Experience with Crystal, Charlotte and Lynx
Second Report

by

Raphael Finkel
Bahman Barzideh, Chandreshekhar W. Bhide
Man-On Lam, Donald Nelson
Ramesh Polisetty, Sriram Rajaraman
Igor Steinberg, G.A. Venkatesh

Computer Sciences Technical Report #649
July 1986

Experience with Crystal, Charlotte, and Lynx
Second Report

Raphael Finkel
Bahman Barzideh
Chandrashekhar W. Bhide
Man-On Lam
Donald Nelson
Ramesh Polisetty
Sriram Rajaraman
Igor Steinberg
G. A. Venkatesh

Computer Sciences Department
University of Wisconsin–Madison

Abstract

This paper describes several recent implementations of distributed algorithms at Wisconsin that use the Crystal multicomputer, the Charlotte operating system, and the Lynx language. This environment is an experimental testbed for design of such algorithms. Our report is meant to show the range of applications that we have found reasonable in such an environment and to give some of the flavor of the algorithms that have been developed. We do not claim that the algorithms are the best possible for these problems, although they have been designed with some care. In several cases they are completely new or represent significant modifications of existing algorithms. We present distributed implementations of PLA folding, a heuristic for the travelling-salesman problem, incremental update of spanning trees, ray tracing, the simplex method, and the Linda programming language. Together with our previous report, this paper leads us to conclude that the environment is a valuable resource and will continue to grow in importance in developing new algorithms.

This work was supported by DARPA contract N00014-85-K-0788 and NSF grant MCS-8105904.

TABLE OF CONTENTS

1. Introduction

At the University of Wisconsin – Madison, we have built an environment for experimenting with distributed programs. This paper is a sequel to a previous one, in which we described projects that use Crystal, Charlotte, and Lynx[Finkel86a].

The **Crystal** multicomputer[DeWitt84a] is a collection of about 20 VAX-11/750 computers called **nodes** connected by an 80 Mb/sec token ring. A subset of nodes, called a **partition**, can be allocated to a distributed program. Partition allocation is mediated by software that resides on a **host** machine running Unix.§ Crystal provides a low-level reliable message facility within each partition. A user can inspect output to the node's terminal through a **virtual terminal** facility that redirects terminal I/O to a terminal (or window) on the host. Output on virtual terminals can be saved in Unix files for later inspection.

Charlotte[Artsy86a] is an experimental distributed operating system that can run in a Crystal partition of any size. Programs running under Charlotte communicate through **links**, which are two-way channels whose ends can be sent in messages (and thus relocated to other processes). The Charlotte user interface consists of a command interpreter process through which one can enter interactive commands to start processes, read command scripts, or interpret a connector file. The **connector** utility process is called to interpret connector files. These files specify what processes to start and how to interconnect them by initial links. Policy matters, such as on which node to start a process, are decided by other utility processes that the casual Charlotte user need not understand. Other utilities available to Charlotte processes include file service and a name service (to find well-known servers).

The **Lynx** programming language[Scott85a] provides linguistic support for distributed applications run under Charlotte. Any number of Lynx processes may be loaded into a Charlotte partition. Processes execute in parallel (with arbitrary interleaving of execution for processes on the same physical machine) and do not share any memory. They communicate with each other across language-defined links, which are in turn based on Charlotte links. Links initialized by the connector are presented as arguments to the main procedure of a Lynx module. Other links can be created and disseminated dynamically. They can be **bound** to entry points, which are like function declarations. If a process executes a remote call through a link bound to an entry point, a new thread of control is created at the destination process to service that call. Threads of control within the same process may share memory. They do not execute in parallel; the current thread continues until it blocks. We call this the **mutual-exclusion** property of threads.

This paper presents several new implementations based on Lynx and presents an evaluation of our distributed computing environment. These projects were conducted as part of a seminar in distributed algorithms during Spring, 1986.

§ Unix is a registered trademark of Bell Laboratories.

2. PLA folding

Experimenter: Sriram Rajaraman

2.1. Introduction

A programmable logic array (PLA) is an effective tool for implementing multiple output combinational logic functions in VLSI circuits[Mead80a]. A PLA has the general structure shown in Figure 1. In this figure, the inputs and their complements run vertically through a matrix of circuit elements called the AND plane. The AND plane generates signals that are combinations of the inputs and their complements. These signals then become inputs to another matrix of circuit elements called the OR plane. The outputs of the OR plane are the sum of products form of the Boolean functions of the PLA inputs. Each horizontal line of the PLA carries a product term. An example of a PLA is shown in Figure 2. A cross in the AND plane shows the presence of corresponding input in the term. A cross in the OR plane shows the presence of a corresponding product term in the output. In general, a PLA can be described in symbolic form by a matrix called the PLA **personality matrix**[Hachte82a]. The personality of the PLA in Figure 2 is shown in Figure 3. A 1 in position (i,j) means that the jth input (output) or its complement is present in the *i* th product term. A 0 in position (i,j) indicates that there is no connection.

Most PLA personality matrices are sparse, so a straightforward physical design will result in a significant waste of silicon area. That is, a significant fraction of the PLA area will be occupied by interconnect only and will not contribute directly to the implementation of logic functions. Row and Column **folding** of a PLA are techniques that attempt to reduce the area of a PLA by exploiting its sparsity.

The remainder of the paper is organized as follows. In Section 2 we introduce the notion of folding and then describe the optimal folding problem. In Section 3 we describe a heuristic algorithm presented by Hachtel[Hachte82a] for the optimal folding problem. In Section 4 we offer a distributed algorithm for solving the folding problem. Section 5 describes the implementation of the algorithm in Lynx. Section 6 presents some performance results obtained by running the Lynx program under Charlotte. Section 7 summarizes the author's experience with Lynx, and Section 8 contains directions for future work.

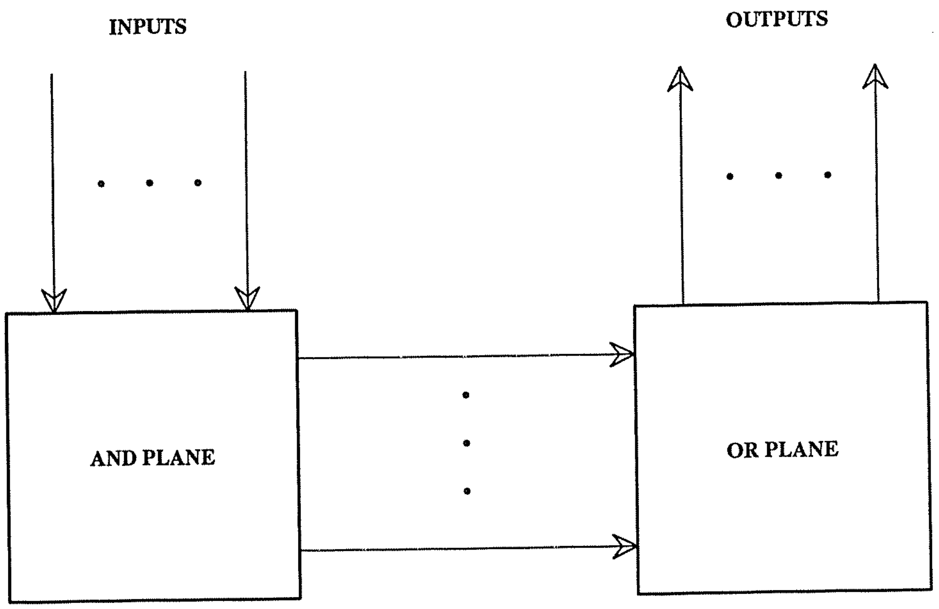


Figure 1. General structure of a PLA

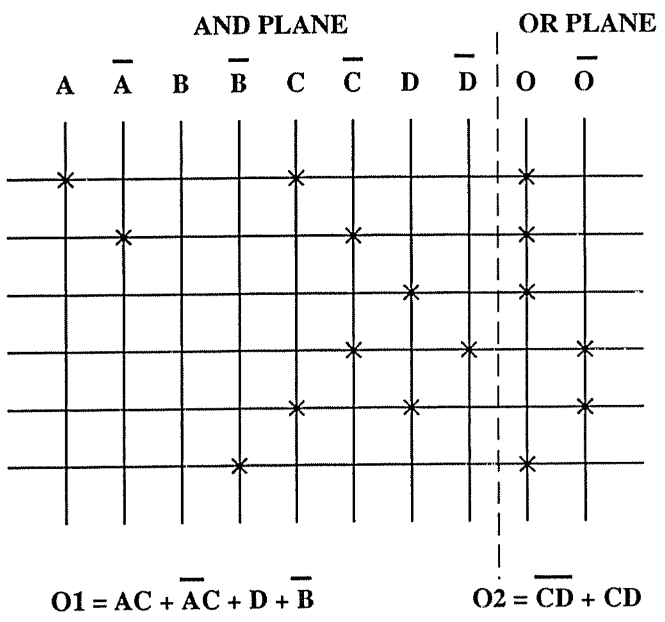

AND PLANE
OR PLANE
A $\bar{A}$ B $\bar{B}$ C $\bar{C}$ D $\bar{D}$ O $\bar{O}$
O1 = AC + $\bar{A}$C + D + $\bar{B}$
O2 = $\overline{CD}$ + CD

Figure 2. Sample PLA

1	0	1	0	1	0
1	0	1	0	1	0
0	0	0	1	1	0
0	0	1	1	0	1
0	0	1	1	0	1
0	1	0	0	1	0

Figure 3. Personality of sample PLA

2.2. Optimal PLA folding problem

In general, many types of folding are possible, depending on the technology used to implement the PLA. In the present paper, we restrict ourselves to one particular kind of PLA folding called SCF — simple column folding[Hachte82a] SCF consists of either input-column folding or output-column folding. In either case, a physical AND (OR) plane column is divided into two parts so that two electrical input (output) signals can share the same physical connections of the two signals in the given physical column. Of course, as illustrated in Figure 4, the electrical connections of the two signals in the given physical column must not be intermixed, but instead be grouped on opposite sides of a physical cut located somewhere in the column. One of the folded input (output) signals must then be routed from the top of the folded PLA and the other from the bottom. In the implementation of the algorithms described in this paper, we do not distinguish between input and output columns while doing the folding (for the sake of simplicity).

Not all columns of a PLA personality can be folded. An obvious necessary constraint is that two columns be disjoint, that is, the 1's of the two columns (in the PLA personality) must be in different

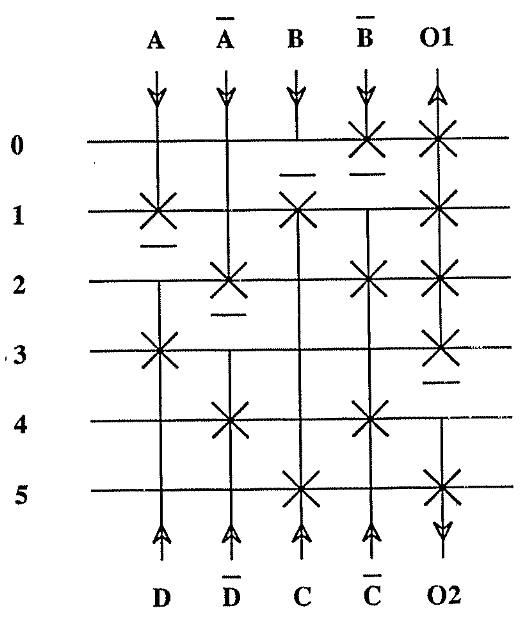


Figure 4. Column folding of the sample PLA

positions. In fact, the folding of two columns in a PLA personality introduces constraints on folding of other columns. Folding column i with column j forces all product terms containing the ith input(output) or its complement to be on top of or on the bottom of all the product terms containing the jth input (output) or its complement. For example, consider the PLA shown in the earlier figure. Folding columns 1 and 2 makes it impossible to fold columns 3 and 4. Hence, to fold a PLA we have to specify the pairs of columns to be folded and their relative position (top or bottom). For the folding to be implementable, the relative position of the columns to be folded should be such that no cyclic constraint on the position of the rows of the PLA personality is implied. The goal of folding is to minimize the area occupied by the PLA. Hence the optimal PLA folding problem is to find a maximal set of folding pairs (the maximum number of such pairs is half the number of columns) without introducing a cycle in the row ordering.

Hachtel has shown the above problem to be equivalent to a graph problem. What follows is an informal description of that problem. Consider the example PLA personality shown earlier. It can be represented by an undirected graph, as shown in Figure 5. There is a one-one correspondence between the vertices of the graph and the column numbers of the PLA personality matrix. There is an edge between vertices i and j of the graph if there exists at least one row in the PLA personality that has non-zero entries for both columns i and j. Each distinct folding for the PLA (each pair of non-adjacent vertices with no vertex in common with any existing folding) can be represented as a directed edge on the graph. A directed edge between columns i and j implies that all rows with non-zero entries for column i will be placed above (or below) all rows with non-zero entries for column j in the folded PLA. For example, consider the PLA represented in Figure 5. The directed graph associated with the folding set {(c1,c4),(c3,c2),(c5,c6)} is shown in Figure 6.

The folding problem reduces to finding the maximum number of directed edges that can be added to the initial undirected graph without creating an **alternating cycle**. An example of an alternating cycle in Figure 6 is the sequence of vertices [5,6,3,2,5].

2.3. Serial algorithm for PLA folding

Hachtel has shown that the optimal PLA folding problem is NP-complete. He presents a heuristic algorithm for solving the folding problem, which is described below.

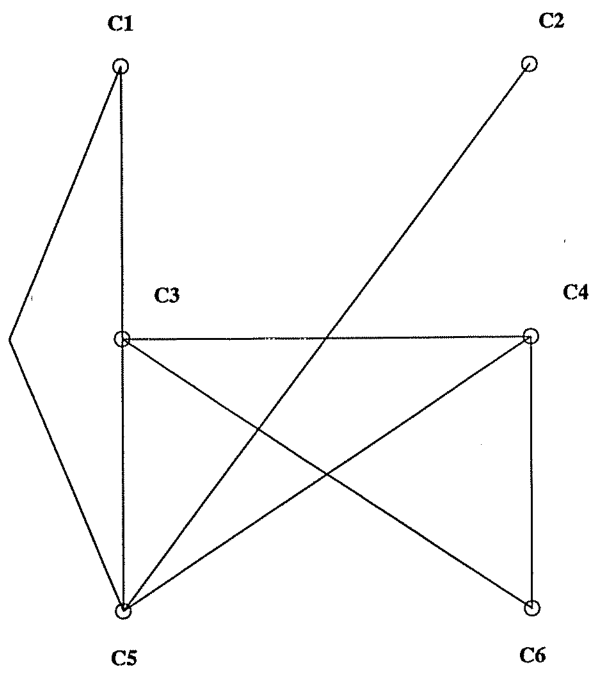


Figure 5. Graph representation of the sample PLA

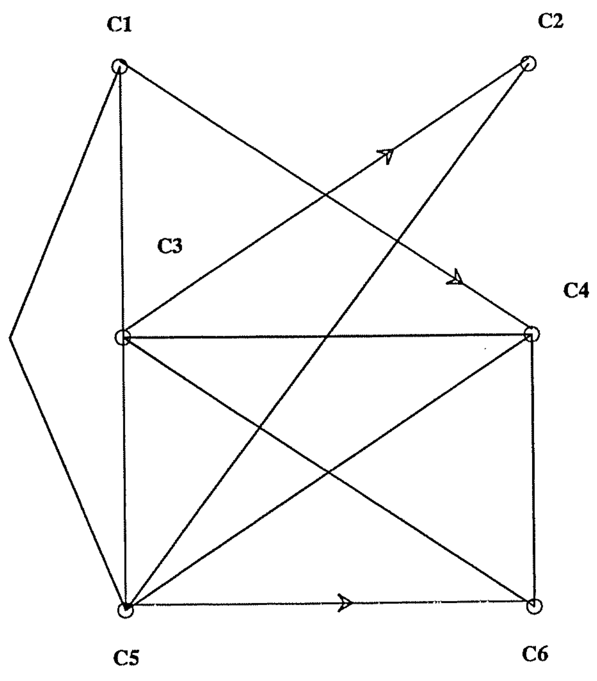


Figure 6. Graph with ordered folding set

```
procedure SerialFolder : set of edge;
const
        Edges = ... — set of edges of graph.
        Vertices = ... — set of vertices of graph
```

```
var
        Foldings : set of edge := {};
        Top : set of vertex := Vertices;
        Bottom : set of vertex := Vertices;
        u, v : vertex;
begin
label:
        while Top <> {} do
                v := TopSelect(Top); — returns element of minimal degree
                Bottom := {u ∈ Bottom | u <> v, (u,v) ∈ E};
                while Bottom <> {} do
                        u := BottomSelect(Bottom); — element of maximal degree
                        if Cycle(v,u) then
                                Bottom := Bottom – {u};
                        else
                                AddEdge(v,u); — update graph
                                Foldings := Foldings + { (v,u) };
                                Top := Top – {v,u};
                                Bottom := Bottom – {v,u};
                                exit; — inner while loop
                        end;
                end;
                Top := Top – {v}; — deletion from top column set
        end;
        return Foldings;
end SerialFolder;
```

This algorithm finds folding pairs as follows :

It first selects a vertex (v) which is of minimum degree and is not yet part of a folding pair. It then tries to find a vertex (u) of maximum degree (and not part of a folding pair) such that the folding (v,u) does not cause an alternating cycle in the graph. The graph is updated whenever a new folding pair is found.

The heuristic portions of the algorithm are represented by procedures TopSelect and BottomSelect.

The cycle detection portion of the algorithm is detailed below.

```
procedure Cycle(v,u) : Boolean;
var
        Answer : Boolean := false;
        Children, Parents : set of vertex;
begin
        Children = { t ∈ As | t ∈ ADJ(u)}; — of u
        Parents = { t ∈ As | v ∈ TRANS(t)} — of v
        return Children and Parents <> {};
end Cycle;
```

In this algorithm,

ADJ(u) is the adjacency set of vertex u:

$ADJ(u) = \{ v \mid \{u,v\} \in E\}$

TRANS(v) is the transitive closure set of vertex v of the mixed graph:

$TRANS(v) = \cup_{u \in AP(v) \cup At} ADJ(u)$

AP(v) is the set of all distinct alternating paths beginning at v.

$At = \{ t \mid s\ (s,t) \in A\}$

$As = \{ s \mid t\ (s,t) \in A\}$

A is set of directed edges in the graph.

The AddEdge portion of the algorithm updates the transitive closure of appropriate vertices in the graph.

2.4. Distributed folding algorithm

An important characteristic of the serial folding algorithm is that computation of each folding pair requires knowledge about all the folding pairs added to the graph till then. The distributed folding algorithm is composed of a variable set of folder processes (depending on the degree of parallelism desired) connected together as shown in Figure 7. We chose a circular structure of processes rather than a master-slave structure (represented as a star), since we feared that the master might become a bottleneck. The I/O process performs the task of reading in the data, initializing the graph structure representing the PLA and printing out the results when the program is done.

After the I/O process has read in the data, it initializes the graph structure and passes it on to its neighboring folder process. The folder process uses this data to initialize its local data structures and passes on the message to its neighbor. This pattern continues until all folder processes have initialized their data structures to represent the PLA being folded. The top column set, which is initially a set of all vertices in the graph, is divided equally among all the folder processes as follows. The vertices are numbered in increasing order of degree. Folder process i gets vertices v such that $v \bmod N = i$. The PLA shown earlier, if folded with three folder processes, would be distributed as shown in Figure 8.

We now describe the algorithm implemented by a folding process. Each folding process executes a three-step sequence for every vertex assigned to it.

(1) It computes all the possible foldings with the current vertex as the upper column (using the current state of the PLA graph).

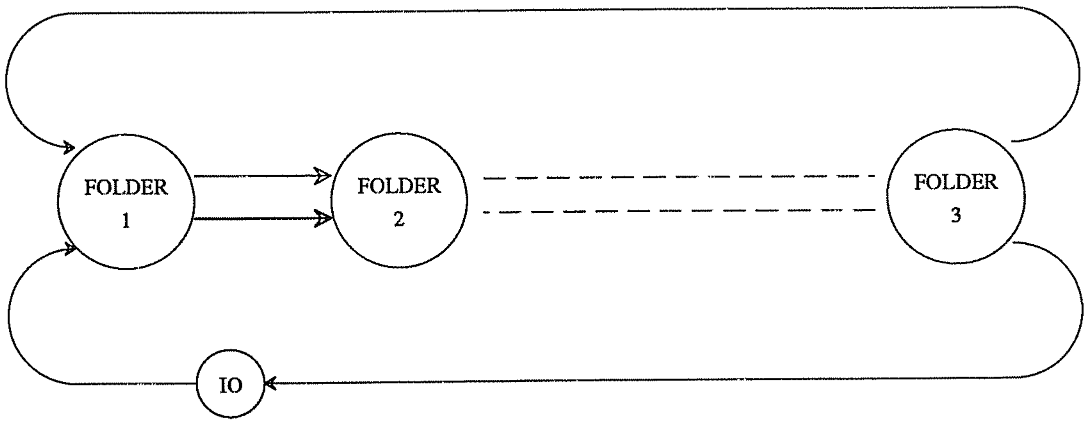


Figure 7. Structure of the distributed algorithm

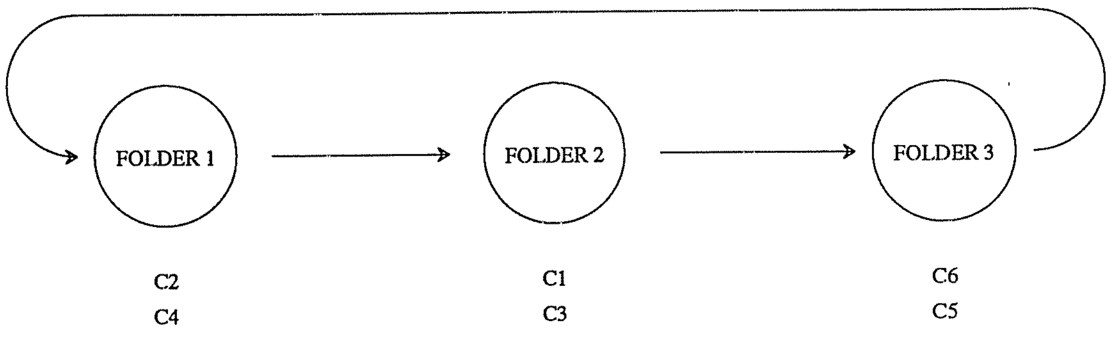


Figure 8. Distribution of vertices

(2) It receives and processes messages from the other N–1 folding processes (through its left neighbor). Each message consists of a folding pair (and some other information as detailed below). The message is forwarded to the right neighbor unless the neighbor happens to be the originator of the message. Processing the message involves pruning the local foldings that are either redundant or will cause a cycle due to the addition of the directed edge mentioned in the message. An incremental cycle detection algorithm is used to detect cycles. It makes use of the fact that the cycle could have been created only by the addition of the directed edge mentioned in the message. Each message received also causes the local graph to be updated to reflect the addition of a directed edge.

(3) The "best" (in the sense of the heuristic algorithm of section 2) local folding remaining after the pruning of step 2 is then chosen. The others are discarded. The chosen folding is sent to the right neighbor after the local graph structure is updated. The algorithm terminates when each process has finished the set of vertices allocated to it.

The following is a more detailed description of this algorithm.

```
const
        U = set of all vertices in graph;
        NumberOfNodes = ... ;

var
        Foldings : set of (edges, parents, children) := {};

process Folder(FolderID, NumNodes);
var
        VertexSet = set of vertex := all vertices assigned to this node;
        CurrentVertex : vertex;
        Count : integer;
begin
        while VertexSet <> {} do
                CurrentVertex := VSELECT(VertexSet);
                VertexSet = VertexSet - { CurrentVertex };
                Foldings = ComputeFoldings(CurrentVertex);
                for Count := 1 to NumberOfNodes do
                        accept an incoming message, with (v,u) pair.
                        forward message to neighbor on right
                                unless that neighbor originated it;
                        AddEdge(v,u);
                        PruneFoldings(v,u);
                end;
                propagate Foldings;
        end;
end Folder;
```

```
procedure ComputeFoldings(v : vertex) : set of edges;
var
        Answer : set of (edges, parents, children) := {};
        Bottom : set of vertex;
        u : vertex;
begin
        Bottom = { u ∈ U | u<>v, {u,v} ∈ E }
        while Bottom <> {} do
                u := BottomSelect(Bottom);
                if not Cycle(v,u) then
                        FoldCount +:= 1;
                        insert ((v,u), v's Parents, u's Children) in Answer;
                end;
                Bottom -:= {u};
        end;
        return Answer;
end ComputeFoldings;

procedure PruneFoldings(var v, u : vertex);
begin
        foreach e in Foldings do
                if v=e.v or u=e.u then
                        Foldings -:= {e};
                else
                        if e.v in TRANS(v) then
                                e.parents +:= msg.parents;
                        endif;
                        if v ∈ ADJ(e.u) then
                                e.children +:= {v};
                        endif;
                        if e.parents and e.children <> {} then
                                Foldings -:= {e};
                        endif;
                endif;
        end;
end PruneFoldings;
```

2.5. Implementation using Lynx

The Lynx program implementing the algorithm consists of two modules: **I/O** and **folder**.

The I/O module has three functions:

(1) Read a PLA personality matrix.

(2) Initialize the graph structure and send it to the neighboring folder process, which will propagate it.

(3) Gather results from the last node and print them out.

The folder is a straightforward Lynx implementation of the algorithm described in the previous section. The folder gets the graph structure corresponding to the PLA being folded. It also receives a set of vertices it should use as upper columns in the foldings it computes. An entry is defined for receiving the initialization data from its neighbor on the left. Another entry is defined to receive messages from the left neighbor during the course of the algorithm (corresponding to the ProcessMessage part of the algorithm). This entry is bound to the right link of the neighbor. The main loop of the process consists of three steps as explained in Section 4.

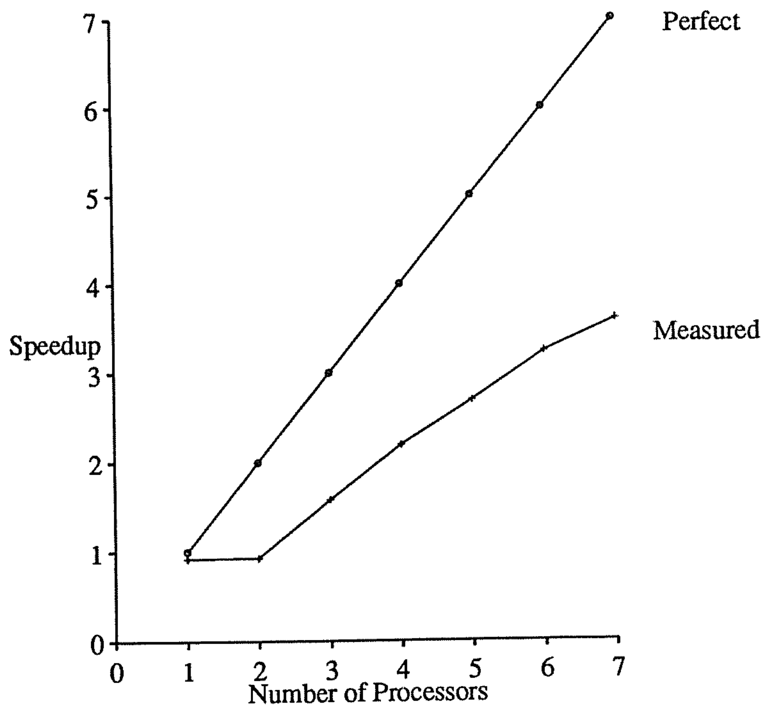
Each folder executes until it has emptied its vertex set. It then tells its right neighbor. When all vertex sets are empty, the algorithm is done.

At the moment, the program is limited by stack space to PLAs with fewer than 300 rows and 150 columns.

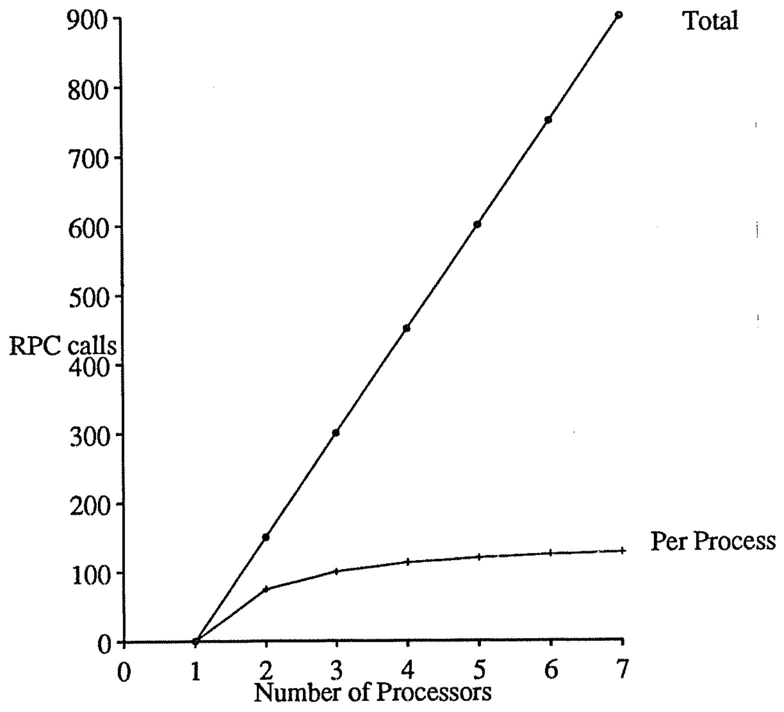
2.6. Performance results

The Lynx program described in previous section was run on Charlotte. Even though the program was run on a number of PLAs, we now describe the test results obtained by running the program on a 300×150 PLA. The entries in the PLA personality were generated by using the Unix pseudo random number generator "random()". An entry in the PLA personality was 1 with a uniform probability of 0.035. The number of rows and columns of the PLA and its density were chosen so that the serial program (also implemented in Lynx and run on one Crystal node) would take sufficient time for speedup measurements to be meaningful. The serial program took 28.91 seconds of CPU time. (This was the time needed to execute the two main loops of the algorithm in Section 3.)

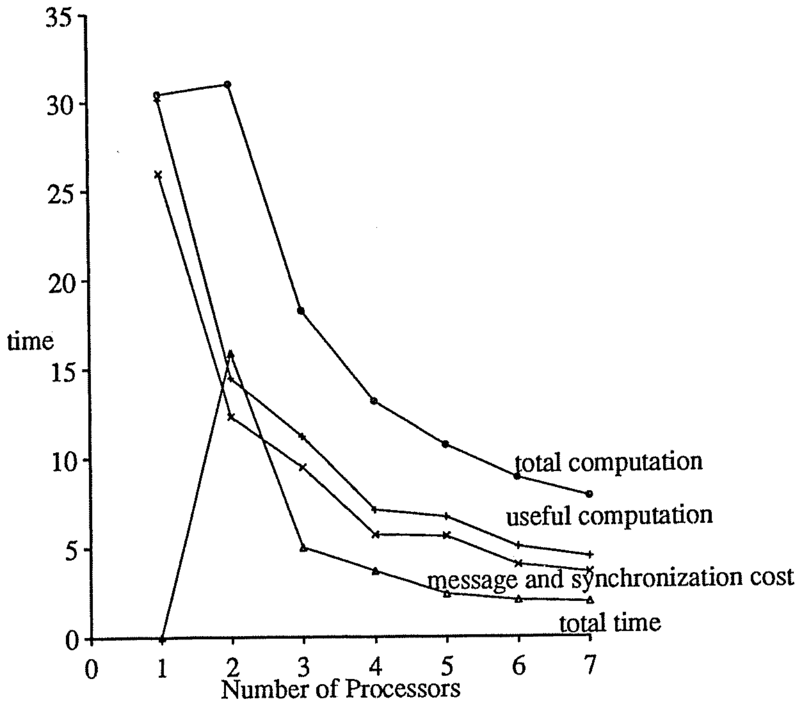
The speedup curve for the distributed algorithm is shown in Graph 1. Speedup is the ratio of serial execution time to parallel execution time. The speedup is plotted with the number of processors varying from 1 through 7. In all cases, each machine had only one folder process. Graph 2 shows the variation of total number of remote procedure calls (the basic communication primitive in Lynx) and number of remote procedure calls (RPC) made by each process with increase in degree of parallelism. Graph 3 shows the amount of time spent in the different tasks performed by each process. Graph 4 shows the percentage of total time spent in each of these tasks. These graphs were plotted using statistics generated by one folding process. Since the algorithm is symmetrical with respect to folding processes, this graph should be typical for all folding processes. "Total computation time" represents the time spent by each processor in computing foldings. The time required to prune foldings was not considered, as it was negligible compared to the time required to compute foldings. Since all possible foldings are computed for each vertex assigned to the process, and only the best (in the sense of the heuristics of Section 3) is chosen, some of the work done will be wasted. The "useful computation" graph represents the minimum work needed to find the best folding.



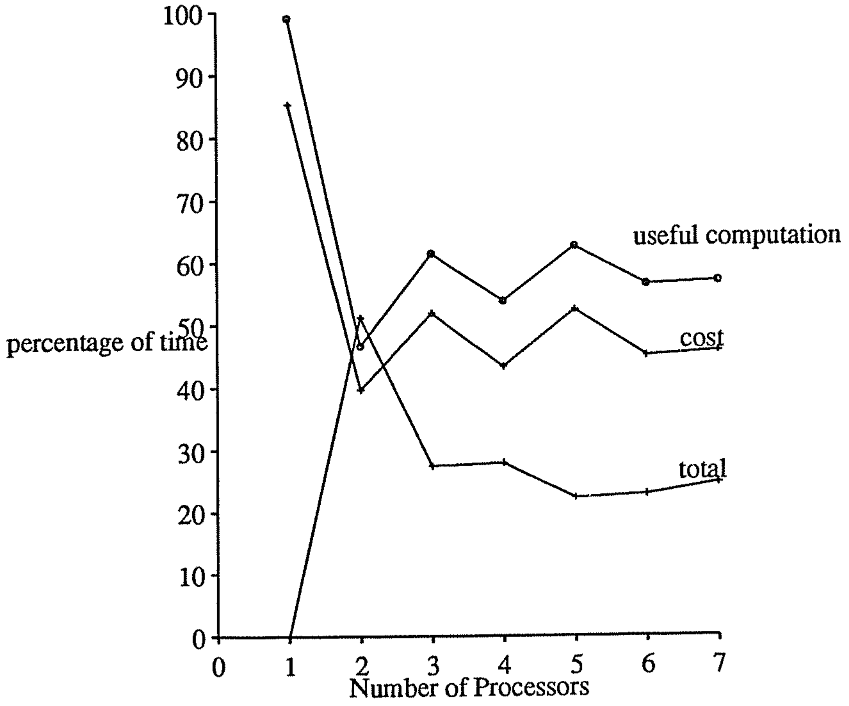
Graph 1: Speedup



Graph 2: Number of RPC calls



Graph 3: Breakdown of time



Graph 4: Breakdown of time by percentage

The cost for remote procedure calls (synchronization and communication costs in Graph 3) decreases with increase in degree of parallelism, even though the number of remote procedure calls increases (as shown in Graph 2). This anomaly can be explained by a peculiarity of the Lynx implementation. Each RPC involves sending a message and receiving a reply. It also results in creation of a new thread in the receiver. However a context switch occurs only on blocking, that is, the receiver gets to send a reply only when its current thread blocks. When the degree of parallelism is low, it is more probable that the receiver process is engaged in some computation when it receives an RPC. Hence it takes more time to service the RPC, explaining the higher synchronization and message cost for lower degrees of parallelism.

Graph 1 shows that speedup is almost linear (with a negative offset and a slope of 0.5). There is almost no speedup obtained for a parallelism of 2, because a large percentage of the time (about 50%) is spent in synchronization and communication. As the degree of parallelism increases, the amount of useful computation done by each processor falls and the percentage of time spent in communication and synchronization also falls. This reflects the speedup obtained.

2.7. Experience using Lynx

The concept of threads of control and policy of context switching only on blocking makes it very easy to write synchronization protocols using Lynx. The author would have preferred a better I/O facility, for example, one akin to that provided by C. A faster Lynx compiler would greatly help.

2.8. Conclusions and future work

This work demonstrates that control synchronism and resource usage are important factors in determining the execution time and speedup of a parallel program.

The performance results can be used to tune the program. It may be a good idea to reduce the number of messages by letting each processor compute n foldings before communication with others (n is 1 at the moment). This may, however, result in a greater percentage of the work done by each processor going to waste. It will also be interesting to compare the distributed algorithm presented here with one having a master-slave structure.

3. Travelling Salesman

Experimenter: Igor Steinberg

The Travelling Salesman Problem (TSP) is well known to computer science researchers. Given a weighted directed graph G = (V,E), the problem is to find the minimal length tour of all the vertices in V. In general, no restrictions are imposed on the set of edges, E. TSP is NP-complete[Garey79a]. We concentrated, therefore, on a high-quality heuristic algorithm.

3.1. The algorithm

The algorithm used to solve TSP is as outlined by Lin and Kernighan[Lin71a] The algorithm works as follows:

(1) A random tour, T, is generated.

(2) The algorithm attempts to identify a set of edges, X, in T and a set Y in (E–T), such that replacing the set X with the set Y results in new tour, T′, of lower cost than T.

(3) Step (2) is repeated until the algorithm can no longer find a feasible set Y.

(4) When no further improvements can be made, the algorithm halts and reports its solution. Such a solution is termed a "local optimum".

(5) Steps (1) through (4) are repeated as many times as desired by the operator. The best local optimum is then reported as the result.

This algorithm is not guaranteed to find the best tour, but it is very fast. The observed running time for the algorithm is close to quadratic.

3.2. Implementation

The algorithm was distributed by allowing the computations of the different local optima to occur in parallel. A Queen process coordinated the work of many Worker processes. The Queen process is just a housekeeper. It first reads the graph and other data from a text file. These "other data" include a number specifying how many Worker processes should be spawned. The Queen then the Workers and provides each with a copy of the graph. The Queen also provides each Worker with a random seed used to generate initial random tours. As each Worker reports a new tour, the Queen either instructs the Worker to continue from a new initial random tour or to terminate. The new local optimum tour is broadcast by the Queen to each of the other Workers. This added information allows the Workers to effectively avoid checkout time. Avoiding checkout time will be discussed later in the report.

The function of the Worker is to generate a random tour, modify it until a local optimum is reached, then report the result to the Queen. Each Worker runs the algorithm described by Lin and Kernighan for one iteration. The reader is referred to that paper for a precise description of the algorithm. We present only a sketch of the method here. An edge of the initial tour is broken, and a search evaluates the benefit of healing the wound by adding various new edges, each of which will cause new wounds, which are themselves healed by a similar method. The most beneficial healing method is chosen, and then a new edge is chosen for breaking. This process continues until no matter which edge is broken, no healing can improve the tour. The phase of the algorithm that fruitlessly tries breaking each edge is called "checking out the tour". Lin and Kernighan claim that it consumes as much as 50% of the total computation, and our experience supports this claim.

Our implementation includes some of Lin and Kernighan's refinements:

(1) Avoiding checkout time. (Before breaking a new edge, the tour is compared with other local optima, and if it is identical, the entire checkout process is avoided.)

(2) Lookahead.

(3) Partial reduction.

These features tend to improve the running time of the algorithm. We did not implement these refinements:

(1) Nonsequential Exchanges.

(2) Breaking the feasibility criterion, by choosing the alternate X2. This is described by the authors in step 6(b). This step was avoided because including it would have made the program much more difficult to debug.

These features tend to improve the quality of the solution found by the algorithm.

3.3. The Lynx program

The Queen process contains the following entries:

inform()	This entry accepts results from the workers. It invokes entry broadcast() and also informs the operator of the results of the program before termination.
broadcast()	This entry connects to the checkout() entry of the Workers. It is used to transmit the results of one Worker to all of the others.

The Worker process contains the following entries:

startup()	This entry is invoked once by the Queen. It accepts the cost matrix as well as other initializing data from the Queen.
checkout()	This entry accepts information broadcast from the Queen. A tour is represented by a hash value. This value is inserted into an ordered list for future searching by the worker() entry. Using a hash value instead of the actual tour saves in communication cost and search cost, but can lead to erroneous pruning of new tours.
worker()	This entry runs the Lin/Kernighan algorithm.

In addition to the graph, the data file contains 20 debugging flags. This feature allows the operator to trace execution selectively. The Lynx compiler is slow, so using run-time debugging checks saved a great number of compilation hours.

3.4. Experiments

The program was tested with two kinds of data. The first set of data used was random, with intercity distances uniformly distributed. The second set was specifically designed to make the algorithm perform poorly. Each set of data was run for graphs of 8, 12, 16, 20, and 24 vertices. Each graph size was run using 1, 2, 4, and 8 Worker processes. We measured the amount of time needed to generate 20 local optima. We also measured the number of checkouts avoided. To see how much time was actually saved, we also ran problems with checkout avoidance turned off.

Each Worker ran on a separate machine. The Queen also ran on its own machine, except in the case of 1 Worker, in which both processes executed on the same machine. The Queen was placed on its own machine only to save it from a greedy Worker process. The Queen could become a bottleneck if it shared a machine with a Worker, because the Worker would use most of the available CPU resource. This problem could be solved elegantly if the Queen could run at higher priority, but Charlotte and Lynx do not have this facility.

As mentioned previously, the non-random data were designed to make the algorithm perform poorly. The data were constructed so that each vertex had, as its six closest neighbors, vertices 0 .. 5 (in order of closeness). The algorithm constructs two sets of edges (X and Y) that are swapped if replacing the X edges by the Y edges results in a tour of lower cost. The algorithm sequentially breaks an X-edge and tries to add a Y-edge to the tour from one of the vertices with the broken X edge. The new Y edge has a vertex in common with the X edge. When searching for a suitable Y edge, the algorithm only considers the shortest k edges from this common vertex. (The parameter k is set in the data file; we used a value of 6.) The algorithm performs quite poorly if the number of vertices is much greater than k.

3.5. Experimental results

Table 1 shows the results of running our algorithm using the random data. In each case, the queen directed that 20 local optima be found. Some of these turned out to be duplicates, particularly for small graphs; the table shows the number of local optima that were actually global optima. The times (given in seconds) are reported both when checkouts are avoided and otherwise, along with the associated speedups.

cities	workers	global optima	checkouts avoided	time	speedup	full time	full speedup
8	1	20	19	27.436	1.00	52.626	1.00
	2	20	18	16.648	1.64	28.215	1.86
	4	20	16	11.289	2.43	15.871	3.32
	8	20	12	7.357	3.73	9.786	5.38
12	1	11	15	45.149	1.00	75.964	1.00
	2	7	14	26.194	1.72	41.236	1.84
	4	8	12	14.757	3.06	20.054	3.78
	8	9	7	10.944	4.12	12.836	5.91
16	1	10	16	85.995	1.00	129.481	1.00
	2	7	11	47.917	1.79	62.073	2.09
	4	9	14	26.399	3.26	35.511	3.65
	8	9	7	17.485	4.92	19.656	6.59
20	1	6	12	108.218	1.00	148.055	1.00
	2	7	9	62.837	1.72	76.449	1.94
	4	3	6	33.844	3.19	38.995	3.79
	8	4	3	21.781	4.97	24.612	6.01
24	1	1	9	170.139	1.00	207.553	1.00
	2	2	8	88.313	1.93	104.732	1.98
	4	1	6	43.765	3.88	49.613	4.18
	8	1	2	28.474	5.97	29.997	6.92

Table 1: Random data

The speedup obtained by distributing the algorithm appears to be directly proportional to the number of processors. The speedups are worse when checkouts are avoided, because a checkout that might have been avoided is often finished (or well underway) by the time a message arrives obviating it. The table shows how the number of checkouts avoided falls with the number of workers. However, it is always beneficial to avoid checkouts.

The data suggest that the time required to solve a problem increases linearly with the size of the problem. If one considers how many of the 20 computations were really needed, however, then the time grows faster than that. For example, one worker solving a 16-city problem finds the best solution in 10 of 20 attempts, while it finds the best only once at 24 cities. As the size of the problem increases, the number of attempts should increase. The Queen could have the Workers continue until no significant improvements have occurred for a while.

The checkout-avoidance savings fluctuate. For some problems the savings appear to be quite substantial, while for others this is not so. The checkout-avoidance savings would likely increase significantly if the unimplemented features were added, since they tend to improve all local optima, and so more high-quality tours would be found.

The true optimal costs for a few of the problems have been found by using DIB[Finkel85a] In each case, our heuristic algorithm found the optimal solution. The largest verified problem is the (random) 12-city problem.

Table 2 shows similar results for the non-random data. Far fewer checkouts were avoided, because many different tours gave the same result. The speedup results for both the random and the non-random 24-city problems are shown in Graph 5.

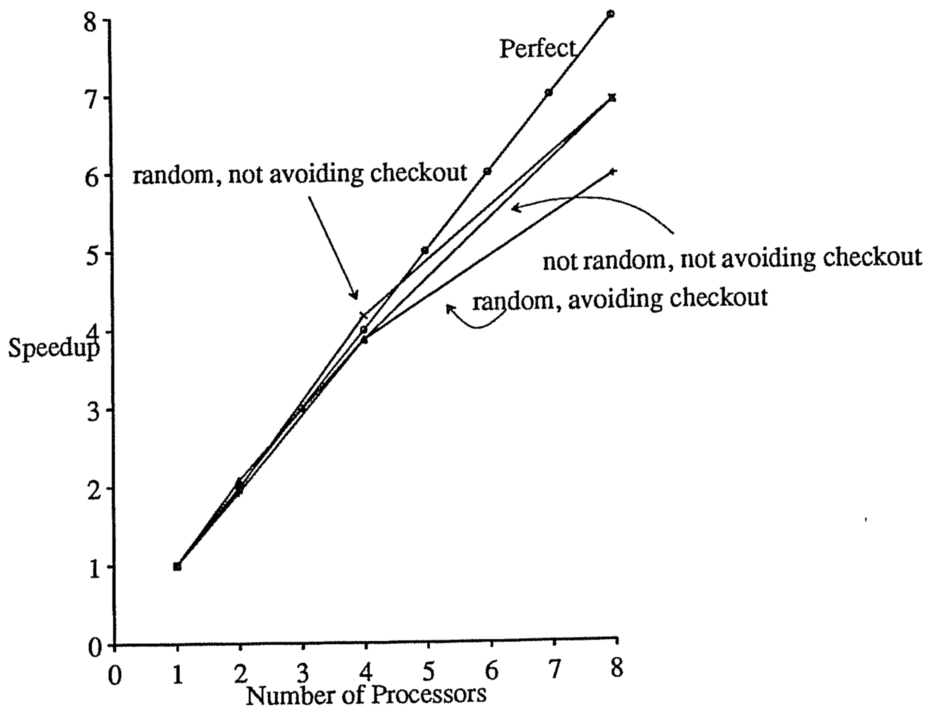
3.6. Comments on Lynx

Lynx is a wonderful programming language. It made implementing this algorithm really quite easy. As is the case with any software, it can be improved in some ways.

- It would be nice if Lynx had built-in file I/O.

cities	workers	global optima	checkouts avoided	time	speedup
8	1	20	7	38.405	1.00
	2	20	3	22.704	1.69
	4	20	0	12.621	3.04
	8	20	0	7.357	5.22
12	1	20	0	110.227	1.00
	2	20	0	61.044	1.80
	4	20	0	32.888	3.35
	8	20	0	16.196	6.80
16	1	14	0	222.971	1.00
	2	12	0	124.137	1.79
	4	13	0	62.584	3.56
	8	13	0	37.826	5.89
20	1	6	0	335.107	1.00
	2	9	0	176.838	1.89
	4	8	0	95.683	3.50
	8	4	0	61.526	5.44
24	1	1	0	469.453	1.00
	2	1	0	224.824	2.08
	4	1	0	120.798	3.88
	8	1	0	67.754	6.92

Table 2: Non-random data



Graph 5: Speedup for the 24-city problem

- It would also be nice if Lynx had a cast operator (like C).

- Any spoiled UNIX programmer would also like Lynx to have a debugger like dbx.
- Lynx should allow the programmer to specify (with compile-time switches) both the amount of stack space to be reserved and the size of the process' link table. I encountered what I believe to be unreasonable limits for both, and this severely limited the size of problem and the number of Workers that I could run.
- As previously noted, Lynx and Charlotte should let the programmer/user specify the run-time priority levels for individual processes.
- Lynx should also let the programmer specify priority levels for different entries within a process. When broadcasting from the Queen to each Worker, it would be nice to interrupt each worker process. This would allow some entries to act as interrupt handlers. Currently, **await** statements are used to achieve a similar effect. This is much less efficient and much more cumbersome than the code that would result from priorities. The run-time cost of **await** should not be underestimated. I could not obtain precise timings, but I believe that a few **await** statements account for 25-40 percent of the program's running time. These statements are not in the program's innermost loops.

3.7. Acknowledgement

We would like to acknowledge the help of Udi Manber, who determined the optimal costs for a number of problems using DIB[Finkel85a]. This information not only confirmed that the algorithm was performing well, it also helped assure us that the algorithm was running correctly. A heuristic algorithm is very difficult to debug simply because one cannot tell whether or not the generated result is correct. A reasonable answer might be the solution that should be produced by the algorithm, or might possibly be due to a subtle bug in the program.

4. Minimal Spanning Tree Updates

Experimenters: Donald Nelson and Ramesh Polisetty

A spanning tree of a connected graph can be defined as a connected subgraph containing all the vertices of the graph but no cycles. Hence, a spanning tree gives a unique path between any pair of vertices. The weight associated with each edge represents a cost defined by the application. Examples of such costs are transmission costs, time delays and distances. A minimal spanning tree (MST) is a spanning tree with the least total cost. In applications such as distributed databases where broadcasting is a frequent occurrence, MST information proves to be very useful in order to minimize the transmission cost associated with each of the channels. As the topology of the network changes, there is a need for incrementally updating the MST information.

In this project, as in the one reported in the first Experience report[Finkel86a], the distributed algorithm by Gallager, Humblet and Spira[Gallag83a] was used as the basis for constructing the MST. The MST information is distributed, with each vertex of the network keeping track of its incident edges that are part of the MST. Starting from this MST, we implemented the distributed algorithms suggested by Chen[Chen85a] for updating the MST information as the edge weights of the graph increase or decrease. These update algorithms require only a subset of the vertices to participate in restructuring the MST, and hence are more efficient than rebuilding the MST from scratch. By making use of a locking scheme in each vertex and providing a deadlock detection mechanism, the algorithms for updates are allowed to proceed concurrently to enhance their performance.

4.1. Construction of the MST

A fragment of an MST can be defined as a connected subgraph of the MST. For an n-vertex graph, Gallager's algorithm starts with n single-vertex fragments, each performing the same local algorithm. Vertices in a fragment cooperate to find the minimum weight outgoing edge leading to a vertex in another fragment. Each fragment is identified by a special edge called the **core** edge. The two fragments can then be combined into a larger fragment and allowed to find the subsequent minimum weight outgoing edges. The algorithm terminates when there is a single fragment, which is the MST. At the end of the algorithm, each vertex knows which of its incident edges are in the MST. The two vertices at each end of the final core edge recognize the termination of the MST algorithm and initiate the computation of the subtree vertex sets (to be described in the next section).

4.2. Incremental edge updates

In a spanning tree, every non-tree edge defines a basic cycle. The basic cycle contains the non-tree edge itself and the set of tree edges connecting both ends of the non-tree edge. For a given tree to be an MST, a necessary and sufficient condition is that that every non-tree edge have the highest cost on the basic cycle it defines. The effect of an edge change is summarized as follows:

- A tree edge, when its cost increases, may be replaced by the non-tree edge with the lowest cost whose basic cycle contains the tree edge.
- A non-tree edge, when its cost decreases, may replace the tree edge with the highest cost in the basic cycle.

In either case, the action to be taken is to replace a tree edge by a non-tree edge in the MST. All other changes will not affect the MST.

To efficiently support the maintenance of an MST, a data structure called the **subtree vertex set** is used. A vertex has a subtree vertex set for every incident tree edge. These sets contain a list of all the vertices in the subtree across that tree edge. A destination can be reached from any place in the network by following the tree edges whose subtree vertex sets contain the desired vertex. Subtree vertex sets are also useful during the search phase of an incremental update for determining if an edge connects two fragments.

The **search phase** is the first of the two phases of the incremental algorithm. The two cases that might affect the MST are increased tree edge cost and decreased non-tree edge cost.

- Increased tree edge cost. If the tree edge whose cost is increased is removed, two fragments of the MST are formed. One of the vertices of the affected edge, (the one with the fewer vertices in its fragment or the one with smaller vertex id in case of a tie) initiates the process to determine the minimum weight outgoing edge from the fragment by passing SearchMin messages and accepting ReportMin messages. If this edge is different from the tree edge, a new MST is formed by exchanging the minimal edge and the tree edge.

- Decreased non-tree edge. The SearchMax message is routed through the vertices in the basic cycle of the non-tree edge. The message returns with the highest cost edge in the basic cycle. If this edge is different from the non-tree edge, then an exchange is initiated between the maximum cost tree edge and the non-tree edge.

If an exchange has been decided in the search phase, the algorithm enters the **update phase**. During this phase, each of the two vertices defining the non-tree edge (which is entering the MST) mark this edge as belonging to the MST and send an update message to the neighboring vertices in the basic cycle. The update message contains a list of vertices of the subtree across the edge being removed from the MST. Vertices receiving this message update their subtree vertex sets using this list and forward the message to the adjacent vertex in the cycle. When the two vertices defining the tree edge (which is leaving the MST) receive the update message, they adjust their vertex sets and remove the tree edge from the MST to complete the exchange.

4.3. Concurrent edge updates

It is very common to have concurrent updates in a real network. However, any particular vertex cannot participate in more than one edge update operation at a time. In order to resolve this conflict, an exclusive lock can be used to serialize the updates at any single vertex. This lock is set whenever there is an active update process involving the vertex.

Deadlock can occur when two update processes on each end of an edge are holding the lock the other needs. This deadlock condition can be detected by storing (in the lock) the set of vertices the current update process will traverse next. If the update process detects that the vertex it is visiting has already been locked and the vertex it last visited is one of the next vertices recorded in the lock, a deadlock situation has occurred. The update with higher priority is allowed to continue while the one with lower priority backs up to the initiating vertex. If there is no possibility of a deadlock, an update process merely waits for the lock to be released.

When the update phase at a vertex is completed, the lock is released. At this point, a waiting update process or an update process originating at that vertex will resume.

4.4. Implementation

Our implementation of the distributed algorithm uses two modules, the IO (input/output) module and the Vertex module. At startup, the IO process is connected to each of the vertex processes and to the Charlotte switchboard, through which the Charlotte fileserver is found. Each vertex process is initially connected only to the IO process.

We implemented these algorithms in Lynx, using remote procedure calls instead of message passing for interprocess communication. Each remote procedure call is counted as a "message" in the results we will present.

4.4.1. The IO process

The IO process is the interface between the user and the vertices of the graph. This module obtains a link to the fileserver, reads the graph information from the input file and initializes the Vertex modules by sending the graph information to each of the vertices. A graph edge is represented by two vertices and an edge cost. The IO process creates a new link to represent each edge, and passes the edge information to both the associated vertices, giving one end of the link to each. After the entire graph is established, the vertices are awakened by the IO process to start executing the distributed algorithm to construct the MST. At the end of the MST computation, the IO process receives the local MST view from each of the Vertex processes and sends the complete MST information to the screen.

The edge updates are then read from the input file and the two affected vertices are informed about the change. After the completion of the incremental update algorithm, the IO process receives the result of the update from one or both of these vertices, and, if necessary, updates its MST information. In order to execute the updates concurrently, the IO process assigns a sequence number to each of the updates so that older updates could be given preference in case of a deadlock. (Concurrent updates were not completely implemented). In our implementation, the IO process also receives the information about the subtree vertex sets from each of the vertices facilitate debugging.

4.4.2. The Vertex process

Vertex processes collect the incident edge information from the IO process. Once awakened by the IO process, they start the distributed algorithm for MST computation. The termination of the MST computation is recognized by the two vertices of the core edge. These vertices initiate the computation of subtree vertex sets by sending messages outwards, towards terminal vertices.

Computation of vertex sets follows a classical ''edge-in'' structure: Each vertex that has heard from all but one of its neighbors sends information to the missing neighbor, and when it hears from that last one, it sends information to all the others. Terminal vertices, which by definition have but one neighbor, can send information immediately. In our case, the information that is sent is the set of all vertices that can be reached via all neighbors except the one to which the information is sent.

All update requests involving a particular vertex are received by that vertex and are stored in a priority queue of pending updates ordered by sequence number. They are removed from the queue whenever the lock becomes free. The type of the edge change is determined by noting whether the edge belongs to the MST and whether the cost has increased or decreased. Depending on the two cases of the updates, SearchMin or SearchMax messages are sent to the concerned vertices. Each vertex upon receiving a search message tries to set the lock of the vertex if it is not already set. If it is set and there is no possibility of a deadlock, the search process waits. If there is a deadlock, the low priority update (the one with higher sequence number) backs up vertex by vertex until it reaches the initiating vertex. All the locks set by this update are released as it backs out, and the update is put back into the pending queue.

In the case of a tree-edge update, a certain amount of pruning of the search occurs while nodes send SearchMin messages outward. If the edge weight of the next edge is greater than or equal to the current minimum edge weight found so far, a SearchMin message is not sent over that edge. While nodes receive ReportMin results, if the minimum edge weight received so far is smaller than all the weights of edges yet to report, subsequent ReportMin results are ignored. This pruning process does not affect the correctness of SearchMin because of the properties of basic cycles in minimum spanning trees.

For any update, only one of the two affected vertices will initiate the search phase. At the end of the search phase, the edge that should be the part of the MST is determined. If this edge is not already in the MST, an update phase is initiated to update the subtree vertex sets of the vertices along the basic cycle. In the case of a decrease in the non-tree edge weight, all the vertices in the basic cycle are locked during the search phase. When a tree edge cost increases, only half the vertices of the basic cycle are locked during the search phase, and the update must wait until all the vertices in the basic cycle are locked. Finally, the two vertices corresponding to the changed tree edge inform the IO process of the edge which was changed in the MST.

4.5. Testing and results

Testing was performed in two areas:

(1) Performance of incremental updates (Chen) versus recalculating the entire minimum spanning tree (Gallager), and

(2) Speedup as more processors were used.

Both performance and speedup were measured with respect to the number of messages and time needed.

4.5.1. Incremental versus full recalculation

Graph sizes of 4, 8, and 16 vertices were tested. Within each graph size, edge densities of 50% and 100% were considered. Three graphs with four edge updates each were run to determine each data point.

Sequential updates to the minimum spanning tree were used to generate the data. The IO module waits until it has heard from the last two vertices involved in the search or update process before reading in a new edge change, so there is no concurrency between updates. The edges were randomly chosen. Edge weights were random numbers uniformly distributed between 1 and 500.

Four edge weight changes were randomly generated for each graph. Only some of these changes affected the MST. For each of the updates that modified the MST, incremental updates were performed. and also the entire MST was recalculated. The times and number of messages for these updates and recalculations were averaged to produce the data below. In addition, we measured the cost of calculating the vertex sets, before any updates were considered.

The results from graphs with 100% edge densities are given in Table 3.

Vertices	Algorithm	MsgCount	Time in ms
4	Rebuilding MST	37	140
	Incremental Update	8	121
	Initial VSet Calculation	12	143
8	Rebuilding MST	129	307
	Incremental Update	10	165
	Initial VSet Calculation	28	304
16	Rebuilding MST	373	753
	Incremental Update	14	189
	Initial VSet Calculation	60	828

Table 3: Average MsgCounts and Time for 100% connected graphs

The test results using graphs with 50% edge densities are shown in Table 4.

Vertices	Algorithm	MsgCount	Time in ms
4	Rebuilding MST	31	162
	Incremental Update	8	133
	Initial VSet Calculation	12	158
8	Rebuilding MST	95	239
	Incremental Update	8	183
	Initial VSet Calculation	28	306
16	Rebuilding MST	305	602
	Incremental Update	13	221
	Initial VSet Calculation	60	814

Table 4: Average MsgCounts and Time for 50% connected graphs

Both the timings and the number of messages are significantly reduced in Gallager's algorithm when the edge density is reduced. This is expected with respect to the number of messages, since its behavior is $O(n \log n + e)$. However, the expected time complexity of Gallager's algorithm is $O(n \log n)$. In all other aspects, the performance does not vary noticeably with respect to the edge density.

The time taken to perform all vertex-set computations is dominated by the initial vertex set computation. It takes as much time to calculate the initial vertex sets as it does to calculate the entire minimum spanning tree for a fully connected graph. This result can be understood by considering the number of entries called at each vertex during the vertex set calculation. Every vertex (except the core vertices) communicates with each MST neighbor three times during the vertex set calculation.

There is no doubt that incremental updates are much more efficient than recalculating the entire MST. The discrepancy becomes more apparent as the size of the graph increases. This is to be expected, since incremental updates limit their involvement to a single cycle in the graph, whereas the entire graph is involved with calculating the MST. When the graph is small, the cycle is more likely to involve a larger portion of the graph.

4.5.2. Speedup

Speedup measurements were performed on three 16-vertex graphs with 100% edge density. Because of Charlotte limitations, a bigger graph could not be attempted. The graphs were run using 4 and 8 charlotte nodes. Again, because of Charlotte restrictions, more nodes could not be attempted. The speedup measurements for Gallager's and Chen's algorithms are given in Table 5:

Charlotte Nodes	Time Taken (in msec)			
	100% density		50% density	
	Gallager	Chen	Gallager	Chen
4	1034	179	705	199
8	753	189	602	221

Table 5: Timing results for 16-vertex graphs.

The cost for incremental updates increased somewhat when more computers were used. This indicates that the running time of Chen's algorithm is so little that its performance is dominated by communication costs even with our largest graphs. However, a 15-25% reduction in running time is realized by doubling the number of computers used to calculate the entire Minimum Spanning Tree using Gallager's algorithm.

The time taken to calculate the initial vertex sets did not change significantly as a function of parallelism, although the times were large enough that communication costs did not dominate its performance (about 900 msec).

4.6. Experience with Lynx

- Too much time had to be spent finding out the details of performing IO. More documentation on this subject should be provided to aid the programmer.
- Better IO handling facilities should be provided. It would be much more convenient to be able to use routines similar to printf and scanf.
- Many of the limitations of Charlotte are poorly documented. A few examples are restrictions in the number of links allowed by any one process and limits in the number of links that any user can use. Much time and effort would be saved if such parameters were easily accessible.
- The lynx language provides an elegant way to express and implement a distributed algorithm.
- Debugging a distributed program can sometimes be a very difficult process, especially when deadlock is concerned. A distributed debugger would be a very useful tool in these situations. We did not use TAP[Gordon85a], but it might have been helpful.
- A convenient way to fork new processes would allow more efficient initialization. Alternatively, we could have processed the graphs into Charlotte connector files and used existing software.
- In order to avoid recompilation for running the program on graphs of different sizes, we had to connect the extra links in the header of the IO process to the links of a series of dummy processes. Alternatively, we could have had them linked to each other. What Lynx really needs is a way to permit sets of links to be passed to processes by the connector.

4.7. Pseudo-code for concurrent updates of MST

```
process IO(SBlink, link1, link2, link3, link4, link5, link6, ... linkn:link);
— SBlink is a link to the switch board and
— each link–i is a link to i th vertex.

type
    weightType      = integer;                 — real, integer etc.
    — matrix to represent the graph connection
    row = array [1 .. MAXNODE] of weightType;
    matrix = array [1 .. MAXNODE] of row;
    VsetType = set of [1 .. MAXNODE]; — set of vertices in the graph
    Vrow     = array [1 .. MAXNODE] of VsetType;
    VsetMatrix = array[1 .. MAXNODE] of Vrow;
```

```
var
    graph : matrix;
    MST : matrix;
    Vsets : VsetMatrix;

    procedure InitGraph;
    begin
        — Read in Graph
        — Create a link for every edge and send them to
        — the corresponding vertices.
        — Wake up each vertex process to begin the
        — Galleger's algorithm.
    end InitGraph;

    entry RecordMST(NodeId : integer; EdgeArray : row);
    begin
        <update MST array>
        if <all vertices heard from> then MSTdone := true; end;
    end RecordMST;

    entry RecordVset(VertID : integer; VsetArray : Vrow);
    begin
        < update Vsetarray>
        if <all vertices heard from> then VsetDone := true; end;
    end RecordVset;

    procedure ReadEdgeChange;
    — Read all the Updates until the end of file
    — and send them to the corresponding vertices.
    begin
        await(MSTdone and VsetDone);
        SeqNo := 0;
        loop
            if eof then
                exit;
                RecordCount :=2;
                seqNo := SeqNo +1;
                ReadEdge(v1, v2, edgeWt);
                if v1<–>v2 is a new edge then
                    EndA:= newlink(EndB);
                else
                    EndA:= nolink;
                    EndB:= nolink;
                    connect EdgeChange(V2,SeqNo,EdgeWt,EndB|) on nodelink[v1];
                    connect EdgeChange(V1,SeqNo,EdgeWt,EndA|) on nodelink[v2];
                end;
            end;
        end loop;
    end ReadEdgeChange;
```

```
        entry RecordUpdate(change: boolean; oldv1, oldv2, newv1, newv2, integer; Vset:Vsettype);
        — oldv1 and oldv2 define the edge which has been removed
        — from the MST. It is replaced by the edge defined by
        — newv1 and newv2.
        begin
                if <two vertices have reported> then
                        < Update MST and Vset arrays>
                        MSTdone := true;
                end;
        end RecordUpdate;

begin — process IO
        MSTdone:= false; VsetDone := false;
        InitGraph;
        ReadEdgeChange;
end IO.

process Vertex(ioLink: link); — ioLink: link to IO process

const   NOVERT          = 0;

type
        weightType      = integer;                      — real, integer etc.
        treenum = 1 .. MAXNODE;
        MSTset = set of treenum;
        lockType = record
                set       : boolean;
                seq       : integer;
                kind      : (tree, nontree);
                nextset   : MSTset;
        end;    — lock
        edgeInfo = record
                linkend   : link;
                edgewt,
                NBTO      : integer;
                state     : edgeState;
                Vset      : VsetType;
        end;    — edgeInfo
        VsetType = set of [1 .. numNodes];
        graphType = array[1 .. numNodes];

var     minSon            : integer;
        pending_queue     : priority queue;
        graph             : graphType;
        myVertex,
        CurXNode,
        CurYNode,
        firstvert,
        outvert,
        degree            : integer;
        lock              : lockType;
```

< The calculation of the Minimum Spanning Tree (MST) using Gallager's algorithm goes here. The pseudo-code is in the First Report[Finkel86a]. When the computation is completed, InitiateVset is called at the core vertices >

```
entry ReleaseLock(seqno, vert: integer);
begin
    if lock.seq = seqno then
        lock.set := false; lock.seq := −1;
        if lock.kind = tree and minSon <> NOVERT then
            connect ReleaseLock(seqno, vert|) on graph[minSon].linkend;
        elsif lock.kind = nontree and vert <> myVertex then
            connect ReleaseLock(seqno, vert|) on graph[NBTO–>vert].linkend;
            if myVertex = vert then
                <add edge update to pending_queue>
            end;
        end;
    end;
end ReleaseLock;

entry CleanUp(seqno: integer);
begin
    if lock.seq = seqno then
        lock.set := false; lock.seq := −1;
        if minSon <> NOVERT then
            connect ReleaseLock(seqno, NOVERT |) on graph[minSon].linkend;
        end;
        <send CleanUp to every neighbor in lock.nextset and
        remove it from lock.nextset>
    end;
end CleanUp;

entry Abort(seqno, vert1, vert2: integer; edgeWt: weightType);
begin
    if lock.kind = tree then
        if minSon <> NOVERT then
            connect ReleaseLock(seqno|) to graph[minSon].linkend;
        end;
        if |nextset| > 0 then
            <send CleanUp to each neighbor in lock.nextset
                                    and remove it from lock.nextset>
        end;
    if myVertex <> vert1 then
        connect Abort(seqno, vert1, vert2, edgewt|) on graph[NBTO–>vert1].linkend;
    else
        <add edge update to pending_queue>
    end;
end Abort;
```

```
procedure ResolveConflict(seqno, from, vert1, vert2: integer; edgeWt: weightType);
begin
    if lock.set then
        if from in lock.nextset then        — deadlock has occurred
        — find out who has lower priority
            if seqno > lock.seq then
                connect Abort(seqno, vert1, vert2, edgeWt|) on
                                    graph[from].linkend;
            else — wait for other process to release lock
                await(lock.set);
            end;
        else                — no deadlock; wait until released
            await(lock.set);
        end;
    end;
end ResolveConflict;

— Update procedure terminates when A is reached. Removal of A<—>B
— is last action taken.
entry Update(A, B, X, Y, peerVertex: integer; AB_Vset: VsetType);
begin
    if myVertex = A then            — last vertex in update
        <remove edge A<—>B from MST>
        graph[B].Vset := {};
        <inform IO module of update completion>
    else
        graph[NBTO–>A].Vset := graph[NBTO–>A].Vset – AB_Vset;
        connect Update(A, B, X, Y, myVertex, AB_Vset|) on graph[NBTO–>A].linkend;
    end;
    <free lock>
    StartNextUpdate;
end Update;

— Update procedure is initiated. This is executed by X and Y.
entry InitiateUpdate(A, B, X, Y: integer; AB_Vset: VsetType);
begin
    <Add edge X<—>Y to MST>
    graph[Y].Vset := AB_Vset;
    if X = A then           — first vertex in update is also last vertex
        <remove X<—>B from MST>
        graph[B].Vset := {};
        <update is done; inform IO module>
    else
        graph[NBTO–>A].Vset := graph[NBTO–>A].Vset – AB_Vset;
        connect Update(A, B< X, Y, myVertex, AB_Vset |) on
                                    graph[NBTO-->A].linkend;
    end;
    <free lock>
    StartNextUpdate;
end InitiateUpdate;
```

```
— X and Y define the edge which is to be included in the tree; A and B
— define the edge which will be removed.
entry StartUpdate (A, B, X, Y: integer; AB_Vset: VsetType);
begin
        if myVertex = X then
                call InitiateUpdate(A, B, X, Y, AB_Vset|);
                connect InitiateUpdate(B, A, Y, X, {1..numnodes – AB_Vset|) on
                        graph[Y].linkend;
        else
                connect StartUpdate(A, B, X, Y, AB_Vset|) on graph[NBTO–>X].linkend;
        end;
end StartUpdate;

entry ReportMin (A, from, X, Y, seqno: integer; MinCost: weightType);
begin
        — see if the other ReportMin calls can be ignored
        if not MinReported then
                if CurrentMin > MinCost then
                        CurrentMin := MinCost;
                        CurXNode := X; CurYNode := Y;
                        if son <> NOVERTEX then
                                connect ReleaseLock(seqno, NOVERT|) on graph[son].linkend;
                                lock.nextset := lock.nextset – {son};
                        end;
                        son := from;
                else
                        lock.nextset := lock.nextset – {from};
                end;
        if <edges to children in MST fragment which haven't yet reported
                                        are all greater than CurrentMin> then
                FindMinDone := true;
                <connect ReleaseLock to all children who haven't yet reported>
                lock.nextset := {son};
        end;
        if FindMinDone then
                if myVertex = A then
                        if <no new minimum edge has been found> then
                                <no update needed; inform IO module>
                                call ReleaseLock (seqno, NOVERT |);
                                StartNextUpdate;
                        elsif lock.seq = seqno then          — if lock is still held
                                call StartUpdate(A, B, CurXNode, CurYNode, AB_Vset|);
                        end;
                else
                        if lock.seq = seqno then     — if lock is still held
                                connect ReportMin(A, myVertex, CurXNode, CurYNode,
                                        seqno, CurrentMin|) on graph[NBTO–>A].linkend;
                        end;
                end;
        end;
end ReportMin;
```

```
entry SearchMin(A, B, seqno, from: integer; MinCost: weightType; AB_Vset: VsetType);
begin
        ResolveConflict(seqno, from, A, B, MinCostl);
        <set lock>
        CurrentMin := MinCost;
        CurYNode := -1; CurXNode := myVertex;
        — look at incident edges and update minimum edge to other MST fragment
        foreach node in AB_Vset do
                if <edge not in current MST> and
                        graph[node].edgeWt < CurrentMin then
                                CurrentMin := graph[node].edgeWt;
                                CurYNode := node;
                end;
        end;            — foreach
        — proliferate SearchMin to children in MST fragment only if
        — edge weight is less than minimum non-tree edge weight
        BA_Vset := {1..numNodes} - AB_Vset - {from};
        foreach node in BA_Vset do
                if <neighbor in current MST and graph[neighbor].edgeWt < CurrentMin> then
                        connect SearchMin(A, B, seqno, myVertex, CurrentMin, AB_Vset l )
                                on graph[neighbor].linkend;
                end;
        end;            — foreach
        minSon := NOVERT;
        if <no children in BA_Vset> then
                if myVertex = A
                        if CurYNode = -1 and CurXNode = A then
                                <no update required; inform IO module>
                                call ReleaseLock (seqno, NOVERT l);
                                StartNextUpdate;
                        elsif lock.seq = seqno then         — if lock is still held
                                call StartUpdate(A, B, CurXNode, CurYNode, AB_Vset l);
                        end;
                elsif lock.seq = seqno then          — if lock is still held
                        connect ReportMin(A, myVertex, CurXNode, CurYnode,
                                seqno, CurrentMinl) on graph[NBTO->A].linkend;
                        MinReported := true;
                end;
        end;
end SearchMin;
```

```
entry SearchMax(maxcost: weightType; maxV1, maxV2, changedV1, changedV2, seqno:
                                                        integer; Vset: VsetType);
begin
        ResolveConflict(seqno, from, A, B, maxcost);
        <set lock>
        NBTO := graph[changedV2].NBTO;
        — see if basic cycle hasn't been completely traversed
        if myVertex <> changedV2 then
                < update maximum values, if necessary >
                — continue the search process around the basic cycle
                connect SearchMax(maxcost, maxV1, maxV2, changedV1, changedV2,
                                        seqno, Vset|) on graph[NBTO].linkend;
        else            — we are at the end of the cycle
                if <Vset hasn't been updated>
                        <no update necessary; inform IO module>
                        call ReleaseLock(seqno, changedV1 |);
                        StartNextUpdate;
                else
                        call InitiateUpdate(maxcost, maxV2, maxV1, changedV2,
                                changedV1, {1 .. numNodes} – Vset |);
                        connect InitiateUpdate(maxcost, maxV1, maxV2, changedV1,
                                changedV2, Vset |) on graph[changedV1].linkend;
                end;
        end;
end SearchMax;

procedure StartNextUpdate;
begin
        if not Empty(pending_queue) then
                <remove first update from queue; this will be of the form
                        (peerVertex, oldEdgeWt, newEdgeWt, Vset, seqno) >
                if <edge in current MST> then
                        call SearchMin(myVertex, peerVertex, seqno, myVertex,
                                newEdgeWt, Vset |);
                else
                        call SearchMax(newEdgeWt, myVertex, peerVertex, myVertex,
                                peerVertex, seqno, Vset |);
                end;
        end;
end StartNextUpdate;
```

```
entry EdgeChange(peerVertex, seqno: integer; weight: weightType; endA: link);
begin
        <if new edge is introduced into graph, store end of link in graph[peerVertex].linkend>
        — determine whether this vertex or peerVertex proceeds with
        — with search phase (if one is necessary)
        if graph[peerVertex].state <> branch and                — non–MST edge decreases
                weight < graph[peerVertex].edgeWt then
                <update graph array>
                — priority is based upon Vertex ID
                if myVertex < peerVertex then
                        if empty(pending_queue) then
                                <call SearchMax>
                else
                        <add to pending_queue>
                end;
        elsif graph[peerVertex].state = branch                  — MST edge increases
                        and (weight > graph[peerVertex].edgeWt) then
                <update graph array>
                <Calculate the size of the two MST fragments formed by removing the MST
                edge which has changed. Do this using Vertex sets. If the fragment rooted at
                myVertex is larger than the other fragment, proceed with the Search phase. If
                the fragments are the same size, use Vertex IDs to arbitrate.>
                if <this vertex proceeds> then
                        if empty(pending_queue) then
                                <call SearchMin>
                        else
                                <add to pending_queue>
                        end;
                end;
        else
                <update graph array>
                <no update is required; report to IO module that update is completed>
        end;
end EdgeChange;
```

```
    — This entry records Vertex Sets sent from other vertices and forwards Vertex
    — Sets for neighboring vertices (unless it is a terminal vertex)
    entry ReportVset(peerVertex: integer; Vset: VsetType);
    begin
        edgeCount := edgeCount – 1;
        — see if there is only one more neighbor to be heard from
        if edgeCount = 1 then
            firstvert := peerVertex;
            outvert := <neighbor not yet heard from>;
            <Form the union of the vertex sets received so far. Call this BranchVset.>
            connect ReportVset(myVertex, BranchVset|) on graph[outvert].linkend;
        else
            await(edgeCount = 0);
            if peerVertex = outvert then
                <form union of all Vertex Sets received except the one
                                from firstvert. Call this BranchVset.>
                connect ReportVset(myVertex, BranchVset|) on graph[firstvert].linkend;
            elsif degree <> 1           — not a terminal vertex
                <calculate BranchVset (union of all Vsets received except
                                        the one from peerVertex).>
                connect ReportVset(myVertex, BranchVset|) on graph[peerVertex].linkend;
            end;
        end;
    end ReportVset;

    entry InitiateVset (from: integer);
    begin
        edgeCount := <number of neighbors in MST>;
        degree := edgeCount; firstvert := NOVERT; outvert := NOVERT;
        — proliferate InitiateVset calls throughout the MST
        foreach <neighboring node <> from> do
            connect InitiateVset(myVertex|) on graph[node].linkend;
        end;    — foreach
        — when a terminal vertex receives this entry, it begins the process of
        — having each vertex calculate its initial vertex set (in ReportVset)
        if edgeCount = 1 then
            connect ReportVset(myVertex, {myVertex}|) on graph[from].linkend;
        end;
    end InitiateVset;

begin — Vertex
    <Initialize graph>
end Vertex.
```

4.8. Current status

4.8.1. Completed work

Gallager's algorithm has been implemented and extensively tested on graphs up to 16 vertices. Much larger graphs were not possible due to a limit in the number of links any process is allowed to hold.

Chen's algorithm for single edge updates has been implemented. This implementation includes edge recovery and failure. Sequential updates have been tested on random graphs up to 16 vertices.

4.8.2. Future opportunities

Concurrent incremental updates have yet to be tested and measured.

The implementation of vertex failures would be straightforward from our implementation of edge failures. Vertex recovery would not be a difficult process either.

5. Ray Tracing

Experimenter: Bahman Barzideh

Ray tracing is a technique for generating high quality computer images. It is different from other computer graphics techniques in the way it approaches the problem. Hidden-line algorithms construct the image by finding coherence characteristics in the object scene. Their main objective is to efficiently find the visible portions of the objects in the scene. In contrast, ray-tracing algorithms attempt to construct the image by simulating the behavior of light through the particular object scene. The algorithm needs to calculate the total intensity of light that arrives at each pixel of the screen. This simulation involves calculating the intensity of light that arrives at the pixel directly from light sources in the scene as well as the intensity of light reflected by and refracted through the objects.

Ray-tracing algorithms are extremely versatile. It is possible to consider illumination models and optical effects that are extremely difficult, if not impossible, to incorporate into other graphics techniques. The technique is, however, very expensive in computation time. The aim of this project was to develop and examine the efficiency of a distributed ray-tracing algorithm.

In the following sections we will first give a short description of ray tracing. We shall then describe the developed program and some of its features. In particular, we will discuss the use of k-d trees for organizing the object scene.

5.1. Method

Ray tracing is a simulation of light's behavior through an object scene. An observer who views an object scene sees the objects by means of the light shed by the light sources that strikes the objects in the scene and somehow reaches the eye. A ray of light reaches the viewer's eye either directly, from the light source, or indirectly, by reflection from and transmission through the objects. Tracing light from light sources to objects to the eye is infeasible except for the simplest problems under trivial illumination models. To make the problem feasible, ray-tracing algorithms trace the path of light in the reverse direction. The ray is fired from the viewer's eye into the object scene and its path through the object scene is traced.

The algorithm can best be though of in recursive terms. A pixel of the screen is picked and a ray is fired through it into the object scene. The objects in the scene are examined to find the first object on the ray's path. When a ray hits a face of an object it decomposes into three portions.

- A part of the ray is absorbed by the object. The frequencies of light that are absorbed specify the color of the object at the point of intersection.
- Part of the ray is reflected by the object. The angle between the reflected ray and the normal to the object at the point of intersection is equal in magnitude to the angle the incident ray forms with the same normal vector.
- The rest of the ray ray is transmitted through the object. The direction of the refracted ray is governed by the direction of the incident ray and the ratio of the refractive indices between the two media at the interface.

How much of the ray is absorbed, reflected, and refracted depends on the local properties of the object. One or two of these classes may not exist at all. For example a ray that hits a rough, opaque surface is completely absorbed.

For the intensity of the ray to be calculated where it hits the object, the intensity of the reflected and refracted rays must first be calculated, giving rise to two recursive computations. Recursion terminates when a ray leaves the object scene, hits a light source, or becomes too dim to be significant. This process is repeated for every pixel of the screen. The reader is referred to the book by Rogers[Rogers85a] for a complete discussion of ray tracing and other graphics techniques.

The above algorithm is the basic version of ray tracing. One may wish to add other optical effects such as shading, shadows, antialiasing, and color graphics. These features are not hard to incorporate into the algorithm but will increase its computational needs. For example, addition of color involves calculating each intensity three times, once for each primary color of light, and then combining the results.

5.2. Bounding volumes

Ray-tracing algorithms spend most of their time locating intersection points between rays and objects. Rogers[Rogers85a] estimates this time to be between 75 to 95 percent of the total time used. The intersection algorithm must examine every face of the object for intersection with a particular ray. It reports the intersection point (if any) with the least distance from the origin of the ray as the result. Since the plane of the face and the line of action of the ray can be in any direction in three space, specialized intersection routines cannot be used. Instead, general parametric equations are used to locate the intersection point of the ray and the plane of the face. A containment test is then performed to decide whether the intersection point is within the boundaries of the face or not. It should be noted that techniques such as the poor man's algorithm (otherwise known as backface culling) can not be used with general ray-tracing algorithms. Such techniques are used in hidden-surface algorithms to eliminate those faces whose outward normals do not point towards the viewer's eye from the intersection process.

To eliminate some of the unnecessary intersection tests, ray-tracing algorithms use bounding-volume techniques. A **bounding volume** is a simple volume such as a sphere or a box that completely encloses the object. If a ray does not intersect the bounding volume, it can not intersect the object. Bounding-volume tests are considerably cheaper and simpler than the intersection algorithm for the object.

The bounding sphere test is particularly simple. Since a sphere is symmetric, the bounding-sphere test is just a distance test. The ray intersects the sphere if the radius of the sphere is larger than the distance from the center of the sphere to the ray's line of action.

The bounding box test is more complicated, since in three faces of the box must be examined for intersection with the ray. It is still cheaper than intersection of the ray with the object, which might have many faces.

5.3. K-d trees

Bounding-volume tests prevent many of the unnecessary intersection tests between rays and objects and are relatively cheap. However, in an unordered object scene, the bounding volume test(s) must be performed for every object in the scene each time we need to locate the first object a particular ray intersects. This number of tests may acceptable if the number of objects in the scene is relatively small. But as the number of objects increases, the cost of such an approach becomes prohibitive. One solution to this problem is to impose an ordering on the objects. Rogers proposes one method for imposing such an ordering[Rogers85a], in which a hierarchy of bounding spheres is placed around several spatially related objects. If a ray does not intersect a particular sphere, the entities inside it do not need to be considered further.

We chose to implement the ordering of the objects in the scene by using a variation of k-d trees[Friedm77a]. A k-d tree is a binary partition tree with multiple partition keys. Each interior node contains partitions its children by one of the keys (dimensions) at some value. As the tree is constructed, one usually chooses the dimension of largest spread and the median value in that dimension as the value of an internal node. Each leaf node of the tree is a bucket containing a limited number of data items.

In our application, the k-d tree has the three partition keys **x**, **y**, and **z**. Each object in the scene is represented by a single point located at the center of its bounding box. The buckets are rectilinear boxes corresponding to regions of object space. The union of all the buckets is the total object space.

Tracing a ray through the object scene involves searching the tree for the bucket in which the ray is found each time it moves from one bucket to another. This search is logarithmic in the number of buckets. Our variant on standard k-d trees results in much faster searches. Each leaf node is associated with eight pointers, one to each of its eight neighboring regions. A neighbor may be another leaf, the outside world, or an interior node of the tree. The third case happens when the bucket has more than one neighboring bucket along the given face.

To trace the ray from one bucket to another, we first determine the face through which the ray leaves its current bucket. The appropriate pointer determines the neighboring region; if the neighbor is an internal node, the search for the correct bucket starts at that node. The root of the whole tree starts the search when the ray is initially fired into the object scene and in the rare occasions when it leaves a bucket through an edge, not a face.

The main problem in representing objects by points in a k-d tree arises when the partition value of an interior node bisects an object. Three solutions were considered:

(1) Place the object in only one of the subtrees, and a "stub" bounding box in the other. The stub bounding box will contain information on where the object is really kept. This solution was not chosen since it unnecessarily complicates the tracing algorithm.

(2) Cut the object into two sections along the plane of the partition value and place each half in its corresponding subtree. This solution introduces an artificial transparent face between the two halves of the object.

(3) Duplicate the entire object and place it into both subtrees. The cost of this solution is primarily in memory space. This solution was adopted mainly because of its simplicity and low execution-time overhead.

5.4. Distributing the algorithm

Since calculating the intensity of one pixel is completely independent of other pixels, one can distribute the algorithm among *n* processes by breaking the screen region into *n* sections and putting one process to work on each section. Each process has a complete copy of the k-d tree that represents the scene. In such a scheme, inter-process communication is only needed for initialization and collecting results. One expects that such a distribution scheme would achieve close to perfect efficiency, except near the end of the calculation, when some machines might have finished, but others might still be busy. This sort of solution could easily make use of DIB[Finkel85a] to assure a balanced distribution of work.

To make the problem more interesting, we decided to distribute the k-d tree itself, as well as the responsibility for different regions of the screen. Each process keeps a complete copy of the k-d tree, but only performs searches within its own part of it. As a search leaves that region, a remote call is made on the process responsible for the adjacent part. Since remote calls start new threads, a thread of execution will exist for each ray segment under computation at any time.

The performance of the distributed algorithm is heavily dependent on how we subdivide both the screen region and the object space. Efficiency can be improved by:

- Reducing the number of requests that the processes must make to one another.
- Ensuring that a processes making a request has some other work to do and does not sit idle while waiting for the results of the request.

5.5. The program

The developed program was written in Lynx, with a few C routines for string-number conversions. Two process types were needed, the **driver** (one instance) and the **tracer** (as many instances as desired). The driver first forms the bounding ball and bounding box of all objects. It then constructs the k-d tree and forms the pointer sets of each leaf node. These pointer sets can be built during the same recursion that builds the tree, with a second pass optionally used to improve the results. Figure 9 shows a simple object scene, its initial k-d tree, with some of the pointers from the time the tree was built, and the improved k-d tree. (This example is in two dimensions.) After forming the k-d tree, the driver partitions both the screen and the tree buckets among the tracers, which are dynamically created at this stage. Changing the number of tracers does not require any changes to the Lynx code. The driver also establishes a link between each pair of tracers. (The Charlotte connector could easily have been used to perform the same function.)

When a tracer process starts, it first receives the k-d tree, the bounding volumes for all objects, and the limits of the screen region it is responsible for. It then waits for a synchronizing start signal from the driver. After it starts, it builds a ray for each pixel it is responsible for and starts tracing them. To insure that a process does not run out of memory, we limit the number of pixels the process works on simultaneously. Each pixel whose computation is in progress is managed by a different thread of control. After starting all the tracers, the driver waits for results, which are buffered by tracers to reduce communication traffic. We could also have buffered ray-trace requests between adjacent tracers, but that would have made the code more complex.

5.6. Input and output

The data for each object in the scene is supplied to the program in a separate file. A directory file contains the identifier of each object in the scene along with the name of the file containing that object's

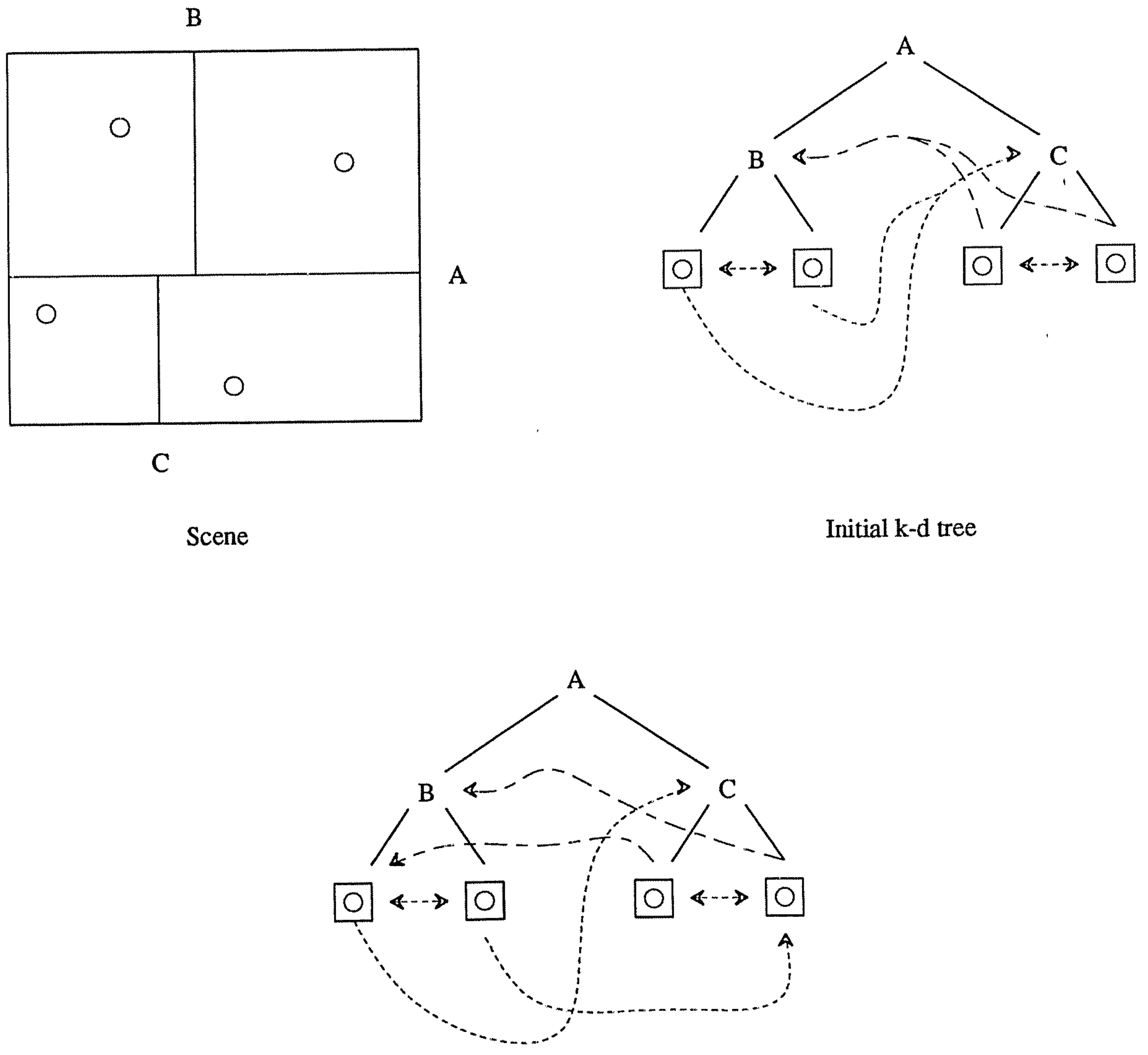


Figure 9: Relationship of scenes to k-d trees.

data. The output of the program is written to an output file by the driver. When the program terminates, this file is supplied as input to a postprocessor that runs under Unix to draw the image on an alphanumeric output device. To reduce the cost of file I/O, all files are buffered. The size of the buffer is the maximum packet size for Crystal, approximately 2K bytes. All routines required for file I/O are grouped in a separate library. The library currently offers the following routines:

fopen	Open a file (either read or write mode can be specified). A file descriptor (an integer) is returned.
fclose	Close a currently open file.
fgets	Get one line of text from the specified file.
fputs	Write a character string to the specified file.

5.7. Current status

The program has been completely written and coded. It has successfully been tested on a number of test sets and is still being debugged. Unfortunately, performance results are not available at the time of this writing.

5.8. Experience with Lynx

table sizes
: Each of the two programs has an associated library. These libraries were created for the sole purpose of avoiding overflows in the compiler's internal tables.

floating point
: Support for floating point would be of great help. The current program declares all floating point variables to be of the user-defined type **float**, which is defined to be an integer. After Lynx compiles the source code into C, we run an editor script to convert the appropriate declarations to float. Although painless, this method is inelegant and error-prone.

pred and succ
: Built-in predecessor and successor functions for set types would be useful and would have resulted in more compact code.

file I/O
: Better file I/O is a necessity! A more sophisticated version of the file I/O library written for this program may not be a bad idea.

broadcast send
: The ability to send the same message on more than one link at the same time can be a useful feature. Its counterpart, that is the ability to receive a message on one of a number of links, would be of equal value.

5.9. Pseudo-code

The following is the pseudo-code for the tracer processes. A possible improvement is to change TraceRay from an entry to a procedure, which would save some stack space when a tracer process calls itself. An entry is still required so that a remote tracer can connect to this tracer.

```
process Tracer;

entry TraceRay (ray)
    hit := FALSE;
    while (not hit) and (ray is in the object scene) do
        if (ray hits an object in current bucket) or
            (ray hits both bounding volumes of an object and
            object owned by another process) then
            hit := true;
        else
            move the ray's origin to the appropriate bucket;
        end;
    end;    — while —
    if no intersection was found then
        reply (background intensity);
    elsif bucket is owned by some other tracer process then
        res := connect to remote TraceRay (ray);
        reply (res);
    else
        intensity := ambient intensity for the intersected face;
        if (the ray is reflected) then
            obj := first object whose bounding volumes are intersect by
            the reflected ray;
            if obj is found then
                if obj is owned by some other process then
                    res := connect to remote TraceRay (reflected ray);
                else
                    res := CALL TraceRay (reflected ray);
                end
            else
                res := background intensity;
            end;
        else
            res := 0;
        end
        incorporate res into current intensity;
        if the ray is refracted then
            obj := first object whose bounding volumes are intersect by
                the refracted ray;
            if (obj is found) then
                if (obj is owned by some other process) then
                    res := connect to remote TraceRay (refracted ray);
                else
                    res := CALL TraceRay (refracted ray);
                end;
            else
                res := background intensity;
            end;
        else
            res := 0;
        end
        incorporate res into current intensity;
    end; — if
    reply (intensity);
end TraceRay;
```

```
entry Start (x, y)
    reply;      — so that more rays can be started
    initialize the incident ray;  — both its origin and direction
    locate first object whose bounding volumes are intersected by ray
    if an intersection is found then
        if the object is owned by this process then
            res := call TraceRay (incident ray);
        else
            res := connect to remote TraceRay (incident ray);
        end;
    else
        res := background intensity;
    end;
    enter result into ResultCache;
    if ResultCache is full then
        connect to display_pixel;      — entry is in driver
        set ResultCache status to empty;
    end
    NumActiveRays -:= 1;
end Start;

begin    — process: tracer —
    initialize global variables;
    receive this process's id, and link to the file server;
    receive number of ray tracer processes;
    receive total limits of the object scene;
    receive one communication link to each of the other tracer process;
    initialize file I/O library (link to file server);
    receive a copy of the k–d tree;
    read object data for all objects in this process's domain;
    calculate face equations for all faces of these objects;
    receive start signal from driver;
    y := maximum y value;
    while y >= minimum y value do
        x := minimum x value;
        while x <= maximum x value do
            await NumActiveRays <= max_active_rays;
            NumActiveRays +:= 1;
            call Start (x, y);
            x +:= delta_x;
        end;    — while —
        y -:= delta_y;
    end;    — while —
    await all active rays finished;
    if ResultCache is not empty then
        connect to display_pixel (ResultCache);
    end;
    send completion signal to the driver;
    receive termination signal from the driver;
end Tracer.
```

6. The simplex method

Experimenter: Chandrashekhar W. Bhide

The simplex method is one of the important linear programming tools[Dantzi63a]. It is used to solve a system that consists of a set of constraints and an objective function. The objective function is a linear combination of constrained variables. The aim of the method is to solve the system such that the objective function is maximized.

The initial step of the simplex method consists of adding slack (and, if necessary, artificial) variables to the constraint equation. This step is elaborated with an example.

Consider the set of constraints

$$2x+3y\leq 6$$

$$3x+4y\leq 10 \quad x,y\geq 0$$

These inequalities are converted to equalities of the desired form by adding slack variables u and w.

$$2x+3y+u \quad =6$$

$$3x+4y+ \quad w=10 \qquad x,y,u,w\geq 0$$

The purpose of adding slack and artificial variables is to create the starting basis, the details of which are not discussed here. We assume that the set of constraints has been converted to the equalities by adding necessary slack and artificial variables. As mentioned earlier, the objective function is a linear function of the constrained variables. For example, the objective function might be $6x+4y$.

6.1. Serial algorithm

We can represent the system in the matrix form as follows.

Ax = d
Maximize cx

where

A is an $m\times n$ matrix
x is a column of n elements
d is a column of m elements
c is a row of n elments

Let the matrix B be constructed as follows.

$$B = \begin{vmatrix} A & d \\ c & -z \end{vmatrix}$$

The variable z denotes the value of the objective function. At the initial step, the value of the constrained variables is assumed to be zero. Hence the value of z, which is a linear combination of the constrained variables, is zero. In the following discussion it is assumed that the system has a unique solution. The absence of such a solution can be easily detected.

The simplex method consists of repeated iterations of the following steps.

(1) Select column j such that $c_j>0$.

(2) Select row i with the smallest positive ratio $d_i/A_{i,j}$.

(3) Perform row operations on matrix B to achieve

$$B_{i,j} = 1$$

$$B_{row,j} = 0, \quad 0 \le row \le m+1, row \ne i$$

A typical way to perform the row operations is as follows:

```
for row := 1 .. m+1 do
        ratio := B[row,j] / B[i,j];
        for col := 1 .. n+1 do
                if row <> i then
                        B[row,col] -:= ratio * B[i,col]
                else
                        B[row,col] /:= B[i,col];
                end;
        end;
end;
```

Failure to find a positive c_j in the first step implies that the objective function can not be improved further and optimal solution is found. In step 2, if no i can be found that satisfies the criterion mentioned, another column j is selected in step 1. Inability to find such an i for all columns with $c_{column}>0$ means that the optimal solution is found.

Selecting the proper i in step 2 ensures that even after row operations are performed, values $d_1 \cdots d_m$ are greater than or equal to zero. The physical significance of the nonnegative d column is that the system is in the feasible region. At each iteration, the simplex method transforms the system increasing the objective function but maintaining the system in the feasible region.

6.2. Rowwise distribution

Let us assume that the task is to be done by p processes such that p divides m. Rows are assigned to **calculator processes** such that process k has the m/p rows starting with $(k-1)m/p + 1$. Each process also has a copy of vector c, so that step 1 of the method can be carried out by each process independently.

Performing step 2 requires comparison of the ratios computed by different calculator processes. A **controller process** carries out this task. The controller process also sends the selected row (B[i,*]) to all calculator processes. The calculator processes need this row to perform the row operations. The selection and dissemination of the appropriate row could also be achieved by arranging the calculator processes in a tree structure, passing values up the tree to select and down to disseminate. The skeleton structure of the processes is as follows.

```
process Calculator(id:integer,duplex : link);
        repeat
                select j ; — using step 1 criterion
                select i ; — step 2 applied to m/p rows of this process
                connect SelectIt (id, row-i | SelectedRow) on duplex;
                RowOperations; — Perform row operations in step 3 using SelectedRow.
        until finished ;
end Calculator.
```

```
process Controller(link1,link2,... : link);

        entry SelectIt (id, receiverow) : sendrow;
                accept the row from the calculator process;
                if necessary update the minratio and other data structures
                — this depends on the ratio corresponding
                — to the receiverow and previous minratio computed
                count +:= 1;
                await (count = NumCalculators);
                reply (sendrow); — corresponding to the least positive ratio
        end SelectIt;

begin — Controller
        bind link1,link2,... to SelectIt;
        other initializations;
end Controller.
```

The speedup achieved by this method depends on the communication cost and the wait periods in the calculator processes. The principle disadvantage of this method is that the calculator processes have to synchronize after each set of m/p row operations.

Thus step 2 of the simplex method is the bottleneck. If calculator processes are allowed to perform some iterations using only their own rows, the system enters the infeasible region. Intuitive explanation of this behavior is that for some steps the calculator processes try to maximize the objective function ignoring other constraints. The amount of work required to bring the system back into feasible region is high (approximately equal to the work done in the wrong steps). Hence this approach won't lead to better results. The interval between the wait periods can be increased by columnwise distribution.

6.3. Columnwise distribution

Let us assume that the task is to be done by q calculator processes such that q divides n. Each calculator process is allocated n/q columns of matrix B, such that process 1 has columns 1 through n/q, process 2 has columns $n/q+1$ through $2n/q$, and so on. Each process also has a copy of column vector d.

The columnwise-distribution method allows several iterations to finish before any communication is needed. We will call intervals between communication **rounds** to distinguish them from **iterations**. At the start of each round, each process chooses any k columns from the set of columns having a positive c value. Of these kq columns, any s columns ($s \leq kq$) are chosen by a controller (or some other distributed algorithm) to broadcast to all processes. Each process then has its own n/q personal columns, plus the s globally known columns, at most k of which are duplicates of its personal ones.

For each iteration of this round, all processes choose a globally known column (taking, for example, the one with highest c value), determine the appropriate row on which to base a row operation, and then perform row operations on the columns they possess. All processes will make the same choices, since they share globally known columns, and they will modify those columns identically, but without communication. A round continues until either a fixed limit on iteration number has been reached, or no globally known column can be used (all c values are nonpositive).

The method is illustrated by skeleton structures of the processes.

```
process Calculator(id:integer, duplex : link);
— each calculator process has a personal matrix T[m+1 rows, n/q columns]
—  and column d[m+1]; the (m+1)st row of matrix T is called vector c.
begin
        loop — each iteration is one round
                Select k positive elements of vector C;
                Send the k corresponding columns to the controller;
                connect GetCols(sendmatrix | receivematrix) on duplex;
                if nothing received then exit end;
                RowOperations; — perform all iterations of this round
        end;
end Calculator;

process Controller(link1,link2,... : link);

        entry GetCols (sendmatrix) : receivematrix ;
        begin
                if at least s good columns received then
                        reply (receivematrix);
                        exit;
                end;
                pick up good columns from sendmatrix;
                add these columns to receivematrix;
                await at least s good columns received;
                reply (receivematrix);
        end GetCols;

begin
        bind link1,link2,... to GetCols;
        other initializations;
end Controller;

procedure RowOperations;
begin
        loop — each time through is one simplex iteration
                pick a globally known column with c > 0; if none, exit;
                select row such that ratio (d[row] / A[row,col]) is the least positive;
                perform row operations;
        end;
end RowOperations;
```

Selection of the values of k and s involve interesting tradeoffs. The controller will not have to wait so long for s usable columns to arrive if k is large, but if k is too large, communications are wasted. Each round can continue through more iterations if s is large, but the cost of each iteration increases with s.

6.4. Analysis of the columnwise distribution method

We assume that the time to select k columns for broadcast and the time to choose a column for an iteration are both negligible. In each iteration, for some constants b and c,

time to choose column = 0 (by assumption)
time to choose row = bm
time for row operations = $c\ (n/q+s+1)\ (m+1)$

Let us approximate the number of iterations in a round by s. Two messages are needed in each round between each calculator process and the controller. Therefore, the communication delay per round is $2d+e$, where d is the per-message cost (roughly proportional to s) and e is the synchronization time at the controller. We can approximate the total number of iterations required to solve the problem by $1.5m$. Hence, the number of rounds is $1.5m/s$. The total time taken is then

$$(1.5m/s)(bms+c\,(n/q+s+1)(m+1)s+2d+e),$$

ignoring time for initialization and final output.

In contrast, for the serial method,

time to choose column	=	0 (by assumption)
time to choose row	=	bm
time for row operations	=	$c\ (n+1)\ (m+1)$

The total time needed by the serial simplex method is then

$$(1.5m)(bm+c\,(n+1)(m+1)).$$

The speedup for q processes is the ratio of these two quantities, namely,

$$\text{speedup}=\frac{(1.5m)(bm+c\,(n+1)(m+1))}{(1.5m/s)(bms+c\,(n/q+s+1)(m+1)s+2d+e)}$$

We can predict speedup by assigning values to the parameters b, c, d, and e. The hardest part is estimating e; we will take it to be equal to *bsm*. Typical values of the other parameters for a Lynx program in the Crystal environment are shown here in milliseconds:

$b = 0.04$
$c = 0.04$
$d = 8.40$

Table 6 shows expected values of speedup based on these constants, with s=6, k=3.

m	n	q	speedup
150	100	1	0.9
		2	1.7
		4	3.0
		8	4.9
		16	6.9
		32	8.8
1500	1000	1	1.0
		2	2.0
		4	3.9
		8	7.5
		16	14.2
		32	25.5

Table 6: Expected speedup

As expected, speedup is better for larger problems.

6.5. Comments about distributed Simplex algorithms

Many distributed algorithms fit into the following pattern. There are two types of processes, a **controller** process and many **worker** processes. The worker processes perform computations and pass intermediate results (''wisdom'') to the controller. The controller process in turn passes this wisdom to other worker processes. For the worker processes, getting up-to-date wisdom is helpful but not critical.

The distributed Simplex algorithm (and perhaps many other linear programming algorithms) do not follow this paradigm. It is essential that the computations done by the worker processes not lead the system to the infeasible region. Hence the worker processes must always have up-to-date wisdom. One of the ways to improve performance is to increase the duration between the wisdom-collection epochs, this is exactly what the columnwise distribution method does.

In spite of this constraint, in the columnwise distribution method, one calculator process may advance many rounds ahead of other calculator processes. However, the controller must ensure that every calculator starts the *r* th round with the globally known columns appropriate to that round.

6.6. Current status

The implementations of rowwise and columnwise distributed algorithms have been successfully tested for small matrices (9 × 21). Extensive experimentation with larger matrices needs to be done to measure the performance of the algorithm. Initial results show that for small matrices, the communication overhead dominates the calculation. Table 7 shows some preliminary figures for the serial and columnwise-distribution methods.

m	n	time (ms)	
		q=1	q=2
6	13	103	1003
9	21	478	1693
17	41	2784	5240
28	69	16787	14063
31	97	22773	18668

Table 7: Timings for columnwise distribution

6.7. Experience with Lynx

- The **connect** statement which works like a remote procedure call was found to be very convenient to use.
- The ability to pass arrays as arguments made implementation easy.
- Entries, which can have multiple instantiations, when used judiciously with **await** statements, lead to a fairly straightforward and compact coding.
- Events occur only when all threads are blocked. This is one of the strong assets of the language. Due to this feature, maintaining data integrity is a fairly easy task.
- The lack of support for floating-point arithmetic was a hindrance during the implementation. Since Lynx compiler generates intermediate code in C, implementing floating point arithmetic should not be too difficult.
- Argument lists do not accept conformant arrays. These would be very helpful, although their value may not offset their implementation expense.
- The library of routines that shield the user from the details of the fileserver is not quite adequate.

7. Implementation 1 of Linda's tuple space

Experimenter: G. A. Venkatesh

Linda[Gelem85a] is a language that consists of a small set of communication primitives and process-control operations to support creation and manipulation of distributed data structures. **tuple space** maintains the distributed information manipulated by Linda processes. The communication primitives in Linda consist of three operations on the tuple space.

The **out** statement appears as

out(list of parameters)

This statement causes the tuple constructed from the parameters to be added to tuple space. Each parameter may be an actual or a formal; formals indicate ''don't care'' slots in the tuple. The first parameter must be actual. This call does not block.

The **in** statement appears as

in(list of parameters)

This statement returns any tuple from tuple space that matches the parameters. Actual parameters must match exactly; formals are used to return the value actually found in their slot. Again, the first parameter must be actual. This call blocks until a matching tuple can be found; the matching tuple is removed from tuple space. If there are several matching tuples, any may be chosen.

The **read** statement appears as

read(list of parameters)

It has the same meaning as **in**, except that the matched tuple is not removed from tuple space.

Our goal was to provide an implementation for tuple space using Lynx on Crystal. Two different projects addressed this problem. This section describes the first of those projects, for which a Linda-Lynx preprocessor was written to recognize Linda's communication primitives embedded inside lynx programs and replace them with appropriate lynx operations.

7.1. The preprocessor

The following restrictions were placed on Linda's communication operations in order to simplify the preprocessor:

(1) Any operation can have at most 5 parameters.

(2) All the parameters must be of type integer.

(3) Formal parameters must be indicated by prefixing the parameter name with the symbol **var**. The name must have been declared earlier.

These restrictions allowed enough flexibility to write some non-trivial programs using Linda's operations. The operators are embedded inside Lynx programs. The operator name is prefixed with the character '$'. The preprocessor replaces these operations with Lynx **connect** statements to communicate with one of the processes implementing the tuple space.

7.2. A centralized tuple-space implementation

The first step was to implement a centralized tuple space manager to test the preprocessor as well as to get acquainted with Linda's approach to distributed programming. A single Lynx process manages the tuple space. The tuples are stored in a hash table. The first element of the tuple, which must be an actual parameter, is hashed using a modulus hash function. The tuple space process provides the following entry interfaces to the user processes:

```
entry OutTuple(
        Arg1,Arg2,Arg3,Arg4,Arg5 : integer — the tuple
        Nargs : integer — Number of elements in the tuple
        Bit1,Bit2,Bit3,Bit4,Bit5 : Bits — 0=actual, 1=formal
        )

entry InReadTuple(
        Inflag : Boolean; — true => IN operation
        Arg1,Arg2,Arg3,Arg4,Arg5 : integer — the tuple
        Nargs : integer — Number of elements in the tuple
        Bit1,Bit2,Bit3,Bit4,Bit5 : Bits — 0=actual, 1=formal
        ) : integer,integer,integer,integer,integer; — return tuple
```

The **connect** calls to these entries are generated by the preprocessor when the corresponding Linda operators are encountered.

If there is no tuple matching the parameters for an **in** or **read** request, the thread of control created for that particular request waits until a tuple is inserted into the same hash bucket, where a matching tuple is to be found. When a match is found, the tuple is deleted from the hash table if the request was an **in** operation. An **out** operation inserts the tuple into the hash table. It also awakens any threads that may be waiting for a tuple to be inserted into the same bucket.

7.3. A distributed tuple-space implementation

The centralized implementation is unsatisfactory for two reasons. First, all requests are serviced serially by the single process, which creates a bottleneck in any computation. Second, a failure in the machine running the tuple-space manager causes a breakdown in communication between the user processes.

Two distributed implementations for the tuple space are discussed in the literature. In the first implementation[Gelern85a], a copy of every tuple is stored in $\sqrt{n}$ machines where n is the total number of machines in the system. Every **in** operation requests $\sqrt{n}$ machines. The machines that have copies of the requested tuple must co-operate among themselves in order to provide the tuple to a single requesting process. (Our second project, discussed in a later section, implements this idea.) The second implementation[Carrie86a] is similar to the first, except that a copy is kept in every machine.

Our implementation is radically different from the above two. It is based on the following observations:

- All tuples that have been generated by the user processes at any given time can be divided into distinct sets of related tuples.
- Each set of related tuples is generated and consumed by processes in a small set of machines as compared to the total number of machines in the system.

The first element of the tuple (which we will call the tuple **name**) can be used as to partition tuples into sets. Only those machines that are involved in the generation or consumption of tuples with a particular name should participate in any protocol to service a request for a tuple with that name.

A second major difference from the previous implementations is that we only keep one copy of any tuple at any time. No protocol is required to achieve mutual agreement involving more than one machine.

Let m and n be the number of machines involved in the generation (sources) and consumption (sinks) of a tuple with a particular name. Then we store any tuple of that name at a source if $m < n$ and at a sink otherwise. To service a request, at most $\min(m,n)$ machines must be queried to obtain the tuple.

Since the number of sources and sinks for a given tuple name can vary as the computation proceeds, the tuple space managers must transfer tuples from a source to a sink or vice versa. Currently, the tuples are always kept in source machines. The design of a protocol for migrating tuples, in the presence of requests for these tuple, is quite complicated and our attempts at obtaining such a protocol have not been successful.

7.4. Implementation details

The tuple space is implemented by a set of identical co-operating processes, one in each machine. All user processes have links to the tuple space **manager** in the machine on which they are running. All managers are connected to each other. The manager provides the same interface as the sequential manager to the user processes. The storage structure in each of the managers is identical to the hash table implementation used in the sequential manager. In addition, the manager maintains information about the sources and sinks for all the tuples that are hashed into the same bucket. This is an approximation to the original design, which required the information to be maintained for each tuple name. Using hash buckets instead of names reduces the amount of bookkeeping and consequently the response time of the managers, at the cost of serializing requests for tuples that hash into the same bucket.

For an **out** request, the tuple is stored in the local hash table. Any thread that may be waiting for a tuple in that bucket is awakened. If there are no entries in the Source set for that bucket, the manager broadcasts to all the other managers identifying itself as a new source for tuples hashed into that bucket. In response to this message, the other managers update their source sets for the corresponding hash bucket to include the broadcasting node. However, if the source set already exists for that bucket, the manager informs only those nodes in the source set to include this new node.

For an **in** request, if the source set for the hash bucket to which the requested tuple will hash is empty, then the thread waits until a broadcast message updates the source set to include the new source. Then a **coordinator** thread is created to retrieve the requested tuple. If the requesting node is itself a source, the coordinator is in the same node; otherwise the coordinator is started on one of the source nodes.

The coordinator broadcasts requests all source-set managers to search their local hash tables. This request creates a Search thread in each such manager, which searches for the tuple (waiting, if necessary, for a new tuple). When the tuple is found, the bucket is locked, and the tuple is sent to the node on which the corresponding coordinator is waiting. Then the Search thread waits for a commit message indicating that the tuple has been accepted and must be deleted from the local hash table. The Search thread can also abort itself at various points in the above procedure if an abort signal is received from the coordinator. Since the requesting coordinator receives a reply message only if a matching tuple is found, it waits until it receives the first message and commits the manager that sent the message. In addition, it sends abort messages to the other participating nodes. During this protocol, the nodes also exchange information about the source sets and update each other's information.

Read requests are handled just like **in** requests except that the tuple is not deleted. The implementation can be optimized by removing the commit protocol for **read** requests.

7.5. Analysis

The implementation requires a global broadcast until at least one source is known to all the nodes for each hash bucket. However, the cost is amortized over any further operations involving tuples hashed into that bucket. Although each **in** request generates several messages, the requesting user process needs to wait for just two communication delays (ignoring the manager processing time) before it obtains the requested tuple (assuming that the tuple was already generated). However, managers process many requests at the same time, and services involving the same bucket are serialized, so there is some delay before the managers can reply to any request. Hence the ideal situation holds only when the number of outstanding requests at any time and the number of conflicting requests are low. Compared to the sequential case, however, each manager handles requests only for those tuples that have been generated at that node and should perform better than the sequential implementation in which the manager handles all requests serially.

Performance studies are currently being carried out to compare the two implementations.

7.6. Comments on Lynx

The lack of a non-deterministic control structure with input or output guards prevented a straightforward implementation of the protocols. A mailbox system coupled with conditional waits was implemented to simulate such a structure.

The various coordinator threads share the same link for communication among themselves. The implementation would have been simplified if virtual links could be defined over a single physical link.

Then a thread could receive a message on a particular virtual link. Since each coordinator lasts for a short time, and the number of such threads active at any time is not predictable, use of a physical link for each coordinator thread is not feasible.

7.7. Pseudo–code

```
process TupleSpace(NodeId : Integer;
                        link1,link2,.....,linkN : link; — links to user processes
                        Tlink1,......TlinkN : link); — links to other managers

    function LookUp(.....) : hash bucket number;
    — called to find a tuple in the hash table
    begin
        HashValue := Hash tuple name;
        index := First entry in the bucket;
        while index <> nil do
            if tuple names match and then
                number of parameters match and then
                tuple parameters match then
                    exit;
            else
                index := next entry in the bucket;
            end;
        end;
        return index;
    end LookUp;

    entry NewSource (Bucket : HashRange; SourceId : Nodes);
        — called when a new source for a bucket is created
    begin
        HashTable[Bucket].SourceSet ∪:= {SourceId};
    end NewSource;
```

```
procedure Enter( ); — enter a tuple into the local hash table
begin
        HashValue := hash first element of tuple;
        Insert new node into the hash table;
        foreach thread waiting for a new tuple do
                wake up thread;
        end;
        if current node is not in the source set then
                — source for the first time
                if SourceSet = { } then
                        — No sources for this bucket known
                        foreach other manager m do
                                — broadcast to all nodes
                                connect NewSource(HashValue,NodeId | )
                                        on link to m;
                        end;
                end;
        else — in source set
                foreach manager m in SourceSet do
                        — Broadcast to only the other sources
                        connect NewSource(HashValue,NodeId | ) on link to m;
                end;
        end;
        Update the Source Set for this node;
        end;
end Enter;

entry MailMessage(MailBoxNo : MailBoxRange; Code : integer;
                Mtuple : Tuple; Abort : Boolean) : Boolean;
— the coordinators and the search threads use a mailbox system to
—  communicate over the same link
begin
        with MailBox[MailBoxNo] do
                if Abort then — the message is a abort message
                        Dispose MailBox;
                        Aborted := true; — set abort flag for the mailbox
                elsif not Aborted then — a valid mail box
                        store the tuple and code in the mail box;
                end;
                reply (aborted); — reply with the status of the operation
        end;
end MailMessage;
```

```
entry Search( ) — The search thread
        : ManagersSet, — to update source set information
        MailBoxRange; — mail box with which the thread will communicate
begin
        LocalMailBox := GetNewMailBox;
        reply (SourceSet,LocalMailBox);
        repeat
                await bucket not locked or
                        MailBox[LocalMailBox].Aborted; — abort message from requesting node
                if not MailBox[LocalMailBox].Aborted then
                        Lock hash table bucket;
                        index := LookUp( ); — look up tuple in the local table
                        if index = nil then
                                await new entry into the bucket or
                                        MailBox[LocalMailBox].Aborted;
                        else
                                if curlink = nolink then — coordinator in the same node
                                        call MailMessage(MailBoxId,NodeId,Tuple,false | AbortFlag);
                                else — coordinator in another node
                                        connect MailMessage(MailBoxId,NodeId,Arg,false |
                                                        AbortFlag) on curlink;
                                end;
                                if not AbortFlag then
                                        — the protocol has not been aborted
                                        — commit wait
                                        await MailBox[localMailBox].aborted or
                                                Commit message arrived;
                                        if not Abort message then
                                                if operation is an in then
                                                        DeleteNode(index);
                                                end;
                                        end;
                                        Aborted := true; — operation completed
                                else
                                        operation aborted
                                end;
                        end;
                        Unlock bucket;
                end;
        until operation is complete;
end Search;
```

```
procedure Coordinator( );
begin
        MailBoxId := GetNewMailBox;
        repeat
                foreach node in current source set not already requested do
                        if node is self then
                                Call Search(... | UpdateInfo,RemoteMailBox);
                        else Connect Search( ...  | UpDateInfo,RemoteMailBox)
                                on link to the manager;
                        end;
                        Use UpdateInfo to update the current source set;
                        if Message has arrived in the mail box then
                                exit; — tuple found
                        end;
                end;
                if no more messages to send and no message in the mailbox then
                        await Message in the mail box or Source Set updated;
                                — further requests must be made
                end;
        until message in the mail box;
        if InFlag then
                returncode := –1; — commit with delete tuple
        else
                returncode := –2; — commit without deleting tuple
        end;
        if tuple found in the same node then
                call MailMessage(MailAddr[InfoCode],returncode,Arg,false | dummy);
        else
                connect MailMessage(MailAddr[InfoCode],returncode,Arg,false | dummy)
                        on LinkArray[InfoCode];
        end;
        foreach requested node do — send abort messages
                if node = self then
                        call MailMessage(MailAddr[j],returncode,Arg,true | dummy);
                else
                        connect MailMessage(MailAddr[j],returncode,Arg, true | dummy)
                                on LinkArray[j];
                end;
        end;
end Coordinator;

entry RemoteManager( ) : tuple;
        — called to start a coordinator at a node different from
        — the node in which the in operation was requested
begin
        Coordinator( );
        reply (tuple);
end RemoteManager;
```

```
    entry InReadTuple(InFlag : Boolean;arg1,arg2,arg3,arg4,arg5:integer;
            Nargs : integer;bit1,bit2,bit3,bit4,bit5:Bits)
    : integer,integer,integer,integer,integer;
    begin
            HashValue := Hash with first element;
            await at least one node in the Source set;
            if this node is itself a source then
                    — coordinator in the — same node
                    Coordinator(Nargs,BitMap,Arg,HashValue,InFlag);
            else
                    — start a coordinator in one of the source nodes
                    connect RemoteManager( ...| tuple)
                            on link to one of the source nodes;
            end;
            reply ( tuple );
    end InReadTuple;

    entry OutTuple(arg1,arg2,arg3,arg4,arg5:integer;
            Nargs : integer;bit1,bit2,bit3,bit4,bit5:Bits);
    begin
            HashValue := Hash using first element;
            Enter(Nargs,BitMap,Arg);
            reply;
    end OutTuple;

begin — TupleSpace
        initialize;
end TupleSpace.
```

8. Implementation 2 of Linda's tuple space

Experimenter: Man-On Lam

There are numerous distributed ways to organize tuple information across machines. The implementation described here is based on Gelernter's method that uses a lattice topology[Gelem85a]. This method always requires $O(\sqrt{n})$ work for each of the three operations on tuples.

We made several simplifying assumptions.

- all operations only accept two parameters.
- the first parameter must be a single character.
- the second parameter to **out** must be actual.
- the second parameter to **in** and **read** must be formal.

Each Linda process communicates with a tuple-server process, known as a **kernel**. The kernels are arranged in a square lattice. Each kernel maintains an array of tuples stored there. All calls to **out** put a note in the **out-state table** of the kernels in the same row as the kernel that services the **out**. All calls to **in** and **read** put a note in the **in-state table** or **read-state table** of the kernels in the same column as the one that services the **in**. These notes indicate both the name (that is, the first parameter) of the tuple and the kernel that is actually storing the tuple (for **out**) or the request (for **in**).

When a kernel services an **in (read)** call, a message is sent cyclically to all the kernels of the same column, setting a lock in the in-state (read-state) tables as they are encountered. Likewise, servicing an **out** call sends a message cyclically to all kernels of the same row, setting a lock in the out-state tables that are encountered.

As a request or tuple (let us call it R) moves down a column or across a row, it might encounter a note about a matching tuple or request M in the tables at some kernel K. In that case, a message is sent to the kernel L responsible for M; if the match is still open, it is now satisfied. Then all notes about M are removed from its row or column, and notes about R are removed from its column or row.

The locks are used on the tables to prevent accidental simultaneous update. When a kernel attempts to modify a locked entry, it either waits (if the originator's id is less then the lock's id) or aborts (in the other case). The locks are released once all a note has propagated to all tables in the row or column.

8.1. Pseudo-code

```
entry Out(tuple); — connected to by client
begin
        reply -- no need to block
        repeat
                call OutRequest(tuple,myid | result);
        until result <> rejected;
end Out;
```

```
entry OutRequest(tuple, originator) : (accepted, rejected, recorded);
— connected to by neighbor on the West; respond true if taken
begin
        if there is an unlocked matching Read R then
                connect AcceptTuple(tuple | ok) on link to R.owner;
                — AcceptTuple removes the note from all read-state tables
        if there is an unlocked matching In I then
                connect AcceptTuple(tuple | ok) on link to I.owner;
                — AcceptTuple removes the note from all in-state tables
                if ok then
                        reply accepted;
                        exit;
                end;
        elsif there is a locked matching Read or In then
                if myid < originator then
                        await R unlocked;
                        call OutRequest(tuple, originator | result); — try again
                        reply result;
                else
                        reply rejected;
                end;
                exit;
        end;
        record this tuple in local out–state table;
        if neighbor on East is not the originator then
                lock the record in the out–state table;
                connect OutRequest(tuple, originator | result)
                        on link to the East;
                reply result;
                unlock the record in the out–state table;
                if (result = accepted) or (result = rejected) then
                        remove the tuple from local out–state table
                end;
        end;
end OutRequest;

entry In(name) : tuple; — connected to by client
begin
        repeat
                call InRequest(tuple,myid | result);
        until result <> rejected;
        await tuple arrives;
end In;

entry InRequest; — similar to OutRequest.

— Read is similar to In.
```

8.2. Current status

For the time being Linda programs with four kernels have been implemented. The kernel is thoroughly tested. We are about to test larger versions with a matrix-multiplication application, where the tuple space will record an entry for each inner product required, and all workers will have both matrices as part of initial data.

8.3. Comments on Lynx

- Whether a procedure is blocking or unblocking depends on where the **reply** statement is put. It is very convenient to use the **reply** statement at the beginning of the entry for **out** and at the end for **in**. implementing the unblocking OUT operation (by putting it at the beginning of the entry) and the blocking IN and READ operations (by putting it statement at the end of the entry).
- We found the **link** data type and the Charlotte connector facility made it easy to initialize the lattice connections.
- The **await** statement made the lock resolution algorithm simple to code.
- The connector allowed us to assign a unique id to each kernel without inserting it in the source code.
- Lynx lacks dynamic memory allocation. This restriction was one reason we did not allow arbitrary tuples.
- Debugging is a difficult task due to the concurrent execution of different processes and due to the lack of debugging facilities in Lynx. Tracing each state table in each node used a very large portion of the programming time. Fortunately, the process id was attached to each debugging message, and threads do not execute concurrently, both of which helped us in debugging.
- Although the strict type checking of Lynx prevents many bugs, it also restricts the flexibility for the communication between different processes. A server that operates on integers might also be able to operate on real numbers. It is hard to write generic servers in Lynx.

9. References

Artsy86a.
Artsy, Y., H-Y Chang, and R. Finkel, ''Interprocess communication in Charlotte,'' Computer Sciences Technical Report #632, University of Wisconsin–Madison (February 1986).

Carrie86a.
Carriero, N. et al., ''Distributed data structures in Linda,'' *Procedings of the 13th ACM Symposium on the Principles of Programming Languages*, pp. 236-242 (January 1986).

Chen85a.
Chen, H. H., *Incremental computation of topological properties in distributed networks* (Ph.D. thesis) (1985).

Dantzi63a.
Dantzig, G. B., *Linear Programming and Extensions*, Princeton University Press, Princeton, NJ (1963).

DeWitt84a.
DeWitt, D., R. Finkel, and M. Solomon, ''The Crystal multicomputer: Design and implementation experience,'' Technical Report 553 (To appear, IEEE Transactions on Software Engineering) , University of Wisconsin–Madison Computer Sciences (September 1984).

Finkel85a.
Finkel, R. and U. Manber, ''DIB — A distributed implementation of backtracking,'' *Proceedings of the Fifth International Conference on Distributed Computing Systems*, pp. 446-452 (May 1985).

Finkel86a.
Finkel, R. A., A. P. Anantharaman, S. Dasgupta, T. S. Goradia, P. Kaikini, C-P Ng, M. Subbarao, G. A. Venkatesh, S. Verma, and K. A. Vora, ''Experience with Crystal, Charlotte, and Lynx,'' Computer Sciences Technical Report #630, University of Wisconsin–Madison (February 1986).

Friedm77a.
Friedman, J. H., J. L. Bentley, and R. A. Finkel, ''An algorithm for finding best matches in logarithmic time,'' *ACM Transactions on Mathematical Software*, pp. 209-226 (September 1977).

Gallag83a.
Gallager, R. G., P. A. Humblet, and P. M. Spira, ''A distributed algorithm for minimum-weight spanning trees,'' *ACM TOPLAS* **5**(1) pp. 66-67 (January 1983).

Garey79a.
Garey, M. R. and D. S. Johnson, *Computers and Intractability - A Guide to the Theory of NP-Completeness*, W. H. Freeman, San Francisco (1979).

Gelern85a.
Gelernter, D., ''Generative communication in Linda,'' *ACM TOPLAS* **7**(1) pp. 80-112 (January 1985).

Gordon85a.
Gordon, A. J., *Ordering errors in distributed programs* (Ph.D. thesis) (May 1985).

Hachte82a.
Hachtel, et al., ''An algorithm for optimal PLA folding,'' *IEEE Transactions on Computer-Aided Design of Integreated Circuits and Systems* **CAD-1**(2) (April 1982).

Lin71a.
Lin, and B. W. Kernighan, ''An Effective Heuristic Algorithm for the Traveling Salesman Problem,'' *Operation Research*, pp. 498-516 (October 1971).

Mead80a.
Mead, and Conway, *Introduction to VLSI systems*, Addison-Wesley (1980).

Rogers85a.
Rogers, D. F. and Procedural elements for computer graphics, , McGraw Hill (1985).

Scott85a.
Scott, M. L., ''Design and implementation of a distributed systems language,'' Ph. D. Thesis, Technical Report #596, University of Wisconsin–Madison (May 1985).