Experience with Crystal, Charlotte and Lynx
Third Report

by

Raphael Finkel
Gautam Das, Dhruva Ghoshal
Kamal Gupta, Ganesh Jayaraman
Mukesh Kacker, Jaspal Kohli
Viswanathan Mani, Ananth Raghaven
Michael Tsang, Sriram Vajapeyam

Computer Sciences Technical Report #673
November 1986

Experience with Crystal, Charlotte, and Lynx
Third Report

Raphael Finkel
Gautam Das
Dhruva Ghoshal
Kamal Gupta
Ganesh Jayaraman
Mukesh Kacker
Jaspal Kohli
Viswanathan Mani
Ananth Raghavan
Michael Tsang
Sriram Vajapeyam

Computer Sciences Department
University of Wisconsin–Madison

Abstract

This paper describes several recent implementations of distributed algorithms at Wisconsin that use the Crystal multicomputer, the Charlotte operating system, and the Lynx language. This environment is an experimental testbed for design of such algorithms. Our report is meant to show the range of applications that we have found reasonable in such an environment and to give some of the flavor of the algorithms that have been developed. We do not claim that the algorithms are the best possible for these problems, although they have been designed with some care. In several cases they are completely new or represent significant modifications of existing algorithms. We present distributed implementations of the stable marriage problem, finding roots of an equation, Gaussian elimination, finding minimal dominating sets, PLA folding, the Hough transform, the Banker's algorithm, the n-queens problem, and quicksort. Together with our previous two reports, this paper leads us to conclude that the environment is a valuable resource and will continue to grow in importance in developing new algorithms.

This work was supported by DARPA contract N00014-85-K-0788 and NSF grant MCS-8105904. 1]

TABLE OF CONTENTS

1. Introduction

At the University of Wisconsin — Madison, we have built an environment for experimenting with distributed programs. This paper is a sequel to two previous ones, in which we described projects that use Crystal, Charlotte, and Lynx [Finkel86, Finkel86b].

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 [Artsy86] is an experimental distributed operating system that can run in a Crystal partition of any size. Programs running under Charlotte communicate through **links**, which are two-way channels whose ends can be sent in messages (and thus relocated to other processes). The Charlotte user interface consists of a command interpreter process through which one can enter interactive commands to start processes, read command scripts, or interpret a connector file, which specifies 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 a connector file are presented as arguments to the main procedure of a Lynx module. Other links can be created and disseminated dynamically. They can be **bound** to entry points, which are like function declarations. If a process executes a remote call through a link bound to an entry point, a new thread of control is created at the destination process to service that call. Threads of control within the same process may share memory. They do not execute in parallel; the current thread continues until it blocks. We call this the **mutual-exclusion** property of threads.

§ Unix is a registered trademark of Bell Laboratories.

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

2. Distributed Banker's algorithm

Experimenter: Ananth Raghavan

2.1. Introduction

All operating systems need a resource allocation policy. One essential duty of a resource allocation policy is to avoid deadlock. Various policies have been developed, such as one-shot allocation, hierarchical allocation, and the Banker's algorithm. We describe the implementation of two versions of a distributed Banker's algorithm [Madduri85].

This report is organized as follows: Section 2 describes the Banker's algorithm and the extensions required to implement it on a distributed system; Section 3 describes the centralized algorithm; Section 4 describes the hierarchical algorithm; Section 5 outlines an implementation in Lynx of these two algorithms; Section 6 describes the performance results obtained from the two algorithms by running the Lynx programs on Crystal; Section 7 summarizes our experiences with Lynx and Crystal; Section 8 points towards further research in this field.

2.2. The Banker's algorithm and minimum need

The Banker's algorithm [Finkel86c] is a liberal resource allocation algorithm known not to lead to deadlock.

The algorithm requires that each process make a **claim** before it acquires any resources. This claim indicates the greatest number of resources that the process will need at any time. The claim may not exceed the total number of resources available. There may be several classes of resources; the process must make a claim for each class.

The resource manager calculates the allocation state for each resource class and decides whether the state is **realizable** or not. An allocation state is realizable if

(1) No one claim is for more than the total resources available.

(2) No process is holding more than its claim.

(3) The sum of all the resources held in a resource class is not more than the total number of resources in that class.

Otherwise the state is unrealizable.

A realizable state is **safe** if it cannot possibly lead to deadlock, that is, a sequence of job completions exists even if each job demands its full claim of resources.

The banker's algorithm considers a request for resources from a process and checks to see whether the state is safe if that request were to be granted. If so it grants the request. Otherwise it denies the request. For simplicity we will consider only one class of resources.

The banker's algorithm only decides if a state is safe or unsafe. Madduri extends the Banker's algorithm by the concept of **minimum need**, which measures the extent of safety or non-safety [Madduri85].

- **Cash** indicates the minimum number of free resources.
- A **minimum need** is the minimum number of resources required to make a given allocation state safe.
- **Holdings** is the number of resources a job holds at present.
- **Need** is the number of resources a job may request in the future.

$$need = claim - holdings$$

The calculation of minimum need is discussed in the following sections. Once the minimum need is calculated, it can be compared with the cash to decide safety.

- If the cash is greater than or equal to the minimum need then the given allocation state is **safe**.

Whenever a request for resources is processed we calculate the minimum need assuming that the request is granted and check for safety. If the state becomes unsafe, then the request is denied.

The following two sections describe the algorithms used to calculate minimum need in a distributed system where resources may be scattered among many computers.

2.3. The centralized algorithm

In this algorithm we have a have a master process to which all demands for claims, requests, and releases are made. Requests come from jobs on the different processors in the system. The master process remembers the safe sequence by storing it in a binary tree. All jobs are kept at the leaves. The tree is sorted in order of need. The leaves read from left to right give the safe sequence.

All nodes of the tree contain both need and holdings information. Internal nodes use that information to summarize the status of all jobs within their subtree. The **need** of an internal node represents the **minimum need** of all jobs within that subtree. The **holdings** of an internal node are the sum of all the resources held by jobs within that subtree.

$$need_{internal\ node} = \max \left\{ need_{left}\ ,\ need_{right} - held_{left} \right\}$$

$$holdings_{internal\ node} = holdings_{right} + holdings_{left}$$

Hence the need of the root of the tree gives us the **minimum need.** This must be compared with **cash** to check for safety. Figure 1 shows such a tree.

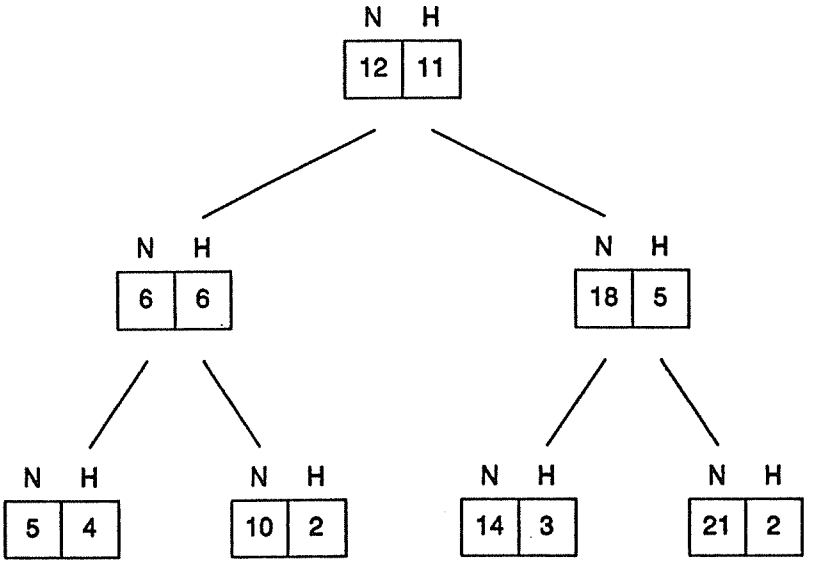


Figure 1. A minimum-need tree

A job is uniquely identified by a pointer to the leaf representing it. As a job's need and holdings values change, its position in the tree may change. Hence it is necessary to be able to insert a node, delete a node and find the position among the leaves where insertion is to be done. When a request is made for resources, the tree is modified tentatively as if the request were granted. If the root indicates safety the request is granted. Otherwise the request is denied and the tree is restored to its original form. Here is a description of this algorithm in pseudo-code.

```
type
        Treeptr  = ^TreeNode;
        CashType = integer;
        TreeNode = record
                        need   : CashType;
                        holdings   : CashType;
                        parent : Treeptr;
                        left : Treeptr;
                        right : Treeptr;
        end;

var
        Jobs = array [1..NumJobs] of Treeptr;
```

```
procedure Propagate (N : Treeptr);
-- change the status of the internal nodes to reflect changes at the leaves.
begin
        P := N's parent;
        loop
                calculate P.holdings;
                calculate P.Need;
                if P = root then exit; end;
                P := P's parent;
        end;
end Propagate;

function findjob (jobid:integer): Treeptr;
-- find the leaf that represents a particular job.
-- pointers to leaves are stored in array Jobs
begin
        return (Jobs[jobid]);
end findjob;

function findleaf (k : CashType): Treeptr;
-- find the leaf where a job should be inserted in order by need.
begin
        repeat
                L := Root;
                if k < L.Need then
                        L := L.left;
                else
                        L := L.right;
                end;
        until L is a leaf;
        return (L);
end findleaf;

procedure insert (S: Treeptr; jobid:integer);
-- insert a job at a leaf.
begin
        L := findleaf (S.Need);
        Replace L by a new node P and make L and S its children;
        Propagate(S);
        Jobs[jobid] := S;
end insert;

procedure delete (jobid : integer);
-- remove a leaf from the tree.
begin
        L := Jobs[jobid];
        P := L's sibling;
        replace L's parent with P;
        Propagate(P);
        Jobs[jobid] := nil;
end delete;
```

```
function safety : Boolean;
begin
        return(cash >= Root.Need);
end safety;
```

2.4. Hierarchical algorithm

This algorithm imposes a logical hierarchy on the processors, represented by a binary tree. Each leaf is a computer running its own processes and owning some resources. Internal nodes are supervisory processors with no other work except supervising their child nodes and reporting status summaries up the hierarchy.

Each node in the tree maintains a status record with fields for cash, minimum need, and holdings. For a leaf node, these fields refer to the status of the processors. For an internal node, these fields refer to the summary of the status of the entire subtree. The cash of an internal node is the sum of the cash at the leaves of the subtree. The holdings of an internal node are the sum of all resources held by jobs within that subtree. The minimum need of an internal node gives us the minimum need for the jobs in that subtree.

$$cash_{internal\ node} = cash_{left} + cash_{right}$$

$$holdings_{internal\ node} = holdings_{left} + holdings_{right}$$

$$minimum\ need_{internal} = \max \left\{ \min(M_{left}, M_{right}), \max(M_{left}, M_{right}) - \right.$$

$$\left. (\text{if} M_{left} < M_{right} \text{ then } H_{left} \text{ else } H_{right}) \right\}$$

where M = minimum need, H = holdings.

When a request for an allocation is made at a node, a new status based on fulfilling the request is tentatively computed. If the status is safe, the request is granted independently of the status of the rest of the system. Otherwise the node sends its tentative status to its parent. The parent examines the status of its children to see if its own subtree is safe. If so it grants the request. Otherwise it sends the status of its subtree up the hierarchy and so on up to the root. The root finally decides whether to grant or not. All denials are made at the root.

Each leaf can run the centralized algorithm to calculate its minimum need. When a new status is calculated at a leaf based on a request made by a job, the cash might become negative. If so, when the status is sent to the parent, the parent will try to borrow cash from its other child to make up the shortfall. If this other child is a leaf, it lends resources to the requesting leaf. If it is an internal node it passes the request to its children. If a request is ultimately denied, the borrowed resources stay with the requesting child and the statuses

are updated to reflect this redistribution of cash.

A node we are trying to borrow from might be busy trying to allocate requests originating in its own subtree. If so, its allocation state is locked, and borrowing requests are denied resources. Locking is needed to avoid deadlock between a borrowing request going down the tree and a new request going up.

This algorithm tends to be a little more conservative than the centralized algorithm. It might deny requests that would have been granted by the centralized algorithm. Conservatism comes from the fact that status is summarized before being sent up the tree.

Here is pseudo-code for the hierarchical algorithm.

```
type
        response = (grant,deny);

function ProcessRequest : response;
var answer : response;
begin
        await not allocatorbusy ;
        allocatorbusy := true;
        update localstatus to include the effect of the request;
        if localstate is safe then
                answer := grant;
        else
                connect ReqFromBelow(localstatus ¦ answer) on parent;
        end;
        return (answer);
        allocatorbusy := false;
end ProcessRequest;
```

```
entry ReqFromBelow (askerstatus ¦ answer);
begin
        await not allocatorbusy;
        allocatorbusy := true;
        if askerstate.cash < 0 then
                connect Borrow(amount_needed,helperstatus,amount_given) on
                        otherchild;
                combine askerstate,helperstate,amount_given to give localstate;
        else
                get status of otherchild;
                combine askerstatus,otherstatus to give localstatus;
        end;
        if localstatus is safe then
                answer := grant;
        else
                if we are the root then
                        answer := deny;
                else
                        connect ReqFromBelow(localstatus,answer) on parent;
                end;
        end;
        allocatorbusy := false;
end ReqFromBelow;

entry Borrow (amount_needed,nodestatus,amount_given);
begin
        if allocatorbusy then
                amount_given := 0;
                nodestatus := locked;
        else
                allocatorbusy := true;
                if we are a leaf then
                        if cash > 0 then
                                amount_given := min(cash,amount_needed);
                                update localstatus to show loss of cash;
                                nodestatus := localstatus;
                        end;
                else
                        connect Borrow(amount_needed,helperstatus,amountgiven) on
                                left;
                        if amount_needed < amount_given then
                                connect Borrow(amount_needed -
                                amount_given,helperstatus,amount_given) on right
                        end;
                        update localstatus;
                        find total amount_given;
                end;
                allocatorbusy := false;
        end;
end Borrow;
```

ProcessRequest is called on the leaf nodes whenever a request is made. If it is possible to grant the request locally it does so. Otherwise it calls

ReqFromBelow on its parent.

ReqFromBelow looks at the status of the requesting child. If necessary it calls **Borrow** on its other child to borrow resources. It combines the statuses of its children and then checks for safety. If safe the request is granted. If unsafe the request is sent further up. If the request reaches the root and is found unsafe then the request is denied.

Borrow checks to see if its node's allocation state is locked. If not it tries to borrow the resources. If Borrow is called on an internal node it sends the request further down. If it is called on a leaf it tries to lend the resources.

2.5. Implementation in Lynx

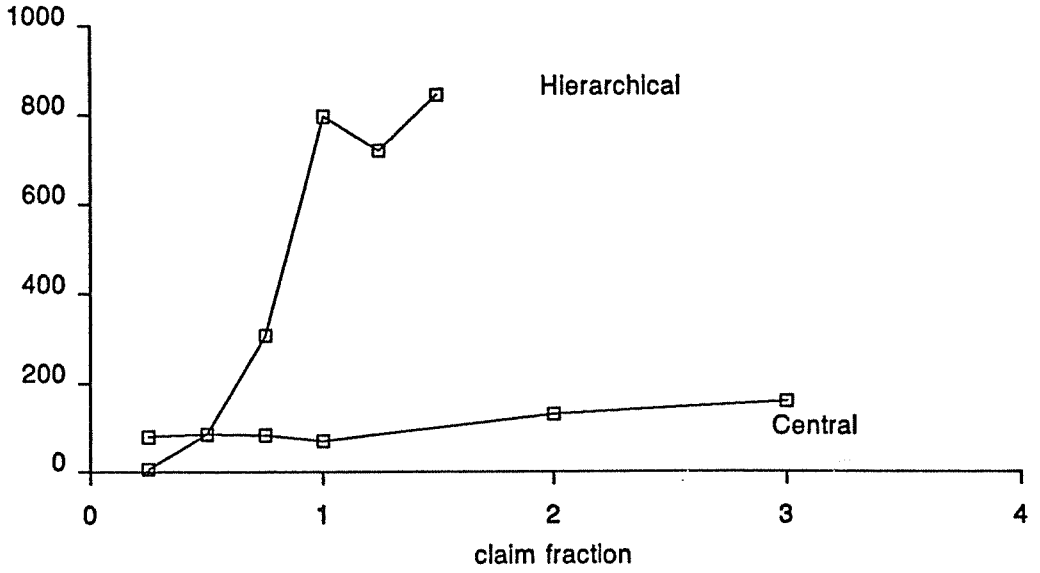
The implementation in Lynx follows the algorithms shown above very closely. Each job is represented as a separate thread of control. Ten threads are initialized on each leaf processor to start with. One of them is chosen randomly to execute. This thread chooses an event randomly from resource request, resource release and job termination. If it chooses resource request, it decides the number of resources to request randomly from its need. If it chooses to release resources, it decides the number of resources to release randomly from its holdings. If a request for resources is denied, the thread is blocked until it is able to acquire the resources. Every time a job releases resources on a node, any thread blocked waiting for resources on that node is unblocked and tries again to acquire the resources. After completing its event, the thread is blocked until it is chosen again. In case the event was job completion, the thread dies. All jobs have the same claim, which is a parameter of the simulation expressed as a fraction of the total number of resources available per machine.

In simulating the algorithms two configurations were used. One used four leaf processors and the other eight leaf processors. Each leaf processor is simulated as a Lynx process. For the centralized algorithm, one master process receives requests from the leaves and allocates resources. For the hierarchical algorithm, the 4 leaves are arranged in a 2-level tree and the 8 leaves in a 3-level tree. The whole simulation was performed on a Charlotte configuration of four machines.

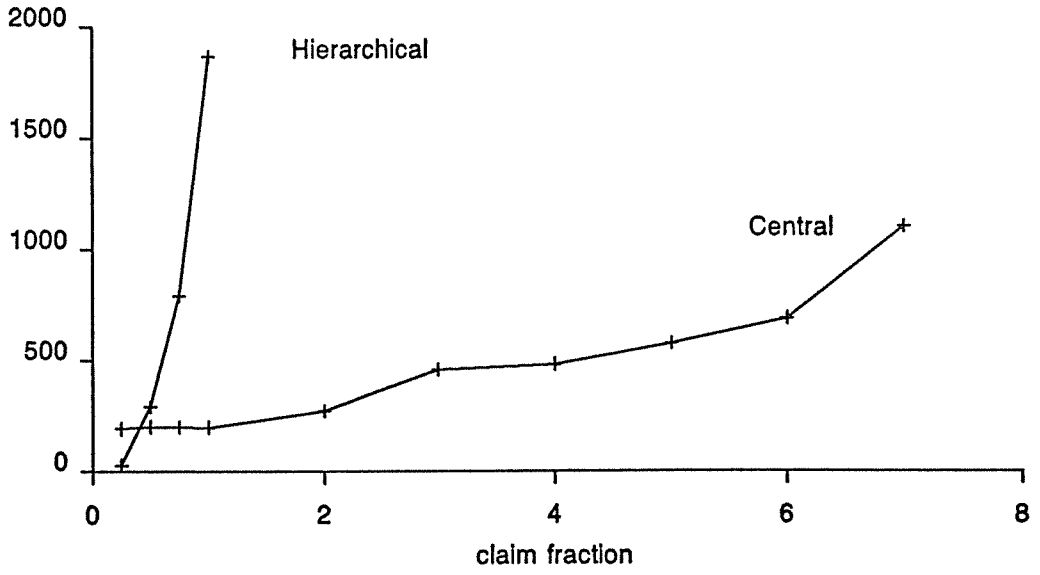
2.6. Performance results

Performance results were taken for both configurations with several claim fractions. We measured the number of messages required and the time taken for a request to be granted.

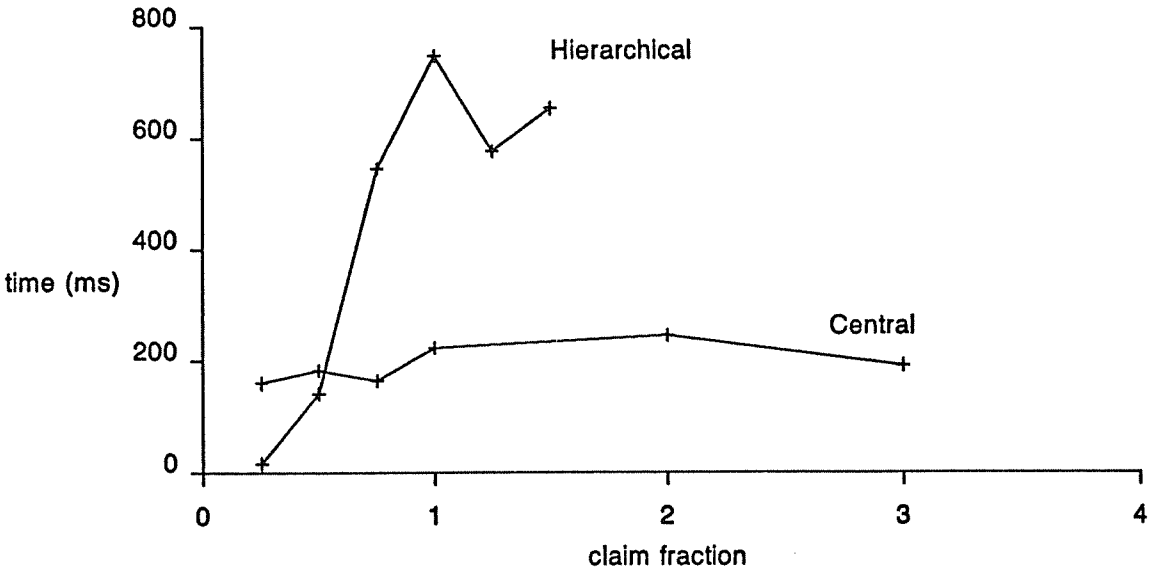
Graph 1 shows how the number of messages in the 4-leaf situation compares between the centralized and hierarchical algorithms over a range of claim fractions. Graph 2 shows the same measure for eight leaves. Graph 3 shows how the time required for requests to be granted in the 4-leaf situation compares between the centralized and hierarchical algorithms over a range of claim fractions. Graph 4 shows the same measure for eight leaves. Since the



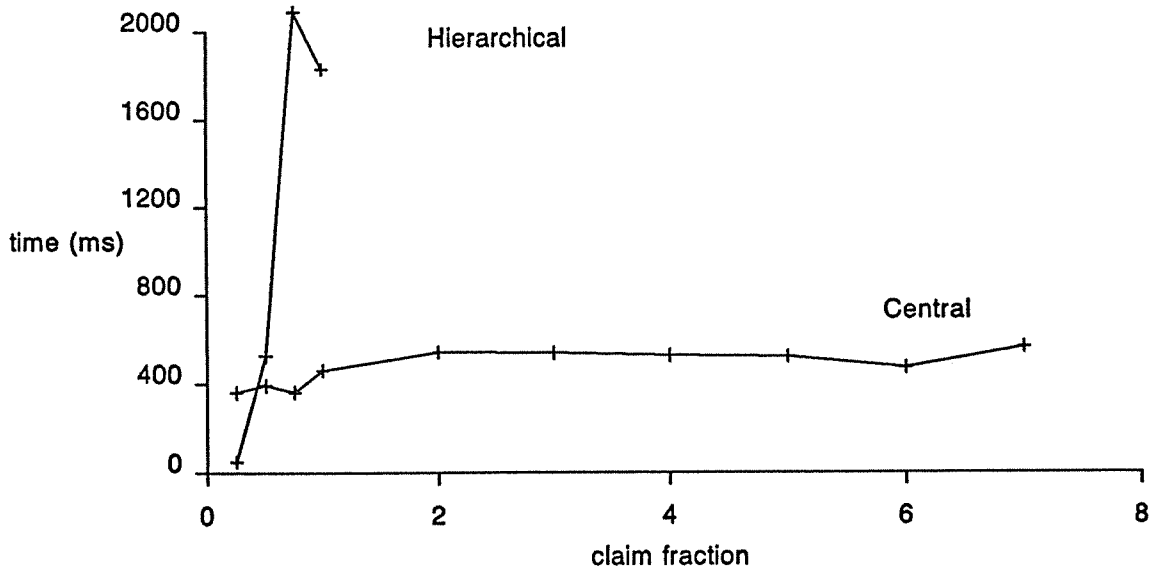
Graph 1. Messages versus claim fraction, four leaves



Graph 2. Messages versus claim fraction, eight leaves



Graph 3. Request time versus claim fraction, four leaves



Graph 4. Request time versus claim fraction, eight leaves

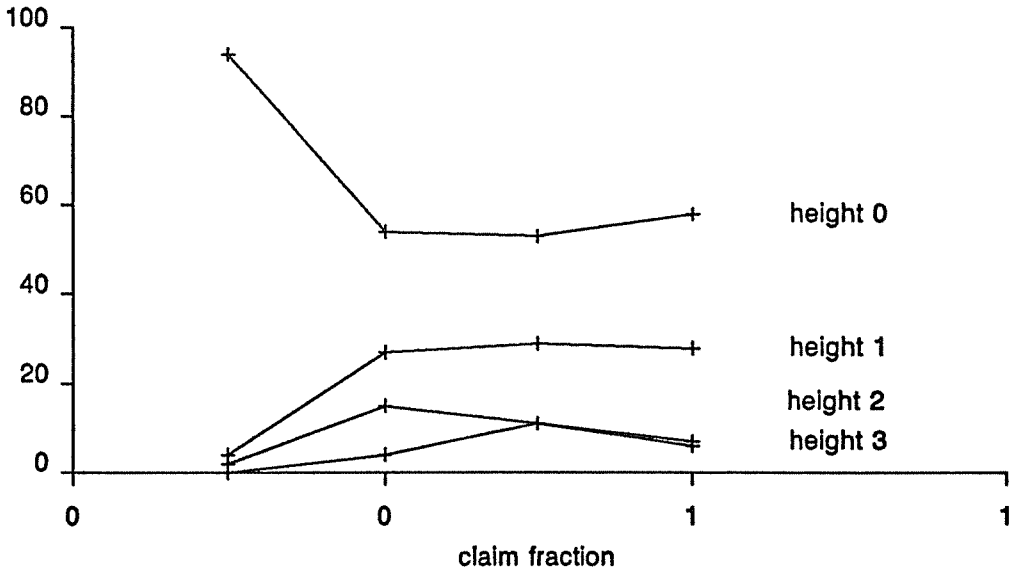
simulation was done on four Charlotte machines, the request time for the eight-leaves configuration is approximately twice what it should be if each leaf process were on a separate machine.

Understandably, both the number of messages and the request time goes up as the claim fraction increases. The hierarchical algorithm's performance is better at low claim fractions. This is because at low claim fractions most of granting is done locally. At high claim fractions the centralized algorithm seems to perform better. The results of the hierarchical algorithm at high claim

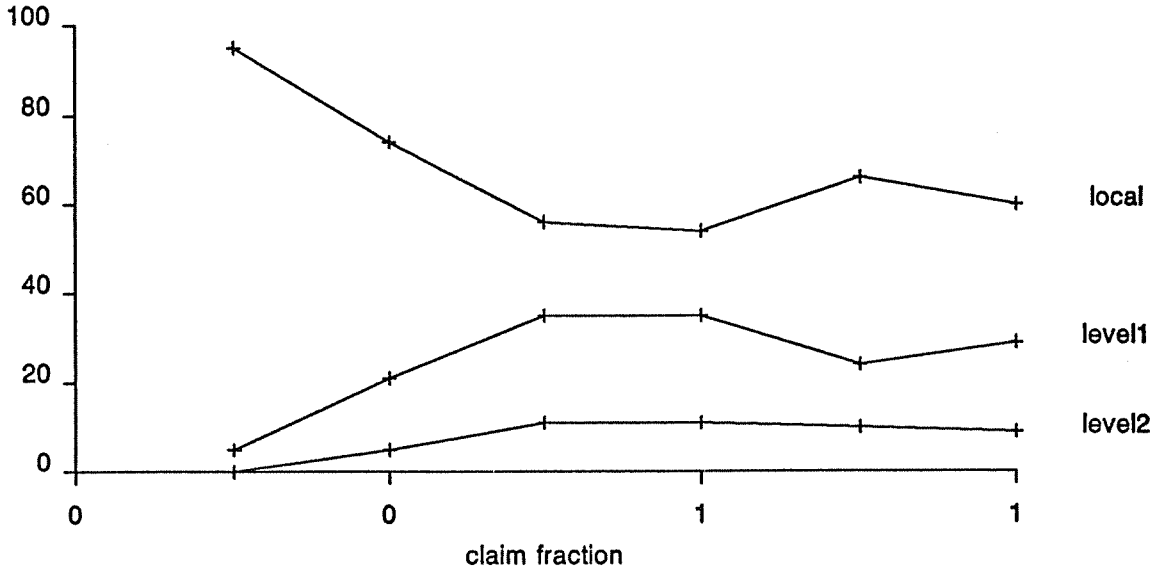
fractions might be misleading. We were unable to simulate a situation in which no process is making demands on the resource allocator. Hence, the allocator is usually busy and hence not able to help other nodes. This fact tends to increase the number of messages and the request time. At very high claim fractions (>2) the resources allocators on all nodes seer to be busy all the time, so no requests are granted. Graph 5 shows the percentage of grants at each level of the tree for the 8-leaf case, and Graph 6 shows the same figure for the 4-leaf case. At low claim fractions, most of the granting is done locally, so the extra cost of processors involved in the hierarchical method seems to be justified.

2.7. Experience with Lynx and Crystal

- The synchronization facilities using **connect** proved very useful.
- Threads of control proved very useful in simulating jobs. We were able to represent each job as a separate thread, so blocking threads proved very simple.
- The lack of dynamic allocation made programming a bit cumbersome. We had to implement tree structures using arrays.
- Shared memory among processes would have been very useful. In the centralized algorithm, every event calls an entry in the master processor. The master processor then modifies a binary tree. There would be no such need for remote calls if each process could access the tree structure directly.
- It was not possible to block a thread for any particular amount of time. During the simulation, we could not, for example, block all threads simultaneously. Hence the resource allocator was busy almost all the time,



Graph 5. Request time versus claim fraction, eight leaves



Graph 6. Request time versus claim fraction, four leaves

which biased the results.

2.8. Conclusions and future work

All the simulations were done using only one class of resources. The simulation could be extended to more than one class of resources. It was not possible to simulate the passage of time in this study. If some software could be developed to simulate a clock and if one could specify that an event take place at a particular time then the results of this study would be a little more meaningful. A third algorithm, called the **Distributed Banker's Algorithm** [Madduri85], could also be implemented to check the results and compare it with the other two algorithms.

3. Gaussian elimination

Experimenter: Kamal Gupta

3.1. Introduction

Numerical applications often require the ability to solve a system of simultaneous linear equations. Gaussian elimination is a common method used to compute the values of variables that satisfy a given set of linear equations. We discuss a distributed method to solve a set of simultaneous linear equations using a Gaussian elimination without pivoting. The input to the algorithm is the coefficients of the given equations (in the form of a matrix) and the right-hand side of each equation. The algorithm involves a sequence of operations on the rows of the coefficient matrix. We parallelize the work by distributing the rows of the coefficient matrix among p processes.

The remaining portion of this report is organized as follows. In Section 2 we discuss the problem of solving a set of linear equations. Section 3 describes the serial Gaussian elimination algorithm. In Section 4 we discuss a distributed algorithm. Section 5 describes our implementation in Lynx. In Section 6 we present performance results obtained by running the Lynx program on Charlotte. Sections 7 and 8 discuss our experiences with Lynx and point towards future directions.

3.2. Simultaneous linear equations

We are given a set of independent linear equations; the aim is to compute the values of the free variables that will satisfy all the equations simultaneously. A linear equation has the following structure:

$$a_{11}x_1 + a_{12}x_2 + a_{13}x_3 + a_{14}x_4 + \ldots + a_{1n}x_n = b_1$$

We have n such equations, and the complete problem is neatly summarized in the following form:

$$Ax=B$$

Here A ($n \times n$), the **coefficient matrix**, is made up of the left-hand sides of the linear equations. The right-hand sides of the n equations that we desire to solve are included in the column vector B ($n \times 1$), and x ($n \times 1$) is the variable column vector that we wish to compute. For the remaining portion of the discussion we shall append B to A and call the result the coefficient matrix. All the row operations performed on A will also be performed on B to maintain the consistency of each equation.

3.3. Serial algorithm

The serial algorithm for solving a set of independent linear equations triangularizes the coefficient matrix. For each row $0 \leq i \leq n$, operations are performed on all the following rows $j > i$ to place a 0 in the ith column of row j.

Then a backsubstitution, starting with row n and moving to row 1, places the results in the last column. More formally, the algorithm is as follows.

```
var
        equations : array (1..n, 1..n+1) of real;

function findlarge ( row : integer );
-- return the column of the element with the largest absolute value
-- in the given row

procedure exchange ( col1, col2 : integer);
-- interchange the two columns specified

procedure reduce1 ( row : integer );
--- divide the row by the diagonal element in order to make that element 1
begin
        for col := row to n+1 do
                a[row,col] /:= a[row,row];
end ;

procedure reduce2 ( row1, row2 : integer );
-- reduce row1 by row2
begin
        for col := row2 to n + 1 do
                a[row1,col] -:= a[row2,col] * a[row1,row2] / a[row2,row2];
end;

procedure backsub;
-- backsubstitute starting from the last row
begin
        for i :=n downto 2 do
                for j := i-1 downto 1 do
                        a[j,n+1] -:= a[j,i]*a[i,n+1];
end;

begin  --  main program
        for row := 1 to n-1 do
         begin
                big := findlarge(row);
                exchange(row,big);
                for j := row + 1 to n do reduce2(j,row);
                reduce1(row);
         end;
        backsub;
end.
```

3.4. Distributed algorithm

We distributed this algorithm along the lines discussed in [Gordon84]. The distributed algorithm is composed of a variable number of **Solver** processes (depending on the degree of parallelism desired) connected in a ring as shown in Figure 2. The **I/O** process initializes the data (equations), distributes them

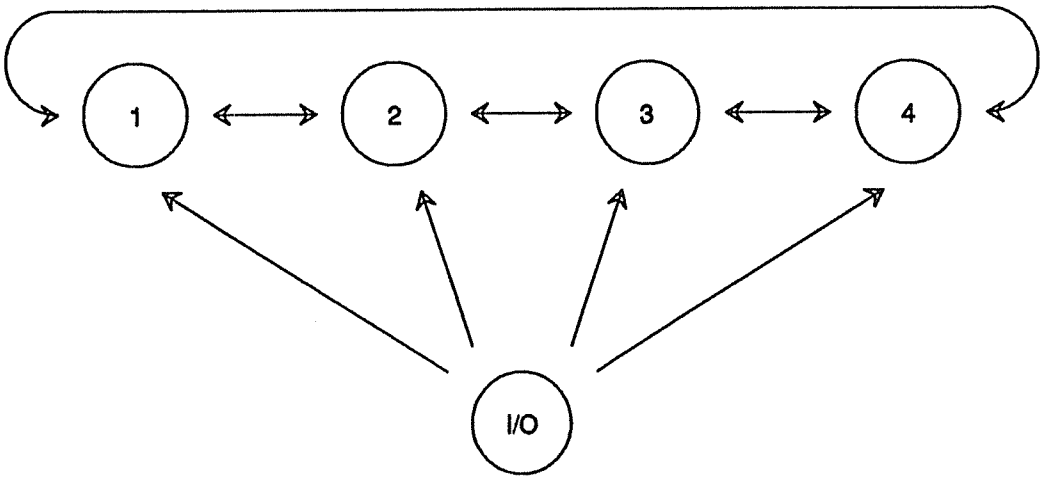


Figure 2. Process structure

among the **solver** processes, and collects and prints the results when execution is complete.

We use p solver processes for n equations where $p<<n$. Communication among the processes is performed by message passing. Each process handles a contiguous cluster of n/p equations and communicates results to other processes after reducing a group of j rows. The current implementation requires that n be a multiple of jp. The equations are distributed among the solver processes in the form of chunks of the coefficient matrix; process i gets equations $(i-1)n/p+1$ to in/p. At any given time the solver process i that is currently reducing its equations and communicating them to other processes is called the **head process**, and all the other processes are called **secondary processes.**

In order to avoid messages, we decided to avoid pivoting.

- The procedures findlarge and exchange are invoked only if the current head process has a zero in the diagonal element. The messages that would be sent to secondary processes to transmit the column pivoting information for each row are eliminated unless absolutely necessary. However, removing pivoting makes the algorithm subject to overflow problems, leading to the next modification.
- A normalization step was introduced to scale down the individual rows of the coefficient matrix when necessary.
- Instead of triangularizing the coefficient matrix, we decided to diagonalize it. A diagonal coefficient matrix permits each process to report its results independent of others. This saves a large number of messages by eliminating the backsubstitution phase altogether.

The distributed algorithm proceeds as follows:

(1) The I/O process initializes the data and distributes the rows among the solver processes: 1 gets the first n/p rows, and so on.

(2) After the rows are transmitted, 1 becomes the head process.

(3) The head process i performs pivoting on its columns (if necessary) and reduces a group of j rows. Subsequently, the reduced rows are passed together with any pivoting information to the secondary processes. This information is passed in both directions along the pipeline.

(4) Each secondary process receives the results from the head process and uses them to eliminate j columns from its cluster. This information is also passed in both directions along the pipeline.

(5) Process $i+1$ becomes the head process.

(6) After all equations are reduced, the results are reported to the I/O process and the program halts execution.

3.5. Implementation using Lynx

The Lynx program that implements the algorithm outlined above consists of two modules corresponding to the two kinds of processes. The I/O process initializes the data and distributes it among the solver processes. It also collects the results from them and prints them out when the program halts.

The solver implements the algorithm discussed in the previous section. It exports entries to implement the various aspects of the algorithm. One entry procedure is responsible for exchanging two columns of the local cluster. It is initiated by the head process that discovers a zero in its diagonal element. An entry is defined for the current head process to reduce a set of j rows from its set of equations. It performs the reduction by successive row subtractions to get ones in the diagonal elements of these j rows and zeroes in these columns in all the other rows. Initially, 1 is the head process, and subsequently each head process calls this entry in the process to its right. Reduced rows are transmitted (along both directions in the circular structure to save time) to the secondary processes. Each secondary process has an entry to accept these j rows from the head process and to eliminate j columns from its cluster by subtracting the equations received from its rows. The pseudo code for the solver process is given below.

```
process Gauss (id, left, right);
        -- each process has a unique id and links to two neighbors in the ring

var
        MyEquations : array [1..n/p][1..n+1] of real;
        LastCol, LastRow : integer; -- past history of  the process

entry pivot (sender, col1, col2);
-- if required then propagate pivot info to other nodes
begin
        if need to propagate then
                if sender=left then
                        connect pivot(id,col1,col2) on right;
                else
                        connect pivot(id,col1,col2) on left;
                end;
        end;
        -- perform pivoting on col1,col2
        foreach row in [1..n/p] do
                temp := MyEquations[row][col1];
                MyEquations[row][col1] := MyEquations[row][col2];
                MyEquations[row][col2] := temp;
        end;
end pivot;
```

```
entry ReduceSelf ;
-- reduce a group of j rows
-- currently this process is the head process
begin
        CurrentRow := LastRow+1;
        CurrentCol := LastCol+1;
        EndRow := LastRow+j; -- last row to be reduced in this step
        while CurrentRow < EndRow do
                if MyEquations[CurrentRow][CurrentCol]=0 then
                        -- pivoting of columns required
                        col2 := nonzero column in CurrentRow;
                        if nonzero col not found then
                                error("not independent");
                        end;
                        transmit pivoting info on both links;
                        call pivot(id, col1, col2);
                end;
                -- reduce a group of j rows
                foreach row in [1..n/p] do
                        if row<>CurrentRow then
                                -- triangularize
                                subtract CurrentRow from row to reduce it;
                        end;
                end;
                CurrentRow +:= 1;
                CurrentCol +:= 1;
                foreach row in [1..n/p] do
                        if row<>CurrentRow then
                                subtract CurrentRow from row to diagonalize;
                        end;
                end;
        end while;
        EquationsToSend := j eqns reduced in the while loop;
        transmit EquationsToSend along both directions in the ring;
        LastRow := CurrentRow;
        LastCol := CurrentCol;
        if not finished then
                connect ReduceSelf on right;
                -- right neighbor is the new head process
        else
                report results;
        end;
end ReduceSelf;
```

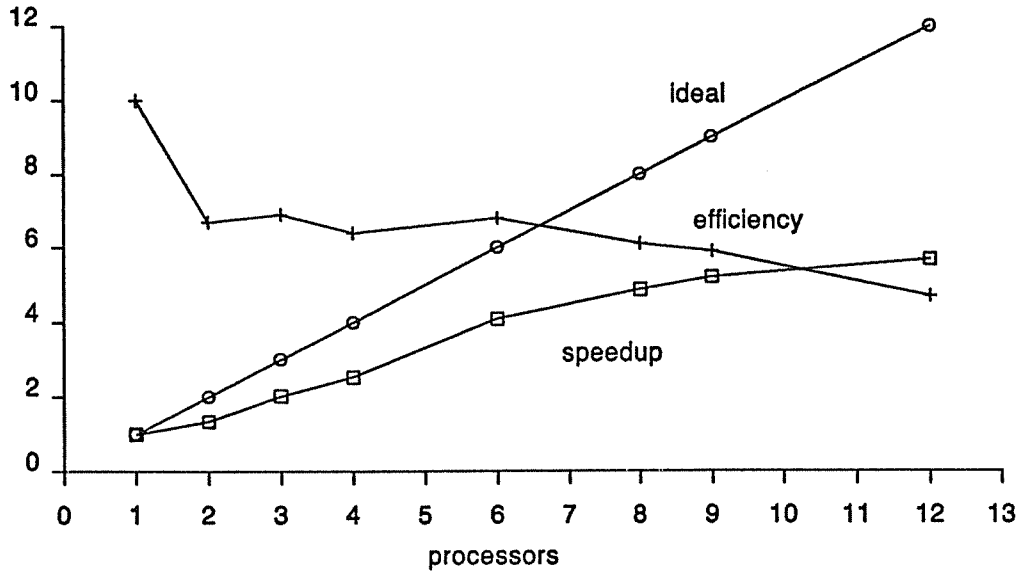
```
entry ReduceRows (EqnsReceived, FirstColumn, LastColumn);
-- use EqnsReceived to eliminate j columns from its cluster
-- this process is currently a secondary process
begin
        if need to propagate then
                transmit EqnsReceived to other neighbor;
        end;
        foreach i in [1..j] do
                foreach row in [1..n/p] do
                        factor := EqnsReceived[i][i+FirstColumn-1] /
                                MyEquations[row][p+FirstColumn-1];
                        foreach col in [1..n+1] do
                                MyEquations[row][col] -:= EqnsReceived[i][col]/factor;
                        end;
                end;
        end;
        LastCol := LastColumn;
        if finished then
                report results;
        end;
end ReduceRows;

end Gauss.
```

3.6. Performance results

The Lynx program described in the previous section was run on Charlotte. Although the program was run on various sets of data, we now discuss the results obtained by running the program on a system of 72 simultaneous linear equations. A limitation on run-time stack space prevented us from running the program on a greater number of equations. The coefficient matrix was generated by using a random number generator, and arbitrary values were assigned to each of the unknown variables (the values of the variables do not affect the execution time of the program). The serial program took 20.76 seconds of CPU time. The serial execution time was the time taken by one solver process to solve all the equations alone.

Speedup is the ratio of the execution time for the serial algorithm to the parallel execution time. Speedup ratio has been plotted in Graph 7 for number of processes varying from 1 to 12. (The number of processes must divide the number of equations to be solved.) Only one solver process is loaded on a Charlotte machine, and j is set to its maximum value, n/p. Efficiency is defined as the speedup divided by the number of processes. The efficiency curve is shown in Graph 7 normalized to 10.

A substantial speedup was obtained by running the program with many processes. For n equations, the algorithm involves $O(n^3)$ floating point operations, so execution time is reduced by dividing the work among several processes. We had anticipated a slowdown for 2 processes because of the overhead of message passing and the nature of the algorithm, but we observed a

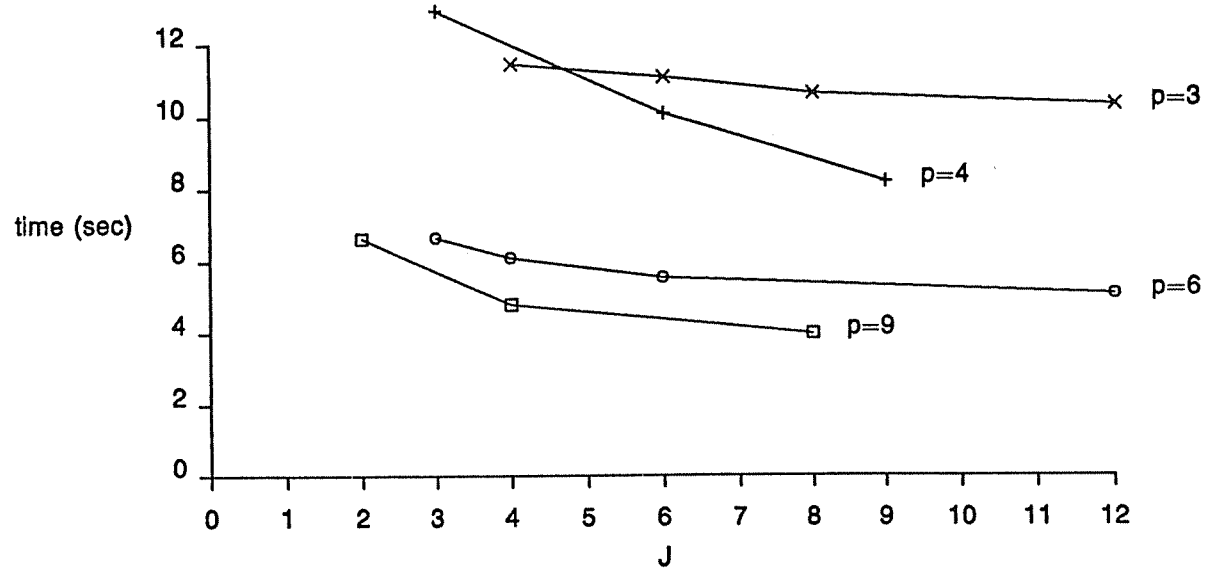


Graph 7. Efficiency and speedup for Gaussian elimination.

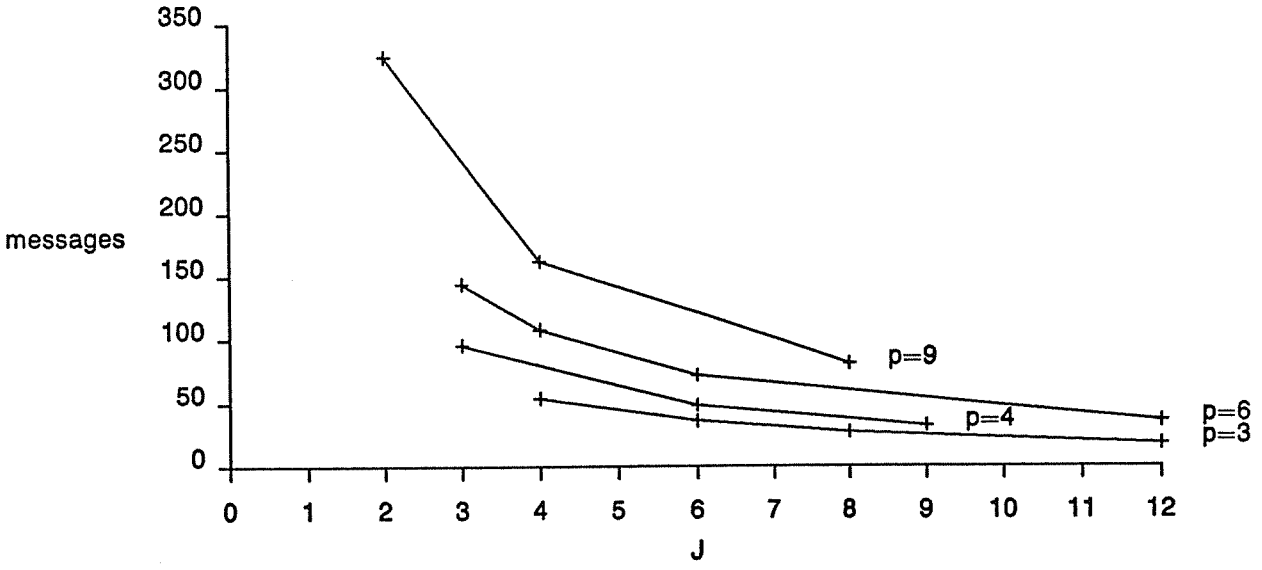
significant speedup even for that case. The speedup is roughly linear with a slope of about 0.6 for up to 9 processes but the seems to level off for a greater number of processes. The problem size (72 equations) is not big enough to justify such a large number of processes, and here the communication time represents a major portion of the total execution time. Therefore the speedup obtained by the division of work among several processes is offset by communication overheads.

Graph 8 shows the effect of varying j (the number of equations reduced in one attempt by a solver process) on the execution time for 3, 4, 6 and 9 processes. In general, increasing j reduces the execution time of the program, because it reduces the number of messages, although it also reduces concurrency. A limitation message size prevented us from increasing j beyond 12, but it can be inferred from graph 8 that indefinite speedup cannot be obtained by increasing it.

The number of remote procedure calls generated by varying j is plotted in Graph 9. The number of messages generated among the processes can be directly inferred from this graph because each RPC gives rise to two messages



Graph 8. Effect of varying j.



Graph 9. Number of remote procedure calls

(connect and reply).

3.7. Experiences with Lynx

Lynx was fairly convenient. Features such as processes, entries and links provide a natural framework for implementing distributed programs.

- The **connect** statement very useful, and the ability to pass arrays as parameters made implementation easy.

- The fact that there is no preemption of threads in Lynx helped us maintain data integrity. However, we can visualize situations where it would be desirable to permit the imposition of priorities on threads.
- Support for floating point operations would be appreciated.
- It would be nice to provide some sort of a **broadcast** primitive which would permit a process to send a message to all other processes. This would lead to a greater degree of parallelism in the implementation.
- The limitation on the run time stack space was quite irksome. It caused us to restrict the execution to a set of 72 or fewer equations. It would be desirable to have support for dynamic arrays in lynx.

3.8. Conclusions and future work

This work demonstrates the feasibility of distributing a computationally intensive algorithm among a number of processes. The performance results indicate that increasing the value of j would tend to give better execution times because it reduces the number of messages. However, an increased value of j would decrease the parallelism in the program because it would take more time for a process to reduce a set of j equations. Therefore the program would require tuning to achieve optimal performance. It would be interesting to compare the performance with a master / slave structure. It would also be illuminating to study the performance on a system with shared memory where the overhead of message passing would be eliminated.

4. Hough Transform

Experimenter: Viswanathan Mani

4.1. Introduction

Computer vision applications often require the ability to recognize the presence of certain standard features in images. We describe a distributed method using the Hough transform technique to detect the presence of straight lines in a given image. The input is assumed to be an array of intensity values in the range 0 .. 255. The Hough transform finds lines in an image by performing strictly local operations. This is in contrast to other methods of tracking down lines that involve very large graph search problems [Nevatia78]. This makes it suited for distribution among p processes.

The report is organized as follows. Section 2 describes the Hough transform in some detail; Section 3 presents a serial algorithm implementing the Hough transform; Section 4 gives a parallel algorithm; Section 5 describes an implementation in Lynx; Section 6 presents some performance results obtained by running the Lynx program on Crystal; Section 7 summarizes our experiences with Lynx; and Section 8 points toward some future directions.

4.2. The Hough transform

One approach to line (or curve) detection involves applying a coordinate transformation to the given picture (image) such that all points (pixels) belonging to a curve of a given type map to a single location in the transformed space [Rosenfeld81]. This approach maps global features (such as the set of points on a line) into local features that are then easily detected and extracted.

To illustrate, assume that the input is a binary image with pixels turned on wherever an edge is present and turned off at background locations. We first construct an accumulator array A with two dimensions, m and b. We will refer to this array as **Hough space.** We use m to represent slope and b for y intercept. To detect straight lines passing through a given edge point P at (x,y) we proceed as follows. This point is a candidate for all lines passing through that point with such combinations of m and b that

$$y = m\ x + b$$

is satisfied. Each edge point thus increments all such A[m,b] locations by one. We avoid an infinite set of m,b tuples by discretizing the (m,b) space. In addition to the actual lines passing through this edge point, many other lines will also be "voted for" by this point. If this procedure is repeated for all points that lie along a line in the image space, then the value of the array A at the location (m,b) corresponding to the slope and intercept of the actual line in image space will have been incremented by as many points there are on that line. In short, a local maximum at (m,b) in Hough space is detected. This detection can be done by purely local detection methods. Such a peak

represents a line in image space.

Apart from discretizing the values of m and b and deciding on a suitable step size, an additional problem arises in that m becomes infinite as the slope angle approaches 90°. To overcome this problem we use a (ρ,θ) parameterization [Ballard82], where

$$\rho = y\ \cos\theta - x\ \sin\theta$$

yielding $m=\tan\theta$ and $b=\rho\sec\theta$. To vote for all candidate bins in Hough space we calculate ρ for all values of θ ranging from 0 to 360 degrees in steps of 5 degrees. This is done for every pixel in the binary edge map that is turned on (those are the points that lie on an edge or line). This vote yields peaks in Hough space at all locations that correspond to the presence of lines in the image space.

4.3. Serial algorithm

4.3.1. Zero-crossing contours

The initial image is an $n\times n$ array of byte intensity values. This image must first be converted into a **binary edge map** with 1's along edges and 0's in all background locations. There are various techniques to accomplish this goal; we have chosen the **zero crossing technique** [Marr82]. A zero crossing is a point along an edge where the second derivative of the intensity gradient changes direction from positive on one side to negative on the other. To isolate zero crossings, we convolve a mask (the Laplacian Gaussian mask) with the original image to yield a convolved image.

A mask of size m is an $m\times m$ array of values such that the absolute sum of the values is 1.00 (the values are normalized). A convolution of an image I with a mask m (denoted as I^*m) means replacing every value in I by the result of the following operation:

$$\sum_{i=-m/2}^{i=m/2}\ \sum_{j=-m/2}^{j=m/2} I(x+i,\ y+j)\cdot m(i,\ j)$$

Each pixel in the image is replaced with its convolved value. This yields a zero-crossed image. The choice of an appropriate mask is a difficult one. We have chosen a mask of size 11 × 11 [Huertas85], a modified version of the original Marr-Hildreth Laplacian Gaussian operator. The mask is constructed using the following equation:

$$\nabla^2 G(x,y) = \frac{-1}{2\pi\sigma^4}\cdot\left\{2-\frac{x^2+y^2}{\sigma^2}\ e^{\frac{-x^2-y^2}{2\sigma^2}}\right\}$$

where σ is the constant of the Gaussian normal operator. It is of second degree, since to be able to detect edges in all directions the lowest non directional

derivative operator is of the second degree.

This convolved image $I^*\nabla^2G$ is then **thresholded** to convert all negative values to 0 and all non-negative values to 255.

The thresholded image is then scanned to detect zero crossings, which are those points that have at least one neighbor of a different value than itself. This is accomplished by another simple convolution operation. The thresholded image is convolved with the following mask:

$$\begin{matrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{matrix}$$

A zero result indicates that the present point is a non zero crossing point while a non zero result indicates that it is a zero-crossing point. All zero crossing points are now replaced with 1's and all others with 0 so that the conversion of the image to the binary edge map is now complete.

4.3.2. Voting for Hough space

The binary edge map is then scanned, and all edge points vote for their appropriate (ρ,θ) bins in Hough space.

The Hough array is then smoothed with its neighbors to allow for some roundoff caused by the discretization of the θ space and approximations during conversion from real numbers to their nearest integers.

The smoothed Hough space is then searched for local maxima or peaks. A **local maximum** is a value greater than all eight neighbors. A bin in Hough space does not even qualify to be a local maximum unless it is greater than some threshold (which represents the minimum length of the line that we would like to be able to find). All non-maximum points are reduced to zero.

Since the Hough space interprets dotted lines as continuous, each line found is verified by tracking the image along the slope of the line found as a peak in Hough space. Although this step is expensive, it is still superior to exhaustive search methods. The computationally intensive step of applying the Laplacian Gaussian mask is needed for all methods (other alternatives of applying directional derivatives are just as computationally intensive).

The essential steps involved are:

1. convert image to edge map.
    1.1 apply the Laplacian Gaussian mask to the image
    1.2 threshold the resulting image
    1.3 detect the zero crossings
2. Let binary edge map vote for Hough bins
3. Smooth the Hough space
4. Detect local maxima or peaks in Hough space
5. Verify the presence of lines and report individual segments.

4.3.3. Serial algorithm pseudo-code

```
process serial;

const
        size = 128;             -- image size
        R    = 1.4142 * size;     -- root two times the size is R
        m    = 5;               -- half the Laplacian Gaussian mask width
        theta_max = 72;           -- 360 degrees in steps of 5
var
        image, zero  : array [1 .. size] [1 .. size] of integer;
        accum,hough  : array [1 .. R] [1 .. theta_max] of integer;
        delG         : array [-m .. m] [-m .. m] of integer;
        -- the image and a backup ; Hough space and a backup and the
        -- Laplacian Gaussian mask

begin
        read in image;
        read in mask into delG;
        read in sin and cos tables;
        CONVOLVE image using delG;
        THRESHOLD image;
        DETECT zero crossings;
        VOTE for Hough;
        SMOOTH Hough;
        SUPPRESS Non Maxima in Hough space;
        VERIFY and Report;
end serial.
```

Some of the important procedures are described below ; others are just briefly mentioned.

```
procedure CONVOLVE;
var sum   : integer;
begin CONVOLVE
        for i from 1 to size do
                for j from 1 to size do
                        sum:= 0;
                        for i1 from −m to m do
                                for j1 from −m to m do
                                        sum +:= image[i+i1][j+j1] * delG[i1][j1];
                                end;
                        end;
                        zero[i][j] := sum / sum of absolute delG mask values;
                        −− zero is the temporary location to store the convolved image
                end;
        end;
end CONVOLVE;

procedure THRESHOLD;
begin THRESHOLD
        for i from 1 to size do
                for j from 1 to size do
                        if zero[i][j] <=0
                                then zero[i][j] := -100;    −− a negative class value
                                else zero[i][j] :=  100;    −− positive value
                        end;
                end;
        end;
end THRESHOLD;

procedure DETECT zero crossings;
var
        mask   : array [−1 .. 1] of array [−1 .. 1] of integer
                :=      { { −1  −1  −1  } { −1   8  −1  } { −1  −1  −1  } }
begin DETECT
        −− convolve zero with mask;
        for i from 1 to size do
                for j from 1 to size do
                        sum := 0;
                        for i1 from −1 to 1 do
                                for j1 from −1 to 1 do
                                        sum +:= zero[i+i1][j+j1] * mask[i1][j1];
                                end;
                        end;
                        if sum = 0 then
                                image[i][j] := 0;   −− not zero crossing point
                        else
                                image[i][j] := 255; −− zero crossing point
                        end;
                end;
        end;
end DETECT;
```

```
procedure VOTE for Hough;
begin VOTE
        for i from 1 to size do
                for j from 1 to size do
                        if image[i][j] > 0 then
                                for theta from 1 to theta_max
                                        -- from 0 to 360 in steps of 5
                                        rho := j * cos[theta] - i * sin[theta];
                                        accum[rho][theta] +:= 1;
                                end;
                        end;
                end;
        end;
end VOTE;

procedure SMOOTH Hough;
begin SMOOTH
        replace each value in accum by the average of itself and its eight
        neighbors by a simple sum and average operation and store the result
        in the backup Hough array;
end SMOOTH;

procedure SUPPRESS non maxima;
begin SUPPRESS
        for rho from 1 to R do
                for theta from 1 to theta_max do
                        if hough[rho][theta] > threshold and
                                        it is greater than all its neighbors
                        then
                                accum[rho][theta] := hough[rho][theta];
                        else
                                accum[rho][theta] := 0;
                        end;
                end;
        end;
end SUPPRESS;

procedure VERIFY;
begin VERIFY
        -- pick out all non zero values in the accum array
end VERIFY;
```

4.4. Distributed algorithm

As can be seen from the serial algorithm, there are two major steps involved in this problem. One is the conversion of the image from its intensity values into a binary edge map. The other is voting, smoothing, and detecting local maxima in Hough space. The first step can be easily distributed by splitting the image into p pieces (where p is the number of processes) and have each process convert its portion of the image into the binary edge map. The splitting is done column-wise, that is, each process is given n/p columns and n rows of the image.

Step 2 can be distributed in two ways. The whole image space must vote for the whole Hough space.

(1) Have p portions of the image vote over the whole Hough space (each process having an independent copy of the whole Hough space). This would lead to a small portion of the image mapping itself on to the whole Hough space. After all the processes are done, merge these individual Hough spaces to yield the overall Hough space, which must still be smoothed and checked for peaks (or local maxima).

(2) Each process has a copy of the whole binary edge map, which then votes for a portion of the Hough space. For example with 2 processes, process 1 would vote for the Hough space in the region Hough[*][1..36], and process 2 would vote in the region Hough[*][37..72]. Each process can independently smooth and select local maxima in its portion of the Hough space.

Both methods have advantages. Method (1) has the advantage that each process already has a pth portion of the image from building the binary edge map. This same piece can be used to vote subsequently. The disadvantage is that the Hough spaces would have to be shared and merged. After merging, the Hough space would have to be distributed again for the computationally intensive steps of smoothing and detecting local maxima. For verification, each process needs the entire binary edge map. In total, the Hough spaces are broadcast twice and the binary edge maps once.

Method (2) has the advantage of never communicating the Hough spaces but the disadvantage of broadcasting binary edge maps. This malady is present in method (1) as well. Although we did not implement verification, it is far more efficient in method (2). The density of lines in portions of the image is certain to vary. The voting step's cost depends on the number of edge points present in that portion of the image. Method (1) would not balance work well among the processes.

For these reasons, we chose method (2). A master process is in charge of I/O and reporting results. It distributes initial information to each of the p slave processes, each of which works on its portion of the problem. After an intermediate conversion of the image into a binary edge map form, the slaves communicate with each other. At the very end, the slaves communicate results to the master. The pseudo-code is shown below.

```
process master;

const
        NO_LINES = 50;
type
        linetype  =  array [ 1 .. NO_LINES ] of integer;
var
        called   : integer;
        count    : integer;
        reported : Boolean;
        lines    : linetype;

begin master
        read in image;
        read in delG;
        read in sin and cos tables;
        for i from 1 to p do
                send delG, sin, cos to each slave[i]
        endfor;
        for i from 1 to p do
                SEND_IMAGE to each slave[i]
        endfor;
        called := 0;
        await reported;
        calculate times and output results;
end master;

entry report (inlines : linetype);
begin report
        i := 1;
        called +:= 1;
        loop
                if count > NO_LINES then
                        exit;  -- too many lines?
                end;
                if inlines[i].rho = -1 then
                        exit;  -- no more from slave?
                end;
                -- copy information
                lines[count].rho := inlines[i].rho;
                lines[count].theta := inlines[i].theta;
                lines[count].score:= inlines[i].score;
                i +:= 1;
                count +:= 1;
         end;
         reported := called = p; -- have all reported?
 end report;
```

```
process slave(id:integer; io_link, in_link, out_link :link);
-- io_link from master
-- in_link from slave id mod p - 1
-- out_link to id mod p + 1
const
        size    =  128; -- 256 for the larger image
        m       =  5;
        R       =  1.4142 * size;
type
        imagetype = array [ 1.. size ][ 1.. size ] of integer;
        houghtype = array [ 1.. R ][ 1 .. 72 ] of integer;
        masktype  = array [ -m .. m ][ -m .. m ] of integer;
var
        image, zero  : imagetype;
        accum, hough : houghtype;
        delG         : masktype;
        image_copied : boolean;
        info_copied  : boolean;

begin -- slave
        await info_copied;
        await image_copied;
        Calculate appropriate columns of image using id number;
        CONVOLVE with delG between appropriate columns of the image;
        THRESHOLD appropriate portion of the image;
        DETECT crossings on appropriate portion of image;
        while not ( id - 1) pieces of image received do
                receive a piece of the image on in_link from previous slave;
                if (id mod p + 1 <> owner) then
                        send piece of image to next slave on out_link;
                end;
                copy piece of image into global image array;
        end;
        send my piece of image to next slave on out_link;
        while not ( p - 1) pieces of image received do
                receive a piece of the image on in_link from previous slave;
                if (id mod p + 1 <> owner) then
                        send piece of image to next slave on out_link;
                end;
                copy piece of image into global image array;
        end;
        VOTE for Hough between theta limits as calculated by id number;
        SMOOTH Hough space between theta limits;
        SUPPRESS non maxima of Hough space between theta limits;
        VERIFY using image;
        send results to master on io_link;
end slave;

entry copy_info;
begin copy_info
        copy the information into global sin, cos and delG arrays;
        info_copied := true;
end copy_info;
```

```
entry copy_image;
begin copy_image
        copy the input image into global arrays;
        image_copied := true;
end copy_copied;
```

4.5. Implementation in Lynx on Crystal

The Lynx implementation of the algorithm has the same structure as the distributed version explained earlier. There are two modules: a master and a slave.

The master module has three functions:

- Read in the relevant information: the image, the mask and the sin and cos tables from files.
- Send this information to all the slave processes.
- Wait for the results from the slaves and print them along with statistics.

The master module follows the algorithm described in the previous section. Charlotte has a stack space restriction on each process, which affects the space allotted to parameters in entries. This restriction meant that SEND_IMAGE could not send the whole array of the image as a parameter and had to do it in small pieces. This is done by defining a type called a **chunk**, which is a smaller array of size 32×128 or 16×256 (depending on whether the 128×128 image or the 256×256 image was being used). The master sends more columns than allotted to each process to allow the neighbor operations to work correctly.

The slave also follows the algorithm outlined above. Due to the stack space problems, transmitting the image pieces around at the intermediate stage became a problem. Each process tries to connect to an entry in another process. We replaced implicit receipt (entries) with explicit (using **accept**) to circumvent the space problem. This change raised synchronization issues which were absent in the implicit paradigm. No longer can each process attempt to connect to the other hoping for one of the threads to start running. Rather each process i waits for all processes with smaller id numbers to send their chunks. Each process on receiving a chunk checks to see if the next process to which it is connected was the owner of the chunk or not. If the neighbor is not the owner then this chunk is forwarded to the neighbor. The chunk is then copied into the process' global structures. Once all neighbors to the left (those with smaller id numbers) have sent their packets, then this process sends its packets to the neighboring process on its right.

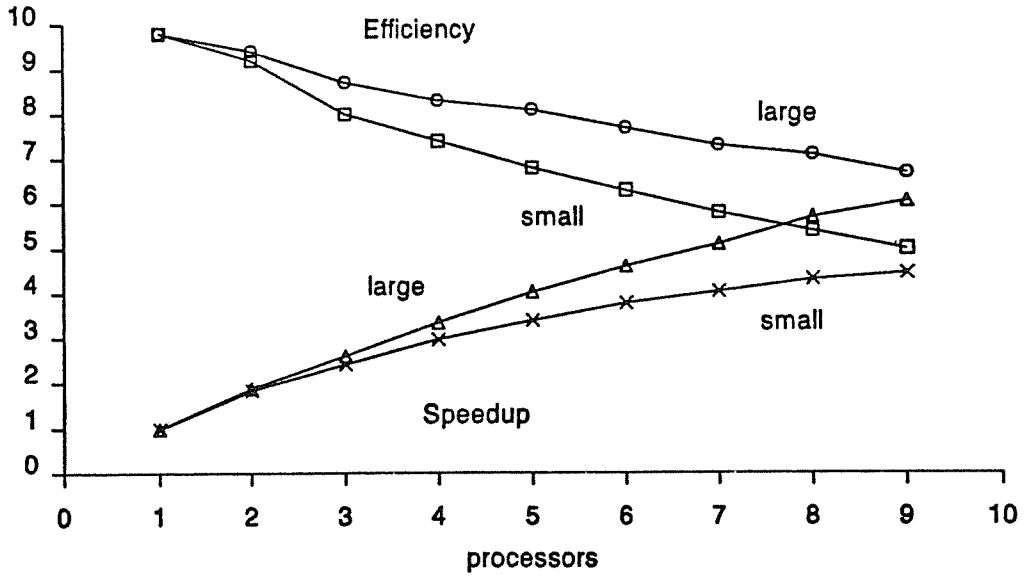
After much experimenting and tuning parameters, test runs were conducted with two images of size 128×128 and 256×256. The 128×128 image used chunk sizes of 32×128 (of one-byte integer values), while the larger image had to use chunk sizes of 16×256 (attempts at 32×256 ran out of stack space). This increased the number of messages and slightly impaired the speedup results that were obtained.

4.6. Performance results

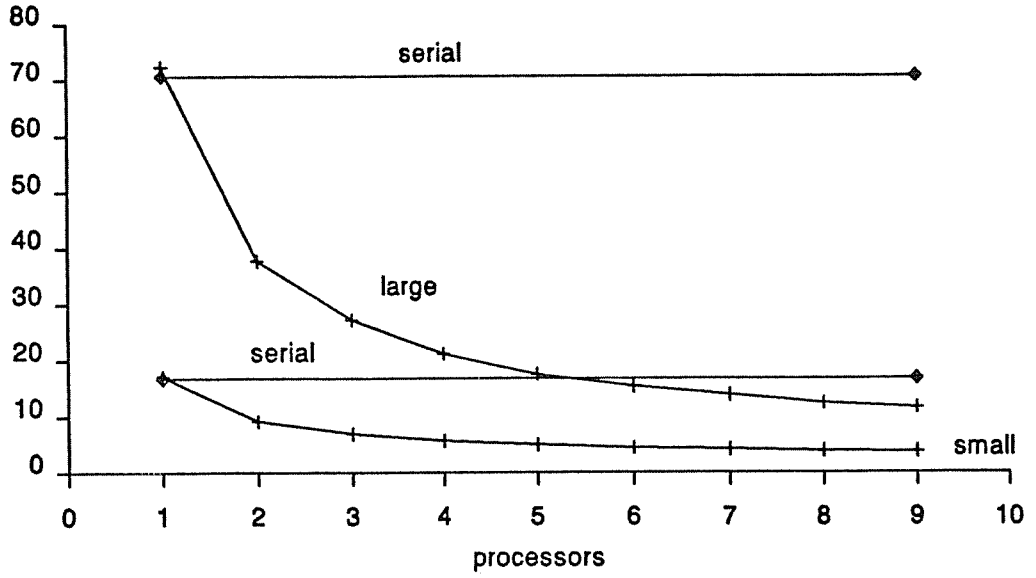
The Lynx program described in section 5 was run on Charlotte. Experiments were conducted up to 9 machines. A master and p slaves were loaded on p machines with the master and one slave process sharing a machine. The master is idle most of the time when the slave processes are running, except for a short while when the master is transmitting information to the slaves. Two runs were made using images of size 128 × 128 (henceforth referred to as the small image) and one of size 256 × 256 (the large image). The serial program took about 16.8 seconds on the small and 70 seconds on the large image. All measurements started from the moment the master had finished sending the image to the slave processes.

The speedup and efficiency for the distributed algorithm are illustrated in Graph 10. Speedup is the ratio of serial execution time to the parallel execution time. Efficiency is the ratio of speedup to the number of processes. The efficiency has been normalized so that 10 is a perfect value. Speedup increases with the number of processes, but not linearly, since message costs make the efficiency decrease somewhat. With larger stack spaces, one could use **connect** statements on entries alone and do away with the **accept**, which would increase the speedup values slightly. Nonetheless, very good efficiency figures were observed, especially for the large image.

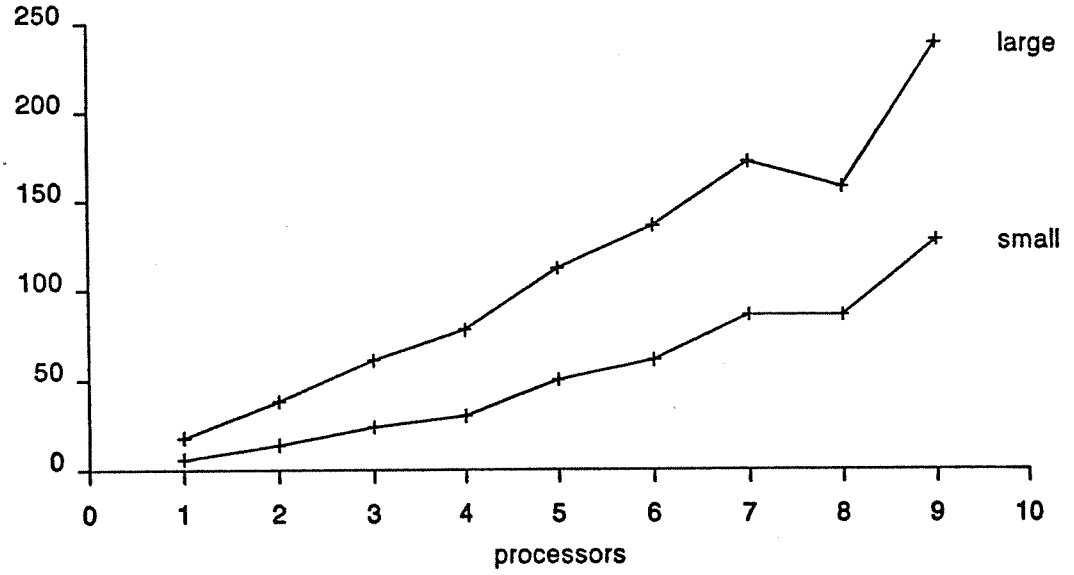
The total time taken to execute with various number of processes is plotted in Graph 11. The number of messages (measured as the number of remote procedure calls) is shown in Graph 12. The number of messages actually went down in one run from 7 to 8 processes. This anomaly is due to the fact that the



Graph 10. Efficiency and speedup



Graph 11. Time required



Graph 12. Number of messages

chunk sizes are 32 × 128 and 16 × 256, respectively, in the two images. This causes the chunks to align exactly on boundaries, which is the most efficient as far as sending chunks are concerned.

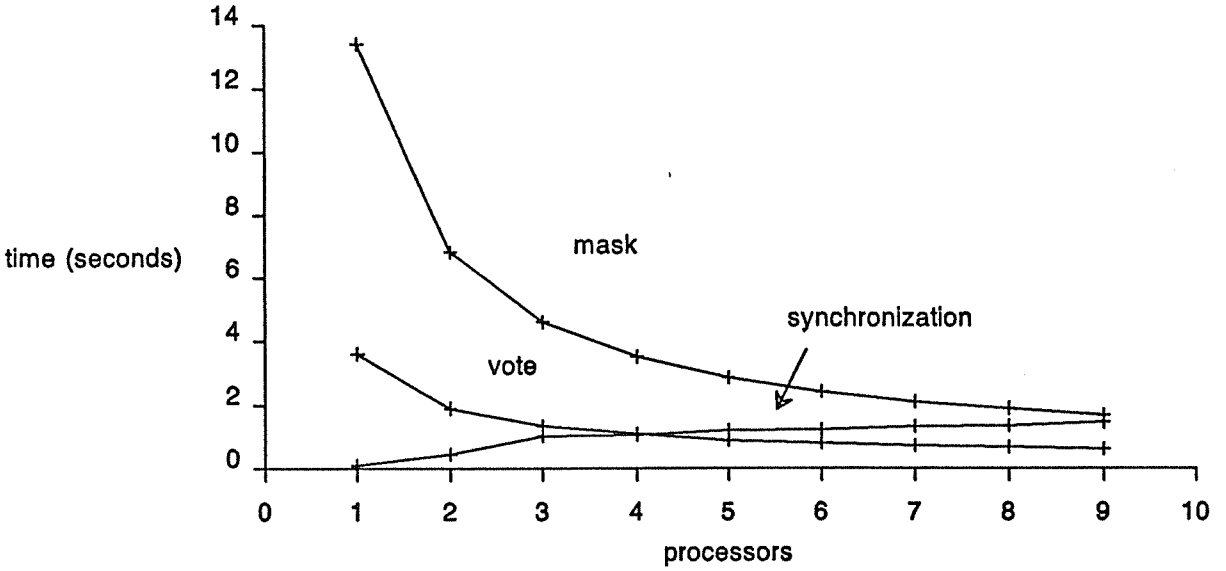
The time taken to

- convert the image to an edge map (**mask time**)
- vote, smooth and detect local maxima in Hough space (**vote time**)

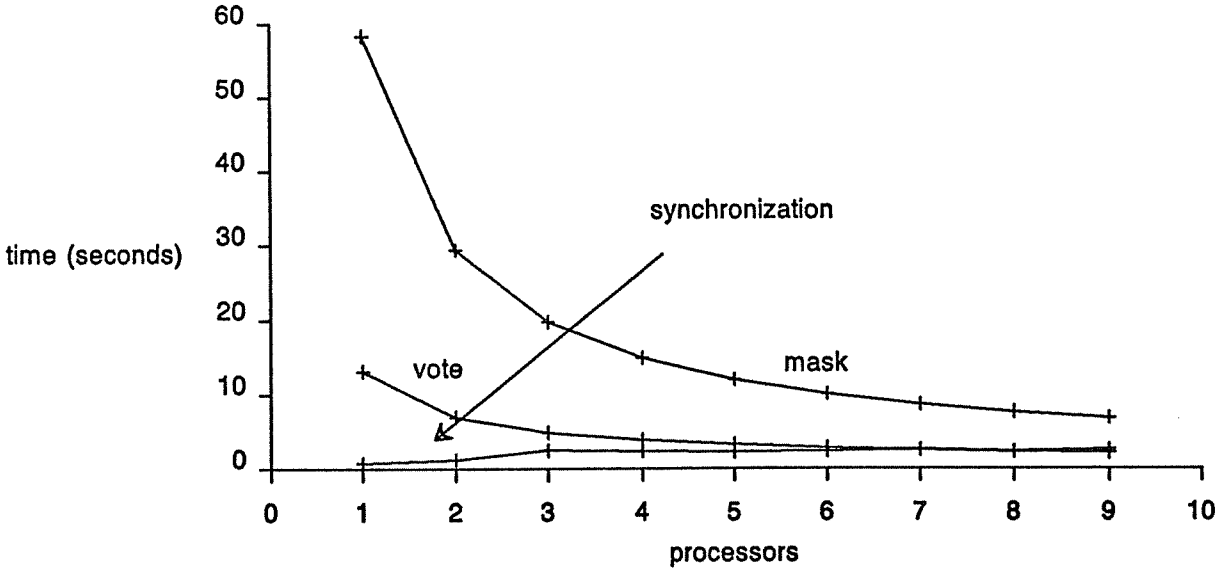
- receive, send and wait for messages (**synchronization time**)

are shown in Graph 13 (for the small image) and in Graph 14 (for the large image). The same information is shown in Graphs 15 and 16 as a percentage of total time.

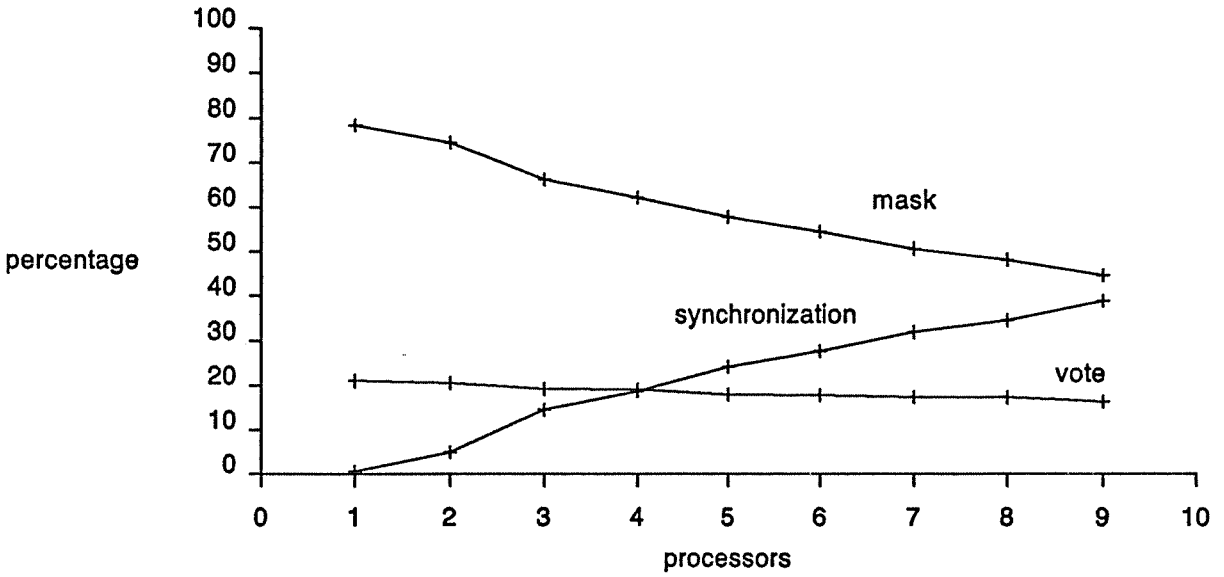
Mask time dominates the cost. As expected, it decreases with the number of processes, both in an absolute sense and when taken as a percentage of the total time. Vote time behaves similarly. As expected, idle time as a percentage



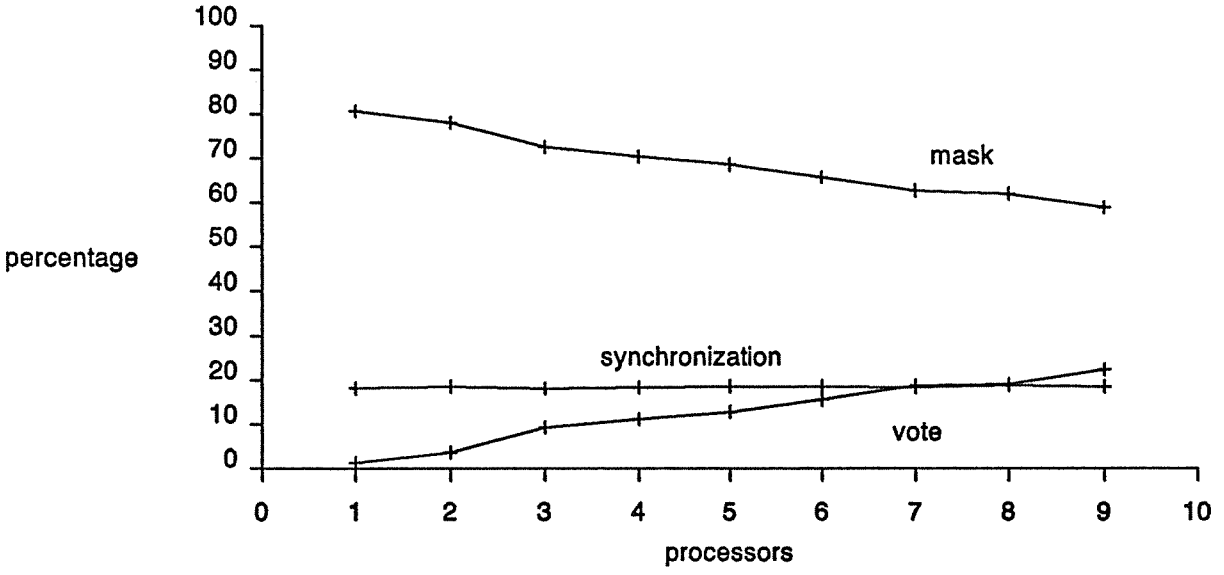
Graph 13. Breakdown of time for small image



Graph 14. Breakdown of time for large image



Graph 15. Percentage breakdown of time for small image



Graph 16. Percentage breakdown of time for large image

of total time increases with the number of processes, and any decrease in absolute values can be attributed to the boundary alignment described earlier.

4.7. Experience with Lynx and Crystal

- The synchronization facilities using **connect** and **accept** were very useful.
- The lack of floating point support was a bit of a hindrance. We got around it by using sine and cosine tables and scaling their values.

- The I/O facility was inadequate. It would be nice if read and write could support file reads and writes by automatically loading the necessary processes and setting up the links.
- A bigger run time stack space would be of great help. Real images tend to be 512 × 512, which would necessitate the usage of large packet sizes to avoid getting bogged down in message passing.

4.8. Conclusions and future work

Image processing (especially low level) is computationally very intensive. Most low-level operations involve the usage of purely local operations which imply that each portion of the image could be worked on independently. This makes distribution of work among many processes ideal.

Future modifications could be made to the program to make it run for a more general Hough transform technique to detect features other than straight lines (circles, patterns etc. are good examples).

The verification step involving tracking along the hypothetical line in the image to find actual line segments could also be implemented. This step would again improve speedup tremendously since this is a computationally very intensive operation.

Lastly, with the aid of better I/O, it would be worthwhile to try distributing the initial input of the image. Since each process works with a fixed portion of the image at first, it would be possible to calculate the number of bytes offset at which the process should start reading the image. Reading the image is rather time consuming, and if parallel reading is possible, it could reduce a great bottleneck (and make the master process superfluous).

5. Stable Marriage

Experimenters: Gautam Das and Ganesh Jayaraman

5.1. Introduction

Suppose we have n men and n women who wish to be engaged. Each person prefers the members of the other sex in some order. This order is known as the **preference list** of the person. We are required to pair them such that the engagements are stable, that is, no man and woman would rather be engaged to each other than to their current partner. Such a set of engagements is known as a **stable matching.** This is known as the **stable marriage** problem [Sedgewick83], and is a special case of a more general problem from graph theory known as **bipartite matching.**

It can be shown that for every instance of the stable marriage problem, there exists at least one stable matching. In the rest of the report we shall discuss a series of serial and distributed algorithms that find a stable matching. All algorithms were implemented in Lynx under Charlotte.

5.2. Serial algorithm

We describe a version of the serial algorithm that is **man oriented** in its approach. Each man, in order, proposes to the first woman on his list. In case of conflicts, a woman chooses between her suitors based on her list, and the rejected man turns to the next woman on his list.

A set called **Suitors**, initialized to contain all men, is maintained. Each man has a pointer containing the number of women he has already proposed to, starting from the head of his preference list. These pointers are stored in an array called the **Frontier.** Each woman has a variable containing the identity of her current fiancé.

At every iteration, a man from Suitors is selected. He proposes to the next woman on his list. If the woman is not engaged, she agrees and he is removed from Suitors and entered as her current fiancé. If she is engaged, she consults her preference list and decides between him and her current fiancé. The rejected man is placed back in Suitors (at the head of the list).

The worst case time of the algorithm is $O(n^2)$, which occurs when all men have to propose to most of the women in their lists before they get permanently engaged. Upon termination, Frontier contains the number of proposals each man had to make before he became permanently engaged. The sum of all elements of Frontier is the total number of proposals made, and is directly proportional to the time of the serial run. We shall refer to this sum as the **work done** by the algorithm.

This is the serial algorithm:

```
const
        MaxSize = ;     -- maximum number of men (women)
type
        PrefArray = array [1..MaxSize][1..MaxSize] of integer;
        MatchArray = array [1..MaxSize] of integer;
        PointerArray = array [1..MaxSize] of integer;
var
        MaleArray, FemaleArray: PrefArray;
        Matching: MatchArray;
                -- Matching[i] = j implies woman i gets engaged to man j
        Frontier: PointerArray;
        SetOfSuitors: set of [1..MaxSize];
        CurrentSuitor, Fiancé, WomanProposed: integer;

procedure Initialize;
begin
        generate MaleArray;
        generate FemaleArray;
        initialize SetOfSuitors to contain all men;
        initialize Frontier to the start of each man's list;
end Initialize;

procedure Propose;
-- a suitor is chosen, he proposes to the next woman in his list, and
-- the identity of her fiancé is established
begin
        CurrentSuitor := an element from SetOfSuitors;
        Frontier[CurrentSuitor] +:= 1;
        WomanProposed := MaleArray[CurrentSuitor][Frontier[CurrentSuitor]];
        Fiancé := Matching[WomanProposed];
end Propose;

function Accept: Boolean;
-- the proposed woman decides between the suitor and her fiancé
begin
        if no Fiancé yet then
                Matching[WomanProposed] := CurrentSuitor;
                return true;
        elsif
                FemaleArray[WomanProposed][CurrentSuitor] >
                FemaleArray[WomanProposed][Fiancé] then
                        return false;
        else
                Matching[WomanProposed] := CurrentSuitor;
                SetOfSuitors +:= { Fiancé }
                return true;
        end;
end Accept;
```

```
procedure SequentialRun;
begin
        while SetOfSuitors <> {} do
                Propose;
                if Accept then
                        SetOfSuitors -:= { CurrentSuitor };
                end;
        end;
end SequentialRun;

begin
        Initialize;
        SequentialRun;
        output matching;
end Serial.
```

5.3. Distributed algorithms

In this section we shall discuss several competing distributed implementations of the man-oriented stable marriage algorithm. Although we were not able to come up with any that performs better than $O(n^2)$ in the worst case (to our knowledge, no one else has yet done so), most of them perform satisfactorily for large problem sizes with random data.

The serial algorithm design, including data structures and their access mechanisms, have been retained in all the distributed versions. Thus speedup and other performance results are solely due to the distribution of the serial algorithm, and not because of any radically different approach.

A few crucial facts result from this observation. Whatever may be the distribution scheme, the final matching will be the same as that obtained by the man oriented serial algorithm. Furthermore, the final Frontier, and consequently the work done, will be the same in all implementations. Speedups are achieved because the Frontier is advanced in parallel.

A process may be given charge of a set of men, its job being to ensure that they eventually get engaged. Due to this, the problem size and the number of processes used are not dependent upon each other.

5.3.1. Bipartite algorithm

The men are divided into almost equal, disjoint sets and entrusted to **male** processes. Likewise, **female** processes are in charge of the women. A **starter** process distributes the original data structures, such as the preference lists. A **terminator** process detects termination and constructs the solution from the partial solutions obtained by the female processes. This configuration is shown in Figure 3.

Each male process operates like the sequential algorithm, trying to engage the men in its custody, except that proposals are messages to appropriate female processes. A female process acts as a server to proposal requests. It

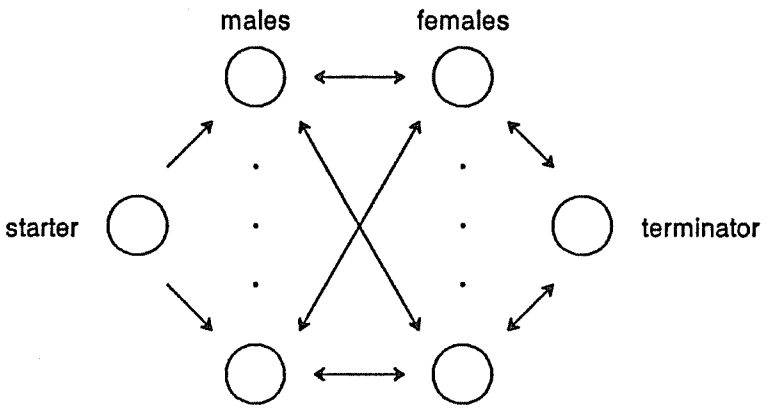


Figure 3. Bipartite configuration

either agrees or disagrees via a reply message. An agreement could be preceded by the female process informing her previous partner of a broken engagement. When a woman agrees to her first proposal, her process sends a message to the terminator, which increments a counter. Termination is detected when this counter reaches n, the number of men (or women) in the problem. When this occurs, the terminator polls each female process for local engagement data, from which it constructs the final results.

Speedup is achieved when proposals are considered in parallel by different female processes. The major problem with this implementation is that each proposal requires a message to a female process, that is $O(n^2)$ messages in the worst case. On the other hand, proposals are implemented as array accesses in the sequential algorithm. If the time taken for message transfer is the same as that for array access, we would achieve speedups even for a two male, two female process configuration. Unfortunately, that is not the case in Lynx, so an order of magnitude more processes are required before speedup can be achieved. This requirement is impractical, so we rejected this approach.

5.3.2. Star algorithm

Our next three algorithms are designed to handle most queries into the women's preference lists by local array accesses rather than through messages. A **master** process initially distributes data and outputs final results. The men are distributed equally among **slave** processes, which also have complete copies of the women's rank arrays. This latter requirement is essential in order to permit local array access by the slaves.

Initially all slave processes operate very much like the serial algorithm. All of them have a copy of Frontier, which they advance during the serial run. They engage their men, resolving conflicts among them but ignoring men not in their custody. They thereby advance their own portion of Frontier.

When all the slaves have finished engaging their men, they reach the **resolution stage**, in which they get together to resolve global conflicts. These occur when men from different slave processes are engaged to the same woman. These conflicts are resolved by accessing the rank array of the woman, retaining her engagement with the man she prefers most, and rejecting the others. This resolution advances the Frontier.

After the resolution stage, the slaves work independently again, attempting to engage their newly rejected men. These men start proposing to the women from where they left off in their preference lists.

The three algorithms differ in their implementations of the resolution stage. In the **Star** algorithm, a centralized design is adopted, as shown in Figure 4. Global data structures containing the latest engagements and the latest Frontier reside in the master. Each slave periodically sends its Frontier to the master, which compares it with its own Frontier and updates the latter, resolves global conflicts and updates global engagements, and reports back any broken engagements plus the latest Frontier to the slave. While this is being done, if the number of engagements reaches n, it announces termination.

The slaves need not all be in the resolution stage at the same time. In fact, two fast slaves can alternately update the master many times before a third, slower slave performs its first update. This asynchrony has the advantage of reducing idle time. Another advantage of this centralized model is that it expends no effort to maintain duplicate copies of data structures. The disadvantage is that the central process is a bottleneck, because it serves slave requests sequentially. If the slaves happen to synchronize their requests, the resolution stage may be time consuming. The next two algorithms have distributed implementations of the resolution stage.

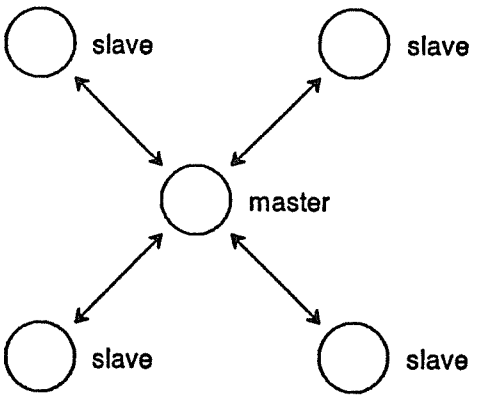


Figure 4. Star configuration

5.3.3. Unidirectional-cycle algorithm

The **Unidirectional-cycle** configuration consists of several slaves connected in a cycle. A master exists to distribute data initially and report final results. No global data are kept; each slave keeps its own latest version of global engagements and the Frontier. This configuration is displayed in Figure 5.

On finishing engaging its men, a slave forwards its Frontier to its clockwise neighbor. It then waits for the Frontier from its counterclockwise neighbor. It then updates its own Frontier and resolves conflicts with its own copy of global engagements. If the engagements are complete, the slave signals termination, otherwise serial iterations for engaging its newly rejected men are resumed.

This is a distributed implementation of the resolution stage. The advantage is that this stage is very fast with each slave requiring to connect only once. There are several disadvantages though. Unlike the Star approach, a slave does not necessarily have the latest engagements or the latest Frontier at any time because it gets external data from only one process, its counterclockwise neighbor. In other words, it requires many iterations before the engagements generated by a distant process affect a slave's global engagements copy. Another extreme in a distributed implementation would be to fully connect all slaves such that they exchange their Frontiers with everyone else during the resolution stage. But this has the same disadvantage as the Star algorithm, because broadcasts can only be sequentially simulated. What is required is a compromise, as found in the next algorithm.

5.3.4. Pulse-cycle algorithm

Pulse cycle is similar to Unidirectional cycle, except that data can flow in both directions along the cycle, as displayed in Figure 6. On finishing its local engagements, a slave sends its Frontier to its two neighbors and then waits for their Frontiers in turn. On receiving them, it updates its own Frontier, resolves

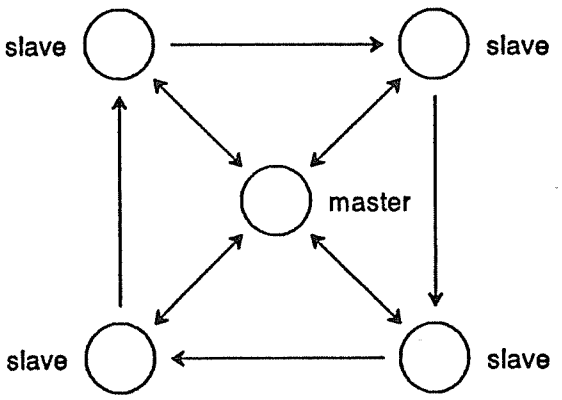


Figure 5. Unidirectional cycle configuration

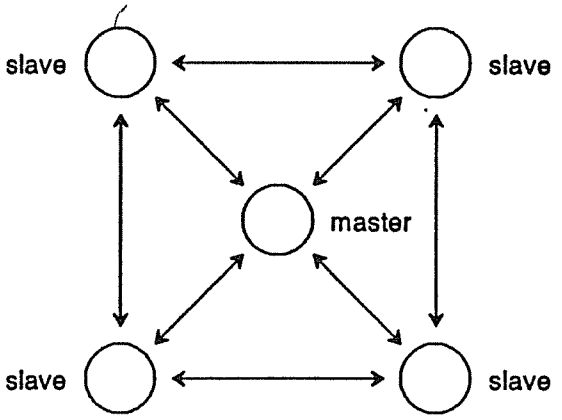


Figure 6. Pulse cycle configuration

conflicts and updates its global engagements, and resumes engagements for its newly rejected men. Termination is announced if the engagements are complete.

The advantage over Unidirectional cycle is that distant updates arrive at a slave more quickly. Of course, the resolution stage is slower, requiring twice the number of messages. We obtained the best performance results from this algorithm.

The following is the pseudo-code for the Pulse-cycle algorithm.

```
process Master(SlaveLink0 .. SlaveLink6 : link; NumProcesses, ProblemSize : integer);
-- 7 links for each of the seven possible slaves
-- NumProcesses is the number of slaves used in this run
-- ProblemSize is the number of men (women) in this run

entry FinalResults(MArray: MatchArray);
-- when a slave detects termination, it connects on this entry
begin
        reply;
        Matching := MArray;
end FinalResults;

procedure Start;
-- problem generated and broadcast to slaves
begin
        bind links to entries;
        generate MaleArray;
        generate FemaleArray;
        foreach i in [0 .. NumProcesses-1] do
                connect SlaveData(MaleArray, FemaleArray, etc) on SlaveLink i;
        end;
end Start;
```

```
procedure Terminate;
begin
        wait for a slave to signal termination;
        output matching;
end Terminate;

begin
        Start;
        Terminate;
end Master.

process Slave (ClockwiseLink, AnticlockwiseLink, MasterLink: link);
-- one link for the master, the other two for its neighbors in the cycle

entry SlaveData(MaleArray, FemaleArray, and other parameters);
--- data that the master sends are collected

entry TransportClockwise(AnticlockwiseFrontier: PointerArray);
-- data from the anticlockwise neighbor are collected

entry TransportAnticlockwise(ClockwiseFrontier: PointerArray);
-- data from the clockwise neighbor are collected

procedure Propose;
-- a suitor is chosen, he proposes to the next woman in his list, and
-- the identity of her fiancé is established
begin
        CurrentSuitor := an element from SetOfSuitors;
        Frontier[CurrentSuitor] +:= 1;
        WomanProposed := MaleArray[CurrentSuitor][Frontier[CurrentSuitor]];
        Fiancé := Matching[WomanProposed];
end Propose;

function Accept: boolean;
-- the proposed woman decides between the suitor and her fiancé
begin
        if no Fiancé yet then
                Matching[WomanProposed] := CurrentSuitor;
                return true;
        elsif
                FemaleArray[WomanProposed][CurrentSuitor] >
                        FemaleArray[WomanProposed][Fiancé] then
                return false;
        else
                Matching[WomanProposed] := CurrentSuitor;
                if Fiancé local to slave then
                        SetOfSuitors +:= { Fiancé };
                end;
                return true;
        end;
end Accept;
```

```
procedure SequentialRun;
-- this is the serial portion of each slave where local men are engaged
begin
        while SetOfSuitors <> {} do
                Propose;
                if Accept then
                        SetOfSuitors -:= { CurrentSuitor };
                end;
        end;
end SequentialRun;

procedure SendGetData;
-- slave sends out its Frontier to neighbors, and waits for theirs in turn
begin
        connect TransportClockwise(Frontier !) on Clockwiselink;
        connect TransportAnticlockwise(Frontier !) on AnticlockwiseLink;
        wait for Frontiers from neighbors;
end SendGetData;

procedure Resolution;
-- the resolution stage
-- after receiving Frontiers from neighbors, updations take place
begin
        foreach i in [1..ProblemSize] do
                Frontier[i] := max(AnticlockwiseFrontier[i],
                        Frontier[i],
                        ClockwiseFrontier[i]);
        resolve conflicts and update Matching;
        if all men engaged then
                connect FinalResults(Matching !) on MasterLink;
        else
                form new SetOfSuitors;
        end;
end Resolution;

procedure Initialize;
begin
        bind links to entries;
        wait until data is received from master;
        initialize SetOfSuitors to contain local men;
        initialize Frontier to the start of each man's list;
end Initialize;

begin
        Initialize;
        while true do
                SequentialRun;
                SendGetData;
                Resolution;
        end;
end Slave.
```

5.4. Performance results

We did not experiment with the Bipartite algorithm. However, we performed extensive tests on the other three distributed implementations and obtained results that reflect their capabilities under different conditions.

The fact that a process could govern any number of men greatly simplified the testing phase. It was fairly easy to run the programs on a number of different problems without having to recompile, edit connector files or perform any other major changes. The programs had very little I/O to perform. Random marriage problems were generated within the processes, based upon a few inputs such as n and a random seed. It was easy to experiment with a large number of problem instances.

The three characteristics measured were average **efficiency**, average **message traffic**, and **worst-case behavior**. A detailed report of our results follows.

5.4.1. Efficiency

In measuring efficiency, our major problem was in constructing large enough problems (150X150 matrices) so as to achieve reasonable speedups. The message overhead of our distributed implementations made speedups impossible for smaller problem sizes. On the other hand, memory limitations prevented us from actually creating such matrices. We circumvented the problem by simulating large matrices. The technique involved actually constructing small, random matrices, and assuming that they are repeated in quadrants (or nonants) of the larger matrices. Simple transformations were employed to avoid the resulting regularity. We obtained reasonably random data in the process.

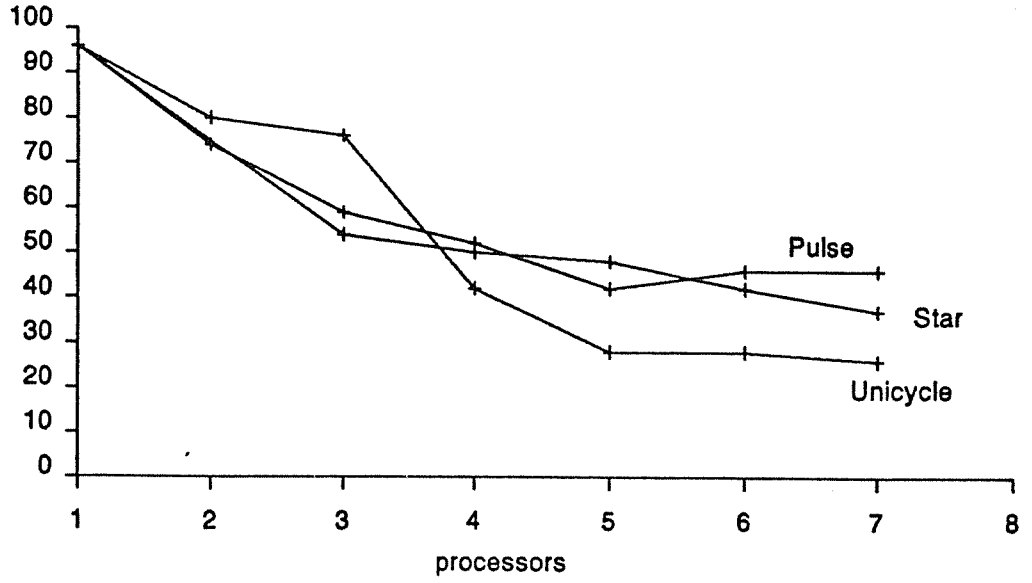
For 180X180 problem sizes (Graph 17), efficiency quickly deteriorates with an increase in the number of machines. There is little to choose between the Star and Pulse algorithm, though they dominate over the Unidirectional approach with an increase in machines. For 400X400 problem sizes (Graph 18), the results are better, with the Pulse model outperforming the others.

5.4.2. Message traffic

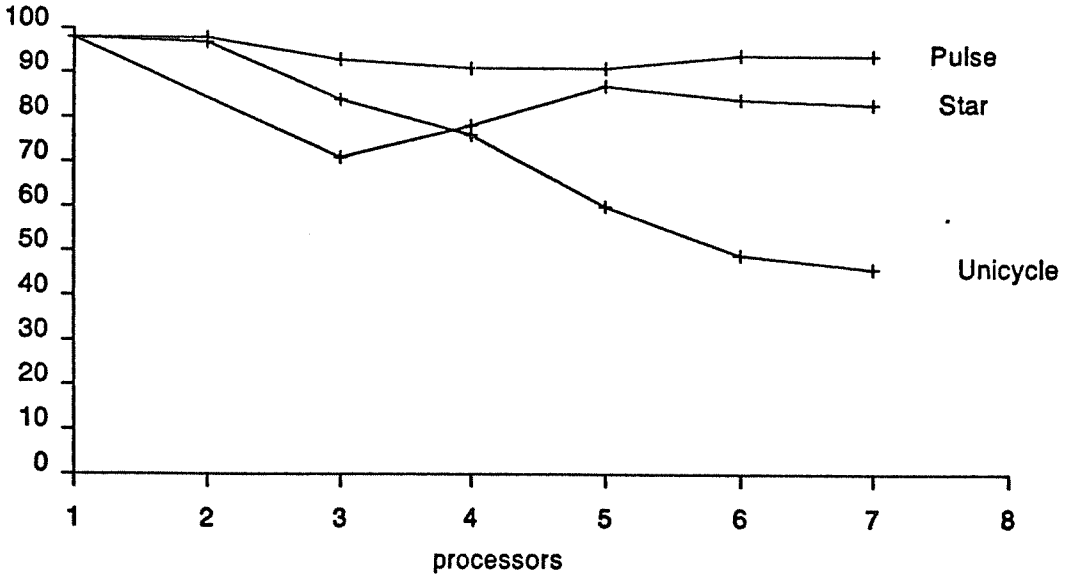
We measured the average number of **remote procedure calls** (RPCs) per process. The linear nature of the curves obtained (Graph 19) leads us to conclude that though increased distribution will not result in combinatorial explosions in message overhead, efficiency will gradually deteriorate. However, this overhead remains relatively unaffected with increases in problem sizes, especially for the Pulse algorithm. We can therefore expect all the implementations to perform better on larger problems.

5.4.3. Worst-case behavior

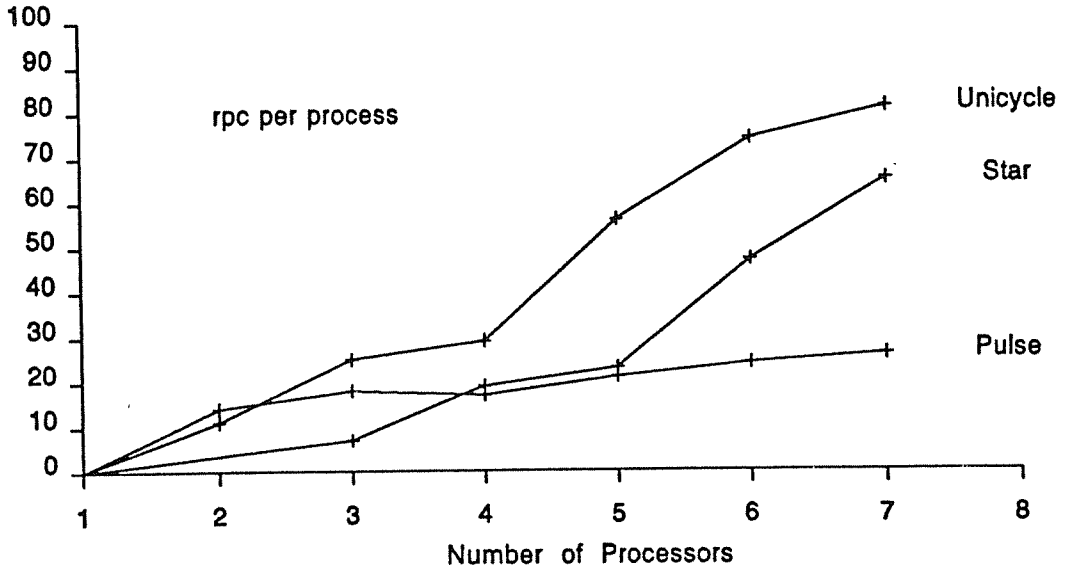
Even though the algorithms have promising average case behavior, special cases of marriage problems can be constructed for which they degenerate to the



Graph 17. Efficiency for n=180

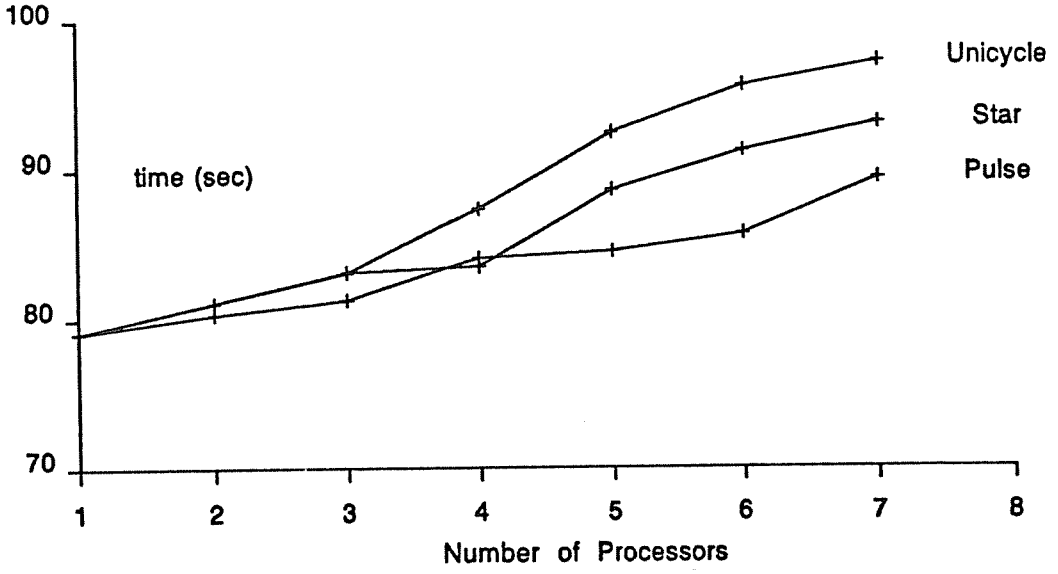


Graph 18. Efficiency for n=400



Graph 19. Remote procedure calls, n=400.

serial algorithm. Specifically, the problems are tailored such that only one machine is allowed to be active at any time. Consequently, apart from having to do the total work in serial, the overhead of the resolution stage also reduces efficiency. The details of constructions of these examples are omitted from the report. Graph 20 shows the time required for solving a worst-case problem with $n = 400$.



Graph 20. Worst case, n=400.

5.5. Experience with Lynx

Programming in a distributed programming language was a welcome change since our prior experience was limited to sequential programming. Consequently, it is very difficult for us to compare Lynx with other similar languages. Even so, we list a few comments on our experiences with Lynx software.

- Remote procedure calls made program writing easy, because it automatically imposes structural programming styles. Of course, it may cause loss of efficiency, but these arguments are reminiscent of the **goto** controversy in conventional programming. Synchronization issues turned out to be easy because of RPC calls.
- Mutual exclusion was automatically provided by Lynx semantics, specifically by the concept of single active threads in processes, because different threads execute as coroutines rather than as separate processes competing for CPU and memory.

A major disadvantage of Lynx is its lack of dynamic features such as dynamic memory. This lack prevents Lynx from being very practical. The syntax is also irritating in spots, such as having to declare all the links a process may need, even though only a few may be used during a run. The real defect lies in having to connect unused links to each other in the connector file which produces extremely inelegant structures.

It is not very clear why links were implemented as duplex channels. A unidirectional implementation may have sufficed. Duplex links can be simulated easily by the unidirectional links.

The Charlotte connector file syntax can obviously be improved, which would avoid having to hardwire connections in the file.

The software, though certainly not of production quality, behaved very nicely in our case. One problem is that runtime checks produce very low level error messages, most of which is meaningless to the inexperienced user. We believe that experimental software should emphasize stronger error reporting facilities.

5.6. Conclusions

We conclude with suggestions for possible improvements in our algorithms. A major deterrent to further efforts is that the serial algorithm itself is very fast. But that fact offers a lot of challenge in trying to fine tune distributed implementations so as to minimize message overhead. For instance, it may be necessary to burden machines with serial computations simply because it takes longer to subdivide the problem and allocate it to other machines.

In all our current implementations, each slave carries a full copy of the women's preference lists. It may be worthwhile to seek a compromise between space and communication time by distributing the matrix across machines.

For the implementation of the resolution phase, we have experimented with centralized and cycle configurations. Many others are possible. For instance in the Star configuration, multiple masters could maintain global data structures so as to distribute the burden from one master. The pulse algorithm would most likely be improved by using a deBruijn network, in which each slave has two predecessors and two successors, and the diameter of the slave graph is $\log p$ for p slaves [Imase81].

One possibility that attracted our interest was the dynamic allocation of work to slave processes. In our current designs, after the resolution stage, each slave resumes with its new set of suitors. The sizes of such sets, and consequently the work to be done before the next resolution, may differ across slaves. Some slaves become idle while others are still busy. It would be profitable to reallocate the rejected men almost equally among the slaves. We did not consider this idea because it would require a lot of redistribution of data structures around the system. Probably a more efficient scheme of storing data would work.

We have emphasized the man-oriented algorithm. Radically different approaches could be tried. An idea is to try the woman oriented algorithm simultaneously. Of course the matching may not be the same but it would be stable. Whichever calculation finishes first can interrupt the other.

Finally, the problem can be extended to find **all** stable matchings, a considerably more difficult problem [Wirth76]. Here distribution becomes a more complicated issue, because care must be taken that the slaves do not generate duplicate answers.

6. Minimal Dominating Sets

Experimenter: Mukesh Kacker

6.1. Introduction

Computers linked together in a network make extensive use of knowledge related to the topology of the interconnection. The networking software in particular relies on such information. The routing tables, for example, include cost measures associated with destination nodes that are derived from the topology. Topological information can be derived by maintaining graphs isomorphic to the network topology at each site. This, however, is not a feasible approach in practical terms. Some of the factors that cause topological changes are temporary and even transient. The practical approach is redetermine topological information at any site when necessary. Echo algorithms, as discussed by Chang [Chang79], address this problem.

In this report we describe our experience at implementing one such algorithm designed to compute the minimal dominating set of a graph. In Section 2 we describe the problem and some motivations for for solving it. In the next section the serial algorithm to compute the problem is presented. In Section 4 we describe the distributed algorithm to solve the problem. In Section 5 we describe our implementation in Lynx and present the pseudo code of the implementation. In Section 6 we discuss the performance measurement experiments and present related tables and graphs. In the next section we discuss experiences with using Lynx and Charlotte. In Section 8 we present conclusions and discuss scope for future work.

6.2. Description of the problem

Let G=(V,E) be a simple undirected graph with a set of vertices V and a set of edges E. We will denote the degree of vertex v by $d(v)$; its neighbors are $N(v) \subseteq V$. By convention, $v \in N(v)$.

D_n is an **n-Dominating set** of a graph G=(V,E) if $D_n \subseteq V$ and $\forall v \in V$, either $v \in D$ or $|N(v) \cap D| \geq n$, that is, every vertex not in D_n is adjacent to at least n members of D.

D is a **dominating set** of a graph G = (V,E) if $D \subseteq V$ and $\forall v \in V$ either $v \in D$ or $N(v) \cap D \neq \emptyset$. In other words every vertex not in D is adjacent to at least one vertex in D. D is a **minimal dominating set** if $(D-\{v\})$ is not a dominating set for any $v \in D$. A *minimal* dominating set is different from a *minimum* dominating set, which is a minimal dominating set with the least number of vertices. The set (V-D) is defined as the **dominated set.**

A possible motivation for computing a dominating set of a network topology graph could be in deciding the placement of replicated resources. There are many applications where several copies of the same resource (such as a directory or table) are placed at various machines of a distributed system in order to

improve availability or fault tolerance of the system. A designer might wish to optimize the number of copies to be maintained in order to provide the required degree of availability or fault tolerance. A typical case may be to ensure there are at least n copies of the resource at most m hops away from each machine in the system. When $m=1$, the problem of placing the minimum number of copies is the same as finding the minimum n-dominating set of the system. For $n=1$, the problem is the same as finding the minimum dominating set. Replicated resources placed at the nodes in the minimum dominating set ensure that there is at least one copy of the resource at most one hop away from each node in the network.

6.3. Serial algorithm

Computing the minimal dominating set of a graph is straightforward. We will say a member of the current set is **responsible** for neighboring vertices that are not neighbors of any other set member. We start with an empty set. For each vertex v in G, v is added to the set if it has neighbors that are not yet adjacent to a set member. At that time, any member of the set that is now responsible for no vertices is dropped from the set. The algorithm follows the pseudo-code presented below.

```
type
        vertex;

var
        MDS : set of vertex := {}; -- the dominating set;
        Cover : set of vertex := {}; -- MDS plus its neighbors
        N : array [vertex] of set of vertex := Initialize(G); -- v ∈ N[v]

function NextNode(L:list of nodes);
--advance current_ptr ;
begin
        NextNode := L(current_ptr)^.next^.node
end NextNode ;

procedure ComputeMDS(G:graph) ;
begin
        forall v ∈ G do
                if N[v] - Cover ≠ [] then
                        MDS +:= {v};
                        Cover +:= N[v];
                        recompute Responsible(n) ∀n ∈ MDS;
                        -- Addition of v may cause Responsible sets to change
                        forall w ∈ MDS do
                                if Responsible(w) = [] then
                                        MDS -:= {w};
                                        recompute Responsible(n) ∀n ∈ MDS;
                                        -- Deletion of w may cause Responsible sets to change
                                end;
                        end;
                end;
        end;
end ComputeMDS;
```

6.4. Distributed algorithm

The distributed algorithm for computing the minimal dominating set [Raghuram86] makes certain assumptions about the underlying model.

- Computation is to be done by processes that cooperate through messages.
- One process is associated with each vertex in the graph. All it knows about the graph is its neighbor set.
- Each process can transmit only to its graph neighbors.
- The graph does not change during the execution of the algorithm.
- All messages are eventually received.
- Messages may arrive out of order.

This model closely matches the environment provided by Lynx.

Any machine can initiate a request to compute the minimal dominating set, and it gets back a reply indicating the dominating set of the graph minus its own vertex. It can then modify this answer to account for its own place in the

graph.

The algorithm uses three kinds of messages:

- A **request** message is used to propagate a request to the neighbors of a node. This message contains the identity of the originator.
- An **ignore** message, which tells the recipient to ignore the sender.
- A **reply** message conveys the result of a previous request.

We need to introduce some terms before describing the distributed algorithm.

A **parent** of a process i is denoted by Parent(i). It is the node from which the first request message is received by i. The **children** of a node i, denoted by Children(i), include i itself and all nodes whose parent is i.

The first step is to build a spanning tree of the graph. This is accomplished by the standard wave algorithm, starting at the requesting process. Every process considers the first neighbor that sends it a request to be its parent. It then sends requests to all potential children, that is, its neighbors except its parent. It sends ignore messages to all later request messages. Its children are the potential children that do not send ignore messages. A process that receives ignore messages from all its potential children is a leaf.

The second step is to compute the minimal dominating set in a bottom-up wave up the spanning tree. A message sent to a parent during this step contains a set of (d,C,NC) triples:

d an element of the minimal dominating set of the subtree rooted at the reporting child

C the neighbors of d (in the graph) that are d's descendents

NC the neighbors of d (in the graph) that are not d's descendents or d's parent.

Again, we say that a vertex in the dominating set is **responsible** for those neighbors that are not neighbors of other members of the dominating set. We will keep a fourth field, S, internally with each element of the minimal dominating set to indicate the neighbors for which d is responsible.

When all children have responded with replies, process i performs the following algorithm:

```
1     Responsible := Children ∪ {i};
2     DomSet := (i, Responsible, N[i] − Responsible, Responsible) −− initial value
3     Cover := N[i];
4     foreach triple t = (r, C_r, NC_r) in each reply do
5           if C_r ⊈ Cover then
6                 foreach quadruple q = (d, C_d, NC_d, S_d) in DomSet do
7                       S_d −:= (S_d ∩ (C_r ∪ NC_r));
8                       if S_d = ∅ then
9                             recompute Cover without q and with t in DomSet.
10                            if C_d ⊆ Cover then
11                                  delete q from MDS;
12                            else
13                                  S_d := C_d - Cover ∩ C_d ;
14                                  Cover ∪:= C_d ∪ NC_d;
15                            end;
16                      end;
17                end;
18                S_r := C_r − C_r ∩ Cover ;
19                Cover ∪:= C_r ∪ NC_r;
20                DomSet +:= (r,C_r,NC_r,S_r);
21          end ;
22    end;
23    transmit first 3 fields of each element of DomSet to parent.
```

In lines 1–3, the global variables are initialized. The minimal dominating set contains the node itself to begin with. It is represented as a set of quadruples q=(d, C, NC, S) described above. Cover is the neighborhood of the nodes in the minimal dominating set. Line 4 scans each node t in the reported dominating sets of i's children. Line 5 succeeds if node t has some children that are not yet covered by nodes in DomSet, in which case, t should be added to DomSet. Before doing that in line 6 we scan all nodes in currently in the minimal dominating set to see if inclusion of t makes any of them a candidate for removal. Line 8 modifies the responsible set by removing from it nodes covered by t. When the conditional in line 10 finds the responsible set empty, q is a candidate for removal. To really see if the entire neighborhood of q is covered by other members of the dominating set, Cover is recomputed in line 9 assuming q is not in DomSet and t is in it. If all children of q are in Cover (line 10), q can be safely removed from DomSet. Otherwise, the responsible set became empty because of testing of q at line 7, and it is restored to its proper value along with Cover. In lines 18–20, node t is added to DomSet after we determine its responsible set and modify Cover. After all nodes in all reply messages are processed, DomSet is ready to be sent to i's parent as a reply to the request message that started i's involvement in the first (tree-building) step of the algorithm. Only the (d, C, NC) fields are sent.

This algorithm need not compute the same result each time. There may be several minimal dominating sets; which one is reported depends on the order in which messages arrive.

6.5. Implementation in Lynx

The implementation of the distributed algorithm presented above was done using two process types. One was the single instance of the **I/O** process. The other was the **calculator.** There is a calculator for every vertex in the graph. The I/O process initializes the links between calculators. (We could have used the connector instead.) The calculator implements the distributed algorithm described in the previous section. Upon getting the first request message, a calculator broadcasts it to *all* its neighbors. There is no way to know which potential children have already submitted request messages that we have not yet seen.

Since the I/O process has links to all nodes, the maximum number of nodes that can be used is limited by the LYNX implementation limit on the maximum number of links a single process can have. The pseudo code of the implementation follows.

6.6. Pseudo code

```
process IO (link1,link2...linkmax);

-- Initialize process graph.
-- Similar to process IO in Minimal Spanning Tree problem described
-- in Second Experience Report
end IO.
```

```
process calculator(MyName: integer; ioLink : link ) ;
--This algorithm executed at each vertex of graph.
type
        nodeState = (
                sleeping,  --initial state of process
                requesting, --state after io done and links for
                        --process graph set up.nodes receive/transmit
                        --requests in this state.
                responding, --state after mds received from  children
                        --local computation of mds in this state
                unknown    -- for debug
                )
        quad_set = set of
                record
                        node_num: nodeId; -- node identification number
                        child_neighbors : set of nodeId;
                                --neighbors of "node_num" which are
                                --its children in spanning tree
                        non_child_neighbors : set of nodeId;
                                --neighbouurs of "node_num" which
                                --are not children or parent.
                        slaves : set of nodeId;
                                --child nodes covered exclusively
                                --by "node_num"
                end;
        triple_set = set of
                record
                        node_num: nodeId;
                        child_neighbors, non_child_neighbors : set of nodeId;
                end;

var
        Neighbor_links : set of links; --links to all neighbors except
                -- parent node.
        Request_Set : set of links; --links to children
        Pending_Requests : set of links; --links to children from whom
                --response is awaited.
        Phase : nodeState; --state variable of calculator.
        MDS : quad_set; --set with nodes in minimal dominating set
                --and information about their neighborhood.
        MDS_child : triple_set; ---set with nodes in mds of children
                --and info about their neighborhood.

procedure Initialize;
        --set initial values of all global variables.
end Initialize;

entry NewEdge(ownlinkend : link ,neighbor_id : nodeId );
        --used by i/o process while setting up process graph
        --receives link to neighbor and identity of process
        --that is the neighbor.
end NewEdge;
```

```
entry WakeUp;
        if Phase = sleeping then
                Phase = requesting;
        else
                --error
        end;
end WakeUp;

entry Request(sender : nodeId );
        if FirstRequest then
                register sender as parent node
                register "curlink" as link to parent
                set up Request_Set,Pending_Requests
                        to all neighboring nodes/links except parent.
                foreach edge in Request_Set do
                        --broadcast request to neighbors
                        connect Request(MyName|) on edge;
                endfor;
        else
                --remove sender from Request_Set,Pending_Requests etc.
                connect Ignore_request(MyName) on curlink;
                        --send ignore message to sender
                if Pending_Requests = null then
                        Phase := responding;
                end;
end Request;

entry Receive_MDS_children( mds_children : triple_set);
        --receives reply triples from children who call this entry
end Receive_MDS_children;

procedure Tx_MDS;
        --once minimal dominating set is computed, this
        --procedure sends it to parent
end Tc_MDS;

procedure ComputeMDS;
        --this procedure computes the main body of the computation
        --described in the algorithm.
end ComputeMDS;
```

```
begin -- main body of the process
        Initialize;
        Phase = sleeping;
        await(Phase = requesting);
        if MyName = originator then
                initialize Request_Set to all neighbors
                initialize Pending_Requsts similarly;
                foreach edge in Neighborhood do
                        connect Request(MyName| ) on edge;
                endfor;
        end;
        await(Phase=responding);
        ComputeMDS;
        if (MyName = originator) then
                print nodes in minimal dominating set
        end;
end calculator;
```

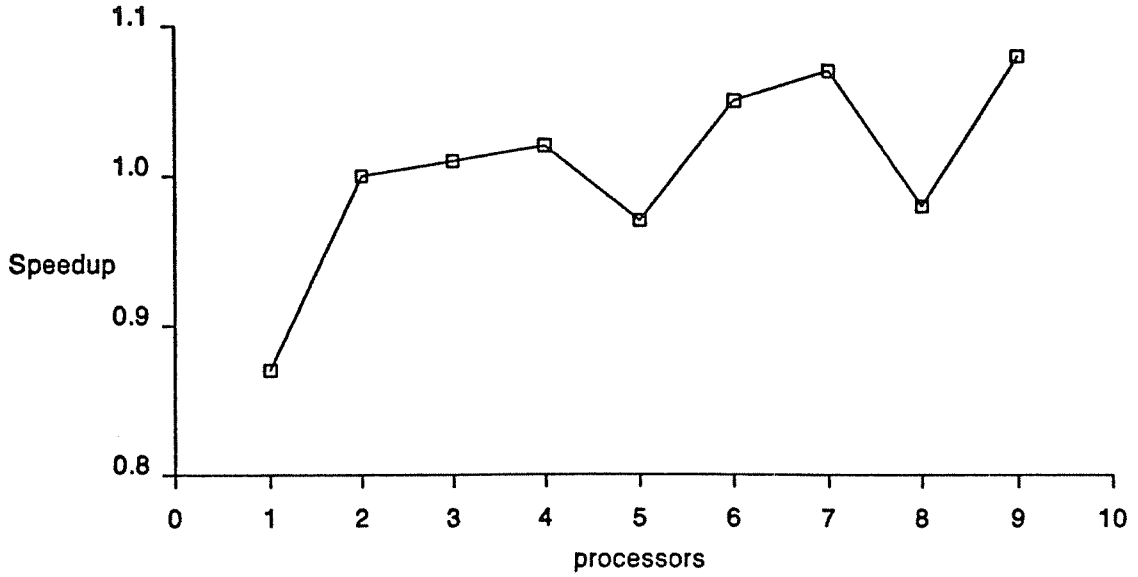
6.7. Experiments and results

The Lynx program presented above was executed on a graph of 10 nodes. Though a 16 node graph can be used, the need to measure performance with all node processes on one machine caused memory overflows for larger graphs. The answer computed by this algorithm is not unique. It depends on order of arrival of messages and hence on network and host processor loads. We report averages taken over 10 execution runs; they showed great variation. The first set of measurements relate to the total running times of the algorithm.

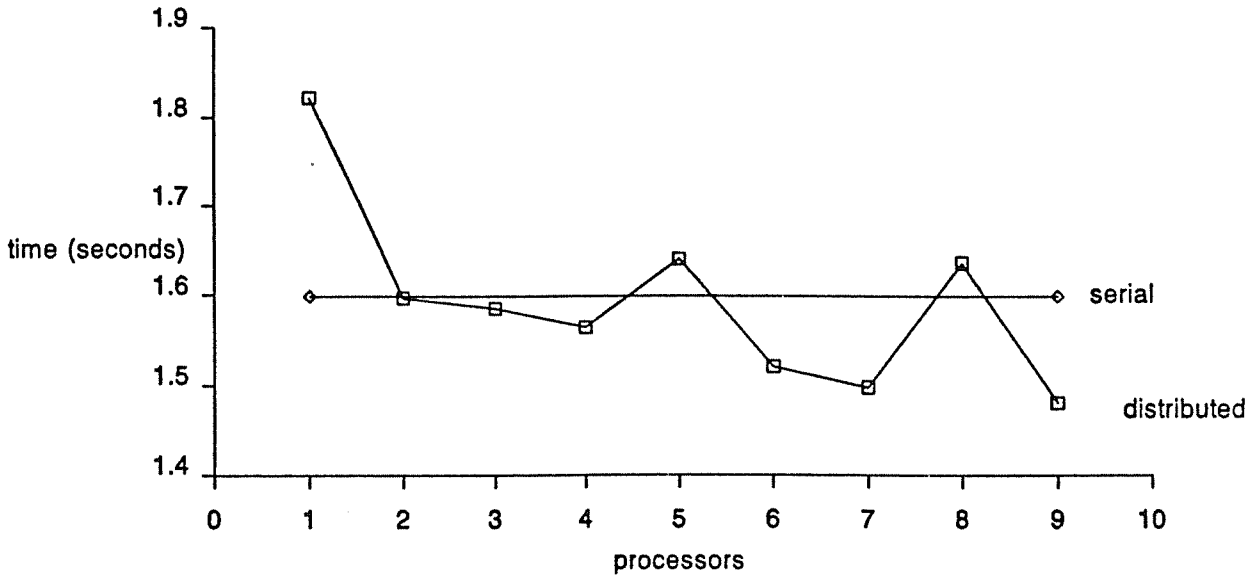
Speedup has been computed comparing the average run times with the run time of a serial version. This algorithm does not run faster than the serial version except on occasion by small amounts. However, this algorithm is an exercise in message-based distributed computing where a global system parameter is computed using little global information at any site. Speedup is not necessarily a relevant measure here. Average speedup values range from 0.87 to 1.08, with extremes of 0.45 to 2.52. The speedup curve is shown in Graph 21, and average run times for different number of processors are plotted in Graph 22.

An interesting measure was the average time taken for messages. The message time measured includes communication time and blocking time at the entries. Most entries have **reply** at the beginning to avoid blocking. The average message time steadily declined as more processors were used. This occurs because the wait for a message to enter the entry is reduced when there is less competition with other threads also waiting for other processes. With more machines, the number of processes per machine is less. These measurements are recorded Graph 23 for average runs.

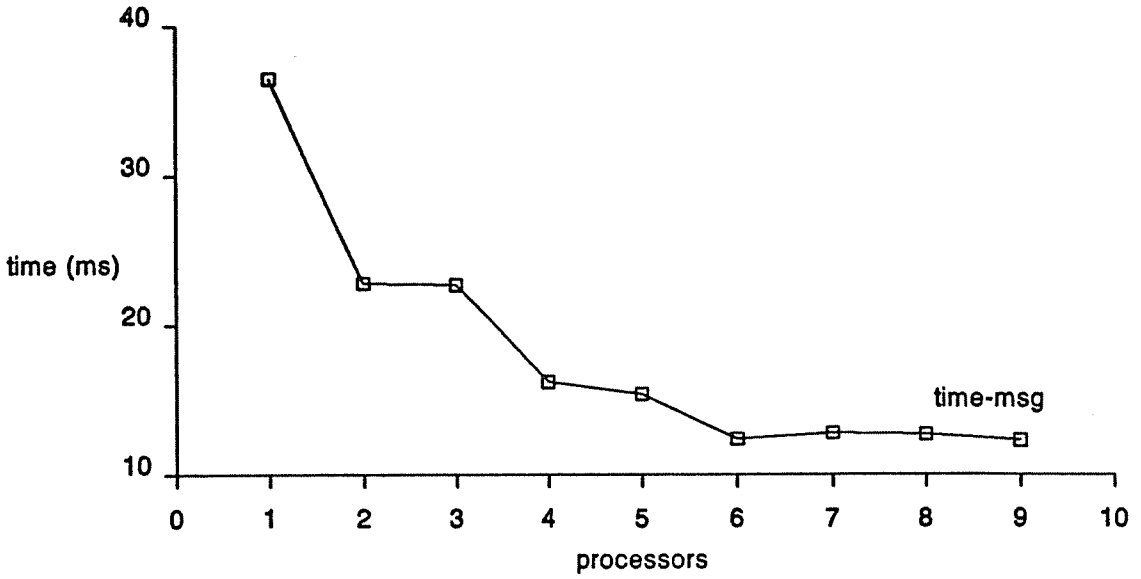
Another interesting measurement shows how time is divided among message sending, synchronization, and computation. **Message-sending** time is the communication time and the time a thread is blocked waiting for its turn to get into an entry. **Synchronization** time is the time a process waits at **await**



Graph 21. Speedup

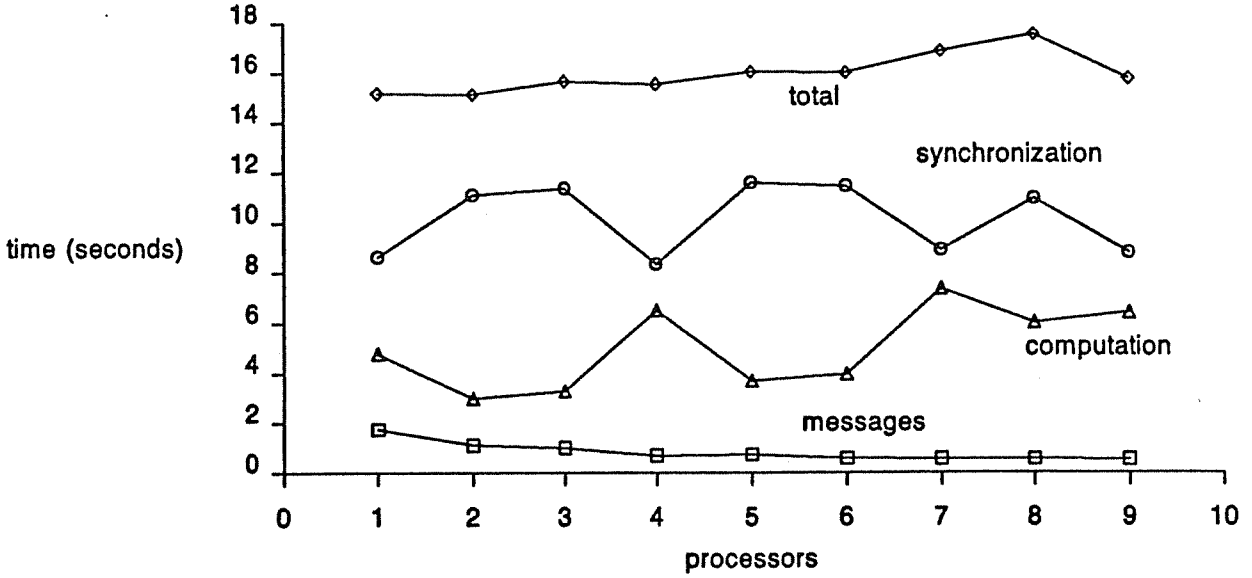


Graph 22. Average run times

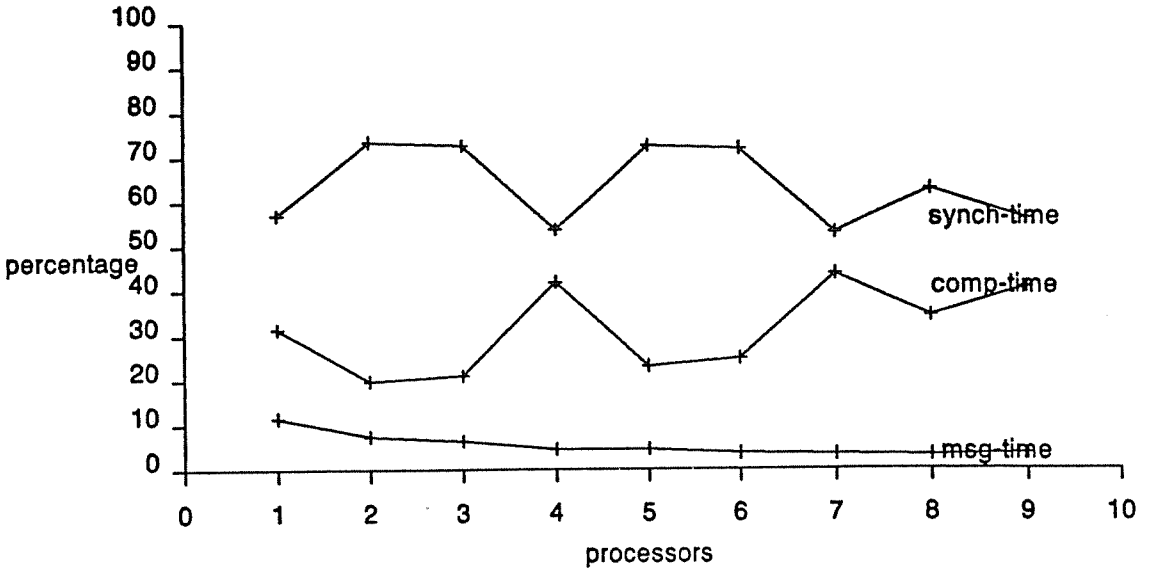


Graph 23. Average message times

statements. **Computation time** includes the time taken in context switches between threads. Graph 24 shows these times as absolute values, and Graph 25 shows them as percentages of total time. Message passing time is not the primary expense, as one might be tempted to believe. The time taken to synchronize various activities is the main culprit. The frequent context switching between various threads also takes much time.



Graph 24. Breakdown of times



Graph 25. Breakdown of times by percentage

6.8. Experience with Lynx and Charlotte

- The data abstraction of links as data types is convenient for initializing process graphs.
- It would be nice to have unique process identification numbers (as in SODA [Kepecs84] and Thoth [Cheriton79], for example).
- If it were possible to screen messages by the identity of the caller, a number of redundant messages could have been eliminated. In our application, the first request message could have avoided sending requests to processes whose requests were already behind it in the queue for the same entry.
- A debugger for Lynx would be greatly appreciated.
- A broadcast mechanism would have been useful for our application to let us send messages over sets of links.

6.9. Conclusions and future work

The Charlotte and Lynx environment is very convenient for experimenting with message-based distributed programs. In our experiment we discovered that synchronization time and not message passing time is a big overhead.

The present algorithm and implementation simulate an environment of a network of processors where computation proceeds by request and response messages. We can imagine an alternative implementation that does not use "ignore" messages. Requests are sent as before, and the reply indicates whether the request was the first one to reach its destination. This implementation eliminates the need for a node to wait for a response from all its

neighbors and it can start computing as soon as any node sends it a response. This method would reduce the heavy synchronization overhead in our implementation. This method was also implemented, but preliminary measurements show that it is slower than our other implementation. The reason for that needs to be investigated.

7. The N-Queen Problem

7.1. Introduction

The objective of the N-Queen problem is to place n queens on a $n \times n$ chess board in such a way that each queen will not attack each other. Two queens attack each other if they are on the same row, the same column or the same diagonal. Finding a solution to the n-queen problem can be very expensive. Only a small fraction of the $\binom{n^2}{n}$ placements are solutions.

In the following sections, we will discuss how to represent the problem as a search tree and how to prune that tree. We then discuss how we can partition searching the pruned tree across processes. In the last section, we discuss the performance of several distributed algorithms and suggest some possible enhancements.

7.2. The problem space

All solutions have the property that each column contains one queen. We can treat the problem as a search through a tree, where a node at level l represents a situation in which a queen has been placed in each of the first l columns. The children of a node are the legal ways an additional queen can be placed in the next column. Figure 7 shows the tree for $n=4$.

It is not hard to write a recursive routine that builds just this tree and stops when it finds the first solution, that is, the first level-l node. The routine can keep track of the current column and diagonal constraints in Boolean

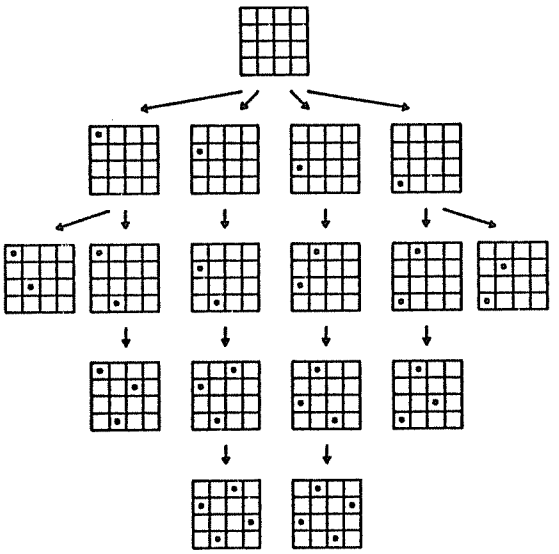


Figure 7. Search tree for four queens

arrays.

A chess board is symmetric; therefore, given a solution pattern, we can generate more solutions by rotating or inverting the chess board. If a solution pattern is asymmetric, it is one of a family of eight equivalent solutions (4 rotations, 2 mirror images). We can reduce the cost of finding solutions by pruning the tree to find only one representative of each solution family.

The first way to accomplish this pruning is to only consider $\lceil n/2 \rceil$ nodes at level 1, that is, those that place the first queen in the first half of the column. All other solutions are mirror images of the ones we will find.

A second pruning is achieved by refusing to consider any solution for which $Q[1] = j$, $Q[k] = n$, and $k<j$. Any such solution is equivalent to one rotated 90^0 counterclockwise, where $Q[1] = k$. We will have already found that solution earlier. This constraint can be generalized to form a set of **prohibited boundaries** that a solution must not touch if it starts with $Q[1] = j$. Figure 8 shows the permissible and prohibited boundaries.

On the other hand, if $Q[k] = j$, then we must continue the search, because we cannot be sure we would have found such a solution earlier. We therefore generate duplicate solutions. We remove duplicates by a third pruning method, which only applies to leaves. A family can have up to 8 duplicate members. We take the first in lexicographic order as the representative member. We discard any solution that is greater than its representative.

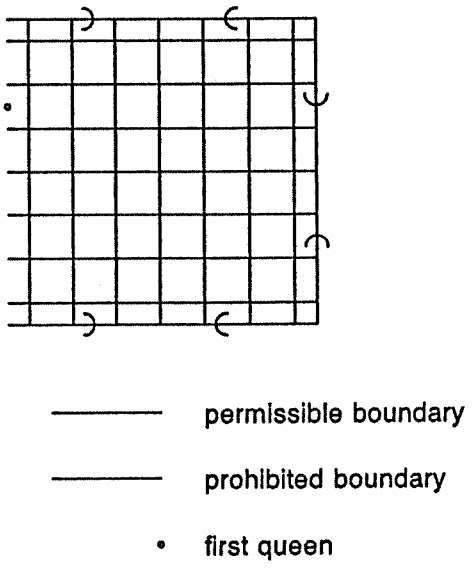


Figure 8. Prohibited boundaries

7.3. Distributed algorithm

A complete path is represented by an n-digit base-n number, ranging from L = <1><2><3>..<n> to U = <n><n−1>..1. We will have different processes search different subranges. We arrange the processes in a master-slave configuration with one master connected to p slaves. The master is responsible for partitioning the problem space according some partition strategy and distributing it to each slave. It is also responsible for collecting results from the slaves.

In our implementation, we do not try to balance work dynamically between slaves, nor do we give a slave more work after it finishes its first batch. A more dynamic method for distributing trees has been described by Manber and Finkel [Finkel86d].

We considered three different partition strategies.

- **Even.** The range from L to U is divided numerically into p equal-sized pieces, each of size $(L-U)/p$ (with some rounding). The first pieces are likely to have more solutions than the later ones, since there are more cutoffs in later ones.
- **Linear-increment.** The range from L to U is distributed with each succeeding range linearly larger than the previous one.
- **Fibonacci**partitioning. Each process i gets a range whose size is the sum of the sizes of the ranges given to i−1 and i−2.

Here is the pseudo-code for the distributed algorithm.

```
process ParentNode(Child_1, Child_2, ..., Child_N: link);

type
        PathArray = array [1..MaxN] of integer;
        TimerArray = array [1..NumChild] of integer;
var
        N: integer;
        Timers: TimerArray;
```

```
entry Report(Cid, NSoln, NNodes, TCount: integer);
var
        Time: integer;
begin
        reply;
        Time := GetTime;
        Report child number
        Report total elapsed time (Time - Timers[Cid])
        Report number of solutions
        Report number of tree nodes visited
        Report number of solutions tossed
end Report;

procedure Init;
var
        StartPath, StopPath: PathArray;
begin
        Ask the user to enter board size
        foreach i in [1..N] do
                Partition the problem space
                Calculate the StartPath and StopPath
                Timers[i] := GetTime;
                connect NQueen(N, StartPath, StopPath |) on Child_i;
        end;

                bind Child_1, Child_2, ..., Child_N to Report;
        end Init;

begin -- ParentNode
        Init;
end ParentNode.
```

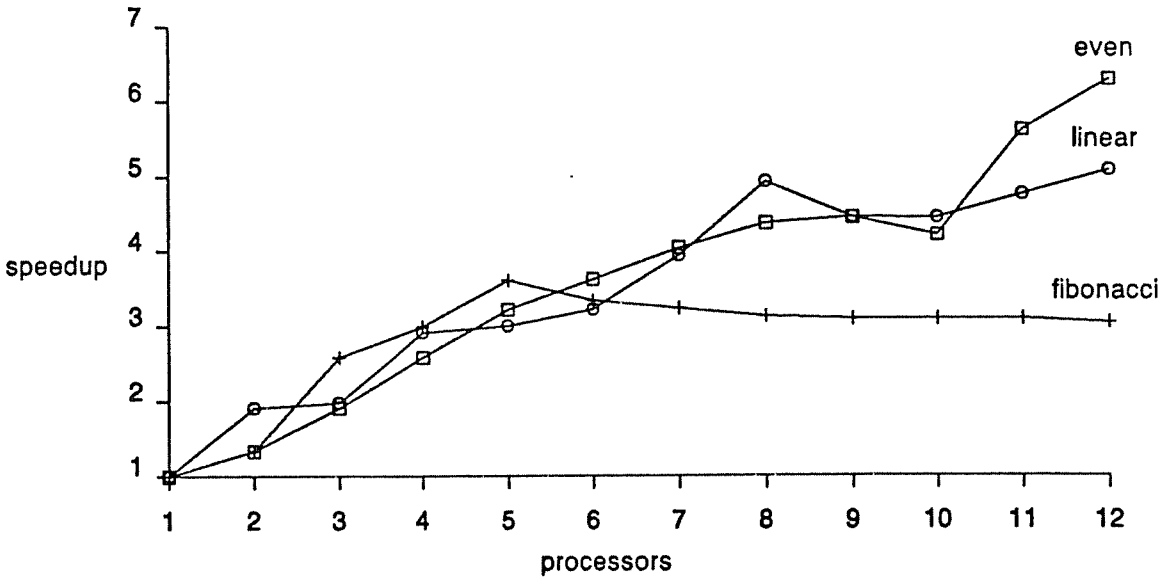
```
process ChildNode(Id: integer; ParentLink: link);

const
        MaxConstraint = 2*MaxN - 1;
type
        DiagConstraintArray = array [1..MaxConstraint] of Boolean;
        ColConstraintArray = array [1..MaxN] of Boolean;
        PathArray = array [1..MaxN] of integer;
var
        ColConstraint: ColConstraintArray;
        MajorConstraint, MinorConstraint: DiagConstraintArray;
        Path, StartPath, StopPath: PathArray;
        N, VisitNodes, NumSoln, TossCount: integer;

procedure AddSoln;
var
        SolnPaths: array [1..7] of PathArray;
begin
        Rotate and invert the primary solution path
        Place the result into the SolnPaths array
        Eliminate symmetry solutions
        Compare each of the remaining solutions with primary solution
        if {Any of those is less than primary solution} then
                Discard the whole solution set
                Increment the TossCount
        else
                Adjust NumSoln to reflect the added new solutions
        end;
end AddSoln;
```

```
procedure Try(Row, Col: integer);
var
        MajorIndex, MinorIndex, NextRow: integer;
begin
        if Path <= StopPath then
                Path[Row] := Col;  -- Add node to Path
                VisitNodes +:= 1;
                if Last Row then
                        if not prohibited region then
                                AddSoln;
                        else
                                TossCount +:= 1;
                        end;
                else
                        MajorIndex := Row + Col - 1;
                        MinorIndex := Row - Col + N;
                        NextRow := Row + 1;
                        ColConstraint[Col] := true;
                        MajorConstraint[MajorIndex] := true;
                        MinorConstraint[MinorIndex] := true;
                        if (NextRow, 1) is not in prohibitive regions and
                                satisfies all constraints then
                                Try(NextRow, 1);
                        end;
                        foreach Col in [2..N-1] do
                                if (NextRow, Col) satisfies all constraints then
                                        Try(NextRow, Col);
                                end;
                        end;
                        if (NextRow, N) is not in prohibited regions and
                                satisfies all constraints then
                                Try(NextRow, N);
                        end;
                        ColConstraint[Col] := false;
                        MajorConstraint[MajorIndex] := false;
                        MinorConstraint[MinorIndex] := false;
                end;
        end;
end Try;
```

```
entry NQueen(BoardSize: integer; StartP, Stop: PathArray);
begin
        reply;
        N := BoardSize;
        StartPath := StartP;
        StopPath := StopP;
        initialize all constraint arrays
        NumSoln := 0;
        VisitNodes := 0;
        TossCount := 0;
        foreach i in [StartPath[1]..StopPath[1]] do
                Try(1, i);
        end;
        connect Report(Id, NumSoln, VisitNodes, TossCount|) on ParentLink;
end NQueen;

begin -- ChildNode
        bind ParentLink to NQueen;
end ChildNode.
```

7.4. Results

The above algorithms have been implemented in Lynx. Experiments were run up to n=12 and p=12. The principal results are shown in Graph 26, which shows the speedup observed for all three distribution methods. Even distribution and linear-increment partitioning have about the same performance, except that even distribution seemed to outperform linear-increment partitioning slightly when the number of machines is high. But the speedup is only about half of perfect performance. Fibonacci partitioning yielded the worst



Graph 26. Speedup for 12 queens

performance. As the number of machines increases, the its speedup is limited to about 3 and then decays.

The reason for the bad performance on Fibonacci partitioning can be observed from Graph 27, which shows the percentage of machine utilization versus number of machines. The percentage of machine utilization is defined as:

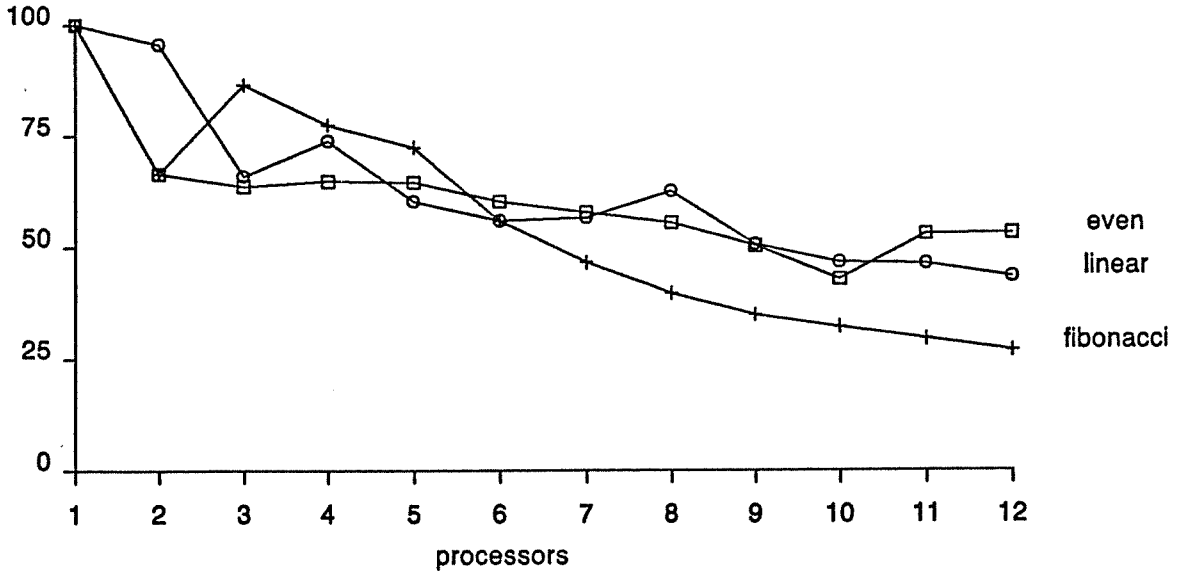
$$\frac{\sum_p Time_p}{p \max_p Time_p}$$

From the graph, we can see that as the number of machines increases, the percentage of utilization falls gradually. This is due to the fact that when the number of machines is large, the probability of a machine getting done early and sitting idle will also increase. The effect is especially significant for the Fibonacci partitioning case.

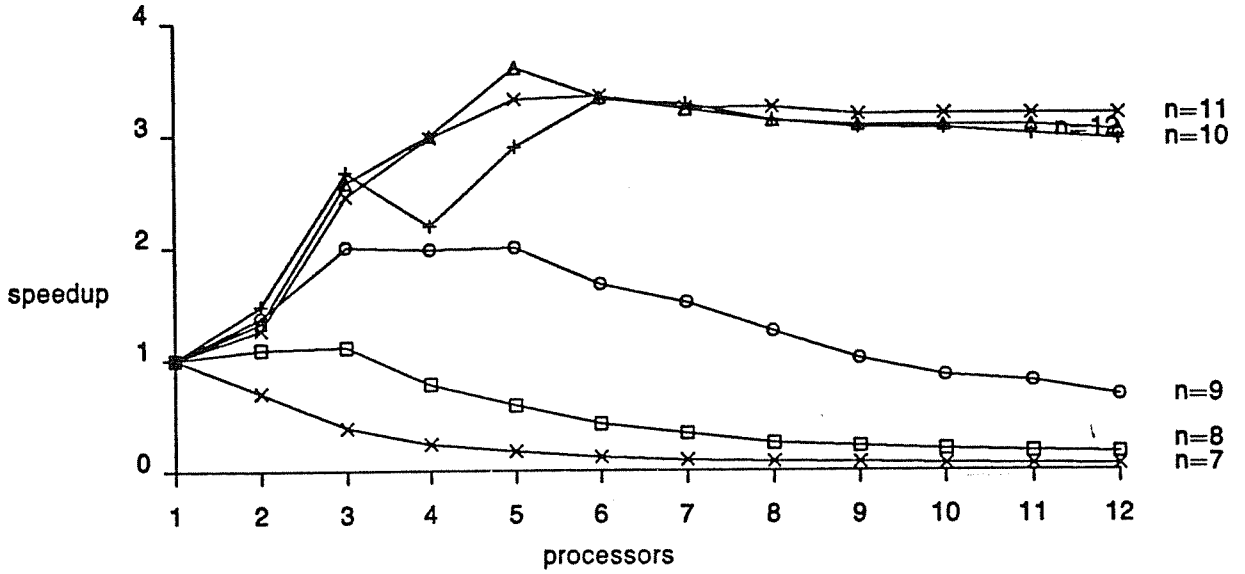
Graph 28 shows how the speedup curve varies for different size problems. When the problem size is decreased below $n=9$, the speedup is generally less than 1, due to the communication and setup overhead on multiple machines.

7.5. Future enhancements

It would be reasonable to try other partitioning methods in an attempt to reduce the idle time on slave machines. Even better, we could start with many small partitions, and let a slave that finishes acquire a new partition to work on. In the most general case, an approach like DIB would work best



Graph 27. Utilization for 12 queens



Graph 28. Speedup, Fibonacci partitioning

[Finkel86d].

8. PLA Folding

Experimenter: Jaspal Kohli

8.1. Introduction

Structured design techniques are becoming essential to ensure logical and electrical correctness while maintaining an acceptably short design time for complex VLSI circuits. One such technique, the Programmable Logic Array or PLA, is an effective tool for implementing multiple-output combinational logic functions. The general structure of a PLA consists of two planes. The AND plane has all inputs (called columns) running through it and produces signals (called rows) corresponding to all product terms in the output functions. These signals then pass through the OR plane and are suitably combined to produce the outputs. The objective of PLA folding is to discover permutations of rows (or columns) which permit a maximal set of column pairs (or row pairs) to be implemented in the same physical column (or row). This results in saving of silicon area as well as a shorter delay through the PLA.

In this report we describe a distributed implementation of an algorithm for Simple Column Folding (to be described later) presented by Hachtel et al [Hachtel82]. This is a follow-up to the article in the previous report [Finkel86b], where the same algorithm was distributed by a different approach, which we will refer to as the **pipelined approach.** We follow a **wisdom-server approach,** in which a server process distributes work and current knowledge and accepts partial results from worker processes. New wisdom is distilled from the partial results and sent back to workers as new problems are distributed.

The report is organized as follows. The simple column folding problem is described in section 2. In section 3, the serial algorithm is described. The distributed algorithm and its implementation in Lynx are described in section 4 and 5 respectively. Results obtained from various runs on Crystal are discussed in section 6. Also a comparison with the results from the pipelined approach is made. Section 7 contains remarks about the experience of using Lynx and Crystal. Concluding remarks and some suggestions for future work constitute section 8.

8.2. Simple column folding

A PLA can be described in symbolic form by a matrix called the **personality matrix.** The columns of the matrix represent the input lines, and the rows represent the product terms. If an input appears in a product term, the corresponding entry in the personality matrix is a one; otherwise it is a zero. Most PLA personality matrices are very sparse, so that a straightforward implementation would result in significant waste of silicon area.

Simple column folding (SCF) involves discovering input pairs that can share the same physical column. Folding requires that rows using the upper part of the column be physically placed above the rows using the lower part of

that column. Two constraints need to be satisfied for a folding to be possible. First, no row should require both inputs sharing the same physical column, although it may require inputs from the upper part of some columns and from the lower part of others. Secondly, the row ordering implied by the folding should not conflict with that implied by other foldings.

The problem of computing folding pairs is equivalent to a graph problem. The PLA without any folding is represented by a graph in which there is a one-one correspondence between the columns of the personality matrix and the vertices of the graph. Undirected edges connect vertices that represent disjoint columns, that is, columns that are not both required by any row. A folding pair can then be represented by a directed edge between the corresponding vertices (from the upper column to the lower column). The folding problem reduces to finding a maximum number of directed edges without creating an alternating cycle in the graph. An example that illustrates SCF and the equivalent graph problem can be found in the previous report [Finkel86b]. The definition of an alternating cycle and other terms used in its detection are given below.

Graph G(V,E,A).

V — set of vertices of the graph.

E — set of undirected edges connecting disjoint columns.

A — set of directed edges between vertices forming folding pairs.

Adj(u)

Adjacency set of vertex u = $\{v \mid (u,v) \in E\}$.

AP (Alternating path)

a sequence of vertices $\pi = [v_1,v_2,...,v_{2k}]$ such that $(v_{2k-1},v_{2k}) \in A$ for $1 \leq k \leq N$ and $\{v_{2k}, v_{2k+1}\} \in E$ for $1 \leq k < N$.

AP(v)

the set of alternating paths beginning at vertex v.

AC (Alternating cycle)

a sequence of vertices $\pi = (v_1,v_2,...,v_{2k})$ if π is an alternating path and $(v_{2k},v_1) \in E$.

A_s the set of start vertices of directed edges = $\{ s \mid$ there exists a t such that $(s,t) \in A\}$.

A_t the set of terminal vertices of directed edges = $\{ t \mid$ there exists a s such that $(s,t) \in A\}$.

Trans(v)

the transitive closure of vertex v = Adj(w) such that $w \in A_t \cap AP(v)$

The detection of an alternating cycle can be done as follows. An alternating cycle is created by adding a directed edge (v,u) if $S_u \cap F_v \neq \phi$, where

$S_u = \{ t \in A_s \mid t \in Adj(u)\}$

$F_v = \{ t \in A_s \mid v \in Trans(t)\}$

8.3. Serial algorithm

It has been shown [Hachtel82] that the SCF problem is NP-complete. A hueristic is described that can be used to achieve near optimal solutions in a worst case complexity of O(N^3, N = | V |) and expected O(M, M = | A |). The heuristic is as follows.

A vertex v of minimum degree, not yet part of any folding pair, is selected as the upper column of a possible folding pair. A vertex u of maximum degree, not part of any folding pair and not adjacent to v, is then found such that the addition of directed edge (v,u) does not cause an alternating cycle in the graph. This is the lower column of the folding pair. If such a pair is discovered, the graph is updated and the pair (v,u) added to the list of foldings. The algorithm terminates when all vertices have been tried as the upper column of a possible folding pair. The pseudo-code for the serial algorithm is given below.

```
process Serial;

const
        Edges = E -- set of edges of graph;
        Vertices = V -- set of vertices of the graph;
var
        Foldings : set of directed edges := {};
        Top : set of vertex := vertices;
        Bottom : set of vertex := vertices;
        Available : set of vertex := vertices; -- vertices not in a folding pair
        u,v : vertex;
begin
        while ( Top <> {}) do
                v = min (Top); -- select vertex of minimal degree
                Bottom := Available - {u | u = v OR u ϵ Adj(v)};
                while ( Bottom <> {}) do
                        u = max(Bottom); -- select vertex of maximal degree
                        if Cycle(v,u) then
                                Bottom -:= {u};
                        else
                                UpdateGraph (v,u);
                                Foldings +:= (v,u);
                                Top -:= {u};
                                Available -:= {v}+{u};
                                exit; -- from inner while loop
                        end;
                end;
                Top -:= {v}; -- deletion from top column set
        end;
end Serial;
```

8.4. Distributed algorithm

The algorithm described above has an inherent serialism in the fact that the folding pairs can be discovered in a fixed order only. However, the possible foldings for a later vertex can be discovered on the basis of an out-of-date state

of the graph. The best folding (on the basis of the hueristic) can then be chosen out of the possibilities when all earlier vertices have been tried and the graph updated. This forms the basis for distributing work.

The distributed algorithm consists of a master-slave structure. The master is the wisdom server. It generates the graph corresponding to the personality matrix and sends it to all the slave processes. Each slave knows its share of vertices to be processed. The vertices are sorted in increasing order of degree and statically assigned so that slave process i solves vertex v if $v \bmod p = i$, where p is the number of slave processes. A slave process finds the possible foldings (based on its copy of the graph) for each of the vertices assigned to it and reports it to the master. In reply it receives, as "wisdom", the set of foldings discovered thus far. It uses this wisdom to update its version of the graph.

The master's job is to accept the possible foldings reported for each vertex in turn (the distribution of vertices is such that this corresponds to each slave in turn) and select the best folding on the basis of the current graph. It sends the set of foldings discovered thus far to the slaves as reply. A slave process terminates when it has reported possible foldings for all vertices assigned to it. The master terminates when it has processed all vertices. The pseudo code for the master and slave processes is given below. A few tuning parameters have been added to the above algorithm to get better timing. These are discussed in section 5.

```
process Master(NumSlaves:integer);

const
        Edges = E -- set of edges of graph;
        Vertices = V -- set of vertices of the graph;
var
        Foldings : set of directed edges := {};
        Top : set of vertex := vertices;
        Bottom : set of vertex := vertices;
        Possible : set of vertex; -- possible foldings reported by slaves
        u,v : vertex;
```

```
begin
        foreach slave in [1..NumSlaves] do
                initialize(slave); -- send the graph to slaves
        end;
        while ( Top <> {}) do
                v = min (Top); -- select vertex of minimal degree
                accept (Possible) from slave;
                reply (Foldings) to slave;
                Bottom := Possible;
                while ( Bottom <> {}) do
                        u := max(Bottom); -- select vertex of maximal degree
                        if (cycle(v,u)) then
                                Bottom -:= {u};
                        else
                                UpdateGraph (v,u);
                                Foldings +:= (v,u);
                                Top -:= {u};
                                exit; -- from inner while loop
                        end;
                end;
                Top -:= {v}; -- deletion from top column set
        end;
end Master;
```

```
process Slave(SlaveID,NumSlaves:integer);

var
        Edges : set of edges;
        Vertices : set of vertex;
        Wisdom : set of directed edges ;
        Folding : directed edge;
        Top : set of vertex ;
        Possible : set of vertex ; -- set of possible folding partners
        Bottom : set of vertex;
        Available : set of vertex; --- set of vertices not part of a folding
        u,v : vertex;
begin
        accept (Vertices,Edges) from master; -- initialization
        Top := { v | v mod NumSlaves := SlaveID};
        while ( Top <> {}) do
                v := min (Top); -- select vertex of minimal degree
                Bottom := Available;
                Possible := {};
                while ( Bottom <> {}) do
                        u := max(Bottom); -- select vertex of maximal degree
                        Bottom -:= {u};
                        if (Cycle(v,u) ) then
                                -- nothing
                        else
                                Possible +:= {u}; -- add to possible
                        end;
                end;
                report (Possible) to master;
                accept (Wisdom) from master;
                foreach new Folding in wisdom do
                        UpdateGraph(Folding);
                end;
                Top -:= {v}; -- deletion from top column set
        end;
end Slave;
```

8.5. Implementation using Lynx

The serial and distributed algorithms were implemented in Lynx. The vertices of the graph were sorted in increasing order of degree at the outset so that the calls to max and min reduce to array accesses. Sorting also helps in the static distribution of vertices to the slave processes in the distributed algorithm. Message passing between the master and slaves is synchronous, in accordance with the nature of the distributed algorithm. A version of the distributed algorithm where the slave processes share wisdom among themselves was also implemented, but speedup was poor due to excessive communication costs.

Measurements on the serial algorithm and an initial straightforward implementation of the distributed algorithm described in section 5 above were taken. Based on these, some tuning parameters were introduced to achieve better speedup. The first tuning concerns the behavior of the serial algorithm. In the

earlier stages of the algorithm, we investigate vertices with smaller degrees (as a possible upper column) in a graph that does not yet have many directed edges. This results in foldings being discovered very quickly (fewer calls to function Cycle are required). As an example, for a 150-column × 300-row PLA, the first 10 vertices required only 0.5 percent of the total time. For this part of the computation, the distributed version (where all possible foldings must be searched by the slaves) will certainly do poorly. To avoid this loss, the master process serially solves a fixed number of vertices before distributing the work among slave processes. This number was found optimal at about 10 percent of the vertices, but that figure most likely depends on the sparsity of the personality matrix.

The second tuning involves the frequency of reports made by the slave processes. The motivation for frequent reports is frequent updates (via wisdom) to the graph in the slave processes. Wisdom is important, since it results in fewer foldings being suggested by the slave processes, which in turn reduces the time required to select a folding in the master process. On the other hand, frequent reports result in extra communication time. Most of the foldings are discovered in the initial stages of the algorithm for the same reason as discussed in the previous paragraph. Therefore, frequent reports are not so important in the later stages of the algorithm. Another factor that needs to be considered is p, the number of slave processes. p affects the best reporting frequency, since a slave gets a turn to report once in p reports to the master. We tuned report frequency based on several measurements. A slave reports in either of the following conditions.

The number of possibilities discovered $\geq$ MAX_LEN,
where MAX_LEN is the maximum number of possible foldings for the vertex that has minimum degree

The number of vertices investigated by any slave $\geq$ MAX_VERTICES,
where MAX_VERTICES at vertex i is $\max(i/50+2, 4/(p/5+1))$.

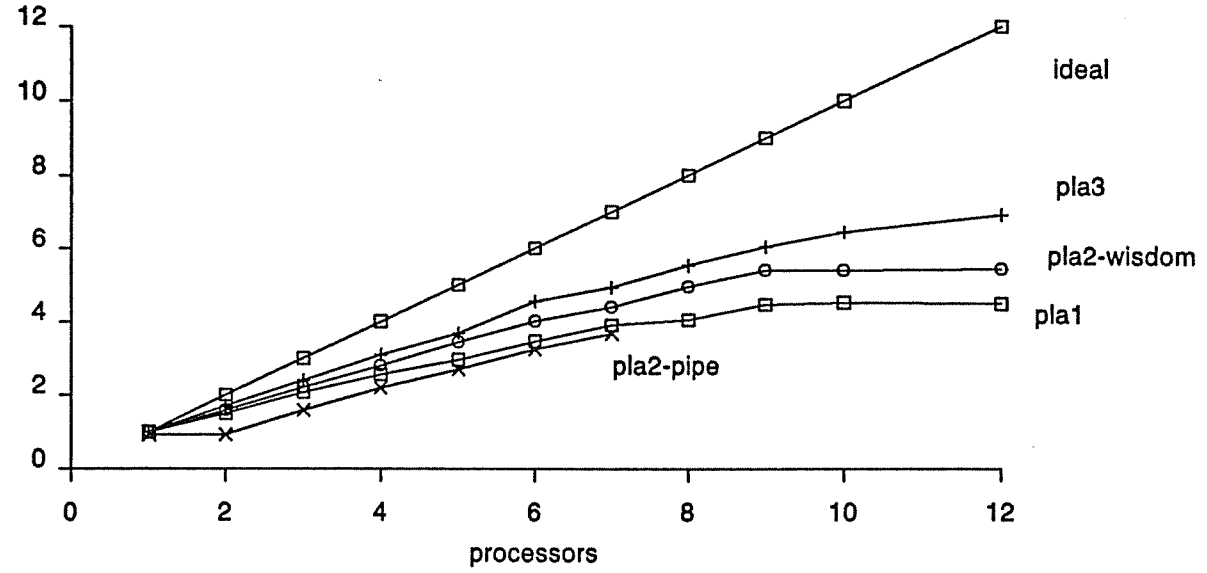
The final tuning concerns the participation of the master process. If the number of slave processes is less than a maximum (depending on the size of the problem) then the master wastes a significant amount of time waiting for reports from slave processes. This time could instead be utilized if the master itself becomes a slave process. The maximum number of slaves before the master stops acting as a slave was set to $p = |V|/50+2$.

8.6. Performance

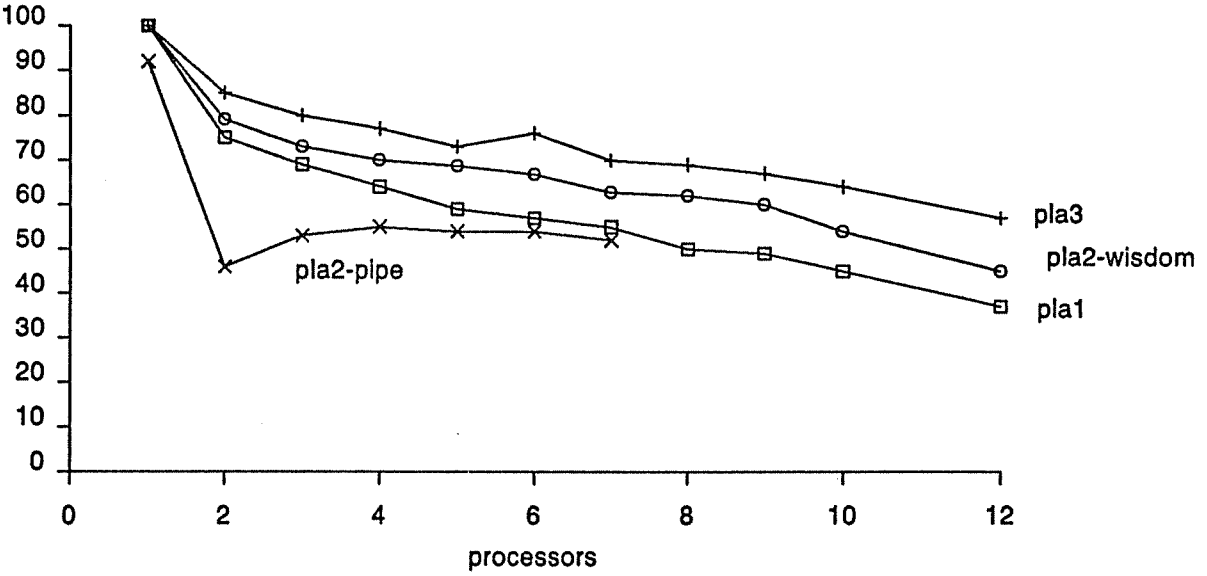
The program was run for several PLAs of varying sizes and personality-matrix sparsity. We report the results obtained from a PLA of size 150 columns × 300 rows (PLA 2) whose personality matrix has an entry of one with a uniform probability of 0.035. This PLA was also the subject of the tests reported in the earlier study. To show the behavior of the program for varying problem

sizes, results from PLAs of size 100 × 300 (PLA 1) and 200 × 300 (PLA 3) were also obtained. In all the runs, only one process (master or slave) is loaded on a processor. The measurements were recorded from the master process and one slave process, which is assumed to represent the average for all slaves.

The speedup curve for different PLAs is shown in Graph 29. Pla2-pipe refers to the results obtained in the pipelined approach. The corresponding efficiency curves are shown in Graph 30. Graph 31 shows the percentage of

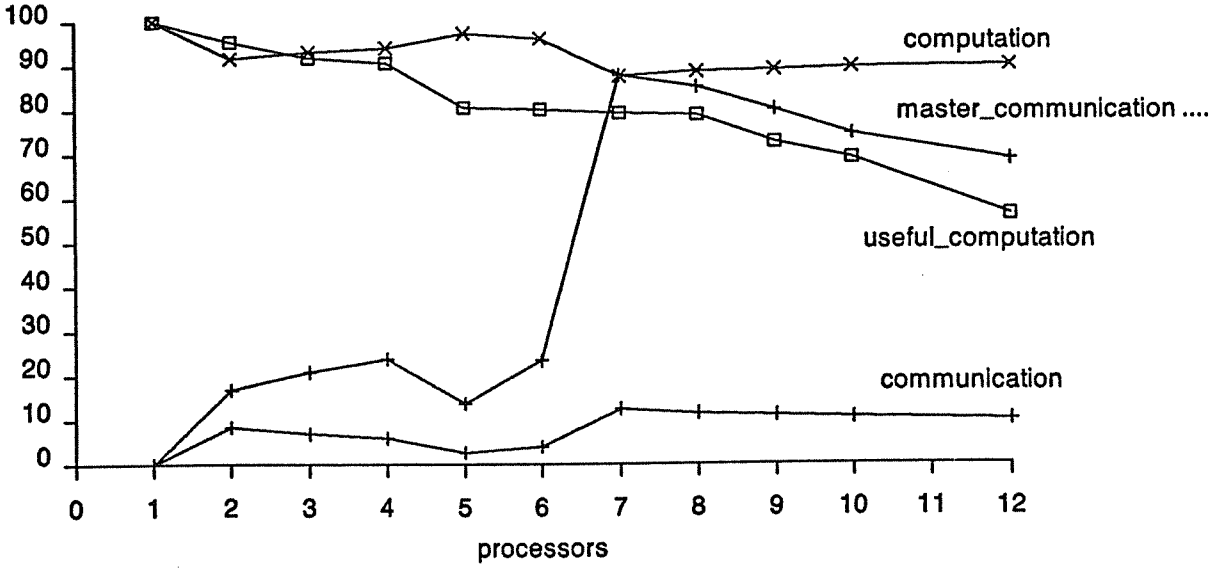


Graph 29. Speedup

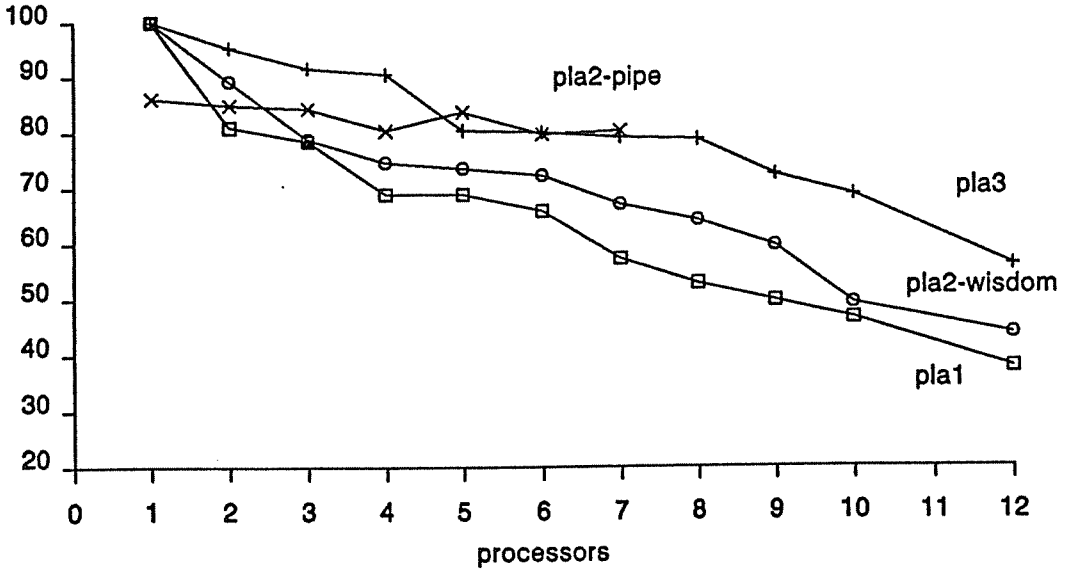


Graph 30. Efficiency

CPU time spent in various part of the algorithm for PLA 3: communication (all processes), actual computation (all processes), communication (master process), and effective computation (serial time over the total computation time). A comparison of the percentage time spent in effective computation for our algorithm versus the pipeline algorithm is shown in Graph 32, and Graph 33 makes the same comparison for time spent in communication.



Graph 31. Percentage of effort



Graph 32. Effective computation

The speedup curve for PLA 2 with the wisdom server approach is linear with a slope of 0.56 and zero offset up to 9 processors. This compares favorably with the pipelined algorithm for PLA 2, which is linear up to 7 processors, with a slope of 0.5 and a negative offset. The main reason for our better performance seems to be lower communication overhead. This can be seen from the two PLA 2 curves in Graph 34. In the pipelined method, each folder process must get a message from every other one, and forward them all. In the wisdom server approach, only the master process has to deal with these messages. The slaves have to send only one message for p messages processed by the master. This number is further reduced by the reporting-frequency tuning factor as discussed in Section 4. A similar tuning in the pipelined approach could be useful.

However, the improvement in performance is not proportional to the difference in communication cost. This can be explained by the measurement obtained for percentage of useful computation. The tuning parameters used to reduce communication costs decrease useful computation, due to infrequent sharing of wisdom (as discussed in section 5). The wisdom server approach thus wastes more time in redundant work than the pipelined approach. This can be seen from Graph 32.

Communication overhead is kept low by the use of tuning parameters. The master has a significant communication cost, partly due to waiting for slave reports. This factor is reduced by letting the master act as a slave as well up to a certain number of processors. The sudden rise in communication cost for the master (Graph 31)) is when the master stops acting as a slave. The slaves have negligible communication cost, which shows that the master is not a bottleneck as far as communication is concerned.

As can be expected, the performance is better for larger problems (PLA 3) than for smaller ones (PLA 1). The main reason for this seems to be the higher percentage of useful computation for larger PLAs rather than lower communication cost. The speedup for PLA 3 is linear up to 9 processors with a slope of 0.65 and zero offset.

The speedup curves in Graph 29 show that no speedup is obtained for more than 9 processors. The reason for this seems to be decreasing percentage of useful computation as wisdom is received infrequently. A similar effect is expected in the pipelined approach, because the amount of wisdom in each folder is same as that in the corresponding slave in the wisdom server approach. The degradation, however, is more pronounced in the wisdom server approach because the work of selecting the folding is centralized in the master. The solution to this problem seems to be a multilevel distribution of work as opposed to a one-level master and slave structure. The master would form the root of a tree and each node in the tree would serve as a slave to its parent and in turn act as the wisdom server for its children. The level of wisdom will be higher at levels closer to the root. Each node will receive possible foldings from its children, prune it using its state of the graph, and report the new set of possibilities

to its parent. It also sends wisdom to its children and receives wisdom from its parent as replies.

8.7. Experience with Lynx

The semantics of remote procedure calls and the **connect** and **accept** statements make it very easy to implement synchronous communication in Lynx. Another useful feature is that structured data types records, arrays and sets can be passed as parameters in remote procedure calls.

The absence of a facility to define an array of links in the header of a process makes it cumbersome to implement a master-slave structure where the number of slave processes is variable. This problem was overcome by connecting the master to only one slave statically. All slave processes are connected together in a circular chain of static links. The link between the master and slave i , $i = 2,3,..p$, is created dynamically from the master by passing a link-end via slave $i-1$.

8.8. Conclusion and future work

We conclude that algorithms with an inherent serialism of the type discussed in this report can be effectively distributed using a wisdom server approach. The results obtained from various runs demonstrate the importance of tuning between message costs and frequency of sharing wisdom. The one-level master-slave structure implemented does not provide speedup for a larger number of processors (> 9). A multilevel extension to the same approach has been suggested as a possible solution.

It would be interesting to implement the multilevel structure suggested in section 7 and compare its performance to the one-level structure presented here. A more challenging task is the formalization of the wisdom server approach into a distributed computing system like DIB [Finkel86d]. That would involve devising a more general and dynamic method of distributing work as compared to the static distribution presented in this report.

9. Quicksort

Experimenter: Sriram Vajapeyam

9.1. Introduction

Quicksort is probably the most popular sorting algorithm in computer applications. Here we describe one way to distribute Quicksort in a multicomputer. Quicksort is a divide-and-conquer algorithm. It splits the set of elements to be sorted in two, sorts them independently, and combines them back together. Quicksort is recursive. These characteristics favor distributing the algorithm over a number of processes. However, the amount of data motion may prevent good speedup.

9.2. Quicksort

The way Quicksort works is very simple. Let us assume that we are sorting a set of numbers. The algorithm chooses some **partitioning element** of the set and shuffles the elements around in such a way that all elements to the left of the chosen element are less than the partitioning element. All elements to its right are larger. The original set has now been partitioned into two subsets. These sets are treated recursively.

Partitioning works as follows. Two pointers, i and j, are initially set to point to the leftmost and rightmost elements of the set. The pointer i is moved towards the center until it encounters an element bigger than the partitioning element. Similarly, j is moved towards the center until it encounters an element smaller than the partitioning element. The two elements pointed to by i and j are now exchanged. This process is repeated until i and j meet or cross each other.

If the partitioning element happens to be the smallest or largest element of the subset, then partitioning produces an empty subset. This means that the order of the original problem is reduced by only one. In this case, Quicksort achieves its worst running time. On average, Quicksort is an $n\log_2 n$ algorithm, but its running time becomes proportional to n^2 in the worst case.

To decrease the chances of the worst case occurring, we use the median-of-three method [Sedgewick83], in which the partitioning element is chosen to be the median of three random elements of the subset (in our case, the first, middle and last elements).

The overhead involved in partitioning is such that it is worthwhile to use Quicksort only for large data sets. We use linear insertion to sort all subsets smaller than a particular size. This particular size could be anything between 5 and 12 for optimum performance [Evans82]. We set the cut-off point at 8 elements for a subset.

The pseudo-code for the serial algorithm is shown here:

```
var
        elements : array [1..N] of integer;

procedure Partition(set, set1, set2);
var
        PartitionElement : integer; -- the partitioning element.
        i, j : integer;  -- the two pointers.
        Done : Boolean;

begin
        PartitionElement := MedianOfThree(set1);
        i := set1.left;
        j := set1.right;
        loop
                repeat
                        i +:= 1
                until (element[i] > PartitionElement);
                repeat
                        j -:= 1
                until (element[j] < PartitionElement);
                if (i >= j) then exit; end;
                exchange elements[i], elements[j];
        end;
end Partition;

procedure QuickSort(set);
begin
        Partition(set, set1, set2);
        for s := set1, set2 do
                if ¦ s ¦ < M then
                        LinearInsertionSort(s)
                else
                        QuickSort(s);
                end;
        end;
end QuickSort;
```

9.3. Distributed algorithm

Quicksort produces a tree of subsets that can be sorted independently. One straightforward way to distribute the algorithm would be to partition the tree into p subtrees, each of which would be given to one process to sort by the serial algorithm, and a root region, which could be distributed as well. This tree could be built dynamically, with process i entering the tree at level $\lfloor \log i \rfloor$, as shown in Figure 9. This method has been analyzed by Evans and Dunbar [Evans82]. Unless the number of subtrees is at least the number of processes, there will be idle processes in the system. The time needed to generate and send a subset to every process is the **startup** time. When the number of subsets is small, each is large, and the work to be done in partitioning is large. Process 1 must partition a set of n elements, whereas process 2 has only $n/2$ to work on.

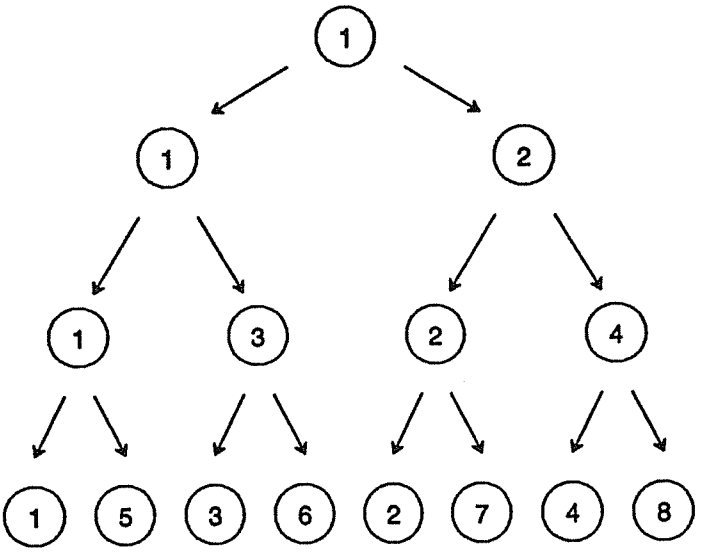


Figure 9. Process allocation

For very large values of n and p, the startup time might be quite large. We try to distribute the work in such a way that the startup time is removed. A master process assigns subsets to processes and gives each process its (unsorted) subset. The master then selects a partitioning element for the whole set (using the median-of-three method) and sends it to each process. Every process partitions its subset using this common partitioning element and reports back to the master the number of elements it has on the left and right of the partitioning element. From this information, the master determines the sizes of the two global subsets and therefore how many processes should deal with each. Each process is told whether it is to work on the lower or higher subset, and it returns to the master those elements that it is not to work on. The master redistributes returned elements to the appropriate processes and also supplies a partitioning element for each of the two sets. As soon a process gets its full complement of data, it starts the next round of partitioning. We use the same master to oversee each round of partitioning and data shuffling. When partitions reach size n/p, they reside on single machines, which proceed to sort them by the serial algorithm. As each serial part finishes, the results are sent back to the master, which knows the position of those results in the global answer.

The pseudo-code for the distributed algorithm is shown here:

```
process Master(proc1, proc2, .. , procP : link);

const
        N : -- the size of the data set.
        P : -- the number of processes.
        MSG_SZ : -- the maximum size of an array parameter
type
        DataArray = array [1..MSG_SZ] of integer;
        subset_type =
                record
                        l, r : integer; -- limits of the subset.
                        PartitionElement : integer;
                        Workers : process list;
                        numleft, numright : integer;
                end;
        process_type =
                record
                        subset : set; -- subset it is working on.
                        numleft, numright : integer;
                        nxtproc : ^process; -- next process working on the same subset.
                end;
var
        DATA : array [1..N] of integer; -- the set to be sorted.
        Data : array [1..MSG_SZ] of integer; -- the array to be passed in messages.
        SET : array [1..P] of subset_type;
        processes : array [1..P] of process_type;

procedure median(SET);
begin
        sorts the first, middle and last elements of SET and returns the middle one.
end;

procedure GetData(proc : integer);
-- collects data of one subset from process 'proc'
-- and puts it in the global array of elements.
begin
        connect SendSortedData(|elements) on proclink[proc];
end;

procedure send_data(Data : DataArray; proc : integer);
-- sends the data in the Data array to process proc.
begin
        connect ReceiveData on proclink[proc];
end;
```

```
procedure DistributeData(subset, n_per_proc : integer);
begin
        MedianOfThree(subset); -- find a partitioning element for the subset.
        foreach process in subset's process-pool do
                determine the number of elements the process should get.
                repeat
                        copy part of the elements to be sent into DataArray;
                        send DataArray to the process;
                until (all elements are sent to the process);
                if the subset has only one process then
                        inform the subset so.
                end;
        end;
end DistributeData;

procedure AcceptData(subset);
begin
        if subset had only one process working on it then
                GetData(subset);
                if all other processes are done then
                        print sorted data;
                        print_statistics;
                        exit;
                end;
        else
                determine the sizes of the subsets given rise to by subset;
                determine the number of processes to be given to each new subset;
                initialize two new subsets;
                discard old subset;
                foreach process in subset's process-pool do
                        GetData(process); -- collect elements of the subset the process has
                                -- not been assigned to, from the process.
                        initialize the process to work on its new subset.
                end;
                DistributeData(NewSubset1);
                DistributeData(NewSubset2);
end AcceptData;

entry AcceptInfo(process, numonleft, numonright);
var
        subset : integer;
begin
        store information regarding the number of elements in the process belonging
        to each of the two new subsets.
        subset := subset the process was working on;
        if all the processes working on the subset have given info then
                position the partitioning element of this subset in its appropriate
                        place in the whole set of data.
                AcceptData(subset);
        end;
end;
```

```
procedure Initialize;
begin
        initialize variables;
        ReadIn (set to be sorted);
        MedianOfThree(SET);  -- find a partitioning element for the whole set.
        distribute the set equally to all the processes;
        send the partitioning element also to each process;
        bind each link to AcceptInfo;
end;

-- Master
begin
        Initialize;
end Master.

process Worker(id : integer; masterlink : link);

const
        MSG_SZ: integer -- size of arrays passed in messages.
        EL_SZ: integer -- size of local array holding elements for sorting.
var
        elements : array [1..EL_SZ] of integer;
        Data : array [1..MSG_SZ] of integer;

entry SendSortedData;
begin
        repeat
                copy some of elements belonging to appropriate subset into Data;
                send Data to master;
        until (all elements of appropriate subset are sent to master);
end;

entry ReceiveData;
begin
        copy Data into elements[];
        store the partitioning element;
        if time to partition then
                Partition;
                connect AcceptInfo on masterlink;
        elsif time to sort then
                Quicksort;
                connect AcceptInfo on masterlink;
        end;
end;

begin -- Worker
        initialize variables;
        bind masterlink to ReceiveData;
        bind masterlink to SendSortedData;
end Worker.
```

9.4. Implementation in Lynx

The implementation in Lynx is structured as shown in the previous section. There are two modules: **master** and **worker**. The master is loaded on one machine, and a worker on each other machine. The master distributes the initial set to all the processes, helps distribute work until sorting is done, collects the sorted data, and prints statistics. A worker partitions elements, sorts sets, and sends results back to the master.

The master is quite busy with remote procedure calls. Initially all the subroutines in the master were implemented as local entries, with the idea that if one thread were to get blocked on a remote procedure call some other thread could continue. It was hoped that this would improve the running time of the algorithm. But, as a result of the large number of threads, we ran into stack problems. So all the local entries were changed to procedures. This helped us increase the set size from 3000 to 6000 elements.

The real restriction was on the parameter sizes allowed for RPCs. The maximum size of arrays that could be passed as parameters in RPCs was 3000 bytes. As n and p increased, the array sizes had to be reduced. We experimented with 1500 and 2500 bytes for array sizes. Also, we restricted elements to one byte to mitigate memory restrictions.

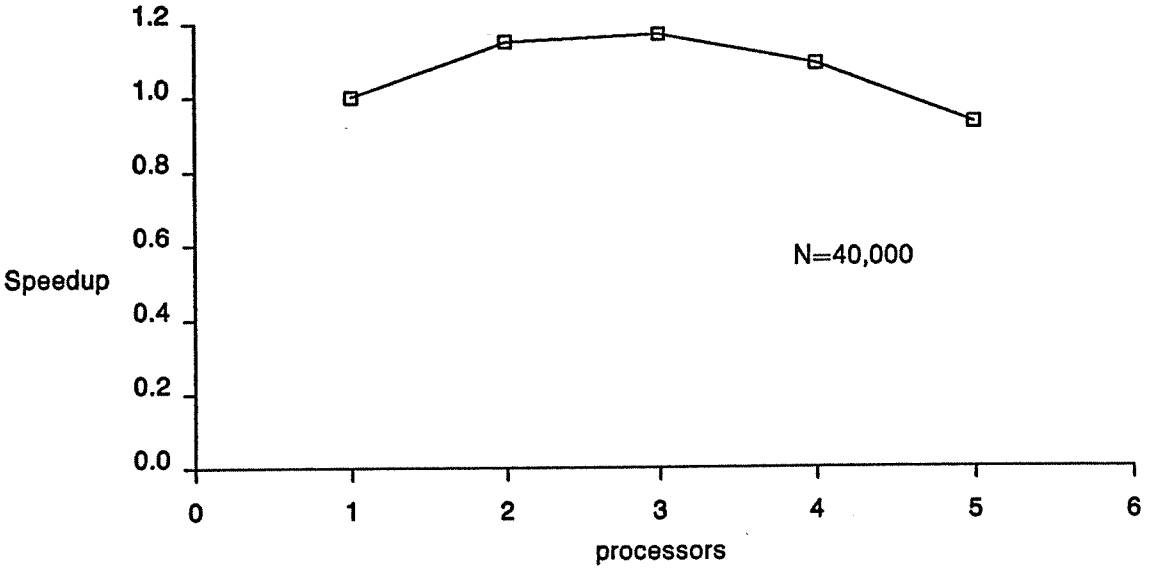
The restriction on parameter sizes affects the algorithm's runtime, because we have a number of RPCs in place of one call whenever sending/receiving data. We could say that the program is making one virtual RPC each time it tries to send/receive data, but actually a number of actual RPCs are needed.

9.5. Results

The program was run on Crystal. Experiments were conducted with up to 5 processes. The master and the processes were loaded on separate machines. Each of the processes were timed from the moment it started on its first set of data till the time it finished sorting its last independent subset fully. The longest time among all the processes was taken to be the time of the distributed algorithm. The time taken to send the final sorted data to the master was not taken into account.

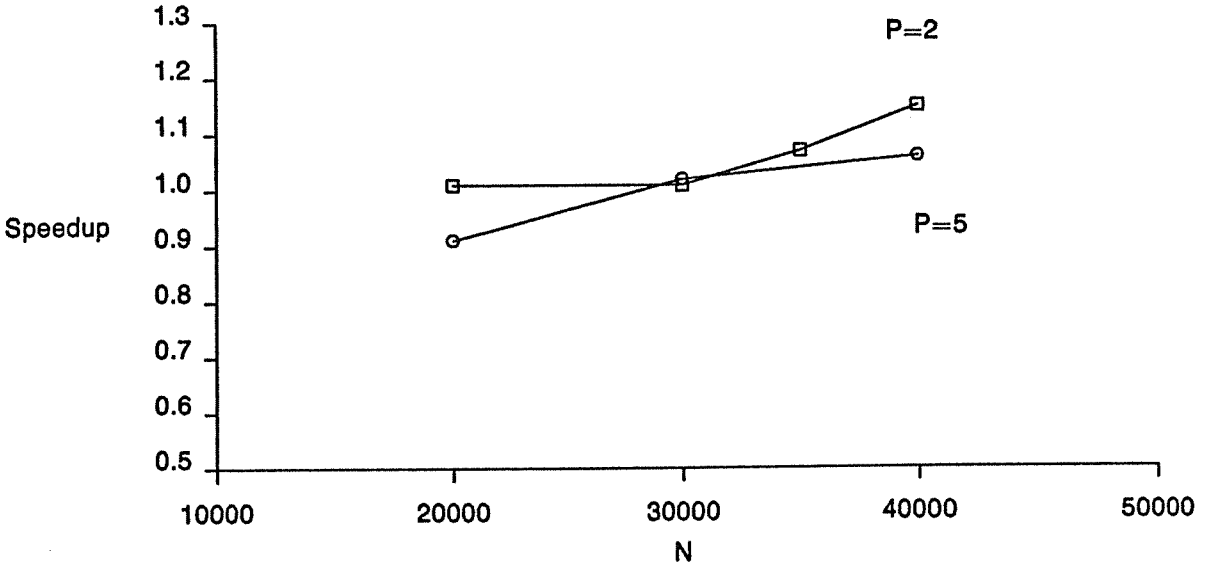
The serial program is very fast, so the cost involved in distributing the algorithm can be overcome only for large n. The serial algorithm sorted 1000 numbers in 0.0646 seconds. It took approximately 2.43 seconds to sort 40,000 numbers.

We were not able to enter the region where distribution pays off. For small n, the distributed algorithm doesn't run faster because it isn't worthwhile trying to remove the startup time. The overall speedup itself is nominal for the cases we have been able to measure. Graph 35 shows the speedup for n=40,000, where message size was set to 1500. As n increases, so does the speedup, as shown in Graph 36. In this graph, message size was 2500 for p=2

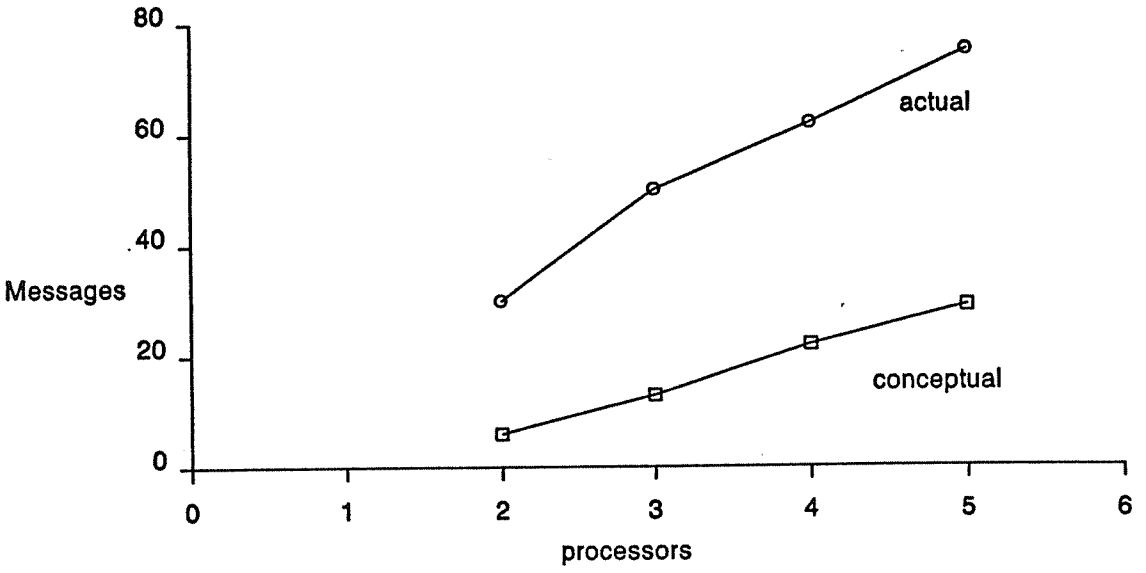


Graph 35. Speedup, n=40000

and 1500 for p=5. The reason for the drop in speedup as p increases is the number of messages, as shown in Graph 37, where n=40,000 and message size is 1500.



Graph 36. Speedup versus n.



Graph 37. Messages

9.6. Experience with Lynx

- The synchronization between processes was easy with statements like connect, and await.
- The stack space provided could be increased to benefit most programs. Useful hints to the user when stack space runs out as to which entries/subroutines could be causing the problem would save the user time and effort.
- It would be good if the policy that chooses which thread to resume were visible. Resuming threads in different orders can affect the logic of a program not written very carefully.

9.7. Conclusions and some future possibilities

Quicksort is an algorithm well suited for distribution over a number of processes. The algorithm is also very fast by itself, which makes distributing it worthwhile only when sorting very large sets of data. Our method is an enhancement of previous methods; we try to reduce the cost of the startup phase. A reasonable direction would be to reduce the idle time by some load balancing strategy, perhaps modelled on that used in DIB [Finkel86d].

10. Solving for roots of f(x) = 0

Experimenter: Dhruva Ghoshal

10.1. Introduction

Our goal was to implement a distributed algorithm to find a root of $f(x)=0$ within the interval $[a,b]$, where f is continuous, and $f(a)f(b) < 0$. We assume that evaluating f is very expensive, and that the cost might vary depending on x. If it is cheap to evaluate f, a straightforward binary search of $[a,b]$ is likely to beat any distributed algorithm.

The basic approach in calculating a root is to compute values of f at various points within the interval and compare the values at the points with the value of f at the end of the interval. For example, if $f(x)f(a)<0$, then there is a root in $[a,x]$; otherwise, there is a root in $[x,b]$. The size of the interval can be reduced by knowing the value of the function at an intermediate point. This process is continued until the reduced interval is small enough to fit the desired accuracy. We formalize this method in the following sections.

10.2. Serial algorithm

The serial algorithm computes the value of the function at the midpoint of the interval in which the root is currently expected, so at each stage, the interval is reduced by half. The time taken by the algorithm is

$$T \times \log_2 \left(\frac{b-a}{err} \right)$$

where T = expected time to calculate the value of a function at a point
err = accuracy desired

Here is a description of the serial algorithm in pseudo-code.

```
process serial;

type
        interval =
                record
                        a     : float ;
                        b     : float ;
                        ValAtA : float;
                end ;

procedure ReduceInterval(current : interval; err : float);
var
        point      : float ;
        val        : float ;
begin
        while (length(current) > err) do
                point := (current.a + current.b)/2;
                val := f(point);
                with current do
                        if (ValAtA * val < 0) then
                                b := point;
                        elsif (ValAtA * val > 0) then
                                a := point;
                                ValAtA := val;
                        else
                                a := point;
                                b := point;
                        end;
                end;
        end;
end ReduceInterval;

end serial;
```

10.3. Distributed algorithm

A master process allocates various x-values to individual calculator processes, which calculate the values of the function at those points and return the values back to the master. The master keeps a list that shows the x-values in which individual calculating processes are working. Suppose each process i is working at a point x_i, and that the points x_i are in increasing order. Assume i reports that the interval can be reduced to $[x_i,b]$. The work being done by processes 1 to $i-1$ becomes unnecessary, so the master can give them new x-values. A similar action takes place if the interval is reduced to $[a,x_i]$.

This distribution was first suggested by Eriksen and Staunstrup [Eriksen83]. Their scheme was to allocate x-values in a certain ratio. Whenever x-values were to be allocated, say in [0,1], processors were allocated x-values α, α^2, $\cdots$, where $\alpha<1$. A simple transformation adapts this allocation to any interval. The initial allocation of x-values covers the whole interval $[a,b]$. When the interval is reduced to $[a,x_i]$, all calculations to the right of x_i are

interrupted, and the associated processors are given new allocations within $[a,x_1]$. Similarly, when the interval is reduced to $[x_i,b]$, calculations to the left of x_i are interrupted, and the associated processors are given new allocations within $[x_p,b]$. Their results show that for p processors, $\alpha=1-\alpha^p$ gives the best results. However, this scheme did not produce any speedup in our tests.

We implemented a different scheme for allocating x-values. We always give the midpoint of the currently longest useful subiterval whenever a processor must be reallocated. Although we obtained limited speedups, we got much better results than for the method suggested by Eriksen and Staunstrup.

Eriksen and Staunstrup implemented their scheme for shared-memory processors where communication is very cheap, and therefore the expense of the allocation mechanism is more significant. In our environment, reducing the number of messages is the most important goal.

Here is our pseudo-code.

```
process Master(calc_1, ..., calc_p link);

const
        MAX_LINK = 15 ;
        MAX_INT  = 1000 ;

type
        LinkArray = array [ 1 .. MAX_LINK ] of link ;
        direction = (left,right) ;
        interval = -- as before
        ProcessRecord =
                record
                        id    :  integer ;
                        point :  float ;
                        new   :  Boolean ;
                end ;
        OrderArray = array [ 1 .. MAX_LINK ] of ProcessRecord ;
        IntervalDescription   =
                record
                        lower  :    float ;
                        higher :    float ;
                end ;
        Intervals  =  array[1 .. MAX_INT] of IntervalDescription ;
```

```
procedure GetPoints (trim : direction; position : integer;
        var start,add_aft : integer;
        var ord1,local_ord : OrderArray;
        var int1 : Intervals);
begin
        case trim of
        { left }
                assign midpoint of currently largest interval to all
                processes to the left and for the process indicated by position;
                -- the currently longest interval changes after every assignment
        { right }
                assign midpoint of currently largest interval to all
                processes to the right and for the process indicated by position;
        end;
        sort list of sub-intervals available in decreasing size;
        sort ord1 according to position where process is working;
end GetPoints;
```

```
entry ReceiveUpdate ;
var
        inter    : interval ;
        trim     : direction ;
        position : integer ;
        last     : float ;
        solution : float ;
        local_ord : OrderArray ;
        j        : integer ;
begin
        reply;
        if (( x ≥ current.a ) and ( x ≤ current.b )) then
                inter := current;
                with inter do
                        if ( hx * ValAtA < 0 ) then
                                b := x;
                                trim := right;
                        elsif ( hx * ValAtA > 0 ) then
                                a := x;
                                ValAtA := hx;
                                trim := left;
                        else
                                current.a := x;
                                current.b := x;
                        end;
                        if (length(inter) < length(current)) then
                                current := inter;
                        end;
                end;
                finish := (length(current) ≤ err);
                if (not finish) then
                        position := 1;
                        while (ord1[position].id <> id) do
                                position +:= 1;
                        end;
                        GetPoints(trim,position,start,add_aft,ord1,local_ord,int1);
                        foreach i in [1 .. N] do
                                if ((local_ord[i].new) and (local_ord[i].point ≥ current.a)
                                                and (local_ord[i].point ≤ current.b)) then
                                        connect f(local_ord[i].point,MAX_WAIT,CALL_AFT ¦ )
                                                on calcs[ local_ord[i].id ];
                                end;
                        end;
                else -- (finish = true)
                        solution := (current.a + current.b)/2;
                        Print(roots);
                end;
        end;
end ReceiveUpdate;

begin   -- master
        finish := false;
        initialize;
```

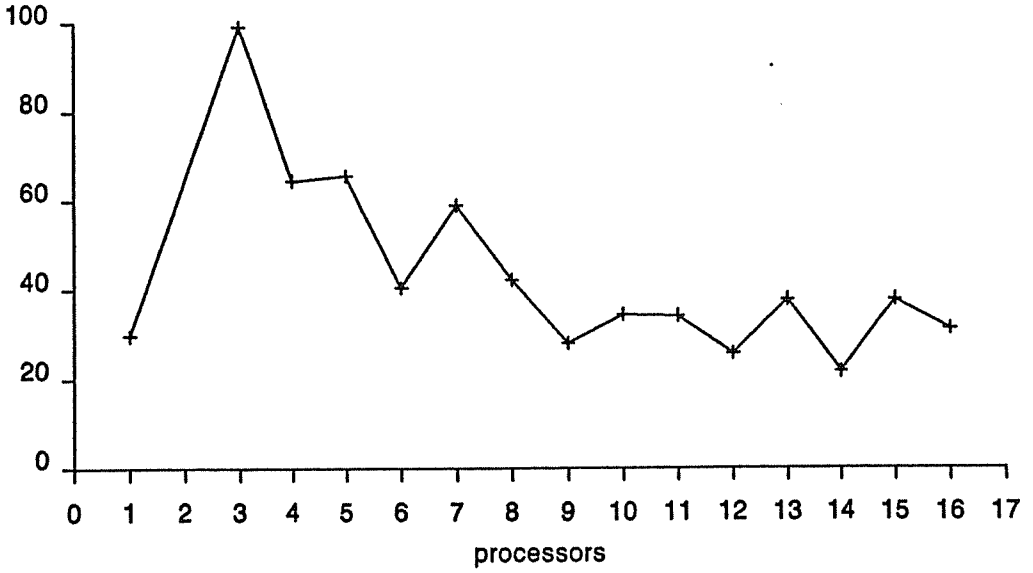
```
end Master.

process Calculator(id : integer; masterlink : link);

entry Evaluate ( point : float) : float;
begin
        reply f(point);
end;

begin       -- calculator
        bind masterlink to Evaluate;
end Calculator.
```

In order to preempt a calculator whose work was rendered worthless by a result from another calculator, we had each calculator periodically allow its Evaluate entry to be invoked by the master. The only way to do this in Lynx is to have the current thread connect to a dummy entry somewhere. This method is costly but works.

10.4. Results

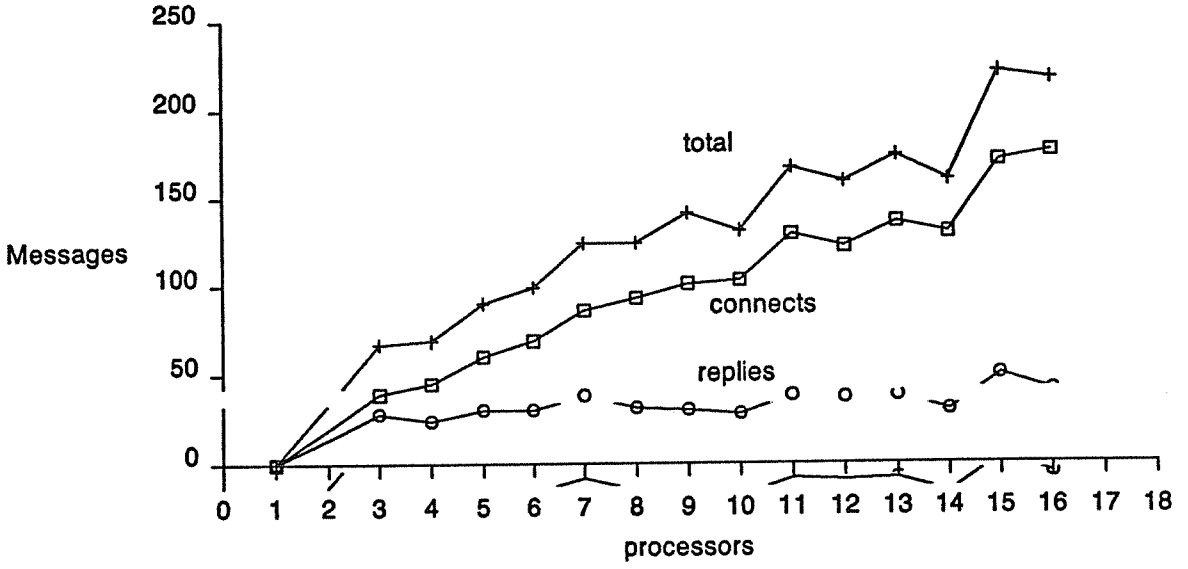
Instead of inventing a complicated function f, we simulate the same effect by looping a random amount of time during each evaluation of the function x^3-x^2-x-2. The random delay was chosen uniformly between 0 and some maximum value.

Graph 38 shows the total time taken as a function of p. It reports on a single run for each number of machines. There was little speedup in the few places any at all was found. The fluctuations can be explained by the random

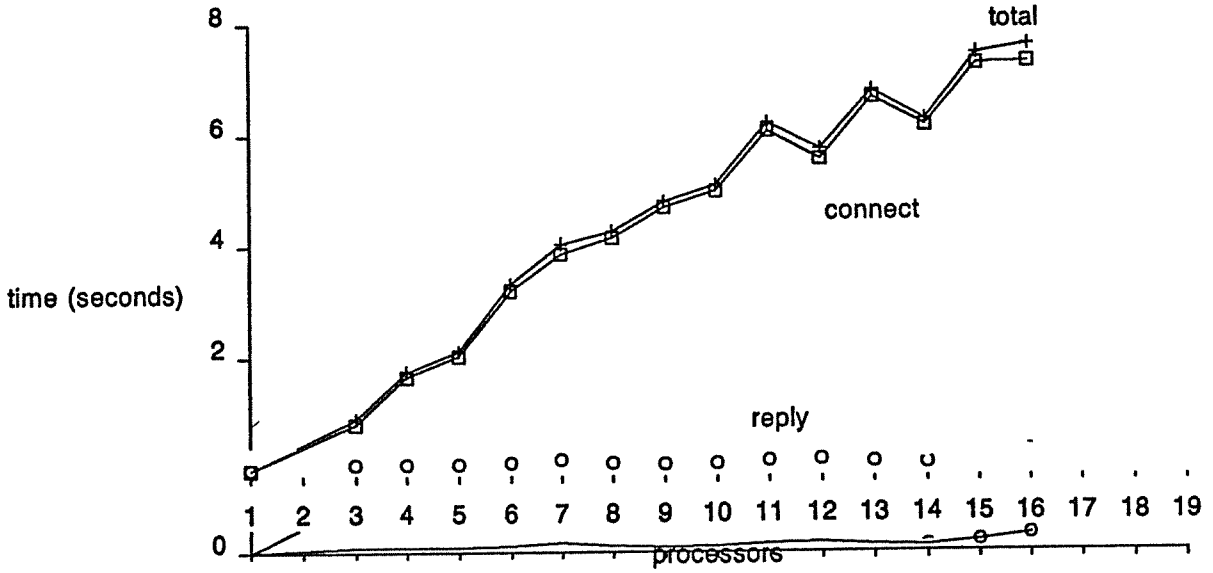


Graph 38. Total time

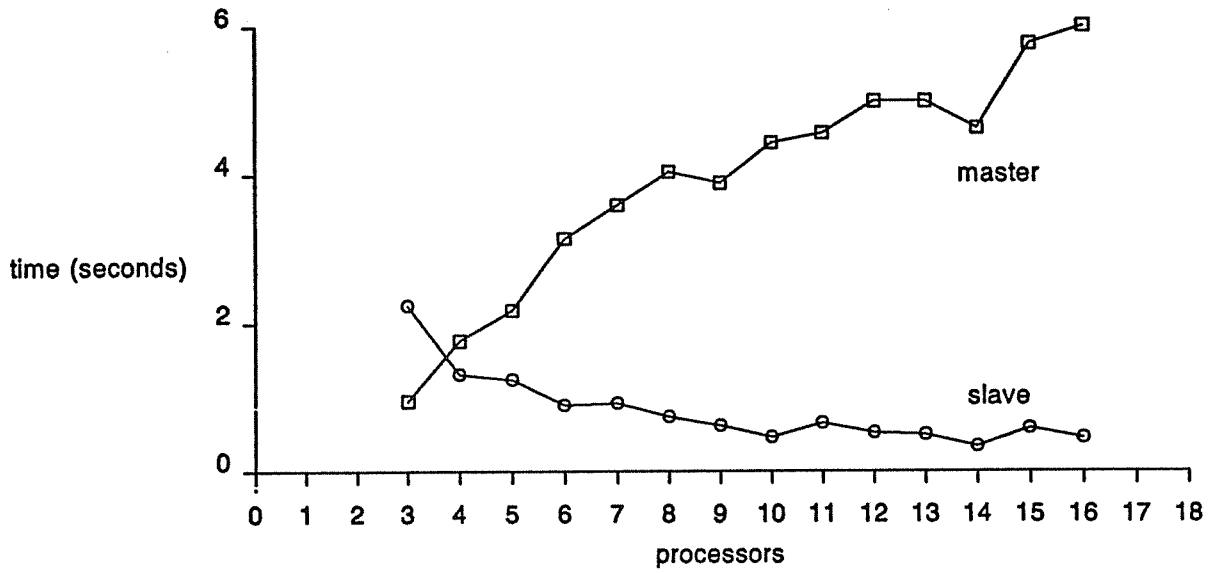
nature of the time needed to evaluate f. Graph 39 shows the number of messages that were sent, both connects and replies. This figure increases with increasing p. The amount of time spent in message passing is shown in Graph 40. The amount of time spent in calculation is shown in Graph 41. The average time spent idle is shown in Graph 42.



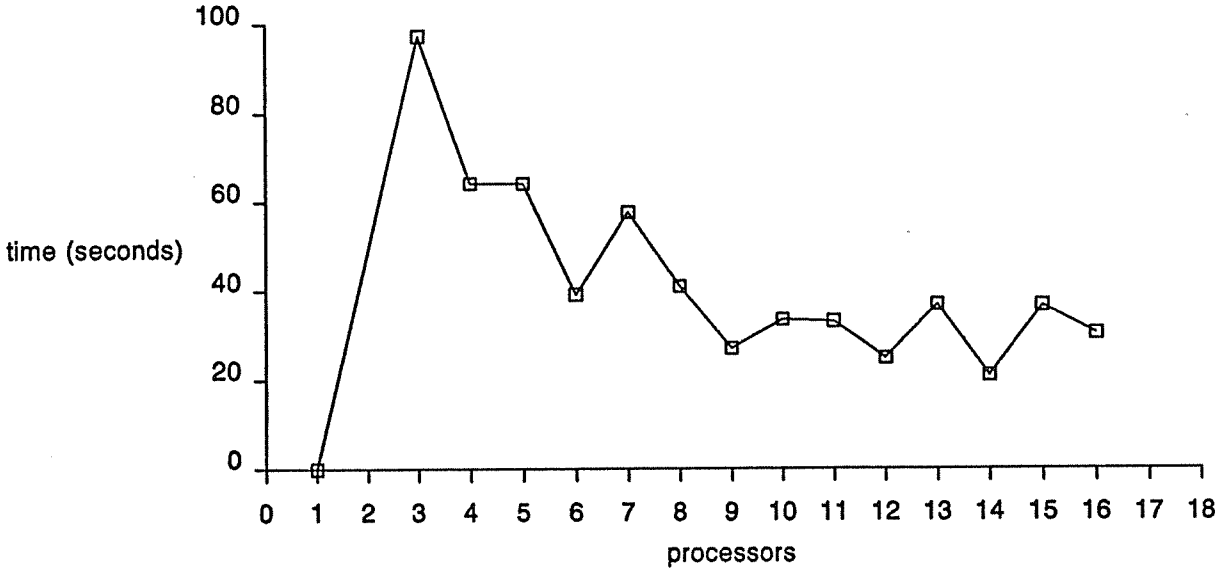
Graph 39. Messages



Graph 40. Communication time



Graph 41. Calculation time



Graph 42. Idle time

10.5. Experience with Lynx

- The lack of any way of aborting an entry in a straightforward manner caused some difficulty; we had to resort to a connect to a dummy entry.
- Allowing processes to have more than 16 links (as at present) might help. Running more than one calculating process per node for 6 nodes showed speedup, but this could not be attempted for more than 8 nodes.

10.6. Conclusions

Since the serial version has logarithmic cost, it is very difficult to get any significant speedup with a distributed algorithm. In general it does not seem to be a good idea to distribute algorithms of logarithmic complexity. This holds especially if there is no definite way of distributing work and if many intermediate communications between processes has to occur.

10.7. Acknowledgements

Udi Manber suggested the idea of this program. Thomas Virgilio helped with Lynx and other details.

11. References

Artsy86.
Artsy, Y., H-Y Chang, and R. Finkel, "Interprocess communication in Charlotte," *IEEE Software*, (July 1986). Accepted subject to revision

Ballard82.
Ballard, D. and C. Brown, *Computer Vision.* 1982.

Chang79.
Chang, E., "An introduction to echo algorithms," *Proc. 1st International Conference on Distributed Computers*, pp. 193-198 (October 1979).

Cheriton79.
Cheriton, D. R., M. A. Malcolm, L. S. Melen, and G. R. Sager, "Thoth, a portable real-time operating system," *CACM* **22**(2) pp. 105-115 (February 1979).

DeWitt84.
DeWitt, D., R. Finkel, and M. Solomon, "The Crystal multicomputer: Design and implementation experience," Technical Report 553, University of Wisconsin—Madison Computer Sciences (September 1984). To appear, IEEE Transactions on Software Engineering

Eriksen83.
Eriksen, O. and J. Staunstrup, "Concurrent algorithms for root searching," *Acta Informatica* **18** pp. 361-376 (1983).

Evans82.
Evans, D. J. and R. C. Dunbar, "The parallel quicksort algorithm, parts I and II," *International Journal of Computer Mathematics*, pp. 19-55 (1982).

Finkel86d.
Finkel, R. and U. Manber, "DIB — A distributed implementation of backtracking," *ACM TOPLAS*, (1986). (to appear)

Finkel86.
Finkel, R. A., A. P. Anantharaman, S. Dasgupta, T. S. Goradia, P. Kaikini, C-P Ng, M. Subbarao, G. A. Venkatesh, S. Verma, and K. A. Vora, "Experience with Crystal, Charlotte, and Lynx," Computer Sciences Technical Report #630, University of Wisconsin—Madison (February 1986).

Finkel86b.
Finkel, R. A., B. Barzideh, C. W. Bhide, M-O Lam, D. Nelson, R. Polisetty, S. Rajaraman, I. Steinberg, and G. A. Venakatesh, "Experience with Crystal, Charlotte, and Lynx: Second Report," Computer Sciences Technical Report #649, University of Wisconsin—Madison (July 1986).

Finkel86c.
Finkel, R. A., *An Operating Systems Vade Mecum,* Prentice-Hall, Englewood Cliffs (1986).

Gordon84.
Gordon, A. and R. Finkel, "A pipeline algorithm for Gaussian elimination," Unpublished memo, Computer Sciences Department, University of Wisconsin (1984).

Hachtel82.
Hachtel, et al., "An algorithm for optimal PLA folding," *IEEE Transactions on Computer-Aided Design of Integreated Circuits and Systems* **CAD-1**(2)(April 1982).

Huertas85.
Huertas, A. and G. Medrioni, "Edge detection with sub pixel precision," *Third IEEE Workshop on computer vision: Representation and Control*, (1985).

Imase81.
Imase, M. and M. Itoh, "Design to minimize diameter on building-block network," *IEEE Transactions on Computers* **C—30**(6) pp. 439-442 (June 1981).

Kepecs84.
Kepecs, J. H. and M. H. Solomon, "SODA: A simplified operating system for distributed applications," *Third Annual ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, (Aug 27-29, 1984).

Madduri85.
Madduri, H. H., *The Banker's algorithm for distributed operating systems*, University of Wisconsin—Madison (May 1985). PhD Thesis

Marr82.
Marr, D., *Vision*. 1982.

Nevatia78.
Nevatia, R. and R. Babu, "Linear feature extraction," *Proceedings of the ARPA Image Understanding Workshop*, Carnegie-Mellon University, (November 1978).

Raghuram86.
Raghuram, S., *A distributed algorithm for computing minimal dominating sets of a graph*, Computer Science Department, University of Massaschussets, Amherst (1986). personal communication

Rosenfeld81.
Rosenfeld, A. and A. Kak, *Digital Picture Processing, Volume 2*. 1981.

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

Sedgewick83.
Sedgewick, R., *Algorithms*, Addison-Wesley (1983).

Wirth76.
Wirth, N., *Algorithms + Data Structures = Programs,* Prentice-Hall (1976).