EXPERIENCE WITH CRYSTAL, CHARLOTTE AND LYNX

by

Raphael Finkel, A. P. Anantharaman, Sandip Dasgupta,
Tarak S. Goradia, Prasanna Kaikini,
Chun-Pui Ng, Murali Subbarao, G.A. Venkatesh,
Sudhanshu Verma and Kumar A. Vora

Computer Sciences Technical Report #630
February 1986

Experience with Crystal, Charlotte, and Lynx

Raphael Finkel
A. P. Anantharaman
Sandip Dasgupta
Tarak S. Goradia
Prasanna Kaikini
Chun-Pui Ng
Murali Subbarao
G. A. Venkatesh
Sudhanshu Verma
Kumar A. Vora

Computer Sciences Department
University of Wisconsin–Madison

Abstract

This paper describes the most 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 B trees, systolic arrays, prolog tree search, the travelling salesman problem, incremental spanning trees, nearest-neighbor search in k-d trees, and the Waltz constraint-propagation algorithm. Our conclusion is that the environment, although only recently available, is already 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. In this paper we describe some of the our projects, in particular, those that use Crystal, Charlotte, and Lynx.

The **Crystal** multicomputer[DeWitt84] 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 is an experimental distributed operating system that can run in a Crystal partition of any size[Artsy84]. 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[Scott85] 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 will call this the **mutual-exclusion** property of threads.

We have started to experiment with this environment for building distributed programs. This paper presents several implementations based on Lynx and presents an evaluation of our distributed computing environment.

---

§ Unix is a registered trademark of Bell Laboratories.

2. Distributed implementation of B+ trees

The B+ tree and its variants have proved to be popular data structures for storing large amounts of information. It is especially good for database applications, because it guarantees fast search, insert and delete operations.

In the past, work has been done to implement concurrent operations on a B+ tree, using both top-down locking[Bayer77] as well as bottom-up locking[Lehman81]. The advantage of the latter method is that only those nodes that need to be updated are locked. This process of locking begins at the leaf node where the updates are first made and propagates upwards to a level where no updates are required.

Our goal was to further enhance concurrency in a B+ tree structure by partitioning the B+ tree and distributing the data among various processes. In addition, multiple processes handling the top two levels provide multiple entry points to the tree, thus increasing the concurrency. Our design of the distributed implementation of B+ trees is based on bottom-up locking as well as a distributed implementation of extendible hashing[Ellis85].

2.1. The Data Model

Controllers and Servers

The B+ tree data structure is divided horizontally into two logical entities: the first two levels of the tree, and the rest of the tree. The first two levels of the tree are replicated among a number of processes called **controllers**. The controllers provide the user interface to the B+ tree by providing entry points to the user process. All operations begin at the root of the tree, and hence replication of the top two levels improves concurrency. Each pointer at level 2 of the tree points to a unique subtree. The lower part of the tree is partitioned such that each partition consists of one or more such subtrees. A **server** process is made solely responsible for all the operations within a partition. The amount of memory available to a server determines the size of the partition it can handle.

Servers and controllers perform operations on a tree concurrently. As a result, concurrent updates can be carried out in mutually exclusive parts of the tree at the same time by different servers. On the other hand, distinct controllers might simultaneously request operations of the same server. All updates begin at a leaf of the tree and propagate upwards. Version numbers, assigned by servers, sequence all updates across the server-controller interface. Updates across this interface are applied only if there is a match between the version numbers of the affected level-3 node and the corresponding pointer at level 2.

Query processing

In a typical database system, a user's query could be directed to any controller. That controller consults its copy of the tree and activates the server responsible for processing the query. Typical information passed to the server includes the key and the level-2 pointer information pointing to the level-3 node. If this pointer is current, the server can begin query processing immediately. If, however, the level-3 node had been split or merged due to a previous insert or delete operation, and the level 2 pointer has not been updated, then the server must find the correct subtree before processing the query. This initial adjustment is more or less trivial, because each split or merge operation at level 3 leaves behind a *next* pointer, so that queries beginning with outdated pointers can quickly recover to the correct path. We explain this algorithm in detail below.

The results of a query are communicated by the server to the controller from which it originated. If the results require no updates at level 2, the user is informed of the outcome. If an update is required, the server additionally returns the version number of the node that existed prior to the update. The version number of the pointer to a node is used to sequence updates of different pointers to the same node by different controllers. A mismatch between the versions communicated by the server and the existing version of the pointer indicates that a previous update to the node by a different controller has not yet been communicated to this controller. As a result, the current update

at level 2 is delayed by the controller until the versions match. If the versions match, an update to a level-2 pointer can be applied immediately. This update is subsequently communicated to the other controllers. Remote update requests received by controllers are subjected to the same version checks as local updates to ensure that out-of-order arrival of messages do not make the tree inconsistent.

Our design does not require the use of locks while updating the tree for two simple reasons. (1) Once a server starts processing a query, the entire subtree is available to it for processing. All the nodes of the tree in the search/update path are contained in the subtree, and hence available exclusively to the server. (2) The mutual-exclusion property of Lynx threads ensures that each query given to a server will be finished before the next one is started.

2.2. The controller

As mentioned above, the distributed implementation of the B+ tree consists of two main modules, the **controller** and the **server**. Processes that wish to use the B+ tree are called **clients**. A brief description of each is given below. The messages used for communication between the controllers and servers are listed in Table 1.

The controller process provides three entry points for the client: Search, Insert, and Delete. The client invokes these operations by remote procedure call along a link to any controller. The chosen controller starts by searching the first two levels of the tree to determine which server is responsible for the query. Additional entry points in the controller are provided for inter-controller communication: to handle remote update requests and to process acknowledgements for remote update requests.

message	meaning
Nop	No modification required in the controller.
VersionChange	A level 3 node has been modified. Requires a change in the version of the the pointer to the node in the controller and an update to other controllers.
Ins	Splitting of node at the third level. Requires insertion of (key,ptr) pair at the second level and an update to other controllers before result returned to client.
ChangeKey	Adjustment of keys at the third level. Requires modification of key at level 2 and an update to other controllers.
Del	Deletion of node at the third level. Requires deletion of level 2 (key, ptr) and an update to other controllers.

Table 1: Values returned to the controller from the server

For Find, the controller invokes Search in the appropriate server, which reports Success or Failure. The controller reports this value, in turn, to the client. For Insert, the server reports a code (Nop, VersionChange, Ins) as detailed in the table above. In those cases where updates are sent to other controllers, the originating controller waits until the version numbers match. Delete is similar to Insert, returning a code (Nop, ChangeKey, or Del). The controller leaves the pointers unchanged, but alters the key and the versions of the pointers to the node which participated in the rearrangement.

Remote operations

Remote operations are performed in response to messages broadcast by other controllers. The action to be taken and the version checks preceding them are similar to those of the updates described

above. When the message is received, if the versions are found to match, the updates are immediately applied. Version mismatch causes the remote update to be held up until the version matches. When updates are actually done, a response is sent to inform the originating controller, which can thus determine when all the controllers have effected the update.

Splitting/merging of nodes at level two

If the updates indicated above cause a node at level two to split, the controller deals with the node just as a server would. We decided not to let a deletion at level two result in merging adjacent nodes because of the complexity it would involve. Deletions may thus result in nodes at level two with fewer than the threshold number of keys. Likewise, insertions could require the height of the B+ tree itself to increase. In this case, our implementation does not attempt to reorganize the tree. It requires that all data be taken from the tree and re-inserted.

2.3. The server

A thread of control in a server is activated by any controller to process a query. Several copies of the server run concurrently, but each copy has exclusive access to only one partition of the tree. Multiple requests start off several threads of control within a server, but these requests are processed strictly sequentially. As a result, server processes do not need to lock their data.

The operations supported by the server module are Search, Insert and Delete. The controller provides the key on which the operation is to be performed and a pointer to the tree node where the operation should begin. All three operations traverse the tree searching for the key. Search reports its success. Insertion and deletion complain if the key is present (for insertion) or absent (for deletion). They also report any change in tree structure at the third level.

We have already discussed NOP, VersionChange, Ins (caused by insertion) and ChangeKey (caused by deletion). Each server sequences updates to its portion of the tree. It associates each node at its top level with a version number that increases with each update to that node. Contollers also record the last-known version number of each level-3 node. Servers report new version numbers to controllers when needed.

Communication delays can give rise to inconsistency among the controllers. A controller that has not yet heard an update may submit a query on a node that has been deleted or split. To recover from such cases, the server maintains a *Next* link for all its nodes. When a node merges into another, it is marked deleted and its Next link points to the node into which it has merged. Similarly, a node that splits points to the new node. A search operation directed to a deleted or split node recovers by following the Next links. As a result, temporary inconsistencies among controllers can be tolerated.

To merge nodes or rearrange keys between two nodes at level three, the server needs to be able to find a node's siblings. It could ask a controller, but this method introduces unnecessary communication. In our implementation, Next points to the right sibling. Often, this sibling is on the same server. If not, our implementation does not merge or rearrange keys.

Our implementation automatically reclaims deleted nodes. The controller updates its copy of the tree only when the versions match and all prior transactions have completed. If the controller did not wait for these previous transactions, they could attempt to follow a dangling pointer.

2.4. Scope for future work

It is inelegant to restructure the tree completely when its height grows. However, such growth requires a reorganization of partitions, which is expensive. One alternative is to let controllers manage a dynamically varying tip of the tree, but to keep the demarcation between controller and server fixed. However, trees that continually grow would put an increasing proportion of the tree in the controllers. A better alternative is to allow dynamic redistribution of data between controllers and servers. We have not pursued this possibility because of its complexity.

Our design assumes that each server has an unlimited amount of memory, so that it is always possible to split a node, should that be needed. In reality, servers may be limited. A better design is to allow a server to split its tree into several parts, giving each to a newly created server. Conversely, neighboring servers whose memory is being scantily utilized should be able to merge their subtrees and terminate one of the processes.

Recovery of from failures has not been considered in the present work. A substantial amount of information may have to be logged because of the presence of the different versions of the pointers. Recovery strategies for a somewhat similar problem have been suggested by Ellis[Ellis85].

2.5. Retrospective

The Lynx language allowed us to program this application elegantly.

- We found that remote invocations that start new threads of control led to a simple program structure. This is in contrast to other languages in which one programs a driver routine responsible for receiving messages and routing them to different tasks by means of a case statement.
- We also appreciated the ability to bind a single link to several entries. Each controller needed only one link connecting it to each server, even though there were several entry points in the server.
- As has been mentioned above, the mutual-exclusion property of threads enabled us to write the program without any locks.
- The await statement of Lynx allowed us to program the synchronization aspect of version numbers in a particularly simple fashion.

On the other hand, Lynx suffered from several shortcomings.

- A process often needs many functionally equivalent links. This situation arises with controllers, which all have links to each other. Unfortunately, Lynx does not allow the module header to describe this set of imported links, so we were forced to assume a fixed-sized community of controllers.
- Processes sometimes need to broadcast a message to many peers. Lynx provides two alternative techniques. (1) The sender sends the message serially to each peer, waiting for it to be accepted. (2) The sender starts a new thread of control to send the message to each peer. The first solution leads to unnecessary synchronization, and the second to a more complex program. A broadcast facility would be useful. Future versions of Lynx will very likely have **cobegin** and **coforeach** constructs that will make broadcast easier.
- There are only rudimentary debugging facilities in Lynx at the moment. Lynx should generate the necessary tables for dbx, the symbolic debugger for Unix.
- It would be helpful to have a library of Lynx routines for dealing with Charlotte utilities, partic-

ularly the file server.

2.6. Pseudo code for the server and controller

Server

```
memory node format
        key (1..2*k),
        pointers (1..2*k + 1),
        HighValue, -- maximum key present in this subtree
        NumEntries, -- number of entries in this node
        version, -- version <> 0 for third level node else 0
        LeftNode, RightNode : integer ; -- next pointers to siblings

memory : [ array of nodes ]

node status : either free, NotFree, or deleted.

-- pend is the unit of communication between the server and controller
--      and contains the following information.

        success, -- indicates if the operation succeeded.
        typ = nop, VersionChange, ChangeKey, del , ins. -- the action to be
                                                        -- taken by the controller
        version,
        OldKey,
        NewKey,
        pointer : integer ; -- to the split/merged node

function SearchNode(key , node : integer): integer ;
begin
        if memory[node] = deleted then -- this node merged with the node
                node := memory[node].LeftNode;-- pointed to by the left node
        end if;
        -- due to insertion of key,node pair a split may have occurred
        -- resulting in relocation of keys in the split node
        if memory[node].version <> 0 then -- top third level node
                if memory[node].HighValue < KeyValue then
                        node := memory[node].RightNode ;
                end if;
        end if;
        return (the node pointer to be followed from the node);
end SearchNode;

entry DirSearch(key, node : integer) : boolean;
begin
        while node <> 0 do -- node <> 0 => not a leaf node
                i := node ;
                node := SearchNode(key , i);
        end;
        reply( Findkey(key, i) ); -- boolean value true or false returned.
end DirSearch;
```

```
entry DirInsert( key, node : integer): boolean, pend ;
begin
        -- same as DirSearch to find the leaf node ;
        success := Findkey( key , node);
        continue := true ;
        if not success then
                while continue do
                        if NumEntries in node < 2*k then
                                just insert
                        else
                                SplitNode(node, KeyValue, NewNode, top);
                                -- top returns whether the node split was a top third level
                                -- node and KeyValue and NewNode return the values to be
                                -- inserted to the parent node ;
                                if not top then
                                        pop ( node );     -- pop parent
                                else
                                        continue := false ;
                                        reply (pend, type = ins, OldVersion,
                                                KeyValue, NewNode);
                                end if;
                        end if;
                end;
        end if;
end DirInsert;
```

```
entry DirDelete ( key, node : integer) : boolean, pend ;
begin
        -- same as search and push nodes as you traverse
        result := Findkey(key, node);
        if result then
                continue := true
                loop
                        DeleteKey(node, key);
                        if NumEntries >= k then -- below threshold then
                                reply (type = NOP);
                                exit -- loop
                        else
                                right := if right sibling then
                                                number of entries in right sibling
                                         else 0;
                                left := similar to right ;
                                if right > left then
                                        if right > k then
                                                if leaf node then
                                                        RightLeafAdjust
                                                else
                                                        RightNodeAdjust;
                                                end if;
                                        else -- right = k
                                                if LeafNode then
                                                        RightLeafMerge
                                                else
                                                        RightNodeMerge
                                                end if;
                                        else --left > right
                                                if left > k then
                                                        if leaf node then
                                                                LeftLeafAdjust
                                                        else
                                                                LeftNodeAdjust;
                                                        end if;
                                                else -- left = k
                                                        if LeafNode then
                                                                LeftLeafMerge
                                                        else
                                                                LeftNodeMerge
                                                        end if;
                                                end if;
                                        end if;
                                end if;
                                if node. version <> 0 then { top node }
                                        if merge then
                                                reply (type = del, deleted node,
                                                        version and KeyValue)
                                        else -- adjustment
                                                reply (type = ChangeKey, KeyToBeChanged,
                                                        its version);
                                        end if;
```

```
                                else
                                        if adjust then
                                                reply (type = NOP);
                                                exit -- loop;
                                        end if;
                                end if;
                        end if;
                end loop;
        end if;
end DirDelete;
```

Controller

```
entry ClientSearch (key : integer) : boolean;
begin
        assign transaction # ;
        search for the node to be followed from the top two levels;
        connect DirSearch (key , node | result );
        reply ( result );
        remove transaction # ;
end ClientSearch ;
```

```
entry UserInsert/UserDelete( key : integer ): boolean ;
begin
        assign transaction # ;
        search for the node to be followed from the top two levels ;
        connect DirInsert/DirDelete( key, node | result, pend );
        if not result then
                reply ( false );
        else
                if version match and for delete no transactions which originated
                  before it have completed then
                        modify the tree ;
                else
                        put it in pending table in slot i ;
                        await dummypend = i ;
                        -- if above stated conditions match then it gets awakened
                end if;
                -- broadcast to all controllers ;
                connect RemoteUpdate(pend, transaction #| OK);
                -- reply OK if the controllers able to update theirs else false;
                if not OK then
                        put it in pending ack table in slot i;
                        await dummyack = i ;
                end if;
                reply ( true ) ; -- to user
                revoke transaction # ;
                if type = del then
                        connect RemoveDelete(NodeNum) ;
                        -- automatic garbage collection ;
                end if;
        end if;
end UserInsert;

entry RemoteUpdate(pend ): boolean ;
begin
        if versions match then
                update directory ;
                reply ( true );
        else
                reply ( false );
                put it in pending table with slot i ;
                await dummypend = i ;
                connect ProcessAck( transno );
        end if;
end RemoteUpdate;

entry ProcessAck( trans : integer ) ;
begin
        accept ack from the sender;
        if acknowledgements from all the controllers have been received then
                set dummyack to the slot no; -- This awakens the blocked process
        end if;
end ProcessAck;
```

3. Systolic arrays

A **systolic system** consists of a set of interconnected cells that rhythmically compute and pass data[Kung78]. Every cell pumps data in and out, each time performing a constant-time computation, so that a regular flow of data is maintained. Information flows between cells in a pipelined fashion, and communication with the outside world occurs only at the boundary cells. Cells are typically interconnected in a regular pattern called a **systolic array**. Systolic arrays have a simple, regular communication and control structure, and therefore can be implemented inexpensively. Common systolic arrays are arranged in two-dimensional lattices with rectangular, triangular, or hexagonal interconnections. Systolic systems provide a model of computation of interest to numerical analysis and computer vision[Kung82].

Our purpose was to implement a software package that would help designers construct, debug, and test new systolic algorithms. We implemented a systolic-array generator that accepts (1) a description of the work to perform in each cell, (2) a description of the systolic array structure, (3) a description of the quotient that maps this structure onto Lynx processes[Fishbu82], and (4) a description of the input data. The generator produces (1) A Lynx program for each process and (2) A Charlotte connector file that specifies initial links among processes.

3.1. Overview

The systolic array generator is a simple compiler. The input specifications are contained in a file that the generator scans in search of keywords. The formats of the specification and output files are given in a later section. The generator creates these files:

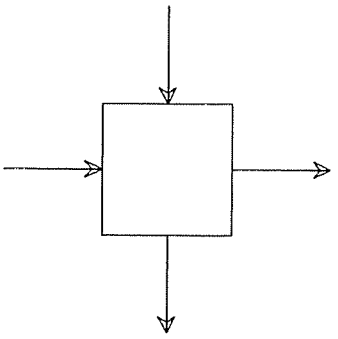
- A set of three Lynx modules, each in a separate file. One module represents the array cells. The second provides input to the array, and the third accepts output from the array.
- A Charlotte connector file to establish the initial linkage among the Lynx processes.
- An input data file.
- A command file that, when executed, causes the modules generated to be compiled.

The number of Lynx processes needed to represent the array depends on the size of the array and the quotient that has been specified. For example, a 100×100 array with quotient factor 25 will require an array of 4×4 Lynx processes. The use of a quotient allows us to implement large arrays with relatively few processes. The number of processes can be tailored to fit the number of machines in order to achieve the best concurrency and efficiency.

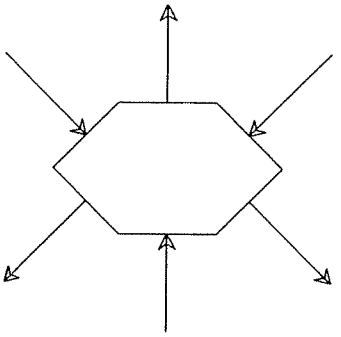
The current implementation of the generator does not provide for error recovery. If it cannot understand the specification file, the generator will print a diagnostic message and quit. The generated Lynx programs do not include tracing or debugging aids.

The generator has a limited vocabulary of cell types (square, hex) and interconnection patterns (Linear, Square, Hex). The square and hex cell types are shown here:

Square

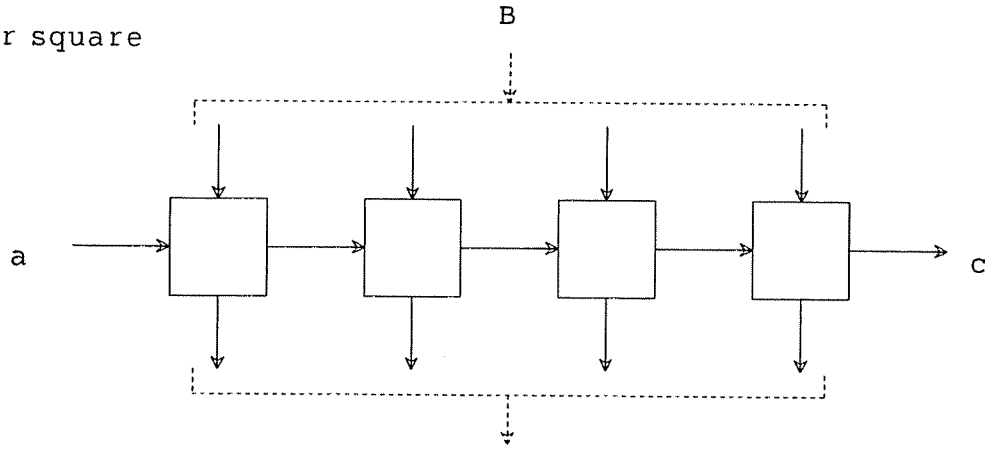


Hex

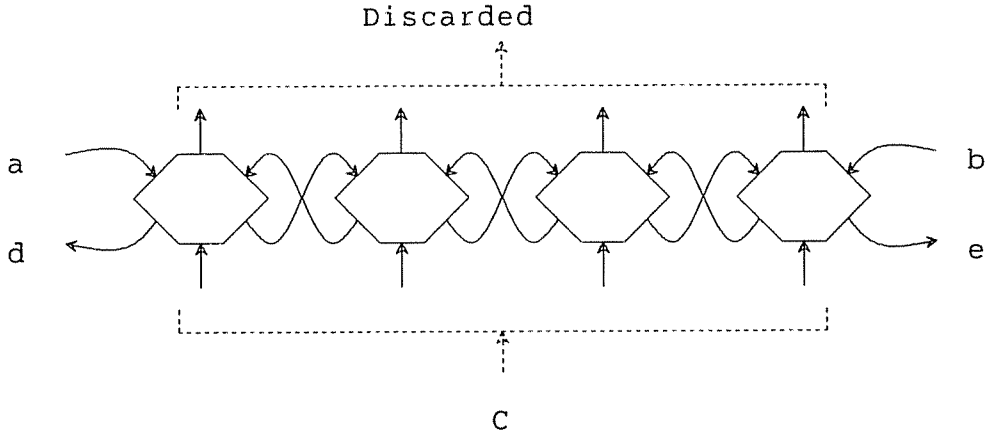


These cell types can be interconnected in various ways. The following pictures show the currently implemented interconnections along with the input/output conventions we use. Lower-case inputs and outputs are vectors; upper-case ones are matrices.

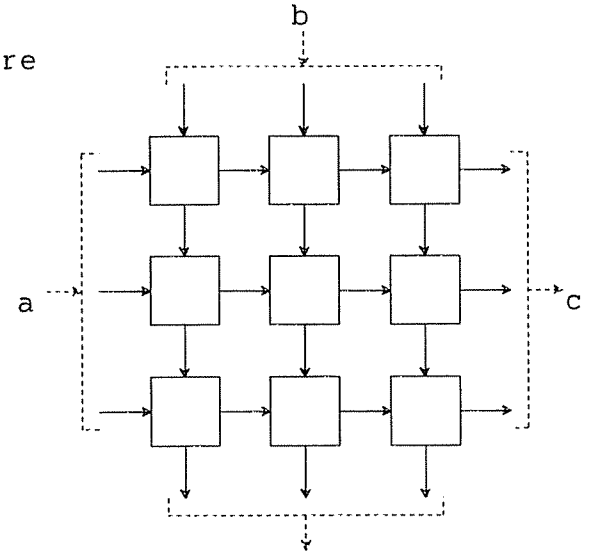
Linear square

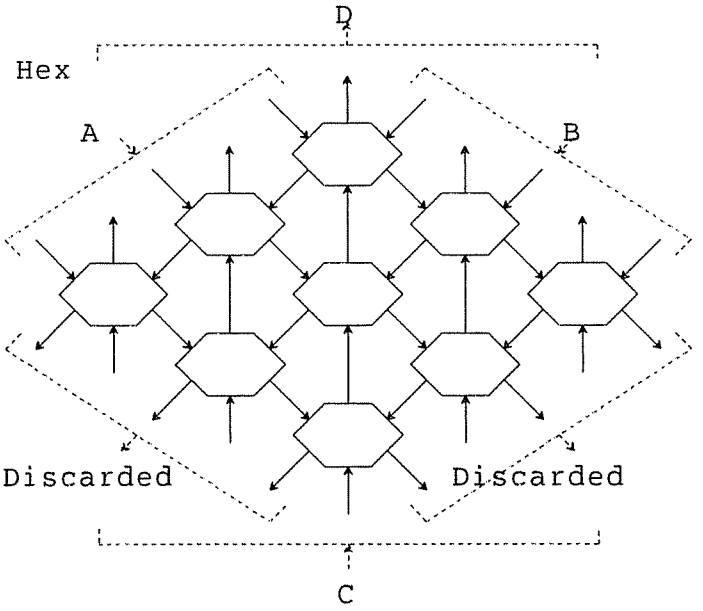


Linear hex



Square





The following table shows the options for combining these interconnections and cell types.

Interconnection Pattern	Cell type	
	Square	Hex
Linear	no	Q
Square	Q	no
Hex	no	yes

In this table, "no" means the combination is not allowed, "yes" means it is allowed, but without quotients (in the current implementation), and "Q" means that quotients are allowed.

Although true systolic arrays are likely to use a central clock, our implementation is data-driven: A cell starts working when it has all the necessary inputs from its predecessors. Input data are presented to the array by an input process, which can be a bottleneck if it must serve many input cells. Likewise, the single output process can be a bottleneck.

The major effort involved in this project was designing the quotient mechanism. Each process simulates a cluster of several cells, using global data structures to simulate intra-cluster communication. We did not allow arbitrary clustering, but required regularity. The quotient parameter for a linear interconnection tells how many cells are to be put in each cluster. Clusters in a square interconnection are themselves square lattices; the parameter here indicates the size of a side of the cluster.

3.2. User guide

A typical session with SystolGen might proceed as follows:

(1) Design a systolic algorithm that can make use of the limited interconnection patterns and cell types provided.

(2) Create a file containing the appropriate generator and interconnection pattern specifications.

(3) Run SystolGen, which will create the modules needed, a connector file "confile", an executable file "cmd.exec", and an input data file "testdata.data". Fix any errors reported by SystolGen.

(4) Compile the Lynx modules, which are called middleman.x, reader.x, writer.x, dumwriter.x (for hex patterns only), and MODULEnnn.x.

(5) Load Charlotte (using Crystal facilities such as the nuggetmaster command interpreter and the virtual terminal package) if it is not already running.

(6) Give the Charlotte command interpreter the instruction to run "confile".

3.3. Input to SystolGen

The specification file has five sections: (1) systolic array shape, (2) quotient, (3) program within each cell, (4) interconnection pattern within a quotient, and (5) input data. Each section is headed by a reserved word:

```
        < comments >
*specs
        < parameters, cell kind, and interconnection pattern >
*quotient
        < parameters for quotient desired >
*algorithm
        < algorithm for each cell >
*connector
        < interconnection pattern within the quotient >
*InData
        < the input data for execution >
        < must be arranged carefully >
*EndInput
        < comments >
```

The specs section has the following data, which must be presented in order:

NumCells (integer)	the total number of cells in the systolic array.
KindCell (string)	Either "square" or "hex".
conpattern (string)	one of "linear", "square", and "hex".
NumByte (integer)	the amount of input data in bytes (an approximation).
LeftIn (integer)	the length of input data entering the left side of the systolic array.
topin (integer)	the length of input data entering the top of the systolic array.
NumOut (integer)	the length of output data from the array.
NumPulse (integer)	number of rounds of input. an integer describing the number of pulsation of data through the array.

The quotient section has the following data, which must be presented in order:

WholelDimen (integer)	the number of cells in one dimension of the systolic array.
quotlDimen (integer)	the number of cells one one side of a cluster. A specification of 1 means no quotient. Quotients are only available for square and linear interconnection patterns. WholelDimen must be divisible by quotlDimen.

The algorithm section contains an integer, indicating how many variables are used, followed by a program. This program must use integer variables of the form "localnnnn", where nnnn ranges from 0001 to the number specified. For a square cell, val[1] and val[2] are the top and left inputs, respectively, They are also used as the bottom and right outputs. For a hex cell, val[1], val[2], and val[3] are the bottom, left, and right inputs and the top, right, and left outputs.

The connector section is currently left empty. It is intended to allow other interconnections to be specified beyond the few currently supported.

The InData section contains a sequence of integers. They are arranged into major groups, one per input cycle. For each input cycle, one input is provided for each edge cell. In a square array, each group contains the input marked 'a' in the previous picture (from bottom to top) followed by 'b'

(from left to right). There is only one group, since all inputs are parallel vectors. In a hex array, each group contains 'A' (bottom to top), 'B' (bottom to top), and 'C' (left to right). There are as many groups as needed to complete these input arrays. In a linear array of squares, 'a' is followed by 'B' (left to right). In a linear array of hex cells, the input order is 'a', 'b', 'C' (left to right) in each group.

Outputs that are not marked as "discarded" are collected in a similar fashion.

3.4. Retrospective

The Crystal multicomputer provides a wealth of facilities. The Charlotte environment and the Lynx language were particularly helpful in implementing this project. We will mention here some of the strengths and weaknesses we found.

Charlotte offers a very user-friendly environment for programmers. But it is not yet possible to run the Lynx compiler under Charlotte. Two very different environments must be mastered by the programmer: Unix for program preparation and Charlotte for execution. New users require more help and information on Charlotte than is provided by the current documentation.

The virtual terminal facility mentioned in the introduction was essential for debugging this project. However, distributed debugging was still quite painful.

Lynx was not hard to learn, especially for someone who is familiar with Modula or Pascal. Its ability to pass arrays as arguments to remote procedures made implementing quotients particularly easy; all the data to be sent from one cluster to the next were bundled into an array and sent together. The Charlotte connector file was especially valuable in setting up the initial connections among the Lynx processes. We could have benefitted from several extensions to Lynx.

- A multi-accept construct for waiting for compound events. This construct would allow a thread to wait for multiple inputs that may come in any order on possibly different links. A cell in the systolic array needs to wait for all its predecessors, but it should not impose an order on accepting their data (or it may reduce the effective parallelism). We simulate this construct by starting a separate subthread for each expected input and waiting for all these subthreads to terminate.
- A multi-connect (that is, a broadcast) construct for invoking multiple threads. This construct is the dual of the multi-accept suggested above. When a cell has finished its calculation, it would like to present the results to its successors, in whatever order they are willing to accept. We do not want to impose an order. Once again, we simulate this construct by starting subthreads.
- Standard I/O libraries. At the time this project was completed, there were no libraries for standard terminal or file I/O. Such libraries are an essential part of a programming environment. If libraries were available, we may have been willing to avoid the separate input and output processes, where file I/O code was localized.
- Array of links as a module's formal argument list. Currently, we have to specify and name each formal link. The generator therefore prepares different module headers for different interconnection and cell types.
- The number of links allowed to a process. At the time we implemented this project, a process could only have 10 links, which is overly restrictive, especially for the input and output processes. Since then, the limit (which is built into the Lynx compiler) has been raised to 25.
- Implement floating-point numbers. We tested the implementation with integer examples, but many systolic algorithms of interest need floating point. This enhancement is currently being implemented.

We found the quotient network concept essential to getting efficient use of a multicomputer for systolic arrays. Devoting a separate process to each cell leads to an overwhelming communication cost. A quotient allows us to cluster cells into one process and to reserve inter-process communication for inter-cluster traffic. This mechanism lets us increase the ratio of computation to

communication.

3.5. Sample specification file

This sample file represents a a sorting algorithm that takes two sets A, B of integers such that |A| = |B| and max(A) < min(B). It employs a 6×6 square interconnection pattern of square cells, clustered with 9 cells in each 3×3 cluster. Therefore, it needs only 4 Lynx processes to run.

```
*specs
36 square square
-- 36 cells, square cell, square pattern
*quotient
6   3
*algorithm
1
-- 1 local variable used

-- smaller value goes to val[1]
if val[1] > val[2] then
  -- swap val[1] and val[2]
  local0001      := val[2];
  val[2] := val[1];
  val[1] := local001;
end;  -- if

*connector
-- nothing to be specified (in current implementation)

*InData
6 3 2 8 4 9   23 34 10 11 56 44

*EndInput
This is the end of the specification file.
```

4. Distributed logic programming

Prolog-like non-procedural logic programming constructs appear to be suitable for distributed implementation. These constructs do not adhere to the Von Neumann style of sequential state machine. Consequently, they can be effectively implemented in a (partially) order-insensitive distributed manner.

Though Prolog has traditionally been viewed as a programming language, it is also being recognized as a powerful database query language. With growing interest in distributed databases, a distributed implementation of Prolog-like constructs appears to be a natural extension.

In past, various implementation models based on a variety of multi-processor and communication architectures have been studied. Warren has proposed a model based on broadcast network[Warren84]. Tamura presents a model for multi-computer with one master and multiple slave machines[Tamura84]. Conery proposes an abstract model independent of processor or communication architectures[Conery81].

We have implemented a model similar to one proposed by Conery in the framework of Lynx. We have chosen to implement a subset of Prolog to show the suitability of distributed implementation for logic constructs. In this spirit, we have deliberately avoided control structures of Prolog.

4.1. Overview

A logic program consists of a set of **clauses**. Clauses can be either **rules** (implications) or **facts** (assertions). A fact may look like

p (5,10,20).

where p is a predicate that "holds" for the tuple (5,10,20). A rule may look like

q(X,5) :- r(X,Y), z(5,Y).

For q to hold for some value for X above, there must exist a value of Y such that r(X,Y) *and* z(5,Y) hold. A predicate with the tuple for which it holds (such as z(5,Y)) is called a **term**. To use a program, the user supplies a **goal**. The system tries to **unify** the goal with an existing set of clauses and presents as its response an affirmative or a negative answer or a set of variable values for which unification is possible.

We have chosen to take an alternate view. We view the set of clauses as a **database** on which the user poses **queries**. These two views reflect the same underlying model, so taking one view rather than the other is just a matter of convenience.

A standard Prolog interpreter solves a conjunction of goals by working from left to right, unifying each term with the database. Unification may involve testing numerous facts and rules before hitting an exact match. From intermediate stages of unification, backtracking may be required if further progress on the same lines cannot lead to a solution. This algorithm is akin to an exhaustive tree search to find a leaf node with some desired properties. Each time we reach a leaf node and find a mismatch, we need to back up to an ancestral node and resume search in another branch.

The opportunities for parallelism become clear when we think of a Prolog program as such an exhaustive tree traversal. In principle, one can think of attempting to unify at each leaf node in parallel. In practice, however, this is infeasible as the number of possible leaf nodes is not finite for such a tree. In the framework of Prolog, one can think of exploiting multiple facts and rules in parallel.

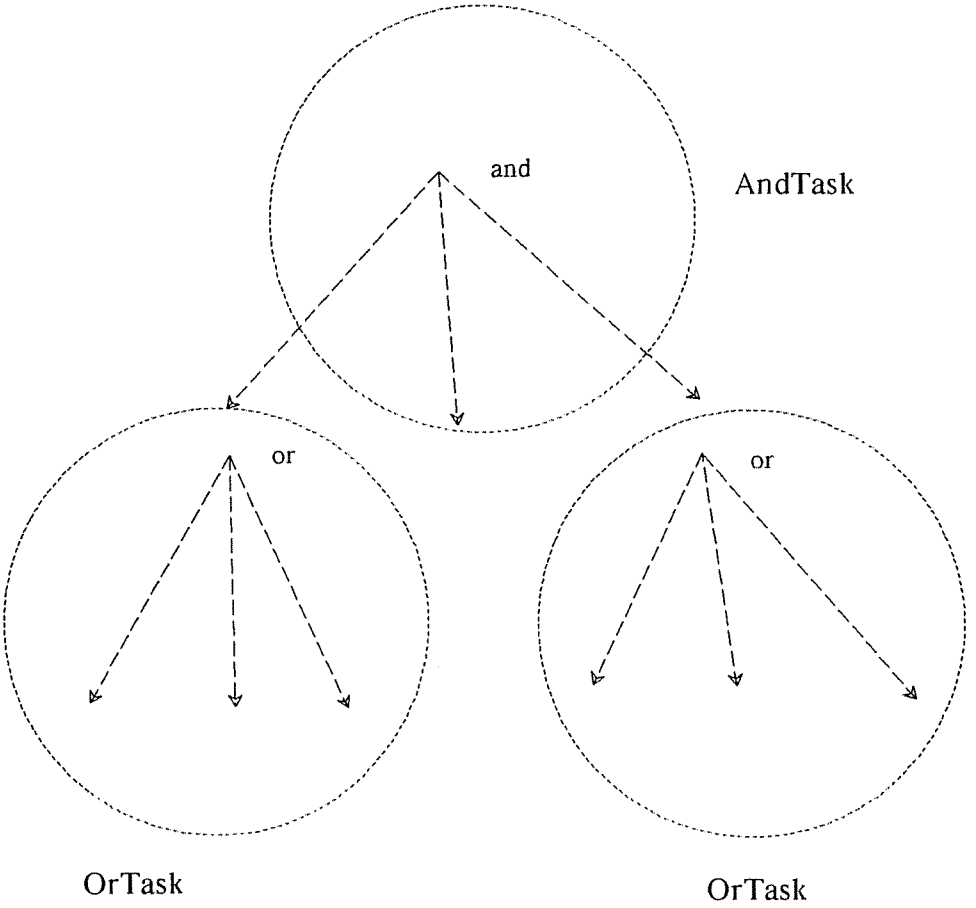
A query may consist of numerous terms. For query to be satisfied, all the terms in the query must hold for a common set of variable values. In this sense, a query is exactly alike the body of a rule.

Two kinds of parallelisms are immediately evident. First, given a query, unification of all its terms can be attempted in parallel (**and** parallelism). The same approach may be applied to bodies of rules. Second, while attempting unification of a term, various facts and rules for the same predicate

may be tested in parallel (**or** parallelism). The parallelism, however, needs to be restricted. First, unrestricted **and** and **or** parallelism can very easily lead to infinite expansion in case of recursive rules. Second, users tend to impart an intuitive, natural order to terms in a query. If solved in the same order to some extent, the interpreter may be able to reduce the number of effective free variables at any stage. We shall explore this point further later on.

4.2. Implementation details

We distributed Prolog search across a family of identical processes, each of which provides five entries. The structure of active threads is shown in the following figure.



A form of load balancing (discussed later) attempts to keep all processes equally busy by choosing in which process to invoke each new thread needed during execution.

- **BuildTables**. Three types of tables are maintained as symbol tables. The RuleTable and fact table are used to store pointers to rules and facts, respectively, and the ClauseTable is used to store actual terms corresponding to bodies of rules. BuildTables initializes these tables using the database supplied by the user. The database is replicated in each process. Consequently, there is no sharing of any symbol table information across the processes. Moreover, this is a one-time effort and the database cannot be updated once it is built.
- **AndTask**. The user's query (at the top of the tree) or the body of a rule (at an intermediate stage) is presented for solution. Calling this entry creates a new thread that we call an *And Task*. Its job is to solve several independent terms simultaneously and arrive at a solution vector that satisfies all of them. Since the solution vector must satisfy all the terms, this task must collect possible solutions for each term (by invoking subordinate threads). An enormous

amount of redundant work may be attempted if all the terms are solved completely independently. To avoid this redundancy, the And Task starts *Or Tasks* in a sequential manner. For example, to solve

p(X), q(X,Y) ,

P is attempted first. Q is started only after the first solution of X becomes available from p(X). Thereafter, solutions suggested by p are piped into q. Once Or Tasks start rolling, each one's eagerness to generate solutions is controlled by providing a statically determined number of solution buffers in the parent And Task. A speedy Or Task will eventually be delayed by a parent And Task unwilling to accept its next solution.

A protocol may be designed so that parent task may request its children to stop processing as soon as a solution is found for the given query. This protocol was not incorporated in the current design for two reasons. First, an And Task may need to generate more than one solution due to backtracking at a higher level. Since queries and rule bodies are treated alike, such a protocol would introduce irregularity in the design. Second, a user may request multiple solutions. In such a case, destroying and re-establishing the pipeline to obtain alternate solutions would be unnecessarily costly.

An And Task terminates (thereby sending a signal to its parent) when its children have finished.

- **OrTask**. An And Task starts an Or Task for each term to be unified. Thus, an Or Task is concerned with unification of one predicate, given a set of initial bindings of variables. Given that there may be numerous facts and rules whose heads match the predicate, the Or Task attempts to explore them in parallel.

  Unification with facts is usually trivial. If the Or Task distributes the task of unifying the term with facts, the communication costs could easily overwhelm the processing costs. Instead, the facts are examined by the Or Task in a sequential manner. As many solutions as possible are extracted from facts, and then all the possible rule bodies are tried in parallel. One And Task is started for each such rule. An Or Task terminates when all of its children And Tasks have finished.

  AndForker and **OrForker**. These entry points are programming conveniences to make forking a set of And and Or Tasks easy. They act as intermediaries between parents and children. To avoid unnecessary communication overhead, no load balancing is attempted at this level; these threads are always in the same process as the parent.

4.3. Some design issues and future scope

Since the motivation behind this study was only to implement Prolog-like constructs in a distributed manner, quite a few simplifications were attempted. In future designs, attempts may be made to remove some of these restrictions and add some useful features, as well as to explore the effects of some orthogonal designs.

- Integers are the only types of objects the system deals with at present. Considering that Prolog's readability stems mainly from its mnemonic object values, this easy extension may be worth attempting.
- The database is assumed to be static and fully replicated. Much of Prolog's usefulness stems from assert-like constructs that expand the database. Efforts to implement a dynamically modifiable database would be well spent.
- A fully replicated database is not necessarily a cost-effective design, especially if the database can change dynamically. A mechanism to share database information across processes may be worth exploring.
- Prolog control structures like Cut were avoided for greater freedom and uniformity of implementation. However, other features of Prolog, like Structures and Functions, may be developed within the framework of distributed implementation suggested above.

- A statically determined limit is imposed to control eagerness of evaluations. A more dynamic mechanism for controlling eagerness based on rate of usage of a clause may be more appropriate. Also, parallel to handling the eager evaluations are the issues of termination and handling of infinite recursive expansions. These are not handled efficiently and need to be given serious consideration.
- Load balancing is performed by having each process delegate work to children in processes chosen in a round-robin fashion. The effect of this strategy on performance remains to be explored.
- Though Prolog is non-procedural, it has a limited notion of order. For example, to avoid infinite recursive expansion, one writes a default or null case before a recursive rule. It is assumed that the null case will be examined first. Within a query, however, shuffling of clauses for greater efficiency may be attempted. For example, a term with a smaller number of free variables, if solved first, may also reduce the number of free variables in the other terms of the clause and thereby reduce the overall number of free variables to be unified. On the other hand, users tend to express the pipelines of solutions or free variables in the clauses themselves. Whether the system should rely on order provided or should attempt to use heuristics to improve the performance is an issue open to debate and worth exploring.

4.4. Retrospective

Having used Lynx as a tool for distributed program development, it is worth looking back to examine the impact it had on the overall design, both positive and negative.

Debugging distributed programs is generally considered a difficult task because of the concurrency of underlying processes. The mutual-exclusion property of Lynx threads of control help a great deal in this area.

Having implicit and explicit receive mechanisms helped in many ways. The explicit sends in the form of connect statements provide an easy mechanism for remote procedure calls. On the other hand, getting multiple replies from the same activation of a thread of control was absolutely essential for this application, and explicit receive statements were used in many places.

For uniform treatment of And and Or Tasks across process boundaries, it is essential that we be able to address them using the same link mechanism. Capability to bind the same link to multiple entry points was thus useful.

To be usable as a language, Lynx should provide some of the basic functionalities that it lacks at the moment, such as (1) the **ord** function, (2) capability to compare packed arrays of characters (it is currently done by a call to a C library function), (3) automatic padding of packed arrays (although this point is debatable). These are all needed in building symbol tables. Lynx resembles Pascal in many ways, so to avoid giving a programmer nasty surprises, the core of the language should probably be made compatible to Pascal in semantics, rather than to C.

Whatever be the strategy for load distribution, a programmer would most likely use a set of links for which process boundaries are invisible. It is therefore important to be able to import an array of links as a formal parameter to the process.

The exception-handling implementation has some remaining bugs, but this is not the fault of the language.

4.5. Appendix: pseudo code

```
entry OrTask (bindings, predicate, link_to_parent);
        entry AndForker (bindings, rule);
        begin
                reply; -- dummy reply to keep the parent moving.
                DestProcess:= NextProcess();
                MyEnd := NewLink (OtherEnd);
                connect AndTask (bindings,rule,OtherEnd | )
                        on LinkArray[DestProcess];
                repeat
                        receive soln on MyEnd;
                        store_solution;
                until soln = EndOfSoln;
                destroy (MyEnd);
        end AndForker;

begin -- OrTask
        reply;
        -- look up fact table first.
        if matched then
                -- send solution up.
                send TempSoln on MyLink;
        -- look up the rule table next.
        NumKids := 0;
        foreach j in RuleTable_index do
                if compare(RuleTable[j].r_name , name) then
                NumKids := NumKids + 1;
                call AndForker (bindings, j | );
                end; -- if
        end; -- for
        -- solution collection phase next....
        repeat
                await (or_latest < > OrLastUsed ); -- to limit eagerness.
                if soln = EndOfSoln then
                        NumKids := NumKids - 1;
                else
                        -- send solution up.
                        send or_solns[OrLastUsed] on MyLink;
                end; --if
        until NumKids = 0;
end OrTask;
```

```
entry AndTask (bindings, rule, link_to_parent);

        entry OrForker (bindings, name, term, row);
        begin
                reply;
                DestProcess := NextProcess();
                MyEnd := NewLink (OtherEnd);
                connect OrTask (bindings, name, OtherEnd |)
                        on LinkArray[DestProcess];
                repeat
                        receive soln on MyEnd;
                        -- next, translate the solution and store it in the and_soln[row].
                        -- The ClauseTable has the indices, TempSoln has the solution
                        -- bindings.
                        AndLatest[row] := AndLatest[row] mod MaxAndSolutions + 1;
                        if AndLatest[row] = AndLastUsed[row] then
                                -- the row is full and cannot accomodate any more solutions.
                                await (AndLatest[row] < > AndLastUsed[row]);
                        end; --if
                        StoreSolutionInTable();
                until soln = EndOfSoln;
                destroy (MyEnd);
        end OrForker;
```

```
begin -- AndTask
        reply;
        NumClauses := RuleTable[rule].r_clauses;
        cc := 0;
<<label2>>  repeat
                cc := cc + 1;
                if (cc < NumClauses) then
                        Modify( cc, rule, bindings) ;
                        -- rearranges the variables for or-forker
                        call OrForker (bindings, ClauseTable[...].c_name, ..., cc |);
                end; -- if
                await (AndLatest[cc] <> AndLastUsed[cc]);
                        -- await arrival of a solution.
                if EndOfSoln is the latest solution then
                        if cc = 1 then  -- and it has come from head of the pipeline
                                -- send EndOfSoln up and die!
                                send EndOfSoln on MyLink;
                                exit label2;
                        else
                                -- look for a solution from the previous clause.
                                -- try to backtrack or move forward in the rule.
                        <<label1>>  repeat
                                        cc := cc - 1;
                                        if latest solution = EndOfSoln then
                                                if cc = 1 then
                                                        send EndOfSoln on MyLink;
                                                        exit label2;
                                                end; -- if
                                        else
                                                backtrack();
                                                exit label1;
                                        end; -- if
                                until false;
                        end; -- if
                elsif cc = NumClauses then
                        -- the last term agrees with solution.
                  send soln on MyLink ;
                end ; -- if
        until false; -- outer loop.
end AndTask;
```

5. Distributed implementation of the traveling salesman problem

The traveling salesman problem is to find a minimal Hamiltonian circuit. As the number of vertices increases, the number of possible paths increases exponentially. Thus, even if a very powerful algorithm is used, it takes a long time to solve the problem on a single computer. There are two fruitful approaches to this problem: use of heuristics and distributing the work. This project follows both approaches.

A number of distributed algorithms have been reported in literature which could be used to solve this problem[Mohan83]. The solution used here is based on simulated annealing algorithm[83].

5.1. The algorithm

The algorithm used here is as follows[Felten85]. The set of cities (vertices) is partitioned into smaller groups. Each group of cities is handled by a different process (on a different machine). Each process tries to find a local shortest path within its group of cities. To do this, it starts off with an initial path. It then picks two cities and tries to swap them in its initial path. It uses an objective function to decide whether a swap is acceptable or not. This objective function is the difference between the old cost of the path (before the swap) and the new cost of the path (after the swap). A swap is always accepted if the objective function decreases. If the objective increases, the swap is accepted with a probability of

e sup {-c over T}

where c is the change in cost, and T is a tuning parameter (the "temperature"). If the swaps were accepted only when the objective function decreases, the algorithm might be trapped in a local minimum. Occasionally accepting swaps that increase the objective function is used to avoid this problem. The idea is borrowed from statistical mechanics. In the statistical mechanics analogy, the objective function is the energy of the system, and the parameter T is the temperature of the system. Iterative improvement freezes the system in a metastable state far from the ground state of the system. By decreasing the temperature slowly, the system is annealed and is given time to escape from the metastable states. At large T the probability of acceptance is close to 1, and swaps are always accepted. As T decreases, swaps are accepted with lower and lower probability. At very low T almost no swaps are accepted.

Each processor repeatedly attempts internal swaps for a certain number of times. (We describe the parameters we chose below.) It then decreases the temperature and again goes through the cycle. After repeating this for a certain number of times, it contacts some other processor and attempts to swap one of its cities with the one of the other processor's. Again the same technique is used to determine whether to accept the swap or not. The algorithm does not find the optimal path, but rather a good approximation.

5.2. Implementation

In our implementation of the distributed algorithm there are two modules, the **Reader** and the **Child**. There can be any number of Child modules, but since they are computationally intensive, it makes best sense to create on Child per machine. At the start, the Reader process is connected to each of the Child processes and to the file server. Each Child is connected to the Reader as well as to the other Child processes.

Reader process

The Reader process is the interface between the user and the Children. The Reader process first reads a description of the graph from a file. The input consists of the total number of cities in the tour, the name/number of the source city, followed by a list of costs from each city to every other city. The Reader also calculates the initial temperature to be used by each of the Children as the difference between the maximum and the minimum edge costs it has encountered in the data.

The Reader then uses a greedy algorithm to compute an initial circuit through the cities. It breaks up this path into roughly equal segments, which it distributes to the Children along with the initial temperature and the cost matrix.

Child process

Children sleep until the Reader gives them initial data. Each Child does its own internal processing as follows. It randomly selects two cities within its segment and attempts to swap them using the annealing technique described above. Initially it uses the temperature supplied by the Reader. It repeats attempting swaps R times per round, where R is set to 100 times the total number of cities. After each round, it decreases the temperature to 90% of its previous value. After N rounds (we set N=3), the Child randomly selects another Child to talk to and a random city to swap. It sends the name/number of this city to this process along with its own id. The other process then picks a random city within its own segment. It then sends the name/number of its own city to the first Child process, along with the change in objective function. The first process then checks whether the swap is acceptable to it by comparing the remote change in objective function with its own change, using its own current temperature. It then informs the second process whether the swap is acceptable or not. If it is acceptable, both processes perform the swap. Otherwise, neither does anything. Thus, the process that originates the conversation is the process that decides whether to implement an inter-process swap or not.

When a process has been contacted by another to find a city to swap, it unbinds its links to other Child processes to avoid being interrupted during the ensuing conversation. It reconnects them only after it has received a reply from the first process.

Each process repeats the cycle of local computation and remote swap M times. (We set M = 10.) The algorithm is assumed to have found a good approximation to the solution by this time.

5.3. Experience with Lynx

The concepts of threads and links are neat and very useful for distributed computing.

The mechanisms used to implement these ideas are really easy to understand. The mechanisms especially useful were *bind*, *unbind*, and *await*. The ability to bind and unbind links dynamically helped a great deal. The await mechanism gives a simple synchronization capability.

The different mechanisms used for inter-process communication are somewhat confusing (at least in the beginning, because there are so many of them). It is difficult to decide which mechanism to use in different circumstances. For example, simple conversations are handled by a remote procedure call (the **connect** statement), whereas more complicated conversations, of the kind we needed, required unbinding and rebinding links and secondary messages.

Broadcast can be simulated by starting multiple entries. However, if a lot of data needs to be sent in a broadcast, the compiler runs out of stack space. This is a fault of the implementation, not the language.

A better I/O facility would be appreciated. Connecting to the fileserver and reading from files is probably done by every programmer, and some sort of a utility routine to do this would be a great help.

The Lynx compiler sometimes gives strange compilation errors, originating from the C program that it generates. These errors are usually due to the incompatibility between the Lynx source program and the C program generated. Also, it is quite slow.

The runtime error messages are often meaningless. Usually these errors have to be tracked down by trial and error and result in the discovery of new limitations in Lynx. The *connect* statement seems to be especially bad in this respect.

If there were an easy way to identify the process that outputs a line to the terminal, debugging would be much easier. At present, it is very difficult to do so.

6. Minimal spanning tree and incremental update

A spanning tree contains all the nodes in a network and provides a unique path between every pair of nodes. A minimal spanning tree (MST) is a spanning tree with the minimum cost. Maintaining the MST information of a network is attractive for broadcast applications where one wishes to broadcast information from one node to all other nodes in the communications network and has to consider the cost associated with each channel of the network. With topology changes in the network, it is desirable to be able to incrementally update the MST data without having to start building the MST from scratch.

Our aim was to implement a distributed algorithm for constructing a MST[Gallag83] and the incremental update of the MST for single edge changes of the tree[Chen85]. The MST information is distributed, with each node of the network knowing only which of its incident edges are part of the MST. The update algorithm requires only a subset of nodes to participate in restructuring the MST, if needed. In our brief description of the algorithms we assume the reader is familiar with the elementary definition and properties of graphs, paths, cycles, trees[Liu68].

6.1. Construction of the MST

Gallager described an asynchronous distributed algorithm which determines the MST of a connected undirected graph with a distinct cost assigned to each node.

A fragment of an MST is defined as a subtree of the MST, which is a connected set of nodes and edges of the MST. Each individual node starts as a fragment performing the same local algorithm, which consists of sending messages over adjoining links, waiting for incoming messages, and processing messages. The algorithm results in a single fragment that is an MST.

Nodes in a fragment cooperate to find the edge with the minimal cost leading from it to a node that is not in the fragment. The two fragments then join across the edge. Each node now knows which adjoining edges are in the tree and also which edges lead to a particular edge designated as the core of the tree.

6.2. Incremental update

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 every non tree edge has the highest cost on the basic cycle it defines. The effect of an edge change to an MST can be 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 edge changes will not affect the MST.

To efficiently support the maintenance of an MST, a data structure called a **subtree vertex set** is used. A subtree vertex set is defined for every incident tree edge of the node and is the set of all nodes of the graph that can be reached from that node traversing through that edge. When an MST is affected by an edge update, the subtree vertex sets associated with the edges in the basic cycle involved are affected.

The **search phase** is the first phase of the incremental update algorithm. The two cases to be considered 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. The nodes of one of the fragments, preferably the one with fewer nodes, participate to determine the minimum outgoing edge from the fragment by passing *SearchMin* requests. If this edge is different from the tree edge, a new MST is formed by exchanging the minimum cost edge and the tree edge.
- **Decreased non tree edge cost.** A *SearchMax* message is routed through the nodes 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.

After an exchange has been decided, the algorithm enters the **update phase**. During this phase, temporary cycles may exist, but the paths defined by the subtree vertex sets between any pair of nodes are always cycle free. Higher level applications can still be supported because looping problems are absent.

Each of the two nodes 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 node in the basic cycle. Nodes receiving the message update their subtree vertex sets and forward the message to the adjacent node in the cycle. When the two nodes 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.

6.3. 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 the file server. Each Vertex process is just connected to the IO process.

IO process

The IO process is the interface between the user and the nodes of the graph. The IO module, as its name suggests, initializes the Vertex modules by sending them information on the graph configuration and the changes to that configuration. The Vertex modules inform the IO process of the MST. Input is read from a file and output is recorded on the screen.

The IO process has the following sequence of execution. It initially reads the graph size and informs the vertex processes, giving each a unique node id. The edges of the graph are read next. For each edge there are two nodes and an edge cost. A new link to represent the edge is created. The edge information is passed to the two nodes, and each is given a different end of the new link. After the entire graph is established, the nodes are awakened by the IO process to execute the distributed algorithm to construct the MST. Each Vertex process of the graph returns local information about the MST to the IO process, which can construct the complete MST after it has heard from all the nodes.

For an edge update, the IO process informs the appropriate nodes about the change. The Vertex processes execute the incremental update algorithm and return their local view of the MST. Since concurrent updates are not permitted, the request for an edge change is made only after the MST has been updated. The pseudo code for the IO module is given as an appendix.

Vertex process

Vertex processes collect the incident edge information from the IO processes. Once awakened by the IO process, they start the distributed algorithm for constructing the MST, and by exchanging messages with adjoining nodes, build the MST of the graph. When this is completed they form the subtree vertex sets of the edges. Then they inform the IO process the incident edges at that node that are part of the MST.

The Vertex process is informed of an edge change if the node it represents forms one end of the edge. It determines the type of edge change by noting whether the edge belongs to the MST and whether the cost is an increment or a decrement. Of the two processes that will be informed, only one of them will initiate the search phase. At the end of the search phase the edge that should be part of the MST is determined. If this edge is not already in the MST, an update phase is executed to update the subtree vertex sets of the nodes of the basic cycle. The two nodes corresponding to the tree edge inform the IO process the result of the edge change to the MST.

6.4. Experience report

Both the distributed algorithm for constructing the MST and the incremental update algorithm for single edge updates translate into Lynx with ease.

- Lynx provides powerful primitives and is yet simple to use and understand.
- The ability to dynamically bind and unbind entries to links and also the capability of binding links to multiple entries gives a good handle on the sequence control of a distributed program. It also aids in verification of program correctness.
- The **await** mechanism in a thread of control is a simple and elegant synchronization primitive.
- **Connect** is always blocking. A non blocking *connect* can be simulated by sending a *reply* at the beginning of an entry procedure. This requires a decision to be made on where to provide for a reply in an entry procedure.
- There is no broadcast command, but broadcast can be achieved by starting multiple entries, one for each send message.
- It would be desirable to name implicitly the process that initiated an operation on a link, as in SODA and THOTH. Such a facility would obviate the need to give each Vertex an integer id that must be passed in each message. On the other hand, using the **curlink** function might work just as well.
- Processes can only be created dynamically by making calls to Charlotte routines. A much simpler method of forking new processes would be useful. The Butterfly version of Lynx, under development at Rochester, has a means of forking processes; a similar feature may be put in the Crystal version.
- The I/O facility provided by Lynx is primitive and much effort is wasted by each programmer to write an I/O interface.
- The Lynx compiler is slow because of the intermediate cross compilation to C. Separate compilation helps.
- Debugging a distributed program is certainly not very easy. A Lynx debugger facility to aid in debugging will go a long way to alleviate the hardship.
- A help document that features information on how to use the system is imperative. In a constantly changing system environment regular updates to this document would be needed. A tutorial at the beginning of the semester to novices might be a good idea.

6.5. Pseudo code for IO module

```
module IO (flink, link1,link2,....., linkn : link);
        -- flink : link to the fileserver
        -- link1, link2,....,linkn : links to Vertex process 1,2,...n

var
        NumNodes, NodeCount, ReportCount : integer;
        MSTdone : boolean;
        MSTarray : array [1..MAXNODES] [1..MAXNODES] of integer;
        NodeLink : array [1..MAXNODES] of link;      -- array of the module link parameters

entry GraphSize (NumVertices, OwnVertex : integer); remote;
        -- NumVertices : number of vertices in the graph
        -- OwnVertex   : vertex number assigned to the node
entry NewEdge   (OwnLinkEnd : link; PeerVertex : integer; EdgeWt : WtType); remote;
        -- OwnLinkEnd : one end of the link representing the edge
        -- PeerVertex : the node on the other end of the edge
        -- EdgeWt     : wt of the edge

entry InitGraph;
        -- determines the graph size and the edges of the graph
        -- informs the vertices the graph configuration
        -- wakes up the vertices

endA, endB      : link;          -- ends of the link representing an edge

begin
        -- find graph size
        read (NumNodes, NumEdges);
        NodeCount := NumNodes;
        foreach node in [1..NumNodes] do
                connect GraphSize (NumNodes, NumVertices |) on NodeLink[node];
        end; --foreach

        -- set up all the edges
        foreach edge in [1..NumEdges] do
                read (vertex1, vertex2, EdgeWt);
                endA := NewLink (endB);              -- create a NewLink for the edge
                connect NewEdge (endA, vertex2, EdgeWt |) on NodeLink [vertex1];
                connect NewEdge (endB, vertex1, EdgeWt |) on NodeLink [vertex2];
        end; --foreach

        --wake up all the nodes
        foreach vertex [1..NumNodes] do
                connect Wakeup on NodeLink [vertex];
                bind NodeLink[vertex] to RecordMST;
        end; --foreach
end InitGraph;

entry RecordMST (NodeId : integer; EdgeArray : array [1..NumNodes] of integer);

begin
        NodeCount := NodeCount + 1; -- count of # of nodes reporting
        <update MSTarray>
```

```
        if NodeCount = NumNodes then
                MSTdone := true;
        end; --if
end ReportMST;

entry InformEdgeChange ;

begin
        loop    -- forever
                await MSTdone;          -- wait until the vertices report MST info
                read (vertex1, vertex2, EdgeWt);
                connect EdgeChange(vertex2, EdgeWt |)
                        on NodeLink [vertex1];
                connect EdgeChange(vertex1, EdgeWt |)
                        on NodeLink [vertex2];
                bind NodeLink[vertex1], NodeLink[vertex2] to RecordUpdate;
                MSTdone := false;
        end; --loop
end InformEdgeChange;

entry RecordUpdate (change : boolean; vertex1, vertex2, EdgeWt : integer);
        -- change : indicates if the MST has been updated
        -- vertex1, vertex2, EdgeWt : give information of the new MST edge

begin
        unbind curlink from RecordUpdate;
        ReportCount := ReportCount - 1;                 -- two edge nodes should report
        if change then
                <update the MSTarray>
        end; --if
        if (ReportCount = 2) then
                MSTdone := true;
                ReportCount := 0;               -- reset count
        end; --if
end RecordUpdate;

begin
end IO.

module Vertex (IoLink : link);
        -- IoLink : link to the IO module

type
        EdgeState = (basic, branch, rejected);
        VsetType  = set of integer;             -- type for the vertex sets
        EdgeInfo = record
                        LinkEnd : link;         -- own link end of the edge
                        EdgeWt  : integer;
                        state   : EdgeState;
                        Vset    : VsetType;
                end;

var
```

```
        TestEdge, bestEdge, inBranch : integer;
        graph   : array [1..MAXNODE] of EdgeInfo;     -- array of incident edges
        SN,             -- node state;
        LN,             -- level number of the fragment
        FN : integer;   -- fragment number
        MyVertex,               -- id of the node as given  by the IO module
        bestWt,                 -- best Wt outgoing edge, known at the node from the fragment
        FindCount : integer;    -- number of nodes reporting the best Wt


-- ENTRIES USED IN THE MODULE ARE GIVEN BELOW

entry Wakeup;
        -- initializes all the variables at the node
        -- wakes up the node to start the algorithm

entry Linkup (levelNo : integer);
        -- request for a connect from a fragment of level 'levelNo'

begin
        if levelNo < LN
                -- <merge the two fragments and initiate search for the next outgoing edge>
        else await (graph[edge].state = basic);
           connect Initiate ( LN+1, EdgeWt, find |) on curlink;
        end; --if
        bind curlink to Report;
end Linkup;

entry Initiate ( levelNo, FragmentNo : integer; status : NodeState );
        -- initiate a search for the next outgoing edge from the fragment
        -- also pass fragment identification
begin
        if  adjacent node in fragment then
                connect Initiate (levelNo, FragmentNo, status|) on adjacent node link;
                FindCount := FindCount + 1;
                bind adjacent node link to Report;
        end; -- if

        if status = find then
                < join search >
                -- find minwt edge and Test if edge can be included into the fragment
        end; --if
end Initiate;

entry Test (levelNo, FragmentNo : integer);
        -- test if edge can be added to the fragment
begin
        await (levelNo <= LN);
        if (FragmentNo <> FN) then
                connect Okay on curlink;
        elsif  graph [edge].state = basic then
                graph [edge].state := rejected;
                connect Reject on curlink;
```

```
        end; --if
end Test;

entry Okay;
        -- since the test edge has been okayed note the edge as the best
        -- edge at the node and Report to the node on the InBranch

entry Reject;
        -- note that the test edge is to be rejected and continue search for the best edge

entry Report (wt : integer);
        -- best edge information reported to the node through this entry

begin
        unbind curlink from Report;
        if <msg not from InBranch> then
                FindCount := FindCount -1;
                if wt < bestWt at the node then
                        bestWt := wt;
                        bestEdge := msg Edge;
                end; --if
                < report on InBranch if all the nodes in the subtree of this
                  node have reported and the test at this node is also completed>
        else    await (SN = found);
                -- wait for the subtree to determine the bestEdge
                if (wt > bestWt) then
                        <include the new edge into the MST by calling entry ChangeRoot>
                elsif (wt = bestWt=MAXwt) then
                        connect RecordMST (EdgeArray) on IoLink;
                                -- inform the IO module of the incident edges that are part of the MST
                        halt;
                end; --if
        end; --if
end Report;
```

6.6. Current status

Given a graph, the minimal spanning tree is constructed as described. The input is read from a file using the file server. Output is reported to the IO module which prints it out on the screen. A graph size of 4 nodes and 6 edges has been tested.

The code for the incremental update for single edge changes has been written but is yet to be tested; it is very simple. After the MST is built, the Vertex sets are constructed. The IO module when given edge change information notifies the appropriate nodes, of which one will initiate the incremental algorithm. On completion of the update, if any, the two nodes then inform the IO module which is then ready to accept the next edge change.

7. Nearest-neighbor search with K-d trees

The nearest neighbor or best match problem applies to data files that store records with several real valued keys or attributes. The problem is to find those records in the file most similar to a query record according to some dissimilarity or distance measure. Formally, given a file of N records (each of which is described by k real valued attributes) and a dissimilarity measure D, find the m records closest to a query record (possibly not in the file) with specified attribute values. Consider, for example, a data file containing information about the various publications in Computer Science. Each publication record may contain its degrees of relevance to different fields of Computer Science such as O.S., Compliers, Programming Languages, Architecture and the like. In such a case, a researcher seeking publications that deal with the the influence of programming languages on computer architecture could use the nearest neighbor search to his advantage.

An algorithm that effectively uses a K-d tree structure of file storage in improving the performance of the nearest neighbor search on a uni-processor machine has been proposed by Friedman[Friedm77]. Our goal was to build a distributed implementation of this algorithm. The motivation behind this attempt was two-fold: to enhance the concurrency in processing queries and to experiment with dynamic restructuring of a distributed K-d tree structure.

The following subsections present a description of the distributed implementation of the above algorithm. We first briefly explain the K-d tree storage structure and the algorithm. The issues involved in designing a distributed implementation are discussed next. We then move on to describe the implementation details. A critical evaluation of Lynx as a distributed programming language is also presented.

7.1. The k-d tree storage structure and the search algorithm

The K-d tree is a generalization of the simple binary tree used for sorting and searching. In case of one-dimensional searching, a record is represented by a single key and a partition at each node in the binary tree is defined by some value of the key. All records in a subfile with key values less than or equal to the partition value belong to the left child, while those with a larger value belong to the right child. In k dimensions, a record is represented by k keys. Any one of them can be used as the discriminator for partitioning the subfile represented by a particular node. The choice of both the discriminator keys and the partition values has influence on the performance of the search algorithm in a K-d tree.

The K-d tree data structure provides an efficient mechanism for examining only those records closest to a query record. At each node, the partition not only divides the current subfile, but it also defines a lower or an upper limit on the value of the discriminator key for each record in the two new subfiles. The accrual of these limits in the ancestors of any node defines a cell in the multidimensional record-key space containing its subfile. The volume of this cell becomes smaller as one descends the tree. A list of the m closest records so far encountered and their dissimilarity to the query record is always maintained as a priority queue during the search. Whenever a record is examined and found to be closer than the most distant member in the queue, the queue is updated.

The search algorithm can best be described as a recursive procedure. If the node under investigation is terminal, all the records in the bucket are examined. The first invocation passes the root of the tree and the query record as the argument. If the node is not terminal, the recursive procedure is called for the node representing the subfile on the same side of the partition as the query record. When control returns, a test is made to determine if it is necessary to consider the records on the side of the partition opposite to the query record. It is necessary to consider that subfile only if the geometric boundaries delimiting those records overlap the ball centered at the query record with radius equal to the dissimilarity to the m-th closest record so far encountered. This is referred to as the "Bounds-Overlap-Ball" test. If the bounds do overlap the ball, then the records of that subtree must be considered and the procedure is called recursively for the node representing that subfile. A "Ball-Within-Bounds" test is made before returning to determine if it is necessary to continue the search. This test determines whether the ball is entirely within the geometric domain of

the node. If so, the current list of m best matches is correct for the entire file, and no more records need be examined.

In order to minimize the expected number of records examined by the algorithm, it is suggested[Friedm77] that at every nonterminal node, the key with the largest spread in values be chosen as the discriminator and the median of the discriminator key values be chosen as the partition.

7.2. Distributed implementation issues

As mentioned earlier, the idea behind distributing the above algorithm was to increase the concurrency in processing queries requiring nearest neighbor search and to attempt dynamic restructuring of the K-d tree.

Improvement in concurrency can be achieved (1) by distributing the work involved in processing a query among a set of processor nodes, andl (2) by allowing simultaneous processing of more than one query. Earlier in this technical report we have seen how the latter is achieved in B+ trees by replicating the first two levels of the tree and designating them as controllers. The implementation discussed there uses the notion of **versions** and the mutual-exclusion property of Lynx threads to guarantee the integrity of data. That approach can be easily modified to achieve concurrency in processing multiple queries for K-d trees as well.

There are several trade-offs involved in achieving concurrency by distributing the processing of a single query among several nodes. The overhead of information passing between nodes can more than offset the benefits of concurrent processing. In the nearest-neighbor search algorithm, which is essentially sequential, concurrency could be introduced as follows. If there is high probability that both children will be called upon to do search for nearest neighbors, they can be asked concurrently. This overlap can potentially get results faster, and the parent does not have to compute the expensive Bounds-Overlap-Ball test. However, the second (that is, the further) child does not have the benefit of the tighter bound discovered by the first child, so it may do more work than it would have if the first child were allowed to finish first. The benefits obtainable from such kind of parallelism (eager evaluation) are therefore limited.

The performance of nearest neighbor search deteriorates as insertions and deletions render the K-d tree far from optimal. The optimality conditions require that the discriminant key at each node be the one that has the largest spread in that subtree, and that the subtree should be balanced with respect to the number of records in each partition. Therefore it is desirable that a mechanism be integrated into the distributed implementation that will dynamically restructure the tree as insertions and deletions change its balance. The advantage of dynamic partial restructuring as opposed to periodic global restructuring is that while one part of the tree is being restructured, other parts can still be processing queries. Thus downtime is avoided at the expense of little increase in response times for insertions and deletions. We have designed and implemented a simple and elegant restructuring mechanism. However, it requires a significant amount of information transfer between processes. Another mechanism that reduces the amount of information transfer is proposed in the next section.

7.3. Implementation details

For the purpose of simplicity of implementation, one NodeProcess (a Lynx process) is assigned to each node of the K-d tree. These processes are linked in the form of a tree at the time of their creation. Each NodeProcess offers a set of entry points to its parent. These entry points facilitate query processing and tree restructuring. The following table describes the functions provided by all such entry points.

Entry	Function desired of the child
GetNearest	Get the m records nearest to the query record.
Submit	Give back all your data.
Insert	Insert the new record, restructure if required.
Delete	Delete the old record, restructure if required, return status.
GiveBounds	Report your current bounds

Entry points offered by the child to the parent

Tree initialization and query processing

An initialization process, Init, reads parameters and data from files on the host Unix machine and then invokes the ServerNode entry of the root process with the data and parameters. Among the parameters are the bucket size, the number of keys in the data records (that is, the dimension) and the tolerance limits for restructuring (explained later). The ServerNode entry in a NodeProcess accepts data given from the parent and determines the bounds of the geometric cell represented by the subfile given to it. If the size of the subfile is greater than the bucket size, it determines the discriminant key, splits the data into two subfiles, and calls the ServerNode entry point of its two children. This action recursively builds the whole K-d tree.

The Init process then reads query records from a file and invokes the GetNearest entry of the root process. GetNearest follows the nearest neighbor search algorithm described above. The details of the algorithm implementation is presented as pseudo code in the appendix.

Dynamic restructuring

Each NodeProcess is given the responsibility of maintaining the balance of the subtree below it. Instead of restructuring at every update, we wait until the imbalance between children exceeds a given parameter. When restructuring is needed, a NodeProcess, it connects to the Submit entries of its children, which submit their data to the parent and terminate. (The thread corresponding to the ServerNode entries of the children terminate.) The parent node then reorganizes the data (possibly with a different discriminant key) and then calls the ServerNode entries of its children with the newly split data as at the time of creation. Since the parent is responsible for the balance of its subtrees, children never need to report exception conditions such as overflow.

This simple restructuring scheme is inefficient, because each time a node decides to restructure its subtree, it has to collect all the data contained in that subtree. A more efficient mechanism was designed to help restructuring without requiring the parent to collect all the data contained for all cases. The following table gives the entry points that a child provides its parent and its sibling under this scheme.

There are two cases that require restructuring: (1) when there is an imbalance in the subtree represented by a node with respect to the number of records with each child, and (2) when the discriminant key needs to be changed. Case (1) is relatively simple to handle; the parent tells one of the children to share some number of records with its sibling and asks both the children to return their updated bounds. The giving child calls Take in the accepting child. For case (2), it is probably better to ask both the children to submit their data and to rebuild the subtree. If the distribution of the key values does not change drastically, the latter case arises less frequently.

Entry	Function desired of the child
GetNearest	Get the m records nearest to the query record.
Insert	Insert the new record, restructure if required.
Delete	Delete the old record, restructure if required, return status.
GiveBounds	Give your current bounds
Submit	Submit all your data.
ShareWithSibling	Give n records(in given direction) to your sibling.
Take	Take n records and add them to your subtree.

Entry points offered by the child under new scheme

When a node gets a ShareWithSibling request, it can make Submit requests on its own children to get the necessary data. Of course, it may want to start a restructuring of its own subtree in this case. When a node gets a Take request, it generates recursive Take requests to its children.

7.4. Evaluation of Lynx as a distributed programming language

Having written this project in Lynx, we can report the following experiences with the language.

- **Ease of distributed programming.** The implicit receive facility of Lynx makes remote procedure call easy, and provides an elegant way to program a client-server relationship. The mutual-exclusion property of threads relieves the programmer of having to program mutual exclusion for the use of global data. In applications where an entry procedure is blocked for service, the programmer must ensure that the global invariant is true before the entry procedure becomes blocked.
- **Programmability for concurrency.** The fact that entry procedures are allowed to reply before termination can be exploited by a programmer to achieve greater concurrency, as we have done in our implementation. The effect of a non-blocking send can be achieved by early reply in most cases, and hence its absence was not felt in our application.
- **Debugging.** The mutual-exclusion property of threads, together with the absence of asynchronous message passing, eases the debugging of a distributed program. The strict type checking of messages at execution time also reduces the chances of errors in programming. Aside from Lynx implementation bugs, most programming errors in our implementation were detected in only 3 runs. A symbolic debugging facility, of course, would be welcome.
- **General programming facilities.** Lynx is a bit parsimonious when it comes to offering the features of a traditional programming language. The absence of dynamic storage allocation, pointers, and a clean library interface is conspicuous. The facility for separate compilation is not well documented.

7.5. Current status of the implementation

Query processing is completely functional, but queries need to be hardwired in the Init module. The simple restructuring mechanism (the one that simply gets all data from children and rebuilds the subtree) is also working for most inputs.

7.6. Appendix: Pseudo-code

```
module NodeProcess(LeftChild,RightChild,ParentLink:link);

--* i node.h : global type and const declarations

entry   ServerNode (NodeNum:integer; MyData:sumbitqueue; parameters:param);
var
        -- the data structures used by each node.
        bound : BoundsType;
        MyDiscr: discriminator;
        current: LeftOrRight;
        BalanceTolerance : integer;
        death,IamLeaf : boolean;
        LeftNodeNum, RightNodeNum : posint;
        MyState, LeftState, RightState : NodeStatus;
        WorkData, LeftData, RightData : SubmitQueue; -- static allocation

        function FindDiscriminant (bounds: BoundsType):discriminator;
        begin
                -- determines as to which key should be the discriminant
        end FindDiscriminant;

        procedure Sort (var Data:SubmitQueue);
        begin
                -- sort MyData on the distance field and put the sorted
                --      records in WorkData
        end Sort;

        procedure CalculateDistance (query:datarec; var Queue:SubmitQueue);
        begin
                -- calculate the dissimilarity between the query record and each of
                -- the records in the Queue
        end CalculateDistance;

        --    entry points provided to each child and to the parent.

        entry GetNearest (NumRecs:submitnum; QueryRec:datarec ) :
                SubmitQueue,boolean;
        var num : integer; done: boolean;

                procedure MergeNearest;
                begin
                 -- Merge the nearest neighbor queues obtained from the left
                 --      and right children
                end MergeNearest;

                function BallWithinBounds: boolean;
                        -- determine whether it is necessary to continue the search
                begin
                        foreach i in [1..NumKeys] do
                                if (Co_distance(bound.low[i],QueryRec.key[i])
                                < = WorkData.distance[WorkData.size])
                                        or
                          (Co_distance(bound.high[i],QueryRec.key[i])
```

```
                    < = WorkData.distance[WorkData.size]) then
                        return (false);
                    end;
                end;
                return (true);
            end BallWithinBounds;

            function BoundsOverlapBall: boolean;
            var sum:integer;
              -- determine whether it is necessary to consider the records
              --  on the side of the partition opposite the query record
            begin
                foreach i in [1..NumKeys] do
                    if (QueryRec.key[i] < bound.low[i]) then
                      sum := sum + Co_distance(QueryRec.key[i] , bound.low[i]);
                    elsif (QueryRec.key[i] > bound.high[i]) then
                      sum := sum + Co_distance(QueryRec.key[i] , bound.high[i]);
                    end;
                    if sum > WorkData.distance[WorkData.size] then
                      return (false);
                    end;
                end;
                return (false);
            end BoundsOverlapBall;

begin  -- GetNearest
  if IamLeaf then
                if NumRecs > MyData.size then
                    num:= MyData.size;
                else
                    num:= NumRecs;
                end;
                CalculateDistance (QueryRec,MyData);
                Sort(MyData);
                if BallWithinBounds then
                    done := true;
                    reply (WorkData,done);
                else
                    done:= false;
                end;
  else
                if QueryRec.key[MyDiscr.key] < = MyDiscr.partition then
                    connect GetNearest(NumRecs, QueryRec| LeftData,done)
                        on LeftChild;
                    if done then
                        reply (LeftData,done);
                    else
                        if BoundsOverlapBall then
                            connect GetNearest(NumRecs,QueryRec| RightData,done)
                                on RightChild;
                            if done then
                                reply (RightData,done);
                            else
```

```
                                                    MergeNearest;
                                            end;
                                    end;
                            end;
                    else
                            -- same as above, reversing left and right.
                    end;
            end;
            if not done then
              done := BallWithinBounds;
              reply (WorkData, done);
            end;
    end GetNearest;

    entry Submit:SubmitQueue;
      -- The parent invokes submit when it thinks that the subtree
      --    needs to be restructured
      -- This entry returns its data to the parent and sets death true,
      --      so that the current activation of ServerNode dies.
    begin
            if not IamLeaf then
                    connect Submit ( |LeftData) on LeftChild;
                    connect Submit ( |RightData) on RightChild;
                    foreach i in [1..LeftData.size] do
                            MyData.rec[i] := LeftData.rec[i];
                    end;
                    foreach i in [1..RightData.size] do
                            MyData.rec[LeftData.size+i] := RightData.rec[i];
                    end;
                    MyData.size := LeftData.size + RightData.size + 1;
            end;
            reply (MyData);
            death:=true;
    end Submit;

    entry ProcessAndSplitData;
            -- used while initial creation and also while restructuring
            --      after a delete or insert

            function WhichPowerOf2(num:integer):integer;
            var x,currentpower:integer;
            begin
             -- returns x such that 2**(x-1) < num <= 2**x ;
            end WhichPowerOf2;

    begin -- ProcessAndSplitData
            -- determine the bounds within which the records in MyData lie.
            MyState.capacity := MAXSUBMIT / WhichPowerOf2(NodeNum);
            MyState.CurNum := MyData.size;
            BalanceTolerance := MyState.capacity/Tol_factor;
            IamLeaf := TRUE;
            if MyData.size > BUCKETSIZE then
                    IamLeaf := FALSE;
```

```
                MyDiscr := FindDiscriminant(bound);
                -- set distance to the discriminant key value for each rec
                Sort(MyData);
                MyDiscr.partition:=  -- the median of the sorted records
                        MyData.rec[MyData.rank[MyData.size/2]].key[MyDiscr.key];
                -- split MyData into LeftData and RightData
                -- update LeftStatus and RightStatus
                connect ServerNode(LeftNodeNum, LeftData, parameters | )
                        on LeftChild;
                connect ServerNode(RightNodeNum, RightData, parameters | )
                        on RightChild;
                reply;
        end;
end ProcessAndSplitData;

entry Insert ( NewRec: datarec );
begin
        -- update status and node bounds if necessary due to the new record
        if IamLeaf then
                -- insert the NewRec in MyData
                -- there can not be an overflow case
                --   since the parent takes precaution as shown below.
        else
                if
                        -- the difference between the number of records in the
                        -- children is more than the BalanceTolerance
                or
                        -- the maximum spread in the current bounds is greater
                        -- than the current spread of the discriminant key;
                then
                        connect Submit ( |LeftData) on LeftChild;
                        connect Submit ( |RightData) on RightChild;
                        -- merge the LeftData and RightData, add the NewRec
                        -- and split again
                        call ProcessAndSplitData;
                end;
        end;
end Insert;

entry Delete( OldRec: datarec ): boolean;
var Done : boolean;
begin
        if IamLeaf then
                -- delete the OldRec from MyData if existent and return status
        else
                if OLdRec.key[MyDiscr.key] <= MyDiscr.partition then
                        connect Delete(OldRec|Done) on LeftChild;
                else
                        connect Delete(OldRec|Done) on RightChild;
                end;
                if successful Delete then
                        -- get NewBounds from that child
                        if
```

```
                                        -- the difference between the number of records in the
                                        -- children is more than the BalanceTolerance
                                or
                                        -- the maximum spread in the current bounds is greater
                                        -- than the current spread of the discriminant key;
                                then
                                        connect Submit ( |LeftData) on LeftChild;
                                        connect Submit ( |RightData) on RightChild;
                                        -- merge LeftData and RightData and split;
                                        call ProcessAndSplitData;
                                end;
                        end;
                end;
                return(Done);
        end Delete;

        begin -- ServerNode
                death := false;
                -- set local variables from parameters
                call ProcessAndSplitData;
                bind ParentLink to GetNearest, Submit, Share, Insert, Delete;
                reply;
                await (death); -- to be set true when the Parent invokes Submit
        end ServerNode;

begin -- NodeProcess
        termination := false;
        bind ParentLink to ServerNode;
        await(termination); -- currently infinite wait
end NodeProcess.
```

8. A distributed implementation of the Waltz algorithm

One well-understood problem in scene analysis is to translate a line drawing into statements about the number of objects, their relationships and their properties. The first step in the analysis consists of determining the types of lines in the drawing. Depth-first or breadth-first search for correct labelling of lines suffers from combinatorial explosion for scenes of some size. The Waltz algorithm circumvents this problem by pruning junction arrangements that are demonstrably wrong[Waltz75]. Our goal was to develop a distributed algorithm based on the Waltz algorithm and to implement it using Lynx on the Crystal multicomputer.

8.1. The distributed Waltz algorithm

Every junction in the line drawing is modelled by a process in the distributed algorithm. The lines between junctions are modelled by communication links between the processes. The algorithm for each process is as follows:

```
NodeProcess =
        begin
                CurrentChoice := All possible labellings for this node;
                SendDataToNeighbors;
                MessageCount := 0;
                repeat
                        while there are messages from Neighbors do
                          begin
                                ProcessMessage;
                                inc(MessageCount);
                          end;
                        if Change of State then
                          SendDataToNeighbors;
                until Termination;
                report results;
        end.

SendDataToNeighbors =
        begin
                foreach i in Neighbors do
                        SendMessage(set of possible labels for Line(Self,i))
                          to process i;
                -- Line(Self,i) is the line between the node sending data and node i
        end;

ProcessMessage =
        begin
                NewChoices := {};
                foreach i in CurrentChoice do
                        if Compatible(Message.Label,Labels[i]) then
                          NewChoices := NewChoices + {i};
                if NewChoices < > CurrentChoice then
                        begin
                          Change of State := true;
                          CurrentChoice := NewChoices;
                        end;
        end;
```

Neighbors is the set of processes that model adjacent junctions in the line drawing. Each line can be labelled with four different labels. These labels along with physically possible junction arrangements

are shown in the first appendix. Each node process is initialized with the set of neighbors, the junction type it models and the initial constraints (if any from boundary conditions) on the lines connected to the junction.

The node process initializes CurrentChoice to the set of all possible junction arrangements depending on the junction type and initial line label constraints. It sends the initial choices to all its neighbors before it starts processing incoming messages. Further messages to the neighbors are sent only if there is a change in the current state of the node represented by CurrentChoice.

A change of state always implies a reduction in the number of elements in CurrentChoice. If the CurrentChoice becomes empty, then further change of state is not possible for this node. All node processes eventually reach a state of equilibrium where no further changes in state are possible and no messages are sent out of any node. This equilibrium state is the termination condition.

The termination algorithm we developed is centralized. A central process communicates with every node process.

```
Termination =
        begin
                SendMessage(Change of State,MessageCount) to Central Process;
                Receive(Status) from Central Process;
                if Status = Done then
                        return true
                else
                        MessageCount := 0;
                        Change of State := False;
                        return False
                end;
        end;

Central Process =
        begin
                foreach node i do
                        Count[i] := number of neighbors of i;
                repeat
                        Receive(Change of State,MessageCount) from any node i;
                        Count[i] := Count[i] - MessageCount;
                        if Change of State then
                          foreach j in Neighbors[i] do
                                Count[j] := Count[j] + 1;
                        Finished := Count[i] = 0 for all nodes i;
                        SendMessage(Finished) to node i
                until Finished for all nodes
        end.
```

No formal proof has been derived for the termination algorithm. Intuitively, the Count represents the number of messages yet to be processed by a node. If this Count becomes zero for all nodes, then no more messages are sent out of any node since a change of state can possibly occur only in response to an incoming message. Therefore, the nodes must have reached the equilibrium state.

After termination, if each node has exactly one element in CurrentChoice then the line drawing has an unique physical interpretation. If the CurrentChoice is empty, then there is no physical interpretation for the line drawing. On the other hand, if CurrentChoice contains more than one element, the line drawing is ambiguous.

8.2. Implementation using Lynx

The implementation written in pseudo-Lynx can be found in the second appendix. The program consists of two Lynx modules: node and starter. The node implements the vertex process of the algorithm. The starter has three functions:

(1) Send initial data to each node process.

(2) Serve as the central process for termination.

(3) Gather results from node processes after termination.

Node Module

This is a straightforward translation of the algorithm shown above into Lynx. The node, as part of its initialization phase, gets the junction data from the starter. An entry is defined for receiving messages from the neighbors. All links to the neighbors are bound to the same entry. When a message arrives on a link, the message is compared to the previous message on the same link and, if different, stored in a data structure corresponding to the link. The index of the link is added to the set ArrivedMessages. This set is used in the inner loop of the main body to determine whether there are messages that need to be processed.

This implementation allows a message on a link to overwrite a previous message even though the previous message has not yet been processed. Thus a node always processes the most recently arrived message from any neighbor.

For detecting termination, the node sends a message to the starter after processing all arrived messages. The reply is a boolean indicating whether the global termination condition has been achieved. If so, the node sends a message to the starter reporting the results for the junction represented by the node.

This code in this module is independent of the problem (that is, scene analysis) and needs to be compiled just once. It can be used for other relaxation algorithms as well.

Starter Module

This module initially creates a thread of control, NodeManager, for each node process. This thread of control remains active until node processes terminate. The NodeManager sends the initial data, accepts termination queries from the corresponding node process, changes the Count values for relevant nodes and determines whether termination condition has been achieved and reports it to the node process. After termination condition has been achieved, it receives the final state of the node and stores it.

After all the NodeManagers have terminated, the starter module checks the stored results from each node and reports whether the line drawing is physically realizable, impossible or ambiguous. If there is an unique physical interpretation, the line labels are printed.

Certain code sections of this model depend on the problem being solved. For example, the number of NodeManagers to be started and the initial data to be sent to each node process are problem dependent. Hence this module needs to be compiled for every new problem. However, the creation of the problem dependent code has been automated. The problem dependent code sections in the starter module are included at compilation time. A generator program reads the problem description and creates the files consisting of problem dependent code sections. Details on usage can be found in the User Guide section.

8.3. Limitations

This implementation assumes that a node can have at most three neighbors. One way to remove this limitation is to have the generator program create code sections that depend on the maximum number of neighbors of any node. However, both the node and starter modules would be affected and would need to be compiled for every problem.

Another solution is to declare a large number of links and use only those that are required for any problem. However, even the unused links which appear in the module parameter list must be specified in the connector file. Currently the unused links are bound to the links of a dummy process whose code is created by the generator program. Increasing the total number of available links would result in a large number of unused links and would consequently increase the number of links to the dummy process. However, the Lynx environment fails during link initialization if more than 9 links appear in the module parameter list. (This limit has been relaxed to 24 in later implementations of Lynx.) The other option of creating many dummy processes is not very appealing. Since the labelling scheme selected is applicable only when junctions have at most three neighbors, this limitation does not seriously affect our application.

The current restriction of nine links creates another limitation. Since the starter process communicates with every node process, it needs one link to every node process. Consequently, the problem must have less than 9 junctions. This is a serious limitation, since most non-trivial drawings have more than 9 junctions.

8.4. Comments on Lynx

The usage of the threads of control feature of Lynx is not naturally suggested by the distributed Waltz algorithm. We needed to use threads since Lynx does not have a non-deterministic loop structure with guarded commands, which would implement the algorithm more precisely and naturally. The usage of the threads of control only approximates a precise implementation of the algorithm. However, the use of this feature did not complicate the program. The mutual-exclusion property of threads made synchronization trivial without any explicit synchronization code.

In the node module, all links were bound to a single entry. The index associated with the link causing the invocation had to be found by search. This binding pattern was designed to avoid multiple declarations of the entry (one for each link) differing in only an index value.

A solution to this problem is to allow the definition of a generic entry in Lynx. A generic entry is then instantiated with parameters. This facility would not create any threads of control. It provides a concise way of defining multiple entries that share the same code and differ in the values of certain parameters.

Finally, the Lynx compiler needs to be tested more thoroughly since the program exposed many bugs.

8.5. User Guide

Four files are required.

(1) Startgen (compiled from Startgen.p)

(2) Node.o (compiled from Node.x)

(3) Starter.x

(4) Dummy.x

Solving a problem consists of the following steps:

(1) Create a problem file. The syntax for the problem description is as follows:

<junction-type <NeighborJunctionNo InitialLabelOfLine>*d eoln>*n

where d = number of neighbors
n = number of junctions in the drawing
eoln = end of line
InitialLabelOfLine = 0 if not initially known.
The description of junction i must appear on line i.

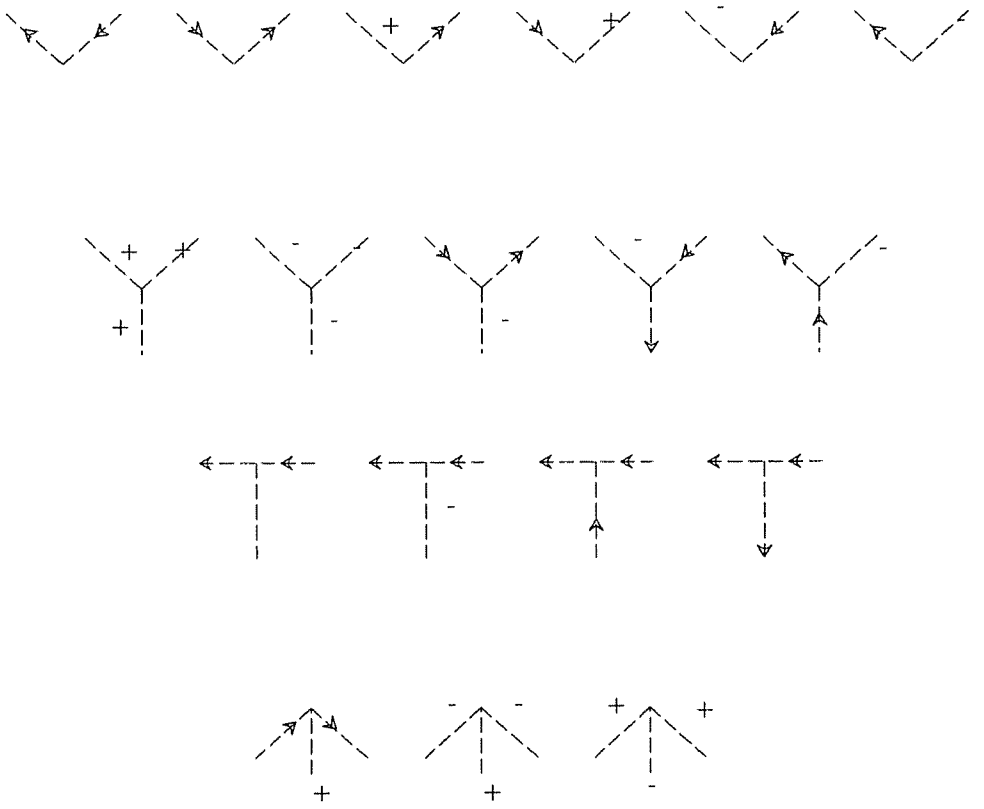
The values for junction type and label can be found in the first appendix.

(2) Run Startgen. Input the name of the problem file and the complete path specification where the node and starter object files can be found (to be used in the generated connector file). Startgen creates the connector file called Connect and the include files.

(3) Compile starter.x and dummy.x. The include files must be in the current working directory.

(4) Use Nci (the Crystal nuggetmaster command interpreter) and Vterm (the crystal virtual terminal package) to run the program.

8.6. Appendix 1: Lines and junctions



8.7. Appendix 2: Pseudo-Lynx code of the implementation

```
module Node(StarterLink,Neighbor1,Neighbor2,Neighbor3 : link);

    entry InformNeighbor(NewData : LineData);
    -- neighbors call this operation to report changes in their state.
    begin
        Index := IndexOf(CurLink); -- search for the index of current link
        if NewData <> PreviousData[Index] then
            -- state has changed since the last message on this link
            ArrivedMessages := ArrivedMessages + {Index};
            NextData[Index] := NewData;
            -- store for processing.Previous data may be overwritten
            -- else ignore message
        end;
        inc(MessageCount);  -- for use in termination
        reply;
    end InformNeighbor;

    procedure InitData;  -- Initialize this node
    begin
        Connect GetNodeData( | JunctionType, InitLabel ) on StarterLink;
        -- Initialize data structures with different possible
            -- alternatives for this junction;
    end InitData;

    procedure CheckMessage;  -- Process a message

        procedure CheckData(LineNo : integer);
        begin
            Impossibles := {};
            foreach Alternative in CurrentChoice do
                if Incompatible(Alternative) then
                    -- this labelling choice is inconsistent with its neighbor
                    Impossibles := Impossibles + {Alternative};
                end;
            end;
            if Impossibles = {} then
                -- No change in state. All current choices still valid
                Changed := Changed or False;
            else
                Changed := True;
                CurrentChoice := CurrentChoice - Impossibles;
                -- remove impossible junction arrangements
                if CurrentChoice = {} then
                    -- no physical interpretation for this junction
                    ImpFigure := True;
                else
                    foreach Alternative in CurrentChoice do
                        NumChoices := NumChoices - 1;
                    end;
                end;
            end;
        end CheckData;
```

```
        begin -- CheckMessage
                if not ImpFigure then
                        foreach Message in ArrivedMessages do
                                CheckData(Message) -- Check for Compatibility
                                if ImpFigure then
                                        exit; -- no more work
                                end;
                        end;
                end;
                ArrivedMessages := {}; -- all messages processed
                PrevData := NextData;
        end CheckMessage;

        procedure SendData;
        begin
                foreach neighbor do
                        Alternatives := possible labels for the line to the neighbor;
                        connect InformNeighbor(Alternatives | ) on link to neighbor;
                end;
        end SendData;

begin -- node
        InitData;
        MessageCount := 0;
        foreach neighbor do
                bind link to neighbor to InformNeighbor;
        end;
        Finished := False; Changed := True; ImpFigure := Flase;
        repeat
                if Changed then SendData; end;
                Changed := False;
                while ArrivedMessages < > {} do
                        CheckMessage;
                end;
                connect ReportStatus(Changed,MessageCount | Finished ) on StarterLink;
                -- check for termination condition
        until Finished;
        connect SendResults(Status,Labels | ) on StarterLink;
end Node.

module Starter --* include mheader.i
-- contains (link1,link2,....,linkn); where n= no. of junctions

        entry NodeManager(NodeNumber : NodeNumRange; -- Junction Number
        var
                NodeConnector : link; -- Link to the node process
                JunctionType : Jtype; -- 0 .. 3
                Neighbors : NodeSet; -- Set of neighbor junction numbers
                Label1,Label2,Label3 : LabelType); -- Initial labels
        begin
                accept GetNodeData on NodeConnector;
                reply (JunctionType, InitLabel);
                -- termination loop
```

```
                repeat
                        accept ReportStatus(Changed,MessageCount) on NodeConnector;
                        Count[NodeNumber] := Count[NodeNumber] - MessageCount;
                        -- MessageCount messages have been processed by the node
                        if Changed then
                                -- the node will have sent messages to its neighbors
                                foreach Neighbor of NodeNum do
                                        inc(Count[Neighbor]);
                                end;
                        end;
                        Finished := Count[i] = 0 for all nodes; -- termination condition
                        reply (finished);
                until Finished;
                -- Node will also terminate and send results. So
                accept SendResults(Status,FinalLabel) on NodeConnector;
                reply;
                store result;
                ActiveNodes := ActiveNodes - {NodeNumber};
        end NodeManager;

begin -- Starter
        ActiveNodes := {1..MaxNodes};
        --* include Count.i
        -- initialize Count Values for each node
        --* include Call.i
        -- Calls to Node Manager. One for each node
        await ActiveNodes = {};
        -- wait until all nodes have terminated
        print results;
end Starter.
```

9. References

83. , "Optimization by Simulated Annealing," *Science* **220**(4598) pp. 671-680 (May 13, 1983).

Artsy84.
Artsy, Y., H.-Y. Chang, and R. Finkel, "Charlotte: design and implementation of a distributed kernel," Computer Sciences Technical Report #554, University of Wisconsin - Madison (August 1984).

Bayer77.
Bayer, R. and M. Schkolnick, "Concurrency of operations on B trees," *Acta Informatica* **9** pp. 1-21 (1977).

Chen85.
Chen, H. H., *Incremental computation of topological properties in distributed networks* (Ph.D. thesis) (1985).

Conery81.
Conery, J. S. and D. F. Kibler, "Parallel interpretation of logic programs," *Proc. of the 1981 ACM Functional Programming and Computer Architecture Conference*, pp. 163-170 (1981).

DeWitt84.
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).

Ellis85.
Ellis, C. S., "Distributed data structures: A case study," *Proc. 5th International Conference on Distributed Computing Systems*, pp. 201-208 (May 1985).

Felten85.
Felten, E., S. Karlin, and S. Otto, "The traveling salesman problem on a hypercubic, MIMD computer," *Proc. 1985 International Conference on Parallel Processing*, pp. 6-10 (August 1985).

Fishbu82.
Fishburn, J. A. and R. A. Finkel, "Quotient networks," *IEEE Transactions on Computers* **C-31**(4) pp. 288-295 (April 1982).

Friedm77.
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).

Gallag83.
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).

Kung78.
Kung, H. T. and C. E. Leiserson, "Systolic Arrays (for VLSI)," *Symposium on Sparse Matrix Computations*, pp. 256-282 (Duff, I. S. and Stewart, G. W., editors) (November 1978).

Kung82.
Kung, H. T., "Why systolic architecture?," *IEEE Transactions on Computers*, pp. 37-46 (January 1982).

Lehman81.
Lehman, P. and S. B. Yao, "Efficient locking for concurrent operations on B trees," *ACM Transactions on Database Systems* **6**(5) pp. 650-670 (December 1981).

Liu68.
Liu, C. L. and Introduction to combinatorial mathematics, , McGraw Hill, New York (1968).

Mohan83.
Mohan, J., "Experience with two parallel programs solving the traveling salesman problem," *Proc. Int. Conf. Parallel Processing*, pp. 191-193 (1983).

Scott85.
Scott, M. L., "Design and implementation of a distributed systems language," Ph. D. Thesis, Technical Report #596, University of Wisconsin - Madison (May 1985).

Tamura84.
Tamura, N. and Y. Kaneda, "Implementing parallel prolog on a multi-processor machine," *IEEE 1984 International Symposium on Logic Programming*, pp. 42-48 (1984).

Waltz75.
Waltz, D., "Understanding line drawings of scenes with shadows," pp. 19-92 in *The psychology of computer vision*, ed. P. H. Winston, McGraw-Hill (1975).

Warren84.
Warren, D. S. et al., "Implementing parallel prolog on a multi-processor machine," *IEEE 1984 International Symposium on Logic Programming*, pp. 122-21 (1984).