A Load Balancing Strategy For Prioritized Execution of Tasks*

Amitabh B. Sinha **Laxmikant V. Kalé**
Department of Computer Science
University Of Illinois
Urbana, IL 61801

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. For such tasks, 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 work (and hence wasteful) being done. The implications are: 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.

Although strategies for load balancing have been extensively studied, most of the work deals with load balancing of non-prioritized tasks. In this paper we sketch the development of a load balancing strategy for the execution of prioritized tasks. This 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 was carried out in a machine independent parallel programming system, Charm. In Section 3, we establish the criterion for a good prioritized load balancing strategy using results of the runs of the application on shared memory machines and the performance of existing load balancing strategies on nonshared memory machines. We have used this analysis to develop a more efficient prioritized load balancing strategy, which is described in Section 4, alongwith the results of the performance evaluation experiments.

2 Application & Programming Environment

The Traveling Salesman Problem (TSP) [1] is a typical example of an optimization problem solved using branch&bound techniques. In this problem the 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 have a cost C_{ij} associated with them (if $i == j$, then C_{ij} is assumed to be of infinite cost).

In the branch&bound approach one starts with an initial partial solution, and an infinite upper bound. New partial solutions are generated by *branching* out from the current partial solution. Each 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. For every partial solution a lower bound on the cost of any solution,

*This research was supported in part by the National Science Foundation grant CCR-90-07195.

that can be found by extending the partial solution is computed. A partial solution is discarded (pruned) if its lower bound is larger than the current upper bound. 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. The upper bound is updated whenever a solution is reached. In our implementation of TSP, we have chosen to implement the branch&bound scheme proposed by Little, et.al [2]. Much better 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 purposes. For a thorough discussion on branch&bound schemes and their parallelizations we refer you to [3, 4, 5, 6].

The application was programmed using Charm. Charm [7] is a machine independent parallel programming language that currently runs 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*. A chare, created dynamically, has associated with it a data area, and some entry functions that can access this data area. The entry functions can be executed by addressing a message to an entry function of a particular chare, which can be uniquely identified with its *chareid*, a system defined value.

A new chare is created using the *CreateChare* system call. As a result of this system call, a *new-chare* message is created, which is at some later point of time picked up for execution by the system. New chare messages may float 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. Messages can be addressed to existing chares using the *SendMsg* system call. This call generates *for-chare* messages.

Charm also provides a type of replicated process called a *branch-office chare*, and five efficient data sharing abstractions *read-only*, *write-once*, *accumulator*, *monotonic* and *dynamic tables*. We refer the interested user to [7, 8] for details of Charm.

New work messages (new-chare and for-chare) are queued up before being picked up for execution. New-chare messages are the only messages having no fixed destination, and therefore are the only messages which can be load balanced. The Charm implementation on shared machines has one queue for new-chare messages shared by all the processors. On nonshared machines, new chare creation requests are queued up in local queues, and a load balancing strategy attempts to balance the sizes of these queues. Charm provides the flexbility of linking user code with different load balancing and queuing strategies without having to make any changes to the code. This allows for convenient testing of different load balancing queuing strategies. In all our execution runs a prioritized queueing strategy has been used for new-work messages.

3 Criterion for a Good Prioritized Load Balancing Strategy

In this section, we establish the criterion for an efficient prioritized load balancing strategy. The analysis to determine such a criterion was carried out in two parts: performance of the application on shared memory machines (where all processors shared one priority queue of tasks), and the performance of the application on nonshared memory machines with existing load balancing strategies.

Figure 1 shows results of execution runs of the TSP program on a shared memory machine, the Sequent Symmetry. The information is presented in terms of speedup with respect to the version of the program running on 1 processor, 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.

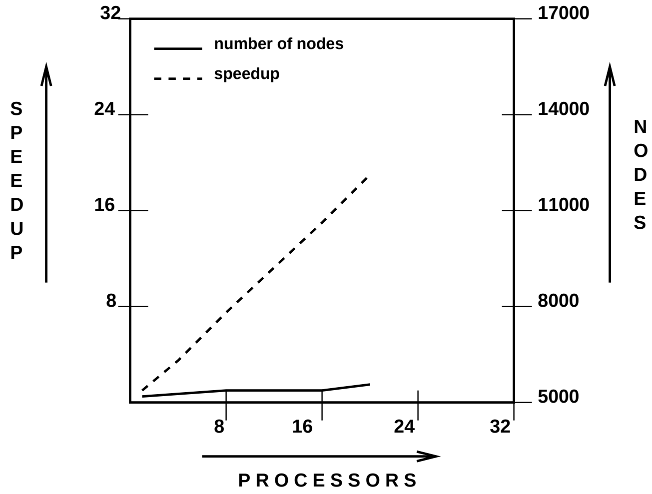

Figure 1: **The figure shows the speedups and the number of nodes generated for executions of an asymmetric 40 city TSP on the Sequent Symmetry.**

We have examined many existing load balancing strategies, e.g., ACWN and Random, to measure and understand performance of these strategies for prioritized execution of tasks. We shall summarize our observations and conclusions for the ACWN strategy.

In the *adaptive contracting within a neighborhood* (ACWM) [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 less than some predefined quantity, *loadDelta*. The parameters, *minHops* and *maxHops*, can be dynamically altered. In addition ACWM does saturation control by classifying the system as being either lightly, moderately or heavily loaded. Figure 2 shows the results of the execution of TSP with an ACWN load balancing strategy on an NCUBE/2.

The important thing to note is the relationship of speedup to the number of branch&bound nodes generated. In the case of the shared memory runs, the number of nodes generated remained nearly constant and speedups were close to linear. In the case of the ACWN load balancing strategy, the number of nodes increases enormously, and speedups are not good. The number of nodes increases substantially in the case of ACWN, because the global order of priorities is not maintained, hence low priority messages (which may be pruned in an optimal implementation) may get processed on some processors, while there are still high priority messages to be processed on other processors. A good load balancing strategy for prioritized task execution would need to maintain as closely as possible the global order of priorities — this, then is the criterion required of a good prioritized load balancing strategy.

Although prioritized ACWM schemes have been discussed in [10], the results are available only till 16 processors, and hence a comparison with our strategy was not possible.

4 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.

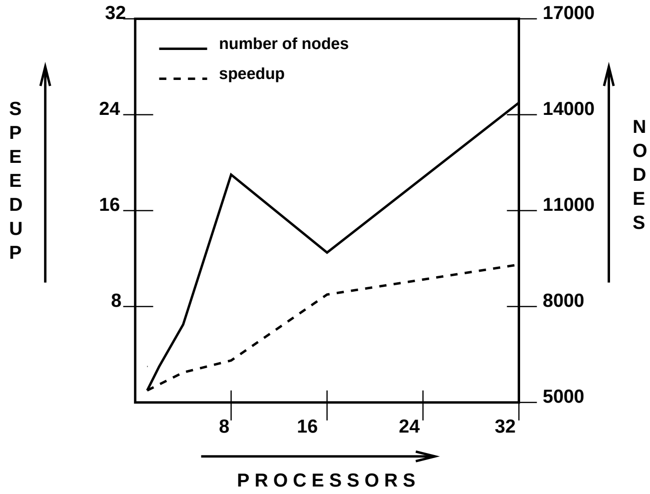

Figure 2: **The figure shows the speedups and number of nodes generated for an asymmetric TSP problem on the NCUBE/2 using the ACWM load balancing strategy.**

4.1 First Step: Load Manager Strategy

In this strategy one processor is chosen as the load manager. Every other processor sends all new chare creation requests to the load manager. The load manager is responsible for the buffering of new chare creation messages and assigning loads to each processor. A processor 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 every new chare message sent to the load manager.

New chare messages arriving at the load manager are buffered in a priority queue. The strategy that the load manager adopts in distributing load among its neighbors is to maintain the load on every client within a minimum and maximum allowable load range. Whenever a load manager receives new load information about a client, it sends it work only if the current load on the client is less than the minimum allowed load.

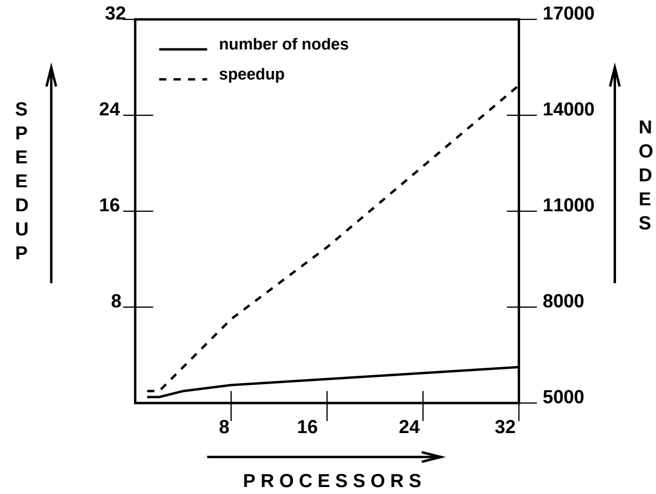

Figure 3: **The figure shows the speedups and the number of nodes generated for executions of a 40 city asymmetric TSP on the NCUBE/2 using the load manager strategy to balance load.**

Figure 3 shows the results of the execution of the TSP program 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 two separate reasons: too many messages per unit time (leading to an increase in time to process a message and message buffer overflow) and excessive memory requirement for the load manager. This motivated the next stage in the development of a prioritized load balancing strategy: the **hierarchical strategy**. Hierarchical strategies for load balancing have been discussed in [11, 12], but these strategies have focussed on load balancing for non-prioritized tasks.

4.2 Second Step: Hierarchical Strategy

In the hierarchical strategy, the processors in the system are partitioned into clusters, each cluster having its own load manager. All new work on a processor is sent to its corresponding load manager. It is the duty of the load manager to distribute this work among the processors in its cluster. In addition, the load managers must balance both load and priorities over all the load managers in the system. This is accomplished by exchange of some high priority tasks between pairs of processors (each processor communicates with a defined set of neighboring processors; in our implementation this was done as a dimensional exchange). A fixed number of tasks are always exchanged — this does the priority balancing. In addition, more tasks depending on the task-loads of the processors involved are exchanged — this does the load balancing.

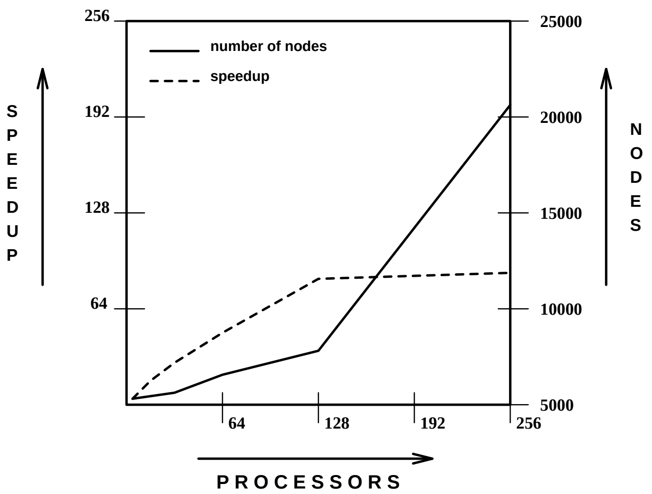

Figure 4: **The figure shows the speedups and the number of nodes generated for executions of a 40 city asymmetric TSP on the NCUBE/2 using the hierarchical strategy to balance load. In this case the cluster size is 8 processors.**

Figure 4 shows the results of runs of the TSP implementation with a hierarchical load balancing strategy. The speedups were good for 128 processors, but thereafter the number of nodes jumped sharply, and the speedup remained unchanged. One of the reasons might be that there was not enough new work centralized at the managers — approximately 7500 nodes were expanded for 128 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. In order to confirm this 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 5. Actual times are provided, 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. We are in the process of further improving this strategy, and expect to be able to present better results.

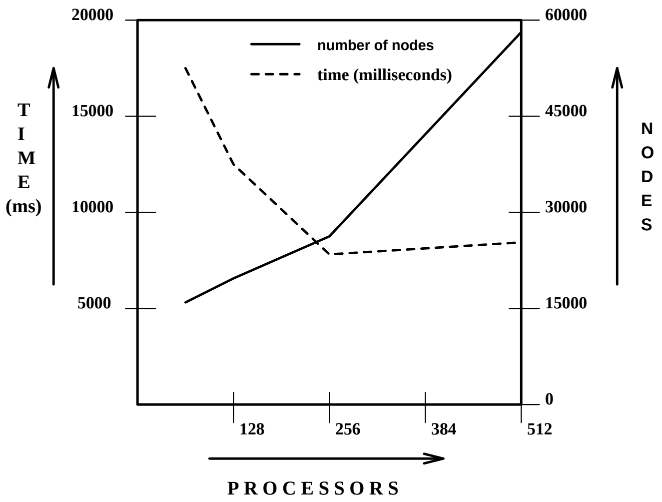

Figure 5: **The figure shows the speedups and the number of nodes generated for executions of a 50 city asymmetric TSP on the NCUBE/2 using the hierarchical strategy to balance load. In this case the cluster size is 16 processors.**

5 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 using some existing load balancing strategies helped us to 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 work done, and hence the better the performance. This motivated the development of a more centralized load balancing strategy. We have found this strategy to perform well upto 128 processors.

A hierarchical strategy poses problems of memory, because the memory requirement of the manager processors are much greater than those of the non-manager processors. We would like to explore the possibility of avoiding this memory bottleneck by balancing memory requirements of all processors.

References

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

[2] 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.

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

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

[5] 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.

[6] 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.

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

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

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

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

[11] M. Furuichi, K. Taki, and N. Ichiyoshi. A multi-level 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.