LOAD BALANCING HEURISTICS AND NETWORK TOPOLOGIES FOR DISTRIBUTED EVALUATION OF PROLOG

Douglas Pase

Oregon Graduate Center
19600 N.W. von Neumann Drive
Beaverton, OR 97006

Technical Report No. CS/E 87-005
April 1987

Load Balancing Heuristics and Network Topologies for Distributed Evaluation of Prolog

Douglas Pase
Dept. of Computer Science and Engineering
Oregon Graduate Center
19600 N.W. Von Neumann Drive
Beaverton, OR 97006
(503) 690-1121 x7303

Peter Borgwardt
Imaging Research Laboratory
Tektronix, Inc.
P.O. Box 500, M.S. 50-662
Beaverton, OR 97077
(503) 627-1118

Abstract

Load balancing and network topology are important issues to the performance of a distributed (message-passing) OR-parallel logic programming system. We discuss the merits of different load balancing schemes and network topologies and their effects on the distributed logic system. We set up a system for evaluating various heuristics in load balancing and for evaluating the efficacy of various network topologies. A non-pre-emptive supply-driven pressure model and the star-graph network will provide an effective execution method for OR-parallel Prolog.

1. Introduction

The need for ever faster logic computing systems continues to grow. Logic programming languages themselves offer a potential solution to this problem through OR-parallelism [1,2]. Parallel logic programs, in fact, must limit or control the parallelism available to prevent the system from being overloaded by a wealth of tasks of varying importance. It is also desirable to control parallelism when an ordering is required for solutions obtained from separate clauses.

Distributed systems (message-passing systems, or systems without shared memory) have a larger upper bound on the number of supportable processors than shared memory systems. They also have a much lower cost per processor. With such systems, two of the most important

This research was partially supported under National Science Foundation Grant No. ECS-8400758 at the Oregon Graduate Center where the second author is an adjunct professor.

issues are the distribution of tasks, or ***load balancing***, and the shape of the interconnection network, or ***network topology***.

Load balancing is a method of assigning tasks to processors. The goal of load balancing is to spread the tasks among the processors in such a way that all are kept busy. A further goal is to minimize the overall execution time required to solve a problem. The load balancing algorithm may depend on *a priori* knowledge, or it may base its decisions on the current state of the system. *Static* load balancing depends completely on information derived prior to the execution of the problem [3]; no system load or state information is taken into account. Optimal static mapping of tasks to processors in a distributed network is known to be NP-complete, and is very difficult even for simple problems. It requires *a priori* knowledge of the resource requirements (CPU time, memory, etc.) for each task, and only merits the expense for very stable programs with a small number of tasks. A large class of numerical problems, such as matrix operations and FFTs fall into this category [4], but the execution patterns of logic programs vary a great deal. This makes any form of static task-to-processor mapping unsuitable for their execution.

Near optimal load balancing can be obtained with much less expense by binding the tasks to processors at the time the tasks are created. This allows the use of system information, such as the current system load to assist in task distribution. This is called *dynamic* load balancing. A load balancing algorithm is *centralized* if there is a single task server which receives and distributes all tasks [5]. Centralized algorithms provide uniform distribution of tasks over the system if the tasks are long lived. Such algorithms are expensive in terms of the time required to bind tasks, since the round-trip time to communicate with the task server may be very long.

Dynamic decentralized load balancing requires less time to decide the placement of tasks. Time spent distributing tasks is additional overhead to the system, and any reduction in system overhead will be felt directly in system performance. This is, of course, on the condition that the task distribution algorithm does not perform significantly worse than a centralized

algorithm. It is the problem of dynamic decentralized load balancing that we consider here.

Network topology has a major impact on the performance of a system as well. Topologies which spread nodes far apart require much more communication time than those that do not. The number of connections per node will affect the dollar cost per node of the system. Some common network topologies, like the NxM grid and the binary tree, have a constant number of connections per node, but have larger distances across the network. Other topologies, like the binary n-cube (hypercube) and the star-graph, vary the number of connections per node as the network increases in size, but have smaller distances between nodes.

Distances between nodes (measured in edges of a graph) affect performance in two ways. Smaller distances require less time to pass messages between nodes. A message which must traverse five edges will take five times as long as a message which must only cross one. The setup time required to inject a message into the network will reduce this ratio somewhat, but the longer distance will always require more time.

A second, less obvious effect, is that shorter distances allow the task load to be distributed more quickly across the system. If the distance between the two most distant nodes is k, then a node in the network is able to reach any other node in the system in k steps or less, some of which may be more lightly loaded. Thus smaller values of k for a network encourage faster load balancing.

In this paper we discuss several load balancing schemes and network topologies, and effects that one may have on the other.

2. Distributed Prolog

OR-parallelism is the concurrent evaluation of OR branches in a search tree [6]. It might include concurrent evaluation of disjunctive expressions within a clause, but it typically refers to the parallel evaluation of separate clauses. Variables are not shared between clauses, but they may be passed to clauses as parameters. The system has no concept of state - there are no

global variables or side effects.

Now because OR-parallelism involves the concurrent evaluation of clauses, clauses are a natural division for tasks. But time spent evaluating an individual clause can be broken down into two categories: time spent in unification and backtracking (execution), and time spent waiting for the evaluation of subordinate clauses (synchronization). The latter far outweighs the former if the search tree has any depth at all. The obvious exception is clauses at the bottom of the search tree, where all the time is spent in unification and backtracking. From this we obtain a large number of small communicating tasks, many of which are suspended at any given time.

In Prolog, search trees are created from top to bottom and from left to right. Prolog uses a depth-first, left to right (iterative) search strategy. In distributed Prolog the left to right restriction is removed for OR-parallel clauses. This creates a useful and a regular pattern to the execution of active tasks in the evaluation of a search tree. A parent task is active until it spawns one or more children, who then become the active tasks. The parent task is suspended until one or more of its children completes. Once a child completes, the parent may then remain active for a period of time before it either spawns more children, or it passes its results on to its parent. The child, however, may either continue to search for the next solution, or it may suspend pending a request from the parent for additional solutions. Extending this reasoning to the whole search tree, the farther a task is from the leaves of a search tree, the less active it will be.

Uncontrolled OR-parallelism, however, is not desirable. It may load up the machine with so many tasks that the machine thrashes or fails altogether due to a lack of resources. Some clauses may spawn many subtasks which consume valuable resources and prevent more critical clauses from being evaluated. The programmer may also desire that certain clauses be evaluated in order. An example might be if successive clauses represent a hierarchy of results, conditional on the failure of previous clauses.

We are experimenting with an annotated subset of Prolog, which allows us to control the amount of OR-parallelism a program exhibits. We use three annotations, PAR, PIPE and SEQ, which determine the search strategy to be used. PAR allows full OR-parallelism, with no order imposed on the results. Results will be consumed in the order they arrive at the parent task. PIPE permits parallelism during the execution of a task, but the results are consumed in the order specified by the programmer. Eventually, the various subtasks will block pending the parent's consumption and subsequent acknowledgement of the result, so the parallelism cannot continue unchecked. SEQ forces complete sequential execution - no additional tasks are formed until the previous task is completely expended.

These annotations are placed around clauses with the same name and arity in the form of a block. Blocks may be nested arbitrarily. When a block occurs within a block, it is treated as if it were a clause whose definition encompassed the clauses contained within the block, but happens to exhibit the desired parallelism.

For example,

```
PAR   /* parallel block */
   a(X) :- b(Y), c(X,Y), d(X).
   SEQ   /* sequential block */
      a(X) :- e(X), f(X).
      a(X) :- g(X).
   END
END
```

establishes a predicate named *a*, with an arity of one. The first clause will execute in parallel with the block that contains the second and third clauses. No order is imposed on those solutions. The third clause, however, is never started unless clause two fails. If the sequential block were instead a pipelined block, using PIPE as the block annotation rather than SEQ, the

third clause would be started with the second one, but the parent would not see the results of clause three until clause two had failed. PIPE allows some measure of parallelism without altering the order in which answers return to a query.

The solutions returned are bindings to variables. When task requests are formed, any binding information which partially or completely defines a variable in the child clause is included in the task request. Solutions generated by the child clause are returned as complete structures, that is, they do not contain remote references or pointers to structures not contained within the message. Messages containing bindings could be shortened by passing remote references to the actual structures or variables, but such a system would suffer additional delays when those values were actually needed. Here we are trading off longer messages against more frequent short messages, which seems reasonable given the message startup overhead in most systems.

3. Load Balancing

As a Prolog query begins evaluation, it unifies each predicate in the body of the query against clauses in the program. If the unification is successful and the clause has a body, it spawns a new task. The task must then be bound to a processor before its execution may begin. If the processor is near the parent task, the parent will spend less time waiting for results than a task whose child is far away. If the node is lightly loaded to which the new task is bound, the child will execute more quickly, hence the parent will also spend less time waiting for the results.

One can imagine a naive task allocation policy which assigns all tasks to the same processor. Such a policy would yield performance which is no better, and possibly much worse than that of a uniprocessor. Thus task allocation can be very important to system performance.

The two major resources required for the evaluation of queries are CPU time and memory. A good load balancing scheme is one which evenly distributes over the whole system, tasks which consume both of these resources. The number of tasks at each node may or may not be uniform over the system, but memory consumption and the CPU idle time should be nearly uniform. The purpose behind this is to minimize both overall execution time and the maximum memory required by a node. The load balancing scheme must execute quickly, because resources required for load balancing are additional overhead to the system.

A load balancing scheme consists of three policies: a ***transfer policy***, a ***location policy***, and an ***information exchange policy*** [3,7,8]. The transfer policy determines when to accept or transfer a new task. If the transfer policy is ***demand-driven*** [9,10], then a node issues a request for more work whenever the policy indicates the node is lightly loaded. A ***supply-driven*** transfer policy determines whether to accept a task whenever a new task becomes available. A transfer policy may also be ***pre-emptive*** or ***non-pre-emptive*** [7]. A pre-emptive transfer policy may transfer tasks which have been previously bound to a node and perhaps partially executed. A non-pre-emptive policy will transfer only newly requested tasks which have not yet been bound to a processor.

Immediately after a task is received, a supply-driven transfer policy forces a decision whether the task will be accepted for evaluation or forwarded to a neighbor. Thus as OR-parallel clauses are submitted for evaluation, a node will bind them for execution or ship them to the node most likely to accept them. A demand-driven policy will measure its load and issue a request whenever its load falls below a certain level. If the node receiving the request has a spare task, that task is shipped to the requestor. Otherwise the request is combined with others, if any, and forwarded to the neighbor most likely to have extra work.

We have chosen to explore the supply-driven policy because of its greater simplicity. It also appears the supply-driven policy will perform better when the system is lightly loaded. The demand-driven policy is very active under light loads, when each node sends requests to its

neighbors for more work. This imposes no penalty for those nodes which have no load, but it interferes with the execution of nodes which are loaded. The supply-driven policy is more of a passive policy - its stimulus comes from outside itself. This suggests it will be most active when the system is loaded. It is not clear at this time which is preferable.

A pre-emptive transfer policy is one in which any task, bound to a processor or otherwise, may be transferred to another node. For example, a demand-driven pre-emptive policy would allow a node to accept and bind any task created by other tasks existing on that node. As nodes reduce their task loads by completing task evaluations, they steal tasks from more heavily loaded neighbors. Thus the load would tend to spread evenly across the system. A non-pre-emptive policy would only allow the transference of tasks which had not been previously bound to a processor. Any task which is bound to a processor remains on that processor until its demise, regardless of the system load.

We have selected a non-pre-emptive policy to avoid the complexity associated with pre-emptive policies, because the additional overhead does not appear justified. First there is the time and resources required to move and relocate a task on a new node. Tasks in our system are fairly small, but the effort required is still much greater than, say, creating a new task. Since parent tasks do not complete before their children, it is expected that the creation of new tasks will be sufficient to balance the system. A second, and more important source of overhead comes from informing both the parent and the children tasks of the relocation. Simply forwarding messages on from the old to the new address is clearly unacceptable, since this would lengthen the communication paths, adding a substantial amount of communication time to the transmission of each message. Some forwarding would still have to take place to re-route messages in transit at the time the task is moved.

A location policy determines the node to which a task will be transferred. It may be random according to some probability distribution [7,11], based on the acceptance of bids [12,13], or based on some measurement of load pressure [9,10], etc. A completely probabilistic

location policy requires no system information whatsoever. It assigns the location according to some random function, the distribution of which has been obtained through experimentation or analysis of similar problems. A policy that uses no system state information is called a *static* policy. It is static in the sense that everything is determined *a priori*, and does not adapt to the needs of the system. A policy which relies on some determination of the current system load is termed *adaptive*, or *dynamic*. Considering the unpredictability of the shapes of the search trees generated by Prolog programs, it is hard to believe that any static approach would yield promising results for this type of problem.

As the load on each node changes, a bidding policy causes each node to send a notice to all neighboring nodes. Each neighbor subsequently returns a bid, which represents its own load. When all bids are received, a partner is chosen with whom the bidder will swap tasks. The bidder and the partner then send each other a task which each believes will be better for the other. This policy may work well if the amount of time required by the bidding process is small when compared to the execution time of each task. As was mentioned earlier, in our system there are many small tasks which causes rapid changes in system load, so a bidding policy would heavily load the communication network and cause extensive delays in the assignment of tasks to nodes.

In a sense, a bidding policy is very similar to a pre-emptive demand-driven pressure model. In a pressure model, a node compares its own load value with those of its neighbors. If its own load value is the lowest, the node retains the task, or steals one from the node with the highest load, depending on the transfer policy. Otherwise, the neighbor with the lowest load value becomes the recipient of the task. A major benefit of this policy is that decisions are made quickly based upon information which is retained locally. The corresponding drawback is that occasionally the system will make use of inaccurate information. It is the responsibility of the information exchange policy to insure this happens only rarely, as will be discussed below. We have chosen a pressure model as the location policy for our system.

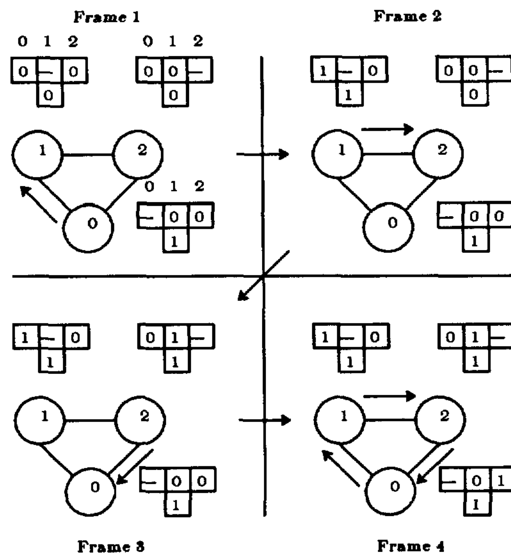

Figure 1 - Uncorrected Channel Effect

The information exchange policy is responsible for maintaining current system load information at each node. Because of the large number of messages inherent to a distributed Prolog system, load information may be piggybacked onto messages already being sent to neighbors. The paths that messages take are unpredictable, but they are not random. Thus some nodes may only receive messages infrequently from certain neighbors, and therefore will retain load values for those neighbors which will eventually become obsolete. Admittedly this is more of a problem when the system is lightly loaded, since the total number of messages is less. Two times when this is most likely to occur are the beginning and end of a run.

Two problems that can occur because of inaccurate information are *channels* and *blackouts* (our terminology). A channel is a path or a cycle in the network through which new tasks travel against the system load because of false information. Figure 1 illustrates how this might happen. The T-shaped boxes represent each node's understanding of the system load. Its own load is the lower box. In frame 1 of the figure, node 0 has one task which it is executing. That task spawns a child task which is sent to node 1, sending its load information along with the new task. Node 1 accepts the task, which spawns another task, which is sent to node 2 (frame 2). This task, in turn, spawns yet another which is sent to node 0 (frame 3). At this point each

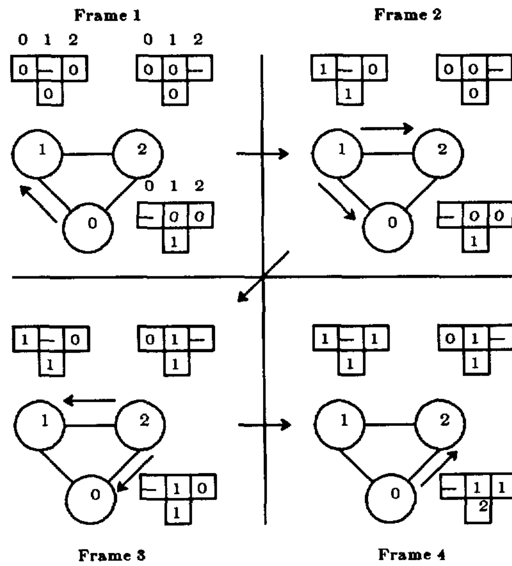

Figure 2 - Corrected Channel Effect

node has exactly one task under execution, and there is one task "floating" around the network. Each node believes it has one neighbor with a load and one without. It is here the channel becomes a problem (frame 4). Node 0, thinking that node 1 has no work to do, sends it the task. Node 1 does the same to node 2, which in turn sends the task back to node 0. The "floating" task will continue to cycle until some node, for whatever reason, sends a message in the opposite direction of the channel.

Figure 2 gives an example of an information exchange policy which does not allow the formation of channels. Briefly, whenever a node accepts a task for execution, it returns a message to the neighbor which sent the request, informing it of its new load value. A second value, the *perceived* value of the receiver's load is included in each message. If the receiver's load does not closely match the sender's perception of its load, a message is returned to the sender (if no other messages are scheduled) correcting it's value.

A blackout is a blockage of communication based on a perceived load which is greater than the actual load. This is in contrast to a channel, which is created because the perceived load is *less* than the actual load. Figure 3 illustrates how a blackout may occur. The load for each neighbor is relatively high, and each node correctly perceives the load. If the load then

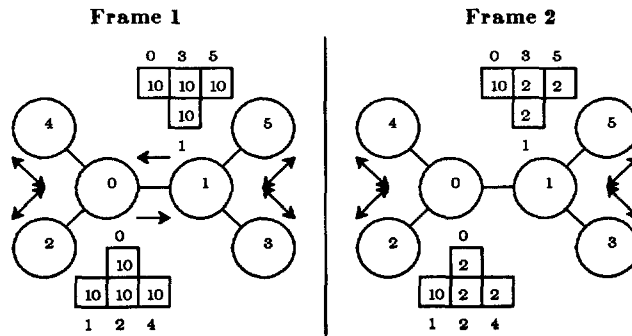

Figure 3 - Blackout Effect

declines rapidly for all nodes without any communication occurring between nodes 0 and 1, node 0 will perceive the load to be very high on node 1 and *vice versa.* This will form a barrier preventing the exchange of tasks between these nodes. This barrier can be broken down by other messages passing between them, but that may not happen if the system is lightly loaded.

A solution to this problem is to use a timeout. After each time period elapses, the perceived load value of the uncommunicating neighbor is reduced, eventually to zero. The rate at which such reductions take place must be carefully chosen so as to avoid basing many decisions on inaccurate information. This reduction rate is static, and will have to be derived through further study.

We are experimenting with a number of heuristics for the transfer and location policy functions. We have a number of different system parameters we are able to measure, which we believe reflect the system load. They include: the number of active tasks, the number of suspended tasks, and current memory usage (in bytes). The balance between active and suspended tasks on any given node affects the utilization of CPU time, whereas it is the total number of tasks which affects the memory usage. In addition, we have two variables which are important to the acceptance or transfer of an unbound task, but do not reflect the system load. They are the distance of the unbound task from the parent task (which affects elasticity) and the network's communication bandwidth (which affects viscosity). A given heuristic would be a weighting function of these factors, so that location, and transfer could be computed, much as a

chess-playing program computes its next move.

The *elasticity* of the location policy is a function which is directly related to the distance between the new task and its parent. Elasticity tends to keep tasks near their parents, much like a band of elastic tends to pull its ends together when stretched. As tasks move further away from their parents, communication becomes more expensive. As tasks move closer to their parents, the load tends to become unbalanced. Thus elasticity is used to balance the pressure created by the system load. The optimal balance of the forces of pressure and elasticity are an important issue, since they represent the conflicting goals of balancing the load and minimizing communication costs.

The *viscosity* is a function which is *inversely* related to the bandwidth of the system. It is a measure of the cost incurred by transferring a task to another node. The viscosity represents the expense incurred by transferring a task to another node rather than accepting it for evaluation. A fast network would have a low viscosity, which would allow tasks to move around the system easily. It is of lesser importance than elasticity, but nevertheless should not be ignored. As elasticity or viscosity increase, greater pressure is required to move tasks around the system.

4. Network Topology

Loosely-coupled message-passing systems have the advantage that a large number of nodes may be connected to a network with a relatively small increase in cost per node. A network is characterized by *degree, order, diameter,* and *average diameter.* The *degree* of a network is the largest number of neighbors held by a node in the network. The dollar cost of each node in a system depends in part on the degree of the network. The maximum performance of a distributed system, however, is determined by the number of processors, or *order* of the system. The *distance* between two nodes is the smallest number of edges that connects them. Performance is also dependent, in part, on the distance between the nodes. This dependency

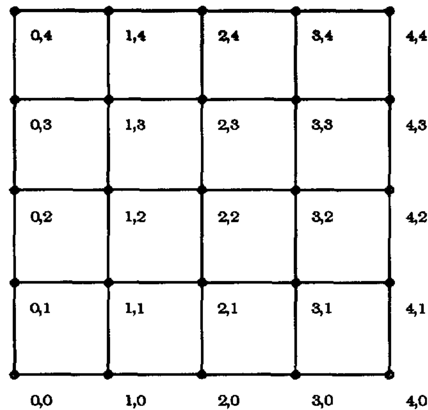

Figure 4 - 5 x 5 Grid

can be reduced by allocating tasks so that messages only travel to local nodes, but one must be careful with this as it may induce an imbalance in the system load. The ***diameter*** is the largest distance a message must travel between any two nodes. The ***average diameter*** is the average distance a message must travel between nodes.

The optimal network for a distributed Prolog system can now be categorized. It would have a small degree to keep the cost per node low, and the order would be suitably large to provide high performance. The average diameter would also be small to minimize the cost of message passing. The average diameter roughly indicates the cost of communication, but with a suitable elasticity in the load balancing algorithm, children tasks can be retained locally, which further reduces communication costs. The average diameter also gives some indication of how quickly the nodes in the network may be reached. A network with a small average diameter would allow more nodes to be reached in fewer hops than a similar network with a larger average diameter. This in turn would allow the load balancing scheme to spread tasks more evenly across the network.

Some network types, such as the NxM grid (figure 4), encourage tasks to either bunch together on few nodes, or travel long distances to find unloaded nodes. This is because of the long distances task requests must travel to reach a significant fraction of the nodes in the network. Each task may request a large number of children tasks. Each child may also request

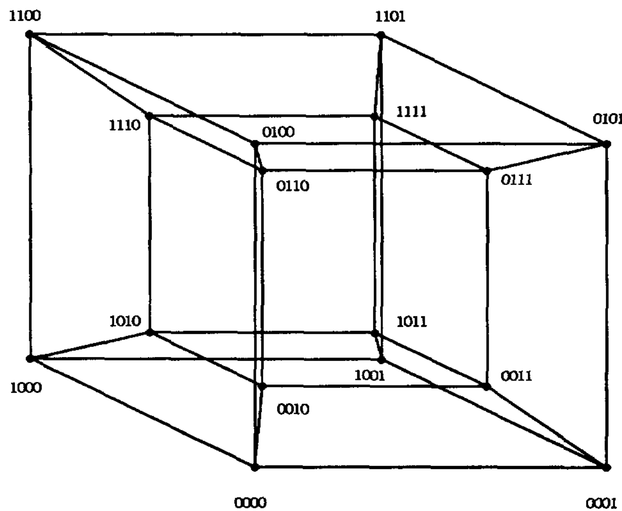

Figure 5 - 16 Node Hypercube

additional tasks. It does not take many generations to create a large number of tasks. If the tasks do not spread quickly to all parts of the system, each successive generation will increase the system imbalance. If the tasks must travel long distances to reach the network extrema, either they will travel long distances or they will bunch up. If tasks bunch up, the system load becomes unbalanced and the performance is reduced. If they travel, communication between nodes becomes more expensive. Again the performance is reduced.

Other networks, such as the binary n-cube (figure 5) and the star-graph (figure 6) encourage better task distribution and offer less expensive communication. The binary n-cube, or hypercube, is particularly well known, being the topology of choice for several commercial products. Tables 1 and 2 give a comparison of degree, order, diameter, and average diameter for several networks. In table 1 H_n represents the nth harmonic number. A definition for H_n may be found in [14] (page 73). The definition of a star-graph and a derivation of its formulae may be found in [15,16,17]. The grid, hypercube, and other network topologies may be found in [16,18].

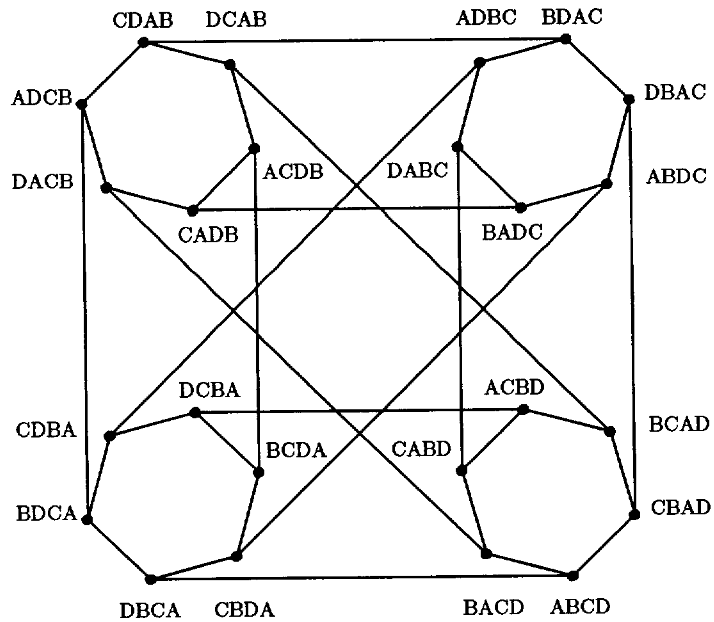

Figure 6 - 24 Node Star-Graph

Network	*Degree*	*Order*	*Average Diameter*	*Diameter*
NxM Grid	4	nm	$\frac{n^2-1}{3n}+\frac{m^2-1}{3m}$	$n+m$
Hypercube	n	2^n	$\frac{n}{2}$	n
Star-Graph	$n-1$	$n!$	$n+H_n-4+\frac{2}{n}$	$\left\lfloor \frac{3(n-1)}{2} \right\rfloor$

Table 1

Network	*Degree*	*Order*	*Average Diameter*	*Diameter*
11x11 Grid	4	121	~7.3	22
Hypercube	7	128	3.5	7
Star-Graph	4	120	~3.7	6
27x27 Grid	4	729	~18.0	54
Hypercube	9	512	4.5	9
Star-Graph	5	720	~4.8	7
71x71 Grid	4	5041	~47.3	142
Hypercube	12	4096	6.0	12
Star-Graph	6	5040	~5.6	9

Table 2

As can be seen from the tables, the NxM grid is not the best possible network topology. Both the binary n-cube and the star-graph do much better in terms of diameter and average diameter. The binary n-cube and the star-graph have very similar performance, with an edge going to the star-graph. The most significant difference between the two is the degree.

The advantage of a small degree is a lower cost per node. Another advantage is that if the number of connections for each node is fixed, as is the case with Inmos Transputers (which have four connections each), one may build a larger star-graph than hypercube. The largest hypercube one may build with the Transputer has 16 nodes. The largest star-graph has 120 nodes, or 7½ times the processing power.

For large networks, the star-graph shows an improvement in both the diameter and the average diameter. They are somewhat difficult to compare, however, because their sizes do not correspond very closely. Advanced networks like the star-graph have not yet been chosen for commercial application partly because they are not well understood. The binary n-cube, on the other hand, is well known to be easily partitioned into many other topologies, for example a ring or a mesh, and is also known to be suitable for many common numeric problems. These properties of the n-cube, though useful in their own right, hold no advantage for our application. Because our system promises no particular regularity in its execution patterns, we expect the best performance from the network which offers the smallest degree and average

diameter for its order.

Of particular interest for our application is the truncated star-graph network, which allows the use of subsets of the full star-graph. The extra connections left open by the truncation are used to increase performance further, by selecting the nodes which are furthest away and connecting them. Some of the network symmetry is lost by this process, but that incurs no penalty for our application and is of little import.

5. Summary

OR-parallel distributed Prolog is able to generate a large number of small, communicating tasks. This makes network performance and load balancing issues of paramount importance. We believe a supply-driven non-pre-emptive pressure model coupled with a high performance network offers excellent possibilities for creating a system to evaluate an OR-parallel distributed Prolog. The star-graph network has a small degree and average diameter considering the number of processors it will support. We believe it is these properties which affect the network performance under our application.

6. Future Research

The implementation of a distributed OR-Prolog simulator is complete, and suitable benchmark programs are being designed. The simulator allows the variation of message delays and transmission rates, network topology and size, and load balancing scheme. Our work is now focused on developing and testing the different heuristic functions used by the transfer and location policies.

We intend to simulate other networks in addition to the star-graph, in order to form a more complete picture of the effect topology has on performance in this environment. Additional networks will include a grid, the binary n-cube (hypercube), pancake network [15,16], and cube connected cycles [18]. Each network will be compared for performance against a

completely connected network, since that represents the network with the highest possible performance, although such networks are rarely built because of the overwhelming expense involved.

In order to validate how accurately the simulator reflects the type of environment we have described in this article, we will implement our system on both an Intel iPSC † (Hypercube) and a network of Transputers, whose link connections will be software reconfigurable using the new Inmos 32x32 crossbar chip.

References

[1] John S. Connery, "The AND/OR Process Model for Parallel Interpretation of Logic Programs", Ph.D. Thesis (TR 204), University of California, Irvine, June 1983

[2] John S. Connery and D. F. Kibler, "Parallel Interpretation of Logic Programs", *Proceedings of the ACM Conference on Functional Programming Languages and Computer Architecture,* pp 163-170, October 1981

[3] Derek L. Eager, Edward D. Lazowska, and John Zahorjan, "Adaptive Load Sharing in Homogeneous Distributed Systems", IEEE Transactions on Software Engineering, Vol. SE-12, No. 5, May 1986

[4] Hassan M. Ahmed, Jean-Marc Delosme and Martin Morf, "Highly Concurrent Computing Structures for Matrix Arithmetic and Signal Processing", *IEEE Computer,* Volume 15, Number 1, pp 65-82, January 1982

[5] Vineet Singh and Michael R. Genesereth, "A Variable Supply Model for Distributing Deductions", in *9th IJCAI,* pp 39-45, August 1985

[6] Kunio Murakami, Takeo Kakuta, Rikio Onai and Noriyoshi Ito, "Research on Parallel

† iPSC is a trademark of Intel Corporation

Machine Architecture for Fifth-Generation Computer Systems", *IEEE Computer*, Volume 18, Number 6, pp 76-92, June 1985

[7] Amnon Barak and Amnon Shiloh, "A Distributed Load-balancing Policy for a Multicomputer", *Software - Practice and Experience*, Vol 15(9), pp 901-913, September 1985

[8] Thomas L. Casavant and Jon G. Kuhl, "A Formal Model of Distributed Decision-Making and Its Application to Distributed Load Balancing", in *6th International Conference on Distributed Computing Systems*, pp 232-239, IEEE Cambridge, Massachusetts, May 1986

[9] Frank C.H. Lin and Robert M. Keller, "Gradient Model: A Demand-Driven Load Balancing Scheme", in *6th International Conference on Distributed Computing Systems*, pp 329-336, IEEE, Cambridge, Massachusetts, May 1986

[10] Robert M. Keller and Frank C.H. Lin, "Simulated Performance of a Reduction-Based Multiprocessor", *IEEE Computer*, Volume 17, Number 7, pp 70-81, July 1982

[11] Chi-Yin Huang Hsu and Jane W.-S. Liu, "Dynamic Load Balancing Algorithms in Homogeneous Distributed Systems", in *6th International Conference on Distributed Computing Systems*, pp 216-223, IEEE, Cambridge, Massachusetts, May 1986

[12] Dan Hammerstrom, "Dynamic, Decentralized Load Leveling", in *Euromicro 80*, London, England, October 1980

[13] John A. Stankovic and Inderjit S. Sidhu, "An Adaptive Bidding Algorithm For Processes, Clusters and Distributed Groups", in *4th International Conference on Distributed Computing Systems*, pp 49-59, IEEE, San Francisco, California, May 1984

[14] Donald E. Knuth, *The art of Computer Programming, Volume 1* (1973), Addison Wesley

[15] Sheldon B. Akers and Balakrishnan Krishnamurthy, "Group Graphs as Interconnection

Networks", in *14th International Conference on Fault-Tolerant Computing,* Kissimmee, Florida, pp 422-427, June 1984

[16] Sheldon B. Akers and Balakrishnan Krishnamurthy, "A Group Theoretic Model For Symmetric Interconnection Networks", in *1986 International Conference on Parallel Processing,* pp 216-223, August 1986

[17] Sheldon B. Akers and Balakrishnan Krishnamurthy, "The Fault Tolerance of Star Graphs", to appear in *Second International Conference on Supercomputing,* Santa Clara, California, May 1987

[18] Tse-yun Feng, "A Survey of Interconnection Networks", *IEEE Computer,* Volume 14, Number 12, pp 12-27, December 1981