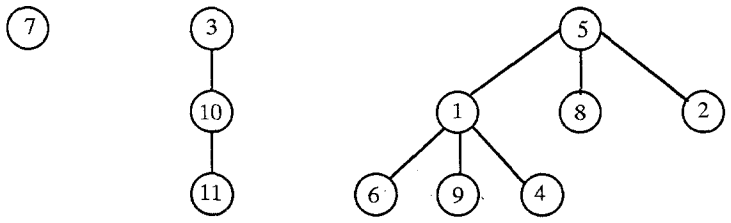

Fig. 1. A forest of trees.

is a function whose value is the root of the tree to which i belongs in the forest F. The function *root* requires pointer chasing only. Furthermore,

$$update(F, i, j)$$

is the procedure that updates the forest, F, using the information that (i, j) is a relation pair. Note that

$$update(F, i, j) = update(F, root(F, i), root(F, j))$$

and that *update* is implemented by calling *root* twice to find which vertices require updating.

The key performance factor in the Union-Find algorithm is the average number of times a pointer has to be advanced to find the root of the tree to which it belongs. In order to keep this number small, it is important to try to keep the trees as shallow and as broad as possible. This can be accomplished by using two standard techniques: path compression and weighted unions.[16–18]

Path compression involves the following construct: after starting with a vertex and following pointers to the root of its tree, retrace the path taken from the vertex to the root and arrange for all vertices on that path to point directly to the root. Clearly, path compression brings all vertices on the path to a depth of one from the root. At the same time it decreases the depth of subtrees hanging off vertices on that path. Another common form of keeping trees shallow is path halving whereby nodes along a searched path are made to point to their grandparants: that is, their parent's parent. The performance of halving based Union-Find algorithms

5	5	-2	1	-6	1	0	5	1	3	10
1	2	3	4	5	6	7	8	9	10	11

Fig. 2. Array representation of the forest in Fig. 1.

is identical to those using path compression[16] and our choice to use path compression was strictly arbitrary.

The other important scheme used in the basic implementation of Union-Find algorithms is weighted union when merging trees. The weighted union rule is simply a protocol for determining which of two roots remains a root and which is demoted when merging two trees. As noted above, two trees are joined by having one root point to the other root. The weighted union protocol specifies that the root of the smaller tree (as indicated by the size of the tree that is stored in the root entry) should point to the root of the larger tree. That is, the root of the smaller tree is demoted to non-root status. The role that this plays in keeping trees shallow is quite clear. Since the smaller tree gets demoted, fewer nodes are at a deeper level in the merged tree. The depth of the nodes in the larger tree is not affected and so the average depth of a node in the merged tree is minimized.

The use of path compression and weighted unions greatly improves the performance of Union-Find algorithms. A complete analysis of the algorithm and its variants can be found in Ref. 16. In that paper, it was shown that for serial algorithms the Union-Find algorithm with path compression and weighted unions is optimal in a reasonably exhaustive class of algorithms for set union type problems.[16] The basic result is that building a set structure of the type described here when k relations of the form "i and j belong to the same set" are processed, the algorithm requires no more than

$$\alpha(k)\, k$$

steps. A step is either a pointer update or a pointer reference. The increasing function $\alpha(k)$ is related to second order logarithms (and the inverse of the Ackermann function) and is bounded by 4 for k less than 2^{65536}. Hence, for practical purposes, we can treat this factor as a constant smaller than 4 and one can regard this algorithm to have linear worst case behavior.

Our parallel algorithms use the basic Union-Find algorithm with path compression and weighted unions in the same way always—our parallel algorithms use an array to represent a forest and new relation pairs are added to this data structure using the Union-Find algorithm. In the case of hypercube algorithms, different processors may build different array representations based on the relation pairs available to them. Those arrays must then be merged and the Union-Find algorithm is used for that as well. In the shared memory algorithm, all processors simultaneously update a single array using the Union-Find algorithm but some updates require exclusive access.

4. HYPERCUBE ALGORITHM

We assume that each processor stores locally some number of relation pairs. We represent the information available to each processor as a list of such relation pairs.

There is always a simple possible solution to clustering on a hypercube: collect all the relation pairs at a single processing node and solve the problem serially on that node. Alternatively, we can perform a sequence of merging steps whereby the forest information is successively merged between pairs of adjacent processors until all information is collected in a single forest. It turns out that there is a whole family of algorithms with these two algorithms as extreme cases.

In our hypercube algorithm, information about the relation is constantly updating forest data structures. The information that updates the forest is in one of two forms:

1. A relation pair, (i, j), updates a forest;
2. One forest, F_2, updates another forest, F_1.

A relation pair, (i, j), updates the forest using $update(F, i, j)$. To update one forest, F_1, by another, F_2, we must scan one forest array element by element, constructing a relation pair of the form

$$(j, root(F_2, j)) \tag{5}$$

for each object j. This collection of relation pairs is iteratively incorporated into the other forest using the simple relation pair scheme in item 1. Namely using

$$update(F_1, j, root(F_2, j))$$

To be more specific, suppose that we have a d dimensional hypercube, H_d. According to the usual convention, we use the binary integers between 0 and 2^{d-1} as labels for the nodes. Nodes whose labels differ by exactly one bit are neighbors in the hypercube.

Select some integer k so that $0 \leqslant k \leqslant d$. Consider the k dimensional subcube, denoted by H_k, that consists of nodes from H_d whose $d-k$ least significant bits are cleared. Thus, for example, H_0 is the zero dimensional hypercube consisting of the node 0, H_1 is the subcube consisting of $00\cdots0$ and $10\cdots0$ while H_d is the whole hypercube itself. The class of algorithms we describe here depend on the choice of the dimension of subcubes in a critical way—the best performance is obtained by selecting a subcube dimension whose size is derived from the problem size itself.

Forest merging in our hypercube algorithm proceeds according to the well-known embedded binary tree scheme. The idea is that at step j, all

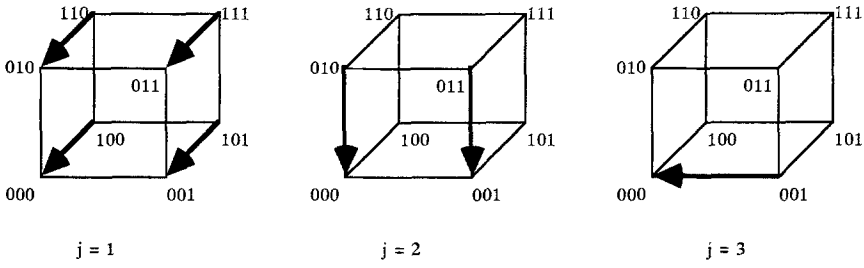

Fig. 3. 3-dimensional gather and merge operation.

processors send their forest arrays to their neighbors whose binary labels differ in precisely the jth bit with the convention that information is sent from the node with its jth bit set to the neighboring node with its jth bit cleared. We need k such steps, one for each dimension in a k-dimensional binary hypercube. It can be easily checked that after all k steps, all local information has been collected at node 0. Figure 3 illustrates the three steps required to gather and merge the forest arrays for $k = 3$.

We now describe an algorithm for each k, $0 \leqslant k \leqslant d$.

Hypercube Clustering Algorithm(k)

1. Hypercube node, labeled $b_1 b_2 \ldots b_d$, communicates its list of relation pairs directly to the node in H_k with label $b_1 b_2 \cdots b_k 00 \cdots 0$.
2. Each node in H_k collects the relation pairs sent to it and builds a forest based on them using the simple function, *update*.
3. Through a sequence of dimension exchanges within H_k as described earlier, the forests are merged using the forest merging scheme.

The particular dimension k that is optimal in this algorithm depends significantly on the problem instance. Recall that m is the total number of objects. In addition, assume that each processor has n distinct relation pairs. Roughly speaking, the algorithm consists of two communication/computation phases.

In the first phase, relation pairs are sent to the closest node in H_k. This requires collecting $(2^{d-k} - 1)\, n$ relation pairs by each node in H_k which is equivalent to communicating $(2^{d-k} - 1)\, 2n$ integers (two integers per relation pair). Next each node in H_k must use the Union-Find algorithm to incorporate $2^{d-k} n$ relation pairs into the forest. This requires no more than $2^{d-k+2} n$ pointer updates by the Union-Find algorithm. Next, these forests on the nodes of H_k must be merged in a total of k steps. At

each step, m relation pairs of the form $(i, R(i))$ where $R(i)$ is the root of i and $1 \leqslant i \leqslant m$ must be incorporated into a forest. This requires a total communication of km integer words. The actual merging takes no more than $4km$ pointer updates.

Summarizing the total communication requirements, we see that

$$(2^{d-k}-1)\,2n + km \tag{6}$$

integers must be serially communicated. This expression does not take into account the startup overhead. The distance that a message must travel is not relevant in the first phase of the algorithm because processors in H_k collect messages from neighbors at all distances between 1 and k. Hence, the nodes in H_k are continuously engaged in the serial reading of $(2^{d-k}-1)\,2n$ integers. The merging of forest arrays within H_k requires k communications between nearest neighbors.

Similarly, a rough total of

$$2^{d-k+2}n + 6km \tag{7}$$

pointer updating operations are required. The first term in Eq. (7) comes from each node of H_k using the Union-Find algorithm on the $2^{d-k}n$ relation pairs that it has collected. The second term comes from merging forest arrays. About $2m$ steps are required for finding the root of each element in the forest array. Next, about $4m$ steps are required to update the other forest array with the resulting relation pair. This must be repeated a total of k times, giving $6km$ steps.

Comparing these two expressions, we see that the computation and communication times are almost proportional. We will minimize the expression in Eq. (7) because the speed of the processors compared with the speed of communication suggest that pointer updating will be more time consuming. The minimums of the expression in Eq. (7) is approximately achieved for

$$2^{d-k} = \frac{3m}{2n \log 2} \tag{8}$$

This is approximate because $d-k$ must be a nonnegative integer. Now $d-k$ is the dimension of the subcubes that are complimentary to H_k and $2^{d-k}-1$ is the number of nodes that send their relation pairs to a single node within H_k. The more objects there are relative to the number of relation pairs, the more overhead there is in the third phase of the algorithm since it involves both computation and communication costs that are proportional to the number of objects.

Asymptotically, the more relation pairs there are relative to the number of objects, the better the distributed memory performance of the previous algorithm will be. To see this, note that there are a total of $2^d n$ relation pairs in the whole problem. A serial algorithm would require about $2^{d+2}n$ steps while if $2m \leqslant 3n \log 2$ we would get $k = d$ for the optimal algorithm so that the parallel algorithm would involve about $4n + 6dm$ steps. For $n \gg m$, the speedup of the parallel algorithm approaches 2^d which is impressive for large d. Note that in the number of distinct relation pairs is bounded by $m^2/2$ so there is an upper bound to the meaningful size of n in relation to m.

5. SHARED MEMORY ALGORITHM

The shared memory parallel algorithm we describe is also based on the Union-Find algorithm but now there is only a single shared array that maintains all the forest information. A list of relation pairs is stored in shared memory as well. A processor goes to the list and obtains the first relation pair that has not already been merged into the forest array. The processor searches the array to find the roots of the two objects in the relation. There are two parts of the Union-Find algorithm that require writing to the array: path compression and root updating (tree merging). These writing phases are the only parts of the shared memory algorithm that require close inspection in order to verify that the consistency of the data structure is maintained. We will treat those two cases separately.

First consider the case of root updating. We claim that root updating forms a critical section of the parallel algorithm and requires exclusive write privileges on the root vertices specifically and the forest array more generally. To see this consider the example of the forest in Fig. 4. There are three root nodes in this example—i, j, k that are roots of trees of sizes 6, 11 and 21 respectively. Suppose we have the two relation pairs (i, j) and (i, k). If we do not require exclusive write access to the root vertices, then it would be possible to interleave the two steps: 1. find the root vertices and 2. update the root vertices. Suppose that processor A finds the two root vertices i and j and processor B finds the two root vertices i and k. Now

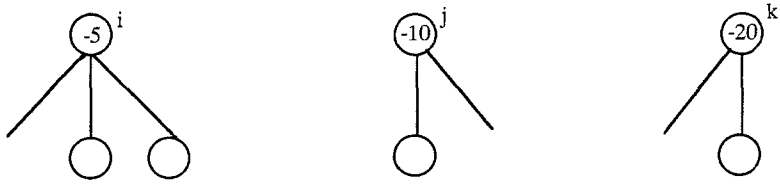

Fig. 4. A sample forest.

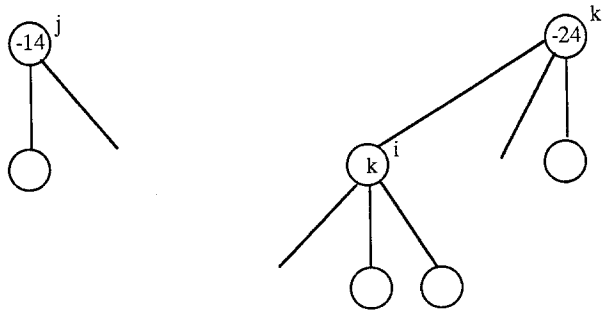

Fig. 5. Incorrectly updated forest.

processor A has i point to j and updates the tree size stored at root j. Next processor B makes i point to k and updates the tree size of k. This results in two trees (see Fig. 5) whereas the correct forest would have only one tree with all three vertices in it (see Fig. 6). Hence, root updating is a critical section and requires exclusive access to root vertices for writing. This could be accomplished by using fine-grained locks, one for each array element. However, we implemented the exclusive access by using a single busy-wait lock (called a spin-lock on the ENCORE) for write privileges on the whole array. The ENCORE spin-lock is a find grained lock that is most useful in situations where short waits are expected. Our decision to use a single busy-wait lock for the whole array arose from the observation that the total time consumed by waiting for locks to be released was a very small part of the total execution time. Accordingly, the increased memory requirements (namely doubly the storage of the forest array) did not appear to be worth the marginal performance improvement obtained.

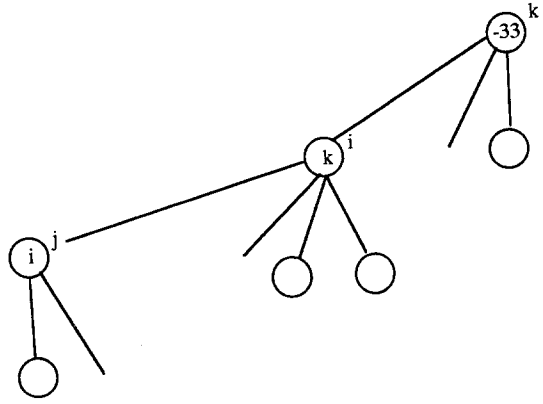

Fig. 6. A correctly updated forest.

The locking protocol works as follows. A processor determines that two root vertices should be updated: one must be made to point to the other and the tree size must be updated. That processor goes into a busy-wait state until the write lock is free. It sets the lock and checks again that the two vertices are still roots since in the intervening time some other processor may have updated those vertices and made them nonroots. If either of the two vertices is no longer a root, pointers must be followed to find the new roots. Once the two roots are found (they must indeed be the required roots since the write lock is set and no other processors are allowed to update the array), the updating occurs and then the lock is released.

The other vertex updating step in the Union-Find algorithm is path compression. We claim that path compression is a safe operation in the sense that no exclusive write access is required for updating during path compression. To see this, assume that we implement the algorithm with root updating as above and path compression without any locking mechanism. We claim that the data structure always enjoys the following two invariants: vertices always point to other vertices in the same tree as them, and vertices never becomes roots once they are nonroots. Using these invariants, it is easy to see that updating vertices for path compression without any exclusive access is safe. Assume that vertex i is a descendant of root j. Between the time that j was determined to be the root of i's tree, either j has remained the root or it has been demoted to nonroot status. Regardless of the case, having i point directly to j is correct although it may actually lengthen the path instead of shortening it. This latter case can happen when j was made to point to some other root say k and i was made to point directly to k. Now having i point to j lengthens the path in this situation.

The Union-Find algorithm for shared memory outlined above is difficult to analyze because of the possibility of such (admittedly modest) path lengthening possibilities. Moreover, it appears difficult to predict the time wasted during busy-waiting on write locks for root updating. At present, there are no noninvasive performance analysis tools for measuring the degradation due to the use of fine grained locks in a shared memory machine.

6. COMPUTATIONAL EXPERIMENTS

We used comparable data in our experiments on the NCUBE hypercube and ENCORE shared memory multiprocessor. The problems we tested had $N = 3200$ and $m = 10, 100, 1000$. Note that for $m = 10$ the binary relation would be extremely dense while for $m = 1000$ the relation would be

extremely sparse. The case $m = 100$ is intermediate. Both shared and distributed memory experiments showed worst performance for sparse problems with best performance on dense problems.

6.1. Hypercube Experiments

Our hypercube, distributed memory experiments were performed on a 64 processor, NCUBE parallel processor. Each processing node in the NCUBE is rated at being about 1/4 MIPS performance with 512 kilobytes of local memory. Communications performance is about a millisecond to send one kilobyte from a node to a neighboring node. Communication times between neighboring processors are affine functions of the message size with a fixed overhead for initiating a communication of any size. This overhead is small enough compared to the actual cost of sending data in our experiments that we did not model it in this paper.

Figures 7–9 illustrate the hypercube performance for 3200 relation pairs and varying numbers of objects, m. The horizontal axes are the values of k so that data can only be obtained for hypercubes whose dimensions are greater than or equal to k. The vertical axes are compute time in milliseconds. The legend at the bottom indicates which bar textures correspond to different dimensioned hypercubes, d. The times do not include the time required to download data to each processing node since we expect that the basic clustering problem will in general result in data

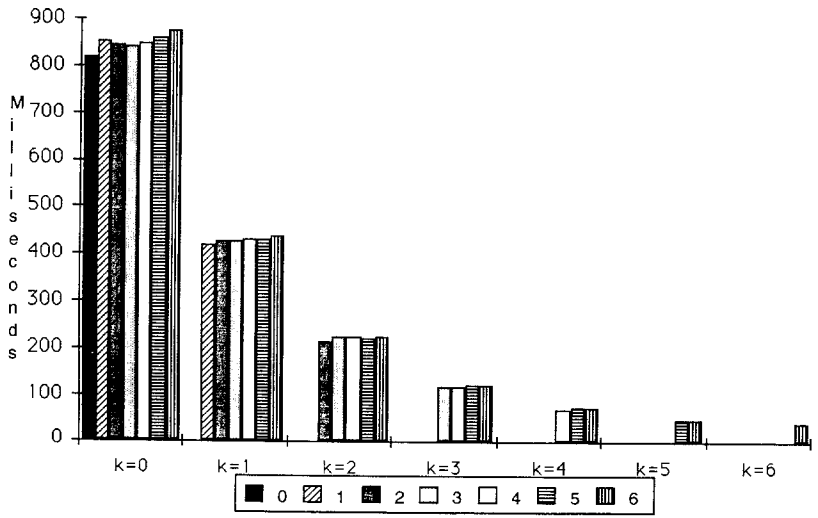

Fig. 7. Hypercube performance for $m = 10$, $2^d n = 3200$ (hypercube dimensions indicated by shading of bars.)

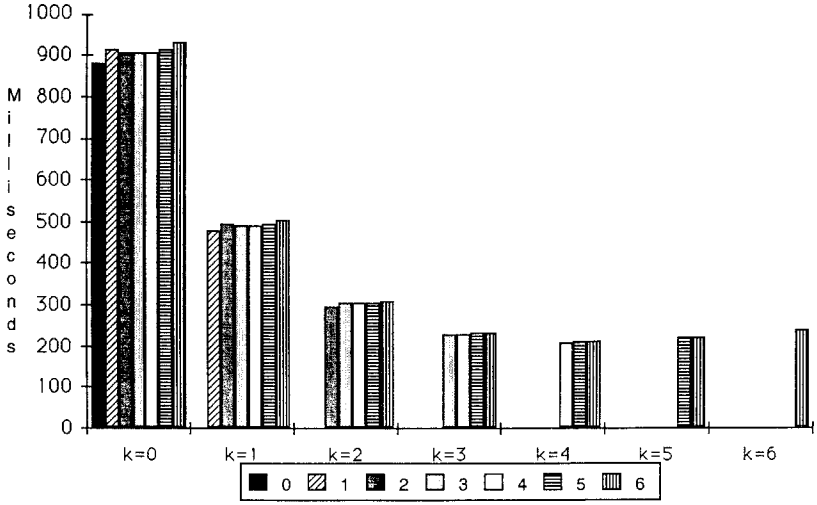

Fig. 8. Hypercube performance for $m = 100$, $2^d n = 3200$ (hypercube dimensions indicated by shading of bars.)

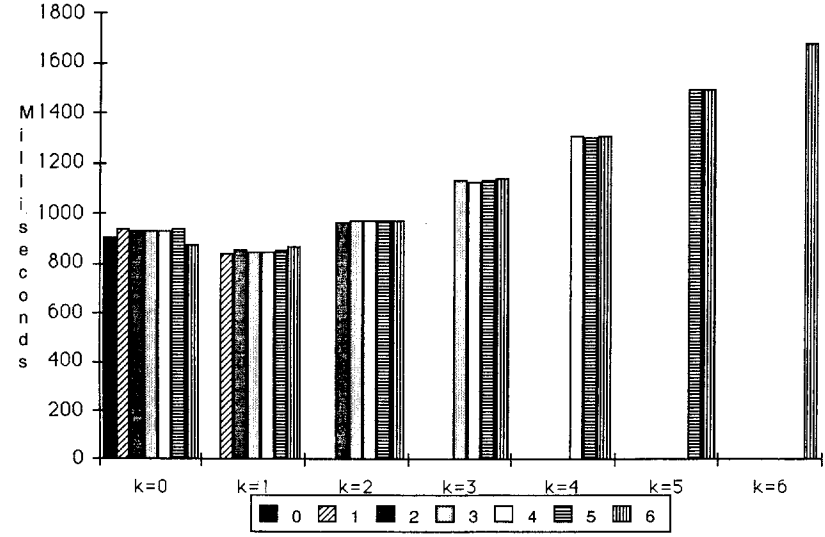

Fig. 9. Hypercube performance for $m = 1000$, $2^d n = 3200$ (hypercube dimensions indicated by shading of bars.)

already distributed to the nodes. The relational data used was generated randomly—uniformly distributed integers between 1 and m were generated and paired together to form the relation pairs.

One fact about the results that should be immediately visible is the insensitivity of performance to the hypercube dimension itself and the sole dependence on k, the dimension of the embedded hypercube that collects relation pairs. This can be verified by going back to Eq. (7) and noting that a total $N=2^d n$ relation pairs exist in a problem of fixed size. Using N instead of n, we write the Eq. (7) as

$$2^{-k+2}N + 6km \tag{9}$$

which is independent of d altogether. Thus, for a fixed k, the dependence on d is not very significant, although close inspection does show a modest slow down as d grows in most cases.

To see how the theoretical results compare with our experiments, rewrite (8) using $N=2^d n$ as the total number of relation pairs in the problem. We then get that

$$2^k = \frac{2N \log 2}{3m} \tag{10}$$

which again is independent of d providing that $k \leqslant d$. In our experiments, we used $N=2^d n = 3200$ giving

$$2^k = \frac{1478.71}{m} \tag{11}$$

For $m=10$, we get $2^k = 148.1$ suggesting that k should be taken as the largest possible subcube. Looking at Fig. 7, we see that the optimal algorithm does in fact correspond to the largest possible value of k within a cube. For $m=100$, we get $2^k = 14.8$ so $k=3$ or $k=4$. Figure 8 supports this with the minimum actually occurring at $k=4$. Finally, with $m=1000$ we get $2^k = 1.48$ so $k=0$ or $k=1$. Indeed, Fig. 9 shows that the minimum occurs at $k=1$. Hence, we appear to have excellent agreement between our theoretical analysis and our computational experiments.

To summarize the hypercube results, we have derived a family of algorithms for any specific problem instance. The family depends on a parameter k in the following way. Given a d dimensional hypercube, select a k dimensional subcube. Each processor sends its relation pairs to the subcube node closest to it. The nodes of the subcube collect the relation pairs and perform serial Union-Find algorithms to form a forest array. The forest arrays are merged and collected at one node finally. If the under-

lying problem involves a total of N relation pairs, with $N/2^d$ at each node in a d dimensional hypercube, then the optimal value of k is predicted by the expression

$$2^k = \frac{2N \log 2}{3m} \tag{12}$$

6.2. Shared Memory Experiments

The shared memory algorithm running on the ENCORE parallel computer exhibited excellent speedups in some cases. The data was generated as in the hypercube experiments—namely uniformly randomly. Figures 10 and 11 show the performance on four different problem sizes by graphing the actual raw times and derived speedups. We have named those four sizes A, B, C, D as follows:

$$A \quad m = 1000, N = 3200 \tag{13}$$

$$B \quad m = 100, N = 3200 \tag{14}$$

$$C \quad m = 10, N = 3200 \tag{15}$$

$$D \quad m = 100, N = 640 \tag{16}$$

Figure 11 has an additional data line, L, to indicate linear speedup for reference purposes. The compute time measured was the time between the

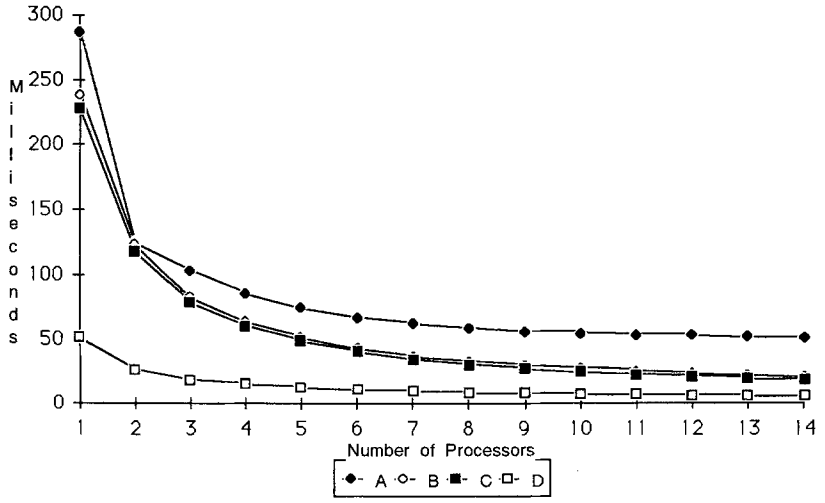

Fig. 10. ENCORE raw timings.

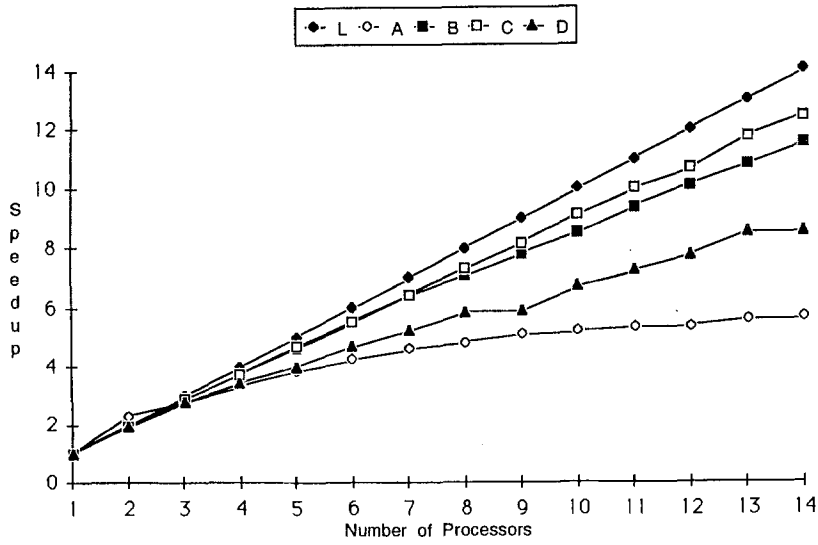

Fig. 11. ENCORE speedups.

completion of process forking and the time that the last process terminated. Hence the time does not include reading data from disk, the allocation of shared memory and other system functions. We should mention that process joining is a time consuming activity on the ENCORE.

The speedups indicate that the larger the number of objects is relative to the number of relation pairs, the worse the speedups are. The explanation appears to be the time spent in critical sections and updating the forest array. As the number of objects gets larger with a fixed number of relation pairs, the likelihood that a new relation pair actually results in an update of the array increases. This forces more time to be spent in root updating and path compression. As mentioned before, the modification of shared variables results in frequent cache updating and hence poorer program performance. Furthermore, the more objects there are in the problem, the larger the shared memory forest array must be so that the cache refreshing becomes costlier from this perspective as well. In particular, case *A* with 1000 objects seemed to be limited to no more than a speedup of 5 even when using 14 processors.

7. CONCLUSIONS

Shared memory architectures appear to be well suited to transitive closure computations. Some problem configurations exhibited excellent speedups using an ENCORE multiprocessor. Other examples seemed to

limit the speedup to about 5 even when 14 processors were being used. In principle, the shared memory algorithm we describe is capable of extremely high performance. However, system dependent factors such as the rate and efficiency of cache refreshing appear to play a major role but cannot be modeled precisely.

By contrast, our theoretical analysis and observed performance of hypercube algorithms agreed very well. The basic problem of transitive closure computation requires global information and so imposes a severe handicap on possible hypercube performance. Even so, theoretical analysis shows that some extremely large problems could be efficiently solved by hypercubes. However, we were not able to experiment with such problems because of memory and communication buffer limitations.

REFERENCES

1. J. L. Gustafson, G. R. Montry, and R. E. Benner, Development of parallel methods for a 1024-processor hypercube. *SIAM Journal on Scient. and Statist. Computing*, Vol. 9, 1988.
2. J. A. George and J. W. Liu, *Computer Solution of Large Sparse Positive Definite Systems*, Prentice-Hall, Englewood Cliffs, New Jersey (1981).
3. T. G. Allen, G. Cybenko, C. Angelli, and J. Polito, Hypercube implementation of tracking algorithms. In *Proceedings of JDL Workshop on Command and Control*, p. 145–153, Washington, DC (1987).
4. T. G. Allen, G. Cybenko, C. M. Angelli, and J. Polito, *Parallel processing for multitarget surveillance and tracking*, Technical Report TR-360, ALPHATECH, Inc., Burlington, Massachusetts (1988).
5. G. Cybenko and T. G. Allen, Parallel algorithms for clustering and classification. In *Proceedings of SPIE Conference on Advanced Architectures and Algorithms for Signal Processing*, p. 126–132, San Diego, California (1987).
6. P. Duda and R. Heart, *Pattern Classification and Scene Analysis*. J. Wiley and Sons, New York (1973).
7. G. S. Almasi and A. Gottlieb, *Highly Parallel Computing*, Benjamin/Cummings, Redwood City, California (1988).
8. M. J. Quinn, *Designing Efficient Algorithms for Parallel Computers*, McGraw-Hill, New York (1987).
9. A. Borodin, S. A. Cook, and N. Pippenger, Parallel computation for well-endowed rings and space-bounded probabilistic machines. *Inf. and Control*, p. 1–3:113–136 (1983).
10. F. Y. Chin, J. Lam, and I. Chen, Efficient parallel algorithms for some graph problems, *Communications of ACM*, p. 649–655 (1982).
11. V. Pan and J. Reif, *Fast and efficient solution of path algebra problems*, Technical Report TR.87.3, SUNY Albany, Department of Computer Science (1987).
12. J. D. Ullman, *Computational Aspects of VLSI*, Computer Science Press, Rockvill, Maryland (1984).
13. U. Vishkin, An optimal parallel connectivity algorithm, *Discrete Applied Mathematics* **9**:197–207 (1984).
14. R. Agrawal and H. V. Jagadish, Multiprocessor transitive closure algorithms. In *Proc. of Int. Symp. on Databases in Parallel and Distributed Systems*, IEEE Computer Society, Austin, Texas (August 1988).

15. P. Valduriez and S. Khoshefian, Parallel evaluation of the transitive closure of a database relation, *Int. Journal of Parallel Programming*, Vol. 17 (1988).
16. R. E. Tarjan and J. Van Leeuwen, Worst-case analysis of set union algorithms. *Jour. of Assoc. for Comput. Machin.*, p. 245–281 (1984).
17. A. V. Aho, J. E. Hopcroft, and J. D. Ullman, *The Design and Analysis of Computer Algorithms*, Addison-Wesley, Reading, Massachusetts (1974).
18. R. Sedgewick, *Algorithms*, Addison-Wesley, Reading, Massachusetts (1983).