Contents

Chapter 1

Customized Dynamic Load Balancing

Mohammed J. Zaki[†], Srinivasan Parthasarathy[‡], Wei Li[§]

[†]Computer Science Department
Rensselaer Polytechnic Institute, Troy, NY 12180, USA

[‡]Computer Science Department
University of Rochester, Rochester, NY 14627, USA

[§]Intel Corporation
2200 Mission College Blvd., Santa Clara, CA 95052, USA

Email: *zaki@cs.rpi.edu, srini@cs.rochester.edu, wei.li@intel.com*

1.1 Introduction

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 runtime. 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 runtime scheduling overhead, in a multi-user 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 chapter 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 runtime system that automatically selects the best load balancing scheme for a given loop/task from the repertoire of different strategies. We also automatically transform an annotated sequential program to a parallel program with appropriate calls to our runtime load balancing library. 3) We present experimental results to substantiate our approach. The evaluation indicates that different strategies are best depending on the parameters. Different phases of the same application may also require different strategies. Our modeling is able to capture these variations quite accurately, and thus our analysis can be used to select an appropriate load balancing scheme for an application.

1.1.1 Related Work

We begin by looking at some existing load balancing schemes.

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. For NUMA (Non-Uniform Memory Access) parallel machines, loop scheduling must take data distribution into account [5]. The simplest approach is the *static block* scheduling scheme, which assigns equal block of iterations to each of the available processors. *Static interleaved* scheme assigns iterations in a cyclic fashion.

There has been relatively little work in static scheduling for heterogeneous clusters. Static scheduling algorithms for heterogeneous programs, processors, memory, and network were proposed in [4].

Dynamic Scheduling

When the execution time of loop iterations is not predictable at compile-time, runtime *dynamic* scheduling can be used at the additional runtime cost of managing task allocation. The dynamic scheduling strategies fall under different models, which include schemes based on predicting the future from past loads, the *task queue model*, and the *diffusion model*.

Predicting the Future A common approach taken for load balancing on a workstation network is to predict future performance based on past information. For example, in data parallel C [8], 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 [12] 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 is 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 [11] 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.

Our approach also falls under this model. 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 Section 1.2. [3] presents an application-specific approach to schedule individual parallel applications. [9] presented an approach, 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 runtime support to automatically select the best load balancing scheme for a given loop from a repertoire of different strategies.

Task Queue Model A host of approaches have been proposed in the literature targeting shared memory machines. These fall under the *task queue model*, where there is a logically 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* [13], where each processor is allocated only one iteration at a time, which leads to high synchronization cost. In *guided self-scheduling* [10], the chunk size is changed at runtime. 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. A number of more elaborate schemes based on this idea are extant. For example, *affinity scheduling* [7] also takes processor affinity into account while scheduling, i.e., iterations using the same data are scheduled on the same processor, unless they must be moved to balance load.

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 load. An example is the *gradient model* [6] approach.

1.2 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.

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 to finish its portion of the work. This processor then sends an interrupt to all other active slaves, who 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 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 remains to be done. If there is a sufficient amount of work that needs to be moved, we invoke a *profitability analysis* routine. We 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 is then redistributed among the slaves.

Data Movement and Profitability Analysis Work redistribution also entails the movement of the data arrays which will be accessed in the iterations. There is a trade-off between the benefits of moving work to balance load, and the cost of data movement. Accounting for this cost/benefit is a subtle matter. The reason is that inaccuracies in data movement cost estimation may predict a higher cost for the work redistribution, thereby nullifying the potential benefits of moving work.

In our scheme, since we synchronize only when a processor needs more work, cancelling work redistribution would lead to an idle processor, lowering the overall utilization, and degrading the execution-time. We thus redistribute work as long as the potential benefit (predicted execution time, excluding the cost of actual data movement) of the new assignment results in at least a 10% improvement (empirically, this number worked well).

1.2.1 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 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.

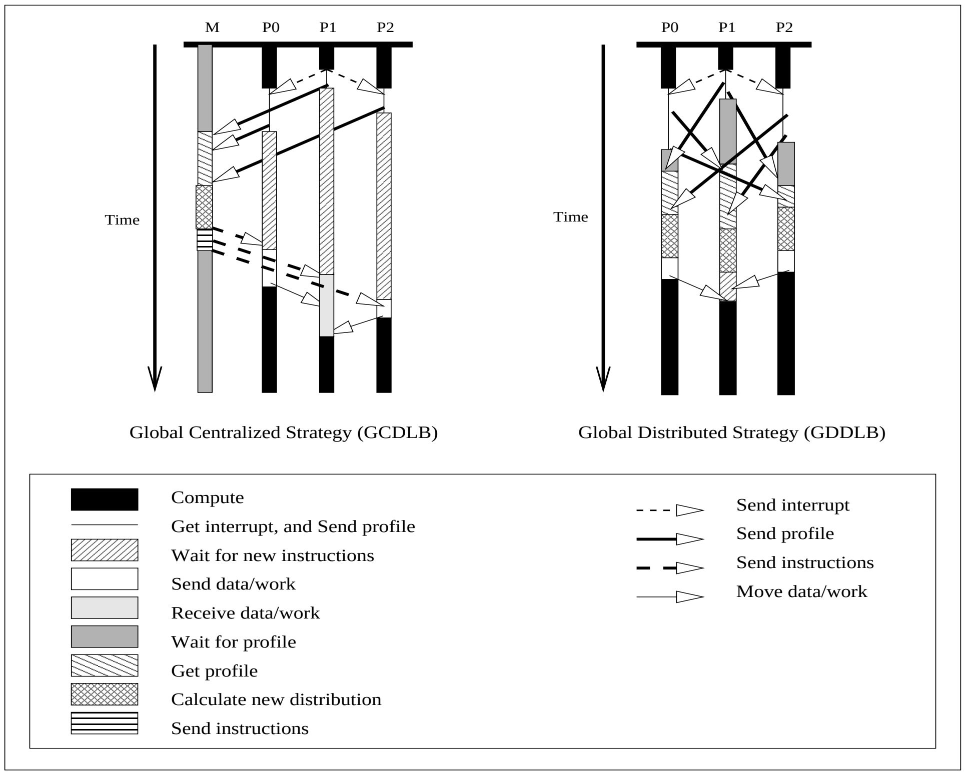


Figure 1.1 Centralized vs. distributed strategies.

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 1.1 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 use the K-block fixed-group approach in our implementation, where the load balancing decisions are made only within a group. If the processors have different speeds, we can perform a static partitioning so that each group has nearly equal aggregate computational power. The global strategies are essentially 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 interrupts only 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 redistribution for that group before proceeding to the other groups.

Local Distributed DLB (LDDLB) Here the load balancer is replicated on all the processors, but profile information is broadcast only to members of the group.

1.2.2 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 redistribution is optimal, based on information known until 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 redistribution 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 chapter 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.

1.3 DLB Modeling and Decision Process

We now present a compile and runtime 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.

1.3.1 Modeling Parameters

The various parameters which affect the modeling are presented below. These include processor parameters such as the number of processors, processor speeds and number of neighbors; program parameters such as the data size, number of loop iterations, work and time per iteration and communication; network parameters such as latency, bandwidth and topology; and finally, modeled external load parameters such as the maximum load and duration of persistence.

Processor Parameters

These give information about the different processors available to the application.

Number of Processors We assume a fixed number of processors available for the computation. This number is specified by the user, and is denoted as P.

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.

Processor Speeds There are a number of ways to calculate the speed of the processor. For example, we could use the MIPS (*million instructions per second*), MFLOPS (*million floating-point operations per second*), Whetstone, or the Dhrystone ratings. In modern processors different operations have different cost, and we would need to consider the speed of a processor in terms of the number of floating-point operations per second, and the number of integer operations per second. We also need to consider memory access time, and the interaction of these with different cache and memory sizes. Multiple instruction issue and instruction pipelining would further complicate the performance model. Therefore, while these figures may give an indication of the processor capabilities, reliable and consistent performance measure can be found only by using the execution time of different real applications on the machines in consideration.

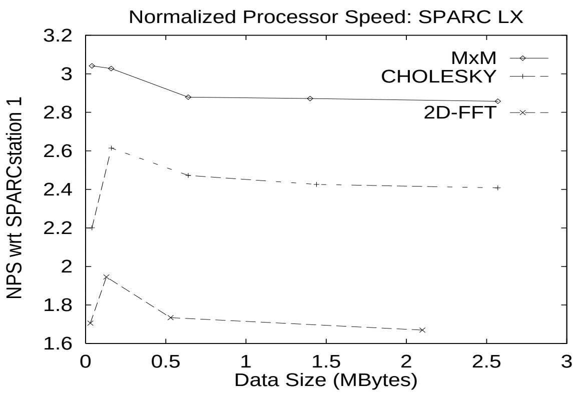


Figure 1.2 NPS: Sun SPARC LX vs. SPARC 1.

Based on our study of heterogeneous loop scheduling [4], we summarize the processor speeds via the notion of *normalized processor speed* (NPS), defined as the ratio of the time taken to execute on the processor in consideration, with respect to the time taken on a base processor. The speed for processor i is denoted as S_i. Consider Figure 1.2, which show the processor performance of a SUN SPARCstation LX on some common scientific kernels – Matrix Multiplication, Cholesky Factorization, and two-dimensional Fast Fourier Transformation. The execution time is normalized against the performance of a SUN SPARCstation 1. Our experiments indicate that machine performance varies for different applications. Since the processor speeds vary from one application to another, we approximate the speed based on small trial application runs. On the other hand, we may obtain these by compile-time performance prediction.

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.

Data Communication This specifies the communication cost due to data movement caused by the load balancing process. 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 $\mathcal{D}_{aij}(N_{ad})$, where a is the array name, i is the loop, and j is the iteration. There is another source of communication, called *intrinsic communication*, which specifies the amount of communication per iteration, which is inherent to the program, for example, communication caused due to data dependencies. In this chapter, we consider only parallel loops, which by definition do not have any intrinsic communication.

Time per Iteration This specifies the time it takes to execute an iteration of a loop on the base processor. It is denoted as $\mathcal{T}_{ij}(\mathcal{W})$, where i is the loop, and j is the iteration. Since this time is with respect to 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 the 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 redistribution. In this chapter, 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 $m\ell$. In our experiments, we set $m\ell = 5$.

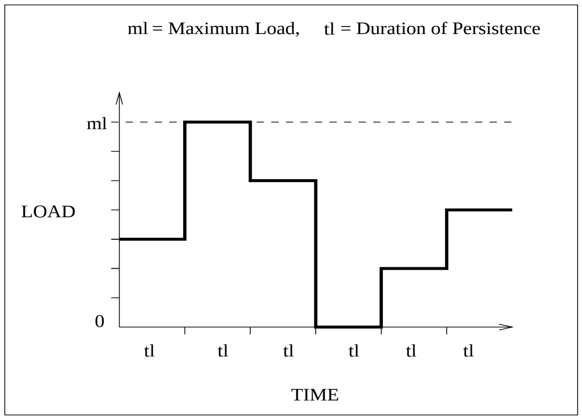


Figure 1.3 Load function.

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 $t\ell$, indicates the amount of time before next load change, i.e., we simulate a discrete random load function, with a maximum amplitude given by $m\ell$, and the discrete block size given by $t\ell$. A small value for $t\ell$ 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 1.3 shows the load function for a processor.

1.3.2 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.

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 : ξ = one-to-all(P) + all-to-one(P)
- GDDLB : ξ = one-to-all(P) + all-to-all(P^2)
- LCDLB (per group) : ξ = one-to-all(K) + all-to-one(K)
- LDDLB (per group) : ξ = one-to-all(K) + all-to-all(K^2)

Cost of Distribution Calculation

This cost, denoted δ, is usually quite small. It 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 redistribute work.

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

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 $\ell_i(k) \in \{0, \cdots, m\ell\}$. 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 $b = \lceil t_j/t\ell \rceil$. Let $\lambda_i(j)$ denotes the *effective load* on processor i between the j-th and the previous synchronization. Then the *average effective speed* of processor i between two these synchronizations is given as

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

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

$$t = t_j - t_{j-1} = \frac{\chi_f(j-1) \cdot \mathcal{T}}{\sigma_f(j)}$$

The iterations left to be done on processor i is simply the old distribution minus the iterations done in time t

$$\gamma_i(j) = \chi_i(j-1) - \left\lceil \frac{t \cdot \sigma_i(j)}{\mathcal{T}} \right\rceil$$

Using the value of t from above, we get

$$\gamma_i(j) = \chi_i(j-1) - \chi_f(j-1)\left(\frac{\sigma_i(j)}{\sigma_f(j)}\right) \tag{1.3.1}$$

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

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

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

$$\sum_{k'=1}^{\aleph} \frac{\mathcal{T}_{k'}}{\sigma_i(j)} \geq t$$

Substituting the value of t from above and moving $\sigma_i(j)$ to the other side, we get

$$\sum_{k'=1}^{\aleph} \mathcal{T}_{k'} \geq \left(\frac{\sigma_i(j)}{\sigma_f(j)}\right) \sum_{k=1}^{\chi_f(j-1)} \mathcal{T}_k$$

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

$$\gamma_i(j) = \chi_i(j-1) - \aleph \tag{1.3.2}$$

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.,

$$\chi_i(j) = \left(\frac{\sigma_i(j)}{\sum_{k=1}^{P} \sigma_k(j)}\right) \cdot \Gamma(j) \tag{1.3.3}$$

Recall that initially we start out with equal work distribution among all the processors, therefore, we have

$$\lambda_i(0) = 1, \;\; \chi_i(0) = \mathcal{I}(N_{ad})/P, \text{ and } \gamma_i(0) = \chi_i(0), \;\; \forall i \in 1, \cdots, P$$

Note that $\lambda_i(0)$ could be proportional to initial processor speed for heterogeneous processors or to the initial processor loads, if known beforehand. These equations, together with equations (1.3.1), (1.3.2), and (1.3.3), 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.,

$$\Gamma(\eta) = 0 \tag{1.3.4}$$

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

$$\kappa(j) = \beta(j) \cdot \mathcal{L} + \alpha(j) \cdot \sum_{a} [\mathcal{D}_a/\mathcal{B}] \tag{1.3.5}$$

where a belongs to the set of arrays that need to be redistributed.

Cost of Sending Instructions

This cost is incurred only 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 sent only 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 1.3.5), and to calculate the number of synchronization points (see equation 1.3.4), 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 redistribution calculation cost, $\kappa(j)$ 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 redistribution 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

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

The total cost of the local strategy is simply the time taken by the last group to finish its computation

$$\mathcal{C} = \mathrm{MAX}_{g=1}^{\lceil P/K \rceil} \{\mathcal{C}_g\}$$

1.3.3 Decision Process – Using the Model

Since all the information used by the modeling process, such as 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 runtime 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 runtime when we have complete information about the system.

Initially at runtime, no strategy is chosen for the application. Work is partitioned equally among all the processors, and the program is run until 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 1.3.1 above, and plugging $j = 1$, i.e., at the first synchronization point we have

$$\chi_f(0) = \mathcal{I}(N_{ad})/P$$

Summing over all processors, we obtain the total iterations done at the first synchronization point as

$$\sum_{i=1}^{P} \left(\frac{\mathcal{I}(N_{ad})}{P} \cdot \frac{\sigma_i(1)}{\sigma_f(1)} \right) > \frac{\mathcal{I}(N_{ad})}{P}$$

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. We plan to study the adaptive load balancing strategy selection approach as part of future work.

1.4 Compiler and Runtime 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 runtime system, using the dynamic load balancing library where appropriate.

1.4.1 Runtime System

The runtime 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 runtime. The compiler can also help to generate symbolic cost functions for the iteration and communication cost.

1.4.2 Code Generation

For the source-to-source code translation from a sequential program to a parallel program using PVM (from Oak Ridge National Labs.) 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 (pseudo-master in the distributed schemes, which is responsible only for the first synchronization, initial scattering, and final gathering of arrays). This involves broadcasting initial configuration information parameters such as the number of processors, the 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, 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 (BLOCK, CYCLIC, or WHOLE). We currently support *do-all* loops only, with data distribution along one dimension (row or column).

SEQUENTIAL CODE

```
for i = 1, n
    for j = 1, m
        for k = 1, r
            Z[i][j] += X[i][k] * Y[k][j]
```

TRANSFORMED CODE

```
DLB_init(DLB, P, K, DLB_arrayZ, DLB_arrayX, DLB_arrayY)
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 = 1, m
                 for k = 1, r
                     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 1.4 Code generation.

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, are highlighted in Figure 1.4. 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, such as the number of dimensions, array size, element type, and distribution type. This structure is also filled by the compiler, and is used by the runtime library to scatter, gather, and redistribute data.

1.5 Experimental Results

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

All the experiments were performed on a network of homogeneous SUN (Sparc LX) workstations, interconnected via an Ethernet LAN (however, our model can easily handle processor heterogeneity). Applications used C as the source code language, and were run on dedicated machines, i.e., there were no other users on

MXM(Matrix Multiplication)

```
for i = 1, n /*parallel*/
    for j = 1, m
        for k = 1, r
            Z[i][j] += X[i][k] * Y[k][j]
```

AC (Adjoint Convolution)

```
for i = 1, n^2 /*parallel*/
    for j = i, n^2
        A[i] += X * B[j] * C[j-1]
```

TRFD

```
for i = 1, n(n+1)/2 /*parallel, L1 */
    for j = 1, n /*uniform*/
        for k = 1, j
            A[j(j+1)/2 + k][i] = B[k]

for i = 1, n(n+1)/2 /*parallel, L1 */
    for j = i, n /*triangular*/
        for k = 1, j
            A[j(j+1)/2 + k][i] = B[k]
```

Figure 1.5 Main computation loop(s): MXM, TRFD, and AC.

the machines. External load was simulated within our programs as described in section 1.3. PVM (from Oak Ridge National Labs.) 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 (pseudocode is shown in Figure 1.5):

- **Matrix Multiply (MXM)**: Multiplication of a $n * m$ with a $m * r$ matrix.
- **TRFD**: It is part of the Perfect Benchmark suite (from University of Illinois, Urbana-Champaign). 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.
- **Adjoint Convolution (AC)**: Convolution of two n^2 length vectors.

The overhead of the DLB schemes is almost negligible, since they are receiver-initiated, and in the absence of external load, all processors will finish work at roughly the same time, requiring only one synchronization.

1.5.1 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 latency obtained with PVM is 2414.5 μs, and bandwidth is 0.96 Mbytes/s. Figure 1.6 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.

1.5.2 MXM: Matrix Multiplication

Matrix multiplication has only one computation loop nest, as shown in Figure 1.5. We have $Z = X \cdot Y$, where X is a $n \times r$ matrix, Y is a $r \times m$ matrix, and Z is a $n \times m$

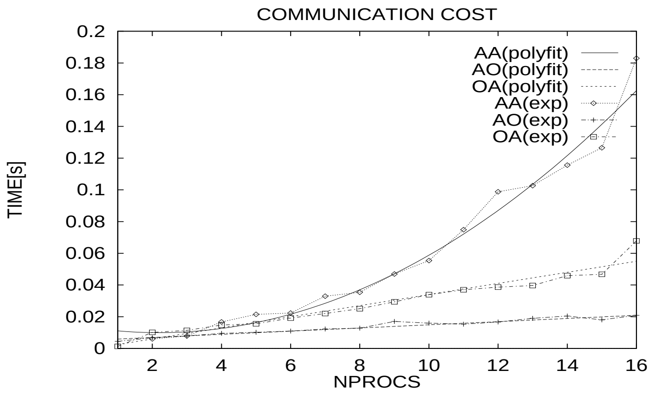


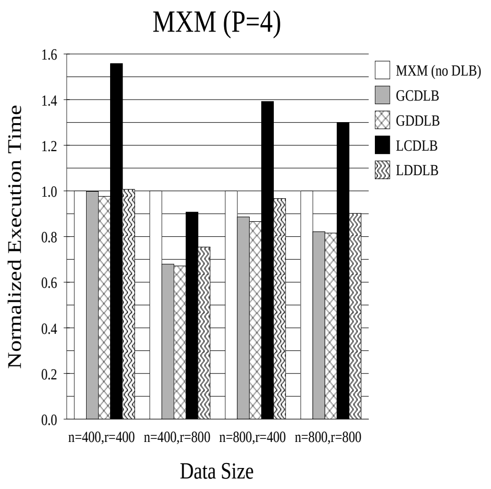
Figure 1.6 Communication cost.

matrix. We parallelize the outermost 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 redistribute work. The data communication is, therefore, given as $\mathcal{C} = N_{X2} = r$. The work per iteration is uniform, and is given as $O(n * m)$, i.e., it is quadratic. We ran the matrix multiplication program over 4 and 16 processors. The local strategies used two groups, i.e., with 2 neighbors on 4 processors, and 8 neighbors on 16 processors. Two sets of experiments were run with $m = 400$ and different values of n and r. In the first set, we used 100 rows per processor, with $n = 400$ on 4 processors, and $n = 1600$ on 16 processors. In the second set, we used 200 rows per processor, with $n = 800$ on 4 processors, and $n = 3200$ on 16 processors.

Experimental Results

Figure 1.7 shows the experimental results for MXM for different data sizes on 4 and 16 processors, respectively. In the figure, the legend "(no DLB)" stands for a run of the 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 program under each of the dynamic load balancing schemes, with time normalized against the case with no dynamic load balancing. Table 1.1 shows the total execution time for the run without load balancing.

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 earlier discussion in Section 1.2.2, local strategies incur less communication overhead than global strategies.



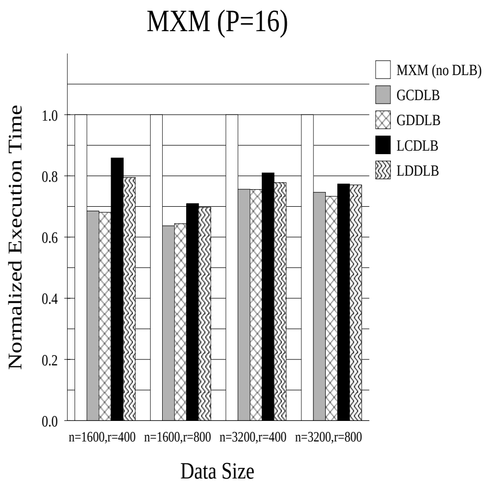


Figure 1.7 Matrix multiplication (P=4, 16).

Table 1.1 MXM: Total Execution Time Without DLB

#Processors	Data Size	Time (s)
4	n=400, r=400, m=400	143.7
4	n=400, r=800, m=400	428.6
4	n=800, r=400, m=400	351.0
4	n=800, r=800, m=400	722.3
16	n=1600, r=400, m=400	266.1
16	n=1600, r=800, m=400	535.9
16	n=3200, r=400, m=400	532.1
16	n=3200, r=800, m=400	1057.3

However, 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 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 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 the delay factor (see Section 1.3.2), 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).

1.5.3 TRFD

TRFD has two main computation loops, shown in Figure 1.5, with an intervening transpose. The two loops are load balanced independently, while the transpose is sequentialized, i.e., after the first loop nest, all the processors send their portion of data to the master, who then performs the transpose. This is followed by the second loop nest. We parallelized the outermost 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, $\mathcal{D}$, is simply the row size. The first loop nest is uniform with $n(n+1)/2$ iterations and work per iteration given as $O(n^3 + 3n^2 + n)$, 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 per iteration, given as $O(n^3 + (3 - r/2)n^2 + (2 - r - r^2/2)n + (r - r^2)/2)$, where $r = (1 + sqrt(-7 + 8 * i))/2$, and i is the outermost loop index. We transform this triangular loop into a uniform loop using the *bitonic scheduling* technique [4].

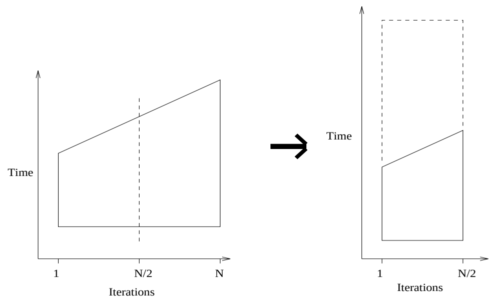


Figure 1.8 Transforming a heterogeneous loop into a homogeneous loop.

Bitonic Scheduling In a triangular loop, the work in iteration i is given as $x_i = ai + b$, for $i = 1, \ldots, n$, where a and b are some constants. Let's assume the n is even (see [4] for the more general case). We can transform this triangular loop with n iterations into a uniform loop with $n/2$ iterations. Note that the sum of the work in iterations i and $(n - i + 1)$ is a constant:

$$x_i + x_{n-i+1} = ai + b + a(n - i + 1) + b = a(n + 1) + 2b$$

We can therefore combine iterations i and $(n - i + 1)$ into one iteration of a new parallel loop. This new loop is homogeneous with $a(n+1)+2b$ operations in every iteration. Figure 1.8 illustrates this transformation.

For TRFD, we combine iterations i and $n(n+1)/2 - i + 1$ into one iteration, to get loops with uniform iterations. The number of iterations for loop 2 is now given as $n(n + 1)/4$. 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).

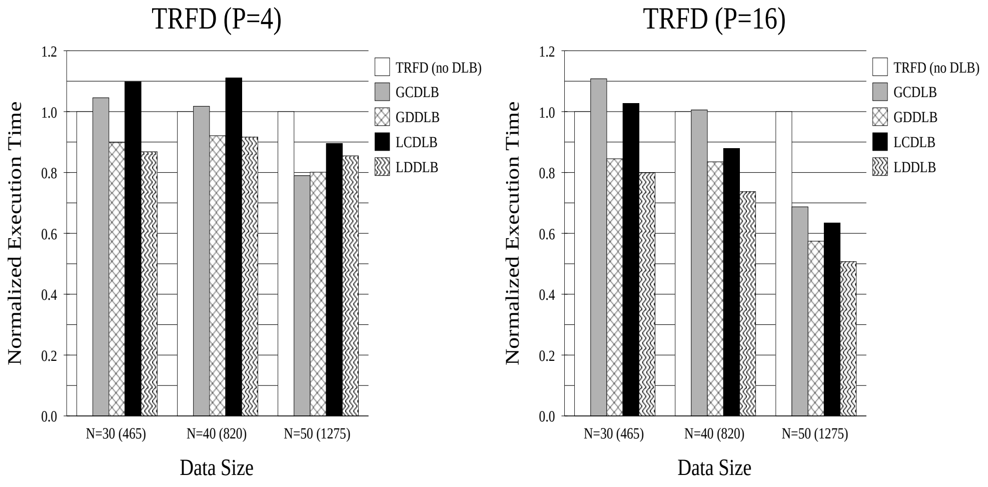


Figure 1.9 TRFD (P=4, 16)

Table 1.2 TRFD: Total Execution Time Without DLB

#Processors	Data Size	Time (s)	#Processors	Data Size	Time (s)
4	n=30(465)	31.4	16	n=30(465)	23.0
4	n=40(820)	111.6	16	n=40(820)	70.0
4	n=50(1275)	417.4	16	n=50(1275)	246.9

Experimental Results

Figure 1.9 shows the results for TRFD with different data sizes for 4 and 16 processors, respectively. Table 1.2 shows the total execution time of a run of TRFD without load balancing.

We observe that on four processors, as the data size increases we tend to shift from local distributed (LDDLB) to global distributed (GDDLB). 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 GDDLB does better. Among the centralized schemes, the global (GCDLB) is better then 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 (GDDLB) 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 total execution time of TRFD. It is also instructive to consider the loops individually, as shown in Table 1.4 under the *Actual* column. Loop 2 (L2) has almost double the work per iteration than in loop 1 (L1). We see that for L1 on 4 processors, LDDLB is the best. For L2, however, since the work per iteration is more, GDDLB tends to do better with increasing data size. On 16 processors, LDDLB remains the best throughout for both L1 and L2.

1.5.4 AC: Adjoint Convolution

Adjoint Convolution has only one computation loop nest, shown in Figure 1.5. We parallelize the outermost loop. All the arrays are replicated, and there is no communication of data when we redistribute work. Therefore, $\mathcal{D} = 0$. The loop nest has triangular work per iteration, given as $O(n^2 - i)$. Using the *bitonic scheduling* technique described above, we transform this into a uniform loop by combining iteration i and $n^2 - i + 1$ into one iteration. The resulting work per iteration is given as $O(n^2 - i + n^2 - (n^2 - i + 1)) = O(n^2 - 1)$. We experimented with input values of $n = 100, 150, 200, 250$, on 4 and 16 processors (2 groups were used for the local strategies).

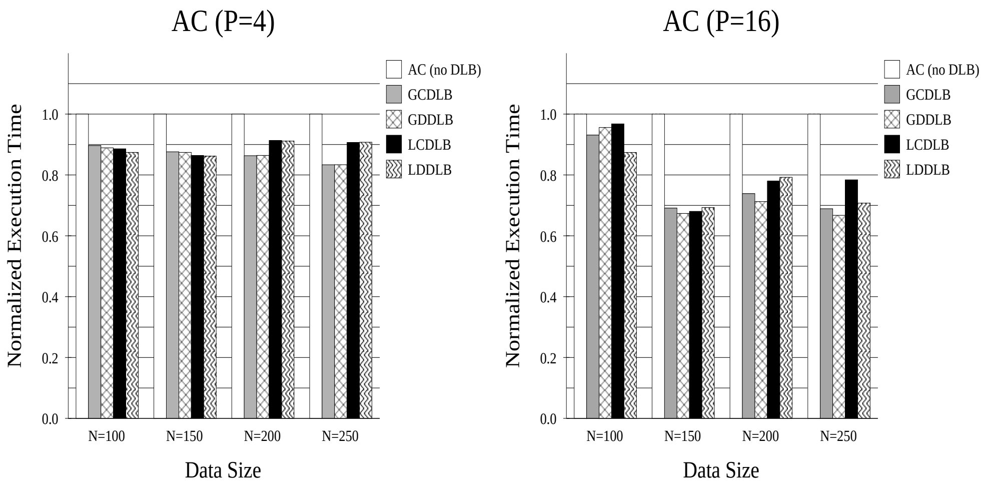


Figure 1.10 Adjoint convolution (P=4, 16).

Experimental Results

Figure 1.10 shows the results for AC with different data sizes for 4 and 16 processors, respectively. Table 1.3 shows the total execution time for a run of the AC program in the presence of external load, but without dynamic load balancing. An mentioned above, this application has no communication due to movement of data arrays

Table 1.3 AC: Total Execution Time Without DLB

#Processors	Data Size	Time (s)	#Processors	Data Size	Time (s)
4	n=100	58.1	16	n=100	16.0
4	n=150	290.3	16	n=150	82.8
4	n=200	879.8	16	n=200	224.2
4	n=250	2163.4	16	n=250	549.7

when we redistribute work. However, communication is still required for profile and instruction exchanges.

On four processors, at small data sizes, the local strategies are better than the globals. This trend reverses as we increase the data size. Similar results were obtained for 16 processors. Moreover, the distributed schemes have a slight edge over the centralized ones.

1.5.5 Modeling Results: MXM, TRFD, and AC

Table 1.4 shows the actual order and the predicted order of performance of the different strategies under varying parameters for the MXM, TRFD and AC programs. We observe that the actual experimental best and the predicted best strategy match in most of the cases. For the cases where our prediction differs from the actual run, the predicted scheme is usually the second best in the actual experiments. For these cases the table shows the difference in the actual execution between the actual and predicted best schemes in terms of the time and as a percentage. For example, consider the row for TRFD, with $P = 4$ and data size $n = 40(820), L2$. The actual best scheme was GDDLB, and the predicted best was LDDLB. Looking at the actual runs, we found that the total execution time of GDDLB and LDDLB was 24.2s and 25.7s, respectively. The difference between these two schemes is thus 1.5s, or about 6.2%. Similarly, we can observe that whenever there is a mismatch between the actual and predicted best schemes, the actual differences in execution time is very small, with an average difference of 2.7% and a maximum of 8.2%. Another factor to keep in mind is that the table presents the actual best scheme averaged over several runs. Since the differences are really small, in practice, one or the other scheme may do better from one run to another, which makes the prediction task extremely difficult. Moreover, the modeling discrepancy usually occurs at the crossover points along the two axes under consideration, i.e., when a best scheme starts to shift from a local to a global strategy (and vice versa), or from a centralized to a distributed strategy (and vice versa). This can be seen, for example, for L2 of TRFD with $P = 4$. As the data size increases, the actual best scheme starts to shift from a LDDLB to a GDDLB. It is well nigh impossible to predict the best scheme with such fine-grained accuracy. As the table shows, even for these difficult points, while our modeling doesn't predict the best scheme, the scheme it predicts is only slightly worse than the actual best.

Table 1.4 MXM & TRFD: Actual vs. Predicted Best DLB Scheme

Program	Parameters		Actual	Predicted	Difference	
	P	Data Size	Best	Best	Time(s)	% Diff
MXM	4	n=400, r=400	GDDLB	GDDLB		
	4	n=400, r=800	GDDLB	GDDLB		
	4	n=800, r=400	GDDLB	GDDLB		
	4	n=800, r=800	GDDLB	GDDLB		
	16	n=1600, r=400	GDDLB	GCDLB	1.2s	0.7%
	16	n=1600, r=800	GCDLB	GDDLB	3.7s	1.1%
	16	n=3200, r=400	GDDLB	GDDLB		
	16	n=3200, r=800	GDDLB	GDDLB		
TRFD	4	n=30(465), L1	LDDLB	GDDLB	0.9s	8.2%
	4	n=40(820), L1	LDDLB	LDDLB		
	4	n=50(1275), L1	LDDLB	LDDLB		
	4	n=30(465), L2	LDDLB	GDDLB	0.2s	3.5%
	4	n=40(820), L2	GDDLB	LDDLB	1.5s	6.2%
	4	n=50(1275), L2	GDDLB	GDDLB		
	16	n=30(465), L1	LDDLB	LDDLB		
	16	n=40(820), L1	LDDLB	LDDLB		
	16	n=50(1275), L1	LDDLB	LDDLB		
	16	n=30(465), L2	LDDLB	LDDLB		
	16	n=40(820), L2	LDDLB	LDDLB		
	16	n=50(1275), L2	LDDLB	LDDLB		
AC	4	n=100	LDDLB	LDDLB		
	4	n=150	LDDLB	GDDLB	3.6s	1.43%
	4	n=200	GCDLB	GDDLB	0.8s	0.1%
	4	n=250	GCDLB	GDDLB	0.5s	0.03%
	16	n=100	LDDLB	LDDLB		
	16	n=150	GDDLB	LDDLB	1.6s	2.9%
	16	n=200	GDDLB	GDDLB		
	16	n=250	GDDLB	GDDLB		

1.6 Summary

In this chapter, 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. We showed that our model is reasonably accurate in its predictions and can guide the decision process effectively.

Presenting a hybrid compile and runtime process, we showed that it is possible to customize the dynamic load balancing scheme for a program under differing parameters. 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 runtime dynamic load balancing library.

1.7 Bibliography

[1] J. Arabe et al. Dome: Parallel Programming in a Heterogeneous Multi-User Environment. *Tech. Report 95-137*, Carnegie Mellon University, April 1995.

[2] R. Blumofe and D. Park. Scheduling Large-Scale Parallel Computations on Network of Workstations. *3rd IEEE International Symposium on High Performance Distributed Computing*, August 1994.

[3] F. Berman et al. Application-Level Scheduling on Distributed Heterogeneous Networks. *Supercomputing*, November 1996.

[4] M. Cierniak, M. J. Zaki, and W. Li. Compile-time Scheduling Algorithms for a Heterogeneous NOW. *The Computer Journal*, vol. 40(6), pages 356–372, December 1997.

[5] W. Li and K. Pingali. Access normalization: Loop Restructuring for NUMA Compilers. *ACM Transactions on Computer Systems*, vol. 11(4), pages 353–375, November 1993.

[6] F. Lin and R. Keller. Gradient Model Load Balancing Method. *IEEE Transactions on Software Engineering*, vol. 13, pages 32–38, January 1987.

[7] E.P. Markatos and T.J. LeBlanc. Using Processor Affinity in Loop Scheduling on Shared-Memory Multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, vol. 5(4), April 1994.

[8] N. Nedeljkovic and M. Quinn. Data-Parallel Programming on a Heterogeneous NOW. *1st IEEE International Symposium on High Performance Distributed Computing*, September 1992.

[9] H. Nishikawa and P. Steenkiste. General Architecture for Load Balancing in Distributed-Memory Environment. *13th IEEE International Conference on Distributed Computing*, May 1993.

[10] C. Polychronopoulos and D. Kuck. Guided Self-Scheduling: Practical Scheduling Scheme for Supercomputers. *IEEE Transactions on Computers*, vol. 36(12), December 1987.

[11] V. A. Saletore, J. Jacob, and M. Padala. Parallel Computing on CHARM Heterogeneous COW. *3rd IEEE International Symposium on High Performance Distributed Computing*, August 1994.

[12] B.S. Siegell. Automatic Generation of Parallel Programs with Dynamic Load Balancing for a NOW. *Ph.D. Thesis*, Carnegie Mellon University, May 1995.

[13] P. Tang and P.-C. Yew. Processor Self-Scheduling for Multiple Nested Parallel Loops. *International Conference on Parallel Processing*, August 1986.

[14] M. J. Zaki, W. Li, and S. Parthasarathy. Customized Dynamic Load Balancing for a NOW. *Journal of Parallel and Distributed Computing*, vol. 43(2), pages 156–162, June 1997.

Index