Genetic Solutions to the Load Balancing Problem *

Joey Baumgartner, Diane J. Cook, and Behrooz Shirazi

Department of Computer Science and Engineering
University of Texas at Arlington
Arlington, TX 76019

Abstract

Efficient use of resources in a parallel machine often requires the redistribution of tasks during the execution of applications. This problem of load balancing is a formidable one because of the unpredictable and dynamic nature of many user programs. In this paper, we present a load balancing method that employs the use of genetic algorithms. The benefits of our approaches to genetic-based load balancing are illustrated in this paper using performance results on a task processing simulator and on two parallel processing machines: the Connection Machine 5 and the Intel Hypercube.

1 Introduction

As parallel and distributed computers become more powerful there are many challenges that must be overcome to fully utilize their capabilities. One such challenge that has existed since parallel machines were first developed is that of keeping such a large number of processors busy. This problem has become even more prominent as these machines have grown from tens to thousands of processors.

This has been the focus of traditional load balancing algorithms. These techniques have been applied to a number of relatively small distributed and parallel systems and a variety of approaches have been presented [1, 4, 7, 9, 11]. Most of these approaches employ the use of heuristics and fixed parameters to make the necessary decisions to distribute work among the processors. While many of these techniques have been effective in increasing the utilization of their target machines, the complexity of the load balancing problem and their dependency on heuristics limit their effectiveness under certain conditions.

*This work was supported by National Science Foundation grant IRI-9308308 and by a grant from the National Center for Supercomputing Applications.

In order for these traditional load balancing schemes to perform well, they must be optimized for each platform and situation in which they are used. This requires laborious testing to fine-tune the system to the most common situation and still does not guarantee good performance in a wide range of conditions.

A more general load balancing algorithm must possess characteristics that allow it to perform well in a variety of situations and under changing parameters. These situations include different communications architectures, application demands, and scalability with increasing system size.

To address the problem of adaptability in load balancing algorithms, new methods are explored with adaptability in mind. These methods are based on an area of Artificial Intelligence called **genetic algorithms**. Our research focuses on developing techniques that make use of genetic algorithms and classifier systems to develop, enhance, and improve the adaptability of load balancing algorithms.

In this paper, we introduce two genetic-based load balancing algorithms: **GCTA** (Genetic Central Task Assigner) and **CBLB** (Classifier-Based Load Balancer) [3]. The results of our experiments demonstrate that application of genetic algorithms and classifier systems does dramatically improve the performance of parallel and distributed software across a variety of application domains and parallel processing systems.

2 Load Balancing Algorithms

In a distributed or parallel system, independent units of work called tasks arrive at a given rate and wait to be processed. Tasks may be executed by the processor at which they first arrive or, if a load balancing algorithm is being used, may be transferred by the load balancing algorithm to a different node.

Load balancing is used to improve the utilization of resources in a multiple computer system by redistributing work from heavily loaded processors to

lightly loaded ones. Load balancing should reduce the mean response time $\overline{R}$ of tasks and improve the performance of the applications running on the system.

One of the most popular *dynamic* load balancing algorithms is the **Threshold Algorithm** [4, 11], which places the burden of information collection, transfer decision, and placement decision are placed on each individual processor. Significant work in the area of centralized approaches was perform by Zhou [10, 11]. The **Central Algorithm** is a solution that uses a central host to make placement decisions. The **Centex Algorithm** also relies on centralized decision making, but improves upon the Central Algorithm by periodically collecting load information from the nodes to more accurately depict the state of the system. The frequency of these load updates is governed by the update period parameter UP.

Xu and Hwang [9] offer a hybrid approach in which the individual nodes determine when to transfer tasks, but placement decisions are based on load information they receive from the host. This algorithm involves a degree of adaptability in that the global threshold and update time are adjusted based on fixed heuristics.

While many researches have produced good results from their experiments, most are quick to mention the dependability their methods have on the choice of fixed parameters. In his comprehensive comparison of several algorithms [11], Zhou points out that

> The sensitivity of load balancing performance to the values of the parameters of load balancing algorithms suggests that some form of *adaptive load balancing* may be able to provide good performance when system load changes widely ... parameter adjustments are ... capable of significantly improving the performance when the system load fluctuates widely.

Kremien and Kramer offer a similar analysis when they note that the parameters they select for one set of conditions, "... may not be the best under different conditions"[7]. Load balancing is important in a variety of architectures and the need for a general-purpose adaptable algorithm is quite clear.

3 Genetic Algorithms

Genetic algorithms have gained considerable attention since their introduction by Holland [6] almost two decades ago. These algorithms have proven themselves useful in a variety of optimization problems and have been given praise for their ease of implementation and near-optimal solutions. Genetic algorithms have been very successful in a variety of scheduling problems where optimal solutions are too costly to produce.

Our genetic algorithm consists of a population of strings called *chromosomes* C, an evaluation function called a *fitness function* $E(C)$, and a life cycle consisting of three operators: *reproduction*, *crossover*, and *mutation* [5].

The basic premise of the genetic algorithm is that solutions to a problem are coded in the chromosomes and each of these chromosomes are evaluated based on the fitness function. The chromosomes with the higher fitness values are selected for reproduction to the next generation. These selected chromosomes mate with each other in the crossover operation creating new chromosomes which are occasionally mutated. The life cycle continues for a certain number of *generations*, ideally producing better and better solutions.

Classifier systems were first developed over a decade ago. They are simple learning systems based on the use of condition/action statements, or *rules*. The rules system along with some *discovery* mechanism allow them to function very well in numerous domains as optimization or control mechanisms.

Classifier systems have three major components: a rule base, a credit system, and a genetic algorithm. Throughout execution of the system, rules whose conditions match the input conditions are allowed to compete to *fire* their output actions.

Rules that are selected to fire receive reward or punishment based on their performance. Rules that gain strength relative to others will be chosen more often in the matching process. Periodically the genetic algorithm is executed on the rule base to allow the creation of new rules. Generally, the stronger rules in the rule base are used to create new rules and replace the weaker ones.

4 Genetic Methods for Load Balancing

In order to accomplish the goal of adaptable load balancing through the use of genetic algorithms, two very different approaches have been taken. The first approach, GCTA, uses a genetic algorithm to perform the entire load balancing task. The second approach, CBLB, augments an existing load balancing algorithm and employs a classifier system to control the parameters of this load balancer. By incorporating the classifier system with the traditional load balancing algorithm, CBLB attempts to *tune* the algorithm with which it is working, providing the necessary adaptability.

4.1 Genetic Central Task Assigner

The genetic-based load balancer [3] is a centralized algorithm in which one dedicated processor in the system periodically collects state information from the other processors.

After receiving load information, the central processor then executes GCTA to determine the best distribution for work in the system. The distribution is broadcast to all processors which then transfer tasks to the specified location and continue execution.

The GA population represents a set of possible transfer decisions. Each position in the chromosome corresponds to a waiting task, and the corresponding value indicate the processor to which the task should be transferred. If the task already exists in the indicated processor's queue, no transfer is necessary.

Load balance is measured using the mean response time $\overline{R}$ of all the tasks in the system. GCTA's simple fitness function $E_s(C)$ calculates the total number of transfers to be performed and the total amount of work to be performed after the transfers. The complex fitness function $E_c(C)$ calculates the entire expected run-time of the set of tasks in the system, accounting for transfer overhead and delays. This function is costly to calculate and is only used in the final generations of the algorithm. The simple function is sufficient to build a *good* population and is run for the first half of the generations.

Individuals are selected for the next generation of the algorithm using the *Biased Roulette Wheel* selection scheme. Crossover is performed by dividing two parent chromosomes at a *cross-site* and combining the two chromosomes at the cross site. Mutation is applied with low probability to individual positions within the chromosome, providing needed diversity in the GA population.

The primary advantage of GCTA lies in its ability to optimize placement of work in the system. However, in order to provide full optimization, the fitness function needs to accurately model the system and becomes quite costly. The CBLB system focuses upon this limitation of GCTA by fine-tuning parameters of an existing load balancing system, rather than using a genetic algorithm to make all of the load balancing decisions.

4.2 Classifier-Based Load Balancer

The **CBLB Algorithm** employs a simple classifier system running in conjunction with Zhou's Centex Algorithm. Centex is executed in normal fashion by the central host and each of the nodes, but the parameters of the algorithm fall under the control of the classifier system running on the central host.

The classifier system consists of several components: input conditions, output actions, a rule base, a matching and selection algorithm, a reward system, and a rule discovery system.

Input conditions for each rule consists of the mean response time since the last update, the mean utilization per node since the last update, and the inverse standard deviation of arrivals since the last update. These three values are matched against the left-hand side of rules in the database.

Output actions have the effect of setting three of the system parameters: transfer queue threshold, update period time, and CPU threshold. These actions are represented by the right-hand side of rules in the database.

As state above, CBLB contains $m = 3$ conditions and $n = 3$ actions. Table 1 describes the meaning of each field. Since each rule is 14 bits long, there are $16,384$ (2^{14}) possible rules. The input conditions make up 5 of these 14 bits; therefore, there are 32 (2^5) possible combinations of conditions. The rule base in CBLB has 128 rules, allowing 4 rules for each of the 32 possible input combinations.

In a classifier system, rules gain strength relative to their performance. The method of strength reward in CBLB is based on the mean response time for that update period $\overline{R}_{UP}$. Rules are given a reward of $\frac{1}{\overline{R}_{UP}}$ if the firing of a rule resulted in a decrease in $\overline{R}_{UP}$. No reward is given for rules that do not decrease $\overline{R}_{UP}$ when fired.

When the input conditions are received from the environment, all rules which have condition fields matching those of the input are selected to *compete* with each other to be able to fire their actions. Competition is carried out through a roulette-wheel selection process in which each rule is given a probability of selection proportional to its strength. All rules that match are allowed to participate to encourage competition in the system. When a rule has been selected, that rule is *fired* and the appropriate changes are made within the system.

Rule creation occurs periodically according to the genetic algorithm update parameter UP. When invoked, the genetic algorithm is executed on each of the 32 divisions of the rule base. Details of the crossover and mutation operators applied to CBLB are provided elsewhere [3].

CBLB is designed to *train* the load balancing system it is working with. In practice, the system would employ a database mechanism to store the best per-

Field	Bits	Description	Coding
c1	1	Mean Response Time Trend	0 decreased 1 increased
c2	2	Mean Utilization	00 < 0.25 01 < 0.50 10 < 0.75 11 < 1.00
c3	2	Load Imbalance	00 < 0.25 01 < 0.50 10 < 0.75 11 < 1.00
a1	3	Set Queue Threshold T_q	T_q = binary value
a2	3	Set Update Time UP	UP = binary value
a3	3	Set CPU Threshold T_{cpu}	T_{cpu} = binary value*0.25

Table 1: Format of Rule in CBLB

forming sets of rules during the training session. When sufficient training has taken place, the classifier system continues to function without the rule creation mechanism provided by the genetic algorithm. In this way, CBLB can be used to optimize the load balancing system, and remain in effect to provide further adaptability during execution.

CBLB offers two improvements to traditional load balancing approaches. The first is the ability to change the parameters of the system. The second and less obvious benefit comes from the partitioning of the rules. By dividing the rules into different groups based on the current conditions of the system, a particular set of parameters can be assigned to meet the current situation. Another advantage of the CBLB algorithm is that it fits nicely into so many simple load balancing algorithms. Simple algorithms have been found by many researchers to be superior to more complex ones because the benefits of complex ones do not outweigh their overhead.

5 Experiments

5.1 Simulation Experiments

An event-driven simulator based on MacDougall's **smpl** simulation language [8] has been implemented. The simulator is designed so that many of the parameters of the system are user-configurable to allow the simulation of a variety of conditions.

Each processor in the system is modeled by a single server and a waiting queue. Tasks arrive to servers with a Poisson arrival rate of λ and exponential mean service time $\overline{S}$. The user-specified utilization factor U is used to adjust the mean inter-arrival time of tasks. Since $U = \lambda\overline{S}$, the mean inter-arrival time of tasks is given by $1/\lambda = \overline{S}/U$.

When a task *arrives* at the system, two different methods are used to determine which processor it will use. In the first method, host processors are chosen at random, yielding a somewhat balanced load. The second method forces an unbalanced load by using an imbalance factor I. The greater the value of I, the more unbalanced the load is.

The cost of task transfer is determined by two parameters. It is assumed that a message transfer incurs an overhead of C_o at both the sending and receiving processors. Also, each message that is passed between two processors incurs a network delay of C_d. The simulator assumes that the distance between processors and the size of messages is uniform.

5.2 GCTA Simulation Results

To test the genetic-based load balancer, several simulation studies were performed comparing the mean response time $\overline{R}$ of GCTA, Threshold, Centex, Central, and no load balancing (NOLB). The first experiment compares the algorithms under different levels of utilization at uniform load distribution. The second experiment varies the load imbalance as well as the utilization.

For each of the methods tested, $\overline{S}$ was set to 1 second. The simulator was run for 10,000 seconds of simulation time and results are averaged over three runs using different random seeds. The number of processors was fixed at $N = 16$ and the communication delays C_d and C_o were fixed at $25ms$ and $8ms$ respectively.

The Threshold Algorithm implemented uses a

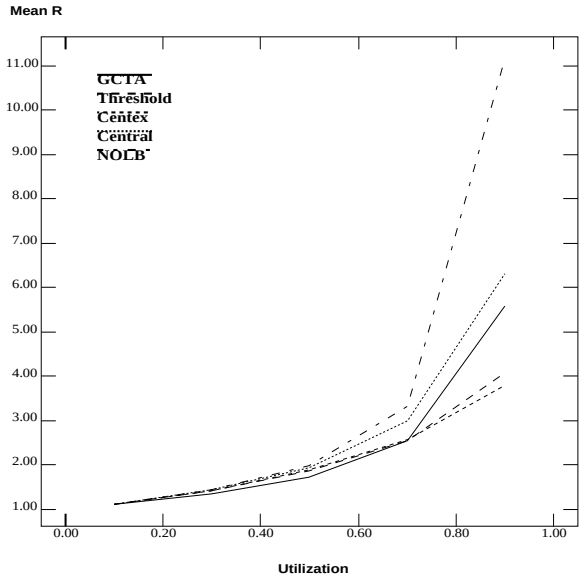

Figure 1: Mean Response Time vs Utilization (uniform load)

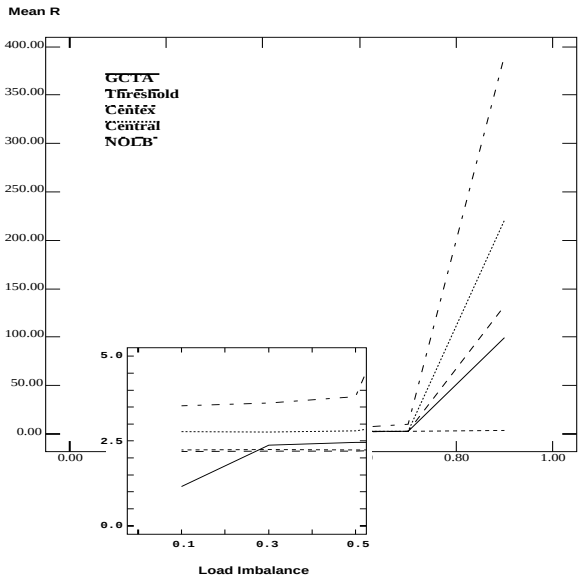

Figure 2: Mean Response Time vs Load Imbalance

threshold of $T_q = 3$. The probing limit is a function of the number of processors in the system, $Pl = N/4$. For each probe, the probing and responding processors incur an overhead of $2C_o$ for processing the probe message. A delay of C_d is added appropriately to account for network travel. Delay due to the genetic algorithm is also added to system cost.

The Central and Centex algorithms both use queue thresholds $T_q = 3$ and CPU thresholds $T_{cpu} = 1.0$. The update time of the Centex Algorithm used was $UP = 2.5$.

Figure 1 shows the results of the utilization experiment. From the graph it can be seen that GCTA provides excellent performance against NOLB. At low utilization GCTA is better than the other algorithms, but performs less well at higher utilization. Centex and Threshold offer the best performance at the higher loads.

To test the effect of load imbalance, the load imbalance factor I was varied from 0.1 to 0.9 in increments of 0.2. At each level of load imbalance, U was varied from 0.1 to 0.9, also in increments of 0.2. The mean response time of each method under different levels of utilization and load imbalance was measured.

Figure 2 [1] shows the results of this experiment. All of the algorithms offer improvement over the NOLB case. GCTA performs better than the others at low load imbalance, but degrades in relation to Centex at high load imbalance. GCTA outperforms all of the algorithms with the exception of Centex at high levels of load imbalance.

5.3 CBLB Simulation Results

Because the primary component of the CBLB algorithm is adaptability of system parameters, a second set of simulation experiments was conducted in which the simulation environment values *vary* as the simulation proceeded. In the first study, environment values were selected randomly every 500 time units. The same random environment values were generated for each algorithm so that they all used the same sequence of values. The second experiment used a more deterministic set of environment values that was generated by hand. Again, the value sequence was the same for all three algorithms.

Two different methods for generating the fluctuating environment values were used so that the simulator could model different situations. The first method represents a consistently random load in which the demands of the system vary greatly from one minute to the next. The second deterministic set of values was generated by hand to better model a parallel system that is running user applications. The values are al-

[1] The Central Algorithm was unable to complete the final simulation run of $U = 0.9$ and $I = 0.9$. The result shown in Figure 2 represents the average of 4 runs instead of five.

Load	NOLB	Centex	CBLB
Random	7.109013	2.363300	2.551288
Deterministic	111.609083	18.950631	9.244940

Table 2: Mean Response Time of CBLB and Centex

lowed to remain stable for longer periods of time, but still fluctuate widely over the simulation run.

For each of the methods tested, the Centex Algorithm was run 1 time while the CBLB Algorithm was run 50 times to test performance over a series of simulations. To allow CBLB to adapt to the input parameters, the rule base was preserved from one run to the next.

To test the adaptability of the CBLB algorithm, the two simulation runs were executed and CBLB was allowed to adjust the load balancing parameters during execution. After each of the 50 simulation runs, the mean response time $\overline{R}$ of the system was measured. To rule out the possibility that the performance of the classifier system is due to reasons other than its effectiveness at learning and adapting, system parameters are adjusted every UP units.

To test the effectiveness of CBLB, the Centex and NOLB algorithms were executed for two separate experiments. In the first experiment, system parameters are randomly generated. In the second experiment, system parameters are determined deterministically in a manner that puts a great deal of stress on the system. Table 2 shows the mean response time $\overline{R}$ for each of the algorithms under both of the fluctuating environments.

Both algorithms offer a great deal of improvement over the NOLB case. In the first experiment, Centex by itself is able to perform better with a fixed set of parameters than CBLB. However, CBLB is able to perform considerably better under the deterministic environment experiment.

To demonstrate the ability of CBLB to adapt to the parameters of existing parallel machines, we implemented the CBLB system on a Connection Machine 5 and an Intel Hypercube. In both sets of experiments, artificial tasks are generated and arrive at a controlled rate to the individual processors. The amount of time needed to process each task is determined by the preassigned size of the task. On the other hand, the communication costs used to drive the classifier system are measured from the machines themselves. The classifier system relies on periodic feedback from the processors to improve its choice of load balancing rules.

Figure 3 shows the results of load balancing using

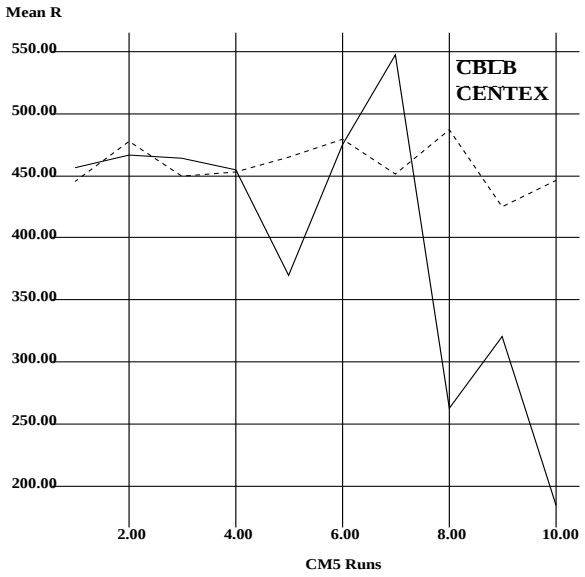

Figure 3: CBLB on a Connection Machine 5

both CBLB and Centex on the Connection Machine 5 with 32 processors. CBLB does improve the system performance over time. The temporary increase in mean response time for CBLB is due to the experimentation with new rules which do not necessarily yield improved performance. As the rule base adapts to the needs of the system, the overall performance steadily improves. Centex alone does not improve, but provides roughly the same performance as the number of runs increases. The Centex values were averaged over three different sets of fixed parameters.

Figure 4 shows a similar set of results comparing Centex and CBLB on an Intel Hypercube with 16 processors. Once an effective set of parameters is discovered by the classifier system, the performance dramatically improves. For both of these experiments, a fixed set of task sizes and arrival times were used. Because of the static task parameters, the classifier system was able to adapt quickly to the system parameters.

6 Conclusions

To address the problem of adaptability in load balancing algorithms, two new approaches have been explored. These new approaches use the benefits of the genetic algorithm to develop a system that adapts to changing conditions and environments.

The benefits of the GCTA algorithm arise from its

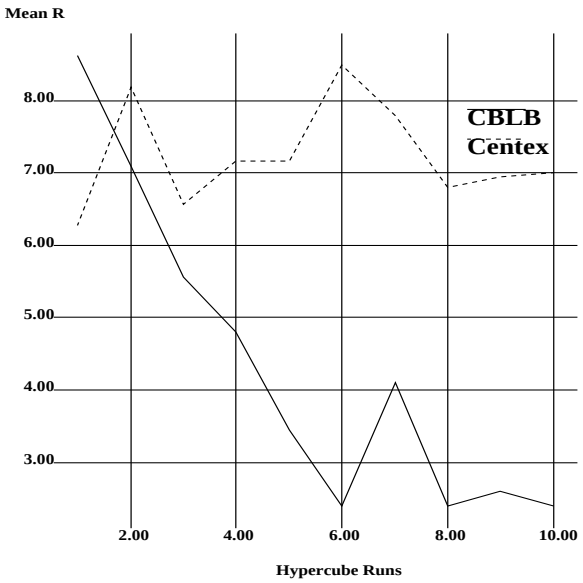

Figure 4: CBLB on an Intel Hypercube

ability to optimize the placement of work within the system it is working in. The unique approach used in GCTA yielded a system that was comparable in performance to the very popular Threshold Algorithm. However, comparison with the Centex Algorithm revealed that the GCTA algorithm was not necessarily the most effective solution available.

An entirely different system that also uses genetic algorithms was developed to explore ways in which the genetic algorithm could be used to improve the performance and adaptability of existing load balancing methods. This method, CBLB, uses a classifier system to control the parameters of existing systems. CBLB has been shown to improve the performance of the fixed-parameter Centex Algorithm and provide the necessary adaptability.

Future directions of this work involve further testing of the CBLB on existing architectures. This paper reports results of testing CBLB on an Intel Hypercube and a Connection Machine 5. We would like to extend these tests to other parallel platforms and distributed networks.

Another direction is the movement of the testing of CBLB from simulation to actual user programs. An earlier version of this work applied a pure genetic algorithm load balancer to a heuristic search program [2]. Further testing can be performed to demonstrate the power of CBLB applied to this and other computation-intensive algorithms.

References

[1] I. Ahmad and A. Ghafoor. Semi-distributed load balancing for massively parallel multicomputer systems. *IEEE Transactions on Software Engineering*, 17(10):987–1004, 1991.

[2] J. Baumgartner and D. J. Cook. A genetic algorithm for load balancing in parallel computers. In *Proceedings of the Seventh International Confernece on Industrial and Engineering Applications of Artificial Intelligence and Expert Systems*, pages 619–628, 1994.

[3] J. Baumgartner and D. J. Cook. A genetic-based solution to load balancing in parallel computers. In *Proceedings of the 1994 ACM Computer Science Conference*, 1994.

[4] D. L. Eager, E. D. Lazowska, and J. Zahorjan. Adaptive load sharing in homogeneous distributed systems. *IEEE Transactions on Software Engineering*, 12(5):662–675, May 1986.

[5] D. E. Goldberg. *Genetic Algorithms in Search, Optimization, and Machine Learning.* Addison-Wesley Publishing Company, 1989.

[6] J. H. Holland. *Adaptation in Natural Artificial Systems.* The University of Michigan Press, 1975.

[7] O. Kremien and J. Kramer. Methodical analysis of adaptive load sharing algorithms. *IEEE Transactions on Parallel and Distributed Systems*, 3(6):747–760, November 1992.

[8] M. H. MacDougall. *Simulating Computer Systems: Techniques and Tools.* The MIT Press, 1987.

[9] J. Xu and K. Hwang. Heuristic methods for dynamic load balancing in a message-passing multicomputer. *Journal of Parallel and Distributed Computing*, 18(1):1–13, 1993.

[10] S. Zhou. A trace-driven simulation study of dynamic load balancing. Technical Report UCB/CSD 87/305, University of California, Berkely, 1986.

[11] S. Zhou. A trace-driven simulation study of dynamic load balancing. *IEEE Transactions on Software Engineering*, 14(9):1327–1341, September 1988.