Throughput Improvement Through Dynamic Load Balance

Hemant B. More and Jie Wu
Department of Computer Science and Engineering
Florida Atlantic University
Boca Raton, FL 33431

Abstract

Dynamic load balance improves the performance of a multiprocessor system by reallocating tasks such that all the processors are evenly loaded. Problems in several areas qualify for dynamic load balance. This paper studies the performance of load balancing while solving a branch and bound problem using hypercubes. If the hypercube has link faults, special measures need to be taken to balance the load. An algorithm for load balancing in the presence of link faults is described. Performance improvement obtained with the help of load balancing using the link-fault-tolerant algorithm is observed through simulation.

1 Introduction

Improvement in the speed with which computations can be performed is very desirable. Parallel processing seeks to do this by employing several processors and breaking the tasks into smaller subtasks. The processors co-operate with each other, share information and complete the tasks while reducing the execution time by several orders of magnitude. The advances in VLSI technology have helped in creating cheap, mass producible computers featuring hundreds and thousands of processors.

Although improving the hardware leads to faster performance, the full exploitation of the power of a multicomputer system needs optimization in the processor usage and task allocation. It is frequently observed in a multiprocessing system that some processors are heavily loaded while others are almost idle. To alleviate these situations, several algorithms exist which improve system utilization by load balancing. A survey of such algorithms is found in [7]. Several researchers have studied this problem [1, 2, 3, 4, 6, 8, 9, 12, 13].

The computational intensity of NP-complete search problem solutions increases with the number of variables. Such problems can be solved heuristically. This paper solves one optimization problem using branch and bound on simulated hypercubes. The arrangement and properties of hypercube systems are described in detail in [5]. The optimization problem is solved here with and without load balance and the gain in improvement with load balance is observed. When the hypercube system has link faults, load cannot be balanced completely with the help of the *dimension exchange* algorithm. A modification to this algorithm is necessary to get a complete balance. We describe and use a load balance algorithm which tolerates link faults in hypercubes. Performance improvement obtained in the branch and bound problem by using this algorithm is observed.

2 Load Balancing Strategies

Different algorithms can be used to balance load on a multiprocessing system based on the topology of the system and other most influencing factors such communication time, migration decision time, load estimation time, number of processors, job arrival rate. The multiprocessor system is defined to be balanced when the load on any pair of processors is same or differs by 1. For hypercube systems, a commonly used method is termed as *dimension exchange*.

2.1 Dimension Exchange

The dimension exchange algorithm balances load on hypercubes by exchanging loads between processors along each dimension. It takes n rounds of load exchanges for an n dimensional hypercube.

The algorithm steps can be described as:

1. Repeat the following step for all the dimensions of the hypercube.

2. Select a dimension d and balance every two neighboring processors a and a^d along that dimension.

0-7803-1797-1/94/$3.00 © 1994 IEEE

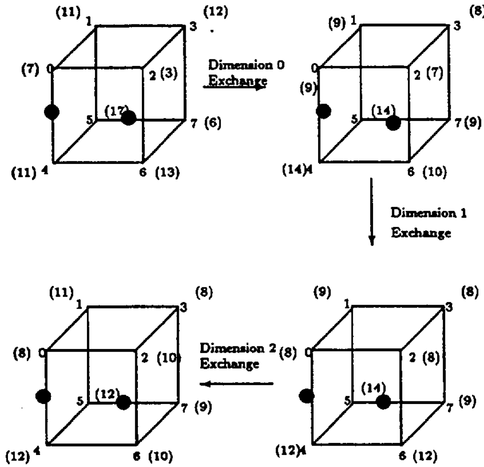

Figure 1: Dimension exchange with link faults

Let the processor having the higher load retain one unit more in case $L(a) + L(a^d)$ is odd, where $L(a)$ is the load of node a.

When the links of a hypercube are broken, the dimension exchange algorithm is not very effective. This can be seen from Figure 1 where the dimension exchange is used on a hypercube with 2 link faults. The numbers next to vertices in Figure 1 are processor IDs while the load on processors are shown in (and) brackets. Since the links between processor pairs {0, 4} and {5, 7} are faulty, dimension exchange results in incomplete balance.

Evidently, dimension exchange is not appropriate for load balancing in case of link faults. A new algorithm is necessary to tolerate the link faults. If a hypercube of n dimensions has n link faults, it is possible to have one processor completely disconnected. But $n-1$ faults ensure that every processor will have at least one healthy link. If a processor has less than average load, it is viewed here as a *hole* and if it has excessive load, it is called as *peg*.

2.2 Link-Fault-Tolerant Algorithm

We first overview a link-fault-tolerant load balancing algorithm proposed by Wu [11] which can tolerate upto $n-1$ link faults in an n dimensional hypercube. The complete description and extensions of this algorithm can be found in [11].

Let Q_n represent a hypercube, where n specifies its dimensions. The algorithm is as follows:

1. Select a dimension d across which all links are fault free. Break Q_n along d into two subcubes, Q_{n-1} and Q'_{n-1}.

2. Balance load across this dimension d between Q_{n-1} and Q'_{n-1}. That is, between each processor a along dimension d and its neighbor a^d, find which processor has more load. Send load units amounting to half of the difference between the two processors to the lightly loaded processor.

3. For the subcubes, apply the above steps recursively until balancing has been done along all dimensions or a fault free dimension cannot be found.

4. If any subcube Q_{n-x} does not have a fault free dimension, find the neighboring subcube Q'_{n-x} which is fault free and is balanced. Since the whole hypercube has only $n-1$ faults, Q'_{n-x} can readily be found.

5. For each processor a in subcube Q_{n-x} find load $L(a)$ and load $L(a')$ on neighboring processor a' in subcube Q'_{n-x}.

6. If $L(a) - L(a') > 0$, send that load as a *peg* to a'. If $L(a) - L(a') < 0$, send that quantity as a *hole* to a'. A hole signifies underloaded condition while peg symbolizes overload.

7. In subcube Q'_{n-x}, balance the pegs and holes using dimension exchange. While balancing the peg and holes, move appropriate amount of actual load among processors.

8. After Q'_{n-x} is balanced, for each processor, send load to its neighbor in Q_{n-x} if it had sent holes.

3 Problem Description

In this section, we describes application of the proposed method in solving branch and bound ($B\&B$) problems. The branching of $B\&B$ is performed by building a search tree over the problem space. The root of the tree represents the complete problem space while the subspaces are denoted by the children. The branching is done from the root to the leaves. This partitions the spaces into smaller subspaces. The leaves are subspaces that can be exhaustively searched for solutions. A subproblem P_i has an associated objective function f that defines the value of best solution obtainable from P_i. The value is unknown until

the subtree rooted at P_i is expanded. Therefore, another function h called as heuristic function is used to give the lower bound on the estimate of f. In general, h can be more easily found as compared to f. $B\&B$ uses selection, branching, elimination and termination tests as its steps. The heuristic h performs selection of order in which subproblems are expanded. Branching is used to break the current subproblem in small sized subproblems. The elimination step terminates the newly created subproblems which will not find better solutions than the currently found ones. A subproblem is deleted if its lower bound is greater than or equal to that of the best feasible solution.

A $B\&B$ problem solving strategy for solving the 0-1 ILP problem described in [5] is considered here. This optimization problem minimizes the values of an objective function $f(x_1, x_2..., x_n)$ subject to a set of constraints. The variables $(x_1, x_2..., x_n)$ can only take values 0 or 1. The problem is stated as:

Minimize

$$f = \sum_{j=1}^{n} c_j x_j$$

with the restriction

$$\sum_{i=1}^{m} a_{ij} x_j \geq b_i$$

where $1 \leq i \leq m, x_j \in \{0,1\}$ and $1 \leq j \leq n$ $c_j \geq 0$. The values in matrices a and b can be positive or negative.

The solution is to find such a sequence for x_j which would make the cross product of x and c a minimum quantity while also satisfying that cross product of x with all a_i is at least equal to or greater than b_i. This entails finding all combinations for the sequence of x and trying out which ones satisfy all the restrictions. The satisfying sequences would then be tested to get the minimum value for product with c. Using $B\&B$, this solution reduces the number of computations by keeping track of the current minimum. Whenever a particular product between a chosen x and c is more than the current minimum, the choice of x is abandoned. Secondly, whenever any one of the restriction is not satisfied, the further testing of restriction compliance is stopped.

This problem can be solved by multiple processor system like a hypercube with each processor getting some sequences of x to test. There can be a central processor giving test sequences to each processor and then later probing them for results. Another way is to link the choices of x to a processor number. For example, if there are 16 processors and 64 choices i.e. the x sequence is 6 bit long, each processor will take 4 test sequences. Node 0 takes 000000, 010000, 100000, 110000. The last four bits reflect the processor number while first two bits vary in order to produce all combinations. In this manner, all processors can determine their test sequence. The test sequence of x comprises of the load for a processor. For each operation like multiplication, addition and comparison, the processor will take finite amount of time. Using $B\&B$, a processor might finish all its inputs if they get eliminated or pursuing a sequence is of no interest. Such a processor will be idle while other processors might be busy. Load balancing can play a role here to prevent load imbalances.

4 Implementation

To solve the above problem, a simulation program is implemented. For a hypercube system, the simulation program generates the matrices a and b. The number of rows and columns of the matrices can be changed. The processors of the hypercube are assigned the test sequences. The global minimum is accessible to each processor (through a broadcast originated from the node that holds this number). Each processor first checks whether the product of its sequence with the c matrix produces a result less than the current minimum. Then it systematically tests whether the product of its sequence with each of the rows in a matrix is greater than or equal to the corresponding value in b matrix. Whenever the product is unsatisfactory, the whole sequence is abandoned. This reduces the load on that processor.

The longest time taken by any of the processors determines the effective time to solve the problem. Each problem is solved on hypercubes with 0 to $n-1$ link faults. The same problem is then solved using load balancing using the link-fault-tolerant algorithm. Load is balanced by sending parts of the problem to other processors. The length of the problem is determined by the number of rows in the a matrix. While migrating loads, the destination gets a new load with the description of starting and ending rows for a given test sequence. This migration results in some processors having to solve partial rows for a given sequence. If the processor is solving a partial sequence, it first tests whether that sequence was abandoned by some other processors working on another part of rows for the sequence. The status of the sequence is appropriately updated so that other processors working on the parts can know whether to pursue the partial sequence. The longest time to solve the problems using

such load balancing is measured.

5 Observations

The following parameters were systematically changed in the experiments: (a) The dimension of the hypercube. (b) The number of link faults on the hypercube. (c) The number of rows in the matrices b and c. (d) The number of columns in the matrices c and a.

A problem satisfying the parameters was generated each time and mapped on the hypercube with link faults. Then it was solved using the link-fault-tolerant algorithm. The longest time taken by any processor was regarded as the time taken to solve that problem. The speedup achieved over the case not using load balancing was measured for each setting of parameters. For each configuration of parameters, 100 different problems were executed and the average speedup was determined. The observations were gathered for problem rows varying between 10 and 50, columns between 1 and 9, link faults between 0 and $n-1$. The points to be noted from these experiments are: (1) Every problem was speeded up with load balancing regardless of the number of rows, columns or processor faults. (2) The value of speedup increases with the increase in hypercube dimension, if the other attributes of the problem are kept the same. (3) A sharp rise in speedup is obtained with load balancing for any dimension of hypercube when the number of rows and number of faults are constant but the number of columns for the problems is increased.

6 Conclusions

This paper shows with a practical example how load balancing can be applied to a problem. Speedup in the performance is clearly obtained with load balancing. Use of the link-fault-tolerant algorithm provides speedup over the no load balance case in the presence of 0 to $n-1$ link faults. The amount of speedup obtained varies with the hypercube dimension, problem type and size. The algorithm designer has to determine whether the problem is amenable to be solved in parallel and how to implement it on the multiprocessing system employed.

References

[1] M. S. Chen and K. G. Shin, "Subcube allocation and task migration in hypercube multiprocessors", *IEEE Transactions on Computers*, Vol. 39, No.9, pp. 1146-1155, Sept 1991.

[2] T. C. K. Chou and J. A. Abraham, "Load Balancing in distributed systems", *IEEE Transactions on Software Engineering*, Vol. SE-8, No.4, pp. 401-412, July 1982.

[3] G. Cybenko, "Dynamic load balancing for distributed memory multiprocessors", *Journal of parallel and distributed computing*, Vol. 7, No.2, pp. 279-301, Oct 1989.

[4] D. Eager, E. Lozowska and J. Zahorjan, "Adaptive load sharing in homogenous distributed systems", *IEEE Transactions on Software Engineering*, Vol. SE-12, No.5, pp. 662-675, May 1986.

[5] J.P. Hayes and T. Mudge, "Hypercube Supercomputers", *Proceedings of the IEEE*, Vol. 77, No.12, pp. 1829-1841, Dec 1989.

[6] R. Mirchandaney, D. Towsley and J. A. Stankovic, "Adaptive load sharing in heterogeneous distributed systems", *Journal of Parallel and Distributed Computing*, Vol. 9, No.4, pp. 331-346, Aug 1990.

[7] H. B. More and J. Wu, "Load balancing on multiprocessor systems", Florida Atlantic University TR-CSE-92-35, Nov 1992.

[8] D. M. Nicol, "Communication efficient global load balancing", *Proceedings of Scalable High Performance Computing Conference SHPCC-92*, pp. 292-299, Apr 1992.

[9] N. G. Shivratri, P. Krueger and M. Singhal, "Load Distributing for locally distributed systems", Computer, pp 33-44, December 1992.

[10] C. L. Seitz, "The cosmic cube", *Communications of the ACM*, Vol. 28, pp. 22-33, Jan 1985.

[11] J. Wu, "Dimension exchange based load balancing in injured hypercubes", Florida Atlantic University TR-CSE-93-31, June 1993.

[12] S. M. Yuan, "An efficient periodically exchanged dynamic load balancing algorithm, *International journal of mini and microcomputers*, Vol. 12, No.1, 1990.

[13] S. Zhou, "A trace driven simulation study of dynamic load balancing", *IEEE Transactions on Software Engineering*, Vol. 14, No. 9, pp. 1327-1341, Sept 88.