

Customized Dynamic Load Balancing for a Network of Workstations *

Mohammed Javeed Zaki, Wei Li, Srinivasan Parthasarathy
{zaki,wei,srini}@cs.rochester.edu

The University of Rochester
Computer Science Department
Rochester, New York 14627

Technical Report 602

December 1995

Abstract

Load balancing involves assigning to each processor, work proportional to its performance, minimizing the execution time of the program. Although static load balancing can solve many problems (e.g., those caused by processor heterogeneity and non-uniform 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 examine the behavior of global vs local, and centralized vs distributed, load balancing strategies. We show that different schemes are best for different applications under varying program and system parameters. Therefore, customized load balancing schemes become 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 runtime library for load balancing.

Keywords: Customized Dynamic Load Balancing, Network of workstations, Performance Modeling, Compile/Run-Time system.

The University of Rochester Computer Science Department supported this work.

*This work was supported in part by an NSF Research Initiation Award (CCR-9409120) and ARPA contract F19628-94-C-0057.

1 Introduction

Network of Workstations (NOW) 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), a NOW is becoming increasingly competitive compared to expensive parallel machines. However, NOWs are much harder to program than dedicated parallel machines. For example, multi-user environment with sharing of CPU and network may contribute to varying performance. Heterogeneity in processors, memory, and network are also contributing factors.

The common programming style on these machines uses explicit message passing to share data, while each process has its own private address space. The programs are usually written using the Single Program Multiple Data Stream (SPMD) model, i.e., all processes essentially execute the same program, but on different data-sets. In these programs, it is typically the loops which provide a rich source of parallelism. The iterations of the loop are partitioned among the available processors, which execute them in parallel. Scheduling parallel loops involves finding the appropriate granularity of tasks so that overhead is kept small, and load is balanced among the processors.

Load balancing involves assigning the tasks to each processor in proportion to its performance. This 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 non-uniform iteration execution times. While static scheduling avoids the run-time scheduling overhead, in a multi-user environment with load changes on the nodes, a more dynamic approach is warranted. A major challenge, is to minimize the additional run-time overhead.

In this paper we compare schemes for dynamically balancing the load across the processors operating under differing external loads. We show that different strategies may be best in differing circumstances for the same application, or across applications. Different phases of the same application may also require different strategies. We also present a compile and run-time modeling and decision process which, combined with a run-time load balancing library, would enable us to choose the best strategy. Since there are a large number of loop scheduling techniques which have been proposed in the literature, this analysis helps a parallelizing compiler in customizing the dynamic load balancing strategy for a given application.

The rest of the paper is organized as follows. The next section looks at related work. In Section 3 we present our load balancing approach, and a description of our strategies. In section 4 we describe a compile-time model for the different strategies, and a compile-time decision methodology for choosing among the different load balancing schemes. Section 5 describes how the model is used to customize the schemes, and how the compiler automatically generates the code incorporating our methodology. Section 6 provides experimental evidence indicating that different load balancing strategies are best for different applications under varying parameters. This section also contains the modeling results. Finally, in Section 7 we present our conclusions.

2 Related Work

In this section we look at some of the load balancing schemes which have been proposed in the literature.

2.1 Static Scheduling

Compile-time *static* loop scheduling is efficient and introduces no additional runtime overhead. For UMA (Uniform Memory Access) parallel machines, usually loop iterations can be scheduled in *block* or *cyclic* fashion [17]. For NUMA (Non-Uniform Memory Access) parallel machines, loop scheduling has to take data distribution into account [11]. Static scheduling algorithms for heterogeneous programs, processors, and network were proposed in [3, 4]. In [8] another static scheme for heterogeneous systems was proposed, which uses a run-time system to select the appropriate number of processors, and the ratios to allocate work to the processors.

2.2 Dynamic Scheduling

The dynamic scheduling strategies fall under different models, which include the *task queue model*, *diffusion model*, and schemes based on predicting the future from past loads.

Task Queue Model Many dynamic scheduling algorithms proposed in the literature based on the are *task queue model*, which targets shared-memory machines. In these schemes there is a central task queue of loop iterations. Once the processors have finished their assigned portion, more work is obtained from this queue. The simplest approach in this model is *self-scheduling* [22], where each processor is allocated only one iteration at a time, which leads to high synchronization cost. To alleviate this problem *fixed-size chunking* was proposed [10], where each processor is allocated K iterations instead of one. The potential for load imbalance, however, is still present. In *guided self-scheduling* [18], the chunk size is changed at run-time. Each processor is assigned $1/P$ -th of the remaining loop iterations, where P denotes the number of processors. Although the large chunk sizes in the beginning reduce synchronization, they can cause serious imbalances in non-uniform loops. Moreover, this scheme degenerates to the case of self-scheduling towards the end due to small chunk sizes. Factoring [9], allocates iterations in batches, which are partitioned equally among all the processors. The current size of a batch is exactly half the size of the previous batch. Other schemes in this model are *adaptive guided self-scheduling* [5], which includes a random back-off to avoid contention for the task queue, and assigns iterations in an interleaved fashion to avoid imbalance; *trapezoidal self-scheduling* [23], which linearly decreases the number of iterations allocated to each processor; *tapering* [14], which is suitable for irregular loops, and uses execution profiles to select a chunk size that minimizes the load imbalance; *safe self-scheduling* [13], which uses a static phase where each processor is allocated a chunk of iterations, and a dynamic phase during which the processors are self scheduled to even out the load imbalances; and *affinity scheduling* [15] which takes processor affinity into account while scheduling. Under this scheme, all the processors are initially assigned an

equal chunk taking into account data reuse and locality. Once the local chunk is finished, an idle processor finds the most overworked processor and removes $1/P$ of iterations (where P is the number of processors) from this processor's work queue.

Diffusion Model Other approaches include *diffusion models* with all the work initially distributed, and with work movement between adjacent processors if an imbalance is detected between their load and their neighbor's. An example is the *gradient model* [12] approach.

Predicting the Future Another approach is to predict future performance based on past information. For example, in Data parallel C [19], loop iterations are mapped to virtual processors, and these virtual processors are assigned to the physical processors based on past load behavior. The approach is global distributed, where the processor's load is given as the average computation time per virtual processor, and load balancing involves periodic information exchanges. Dome [1] implements a global central scheme and a local distributed scheme. The performance metric used is the rate at which the processors execute the dome program, and load balancing involves periodic exchanges. Siegell [21] also presented a global centralized scheme, with periodic information exchanges, and where the performance metric is the iterations done per second. The main contribution of this paper was the methodology for automatic generation of parallel programs with dynamic load balancing. In Phish [2], a local distributed receiver-initiated scheme is described, where the processor requesting more tasks, called the *thief*, chooses a *victim* at random from which to steal more work. If the current victim cannot satisfy the request, another victim is selected. CHARM [20] implements a two phased scheme. Initially, in the static phase, work is assigned to the processors proportional to their speed, and inversely proportional to the load on the processor. The dynamic phase implements a local distributed receiver-initiated scheme. The information exchanged is the *Forecasted Finish Time* (FFT), i.e., the time for the processor to finish the remaining work. If the FFT falls below a threshold, the node requests a neighbor with higher FFT for more work. If the request cannot be satisfied, another neighbor is selected.

In our approach, instead of periodic exchanges of information, we have a interrupt-based receiver-initiated scheme. Moreover, we look at both central vs distributed, and local vs global approaches. In the local schemes, instead of random selection of a processor from which to request more work, work is exchanged among all the neighbors (the number of neighbors is selected statically). These strategies are explained in more detail in the next section. In [16], an approach was presented, where a user specifies homogeneous load balancers for different tasks within a heterogeneous application. They also present a global load balancer that handles the interactions among the different homogeneous load balancers. However, 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)

The goal of load balancing is to assign to each processing node work proportional to its performance, thereby minimizing the execution time of the application. In this section we describe our dynamic load balancing approach, and the different strategies we chose to study our concepts.

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 the actual data movement. The data is moved directly between the slaves, and the load balancing decisions are made by the *load balancer*.

3.1 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, who then send their performance profiles to the load balancer.

3.2 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.

3.3 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, and move the work only if there is 10% improvement in execution time. If it is profitable to move work, then the load balancer broadcasts the new distribution information to the processors. The work is then re-distributed between the slaves.

3.4 Profitability Analysis

There is a trade-off between the benefits of moving work to balance load, and the cost of work movement. From our experiments, we found that it is generally better to exclude the cost of the actual work movement. The reason is that inaccuracies in work movement cost estimation may predict a higher cost for the re-distribution, thereby nullifying the potential benefits of moving work. This would cancel some of the work movement instructions. Since we synchronize only when a processor needs more work, cancelling work re-distribution means that this processor will now become idle, lowering the overall utilization of the

processor. We thus redistribute work as long as the potential benefit (predicted execution time, excluding the cost of actual work movement) of the new assignment results in at least a 10% improvement.

3.5 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 if the load balancer is distributed among the processors, respectively. For all the strategies, the compiler initially distributes the iterations of the loop equally among all the processors.

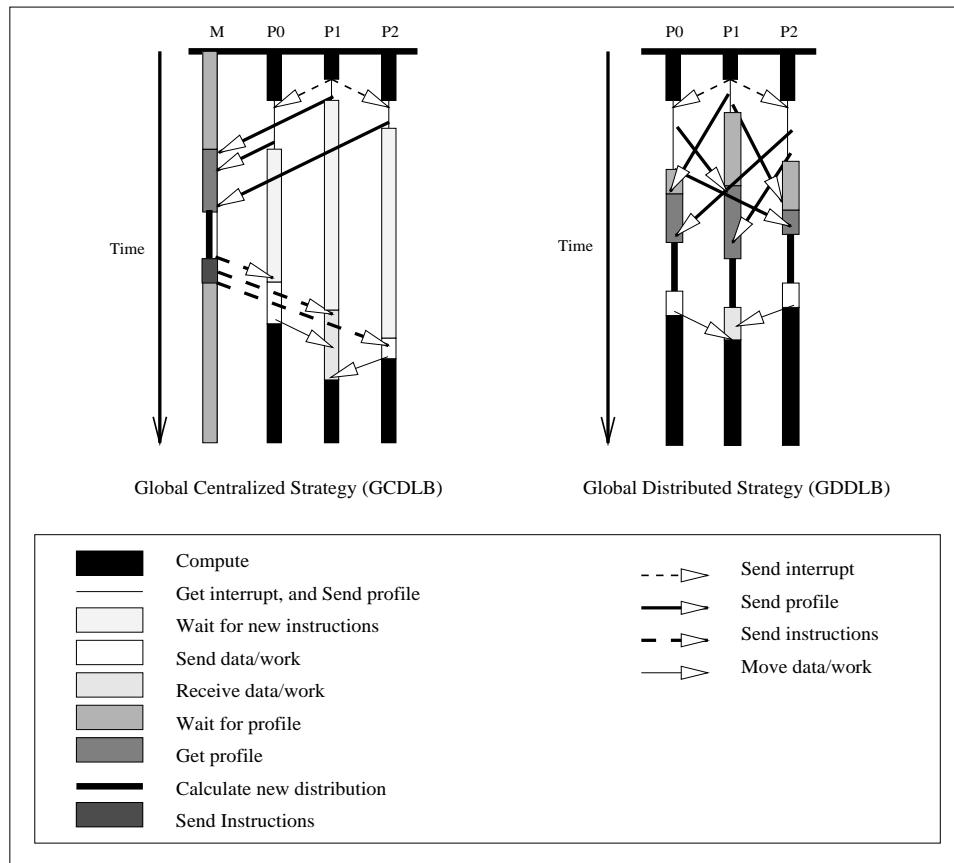


Figure 1: Centralized vs. Distributed Strategies

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 given below.

- **Global Centralized DLB (GCDLB):** In this scheme the load balancer is located on a master processor (centralized). After calculating the new distribution, and profitability of work movement, 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.
- **Global Distributed DLB (GDDLb):** In this scheme the load balancer is replicated on all the processors. So, unlike GCDLB, where profile information is sent to only the master, in GDDLb, the profile information is broadcast to every other processor. This also eliminates the need for the load balancer to send out instructions, as that information is available to all the processors. The receiving processors wait for work, while the sending processors ship the data. Figure 1 highlights the differences between the two strategies pictorially.

Local Strategies

In the local schemes, the processors are partitioned into different groups of size K . This partition can be done by considering the physical proximity of the machines, as in *K-nearest neighbors* scheme. The groups can also be formed in a *K-block* fashion, or the group members can be selected randomly. Furthermore, the groups can remain fixed for the duration of execution, or the membership can be changed dynamically. We used the K -block fixed-group approach in our implementation. In these schemes the load balancing decisions are only done within a group. In fact the Global strategies are an instance of the respective local strategies, where the group size, K , equals the number of processors. The two local strategies we look at are:

- **Local Centralized DLB (LCDLB):** This scheme is similar to GCDLB. The fastest processor in a group only interrupts the other processors in that group. There is one centralized load balancer, which asynchronously handles all the different groups. Once it receives the profile information from one group, it send instructions for re-distribution for that group before proceeding to the other groups.
- **Local Distributed DLB (LDDLb):** In this scheme, the load balancer is replicated on all the processors, but the profile information is broadcast only to the members of the group.

3.6 Discussion

These four strategies lie at the four extreme points on the two axes. For example, in the local approach, there is no exchange of work between different groups. In the local centralized (LCDLB) version, we have only one master load balancer, instead of having one master per group. Furthermore, in the distributed strategies we have full replication of the load balancer. There are many conceivable points in between, and many other hybrid strategies possible. Exploring the behavior of these strategies is part of future work. At the present

time, we believe that the extreme points will serve to highlight the differences, and help to gain a basic understanding of these schemes.

Global vs. Local The advantage of the global schemes is that the work re-distribution is optimal, based on information known till that point (the future is unpredictable, so it's not optimal for the whole duration). However, synchronization is more expensive. On the other hand, in the local schemes, the work re-distribution is not optimal, resulting in slower convergence. However, the amount of communication or synchronization cost is lower. Another factor affecting the local strategies is the difference in performance among the different groups. 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. It could also be fixed by having dynamic group memberships, instead of having static partitions. In this paper we restrict our attention, to the static group partition scheme only.

Centralized vs. Distributed In the centralized schemes, the central point of control could prevent the scalability of the strategy to a large number of machines. The distributed schemes help solve this problem. However, in these schemes the synchronization involves an all-to-all broadcast. The centralized schemes require an all-to-one profile send, which is followed by a one-to-all instruction send. There is also a trade-off 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

In this section, we present a compile and run-time modeling and decision process for choosing among the different load balancing strategies. We begin with a discussion of the different parameters that may influence the performance of these schemes. This is followed by the derivation of the total cost function for each of these approaches in terms of the different parameters. Finally we show how this modeling is used.

4.1 Modeling Parameters

The various parameters which affect the modeling are presented below.

- **Processor Parameters:** These give information about the different processors available to the application.
 - **Number of Processors:** We assume that we have a fixed number of processors available for the computation. This number is specified by the user, and is denoted as P .

- **Processor Speeds:** This specifies the ratio of a processor’s performance w.r.t a base processor. Since this ratio is application specific [25], we can obtain this by a profiling run. We may also try to predict this at compile-time. The speed for processor i is denoted as S_i .
- **Number of Neighbors:** This is used for the local strategies, and may be dictated by the physical proximity of the machines, or it may be user specified. It is denoted as K .
- **Program Parameters:** These parameters give information about the application.
 - **Data Size:** This could be different for different arrays (it could also be different for the different dimensions of the same array). This is denoted as N_{ad} , where d specifies the dimension, and a specifies the array name.
 - **Number of Loop Iterations:** This is usually some function of the data size, and is denoted as $\mathcal{I}_i(N_{ad})$, where i specifies the loop.
 - **Work per Iteration:** The amount of work is measured in terms of the number of basic operations per iteration, and is a function of the data size. This is denoted as $\mathcal{W}_{ij}(N_{ad})$, where i specifies the loop, and j specifies the iteration number.
 - **Intrinsic Communication:** This specifies the amount of communication per iteration, which is inherent to the algorithm. This is denoted as $\mathcal{IC}_{ij}(P, N_{ad})$, where i is the loop, and j is the iteration.
 - **Data Communication:** This is a per array cost, which indicates the number of bytes that need to be communicated per iteration. In a row or a column distribution of the data arrays, this is simply the number of the columns, and number of rows respectively. This is denoted as $\mathcal{DC}_{aij}(N_{ad})$, where a is the array name, i is the loop, and j is the iteration.
 - **Time per Iteration:** This specifies the time it takes to execute an iteration of a loop on the base processor. It is denoted as $\mathcal{T}_{ij}(\mathcal{W}, \mathcal{IC})$, where i is the loop, and j is the iteration. Since this time is w.r.t. the base processor, the time to execute an iteration on processor k is simply \mathcal{T}_{ij}/S_k . This time could be obtained by profiling, static analysis, or with the help of the programmer.
- **Network Parameters:** These specify the properties of the interconnection network.
 - **Network Latency:** This is the time it takes to send a single byte message between processors. Although the communication latency could be different for the various processor pairs, we assume it to be uniform, and denote it as \mathcal{L} .
 - **Network Bandwidth:** This is number of bytes that can be transferred per second over the network. It includes the cost of packing, receiving, and the “real” communication time in the physical medium. We denote this as \mathcal{B} .
 - **Network Topology:** This influences the latency and bandwidth between pairs of processors. It also has an impact on the number of neighbors (for local strategies), and may help in reducing expensive communication while re-distribution.

In this paper, however, 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:
 - **Maximum Load:** This specifies the maximum amount of load per processor, and is denoted as m_ℓ . In our experiments we set $m_\ell = 5$.
 - **Duration of Persistence:** The load value for a processor is obtained by using a random number generator to get a value between zero and the maximum load. The duration of persistence, denoted as t_ℓ , indicates the amount of time before the next invocation of the random number generator, i.e., we simulate a discrete random load function, with a maximum amplitude given by m_ℓ , and the discrete block size given by t_ℓ . A small value for t_ℓ 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 k -th duration of persistence. Figure 2 shows the load function for a processor.

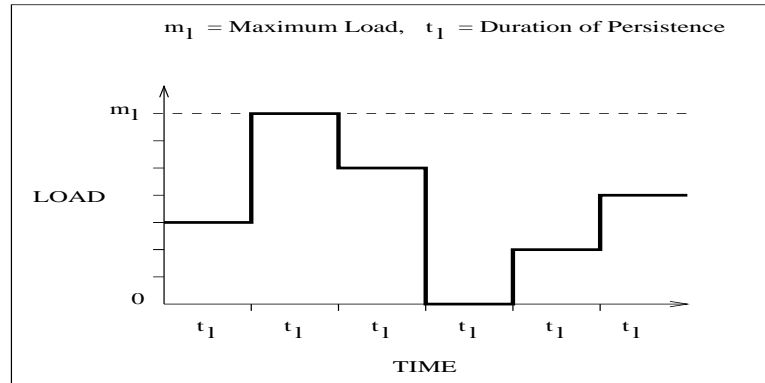


Figure 2: Load Function

4.2 Modeling the Strategies – Total Cost Derivation

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. We look at each of these below:

Cost of Synchronization

The synchronization involves the sending of interrupt from the fastest processor to the other processors, who then send their performance profile to the load balancer. This cost is specified in terms of the kind of communication required for the synchronization. The cost for the different strategies is given below:

- GCDLB : $\xi = \text{one-to-all}(P) + \text{all-to-one}(P)$
- GDDL B : $\xi = \text{one-to-all}(P) + \text{all-to-all}(P^2)$
- LCDLB (per group) : $\xi = \text{one-to-all}(K) + \text{all-to-one}(K)$
- LDDL B (per group) : $\xi = \text{one-to-all}(K) + \text{all-to-all}(K^2)$

Cost of Distribution Calculation

This cost is usually quite small, and we denote it as δ . This calculation is replicated in the distributed strategies. The cost for the local schemes would be slightly cheaper, since each group has only K instead of P processors. However, we ignore this effect.

Cost of Data Movement

We now present our analysis to calculate the amount of data movement and the number of messages required to re-distribute work.

Notation Let $\chi_i(j)$ denote the iteration distribution, and let $\gamma_i(j)$ denote the number of iterations left to be done by processor i at the j -th synchronization point. Let $\Gamma(j) = \sum_{i=1}^P \gamma_i(j)$, and let t_j denote the time at which the j -th synchronization happens.

Effect of discrete load The *effective speed* of processor is inversely proportional to the amount of load on it, which is given as $S_i/(\ell_i(k) + 1)$, where $\ell_i(k) \in \{0, \dots, m_\ell\}$. Since the performance metric used by the different schemes is the processor performance since the last synchronization point, the processor's performance is given as the average effective speed over that duration. Let the $(j - 1)$ -th synchronization be during the a -th duration of persistence, i.e., $a = \lceil t_{j-1}/t_\ell \rceil$. Similarly, let $b = \lceil t_j/t_\ell \rceil$. Then the *average effective speed* of processor i is given as

$$\frac{\sum_{k=a}^b S_i/(\ell_i(k) + 1)}{b - a + 1} = S_i / \left(\frac{b - a + 1}{\sum_{k=a}^b 1/(\ell_i(k) + 1)} \right) = S_i/\lambda_i(j)$$

where $\lambda_i(j)$ denotes the *effective load* on processor i between the j -th and the previous synchronization.

Total iterations done We now analyze the effect of the j -th synchronization. We will first look at the case of uniform loops, i.e., where each iteration of the loop takes the same time.

Uniform Loops We will use \mathcal{T} for the time per iteration. At the end of the $(j-1)$ -th synchronization, each processor had $\chi_i(j-1)$ iterations assigned to it. Let f denote the first processor to finish its portion of the work. Then the time taken by processor f is given as

$$t = t_j - t_{j-1} = \chi_f(j-1)\mathcal{T}(\lambda_f(j)/S_f)$$

The iterations left to be done on processor i is simply the old distribution minus the iterations done in time t , i.e., $\gamma_i(j) = \chi_i(j-1) - \lceil \frac{t}{\mathcal{T}\lambda_i(j)/S_i} \rceil$. Using the value of t from above, we get

$$\gamma_i(j) = \chi_i(j-1) - \chi_f(j-1) \left(\frac{\lambda_f(j)}{S_f} \right) \left(\frac{S_i}{\lambda_i(j)} \right) \quad (1)$$

Non-Uniform Loops We now extend the analysis for non-uniform loops. The time taken by processor f to finish its portion of the work is given as

$$t = t_j - t_{j-1} = \sum_{k=1}^{\chi_f(j-1)} \mathcal{T}_k \lambda_f(j) / S_f$$

where k is in set of iterations assigned to processor f . The iterations done by processor i in time t , denoted by $\aleph \leq \chi_i(j-1)$, is now given by the expression

$$\sum_{k'=1}^{\aleph} \mathcal{T}_{k'} \lambda_i(j) / S_i \geq t = \sum_{k'=1}^{\aleph} \mathcal{T}_{k'} \geq \left(\frac{\lambda_f(j)}{S_f} \right) \left(\frac{S_i}{\lambda_i(j)} \right) \sum_{k=1}^{\chi_f(j-1)} \mathcal{T}_k$$

The iterations left to be done on processor i is then given as

$$\gamma_i(j) = \chi_i(j-1) - \aleph \quad (2)$$

New distribution The total amount of work left among all the processors is given as $\Gamma(j) = \sum \gamma_i(j)$. We now distribute this work proportional to the average effective speed of the processors, i.e.

$$\chi_i(j) = \left(\frac{S_i / \lambda_i(j)}{\sum_{k=1}^P S_k / \lambda_k(j)} \right) \Gamma(j) \quad (3)$$

Recall that initially we start out with equal work distribution among all the processors, therefore we have $\lambda_i(0) = 1$, $\chi_i(0) = \mathcal{I}_i(N_{ad})/P$, and $\gamma_i(0) = \chi_i(0) \forall i \in 1, \dots, P$. These equations together with equations 1, 2, and 3 give us recurrence functions which can be solved to obtain the total iterations left to be done, and the new distribution at each synchronization point. The termination condition occurs when there is no more work left to be done, i.e.,

$$\Gamma(\eta) = 0 \quad (4)$$

where η is the number of synchronization points required.

Amount of work moved The amount of basic units of work (usually iterations) moved during a synchronization is given as

$$\alpha(j) = \frac{1}{2} \left(\sum_{i=1}^P |\gamma_i(j) - \chi_i(j)| \right)$$

Data Movement Cost The movement of iterations entails movement of data arrays. The number of messages required to move the work and data arrays, denoted by $\beta(j)$ can be calculated from the old and new distribution values. The total cost of data movement is now given by the expression

$$\kappa(j) = \beta(j)\mathcal{L} + \alpha(j) \sum_a [\mathcal{DC}_a/\mathcal{B}] \quad (5)$$

where a belongs to the set of arrays that need to be re-distributed.

Cost of Sending Instructions

This cost is only incurred by the centralized schemes, since the load balancer has to send the work and data movement instructions to the processors. The number of instructions is the same as $\beta(j)$, which is the number of messages required to move data, since instructions are only sent to the processors which have to send data. The cost of sending instructions is, therefore, $\psi(j) = \beta(j)\mathcal{L}$ for the centralized schemes, and $\psi(j) = 0$ for the distributed schemes.

Total Cost

Global strategies The above set of recurrence relations can be solved to obtain the cost of data movement (see equation 5), and to calculate the number of synchronization points (see equation 4), thereby getting the total cost of the global strategies as

$$\mathcal{TC} = \eta(\xi + \delta) + \sum_{j=1}^{\eta} [\kappa(j) + \psi(j)]$$

where ξ is the synchronization cost, η is the number of synchronizations, δ is the re-distribution calculation cost, and $\kappa(j)$ is the data movement cost and $\psi(j)$ the cost of sending instructions for the j -th synchronization.

Local strategies In the local centralized (LCDLB) strategy, even though the load balancer is asynchronous, the assumption that groups can be treated independently from the others may not be true. This is because the central load balancer goes to another group only once it has finished calculating the re-distribution and sending instructions for the current group.

Delay Factor This effect is modeled as a delay factor for each group, which depends on the time for the synchronization of the different groups, and is given as $\Delta_g(j) = \sum_{k=1}^{\nu(j)} [\delta + \psi_k(j)]$, where $\nu(j)$ is the number of groups already waiting in the queue for the central load balancer. Note that in the local distributed scheme, the absence of a central load balancer eliminates this effect (i.e., $\Delta_g(j) = 0$). There may still be some effect due to overlapped synchronization communication, but we do not model this.

For the local schemes, the analyses in the previous subsections still hold, but we have a different cost per group for each of the different categories. The total cost per group is given as

$$\mathcal{TC}_g = \eta_g(\xi + \delta) + \sum_{j=1}^{\eta_g} [\kappa_g(j) + \psi_g(j) + \Delta_g(j)]$$

The total cost of the local strategy is simply the time taken by the last group to finish its computation, i.e., $\mathcal{TC} = \text{MAX}_{g=1}^{\lceil P/K \rceil} \{\mathcal{TC}_g\}$.

4.3 Decision Process – Using the Model

Since all the information used by the modeling process, like the number of processors, processor speeds, data size, number of iterations, iteration cost, etc., 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. The actual decision making for committing to a scheme is deferred until run-time when we have complete information about the system.

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. This can be seen by using equation 1 above, and plugging $j = 1$, i.e., the first synchronization point, $\chi_f(0) = \mathcal{I}_f(N_{ad})/P$, and summing over all processors, to obtain the total iterations done at the first synchronization point as $(\sum_{i=1}^P [\mathcal{I}_f(N_{ad})/P(\lambda_f(1)/S_f)(S_i/\lambda_i(1))]) > \mathcal{I}_f(N_{ad})/P$. At this time we also know the load function and 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.

5 Compiler and Run-Time Systems

In this section we describe how our compiler automatically transforms annotated sequential code into code that can execute in parallel, and that calls routines from the run-time system using the dynamic load balancing library where appropriate.

5.1 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. The compiler also helps to generate symbolic cost functions for the iteration cost and communication cost.

5.2 Code Generation

For the source-to-source code translation from a sequential program to a parallel program using PVM [7] for message passing, with DLB library calls, we use the Stanford University Intermediate Format (SUIF) [24] compiler. 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.

```
SEQUENTIAL CODE:
for i = 0, R
  for j = 0, R2
    for k = 0, C
      Z(i,j) += X(i,k) * Y(k,j)

TRANSFORMED CODE:
DLB_init(argcnt, &dlb, P, K, task_ids, master_id,
         &DLB_array_Z, &DLB_array_X, &DLB_array_Y);
DLB_scatter_data(&dlb)
if (master) DLB_master_sync(&dlb);
else {
  while (dlb.more_work) {
    for (i = dlb.start; i < dlb.end && dlb.more_work; i++) {
      for (j = 0; j < R2; j++)
        for (k = 0; k < C; k++)
          Z(i,j) += X(i,k) * Y(k,j);
      if (DLB_slave_sync(&dlb) && dlb.interrupt)
        DLB_profile_send_move_work(&dlb);
    }
    if (dlb.more_work) {
      DLB_send_interrupt(&dlb);
      DLB_profile_send_move_work(&dlb);
    }
  }
}
DLB_gather_data(&dlb)
```

Figure 3: Code Generation

The compiler generates code for setting up the master processor (pseudo-master in the distributed schemes, which is only responsible for the first synchronization, initial scattering, and final gathering of arrays). This involves broadcasting initial configuration information parameters, like number of processors, size of arrays, task ids, etc., 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, amount of work moved, etc.) and a call to the *DLB_master_sync()* 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 (we currently support the BLOCK, CYCLIC and WHOLE data distribution annotations along a given dimension).

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. The sequential matrix multiplication code and the code generated by the compiler with appropriate calls to the DLB library, is highlighted in figure 3. In the figure *dlb.more_work* is a flag which indicates whether this processor is active. It becomes false when there are no more iterations assigned to the processor. This may happen if there is no more work left, or if the processor is extremely slow, and all the work migrates to the other processors. For each shared array we also have an *DLB_array* structure, which holds information about the arrays, like the number of dimensions, array size, element type, and distribution type. This structure is also filled by the compiler, and is used by the run-time library to scatter, gather, and redistribute data.

6 Experimental Results

In this section we present our experimental and modeling results. We first present experimental evidence showing that different strategies are better for different applications under varying parameters. We also present our modeling results for the applications

All the experiments were performed on a network of Sun workstations, interconnected via an Ethernet LAN. Applications used C as the source code language, and were run on dedicated machines, i.e., there were no other users on the machines. External load was simulated within our programs as described in section 4. PVM [7] was used to parallelize the applications. PVM (Parallel Virtual Machine), is a message passing software system mainly intended for network based distributed computing on heterogeneous serial and parallel computers. PVM supports heterogeneity at the application, machine and network level, and supports coarse grain parallelism in the application.

The applications we consider are given below:

- Matrix Multiply (MXM): Multiplication of two matrices, given as $Z = X \cdot Y$. The data size for each array is given as $Z = R \cdot C$, $X = R \cdot R2$, and $Y = R2 \cdot C$.
- TRFD: TRFD is part of the Perfect Benchmark application suite [6]. It simulates the computational aspects of two-electron integral transformations. We used a modified version of TRFD, in the C programming language, which was enhanced to exploit the parallelism.

6.1 Network Characterization

The network characterization is done off-line. We measure the latency and bandwidth for the network, and we obtain models for the different types of communication patterns. The

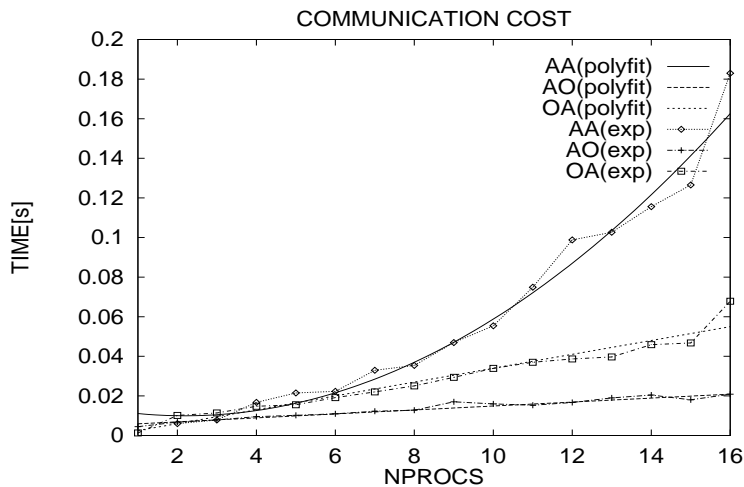


Figure 4: Communication Cost

latency obtained with PVM is $2414.5 \mu s$, and bandwidth is 0.96 Mbytes/s . Figure 4 shows the experimental values (exp), and the cost function obtained from the experimental values by simple polynomial fitting (polyfit), for the all_to_all (AA), all_to_one (AO), and one_to_all (OA) communication patterns.

6.2 MXM : Matrix Multiplication

Matrix multiplication has only one computation loop nest (see figure 3). We parallelize the outer most loop i , by distributing the rows of Z and X , and replicating Y on the processors. Only the rows of array X need to be communicated when we re-distribute work. The data communication is, therefore, given as $\mathcal{DC} = \mathcal{N}_{X^2} = C$. There is no intrinsic communication ($\mathcal{IC} = 0$). The work per iteration is uniform, and is given as $\mathcal{W} = C \cdot R^2$, i.e., it is quadratic. We ran the matrix multiplication program over 4 and 16 homogeneous processors (SPARC LX's). The local strategies used two groups, i.e., the number of neighbors was 2 and 8, for the 4 and 16 processor case, respectively. The data size used was, $R = 400$ and $R = 1600$ for the 4 and 16 processor case (i.e., $R/\text{Proc} = 100$), with $C = 400$, and $R^2 = 400$ for the first set, and $R = 800$ and $R = 3200$ for the 4 and 16 processor case (i.e., $R/\text{Proc} = 200$), with $C = 800$, and $R^2 = 400$ for the second set of experiments.

Experimental Results

Figures 5 and 6 show the experimental results for MXM for different data sizes on 4 and 16 processors, respectively. In the figure the legend “MXM (no DLB)” stands for a run of the matrix multiplication program in the presence of external discrete random load, but with no attempt to balance the work, i.e., we partition the iterations in equal blocks among all the processors, and let the program run to completion. The other bars correspond to running the MXM program under each of the dynamic load balancing schemes.

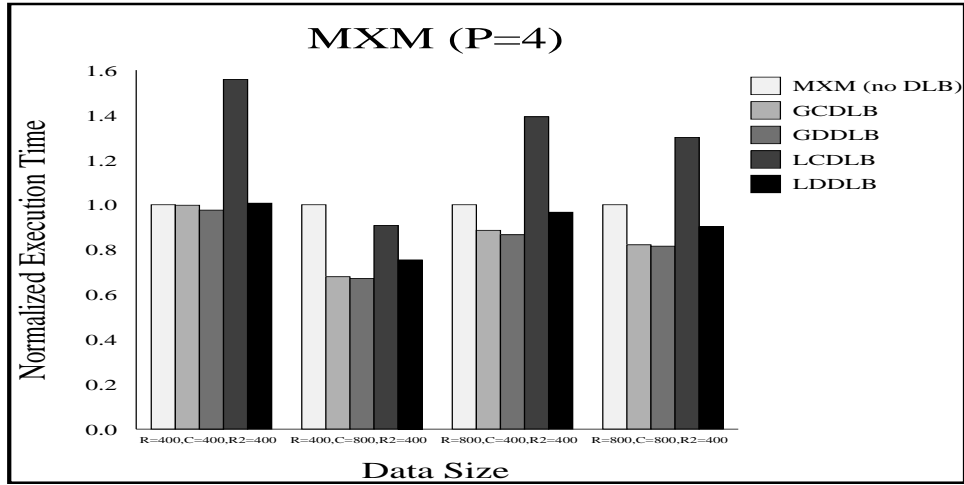


Figure 5: Matrix multiplication (P=4)

We observe that the global distributed (GDDLb) strategy is the best, which is followed closely by the global centralized (GCDLB) scheme. Among the local strategies, local distributed (LDDLb) does better than local centralized (LCDLB). Moreover, the global schemes are better than the local schemes. We also notice that on 16 processors the gap between the globals and locals becomes smaller. From our discussion in section 3.6, local strategies incur less communication overhead than global strategies. However the redistri-

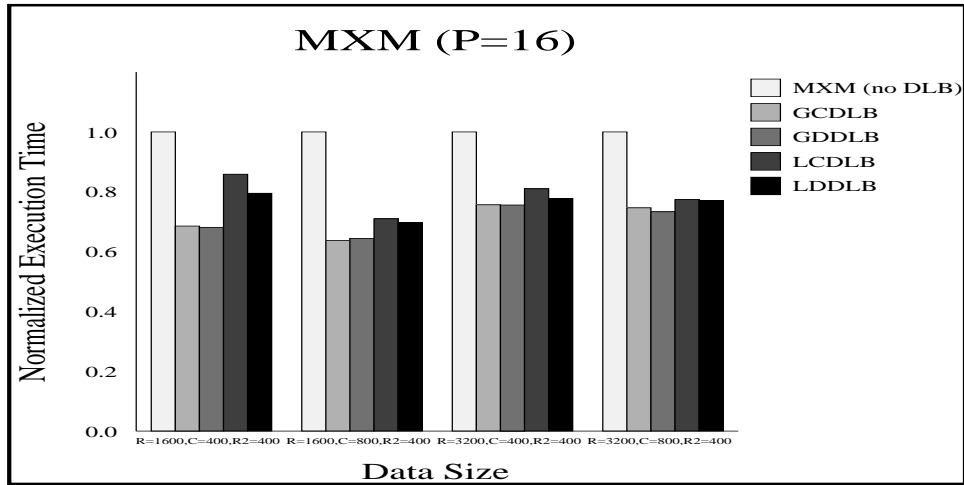


Figure 6: Matrix multiplication (P=16)

bution 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 towards the local strategies as this ratio decreases. The factors that influence this ratio are the work per iteration, number of iterations, and the number of processors. More processors increase the syn-

chronization 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 the delay factor (see section 4.2), and also due to the context switching between the load balancer and the computation slave (since the processor housing the load balancer also takes part in computation).

Modeling Results

Table 1 shows the actual order and the predicted order of performance of the different strategies under varying parameters for MXM. We observe that the actual experimental order and the predicted order of performance matches very closely.

Parameters				Actual				Predicted			
P	R	C	R2	1	2	3	4	1	2	3	4
4	400	400	400	GD	GC	LD	LC	GD	GC	LD	LC
4	400	800	400	GD	GC	LD	LC	GD	GC	LD	LC
4	800	400	400	GD	GC	LD	LC	GD	GC	LD	LC
4	800	800	400	GD	GC	LD	LC	GD	GC	LD	LC
16	1600	400	400	GD	GC	LD	LC	GD	GC	LD	LC
16	1600	800	400	GC	GD	LD	LC	GD	GC	LD	LC
16	3200	400	400	GD	GC	LD	LC	GD	GC	LD	LC
16	3200	800	400	GD	GC	LD	LC	GD	LD	GC	LC

Table 1: MXM: Actual vs. Predicted Order

6.3 TRFD

TRFD has two main computation loops with an intervening transpose. The two loops are load balanced independently, while the transpose is sequentialized, i.e., at the end of the first loop nest, all the processors send their portion of data to the master, who then performs the transpose, which is followed by the second loop nest. We parallelized the outer most loop of both the loop nests. There is only one major array used in both the loops. Its size is given as $[n(n+1)/2] \cdot [n(n+1)/2]$, where n is an input parameter. The loop iterations operate on different columns of the array, which is distributed in a column block fashion among all the processors. The data communication, \mathcal{DC} , is simply the row size. Since the loops are *doall* loops, there is no intrinsic communication. The first loop nest is uniform, with $n(n+1)/2$ iterations and work per iteration given as $n^3 + 3n^2 + n$, which is linear in the array size ($\frac{n^3 + 3n^2 + n}{(n^2 + n)/2} \approx 2n + 4$). The second loop nest has triangular work, given as $n^3 + 3n^2 + n(1 + i/2 - i^2/2) + (i - i^2)$, where $i = (1 + \text{sqrt}(-7 + 8 * j))/2$, and j is the outermost loop index. We transform this triangular loop into a uniform loop using the

bitonic scheduling technique [4], i.e., by combining iterations i and $n(n+1)/2 - i + 1$ into one iteration. Note that the number of iterations for loop 2 is given as $n(n+1)/4$, and the work is also linear in the array size. We experimented with input parameter value of 30, 40, and 50, which correspond to the array size of 465, 820, 1275, respectively, and we used 4 and 16 processors, with the local strategies using 2 groups (2 and 8 processors per group, respectively).

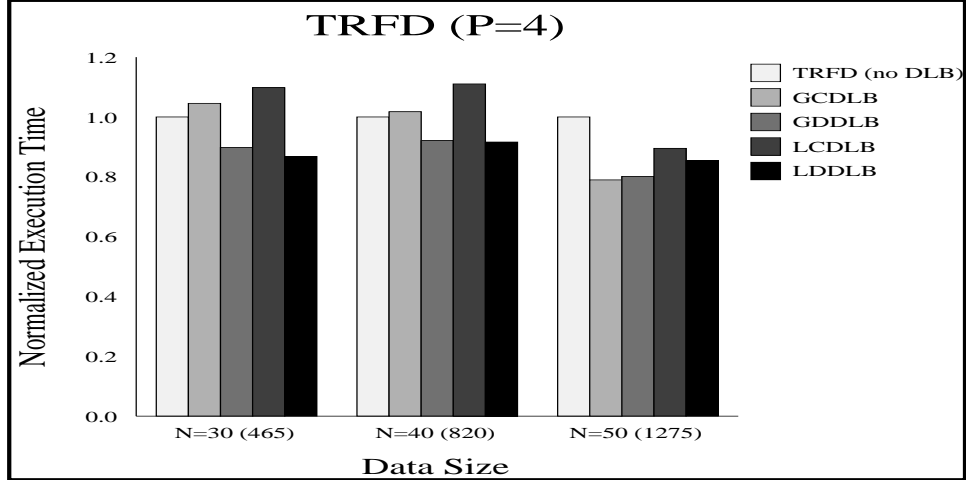


Figure 7: TRFD (P=4)

Experimental Results

Figures 7 and 8 show the results for TRFD with different data sizes for 4 and 16 processors, respectively. In the figure the legend “TRFD (no DLB)” stands for a run of the TRFD program in the presence of external discrete random load, but with no attempt to balance the work.

We observe that on 4 processors, as the data size increases we tend to shift from local distributed (LDDLB) to global distributed (GDDL). Since the amount of work per iteration is small, the computation vs. communication ratio is small, thus favoring the local distributed scheme on small data sizes. With increasing data size, this ratio increases, and GDDL does better. Among the centralized schemes the global (GCDLB) is better than the local (LCDLB). On 16 processors however, we find that the local distributed (LDDLB) strategy is the best, which is followed by the global distributed (GDDL) scheme. Among the centralized strategies also the local (LCDLB) does better than the global (GCDLB), since the computation vs. communication ratio is small. Furthermore, the distributed schemes are better than the centralized ones.

The results shown above are for the total execution time of the TRFD program. It is also instructive to consider the loops individually, as shown in table 2 under the *Actual* column. Loop 2 has almost double the work per iteration than in loop 1. We see that on 4 processors, LDDLB is the best, followed by GDDL. For loop 2, however, since the work per iteration is more, GDDL tends to do better with increasing data size. For both the

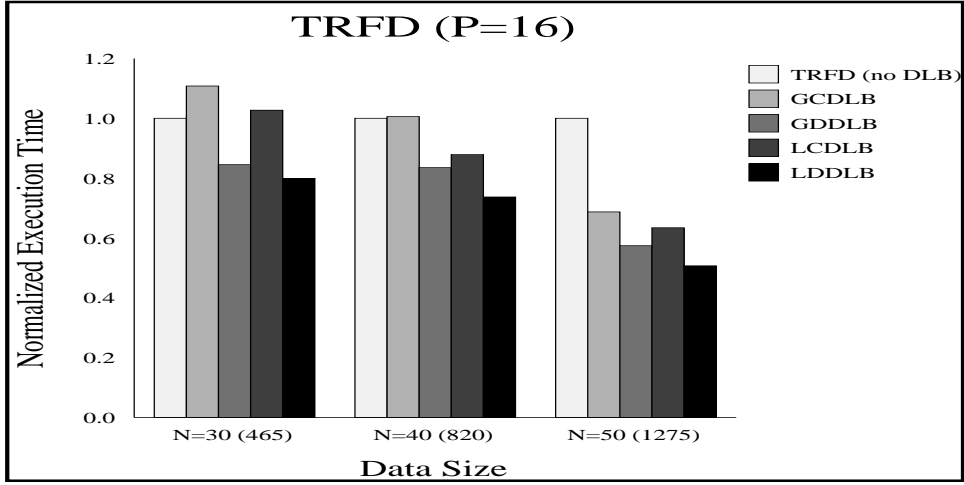


Figure 8: TRFD (P=16)

loops GCDLB is better than LCDLB. On 16 processors, the order is LDDL and LCDLB for loop 1, instead of LDDL and GDDL for loop 2. This is also because of the higher work per iteration for loop 2.

Modeling Results

Table 2 shows the actual order and the predicted order of performance of the different strategies under varying parameters for TRFD. It can be seen that our results are reasonably accurate.

Parameters		Loop	Actual				Predicted			
P	N		1	2	3	4	1	2	3	4
4	30(465)	L1	LD	GD	GC	LC	GD	GC	LD	LC
4	40(820)	L1	LD	GD	GC	LC	LD	GD	GC	LC
4	50(1275)	L1	LD	GD	GC	LC	LD	GD	GC	LC
4	30(465)	L2	LD	GD	GC	LC	GD	LD	GC	LC
4	40(820)	L2	GD	LD	GC	LC	LD	GD	GC	LC
4	50(1275)	L2	GD	LD	GC	LC	GD	LD	GC	LC
16	30(465)	L1	LD	LC	GD	GC	LD	LC	GC	GD
16	40(820)	L1	LD	LC	GD	GC	LD	GD	LC	GC
16	50(1275)	L1	LD	LC	GD	GC	LD	GD	GC	LC
16	30(465)	L2	LD	GD	GC	LC	LD	GC	GD	LC
16	40(820)	L2	LD	GD	LC	GC	LD	GC	GD	LC
16	50(1275)	L2	LD	GD	LC	GC	LD	GD	LC	GC

Table 2: TRFD: Actual vs. Predicted Order

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, communication cost, etc. We then presented a modeling process to evaluate the behavior of these schemes, and described how the compiler together with the run-time system makes use of this model to customize the dynamic load balancing for a program under differing parameters. To take the complexity away from the programmer, we automatically transform an annotated sequential program to a parallel program with the appropriate calls to the dynamic load balancing library.

References

- [1] J.N.C. Arabe, A. Beguelin, B. Lowekamp, E. Seligman, M. Starkey, and P. Stephan. Dome: parallel programming in a heterogeneous multi-user environment. CMU-CS-95-137 30786, Carnegie Mellon Univ - Sch. of Computer Science, April 1995.
- [2] R. D. Blumofe and D. S. Park. Scheduling large-scale parallel computations on network of workstations. *3rd IEEE Intl. Symposium on High-Performance Distributed Computing*, april 1994.
- [3] A. L. Cheung and A. P. Reeves. High performance computing on a cluster of workstations. *Proc. of the 1st Int. Symposium on High Performance Distributed Computing*, pages 152–160, September 1992.
- [4] M. Cierniak, W. Li, and M. J. Zaki. Loop scheduling for heterogeneity. In *4th IEEE Intl. Symposium on High-Performance Distributed Computing, also as URCS-TR 540, CS Dept., Univ. of Rochester*, August 1995.
- [5] D. L. Eager and J. Zahorjan. Adaptive guided self-scheduling. Technical Report 92-01-01, Department of Computer Science, University of Washington, January 1992.
- [6] L. Kipp (ed.). *Perfect Benchmarks Documentation, Suite 1*. Center for Supercomputing Research and Development, University of Illinois at Urbana-Champaign, October 1993.
- [7] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. Sunderam. PVM 3 user's guide and reference manual. Technical Report ORNL/TM-12187, Oak Ridge National Laboratory, Oak Ridge, Tennessee, May 1993.
- [8] A. S. Grimshaw, J. B. Weissman, E. A. West, and E. C. Loyot. Metasystems: An approach combining parallel processing and heterogeneous distributed computing systems. *Journal of Parallel and Distributed Computing*, 21(3):257–270, 1994.
- [9] S.F. Hummel, E. Schonberg, and L.E. Flynn. Factoring: a practical and robust method for scheduling parallel loops. *Communications of the ACM*, 35:90–101, aug 1992.
- [10] C. Kruskal and A. Weiss. Allocating independent subtasks on parallel processors. *IEEE Transactions on Software Engineering*, 11:1001–16, October 1985.
- [11] W. Li and K. Pingali. Access normalization: Loop restructuring for NUMA compilers. *ACM Transactions on Computer Systems*, 11(4):353–375, November 1993.
- [12] F.C.H. Lin and R.M. Keller. The gradient model load balancing method. *IEEE Trans. on Software Engineering*, SE-13:32–38, jan 1987.
- [13] Liu and et al. Scheduling parallel loops with variable length iteration execution times on parallel computers. *Proc. 5th Intl. Conf. Parallel and Distributed Computing and System*, oct 1992.

- [14] S. Lucco. A dynamic scheduling method for irregular parallel programs. *ACM SIGPLAN'92 Conf. Programming Language Design and Implementation*, pages 200–211, 1992.
- [15] E.P. Markatos and T.J. LeBlanc. Using processor affinity in loop scheduling on shared-memory multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 5(4), April 1994.
- [16] H. Nishikawa and P. Steenkiste. A general architecture for load balancing in a distributed-memory environment. *13th IEEE International Conference on Distributed Computing*, may 1993.
- [17] C. D. Polychronopoulos. *Parallel Programming and Compilers*. Kluwer Academic Publishers, 1988.
- [18] C. D. Polychronopoulos and D. J. Kuck. Guided self-scheduling: a practical scheduling scheme for parallel supercomputers. *IEEE Transactions on Computers*, C-36(12):1425–1439, December 1987.
- [19] N. Nedeljkovic M. J. Quinn. Data-parallel programming on a network of heterogeneous workstations. *Proc. of the 1st Int. Symposium on High Performance Distributed Computing*, pages 152–160, sep 1992.
- [20] V. A. Saletore, J. Jacob, and M. Padala. Parallel computations on the charm heterogeneous workstation cluster. *3rd IEEE Intl. Symposium on High-Performance Distributed Computing*, april 1994.
- [21] B.S. Siegell. Automatic generation of parallel programs with dynamic load balancing for a network of workstations. CMU-CS-95-168 30880, Carnegie Mellon Univ. - Sch. of Computer Science, May 1995.
- [22] P. Tang and P.-C. Yew. Processor self-scheduling for multiple nested parallel loops. In *Proc. of '86 International Conference On Parallel Processing*, August 1986.
- [23] T.H. Tzen and L.M. Ni. Trapeziod self-scheduling: A practical scheduling scheme for parallel computers. *IEEE Trans. Parallel Distributed Systems*, 4:87–98, jan 1993.
- [24] R. Wilson, R. French, C. Wilson, S. Amarasinghe, J. Anderson, S. Tjiang, S. Liao, C. Tseng, M. Hall, M. Lam, and J. Hennessy. An overview of the suif compiler system.
- [25] M. Zaki, W. Li, and M. Cierniak. Performance impact of processor and memory heterogeneity in a network of machines. In *the Proceedings of the Fourth Heterogeneous Computing Workshop*, Santa Barbara, California, April 1995.