TR 516

This paper presented at the Proceedings of the Second Conference on Hypercube Multiprocessors, Knoxville, TN, September 29-October 1, 1986.

Communication-Efficient Distributed Data Structures on Hypercube Machines

ZHIJING MU* AND MARINA C. CHEN*

Abstract

Sets, as programming data structures, have become more important than before due to the emergence of parallel machines. Efficient communication over sets of distributed data therefore is the crucial issue in parallel computing. We identify three distributed set data structures and two general operations over sets. Algorithms are developed to support the operations over distributed set structures in poly-logarithmic time. The algorithms have been implemented on the iPSC and the performance is presented. A full example is given to demonstrate the usage of the data structures. Discussion is made on the spectrum of different granularities.

1 Introduction

The concept of the set is fundamental in mathematics. Its importance has been recognized and reflected in programming languages like SETL [5] and SASL [6]. However, the set as a data structure is not widely used in most conventional programming languages due to the inherent sequentiality imposed by the von Neumann machine model. The emergence of parallel computers might change the situation. When a set of data is distributed over multi-processors, an operation defined over the set may be carried out with the participation of all the processors involved. Therefore, sets have become sources of parallelism and perhaps the most important data structures in parallel languages [1].

Suppose that we want to compute a function that depends on the values distributed over a group of processors on machines like the iPSC, the time used depends on largely the communication speed. To avoid message congestion, which would slow down communication, programmers have to specify the tedious message routing explicitly. We intend to treat sets as data structures with some general built-in operators. The algorithms presented in this paper will take care of the message routing so that the level at which the parallel programs are designed can be elevated without sacrificing performance.

In general, operations over a set of distributed objects may require communications. For example, to increase each object by a constant, local computation is sufficient, but if

*Department of Computer Science, Yale University, P.O.Box 2158, Yale Station, New Haven, CT 06520-2158. Work supported in part by the Office of Naval Research under Contract No. N0014-86-0296

we want to get the sum of all objects, communication among the processors containing the objects is necessary. Operations that demand communication over a set often can be classified as one of the following two types: broadcast and merge (see Section 3). These two types of operations have been shown to be general and powerful enough to express a broad class of parallel algorithms [3] [4]. This paper will concentrate on algorithms to support broadcast and merge operations over sets.

In real problems, we can often find the situation where the same operations are to be carried out for many sets. For example, with a matrix, we may want to find the minimum for each row. As the sets are independent of each other, the operations ideally are to be done in parallel. Our algorithms are designed to carry out the operations on more than one set in parallel with complexities independent of the number of sets involved.

Ho and Johnsson have developed algorithms to support broadcast for static sets [1] distributed over a cube or subcubes [2], which can be used to support operations over uniform sized sets; Leiserson and Maggs [4] have considered communication issues over dynamic sets, and developed randomized algorithms for the fat-tree machine model. In this paper, we will consider static sets with non-uniform sizes, and deterministic algorithms for dynamic sets.

The paper is organized as follows: In the next section, we will specify and discuss the communication issues of distributed data structures. In Section 3, three set implementations with communication algorithms are introduced. The performance of the data structures implemented on the Intel iPSC is presented in Section 4. In Section 5 the connected components problem is used to demonstrate the usage of three data structures. The final section is concerned with the adaptation of the data structures over the spectrum of different granularities of parallelism.

2 Communication Issues in Distributed Data Structures

In some graph problems, we may want to find for *each* vertex an edge of the smallest weight adjacent to it; in scientific computing, we may sum over *each* row of a matrix. These kinds of computations can all be described by the following general operation:

Given k disjoint sets of data $S_1, \ldots, S_k$, and an associative operator $\oplus$, compute $\bigoplus_{a \in S_i} \mathrm{a}$ for $i = 1, \ldots, k$.

There are two levels of parallelism in the operation. First of all, the computation for the k sets can be done in parallel since the sets are independent from each other. Secondly, for each set, the operation $\oplus$ can be performed via a spanning tree structure with cost logarithmic to the size of the set.

The simplest case is that of static sets with a uniform size, for instance, rows of a matrix. In this case, we can simply map each set into some subcube in a hypercube, and construct a spanning tree structure for each subcube to explore the parallelism at both levels [2].

However, we may encounter the case of k static sets of non-uniform sizes, e.g. the adjacency lists of a graph. As forcing each of the non-uniform sized sets into a subcube will waste processors, objects in one set should be distributed across the boundaries of subcubes. The first implementation of sets, what we call consecutive lists, will facilitate the operations over non-uniform size static sets with logarithmic time complexity.

When the operation is to be carried out over dynamically formed sets, the efficiency becomes a more complex issue. The locations of the elements of a dynamic set are unpredictable before run-time; therefore, efficient communication can only be supported by dynamic routing schemes.

[1] A set defined before run-time and not subject to change is static, otherwise is dynamic.

The problem with dynamic sets is further complicated by the following constraint: as dynamic sets may be changed by *union*, *intersection*, etc., operations, it is essential that the memory required to represent a set does not grow with the size of the set. This implies that information about elements' locations should be distributed rather than centralized at any one processor. One way of defining the scope of a set under the constraint is to have a pointer for each element designating another element in the same set.

We will present two implementations for dynamic sets. Non-consecutive lists are used to deal with the situation where the pointers link elements in a set into a list; rooted trees are for the situation where the pointers link the elements into a rooted tree. It will be shown that operations over both the two kinds of dynamic sets can be carried out in poly-logarithmic time.

3 Three Distributed Data Structures Implementing Sets

3.1 Preliminary

The development of the data structures is based on the following three principles:

- *Distributed Data*: the elements in sets are distributed over processors of a message-passing parallel machine.
- *Distributed Computations*: a function over a set is computed by all the processors where the elements reside, each processor computes part and only part of the function.
- *Distributed Information*: Each processor representing an element in a set has part and only part of the information about the entire set structure. The required local memory for storing the information should not grow with the size of the set.

We will consider the following two types of operations over distributed sets:

- *Broadcast*: given a set S and a value $v(x)$ associated with an element $x \in S$, send $v(x)$ from x to all elements in the set.
- *Merge*: given a set S, a value $v(x)$ associated with each $x \in S$, an associative and commutative binary operator $\oplus$, compute $\bigoplus_{x \in S} v(x)$.

The operations can be carried out in poly-logarithmic time if we can find some tree structure that has the following properties.

- All elements in the set are in the tree.
- The height and fan-in of the tree are bounded by $O(\log n)$ or $O(\log^k n)$ for some small constant k.
- It can be constructed in poly-logarithmic time.

The problem of supporting the set data structures therefore can be reduced to the problem of construction of such trees, which, following Leiserson and Maggs [4] will be referred to as communication trees (CTs) throughout the paper.

To simplify the discussion, in this section we assume:

- Time complexity of the algorithms is measured by the number of communication steps required. This assumption is well justified since local computations in our algorithms take constant time.
- Synchronized machine model. As the synchronization can be achieved by paying logarithmic factor in time [8] , the given complexities are valid within a logarithmic factor for asynchronous machines like the Intel iPSC.
- One and only one datum is mapped to one processor. (However, we will devote Section 4 to the discussion on the spectrum of machine granularity.)

3.2 Consecutive lists (C-lists)

Consecutive lists are the representation of static sets with non-uniform sizes, e.g., the adjacency lists of a graph. Notice that rows and columns of matrix are just special cases of C-lists.

Consider p static sets $S_1, \ldots, S_i, \ldots, S_p$ with arbitrary sizes $m_1, \ldots, m_i, \ldots, m_p$. For the jth elements in the ith set, we can assign a new index l by the following equation:

$$l = j + \sum_{k=1}^{i-1} m_k \tag{1}$$

The new indices actually impose a total order over elements in all sets. With the gray code we can map each element according to its index l onto the hypercube. Obviously, two elements will be physically adjacent in the cube if and only if they were adjacent elements in the sets, and therefore the processors containing all elements in one set form a consecutive list.

The following algorithm constructs CTs for the C-lists. It is a simple recursive process starting from the head of each C-list and terminating at the elements found to be leaves in the CT. The variables in all algorithms of this paper are local, global variables shared by all processors are not used.

Algorithm 1

All processors in parallel:
(1) if the processor contains the head element of a list $\{e_{l_1}, \ldots, e_{l_m}\}$, then
    FATHER := self;
    INTERVAL := $[l_1, l_m]$;
  else INTERVAL := NIL.
(2) if INTERVAL is not NIL , then
    if INTERVAL = $[s, f]$ and $s \neq f$ then
      for D := cube dimension to 0 do
        NB := the Dth dimension neighbor of the processor;
        if NB > s and $\leq f$ then
          claim that NB is a son in CT;
          INTERVAL := [s, NB-1];
          send INTERVAL [NB, f] to the son processor.
    else the processor is a leaf of the CT.
  else
    receive (INTERVAL, sender);
    FATHER := sender;
    do step (2);
end.

An example of a C-list is given in Figure 1. The C-list consists of elements mapped to processors with IDs from 6 to 13 (gray code). The constructed CT is given on the right side of Figure 1.

The bound of the height of constructed CTs, though not very apparent, can be shown to be $2 \log n$ with the help of the following lemma:

Lemma 1 *From the root of the CT along any path to a leaf, the dimension of the link decreases by one in at most every two links.*

Following the lemma, we have,

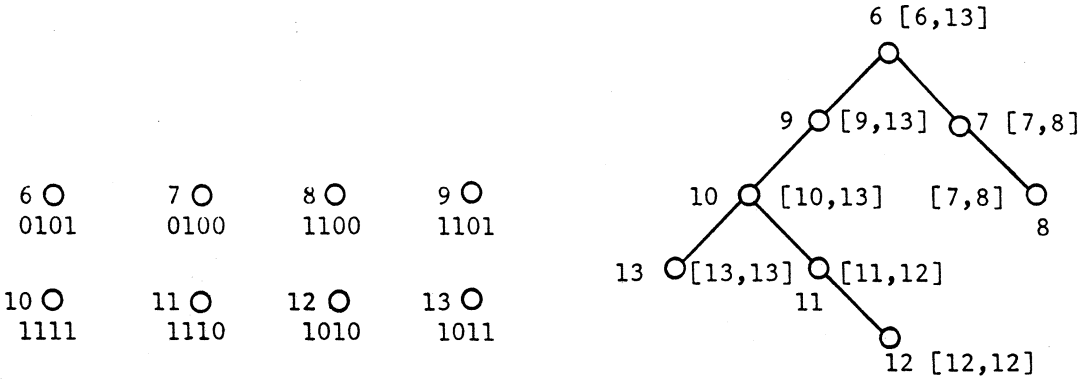

Figure 1: Building the CT for a C-list.

Theorem 1 *The height of a CT for a consecutive list is bounded by* $2\log n$*; fan-in of any node in CT is bounded by* $\log n$*, where* n *is the length of the list.*

Notice that the CT for a C-list consists of only links connecting the elements in the list; therefore, when there is one or more consecutive lists, both the process of CT construction, and the broadcast or merge operation can proceed for all C-lists in parallel without communication interference from each other. Since the links of the CTs correspond to physical links between processors in the hypercube, a message sent across a link can arrive in unit time. Therefore, we conclude the time required for CT construction, broadcast, and merge operation are all $O(\log n)$ for a static set of arbitrary size n.

In fact, assume that each element in a C-list has information about the head and tail element indices, each elements can obtain the INTERVAL by local computation. So, the CTs for C-lists can be constructed with no communication cost.

3.3 Non-consecutive lists (N-lists)

Dynamically formed sets can be randomly located in a group of processors and two adjacent elements can be physically far away in the hypercube. According to the information distributing principle, the elements' locations are not known by any one processor. Instead, an element A knows of only element B in the same set by a pointer designating B. In the case where one element is pointed to by at most one other element in the set, the dynamic set forms what we call a non-consecutive list.

Algorithm 2 constructs CTs for the N-lists. The basic idea is simple: initially only the head of a list is inactive, all other elements will query first the left neighbor, and then the left neighbor of the left neighbor, and so on, until it reaches an inactive element. Then the inactive element reached becomes the father of the element in the CT and the element itself becomes inactive.

Algorithm 2
All processors in parallel:
(1)if the processor is correspondent to the head of a list then
    ACTIVE := false;
    FATHER := self;
  else

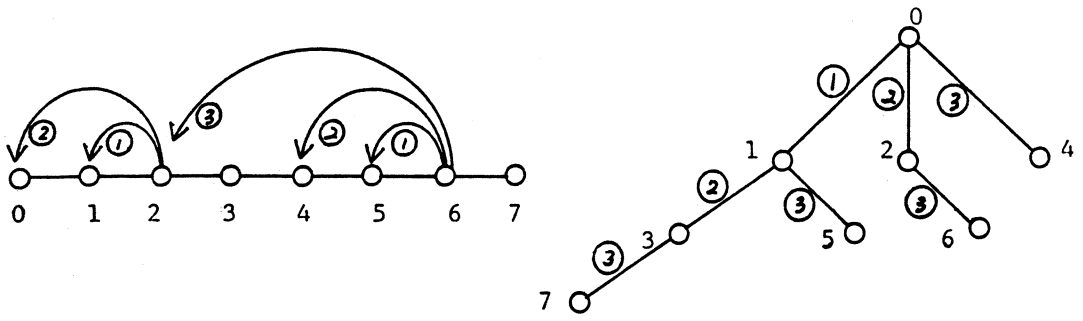

Figure 2: Building the CT for a N-list.

```
    ACTIVE := true;
    PRED := left-neigbor;
(2) while ACTIVE = true do
    send query to PRED;
    if PRED is active then PRED := the PRED of PRED;
    else
      ACTIVE := false;
      FATHER := PRED;
end.
```

In Figure 2 we give a N-list with 8 elements.[2] Notice how element 6 queries node 5, 4 and 2 in sequence, and becomes the son of element 2, since element 2 was inactive when the query was made at step 3.

Theorem 2 *The height and fan-in of the CT for any N-list of length n are bounded by* $\log n$.

Proof: Let the time step of a query sent initially be 0, the time step of the second query be 1, and so on. Give the links of the CT labels which correspond to the final query's time step. Assume an element in CT has a path from the root labeled $l_0, l_1, \ldots, l_m$, then observe that the index of the element in the list is $\sum_{i=0}^{m} 2^{l_i}$.

Observe that at any step in constructing or using the CT, one processor is accessed by at most one other processor. The communication pattern at each step is therefore a partial permutation, and can be done in logarithmic time in the hypercube [8]. The number of steps is bounded by the height of CT, which is $\log n$. Broadcast and merge over N-lists can therefore be done in $O(\log^2 n)$ time.

Algorithm 2 is similar but essentially different from the "pointer doubling" technique [3]. Communication with "doubling" is achieved by handshaking between the head and each element in a list. In $\log n$ time step, the head must deal with n elements. To avoid message congestion $O(n)$ processors must be used for each element. By contrast, Algorithm 2 assumes only one processor for one element.

The CT constructed is either a partial or complete binomial tree [2] depending on whether or not the length of the list is a power of two. It can be shown for the purpose of

[2]The given indices of the elements do not correspond to the host processor IDs.

communication that binomial trees is superior to balanced trees for machines like iPSC, where the message start-up time is expensive.

We have treated the CT construction and operations as two separate phases. In fact the operations can be done while the CT is being constructed. In the process of CT construction, an element must reach an inactive element before it stops, but the inactive element is either the head of the list or an element that that has directly or indirectly reached the head. A message from the head thus can pass to each element in the list. Similarly, merge operation can also be done while the CT being constructed.[3]

3.4 Rooted trees (R-trees)

The rooted tree is another type of dynamic set. Similar to the N-list, a node is linked to the set only by a pointer to another node. While an element can be pointed by at most one element in a N-list, a node in a R-tree may be pointed by more than one other nodes in a R-tree.

As there is no restriction on the shape of the R-trees, a R-tree can be at one extreme very flat and fat and at another extreme very thin like a chain. In either case, using the R-tree itself to achieve the communication would require at least linear time.

Because a processor containing a node of R-tree knows of only another node, i.e., the father in R-tree, it must communicate with the father as the first step to communicate with other elements in the set. However in some cases, for example a flat tree, in the very first step serious message congestion will occur(see Fig.5.(right)). This problem suggests that something must be done between the source and destination of a message.

A technique, *message merge*, has been devised to overcome this problem: When a message x is being sent from processor S to processor D, passing processors $m_1, ..., m_i, ..., m_p$, each intermediate processor m_i will check if a message y of the destination and same type has passed before. If so, the message x will be held in processor m_i and will be responded to by the reply for y; otherwise it will be passed to processor m_{i+1}. It is clear that two message for the same destination and of the same type will never be sent through the same link in the cube with message merge technique.

The CT construction algorithm for R-trees with *message merge* technique is sketched below.

Algorithm 3

All processors in parallel:

(1)if the processor is correspondent to the root of R-tree then
    FATHER := self;
    ACTIVE := false;
  else
    ACTIVE := true;
    PRED := father in R-tree;

(2) while ACTIVE = true do
      send query to PRED;
      if PRED is active then PRED := the PRED of PRED;
      else

[3]Let the value associated with an element x be $v_0(x)$. While an element queries $x_1, \ldots, x_m$ do: $v_1(x) := v_0(x) \oplus v_0(x_1), \ldots, v_m(x) := v_{m-1}(x) \oplus v_m(x_m)$. Then $v_i(x)$ is the merge result with the 2^i elements to the left of x. When the tail element becomes inactive, it contains the result of merging all elements in the list.

Notice that an element x does not merge with an element y until it has merged with all elements to the right of y, hence, the binary operator $\oplus$ is allowed to be non-commutative. Moreover, the procedure actually computes the listfix function [4] with the result distributed over all the elements in the list.

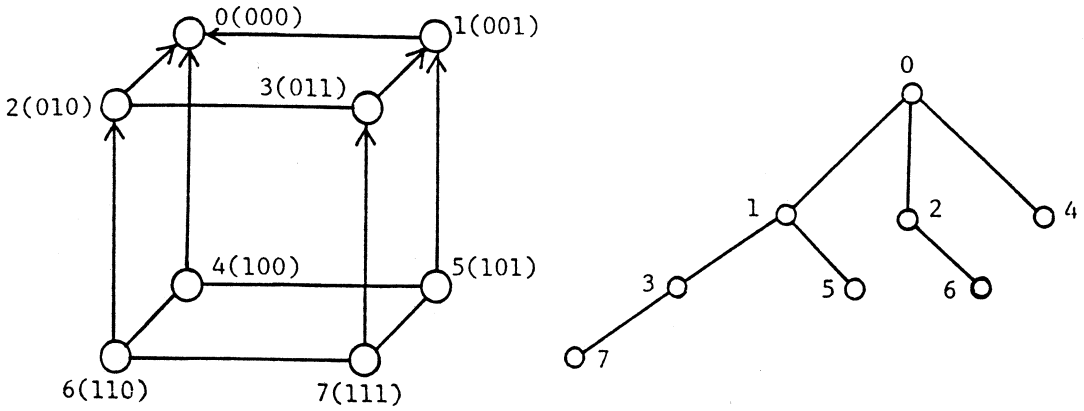

Figure 3: Building the CT for a rooted tree.

ACTIVE := false;
if the query was merged at node M then FATHER := M;
else FATHER := PRED;
end.

A R-tree is given in Figure 3 to illustrate the algorithm. The R-tree is flat, node 0 is the root and all other nodes are sons of node 0.

Theorem 3 *Let n be the size of a R-tree, N the size of cube; assume $N = O(\log n)$, then the height and fan-in of the CT for the R-tree are bounded by $O(\log n)$.*

Proof: The bound of the height can be shown with the same argument for N-lists. Because of message merge, a processor can at most have $\log N$ sons in the same CT, hence, the fan-in $<= O(\log n)$.

When a message is broadcasted through the CT, the left-most leaf of the left-most branch will receive the message in $O(\log^2 n)$ time, since each link of the CT may take up to $O(\log n)$ time and the height of the tree is $O(\log n)$. On the other hand, for the right-most leaf of the right-most branch in CT, the time used may add to another $O(\log n)$ factor, as at each level of the CT the processor may have to send $O(\log n)$ messages. The conclusion is that operations over a set of n data with R-tree structures can be carried out in $O(\log^3 n)$ time in the hypercube [4].

Notice that when $k > 1$ R-trees exists , a processor may become a node in more than one CT due to the message merge. The communication time certainly increases when taking the problem into account. However, poly-logarithmic time can be proven sufficient for any number of R-trees. [5]

[4] In comparison with N-lists, we are paying an extra $O(\log n)$ factor of time for the operations. The reason being that the shape of the CTs for N-lists is always a partial or perfect binomial tree, where the depth of a given branch is always smaller than the branch to its left.

[5] Limited by space, the proof cannot be given here. We are also working on another version of the algorithm, where a node can be in at most one CT.

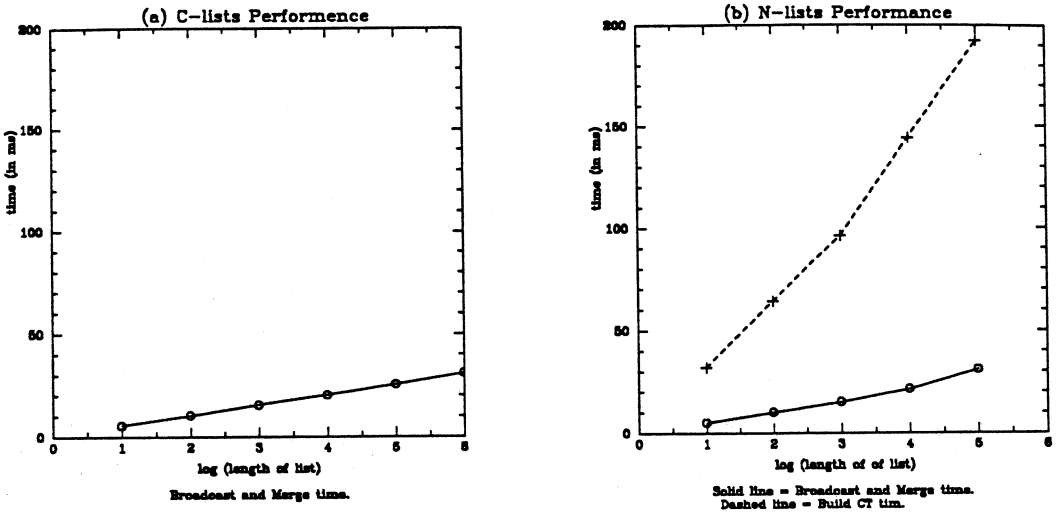

Figure 4: Performance of the CTs for C-lists and N-lists.

4 The Performance of the Algorithms

The data structures have been implemented on the Intel iPSC. The performance is given in Figures 4 and 5. [6] It can be observed that the time used for the CT construction and operations is indeed within the bound given by the theorem for each data structure.

The performance of R-trees algorithm can vary a great deal depending on the shapes the number of existing trees. The performance given in Figure 5(a) is only for the case of single flat trees. For comparison, we measured the time required to do the same operations over the same flat trees without the CTs. The results (Figure 5(b)) show that linear time is required.

5 On the Spectrum of the Granularity

Even with high dimension cubes, we may have to map more than one datum into one physical node for large scale problems. For algorithms supporting data structures to be really useful, they must be able to adapt to different granularities; moreover, the performance should scale well, i.e., the time cost should vary approximately linearly along with the number of given processors. This section is to show that the algorithms for the set structures indeed have the above properties.

Throughout the following discussion we will assume that n is the size of the data, m is the number of data mapped to one processor, and the size of the hypercube is $O(n/m)$.

(1) C-lists: Assume that the m elements are always in the same set. A CT can be constructed by Algorithm 1 if we treat each m elements in a processor as one element in the algorithm. One node of the CT still corresponds to one processor but contains m elements. The m elements can share the message broadcasted to the host processor and can merge with each other and send a single result up to the father processor of the CT. The amount of communication work of a processor in broadcasting and merge is therefore independent of the number m. The time required is decided by the height of the CT, i.e., $\log n/m$.

[6] The operation time given is the average of two hundred broadcast and merge over the CTs.

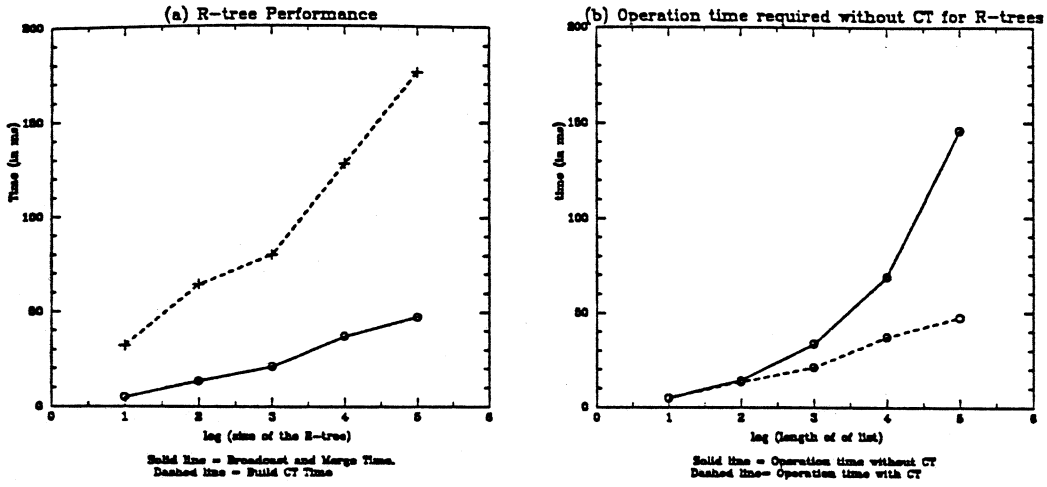

Figure 5: Performance for R-trees with CTs and without CTs.

(2) N-lists: As N-lists are dynamically formed sets, the m elements in one processor may belong to m different sets. We therefore have to apply Algorithm 2 to each element [7] in the processor and a factor of m is added to the complexity given by Theorem 2. The height of CT is still in the worst case $\log n$; however, each link of the CT may take less time for a message to go across as the cube size decreases when m increases. The time complexity is therefore $O(m \log n \log n/m)$.

(3) R-trees: Just as with Algorithm 2 in the N-lists, Algorithm 3 should be applied to each of the m nodes residing in one processor. With the same arguments as in (2), we can show that for one R-tree, the time cost of communications is bounded by $O(m \log n \log^2 n/m)$.

If n is fixed, the time complexities for N-lists and R-trees decrease linearly within a $\log n$ and $\log^2 n$ factor respectively when m decreases. Put in another way, the speed-up of the algorithms increases about linearly with the number of processors used.

6 An Application Example

A parallel algorithm for the connected component problem is as follows [7]:

Algorithm 4
Initial: each node is a partially connected component (PCC);
Iterate $2 \log n$ **times:**
(1) for each node i in G in parallel do
compute m_i = minimum PCC of a neighbor of i;
(2) for each PCC p in parallel do
find n_p = minimum of p and the $m_i's$ for i in p;
(3) for each PCC p in parallel do
find h_p, the limit of the sequence $p, n_p, n_{n_p}, ...,$;
(4) for each node i in parallel do
$p_i := h_p$;
end.

[7] A more sophisticated algorithm is to find out which of the m elements indeed belong to one set by local computation, and package the messages of those elements together. The message traffic will be certainly reduced when elements in that same set tend to reside in the same processors.

The above algorithm is independent of the implementation. Now assume we map the adjacency lists with gray code onto a hypercube (Section 3.2). Then obviously step (1) is a merge over consecutive lists with a binary operator MIN. The partially connected components are dynamically formed sets of vertices of the graph, and step (2) is just a merge over non-consecutive lists with again the MIN operator. In step (3), if we regard the h_p as a pointer from one PCC to another, all the pointers can be shown to define a rooted forest [7], and the limit of the sequence $p, n_p, n_{n_p}, ...$, for a PCC p actually is the value of the root of a R-tree; hence, step (3) can be achieved by broadcasting over R-trees. Finally step (4) can be done by broadcasting over PCCs, which are N-lists.

The algorithm iterate $2 \log n$ times, and each step of the algorithm can be done in poly-logarithmic time. Therefore, the implementation of the connected component algorithm with the data structures has poly-logarithmic complexity.

Let V be the number of vertices, E the number of edges of a graph. As adjacency lists are used in the above implementation, we use $O(\log E)$ processors. By contrast, implementations that require an adjacency matrix as input use $O(\log V^2)$ processors [7] [3]. For sparse graphs, (where $E = O(V)$), the implementation with the data structures can save $O(V)$ number of processors and still gives poly-logarithmic performance.

7 Concluding Remarks

It is obvious that the C-lists are just special cases of the N-lists whereas the N-lists are special cases of the R-trees. The trade-off between the generality and efficiency among the three data structures should be observed.

The set structures with the supported operations are general enough to be applied to a wide class of graph, combinatorial, set operation and system programming problems such as minimum spanning tree, max-flow, garbage collection and load balancing problems.

References

[1] M. C. Chen, *Very-high-level Parallel Programing by Aggregate Set Operations*, Research Report YALEEU/DCS/RR-499, Yale University, 1986.

[2] C. -T. Ho and S.L. Johnsson *Distributed Routing Algorithms and for Broadcasting and Personalized Communication in Hypercube*, 1986 ICPP p640 - p648.

[3] M-D A. Huang, *Solving Graph Problems with Optimal Speed up on Mesh-of-Tree Networks*, 26th Annual Symposium on Foundations of Computer Science. 1985.

[4] C. E. Leiserson and C. M. Maggs, *Communication-Efficient Graph Algorithms*, Proceeding of the 1986 International Conference on Parallel Processing.

[5] J. T. Schwartz *On Programming - An Interim Report on the SETL Project*, New York University, 1973.

[6] D. A. Turner *SASL Language Manual*, St Andrews University Technical Report, 1976.

[7] J. D. Ullman, *Computational Aspects of VLSI.* Computer Science Press, Inc., 1984.

[8] L. G. Valiant, *A Scheme for Fast Parallel Communication,*. SIAM J. Computing 11:2. 1981.