A Load Balancing Strategy For Prioritized Execution of Tasks*

Amitabh B. Sinha
Department of Computer Science
University of Illinois
Urbana, IL 61801
email: sinha@cs.uiuc.edu

Laxmikant V. Kalé
Department of Computer Science
University of Illinois
Urbana, IL 61801
email: kale@cs.uiuc.edu

Abstract

Load balancing is a critical factor in achieving optimal performance in parallel applications where tasks are created in a dynamic fashion. In many computations, such as state space search problems, tasks have priorities, and solutions to the computation may be achieved more efficiently if these priorities are adhered to in the parallel execution of the tasks. For such tasks, a load balancing scheme that only seeks to balance load, without balancing high priority tasks over the entire system, might result in the concentration of high priority tasks (even in a balanced-load environment) on a few processors, thereby leading to low priority work being done. In such situations a load balancing scheme is desired which would balance both load and high priority tasks over the system. In this paper, we describe the development of a more efficient prioritized load balancing strategy.

1 Introduction

Load balancing is a critical factor in achieving optimal performance in parallel applications where tasks are created in a dynamic fashion. In many computations, tasks have priorities, and solutions to the computation may be achieved more efficiently if these priorities are adhered to in the parallel execution of the tasks. This is particularly important for computations with a speculative component, such as state space search and branch&bound problems, where the order of execution of the tasks can determine the amount of computation that needs to be performed. In such computations, a solution strategy provides a prioritization of tasks such that executing tasks in the order of their priorities would minimize the amount of computation. A load balancing scheme that only seeks to balance load, without balancing high priority tasks over the system, might result in the concentration of high priority tasks (even in a balanced-load environment) on a few processors, thereby perhaps leading to low priority (and hence wasteful) work being done. This will lead to longer execution times and substantially higher memory requirements. In such situations a *prioritized* load balancing scheme is desired which would balance both load and high priority tasks over the system.

In this paper, we sketch the development of a load balancing strategy for the execution of prioritized tasks, particularly in the context where there are a large number of processors and an unbounded number of priority levels. This load balancing strategy is applicable to various applications such as branch&bound and state space search problems. We chose to evaluate the strategy with a branch&bound solution of the Traveling Salesman Problem. The implementation of the TSP application was carried out in a machine independent parallel programming system, Charm.

We describe Charm and the branch&bound TSP implementation in Section 2 and Section 3, respectively. In Section 4, we provide the motivation and basis for a new prioritized load balancing strategy. In Section 5, we describe the evolution of the load balancing strategy along with the results of performance evaluation experiments. Finally in Section 6, we review previous work on prioritized load balancing strategies and discuss future improvements to our load balancing strategy.

2 Programming Environment

Charm [1] is a machine independent parallel programming language. Programs written in Charm run unchanged on shared memory machines including Encore Multimax and Sequent Symmetry, nonshared memory machines including Intel i860 and NCUBE/2, UNIX based networks of workstations including a network of IBM RISC workstations, and any UNIX based uniprocessor machine.

The basic unit of computation in Charm is a *chare*.

*This research was supported in part by the National Science Foundation grants CCR-90-07195 and CCR-91-06608.

A chare's *definition* consists of a data area and entry functions that can access the data area. A chare *instance* can be created dynamically using the *CreateChare* system call. As a result of this system call, a *new-chare* message is created. Each chare instance has a unique address. Entry functions in a particular chare instance can be executed by addressing a message to the desired entry function of the chare. Messages can be addressed to existing chares using the *SendMsg* system call. This call generates *for-chare* messages.

In the Charm execution model, all *new-chare* and *for-chare* messages are deposited in a message-pool from where messages are picked up by processors whenever they become free. In the shared memory implementation of Charm, there is a single pool of messages shared by all processors; in the nonshared memory implementation, the message-pool is implemented in a distributed fashion with each processor having its own local message-pool. New-chare messages are the only messages that don't have a fixed destination, and are therefore the only messages which can be load balanced. In nonshared memory implementations, load balancing strategies attempt to balance the sizes of the local message-pools on each processor. New chare messages may move among the available processors under the control of a load balancing strategy till they are picked up for execution. Once picked up, a new chare message results in the creation of a new chare, which is subsequently anchored to that processor.

Charm provides the flexibility of linking user code with different load balancing and queuing strategies without having to make any changes to the code. Thus Charm provides a good test-bed for different load balancing and queuing strategies. Charm also provides a type of replicated process called a *branch-office chare*, and five efficient data sharing abstractions called *read-only*, *write-once*, *accumulator*, *monotonic* and *dynamic tables*. We refer the interested user to [1, 2] for details of Charm.

3 Application: Traveling Salesman Problem

The Traveling Salesman Problem (TSP) [3] is a typical example of an optimization problem solved using branch&bound techniques. In this problem a salesman must visit n cities, returning to the starting point, and is required to minimize the total cost of the trip. Every pair of cities i and j has a cost C_{ij} associated with them (if $i = j$, then C_{ij} is assumed to be of infinite cost).

We have implemented the branch&bound scheme proposed by Little, et. al. [4]. More sophisticated branch&bound schemes are available; since our focus is not on the best branch&bound scheme, but rather on an efficient prioritized load balancing strategy, Little's scheme is sufficient for our purpose. For a thorough discussion on branch&bound schemes and their parallelizations we refer you to [5, 6, 7, 8]. In Little's approach one starts with an initial partial solution, a cost function (C) and an infinite upper bound. A partial solution comprises a set of edges (pairs of cities) that have been included in the circuit, and a set of edges that have been excluded from the circuit. The cost function provides for each partial solution a lower bound on the cost of any solution found by extending the partial solution. The cost function is monotonic, i.e., if S_1 and S_2 are partial solutions and S_2 is obtained by extending S_1, then $C(S_1) <= C(S_2)$. Two new partial solutions are obtained from the current partial solution by including and excluding the "best" edge (determined using some selection criterion) not in the partial solution. A partial solution is discarded (pruned) if its lower bound is larger than the current upper bound. The upper bound is updated whenever a solution is reached.

Let all nodes with cost less than the cost of the best solution be called *useful* nodes, and all nodes with cost greater than the cost of the best solution be called *useless* nodes. In the execution of a branch&bound application, *useless* nodes need not be examined because such nodes cannot generate solutions better than the best solution. Even though we do not know the cost of the best solution at the outset, we can minimize the number of *useless* nodes examined, and hence the amount of wasteful (speculative) work, if we process the nodes in the order of their costs.

In the Charm implementation of the branch&bound solution of TSP, each partial solution is represented by a chare and the cost of the partial solution is the priority of the new chare message. A monotonic variable is used to maintain the upper bound.

4 The Basis for our Approach

In this section, we describe the basis for our approach towards the development of an efficient prioritized load balancing strategy. All speedups reported in this paper are with respect to the execution time needed to find the best solution on one processor.

Figure 1 shows results of execution runs of a 40-city TSP on a shared memory machine, the Sequent Symmetry. The information is presented in terms of speedups and the number of nodes (of the branch&bound tree) that are generated during the computation. A look at the figure shows that the number of branch&bound nodes generated remains almost constant in all the runs, and the speedups are close to linear.

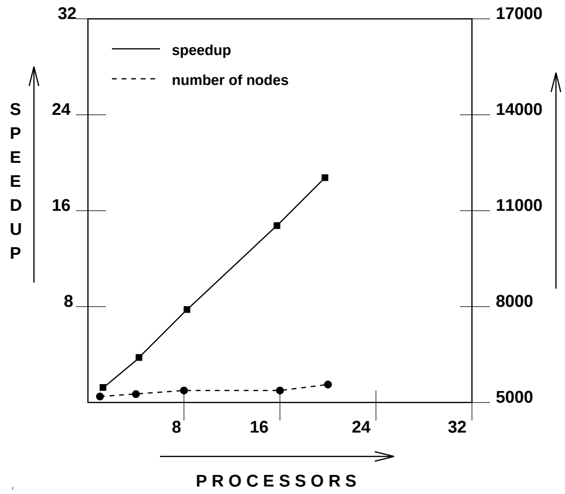

Figure 1: **The figure shows the speedups and the number of nodes generated for executions of an asymmetric 40 city TSP on the Sequent Symmetry.**

We have examined two existing load balancing strategies, ACWN (*adaptive contracting within a neighborhood*) and Random, to measure and understand performance of fully distributed strategies for prioritized execution of tasks. In the random load balancing strategy, new work is sent out to a random processor. In the ACWN strategy [9], newly generated work is required to travel between a minimum distance, *minHops*, and a maximum distance, *maxHops*. Work always travels to topologically adjacent neighbors with the least load, but only if the difference in loads between the two neighbors is more than some predefined quantity, *loadDelta*. The parameters, *minHops* and *maxHops*, can be dynamically altered. In addition ACWN does saturation control by classifying the system as being either lightly, moderately or heavily loaded.

Figures 2 and 3 show the results of the execution of a 40-city asymmetric TSP problem on an NCUBE/2 with a random and ACWN load balancing strategy, respectively.

Notice that we get nearly linear speedups in the case of the shared memory machine runs, while in the case of the nonshared memory machine runs (with either load balancing strategy) the speedups seem to saturate after 8 processors. With the aid of Projections [10], a performance tool developed for Charm, we were able to determine that the average busy time for each processor was 95% in the shared memory runs and 80% in the nonshared memory runs.

Why are the speedups good in the shared memory

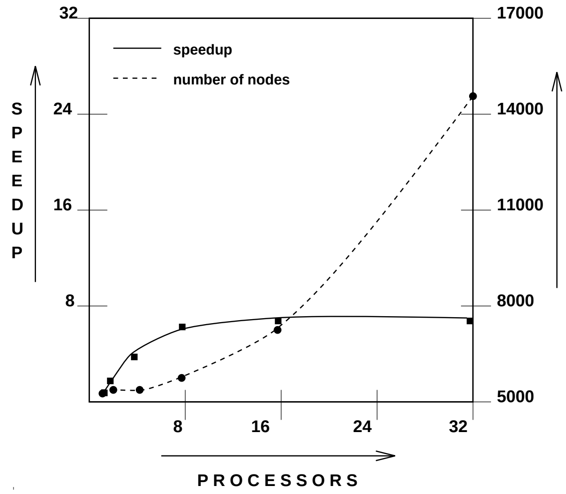

Figure 2: **The figure shows the speedups and number of nodes generated for a 40-city asymmetric TSP problem on the NCUBE/2 using a random load balancing strategy. New work is sent to a randomly chosen processor in the system.**

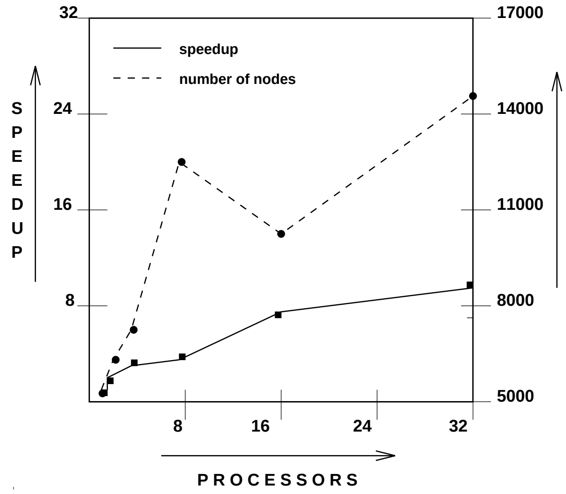

Figure 3: **The figure shows the speedups and number of nodes generated for an asymmetric TSP problem on the NCUBE/2 using the ACWN load balancing strategy.**

implementation? In the shared memory implementation, all processors share one priority queue of tasks. Therefore tasks are processed in the order of their priorities; consequently very little *useless* work is done, and the amount of speculative work is low. Since the total amount of work remains fairly constant even as the number of processors increase, and since all processors are busy 95% of the time the completion time is much faster.

Why are the speedups not good in the nonshared memory implementation? Since the average busy time for each processor is 80% we can eliminate longer idle times (as the number of processors grow) as a reason for poor speedups in the case of nonshared memory runs. In the nonshared memory implementation, tasks are distributed across all processors. Non-prioritized load balancing strategies do not balance priorities so that a lot of low priority messages (which may be pruned in an optimal execution) may get processed on some processors, while there are still high priority messages to be processed on other processors. This leads to a great deal of speculative work which manifests itself in the increase in the number of branch&bound nodes. In the case of both the random and the ACWN load balancing strategies, the number of nodes increases by almost 300%, and speedups were not good — even though there are more processors, there is more work (indicated by increase in number of nodes) to be done, hence the completion time does not decrease in proportion to the increase in the number of processors.

The above results indicate that in speculative computations it is important that nodes be processed in the order of their priorities. Any efficient prioritized load balancing strategy should be able to ensure, as far as possible, that the processing of tasks occurs in the global order of their priorities. A simple measure of how well the load balancing strategy follows the above criterion is the variance in the number of nodes created with the number of processors — the lesser the variance, better is the criterion being adhered to, and vice versa.

What should be the nature of a load balancing strategy so that tasks are processed in their global order of priorities? Our experience with the centralized queue for tasks in the shared memory model versus the completely distributed queues for tasks in the case of nonshared memory models (using random and ACWN load balancing strategies) suggests that a prioritized load balancing strategy would perform better if it balanced load and priorities between *partially distributed queues*.

5 Development of a Prioritized Load Balancing Strategy

In this section we outline the development of a prioritized load balancing strategy. The first step towards the development of a good prioritized load balancing scheme was a centralized load manager strategy. Clearly this strategy would not scale well — the load manager would be a bottleneck. However, implementing and experimenting with this strategy allowed us to confirm the validity of the criterion mentioned earlier, and to determine the modes in which the bottleneck occurs.

5.1 First Step: Load Manager Strategy

In this strategy one processor is chosen as the load manager, the remaining processors are its managees. Managees send all new work to the load manager. The load manager is responsible for the buffering of new work in a prioritized queue and assigning loads to each of its managees. A managee keeps the load manager informed about its load status in two ways — first, by periodically sending load information to the load manager, and second, by piggybacking load information onto each piece of new work sent to the load manager. The strategy that the load manager adopts in distributing load among its managees is to maintain the load on every managee within a minimum and maximum allowable load range. Whenever a load manager receives new load information about a managee, it sends it work only if the current load on the managee is less than the minimum load — we define the minimum acceptable load on a processor as the *leash* size. The leash is used to keep managees busy with work, while the manager sends it more work. We had expected that varying the leash size would make a difference. However in all our experiments any leash size of greater than 1 performed equally well. One of the reasons for this might be that the average granularity of work for the 40-city case is about 0.8 seconds, and this might be sufficient for the manager to receive a request for load and send work back to the managee.

Figure 4 shows the results of the execution of a 40-city TSP with the Load Manager strategy. Notice that the number of nodes have remained fairly constant, and the speedup is almost linear. The Load Manager strategy works well upto 32 processors, but its primary drawback is that it is not scalable to many more processors. In fact, it failed to run for the problem at hand for 64 processors. The failures were due to too many messages per unit time, which lead to an overflow of the message buffer.

This motivated the next stage in the development of a multi-level prioritized load balancing strategy:

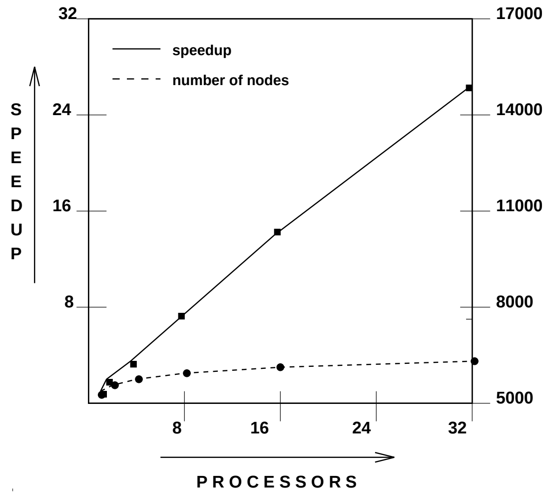

Figure 4: **The figure shows the speedups and the number of nodes generated for executions of a 40 city asymmetric TSP on the NCUBE/2 using the load manager strategy to balance load.**

the multiple managers strategy. Multi-level strategies have been studied before. Furuichi et. al. [11] present a strategy to partition the search of an OR-parallel graph in a distributed and hierarchical fashion among various processors — some processors function as sub-task generators and distribute the tasks among the remaining processors. One critical issue in their strategy is the generation of sub-tasks of reasonable granularity — if the grainsize is small, then the overheads of distributing would be substantial, if the grainsize is too large, then there might not be enough work to distribute. The model of computation in [11] is different from ours: in their model tasks are generated at and divided by only the task-generators, while in ours tasks can be generated at any managee. Ahmad and Ghafoor [12] have presented a semi-distributed strategy for task allocation for regular topologies, e.g., hypercubes as an alternative to completely centralized and completely distributed task allocation strategies. Neither of the above strategies take into account the additional factor of balancing priorities of tasks over processors.

5.2 Second Step: Multiple Managers Strategy

In the multiple managers strategy, the processors in the system are partitioned into clusters. One processor in each cluster is chosen as the load manager, the remaining processors in the cluster being its managees. Managees send all new work created on themselves to their corresponding load manager. Each load manager has two responsibilities:

1. It must distribute the work among its managees. As in the load manager strategy, the managees inform their load managers of their current work load by sending periodic load information and piggybacking load information with every piece of new work they send to the manager. The manager uses load information from its managees to maintain the load level within a certain range for all its managees.

2. It must balance both load and priorities over all the load managers in the system. This is accomplished by an exchange of high priority tasks between pairs of managers. Each manager communicates with a defined set of neighboring managers — in our implementation the managers were assigned positions in an n-dimensional cube, and the neighbor relation was defined as neighbors in the cube. An exchange of tasks between a pair of managers occurs in two steps. In the first step, the managers exchange their load information. In the second step each manager sends over some tasks to the other manager. Even if the loads are balanced, the managers exchange a fixed number of high priority tasks — this does the priority balancing. Further, if the loads are unbalanced, the manager with greater load sends to the manger with the lesser load additional tasks — this does the task-load balancing. Note that the tasks exchanged are the highest priority tasks on each manager. We have experimented with a strategy in which one half of the top priority tasks were exchanged, but this resulted in a degradation in performance, perhaps because of the cost of determining the top half elements. We can intuitively explain why exchanging the top priority tasks might be sufficient: the managees of each manager are already working on the top priority elements on their load managers. Therefore an exchange of work between managers causes a distribution of the top priorities between two managers and their managees.

Figure 5 shows the results of runs of a 40-city TSP with a multiple managers load balancing strategy. The speedups were good for 128 processors, but thereafter the number of nodes increased sharply, and the speedup remained unchanged. One of the reasons might be that there was not enough new work at the managers. In that case the priority balancing would not be effective causing expensive nodes to be processed early. In the example in Figure 5 approximately 7500 nodes were expanded for 128 processors,

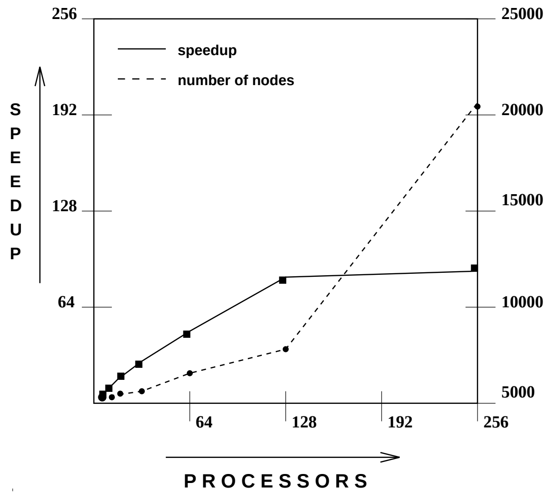

Figure 5: **The figure shows the speedups and the number of nodes generated for executions of a 40 city asymmetric TSP on the NCUBE/2 using the multiple managers strategy to balance load. In this case the cluster size is 8 processors.**

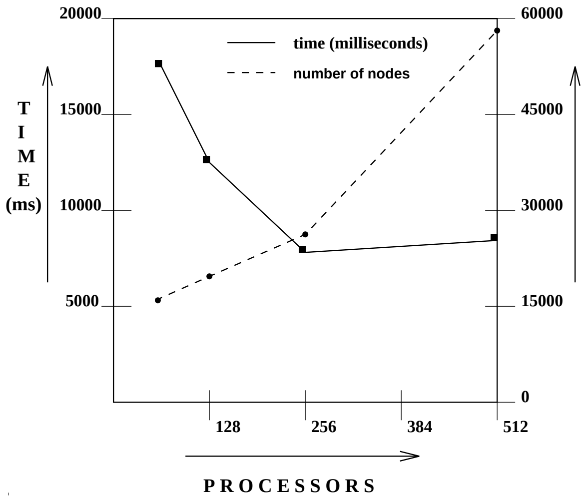

Figure 6: **The figure shows the speedups and the number of nodes generated for executions of a 50 city asymmetric TSP on the NCUBE/2 using the multiple managers strategy to balance load. In this case the cluster size is 16 processors.**

which works out to 50 nodes per processor for the 128 processor case and only 25 nodes per processor for the 256 processor case for the entire duration of the computation, which does seem to be a small number of nodes on each processor. In order to confirm our analysis we ran the TSP program for a larger problem size. The results for the 50-city run of the TSP are shown in Figure 6. Actual times are provided, instead of speedups, because the program did not run on a single processor due to insufficient memory. The results show better speedups till 256 processors and confirm our analysis.

The multiple managers strategy scales up reasonably well to 256 processors. Could we have used larger problems and obtained speed-ups for even more processors? Probably, yes. However, we ran out of memory for larger problem sizes. The reason for this is that there is an imbalance in the memory requirements of the load managers and the managees in the multiple managers strategy. The imbalance arises because all newly created work is queued up at the load managers. This poses problems because the amount of new work that can be created becomes limited by the number of managers and their available memory, even though there is a larger amount of memory available on the managees (assuming all processors in the system have equal amount of memory, and that there is more than one managee for each manager). Our final load balancing strategy attempts to balance the memory requirements of the load manager and the managees.

5.3 Third Step: Token Strategy

The token strategy is very similar to the multiple managers strategy. The processing elements in the system are split up into clusters — one processor in each cluster is chosen as the load manager, the remaining processors are its managees. New work created on managees is stored in hash-tables on the processor itself, while a token containing the priority of the new work is sent to the load managers. The load managers balance tokens and priorities among themselves by exchanging their high priority tokens — a fixed number of tokens is always exchanged to accomplish priority balancing, while some more tokens may by exchanged to balance the number of tokens on the load managers. Each managee informs its manager of its load by (1) piggybacking load information with each token it sends to the manager, and (2) periodically sending load information. When a manager decides that one of its managees (say M) needs work, it selects the highest priority token on it, and sends a request to the processor storing the work corresponding to the token asking for the work to be sent to M.

There were two problems that we anticipated with the token strategy:

1. In the manager and the multiple managers strategies, new work traveled two hops — one hop for the work to be sent to the manager and another hop for the work to be sent from the manager to a managee (ignoring the number of hops a message might take because of load balancing). In

the token strategy, each piece of new work causes three hops — one hop for the token to be sent to the manager, one hop for the manager to send managee a request to send work, and a third hop for the work to be sent.

2. There may be a delay in a processor responding to a request to forward work it owns because it may be busy processing itself.

We had hoped that the second problem could be mitigated if we managed the leash size so that the managees had some work to do, while work was being sent to them — however, experimental results indicate that any leash size of greater than 1 performed equally well.

Figure 7 shows the execution times and the number of nodes generated for runs of a 50-city asymmetric TSP on the NCUBE/2 with the token strategy. The results are comparable to those obtained with the multiple managers strategy, though the multiple managers strategy performed better than the token strategy. The reason for the better performance of the multiple managers strategy is that each message can cause atleast three hops in the case of the token strategy compared to two hops for the multiple managers strategy. The token strategy, however, is superior when it comes to solving larger problems because it utilizes the available memory much more efficiently. In Section 6, we discuss future improvements to the token strategy.

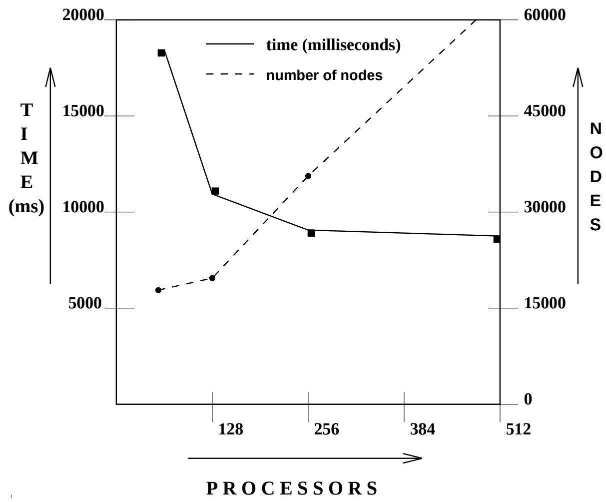

Figure 7: **The figure shows the execution times and the number of nodes generated for executions of a 50 city asymmetric TSP on the NCUBE/2 using the tokens strategy to balance load. In this case the cluster size is 16 processors.**

Figure 8 shows the execution times and the number of nodes generated for runs of a 60-city asymmetric TSP on the NCUBE/2 with the token strategy. Notice that the number of nodes generated in this case are fairly constant for up to 512 processors and the speedups are good.

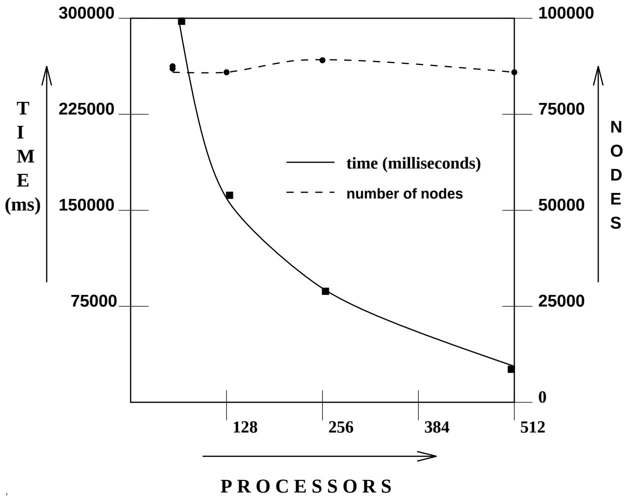

Figure 8: **The figure shows the execution times and the number of nodes generated for executions of a 60 city asymmetric TSP on the NCUBE/2 using the tokens strategy to balance load. In this case the cluster size is 16 processors.**

How good is the token strategy? The results of execution runs of the 60 city asymmetric TSP seem to indicate that the number of nodes created remains nearly constant with the number of processors. Is there any room for further improvement? In order to answer these questions we need to examine two quantities: the amount of wasteful work done and the fraction of time spent waiting for new work by each processor.

We have attempted to estimate the amount of wasteful work done by determining the distribution of nodes in terms of cost over time of the nodes created and processed in the execution runs of the 60-city TSP problem. Table 1 shows the number of *useful* and *useless* nodes created and processed for various stages of the program execution for two separate runs of the 60 city asymmetric TSP. In the first run, A, the initial upper bound is selected to be infinite, while in the second run, B, the initial upper bound is selected to be one unit greater than the cost of the best solution. In our implementation, we prune at creation all nodes with cost greater than the current upper bound. Since the initial upper bound is one unit greater than the cost of the best solution, no *useless* nodes are created in run B. Note that in Table 1 the number of *useless* nodes created in Run B are more than zero. This is because we have counted nodes whose cost equals the cost of the best solution as *useless* nodes.

An examination of the distribution of the number of nodes processed before the best solution in case of Run A shows that the number of *useless* nodes processed are very few, and the number of *useful* nodes are substantially more. This means that very little wasteful work is done before the best solution is found. However, the number of *useless* nodes processed after the best solution is found is considerable — is this wasteful work? The answer is no, because these nodes were created before the solution was found when the upper bound in this case was infinite, and they are simply being picked up and discarded after the best solution was found. This analysis is confirmed if we repeat the above run with an initial upper bound which is one greater than the cost of the best solution. In this case (Run B), the distribution of costs of nodes processed before the solution was found look very similar to the results in Run A. However, the number of *useless* nodes processed after the solution was found are substantially less than the corresponding number in Run A — much fewer *useless* nodes are created in Run B, because our implementation prunes all *useless* nodes at creation time. This reduction manifests itself in a faster finish time, though the time to find the first solution remains virtually unchanged. The time spent in wasteful work before the best solution is found is a small fraction ($< 1\%$) of the time spent in doing useful work. The finish time could be improved with more efficient methods to discard *useless* nodes, but the strategy itself need not be changed.

The next quantity — fraction of time spent in waiting for work by processors — was determined to be an average of less than 6% for each one of the managees.

These two quantities indicate that the token strategy performs fairly well, though some small improvement might be possible.

6 Conclusions & Future Work

The execution of prioritized tasks in parallel presents unique load balancing problems — in addition to balancing the tasks amongst processors, it is also essential to balance priorities. Executions of a branch&bound application on a shared machine and with existing load balancing strategies helped us establish that efficiency was critically dependent on the order of execution of the prioritized tasks — the more closely it followed the global order of priorities, the lesser the wasteful work done, and hence the better the performance. This motivated the development of a prioritized load balancing strategy. We have developed a token load balancing strategy that takes care of (1) load imbalances, (2) balances priorities by centralizing queues, and (3) balances memory requirements of processors by using tokens.

The work in this paper is an extension of a paper presented at a workshop on dynamic object placement and load balancing at ECOOP'92 [13]. A recent paper by Saletore [14] present strategies similar to the manager and the multiple manager strategies presented in this paper. However they present results for upto 32 processors only, for which we have found the manager scheme to be sufficient. Prioritized ACWN schemes have been discussed in [15]. However the results are available only till 16 processors, and hence a comparison with our strategy was not possible.

In Section 5.3 we saw that the multiple managers strategy out-performs the token strategy for certain problems. We attribute the superior performance of the multiple managers strategy to the fewer hops taken by each message. Some of the delays due to extra message hop in the token strategy can be reduced by caching work corresponding to the best tokens on the managers. We would to like to investigate the effect of caching high priority nodes on the managers on the performance of the token strategy.

7 Acknowledgements

We would like to thank the Sandia National Laboratories for use of their NCUBE. We would also like to thank the referees for useful comments.

References

[1] L. V. Kale. The Chare Kernel Parallel Programming System. In *International Conference on Parallel Processing*, August 1990.

[2] L. V. Kale *et. al.* The Chare Kernel Programming Language Manual (Internal Report).

[3] Edward W. Reingold, Jurg Nievergelt, and Narsingh Deo. *Combinatorial Algorithms: Theory and Practice*. Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1977.

[4] J. D. C. Little, K. G. Murty, D. W. Sweeney, and C. Karel. An algorithm for the traveling salesman problem. *Operations Research*, 11:972–989, 1963.

[5] M. Bellmore and G. Nemhauser. The traveling salesman problem: a survey. *Operations Research*, 16:538–558, 1968.

[6] M. Held and R. Karp. The traveling salesman problem and minimum spanning trees. *Operations Research*, 18:1138–1162, 1970.

Nodes Created				Nodes Processed			
Before Soln.		**After Soln.**		**Before Soln.**		**After Soln.**	
Useful	**Useless**	**Useful**	**Useless**	**Useful**	**Useless**	**Useful**	**Useless**
43259	40659	546	1236	41788	141	2094	41748

Initial Upper Bound: INFINITY (999999)
Time first solution was found: 118424 ms
Time finished: 155809 ms

(A)

Nodes Created				Nodes Processed			
Before Soln.		**After Soln.**		**Before Soln.**		**After Soln.**	
Useful	**Useless**	**Useful**	**Useless**	**Useful**	**Useless**	**Useful**	**Useless**
43585	6025	371	242	42081	3	1880	6260

Initial Upper Bound: 826
Time first solution was found: 117806 ms
Time finished: 124831 ms

(B)

Table 1: **This table shows the number of useful/useless nodes created/processed, before/after the best solution was found. In Run A, the initial upper bound is infinite, while in Run B the initial upper bound is one unit greater than the cost of the best solution. In our solution, we prune at creation all nodes with cost greater than the current upper bound. Hence, no *useless* nodes are created in Run B.**

[7] B. W. Wah, G. Li, and C. Yu. Multiprocessing of combinatorial search problems. In V. Kumar, P. S. Gopalakrishnan, and L. N. Kamal, editors, *Parallel Algorithms for Machine Intelligence and Vision.* Springer-Verlag, 1990.

[8] B. Monien and O. Vornberger. Parallel processing of combinatorial search trees. *Proceedings International Workshop on Parallel Algorithms and Architectures*, Math. Research Nr. 38, Akadmie-Verlag, Berlin, 1987.

[9] W. Shu and L. V. Kale. Dynamic scheduling of medium-grained processes on multicomputers. Technical report, University of Illinois, Urbana, 1989.

[10] L. V. Kale and A. B. Sinha. Projections: a scalable performance tool. In *International Parallel Processing Symposium*, April 1993.

[11] M. Furuichi, K. Taki, and N. Ichiyoshi. A multilevel load balancing scheme for or-parallel exhaustive search programs on the multi-psi. *Second ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 1990.

[12] I. Ahmad and A. Ghafoor. A semi distributed allocation strategy for large hypercube supercomputers. *Supercomputing*, 1990.

[13] A. B. Sinha and L. V. Kale. A load balancing strategy for prioritized execution of tasks. In *Workshop on Dynamic Object Placement and Load Balancing in Parallel and Distributed Systems (in co-operation with ECOOP '92)*, June 1992.

[14] Vikram A. Saletore and Mannan A. Mohammed. Hierarchical load balancing schemes for branch-and-bound computations on distributed memory machines. In *Hawaii International Conference on System Software*, January 1993.

[15] V. Saletore . *Machine Independent Parallel Execution of Speculative Computations.* PhD thesis, University of Illinois, Urbana, September, 1990.