Comparing the Performance of Two Dynamic Load Distribution Methods

L.V. Kalé

Department of Computer Science
University of Illinois at Urbana-Champaign
1304 W. Springfield Ave, Urbana, IL-61801
(217) 244-0094

ABSTRACT

Parallel processing of symbolic computations on a message-passing multiprocessor presents one challenge: To effectively utilize the available processors, the load must be distributed uniformly to all the processors. However, the structure of these computations cannot be predicted in advance. So, static scheduling methods are not applicable. In this paper, we compare the performance of two dynamic, distributed load balancing methods with extensive simulation studies. The two schemes are the *Contracting Within a Neighborhood* (CWN) scheme proposed by us, and the *Gradient Model* proposed by Lin and Keller. We conclude that although simpler, the CWN is significantly more effective at distributing the work than the Gradient model.

This work was supported in part by a grant from the National Science Foundation, grant # CCR-87-00988

Comparing the Performance of Two Dynamic Load Distribution Methods

ABSTRACT

Parallel processing of symbolic computations on a message-passing multi-processor presents one challenge: To effectively utilize the available processors, the load must be distributed uniformly to all the processors. However, the structure of these computations cannot be predicted in advance. So, static scheduling methods are not applicable. In this paper, we compare the performance of two dynamic, distributed load balancing methods with extensive simulation studies. The two schemes are the *Contracting Within a Neighborhood* (CWN) scheme proposed by us, and the *Gradient Model* proposed by Lin and Keller. We conclude that although simpler, the CWN is significantly more effective at distributing the work than the Gradient model.

1. Introduction

Processor utilization is a key factor that decides the speedup provided by a parallel system. A thousand processor system can provide a speedup of 1000 only *if all the processors can be kept busy all the time*. Ideally, we would like to divide the computation in P equal parts (where P is the number of processors), one for each processor. Of course, it is usually impossible to identify 'P equal chunks' except for highly structured computations. A potential alternative is to divide the computation into a large number of medium granules. (Too small a grainsize would lead to undue overhead.) Then, even if these granules are of unequal sizes, their large number would allow us to distribute them equally. Many parallel evaluation schemes for functional programs, logic programs, problem-solving etc., offer such a medium grain of parallelism.

The large pool of tasks resulting from a medium grain-size may lead to a increased speedup only if there is an effective load distribution scheme that ensures that no processors remain idle while there is work available in the system. On shared

memory machines, the load balancing is relatively simple. We can maintain all the work in a central pool, and let the PEs draw it from there. The obvious problems of contention can be resolved with clever schemes. Load distribution is a much more complicated task on a message–passing multiprocessor. Nevertheless, message–passing systems have their attractions. Many researchers believe them to be more scalable and cost effective than shared memory systems.

What sort of load balancing system is needed for a message passing system? The unpredictability of computation structures in general implies that it must be a dynamic or run–time strategy, as opposed to a static or compile–time strategy. For scalability, it must not be centralised at a few PEs, but distributed on all of them. Also, it should not depend on global information. Each PE should only use the information provided by its neighbors.

In this paper, we compare the performance of two such dynamic and distributed load balancing schemes. One of them is '*contracting within a neighborhood*' (CWN), a relatively simple strategy proposed by us [1]. The other is the *Gradient Model* (GM) proposed by Lin and Keller [3]. In the next section we describe the two schemes and their rationale. Section 3 describes the simulation set–up, and some preliminary experiments to select the parameters for the main simulations in a fair manner. The results of the simulation experiments are presented in Section 4. We conclude with some ideas for future work.

2. The Competitors

The medium grain tasks found in most application domains have some interesting features in common. When activated, such a task executes for a short time, and then either completes, or starts some sub–tasks and awaits response from them. When it receives a response, it repeats the same cycle. Usually, it is prohibitively expensive to move a task from a PE to another after it has spawned sub–tasks. Both the strategies we describe avoid that. They do differ as to when a task is distributed: in CWN, a

task is scheduled on some PE as soon as it is created, whereas the Gradient Model keeps the newly created tasks on the source PE, and distributes them when required.

2.1. Contracting Within Neighborhood

This scheme is based on the fact that global communication – allowing communication between arbitrary pairs of PEs– is not scalable. In a system with global communication, as the number of PEs is increased, a point is reached beyond which the system is always communication bound. This is true for any interconnection scheme which uses a fixed number of connections per PE. Luckily, in the tree structured computation domains it is possible to avoid global communication as the communication is almost exclusively between parent and child tasks. Therefore this scheme restricts a child task to be within a fixed radius from its parent, i.e. within a fixed communication *neighborhood.* Also, in the interest of agility, this scheme sends every subgoal out to another PE as soon as it is created. The algorithm followed by each PE is as follows:

Each PE maintains the load information about its immediate neighbors. This information can in general be a combination of various factors that gauge the current and future 'load' on that PE. A simple measure may be simply the number of messages waiting to be processed by that PE. This information is obtained by broadcasting a very short message to all the neighbors periodically, or as an optimization, piggybacking the load information 'word' with regular messages, whenever possible. Any time a subgoal is created on a PE, it consults this load information, and sends the new goal message to its least loaded neighbor. The message also includes a count field that says how many hops the message has travelled from the source. A PE that receives such a message checks to see if the hop count is equal to the allowed *radius.* If so, it must keep the goal for processing. Otherwise it sends the goal to its least loaded neighbor after adding 1 to the count. If a PE finds its own load is less than its least loaded neighbors, it keeps the goal provided the message has travelled a stipulated *minimum hops* already. Thus, a new subgoal travels along the steepest load gradient

to a local minimum. A goal, once it is accepted by a PE, remains there, and is finally executed by that PE. It cannot be re-sent elsewhere.

Because it follows the local load gradients, it is possible that this scheme does not send a given subgoal to the least loaded PE in the neighborhood, because of the horizon affect. However, looking for the least loaded PE in the neighborhood would be expensive. The minimum hops are stipulated to alleviate this problem to some extent. A source PE cannot keep a piece of work simply because it thinks it is the least loaded among its neighbors. It must send it some distance to 'look over the horizon', and then possibly get it back.

The scheme is naive on several counts. First, requiring every piece of work to be contracted out to another PE seems excessive. Also, once a goal reaches its 'destination' it remains stuck there, which removes opportunities for a correction as time goes on. However, the strategy is meant as a starting point. The simulation studies should suggest specific ways of improvement.

From the point of view of simulation, it is important to remember that the scheme has two parameters: the *radius*, i.e. the maximum distance a goal message is allowed to travel, and the *horizon*, i.e. the minimum distance a goal message is required to travel.

2.2. The Gradient Model

The gradient model is a more elaborate scheme than CWN. Whenever a subgoal is generated, it is simply entered in the local queue. A separate, asynchronous process exists for the load-balancing functions. This process wakes up periodically, and computes the load on the PE as in CWN. Using two parameters, the *low-water-mark* and *high-water-mark*, it decides the *state* of the node as follows. If the load is below the *low-water-mark*, the state is *idle*. If the load is above the *high-water-mark*, the state is *abundant*; Otherwise, it is *neutral*. It then computes its *proximity*: An idle node has a 0 proximity. For all other nodes, the proximity is one more than the smallest

proximity among the immediate neighbors. If the calculated proximity is more than network diameter, then it is set to (network diameter +1), to avoid unbounded increase in proximity values. If the proximity so calculated is different than the old value, then it is broadcast to all the neighbors. All the PEs initially assume that the proximities of their neighbors are 0. After this, if the state were idle or neutral, the process sleeps until the next interval. If the state is abundant, it sends a goal message from the local queue to the neighbor with least proximity. Any PE that receives a goal message from its neighbor just adds it to its queue. This may, of course, change its state which will be noticed when the gradient process wakes up.

The proximity of a PE represents a guess at the shortest distance to an idle PE. It is a 'guess' because by the time the information about an idle PE reaches another PE via the update-and-broadcast-proximity sequence, the state of some PEs may have changed. The proximity is a good example of how approximate global information can be maintained using only local checks.

The rationale behind the gradient model is to keep work locally as far as possible, and to send work out towards a PE that is in danger of being idle.

This strategy is parameterized by: the *low-water-mark*, the *high-water-mark*, and the sleeping interval between two execution cycles of the gradient process.

3. The simulation set-up

The simulations were carried out on ORACLE, a multi-processor simulation system we are developing. ORACLE is written in SIMSCRIPT, a discrete-event simulation language with excellent statistical support. In addition to the *events* it supports the *process* abstraction. Thus the code written for ORACLE looks the same as that for a real multi-processor. It is not ideal for complex (recursive) programs or data structures, but we found the benefits outweighed this drawback. ORACLE has one process for each user process running on a PE, and one process for each communication channel. Thus it models contention for the basic resources of a parallel system.

ORACLE accepts input specifications such as the number of PEs and their interconnection scheme, the load balancing strategy to be used (from its repertoire of strategies), control strategy options, form and content of the output information required, a program to execute and times to be charged for primitive operations.

ORACLE can provide statistics on a variety of performance aspects such as the overall average PE utilization, average utilization of individual PEs, average and individual utilizations of communication channels, the time to completion, and the time of any output produced by the user program. It also provides a specially formatted output that can be used to drive a graphics program to monitor load distribution. Here the utilization of each PE is output at every sampling interval. This data is displayed on the graphics device with a continuum of colors representing relative activity on each PE. (red: busy,.. blue: idle). We found this facility particularly useful for debugging the load balancing strategies.

A point worth noting is that when we run a program on the simulation system, we get the result of the program, in addition to the performance stastistics. In contrast, a trace driven simulation approach would be to carry out the computation in advance, producing a trace, which will then be used by the simulation system to get the performance figures. We found such an approach would not save much in terms of simulation time. Another approach could be to use a statistical model of computation. In absence of any uniform model of parallel computations, it was thought to be too unreliable and ad-hoc an option. So we opted for executing specific computations with well-understood structures.

Given the two schemes, we needed to select situations, or sample points at which to compare them. The choices varied on many dimensions:

- The interconnection topologies.
- The number of processors.

- The computation: structure and size.
- The communication to computation ratio.

We selected 2 interconnection topologies: the 2–dimensional grid (nearest neighbor grid) with wrap–around connections and the double–lattice–mesh topologies. The grid was used in some of the preliminary simulations of the gradient model by Lin [4]. The double–lattice–mesh (See Figure 1) is a bus–based topology that we have proposed [2], and is therefore of some interest to us. We also decided to simulate systems with 25 to 400 PEs. Beyond 400 PEs, the time required for simulations was prohibitive. Also, we feel that range should be sufficient to understand how the schemes will behave when the size of the system changes.

In general, the parallel computations may have arbitrary data–dependencies among sub–tasks, and the inherent parallelism in the computation may wane and rise as computation progresses. To be able to interpret the simulation results, and get an understanding of how the load balancing schemes behave, we needed a predictable computation, whose structure is easy to grasp. Then, there won't be ambiguities about whether a certain feature that is seen in the simulation data is due to the nature of the computation or due to the load–balancing scheme. We chose to use *divide–and–conquer*, and *naive–fibonacci* programs for these reasons, and also because they were simple to implement. The *divide–and–conquer* (abbreviated *dc*) program was used by Lin, and may be written as:

dc(M,N) ← **if** M = N **then** M **else** dc(M,(M+N)/2) + dc(1 + (M+N)/2, N)

The *naive–fibonacci* is the doubly recursive function to compute fibonacci numbers.

fib(M) ← **if** M < 2 **then** M **else** fib(M–1) + fib(M–2)

It must be pointed out that we are not really interested in how to compute this functions in parallel. There are much more efficient methods for computing them. We are simply interested in the computation trees they yield. The *dc* computation provides a well balanced tree, whereas the *fibonacci* yields a not–so–well–balanced tree. The

inherent parallelism in both computation rises steadily and then falls steadily. In real life computations, the parallelism may rise and fall in cycles. So the behaviour of the schemes on these computations can be used to infer it in the general situation. Also, observing them in the controlled situation will give us insights into how the schemes behave.

We used 6 different computation sizes for each program. Fibonacci of 7, 9, 11, 13, 15 and 18, and the *dc* computations of the same sizes, namely: dc(1,X) for X=21, 55, 144, 377, 987 and 4181.

As we wanted to focus on effectiveness of load distribution, we decided to isolate the factor of communication load. We chose the ratio of communication to computation to be such that communication stagnation does not occur.

3.1. The optimization experiments

Recall that each of the schemes has a few parameters that have to be selected (The water-marks for GM, and the radius for CWN, for example). In the interest of fairness, the parameters must be chosen in such a way each scheme is working at its best. We chose a few sample points in the space of planned experiments, and ran the simulations for various combination of parameters. The winning combinations were used for the comparison experiments. The parameters so chosen are shown in the table below.

It is worth noting that the 20 units interval is fairly low, as the total execution time for simulations ranged from 1000 to 23000 units. That means the gradient process is running very frequently, which should be an asset to its performance. Also, we assume a communication co-processor to handle the routing and load-balancing functions. Without such a co-processor, the gradient model will suffer more, because it needs to execute a more complex code and more frequently.

parameter	for experiments on the grid topologies	for experiments on the lattice–meshes
CWN: *radius*	9	5
CWN: *horizon*	2	1
GM: high–water–mark	2	1
GM: low–water–mark	1	1
GM: interval between cycles:	20 units	20 units

Selected Parameters

Table 1

4. Simulation Results, and Interpretation

The choices of sample points mentioned above lead to 240 simulation runs (2 problem types * 6 problem sizes * 2 topology types * 5 topology sizes * 2 strategies). The simulations were run on a VAX–750. Each run took between 15 minutes to 3 hours of time on the Vax.

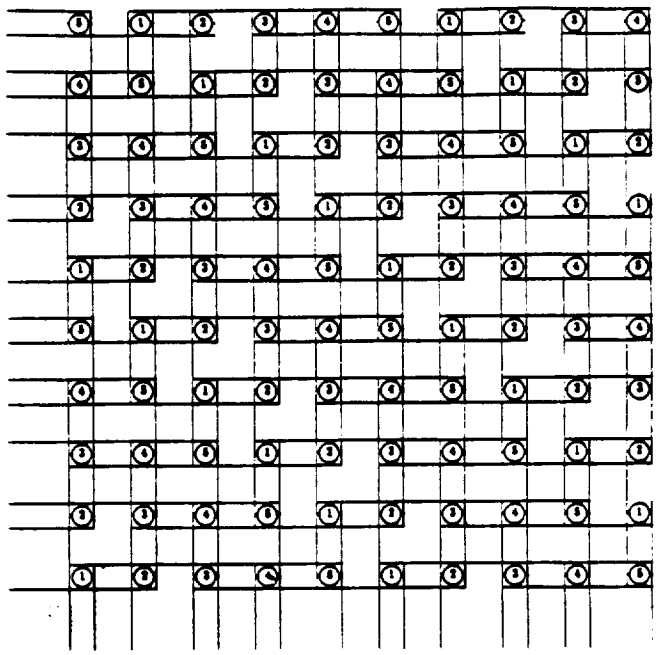
Plots 1 through 10 show the performance of the two schemes on the divide–and–conquer computations. Each plot depicts experiments done on a specific topology, for one problem type. Thus Plot 1 shows the results of 6 *dc* computations of varying sizes, running on a double–lattice–mesh with 400 (20x20) PEs. The Y–axis shows the average PE utilization in percents. The X–axis is the problem–size in total number of goals generated during the computation. The speedup can be computed by multiplying the number of PEs by (average utilization percentage/100).

On the grid topologies, the CWN is a clear winner by substantial margins. On the double lattice–meshes also CWN consistently performs better than the GM. The only one case seen in these plots where CWN is outperformed by the GM occurs in plot 3, while running *dc(1,4181)* on a DLM with 100 PEs.

	Grids					Double Lattice Meshes				
PEs	25	64	100	256	400	25	64	100	256	400
fib(7)	1.56	1.57	1.44	1.57	1.57	1.30	1.18	1.24	1.18	1.23
fib(9)	1.56	1.53	1.30	1.56	1.56	1.06	1.14	1.33	1.14	1.21
fib(11)	1.56	1.56	1.79	1.92	1.92	1.09	1.12	1.06	1.11	1.16
fib(13)	1.60	1.92	1.83	1.71	1.71	1.09	1.08	1.09	1.04	1.10
fib(15)	1.58	2.14	2.03	2.56	2.56	1.21	1.14	1.04	1.05	1.04
fib(18)	1.74	1.72	2.18	3.03	3.09	1.24	1.20	0.87	1.09	1.08
dc(1,21)	1.46	1.47	1.44	1.47	1.47	1.41	1.46	1.51	1.46	1.51
dc(1,55)	1.37	1.33	1.37	1.33	1.33	1.17	1.51	1.35	1.51	1.38
dc(1,144)	1.39	1.48	1.38	1.48	1.48	1.25	1.25	1.40	1.32	1.52
dc(1,377)	1.28	1.72	1.34	1.65	1.65	1.17	1.16	1.11	1.12	1.44
dc(1,987)	1.38	1.89	1.98	2.09	2.09	1.17	1.21	1.09	1.06	1.29
dc(1,4181)	1.36	1.42	2.27	2.91	2.82	1.30	1.27	0.96	1.18	1.31

Speedup of CWN over GM

Table 2



A 10x10 Double Lattice Mesh with bus-span = 5

Figure 1

The Fibonacci plots are very similar, so we omit them from the plots. However, the comparative figures from all the runs are shown in table 2. For each run, we show the ratio of speed-ups obtained using CWN to that obtained using GM. In 118 out of 120 cases, the CWN is seen to be better. In 110 of those cases, the difference is significant, i.e. more than 10%. On grids at times the CWN leads to thrice as much speed as GM.

The DLM topologies have smaller diameters (4–5) compared to the grids (ranges from 8 to 38). The superior performance of CWN on the grids leads us to conjecture that it performs better than the GM on large systems, which of course tend to have larger diameters.

To understand the operation of each method, we plot the utilizations during short sampling intervals throughout the course of computation, for a few selected computations. We included the case where CWN was weakest. Plots 11 through 13 show the utilization as time varies for the 100 PE double-lattice-mesh for 3 Fibonacci computations. Plots 14–16 show similar plots for the 100 PE grid. The first thing we notice is that the CWN has much faster 'rise-time' than GM: it spreads work quickly to all the PEs at beginning. Plots 11 and 12 also show its pitfalls. Although it takes the system close to 100% utilization quickly, it cannot maintain the performance at that level. The Gradient model manages to maintain 100% when it reaches that level. This is because of the re-distribution of work that the GM is capable of. For CWN, once a goal is sent to a PE, it must be executed there, although the load conditions may change after that. The only way it has to correct such imbalances is using newly created goals, which limit its ability to supply work to idle processors. Another problem we notice is the extended tail in plot 11. This suggests that only a few processors were involved in the computation in that phase. We believe the reason for this to be our current method for computing the load on a PE. We simply count all the messages waiting to be processed as 'load'. This ignores potential future commitments,

indicated by the count of the tasks that are waiting for messages.

The main problem with GM is that it is not agile enough. PEs hoard work until they are sure they are 'abundant'. On the grids, a stronger flattening is seen in, say, plot 15. When about 40% of the PEs have received work, most PEs think there is not sufficient work to distribute it to others, and so keep the new goals they generate, which leads to loss of parallelism, and as a result not enough work gets generated. This 'vicious cycle' is responsible for the flattening of the plot.

Examination of the detailed simulation output, not shown here, reveals another potential problem with CWN. Typically, it requires thrice as much communication as the GM. In the Gradient Model, the average distance travelled by a goal message is typically less than 1. A significant number of goals just stay at the PE they were created on. On the grids, with CWN the distance travelled is about 3. Table 3 shows the distribution of distances traveled by messages for Fibonacci of 18 on a 10x10 grid.

Hops	0	1	2	3	4	5	6	7	8	9	Average
CWN	1	3979	1024	713	514	375	298	223	202	1032	3.15
GM	4068	2372	1045	527	195	84	43	20	4	3	0.92

Table 3: distribution of message distance.

The cost of sending every goal away is clearly seen. The sudden rise at 9 hops for CWN is because 10 is the allowed radius. A message that has gone that far must stop at that distance.

5. Conclusions and Future Work

Dynamic load balancing methods are necessary for parallel processing of unpredictably structured computations. We compared the performance of two such methods. The *contracting within neighborhood* method (CWN) is simpler: every time a new goal is created, it is sent along the steepest load gradient towards a local minimum within a specified radius from the source PE. The goal cannot be re-sent

from that PE afterwards. The Gradient Model (GM) is more sophisticated. It attempts to direct work from abundant PEs to those in danger of being idle. By default, the work is kept locally, and sent out only when the presence of an idle node is inferred. Unlike CWN, the GM requires a separate, asynchronous process to handle the load balancing functions.

We conducted some simulation experiments to select the parameters for the strategies that will ensure they operate in their optimal ranges. We then ran an extensive set of simulation experiments, each consisting of running a specific computation on a specific topology. We compared the perfomance, i.e: the speed-ups achieved, by both schemes. CWN was found to yield substantially larger speed-ups than GM in most situations examined.

Although CWN performs better than GM in most experiments reported here, it still has a large room for improvement. First, CWN does not allow a goal to be re-distributed once it has been sent to another PE. As seen in Plots 1 and 2, CWN can benefit from some re-distribution of work. There, the available work is just sufficient to keep every PE busy, but as the CWN cannot re-shuffle work, some PEs remain idle. However, this is not of much use when the work is more than sufficient or when it is too little. So, a small, well-controlled (i.e. responsive to runtime conditions) re-distribution component should be added to CWN. As seen from the communication distances (Table 3), CWN certainly needs *saturation control.* When the system is running at 100% utilization, there is no need to send every goal out to other PEs. Detecting such a situation and then keeping goals locally until the situation changes would be worth investigating. Notice that both of these amount to incorporating the good features of GM in CWN. Taking future commitments into account while computing the load is another suggestion stemming from the observation about extended tail of computation in Plot 11. Care must be taken not to lose the agility of CWN while modifying it.

A note of caution is in order. We chose a low communication to computation ratio to ensure that communication stagnation does not interfere with the property we were trying to measure: namely, the ability to distribute computation load effectively. When the ratio is higher, CWN may lose some of its edge. Techniques mentioned in the last paragraph will then be necessary.

In conclusion, although a better strategy, CWN still needs to be improved. Incorporating some of the features of the GM in CWN may overcome its drawbacks. Much research is needed to decide how to do that, because the space of possible strategies is very large.

Acknowledgement: I am grateful to Michael Carroll, Valerie Rasmussen, and Wennie Shu for their help with the simulations.

6. References

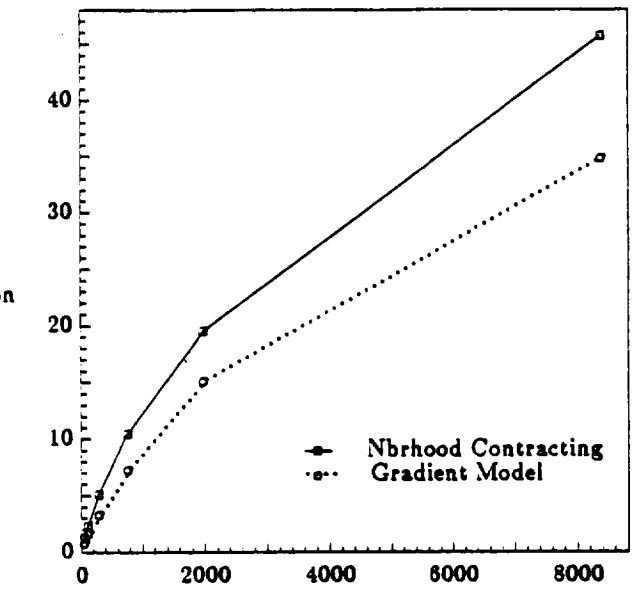
1. L. V. Kale, "Parallel Architectures for Problem Solving", Doctoral Thesis, Dept. of Computer Science, SUNY, Stony Brook, NY-11794., December 1985.

2. L. V. Kale, "Optimal Communication neighborhoods", Proc. of ICPP, St. Charles, Illinois, August 1986.

3. R. Keller and F. C. H. Lin, "Simulated Performance of a Reduction Based Multiprocessor", *Computer*, **17**, 7 (July 1984), .

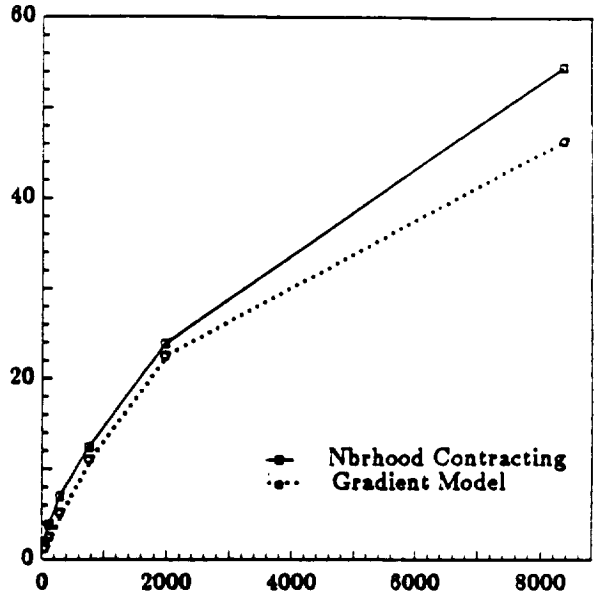
4. F. C. H. Lin, "Load Balancing and fault tolerance in applicative systems", Doctoral Thesis, Dept. of Computer Science, Univ. of Utah, August 1985.

Double Lattice–Mesh of 5 20 20
Query: Divide and Conquer



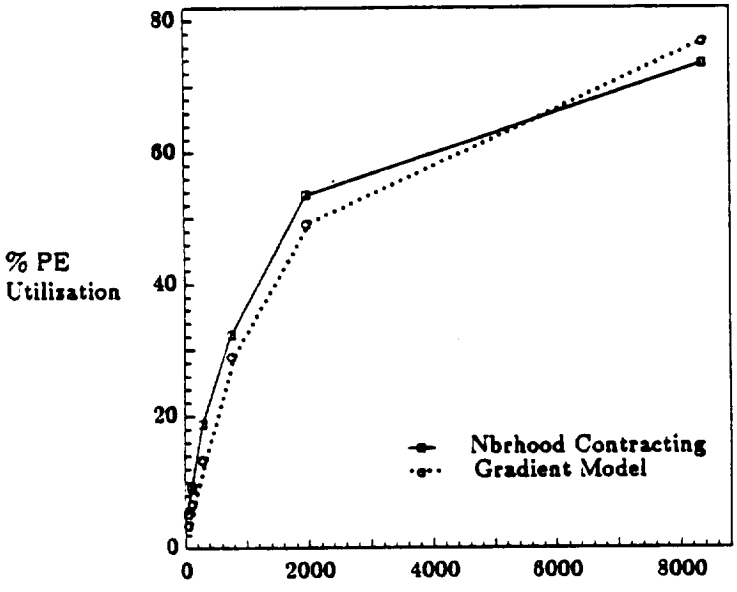
Plot 1 No. of Goals

Double Lattice–Mesh of 4 16 16
Query: Divide and Conquer



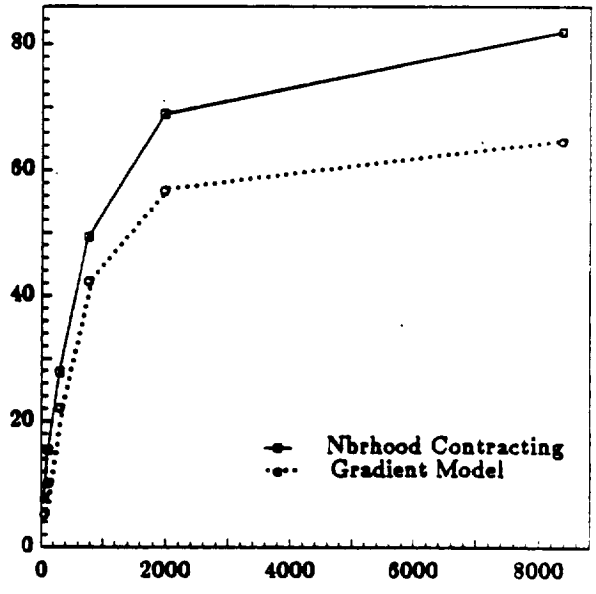
Plot 2 No. of Goals

Double Lattice–Mesh of 5 10 10
Query: Divide and Conquer



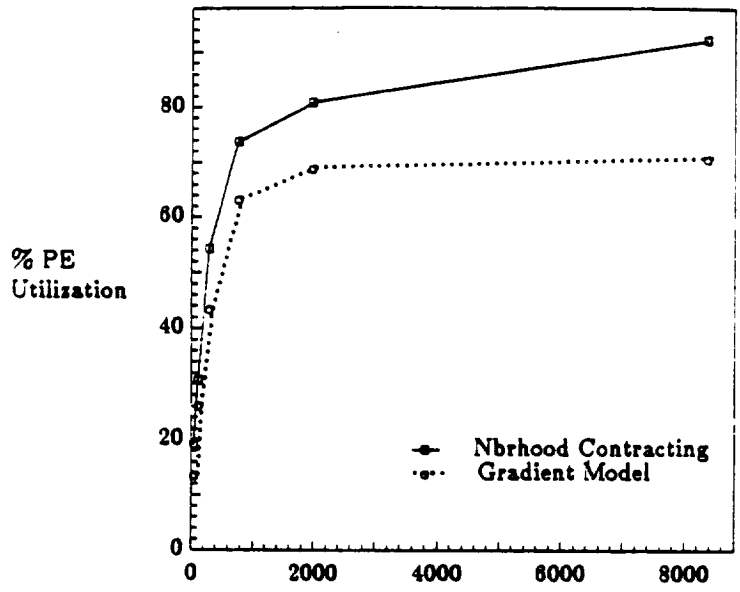
Plot 3 No. of Goals

Double Lattice–Mesh of 4 8 8
Query: Divide and Conquer



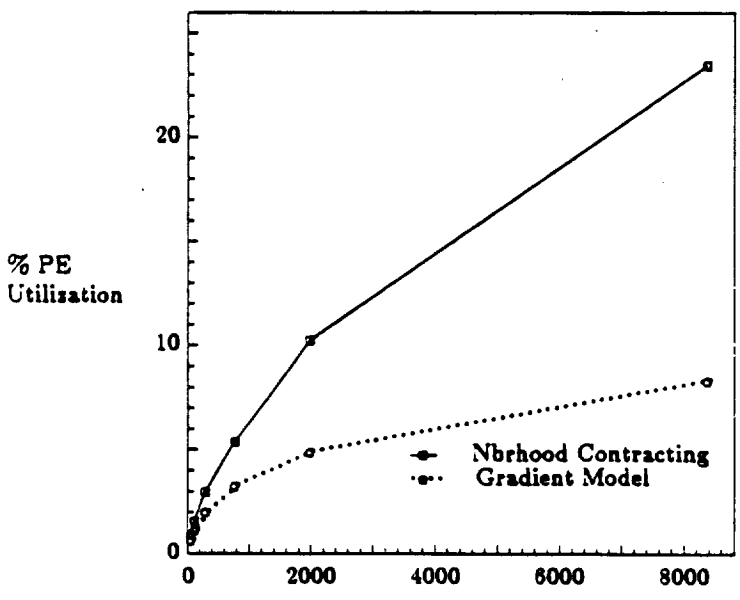
Plot 4 No. of Goals

Double Lattice-Mesh of 5 5 5
Query: Divide and Conquer



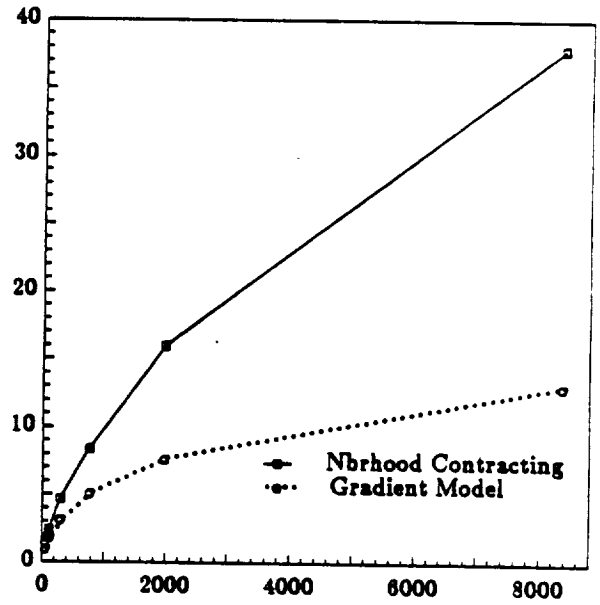
Plot 5 No. of Goals

Grid of 20 x 20
Query: Divide and Conquer



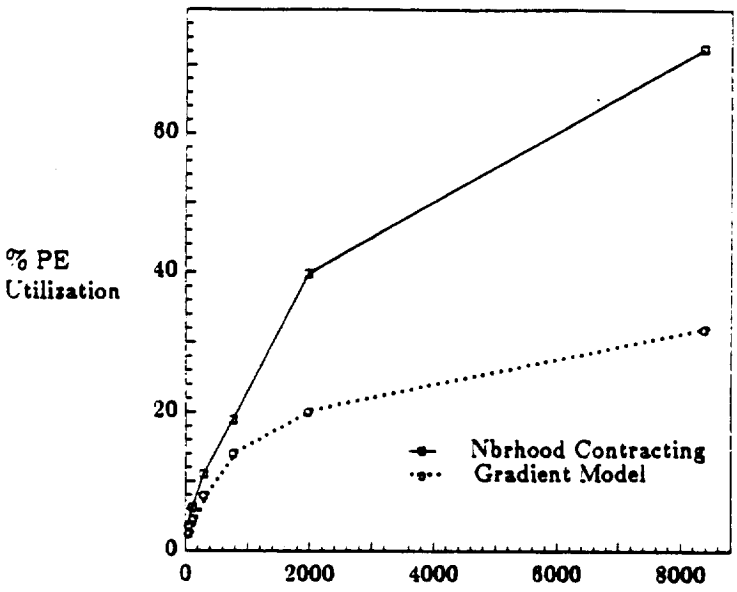
Plot 6 No. of Goals

Grid of 16 x 16
Query: Divide and Conquer



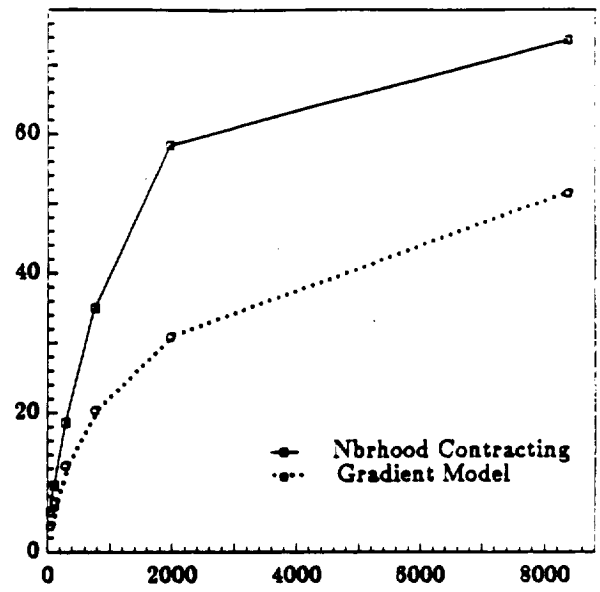
Plot 7 No. of Goals

Grid of 10 x 10
Query: Divide and Conquer



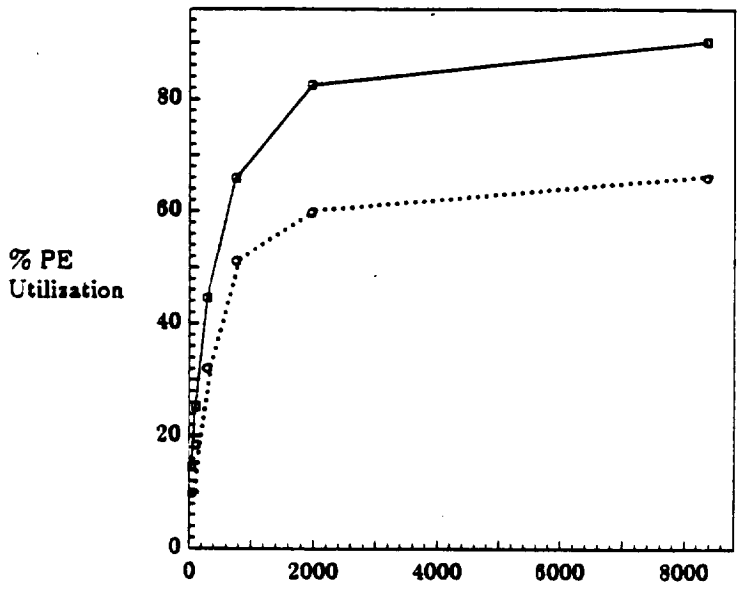
Plot 8 No. of Goals

Grid of 8 x 8
Query: Divide and Conquer



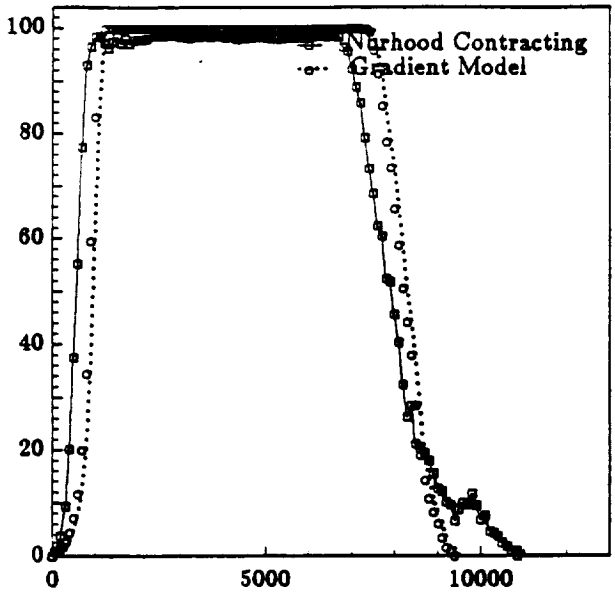
Plot 9 No. of Goals

Grid of 5 x 5
Query: Divide and Conquer



Plot 10 No. of Goals

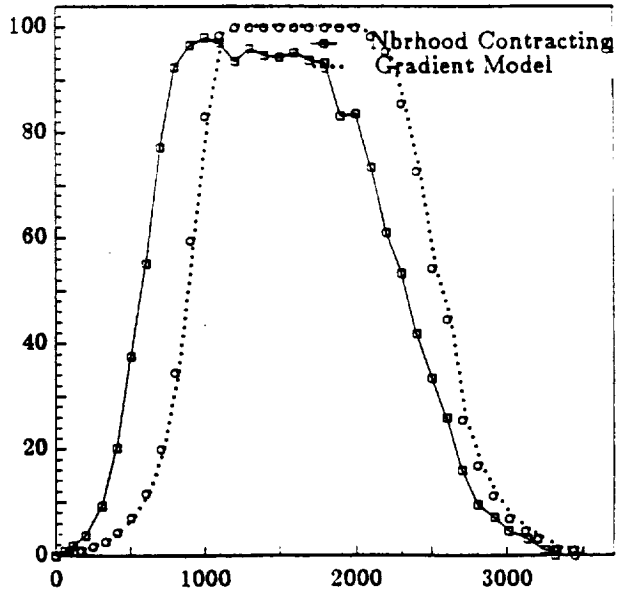
Double Lattice–Mesh of 5 10 10
Query: Fibonacci of 18



% PE
Utilization

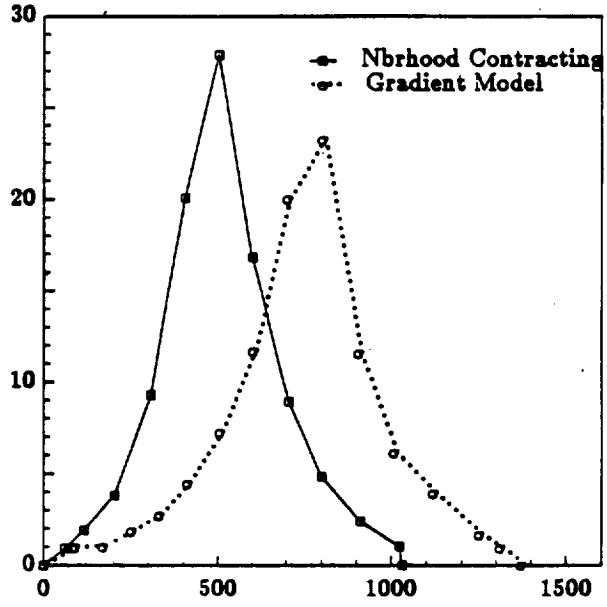
Plot 11 Time

Double Lattice–Mesh of 5 10 10
Query: Fibonacci of 15



Plot 12 Time

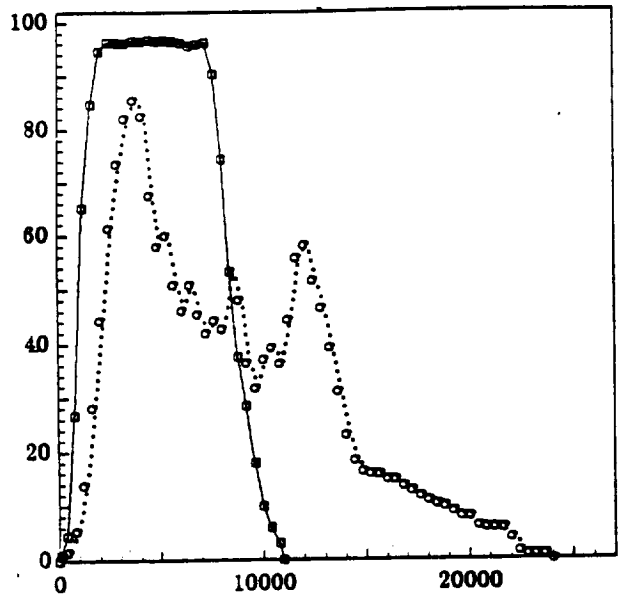
Double Lattice–Mesh of 5 10 10
Query: Fibonacci of 9



% PE
Utilisation

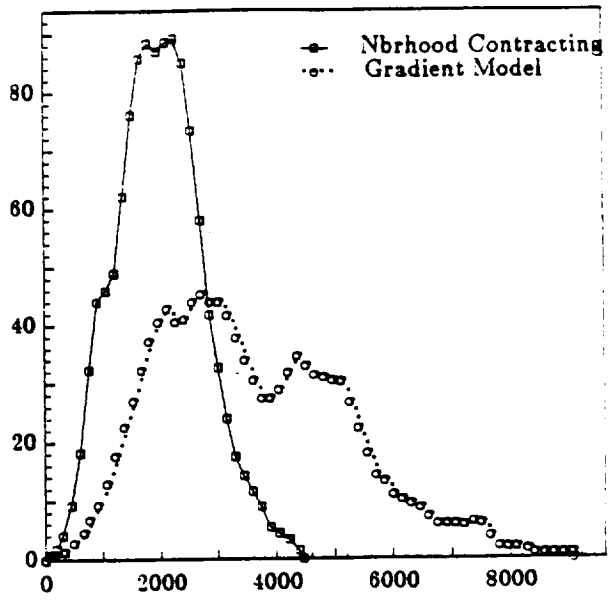
Plot 13 Time

Grid of 10 x 10
Query: Fibonacci of 18



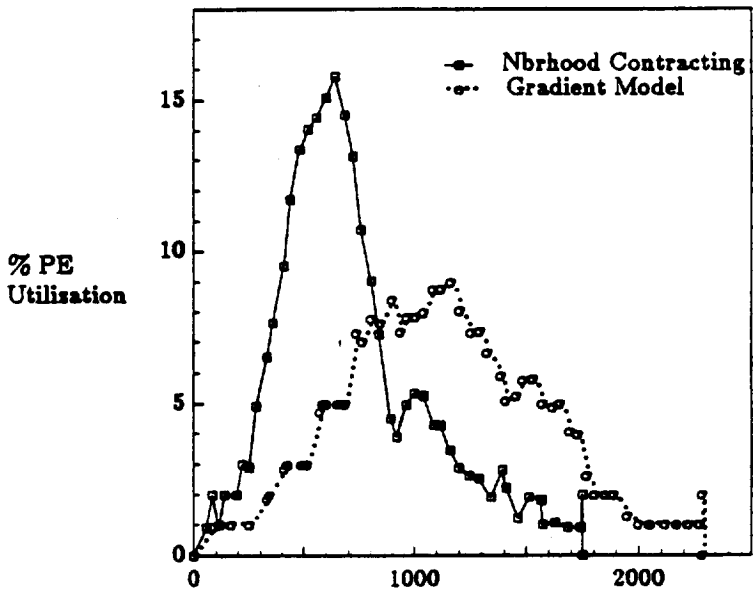
Plot 14 Time

Grid of 10 x 10
Query: Fibonacci of 15



Plot 15 Time

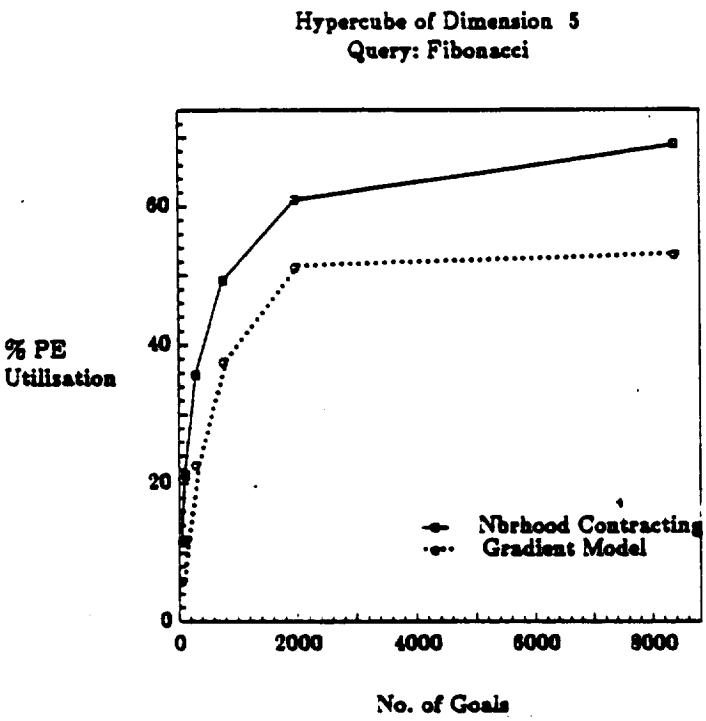
Grid of 10 x 10
Query: Fibonacci of 9



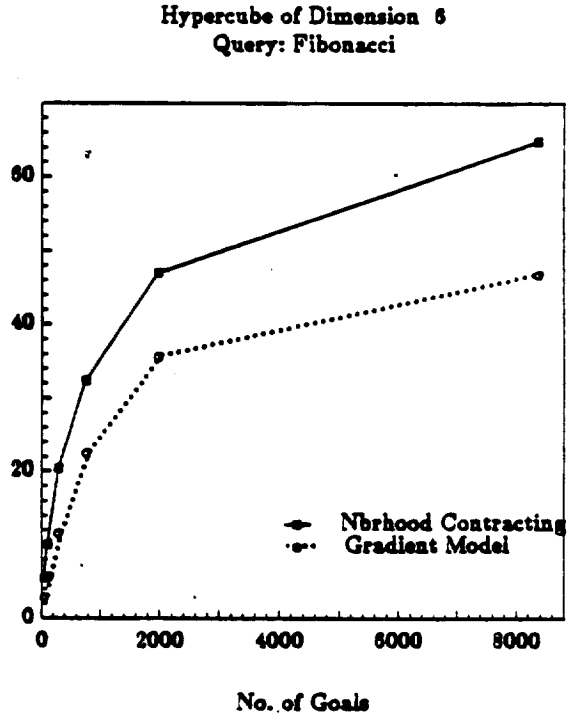
Plot 16 Time

Appendix I

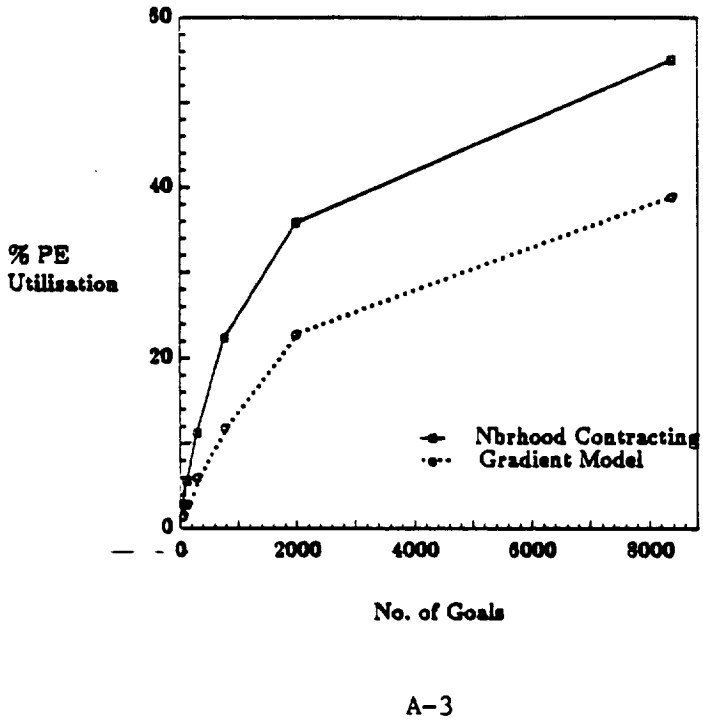
Simulation Experiments for the Hypercubes



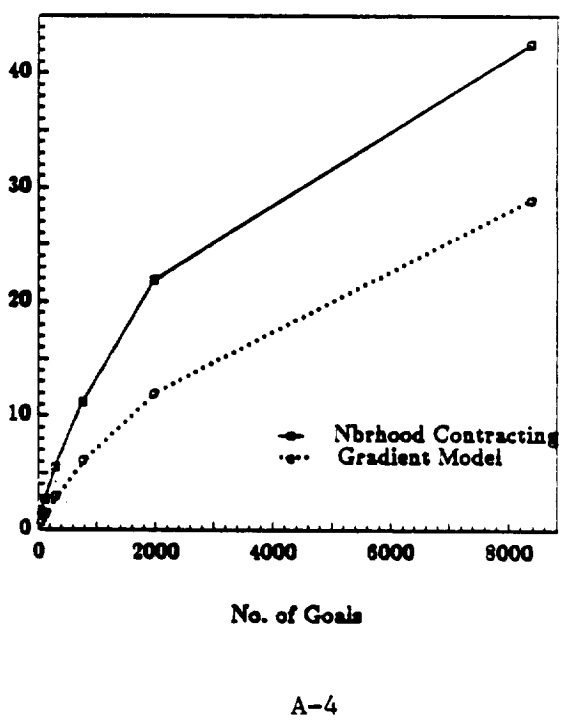
A-1



A-2

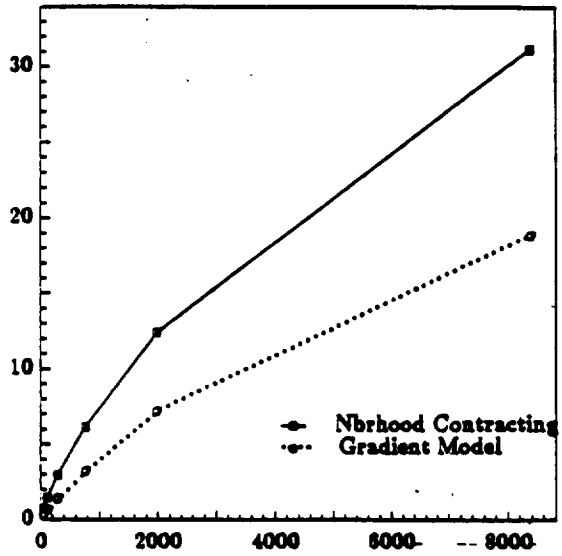


A-3



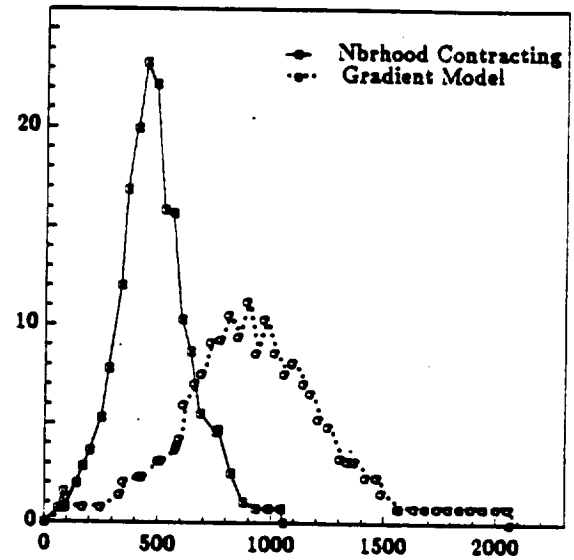
A-4

Hypercube of Dimension 9
Query: Fibonacci



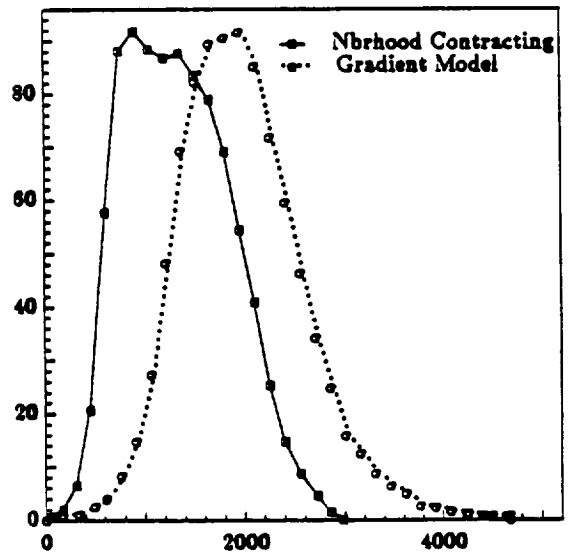
A-5

Hypercube of Dimension 7
Query: Fibonacci of 9



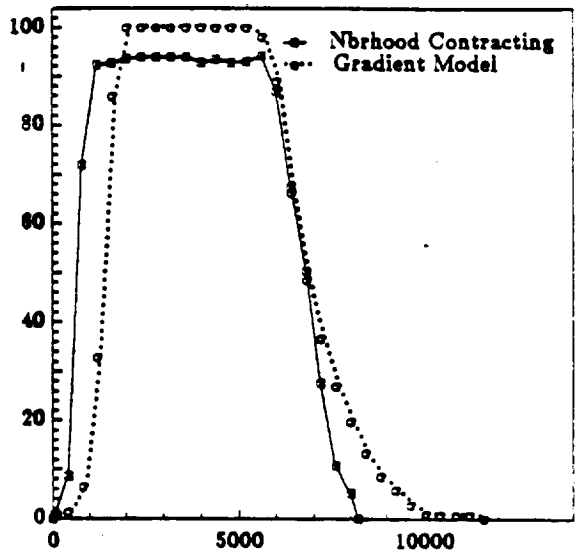
A-6

Hypercube of Dimension 7
Query: Fibonacci of 15



A-7

Hypercube of Dimension 7
Query: Fibonacci of 18



A-8

BIBLIOGRAPHIC DATA SHEET	1. Report No. UIUCDCS-R-87-1387	2	3. Recipient's Accession No.
4. Title and Subtitle Comparing the Performance of Two Dynamic Load Distribution Methods			5. Report Date November 11, 1987 6.
7. Author(s) L.V. Kale'			8. Performing Organization Rept. No.
9. Performing Organization Name and Address Department of Computer Science University of Illinois 1304 W. Springfield, Urbana, IL 61801			10. Project/Task/Work Unit No. 11. Contract/Grant No. CCR-87-00988
12. Sponsoring Organization Name and Address National Science Foundation Division of Computer and Computational Research Washington DC 20550			13. Type of Report & Period Covered 14.

15. Supplementary Notes

16. Abstracts

Parallel processing of symbolic computations on a message-passing multiprocessor presents one challenge: To effectively utilize the available processors, the load must be distributed uniformly to all the processors. However, the structure of these computations cannot be predicted in advance. So, static scheduling methods are not applicable. In this paper, we compare the performance of two dynamic, distributed load balancing methods with extensive simulation studies. The two schemes are the Contracting Within a Neighborhood (CWN) scheme proposed by us, and the Gradient Model proposed by Lin and Keller. We conclude that although simpler, the CWN is significantly more effective at distributing the work than the Gradient model.
This work was supported in part by a grant from NSF # CCR-87-00988

17. Key Words and Document Analysis. 17a. Descriptors

Parallel Processing
Dynamic Load Balancing
Symbolic Computations

17b. Identifiers/Open-Ended Terms

17c. COSATI Field/Group

18. Availability Statement	19. Security Class (This Report) UNCLASSIFIED	21. No. of Pages 24
	20. Security Class (This Page UNCLASSIFIED	22. Price