JOURNAL OF PARALLEL AND DISTRIBUTED COMPUTING **43,** 156–162 (1997)
ARTICLE NO. PC971339

Customized Dynamic Load Balancing for a Network of Workstations[1]

Mohammed Javeed Zaki, Wei Li, and Srinivasan Parthasarathy[2]
Computer Science Department, University of Rochester, Rochester, New York 14627

Load balancing involves assigning to each processor work proportional to its performance, thereby minimizing the execution time of a program. Although static load balancing can solve many problems (e.g., those caused by processor heterogeneity and nonuniform loops) for most regular applications, the transient external load due to multiple users on a network of workstations necessitates a dynamic approach to load balancing. In this paper we show that different load balancing schemes are best for different applications under varying program and system parameters. Therefore, application-driven customized dynamic load balancing becomes essential for good performance. We present a hybrid compile-time and run-time modeling and decision process which selects (customizes) the best scheme, along with automatic generation of parallel code with calls to a run-time library for load balancing.

1. INTRODUCTION

Networks of workstations (NOWs) provide attractive scalability in terms of computation power and memory size. With the rapid advances in new high speed computer network technologies (e.g., ATMs), NOWs are becoming increasingly competitive compared to expensive parallel machines. However, NOWs are much harder to program than dedicated parallel machines. For example, a multiuser environment with sharing of CPU and network may contribute to varying performance. Heterogeneity in processors, memory, and network are also contributing factors.

Efficient scheduling of loops on a NOW requires finding the appropriate granularity of tasks and partitioning them so that each processor is assigned work in proportion to its performance. This load balancing assignment can be *static*—done at compile-time—or it may be *dynamic*—done at run-time. The distribution of tasks is further complicated if processors have differing speeds and memory resources, or due to transient external load and nonuniform iteration execution times. While static scheduling avoids the run-time scheduling overhead, in a multiuser environment with load changes on the nodes, a more dynamic approach is warranted. Moreover, different schemes are best for different applications under varying program and system parameters. Application-driven customized load balancing thus becomes essential for good performance. This paper addresses the above problem. In particular we make the following contributions: (1) We compare different strategies for dynamic load balancing in the presence of transient external load. We examine both global vs local and centralized vs distributed schemes. (2) We present a hybrid compile and run-time system that automatically selects the best load balancing scheme for a loop/task. We also automatically transform an annotated sequential program into a parallel program with appropriate calls to our run-time load balancing library. (3) We present experimental results to substantiate our approach.

The rest of the paper is organized as follows. We first present related work (Section 2), which is followed by a description of the load balancing schemes (Section 3). We then present the compile-time model and decision methodology (Section 4) and describe the hybrid compile and run-time system (Section 5). Finally, we present the modeling and experimental results (Section 6) and our conclusions (Section 7).

2. RELATED WORK

Compile-time *static* loop scheduling has been well studied [9, 14]. Static scheduling for heterogeneous NOWs was proposed in [4, 5, 7]. The *task queue model* for dynamic loop scheduling has targeted shared-memory machines [11, 14], while the *diffusion model* has been used for distributed-memory machines [10]. A method for task-level scheduling in heterogeneous programs was proposed in [13], and [2] presents an application-specific approach to schedule individual parallel applications.

A common approach taken for dynamic load balancing on a workstation network is to predict future performance based on past information. For example, [12] presents a global distributed scheme, Dome [1] implements a global central and a local distributed scheme, and Siegell [16] presents a global centralized scheme with automatic generation of parallel programs with dynamic load balancing. In all cases the load balancing involves periodic exchanges. Both Phish [3] and CHARM [15] implement a local distributed receiver-initiated scheme, but they use different performance metrics. Our approach also falls under this model. Instead of periodic

[1]This work was supported in part by NSF Research Initiation Award (CCR-9409120), and ARPA contract F19628-94-C-0057.
[2]E-mail: {zaki, wei, srini}@cs.rochester.edu.

0743-7315/97 $25.00

exchanges of information, we have interrupt-based receiver-initiated schemes. Moreover, we look at both central vs distributed and local vs global approaches. Our work differs from the above approaches in that our goal is to provide compile and run-time support to automatically select the best load balancing scheme for a given loop/task from among a repertoire of different strategies.

3. DYNAMIC LOAD BALANCING (DLB)

After the initial assignment of work (the iterations of the loop) to each processor, dynamic load balancing is done in four basic steps: monitoring processor performance, exchanging this information between processors, calculating new distributions and making the work movement decision, and actually moving the data. The data is moved directly between the slaves, and the load balancing decisions are made by the *load balancer*.

Synchronization. In our approach, a synchronization is triggered by the first processor that finishes its portion of the work. This processor then sends an interrupt to all the other active slaves, which then send their performance profiles to the load balancer.

Performance Metric. We try to predict the future performance based on past information, which depends on the past load function. We can use the whole past history or a portion of it. Usually, the most recent window is used as an indication of the future. The metric we use is the number of iterations done per second since the last synchronization point.

Work and Data Movement. Once the load balancer has all the profile information, it calculates a new distribution. If the amount of work to be moved is below a threshold, then work is not moved, since this may indicate that the system is almost balanced, or that only a small portion of the work still remains to be done. If there is a sufficient amount of work that needs to be moved, we invoke a *profitability analysis* routine. This is required since work redistribution also entails the movement of the data, and there is a trade-off between the benefits of moving work to balance load and the cost of data movement. We thus redistribute work as long as the potential benefit of the new assignment results in an improvement. If it is profitable to move work, then the load balancer broadcasts the new distribution information to the processors. The work and data are then redistributed between the slaves.

3.1. Load Balancing Strategies

We chose four different strategies differing along two axes. The techniques are either *global* or *local*, based on the information they use to make load balancing decisions, and they are either *centralized* or *distributed*, depending on whether the load balancer is located at one master processor (which also takes part in computation) or is distributed among the processors, respectively. For all the strategies, the compiler initially distributes the iterations of the loop equally among all the processors.

Global Strategies. In the global schemes, the load balancing decision is made using global knowledge; i.e., all the processors take part in the synchronization and send their performance profiles to the load balancer. The global schemes we consider are (1) *Global Centralized DLB* (*GCDLB*)—the load balancer is located on a master processor (centralized). After calculating the new distribution, the load balancer sends instructions to the processors who have to send work to others, indicating the recipient and the amount of work to be moved. The receiving processors just wait till they have collected the amount of work they need. (2) *Global Distributed DLB* (*GDDLB*)—the load balancer is replicated on all the processors. The profile information is broadcast to all the processors, eliminating the need to send out instructions. The receiving processors wait for work, while the sending processors ship the data.

Local Strategies. In the local schemes, the processors are partitioned into different groups of size *K*. We used the *K*-block static partitioning approach, where the load balancing decisions are only done within a group (other approaches such as *K-nearest-neighbors* or dynamic partitioning are also possible). If processors have different speeds, we can do the static partitioning so that each group has nearly equal aggregate computational power. The local schemes are the same as their global counterparts, except that profile information is only exchanged within a group. The two local strategies we look at are (1) *Local Centralized DLB* (*LCDLB*) and (2) *Local Distributed DLB* (*LDDLB*).

These strategies were chosen to lie at the four extreme points on the two axes. In the local approach, there is no exchange of work between different groups. For LCDLB, we have only one master load balancer, instead of one per group. Furthermore, in the distributed strategies we have full replication of the load balancer. Exploring the behavior of other hybrid schemes is part of future work. We now look at the major differences among the different strategies:

Global vs Local. For the global schemes, since global information is available at a synchronization point, the work distribution is near-optimal, and convergence is faster compared to the local strategies. However, the amount of communication or synchronization cost is lower in the local case. For the local case, difference in performance among the different groups can also have a significant impact. For example, if one group has processors with poor performance (high load), and the other group has very fast processors (little or no load), the latter will finish quite early and remain idle, while the former group is overloaded. This could be remedied by providing a mechanism for exchange of data between groups or by having dynamic group membership.

Centralized vs Distributed. In the centralized schemes, the central point of control could limit the scalability. The

distributed schemes help solve this problem. However, in these schemes the synchronization involves an all-to-all broadcast, while the centralized ones require an all-to-one profile send, followed by an one-to-all instruction send. There is also a tradeoff between sequential load balancing decision making in the centralized approach and the parallel (replicated) decision making in the distributed schemes.

4. DLB MODELING AND DECISION PROCESS

We now describe our compile and run-time modeling and decision process for choosing among the different load balancing strategies. We present the different parameters that may influence the performance of these schemes, followed by the derivation of the total cost function. Finally we show how this modeling is used.

4.1. Modeling Parameters

Processors Parameters. These give information about the different processors available to the application. These include: (1) number of processors (P), (2) processor speeds (S_i), the ratio of processor i's performance w.r.t. a base processor [18], and (3) number of neighbors (K).

Program Parameters. These parameters give information about the application. These include: (1) data size (N), (2) number of loop iterations ($\mathcal{I}$), (3) work per iteration ($\mathcal{W}$), (4) data communication ($\mathcal{D}$), indicating the aggregate number of bytes that need to be communicated per iteration, and (5) time per iteration ($\mathcal{T}$) on the base processor. The time to execute an iteration on processor k is simply $\mathcal{T}/S_k$.

Network Parameters. These specify the properties of the interconnection network, and include (1) network latency ($\mathcal{L}$), (2) network bandwidth ($\mathcal{B}$), and (3) network topology. In this paper, we assume full connectivity among the processors, with uniform latency and bandwidth.

External Load Modeling. To evaluate our schemes, we had to model the external load. In our approach, each processor has an independent load function, denoted as ℓ_i. The two parameters for generating the load function are: (1) maximum load (m), specifying the maximum amount of load per processor (in our experiments we set $m = 5$), and (2) duration of persistence (d), indicating the amount of time before the next load change (a uniformly generated random number in the range $[0, m]$); i.e., we have a discrete random load function, with a maximum amplitude given by m, and the discrete block size given by d. A small value for d implies a rapidly changing load, while a large value indicates a relatively stable load. We use $\ell_i(k)$ to denote the load on processor i during the kth duration of persistence.

4.2. Modeling—Total DLB Cost

We now present the cost model for the various strategies. The cost of a scheme can be broken into the following categories: cost of synchronization, cost of calculating new distribution, cost of sending instructions, and cost of data movement (see [19] for a more detailed study).

Cost of Synchronization. The synchronization involves the sending of interrupt from the fastest processor to the other processors, which then send their performance profile to the load balancer. The cost for the different strategies is given below (for the local case $P = K$): (1) GCDLB: ξ = one-to-all(P) + all-to-one(P), and (2) GDDLB: ξ = one-to-all(P) + all-to-all(P^2).

Cost of Distribution Calculation. This cost is usually quite small, and we denote it as δ.

Cost of Data Movement. Let $\chi_i(j)$ denote the iteration assignment after the jth synchronization, and $\sigma_i(j)$ the *average effective speed* [19], for processor i during the jth and the previous synchronization. Let $\mathcal{T}$ be the time for one iteration. The time taken by the fastest processor f to complete its work is, $t = (\chi_f(j-1) \cdot \mathcal{T})/\sigma_f(j)$. The number of iterations remaining on processor i is simply its old distribution minus the iterations done in time t; i.e., $\gamma_i(j) = \chi_i(j-1) - \chi_f(j-1) \cdot (\sigma_i(j)/\sigma_f(j))$. The total amount of work left among all the processors is then given as $\Gamma(j) = \Sigma\ \gamma_i(j)$. We distribute this work proportionally to the average effective speed of the processors to get $\chi_i(j) = (\sigma_i(j)/\sum_{k=1}^{P} \sigma_k(j)) \cdot \Gamma(j)$.

The initial values $\chi_i(0) = \mathcal{I}/P$, and $\gamma_i(0) = \chi_i(0)$, $\forall i \in 1, \cdots, P$, give us recurrence functions which can be solved to obtain the new distribution at each synchronization point. The termination condition occurs when there is no more work left to be done. Along with the number of iterations moved ($\alpha(j) = \frac{1}{2}(\sum_{i=1}^{P} |\gamma_i(j) - \chi_i(j)|)$), and the number of messages required to do this ($\beta(j)$), the total cost of data movement is then $\kappa(j) = \beta(j) \cdot \mathcal{L} + \alpha(j) \cdot (\mathcal{D}/\mathcal{B})$.

Cost of Sending Instructions. This applies only to the centralized schemes, since the load balancer sends the work and data movement instructions to the processors. This cost is simply $\psi(j) = \beta(j)\mathcal{L}$.

Using the above analysis, the per group cost is $\mathcal{C}_g = \eta_g(\xi + \delta) + \sum_{j=1}^{\eta_g}[\kappa_g(j) + \psi_g(j)]$, where η is the number of synchronizations and g the group number. The total cost for the DLB schemes is simply the maximum group cost, $\mathcal{C} = \mathrm{MAX}_g\{\mathcal{C}_g\}$ (for global schemes $g = 1$).

4.3. Decision Process—Using the Model

Initially at run-time, no strategy is chosen for the application. Work is partitioned equally among all the processors, and the program is run till the first synchronization point. During this time a significant amount of work has been accomplished; namely, at least $1/P$ of the work has been done. At this time we also know the load function seen on all the processors so far and the average effective speed of the processors. This load function, combined with all the other parameters, can be plugged into the model to obtain quantitative information on the behavior of the different schemes. This information is then used to commit to the best strategy after this stage. This also

suggests a more adaptive method for selecting the scheme, where we refine our decision as more information on the load is obtained at later points. This is part of future work.

5. COMPILER AND RUN-TIME SYSTEMS

Since all the information used by the modeling process, such as the number of processors, processor speeds, data size, number of iterations, and iteration cost, and particularly the load function, may not be known at compile time, we propose a hybrid compile and run-time modeling and decision process. The compiler collects all necessary information and may also help to generate symbolic cost functions for the iteration cost and communication cost (some of the parameter values can also be obtained by performance prediction [17]). The actual decision making for committing to a scheme is deferred until run-time when we have complete information about the system.

Run-Time System. The run-time system consists of a uniform interface to the DLB library for all the strategies, the actual decision process for choosing among the schemes using the above model, and it consists of data movement routines to handle redistribution. Load balancing is achieved by placing appropriate calls to the DLB library to exchange information and redistribute work. The compiler, however, generates code to handle this at run-time.

Code Generation. For the source-to-source code translation from a sequential program to a parallel program using PVM [6] for message passing, with DLB library calls, we use the SUIF compiler from Stanford University. The input to the compiler consists of the sequential version of the code, with annotations to indicate the data decomposition for the shared arrays, and to indicate the loops which have to be load balanced. The compiler generates code for setting up the master processor. This involves broadcasting initial configuration information parameters, such as number of processors, size of arrays, and task IDs, calls to the DLB library for the initial partitioning of shared arrays, final collection of results and DLB statistics (such as number of redistributions, number of synchronizations, and amount of work moved), and a call to a special routine which handles the first synchronization, along with the modeling and strategy selection. It also handles subsequent synchronizations for the centralized schemes. The arrays are initially partitioned equally based on the data distribution specification (BLOCK, CYCLIC, or WHOLE). We currently support *do-all* loops only, with data distribution along one dimension (row or column). The compiler must also generate code for the slave processors, which perform the actual computation. This step includes changing the loop bounds to iterate over the local assignment, and inserting calls to the DLB library checking for interrupts, for sending profile information to the load balancer (protocol dependent), for data redistribution, and if local work stack has run out, for issuing an interrupt to synchronize.

6. EXPERIMENTAL RESULTS

All the experiments were performed on a network of homogeneous Sun (Sparc LX) workstations interconnected via an Ethernet LAN (our model, however, can easily handle processor heterogeneity). External load was simulated as described in Section 4. PVM [6] was used to parallelize the following applications: (1) *Matrix Multiplication* (MXM) and (2) *TRFD*, from the Perfect Benchmark suite [8]. The overhead of the DLB schemes is almost negligible, since they are receiver-initiated, and in the absence of external load, all processors finish work at roughly the same time, usually requiring only one synchronization. The network characterization under PVM was done off-line. We obtained the latency (2414.5 μs), the bandwidth (0.96 MB/s), and models for the different communication patterns (e.g., all-to-one, one-to-all, all-to-all). The experiments were run with P = 4, 16, and with K = 2, 8, respectively. For more detailed results, see [19].

6.1. MXM: Matrix Multiplication

Matrix multiplication is given as $Z = X \cdot Y$ (X is a $n \times r$ and Y an $r \times m$ matrix). We parallelize the outermost loop by distributing the rows of Z and X and replicating Y on the processors. Only the rows of X need to be communicated when we redistribute work ($\mathcal{D} = r$). The work per iteration is uniform and quadratic. We ran experiments with m = 400 for different values of r and n.

Figure 1 shows the execution time for the different DLB schemes, normalized against the case with no dynamic load balancing (with iterations equally partitioned among the processors). We observe that the global distributed (GDDLB) strategy is the best, followed closely by the global centralized (GCDLB) scheme. Local distributed (LDDLB) also does better than local centralized (LCDLB). Moreover, the global schemes are better than the local schemes. However, on 16 processors the gap between the globals and locals becomes smaller. From our earlier discussion (Section 3.1), local strategies incur less communication overhead than global strategies, but the redistribution is not optimal. From the results, it can be observed that if the computation cost (work per iteration) versus the communication cost (synchronization cost, redistribution cost) ratio is large, global strategies are favored. This tilts toward the local strategies as this ratio decreases. The factors that influence this ratio are the work per iteration, the number of iterations, and the number of processors. More processors increase the synchronization cost and should favor the local schemes. However, in the above experiment there is sufficient work to outweigh this trend, and globals are still better for 16 processors. Comparing across distributed and central schemes, the centralized master, and sequential redistribution and instruction send, add sufficient overhead to the centralized schemes to make the distributed schemes better. LCDLB incurs additional overhead due to a delay factor [19], and also due to context switching between the load balancer and the computation slave.

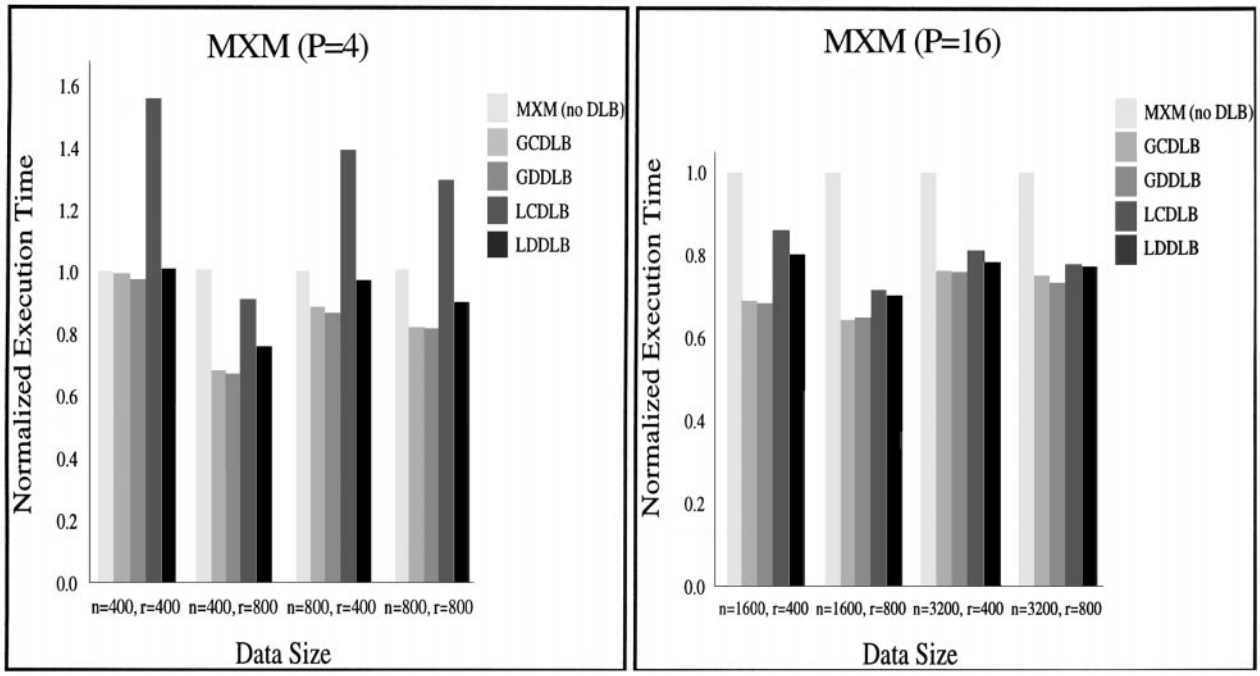

FIG. 1. Matrix multiplication (P = 4, 16).

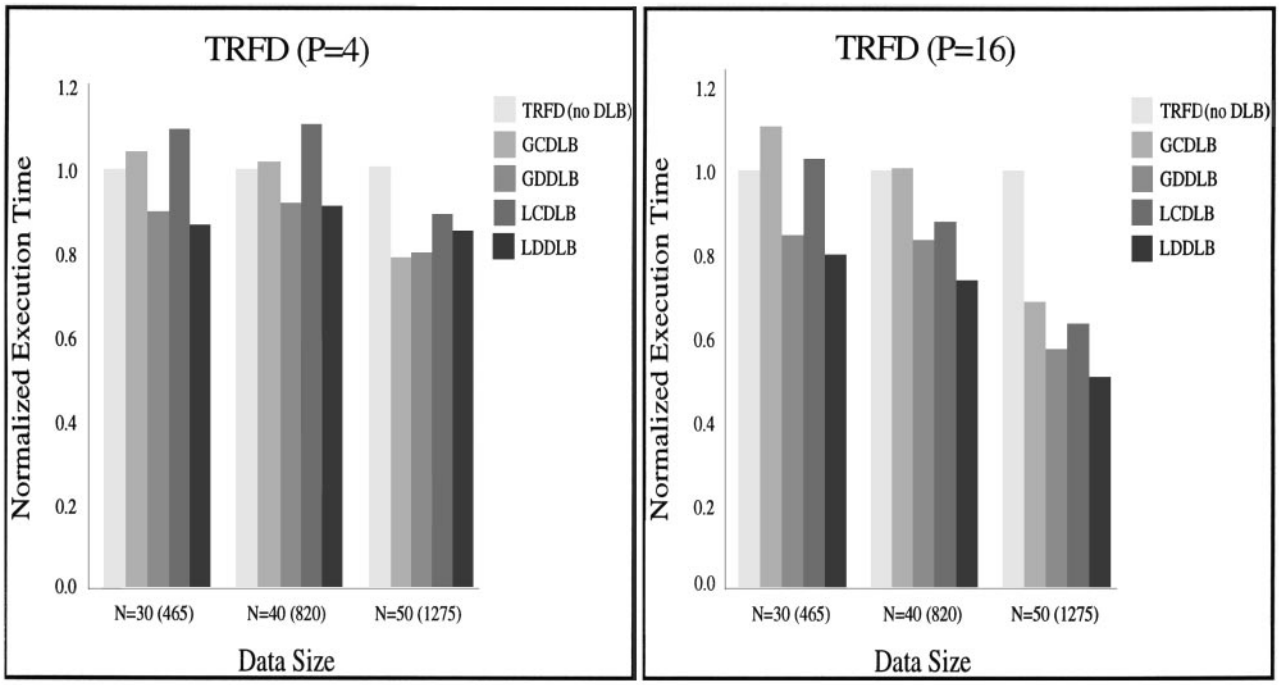

FIG. 2. TRFD (P = 4, 16).

6.2. TRFD

TRFD has two main computation loops, which are load balanced independently, and an intervening sequential transpose. We parallelized the outermost loop of both the loop nests. There is only one major array of size $[n(n + 1)/2] \cdot [n(n + 1)/2]$, used in both the loops. The loop iterations operate on different columns of the array (thus $\mathcal{D} = [n(n + 1)/2]$, the row size). The first loop nest (L1) is uniform, with $n(n + 1)/2$ iterations, and work linear in the array size. The second loop nest has triangular work per iteration, which is made uniform using the *bitonic scheduling* technique [5]. The resulting loop (L2) has $n(n + 1)/4$ iterations, with linear work. We ran experiments with $n = 30, 40, 50$ (i.e., for array sizes of 465, 820, and 1275, respectively).

On four processors, as the data size increases we tend to shift from LDDLB to GDDLB (see Fig. 2). Since the amount of work per iteration is small, the computation vs communication ratio is small, favoring the local distributed scheme on small data sizes. With increasing data size, this ratio increases, and GDDLB does better. Among the centralized schemes GCDLB is better than LCDLB. On 16 processors, however, we find that LDDLB is the best, followed by GDDLB. Among the centralized strategies also LCDLB does better than GCDLB, since the computation vs communication ratio is small. The distributed schemes are thus better than the centralized ones.

6.3. Modeling Results: MXM & TRFD

Table I shows the actual and the predicted best strategy under varying parameters for MXM and TRFD. We observe that the experimental and the predicted best strategy match in most of the cases. For the cases where our prediction differs, the predicted scheme is the second best in the actual run. We present the difference between the two strategies (in terms of time, and as a percentage) in the actual run for these cases. It can be seen that the differences in execution time between the two schemes are quite small at these points. While the table presents the best scheme over several runs, in actuality, one or the other scheme may do better from one run to another, which makes the prediction task extremely difficult. Moreover, this usually happens at the crossover point along the two axes under consideration (local $\leftrightarrow$ global or central $\leftrightarrow$ distributed).

TABLE I
MXM and TRFD: Actual vs Predicted Best DLB Scheme

	Parameters				Difference	
Program	P	Data size	Actual best	Predicted best	Time(s)	% Diff
M×M	4	$n = 400, r = 400, m = 400$	GD	GD		
	4	$n = 400, r = 800, m = 400$	GD	GD		
	4	$n = 800, r = 400, m = 400$	GD	GD		
	4	$n = 800, r = 800, m = 400$	GD	GD		
	16	$n = 1600, r = 400, m = 400$	GD	GC	1.2 s	0.7%
	16	$n = 1600, r = 800, m = 400$	GC	GD	3.7 s	1.1%
	16	$n = 3200, r = 400, m = 400$	GD	GD		
	16	$n = 3200, r = 800, m = 400$	GD	GD		
TRFD	4	$n = 30$(465), L1	LD	GD	0.9 s	8.2%
	4	$n = 40$(820), L1	LD	LD		
	4	$n = 50$(1275), L1	LD	LD		
	4	$n = 30$(465), L2	LD	GD	0.2 s	3.5%
	4	$n = 40$(820), L2	GD	LD	1.5 s	6.2%
	4	$n = 50$(1275), L2	GD	GD		
	16	$n = 30$(465), L1	LD	LD		
	16	$n = 40$(820), L1	LD	LD		
	16	$n = 50$(1275), L1	LD	LD		
	16	$n = 30$(465), L2	LD	LD		
	16	$n = 40$(820), L2	LD	LD		
	16	$n = 50$(1275), L2	LD	LD		

For example, for loop2 (L2) in TRFD on $P = 4$, as the data size increases the trend starts shifting from LDDLB to GDDLB.

7. CONCLUSIONS

In this paper we analyzed both *global* and *local*, and *centralized* and *distributed* interrupt-based receiver-initiated dynamic load balancing strategies on a network of workstations with transient external load per processor. We showed that different strategies are best for different applications under varying parameters, such as the number of processors, data size, iteration cost, and communication cost. Presenting a hybrid compile and run-time process, we showed that it is possible to customize the dynamic load balancing scheme for a program. Given the host of dynamic scheduling strategies proposed in the literature, such analysis would be useful to a parallelizing compiler. To take the complexity away from the programmer, we also automatically transform an annotated sequential program to a parallel program with the appropriate calls to the run-time dynamic load balancing library.

REFERENCES

1. Arabe, J., *et al.* Dome: Parallel programming in a heterogeneous multi-user environment. CMU-CS-95-137 30786, Carnegie Mellon University, Apr. 1995.
2. Berman, F., *et al.* Application-level scheduling on distributed heterogeneous networks. *Supercomputing* (Nov. 1996).
3. Blumofe, R. D., and Park, D. S. Scheduling large-scale parallel computations on network of workstations. *3rd IEEE International Symposium on High-Performance Distributed Computing.* Aug. 1994.
4. Cheung, A. L., and Reeves, A. P. High performance computing on a cluster of workstations. *1st IEEE International Symposium on High-Performance Distributed Computing.* July 1992.
5. Cierniak, M., Li, W., and Zaki, M. J. Loop schedulng for heterogeneity. In *4th IEEE International Symposium on High-Performance Distributed Computing* [also TR 540], U. Rochester. Aug. 1995.
6. Geist, A. *et al.* PVM user guide and reference manual. TM12187, Oak Ridge National Laboratory, May 1993.
7. Grimshaw, A. S., *et al.* Metasystems: An approach combining parallel processing and heterogeneous distributed computing systems. *J. Parallel Distrib. Comput.* **21,** 3 (1994).
8. Kipp, L. *Perfect Benchmarks Doc. Suite 1.* CSRD, Univ. Illinois, Urbana–Champaign, Oct. 1993.
9. Li, W., and Pingali, K. Access normalization: Loop restructuring for NUMA compilers. *ACM Trans. Comput. Systems* **11,** 4 (Nov. 1993).
10. Lin, F., and Keller, R. The gradient model load balancing method. *IEEE Trans. Software Engng.* **13** (Jan. 1987).
11. Markatos, E., and LeBlanc, T. Using processor affinity in loop scheduling on shared-memory multiprocessors. *IEEE Trans. Parallel Distrib. Systems* **5,** 4 (Apr. 1994).
12. Nedeljkovic, N., and Quinn, M. J. Data-parallel programming on a network of heterogeneous workstations. *1st IEEE International Symposium High Performance Distributed Computing.* 1992.
13. Nishikawa, H., and Steenkiste, P. A general architecture for load balancing in a distributed-memory environment. *13th IEEE International Conference on Distributed Computing.* May, 1993.
14. Polychronopoulos, C. D. *Parallel Programming and Compilers.* Kluwer Academic, 1988.
15. Saletore, V. A., Jacob, J., and Padala, M. Parallel computations on the charm heterogeneous workstation clusters. *3rd IEEE International Symposium on High-Performance Distributed Computing.* 1994.
16. Siegell, B. Automatic generation of parallel programs with dynamic load balancing for a network of workstations. CMU-CS-95-168 30880, Carnegie Mellon Univ., May 1995.
17. Yan, Y., Zhang, X., and Song. Y. An effective and practical performance prediction model for parallel computing on non-dedicated heterogeneous NOW. *J. Parallel Distrib. Comput.* **38,** 1 (1996).
18. Zaki, M., Li, W., and Cierniak, M. Performance impact of processor and memory heterogeneity in a NOM. *4th Heterogeneous Computing Wkshp* [also TR574], Univ. Rochester, 1995.
19. Zaki, M., Li, W., and Parthasarathy, S. Customized dynamic load balancing for a NOW. *5th IEEE International Symposium on High-Performance Distributed Computing* [also TR 602], Univ. Rochester, 1996.

Received March 1, 1996; revised April 1, 1997; accepted April 30, 1997

MOHAMMED JAVEED ZAKI is a Ph.D. candidate in computer science at the University of Rochester. He received his M.S. in computer science from the University of Rochester in 1995. His research interests include parallel data mining, heterogeneous computing, and parallelizing compilers.

WEI LI has been an assistant professor in computer science at the University of Rochester since he received his Ph.D. in computer science from Cornell University in 1993. His research interests include compilers for parallel architectures, programming languages, distributed systems, parallel data mining, and scientific computing. He receivd a NSF Research Initiation Award in 1994 and a best paper award in ASPLOS'92.

SRINIVASAN PARTHASARATHY is a Ph.D. candidate in computer science at the University of Rochester. He received an M.S. in electrical engineering from the University of Cincinnati in 1994. He also recevied an M.S. in computer science from the University of Rochester in 1996. His research interests include compilers for parallel and distributed systems, distributed shared memory architectures, and data-mining.