Automatic Selection of Load Balancing Parameters Using Compile-time and Run-time Information

Bruce S. Siegell
Department of Electrical and Computer Engineering
Carnegie Mellon University
Pittsburgh, PA 15213

Peter A. Steenkiste
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

November 15, 1995

Abstract

Clusters of workstations are emerging as an important architecture. Programming tools that aid in distributing applications on workstation clusters must address problems of mapping the application, heterogeneity and maximizing system utilization in the presence of varying resource availability. Both computation and communication capabilities may vary with time due to other applications competing for resources so dynamic load balancing is a key requirement. For greatest benefit, the tool must support a relatively wide class of applications running on clusters with a range of computation and communication capabilities. We have developed a system that supports dynamic load balancing of distributed applications consisting of parallelized DOALL and DOACROSS loops. The focus of this paper is on how the system automatically determines key load balancing parameters using run-time information and information provided by programming tools such as a parallelizing compiler. The parameters discussed are the grain size of the application, the frequency of load balancing, and the parameters that control work movement. Our results are supported by measurements on an implementation for the Nectar system at Carnegie Mellon University and by simulation.

1 Introduction

Workstation clusters are emerging as an important architecture for scientific computing applications. However, the tools for managing the distributed resources are in a primitive state. Many message passing libraries exist for networks of workstations, e.g. PVM [34] and MPI [9], but it is not straightforward to port tools such as parallelizing compilers from tightly coupled multicomputers because of the substantially higher communication costs (only partially addressed by higher speed networks) and the heterogeneity and variability of the computation and communication resources. On workstation clusters, both computation and communication capabilities may vary with time due to other applications competing for resources so dynamic load balancing is critical to achieving good system utilization and performance. Load balancing tools have been developed on an ad-hoc basis for specific applications and require tuning by the programmer to perform well on specific systems. More general load balancing packages must be developed so that a

This work was sponsored by the Defense Advanced Research Projects Agency (DOD) under contract MDA972-90-C-0035.

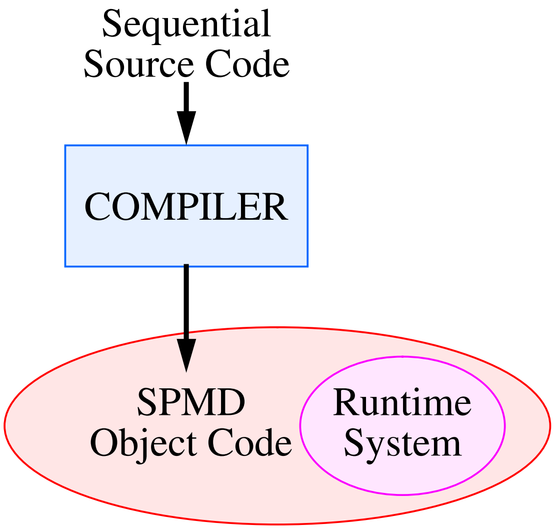


Figure 1: System architecture.

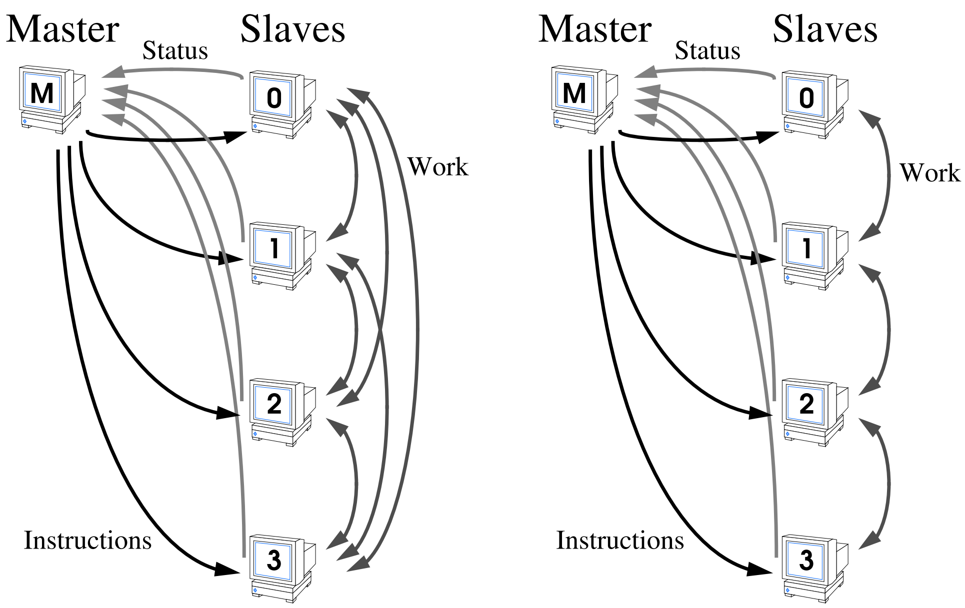
wider range of applications can be run efficiently on a range of systems, so that applications can be moved between systems with minimal effort by the programmer.

We have developed a load balancing system for applications consisting of parallelized DOALL and DOACROSS loops [29] [30]. The system *automatically* selects the load balancing parameters based on application and system information *without involvement from the programmer*, providing application portability. Since the load balancing parameters depend both on application and runtime information, parameter selection requires close cooperation between the compiler and runtime system. The automatically selected parameters are the grain size for the application, the frequency of load balancing, and parameters controlling work movement.

The remainder of this paper is organized as follows. In Section 2 we describe the parameters critical for good performance with dynamic load balancing, and the application and environment features that affect selection of the parameter values. In Section 2, we present an overview of our load balancing system. Section 3 describes the efficiency measure used in evaluating performance. Sections 4, 5 and 6 describe the algorithms for selecting the load balancing parameters. We evaluate and model system performance in Sections 7 and 8. We look at the relationship with related work in Sections 9 and 10 and we conclude in Section 11.

2 System architecture

Programming tools are playing an increasing role in distributed computing. In this paper we look at automatic dynamic load balancing for applications that have been distributed using a programming tool such as a parallelizing compiler or a distributed object library. Figure 1 shows an example: a parallelizing compiler generates code which is linked with a runtime library that includes a dynamic load balancing system. The load balancer has access to information provided by the compiler and the runtime system. In this paper we describe how this information can be used to automatically select the parameters controlling the load balancer.



(a) Unrestricted (b) Restricted by dependences

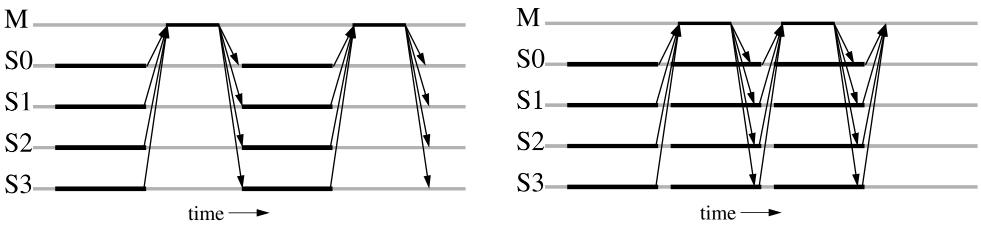
Figure 2: Communication for load balancing.

2.1 Load balancing architecture

For simplicity, our implementation uses a centralized load balancing architecture (Figure 2a) consisting of a *master* and a number of nodes executing the application, called the *slaves*. Because our target is a distributed memory system, the high cost of moving work and corresponding data back and forth from a central location to the slaves would result in poor resource utilization. Therefore, the *work is distributed among the slave processors*, and load balancing is done by shifting the work directly between the slaves. At fixed points in the application code, the slaves send performance information to the master, which generates work movement instructions to balance the load. Because the central load balancer has access to global information, it can instruct overloaded processors to move load directly to processors with surplus processing resources in a single step. To reduce the overhead of interactions between the slaves and the master, the interactions are pipelined (Figure 3).[1]

The performance information sent by the slaves to the master is specified in loop iterations executed per second. With this application-specific performance measure, there is no need to explicitly measure the loads on the processors or to give different weights to different processors in a heterogeneous processing environment. Using the information provided by the slaves, the load balancer calculates the aggregate computation rate of the entire system and computes a new work distribution where the work assigned to each processor is proportional to its contribution to the aggregate rate. This approach relies on the assumption that previous computation rates are indicative of future computation rates [21] [24]. If the load balancer decides that work should be moved, it computes instructions for redistributing the work (using an $O(P \log P)$ algorithm, where P is the number of slaves [29]) and sends the instructions to the slaves. For applications with loop-carried dependences, the instructions only move work between logically adjacent

[1] Pipelining takes the interactions with the load balancer out of the critical path of the application, but delays the effects of the load balancing instructions. Experiments comparing the pipelined and synchronous approaches show that pipelining is effective in reducing costs in a stable system and that pipelining is not detrimental in a dynamic system.



(a) Synchronous (b) Pipelined

Figure 3: Master-Slave interactions for load balancing.

slaves so intermediate processors may be involved in a shifting of load (Figure 2b); this restriction is necessary to maintain a block distribution so that communication costs due to loop-carried dependences are minimized [24]. For applications without such restrictions, work may be moved directly between the source slave and the destination slave (Figure 2a).

Centralized load balancers do not scale well, and many more complex but scalable architectures have been defined. In Section 9 we will discuss how our results apply to other architectures.

2.2 Load balancing parameters

The quality of a load balancer is characterized by the overhead added by load balancing, and the effectiveness of the system in using available resources productively. Three key load balancing parameters have a big impact on these performance measures. First, to control overhead and have effective load balancing, an appropriate *load balancing frequency* must be selected. Second, *work movement decisions* must consider the costs of work movement as well as the goal of balancing the load. Finally, load balancing can have little effect unless the application can be executed efficiently when the load is balanced; for efficient execution, the *grain size* of the application must be selected considering tradeoffs between communication costs and parallelism.

The optimal value of these load balancing parameters depends both on the system and application characteristics. The main application feature that has an impact on load balancing is the *data dependences*. Dependences in the sequential code may correspond to communication, and thus synchronization in the parallelized code. The goal of the load balancer is to balance the computation times between these synchronization points. If a distributed loop is a DOACROSS loop [25], i.e., has loop-carried flow dependences, the mapping of iterations to processors also affects time spent on communication: iterations that share data should be kept together to minimize communication costs. We call the amount of computation performed between synchronization points the *grain size*. In some cases, loops may be restructured to change the grain size, and one of the main tasks of our load balancing system is to select an optimal grain size (Section 4).

The main feature of distributed computing systems that affects the load balancing parameters is the cost of communication. Communication costs influence both the cost of collecting load/performance information and the cost of work movement to balance the load. If the performance information is collected too frequently, the collection overhead may be unacceptable. Moreover, the overhead associated with moving work means that it is impractical to track load changes that happen very quickly. Both of these factors limit how frequently load balancing should be performed (Section 5). Also, if communication costs for an application are too high, distribution of the application may be impractical. Sometimes code can be restructured to reduce the amount or frequency of communication required by the application, e.g., strip mining of loops to increase the grain size.

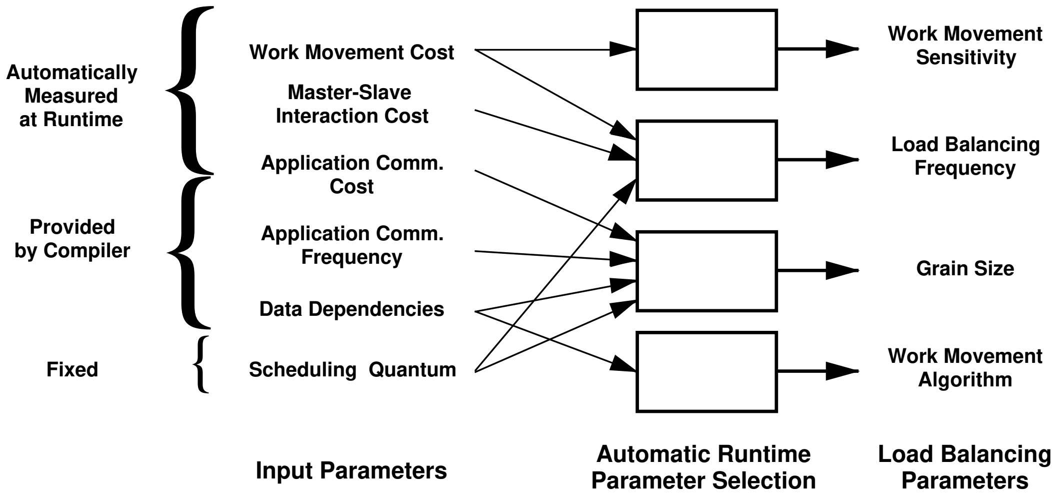


Figure 4: System and application features affecting load balancing parameters.

The scheduling granularity used by the operating system—the *time quantum*—is also a critical system parameter. Process scheduling by the operating system may interact with both the measurements performed by the nodes for performance evaluation and with the synchronizations performed by the application. As a result, the time quantum should be considered when selecting the grain size (Section 4) and the frequency of measurement (Section 5).

To ease the programmer's burden, the code generated by the compiler must automatically adjust to the characteristics of the application and run-time environment without programmer involvement. Specifically, the system has to automatically select the load balancing parameters. Figure 4 summarizes how the application and system features affect the key load balancing parameters: grain size, load balancing frequency, and work movement decisions. In many cases, close cooperation between the compiler and the runtime system is needed to control the load balancing parameters. The remainder of this paper describes the details of how the load balancing parameters are automatically selected and controlled.

3 Evaluation strategy

We evaluate specific system features throughout the paper and the overall system in Section 7. We will present results for systems both with and without competing loads. In the latter case, the performance of the load balancer depends strongly on the specific competing loads that are applied. Practical competing loads can be very diverse and complex, making results hard to interpret. For this reason, we will use an artificial competing load consisting of a process that cycles through compute and sleep cycles. This choice is attractive since it allows us to evaluate the sensitivity of the load balancer to loads of different frequencies by changing the duration of the compute and sleep phases.

Our main performance measure is the *efficiency* of the parallel application in using available processing resources. It is defined as the amount of time spent on useful computation performed by the parallelized application ($t_{productive}$) divided by the total available computation time (e.g., CPU time) while the application is running ($t_{available}$):

$$efficiency = \frac{t_{productive}}{t_{available}} \tag{1}$$

For a homogeneous system, the productive time is estimated with the sequential time for the application measured on one dedicated processor. The available computation time is estimated as the number of slave processors[2], P, times the elapsed time for the application minus the total time spent on competing applications while the application is running. (In some cases, this estimate of the available computation time may be low because the competing processes can take advantage of time not used productively by the load balanced application, but not for the simple loads considered in this paper.) Thus,

$$efficiency = \frac{t_{sequential}}{P \times t_{elapsed} - \sum_{i=1}^{P} compete_i} \tag{2}$$

Our results also apply to heterogeneous processors but the definition of efficiency is more complicated: $t_{sequential}$ and $compete_i$ must be scaled by a relative measure (the scaling factor) of the computational abilities of the processor on which the time was measured or estimated, and the P in the $P \times t_{elapsed}$ term must be replaced by the sum of the scaling factors for the processors [29]. Because we use the rate of computation as the measure of processor performance, dynamic load balancing automatically handles the heterogeneous case. In a heterogeneous system, the slower processors are handled the same way as processors with constant competing loads in a homogeneous system.

4 Automatic grain size selection

We define *grain size* as the amount of computation between synchronizations required by the parallelized application. Generally, increasing grain size reduces communication overhead [33] [35]. However, for applications parallelized by pipelining, increasing grain size also reduces parallelism. If an application has an inappropriate grain size it will not perform well, even if load is perfectly balanced; thus, selecting an appropriate grain size is a prerequisite for dynamic load balancing. Also, the presence of competing loads has an impact on what grain size is appropriate [31].

In discussing grain size, we have to consider the nature of the synchronization in the application. In applications that require no synchronizations, e.g., some versions of matrix multiplication, there is no interaction between the processors so grain size is not an issue. Load balancing may add synchronization points, but their overhead will be limited by the automatic load balancing frequency selection algorithm, which is discussed in the next section. For other applications, we distinguish two types of synchronizations: communication resulting from the pipelining of DOACROSS loops causes *unidirectional synchronizations* between the processors; and communication outside of distributed loops may cause *bidirectional* or *barrier synchronizations* between some or all of the processors. In this section we first discuss loop restructuring transformations used in controlling grain size both at compile time and at run time. We then discuss grain size selection for applications with uni- and bidirectional synchronization.

4.1 Loop restructuring transformations for increasing and controlling grain size

The synchronizations in a parallelized application result from its communication requirements, which are in turn determined by the distribution of the loop iterations and the dependences and loop structure of the original sequential code. In some cases, the amount of computation between synchronizations, i.e., the grain size, can be changed and controlled by modifying the loop structure of the parallelized code. Of greatest interest are loop restructuring transformations such as *strip mining* [18] that allow the grain size to be parameterized with a continuum of grain size choices. Other transformations, such as loop interchange [2, 25, 41, 43] and loop skewing [40, 43], also can be used to modify the grain size, but their

[2] Our efficiency measure does not include the master processor.

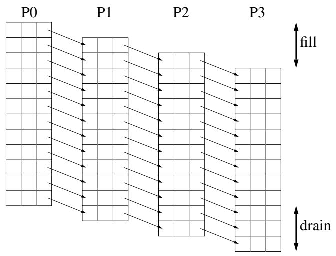
```
DO i = 0, n                DO i0 = 0, n/blocksize
      . . .                      DO i = i0 * blocksize, min((i0+1)*blocksize, n)
END DO                                 . . .
                                 END DO
                           END DO
```

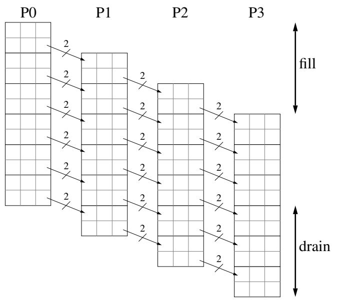
a) Original loop   b) Strip-mined loop

Figure 5: Strip-mining transformation.

```
for (iter = 0; iter < maxiter; iter++) {
    for (i = 1; i < n - 1; i++) {
        for (j = 1; j < n - 1; j++) {
            b[j][i] = 0.493 * (b[j][i-1] + b[j-1][i] + b[j][i+1] + b[j+1][i]) + (-0.972) * b[j][i];
        }
    }
}
```

(a) Sequential






```
for (iter = 0; iter < maxiter; iter++) {
    if (pid != 0) send(left, &b[firstcol][0], n);
    if (pid != pcount-1) receive(right, &b[lastcol][0], n);
    for (i = 1; i < n - 1; i++) {
        if (pid != 0) receive(left, &b[firstcol-1][i], 1);
        for (j = firstcol; j < lastcol; j++) {
            b[j][i] = 0.493 * (b[j][i-1] + b[j-1][i] + b[j][i+1] + b[j+1][i])
                    + (-0.972) * b[j][i];
        }
        if (pid != pcount-1) send(right, &b[lastcol-1][i], 1);
    }
}
```

(b) Pipelined

```
for (iter = 0; iter < maxiter; iter++) {
    if (pid != 0) send(left, &b[firstcol][0], n);
    if (pid != pcount-1) receive(right, &b[lastcol][0], n);
    for (i0 = 0; i0 < n / blocksize; i0++) {
        if (pid != 0) receive(left, &b[firstcol-1][i0*blocksize], blocksize);
        for (i = i0 * blocksize; (i < (i0+1) * blocksize) && (i < n-1); i++) {
            for (j = firstcol; j < lastcol; j++) {
                b[j][i]= 0.493 * (b[j][i-1] + b[j-1][i] + b[j][i+1] + b[j+1][i])
                        + (-0.972) * b[j][i];
            }
        }
        if (pid != pcount-1) send(right, &b[lastcol-1][i0*blocksize], blocksize);
    }
}
```

(c) Blocked and pipelined

Figure 6: Parallelization options for SOR (simplified version; error computation not shown). Figures show pipelined execution of a single relaxation phase. Code portions affected by strip mining and message aggregation are shaded.

application is more difficult to parameterize. Basic techniques for manipulating communication code, such as *loop splitting* [18] and *message aggregation* [4, 12, 13, 33, 35] are also necessary so that the above techniques can be effective. For a given application, the communication patterns in the parallelized code determine which of the above transformations are useful for controlling the grain size. Because of the dynamic nature of computation and communication costs on a network of workstations, the actual selection of the parameters for the transformations may require run-time information.

4.2 Unidirectional synchronizations

For applications with loop carried dependences, i.e., with DOACROSS loops, parallelism is obtained by pipelining multiple instances of the distributed loop. Two factors influence the efficiency of parallelization by pipelining: the time spent on communication of intermediate values due to the loop carried dependences; and the time spent filling and draining the pipeline. The minimum execution time is attained with a grain size that is a compromise between parallelism and communication overhead.

4.2.1 Controlling grain size at run time

For applications with DOACROSS loops, the grain size can be parameterized using techniques such as *strip mining* [18]. Strip mining replaces a single loop with two nested loops. Communication is moved out of the inner loop and *message aggregation* [4, 12, 13, 33, 35] is used to combine messages with common destinations. The number of iterations of the resulting inner loop is the *block size* of the computation. Figure 6 demonstrates the use of these techniques for a successive overrelaxation (SOR) example. To control the grain size at run time, the compiler strip mines the loop, but the block size is a variable, e.g., *blocksize* in Figure 6c, that is set at run time. The grain size, t_{grain}, and the block size are related as follows:

$$blocksize = \frac{t_{grain}}{t_{iteration}} \tag{3}$$

where $t_{iteration}$ is the longest execution time for the local portion of the distributed DOACROSS loop on any of the processors.

Loop tiling [35, 39, 42] is a more complicated approach that combines strip mining and loop interchange, usually to localize memory references by the resulting inner loops. Loop tiling also can be used to control grain size and communication overhead in a manner similar to strip mining [35]. However, compared with strip mining, tiling significantly complicates data management in a system with load balancing, and it will not be addressed in this paper.

4.2.2 Grain size selection

Figure 7 outlines the model of a pipelined computation. The distributed loop has n iterations distributed across P processors, and the strip mined loop has a total of m iterations, divided into M blocks. The size of each block is the block size, b:

$$b = \frac{m}{M}. \tag{4}$$

The pipelining allows blocks on different processors to be executed in parallel, but in a staggered fashion due to the dependences in the application.

For pipelined applications, a tradeoff between parallelism and communication costs must be considered to select an appropriate block size for the strip mined loop. Communication costs are minimized when the block size is made as large as possible. However, due to time required to fill and drain the pipeline, increasing the block size reduces the parallelism in the application; e.g., compare Figures 6b versus 6c for the SOR example in Figure 6. These two conflicting effects can be modeled to estimate the total computation time for the application. Using this model, we select a block size for the strip mined loop that maximizes performance.

4.2.3 Pipeline fill and drain times

If communication of intermediate values is made too infrequent, a large fraction of the execution time will be spent filling and draining the pipeline, resulting in reduced parallelism. Figure 7 shows that for a m iteration loop divided into M blocks ($M = \frac{m}{b}$, where b is the block size), the elapsed time for the application, ignoring communication costs, is $M + P - 1$ times the time to execute one block, t_{block}:

$$t_{elapsed} = (M + P - 1) * t_{block}$$

Thus, the time to fill and drain the pipeline, $(P - 1) \times t_{block}$, increases with the block size and with the number of processors. The total number of blocks to be executed is $P \times M$ so

$$t_{sequential} = P \times M * t_{block} = P \times m * t_{iteration} \tag{5}$$

We can now compute an upper bound on efficiency for an environment with no competing loads:

$$\begin{aligned} \mathit{efficiency} &= \frac{P \times M \times t_{block}}{P \times (M + P - 1) \times t_{block}} \\ &= \frac{M}{M + P - 1} \end{aligned} \tag{6}$$

The efficiency approaches 1.0 as M approaches infinity. However, M can be no more than the number of iterations, m, in the pipelined loop.

4.2.4 Selecting the optimal blocksize

The total execution time for the pipelined loop is the sum of the times for the pipelined phases plus the sum of the communication costs. The loop is executed in $M + P - 1$ computation phases, and communication of boundary values occurs between the computation phases (Figure 7). Not all processors shift boundary values when the pipeline is filling or draining, but to simplify the analysis, we assume that all communication phases take the same amount of time, t_{shift}. Thus, the total execution time is modeled as follows:

$$t_{total} = (M + P - 1) \times t_{block} + (M + P - 2) \times t_{shift} \tag{7}$$

To use this model, we need values for M, t_{block}, and t_{shift}.

M and t_{block} are related by Equation 5 so, if we can determine $t_{sequential}$, we can eliminate one of the unknowns. If the compiler has an accurate model of execution times for the processors, it can predict the sequential execution time. Otherwise, at run time, if all iterations of the pipelined loop require the same amount of computation, a prediction of $t_{sequential}$ can be made by extrapolating from measurements of $t_{iteration}$ (Section 4.2.1). Our implementation uses the latter approach based on measurements of $t_{iteration}$ at startup time with consistent results. On a processor with competing loads, the smallest of several measurements of $t_{iteration}$ is used to approximate $t_{sequential}$ on a dedicated processor. (This works reasonably well if $t_{iteration}$ is smaller than the scheduling quantum for the system.)

t_{shift} can be estimated by measuring communication costs when the application is started. At each communication point, communication is modeled as follows:

$$t_{shift} = t_{fixed} + t_{incr} * elements \tag{8}$$

where t_{fixed} is the fixed overhead of sending messages between processors (i.e., the setup time), and t_{incr} is the cost per data element sent. $elements$ is the number of data elements that must be sent at each communication point and is equal to the block size, b (Equation 4). t_{fixed} is estimated by measuring the time to shift empty messages through the processors. t_{incr} is estimated by measuring the cost of sending fixed length messages through the processors and subtracting t_{fixed}.

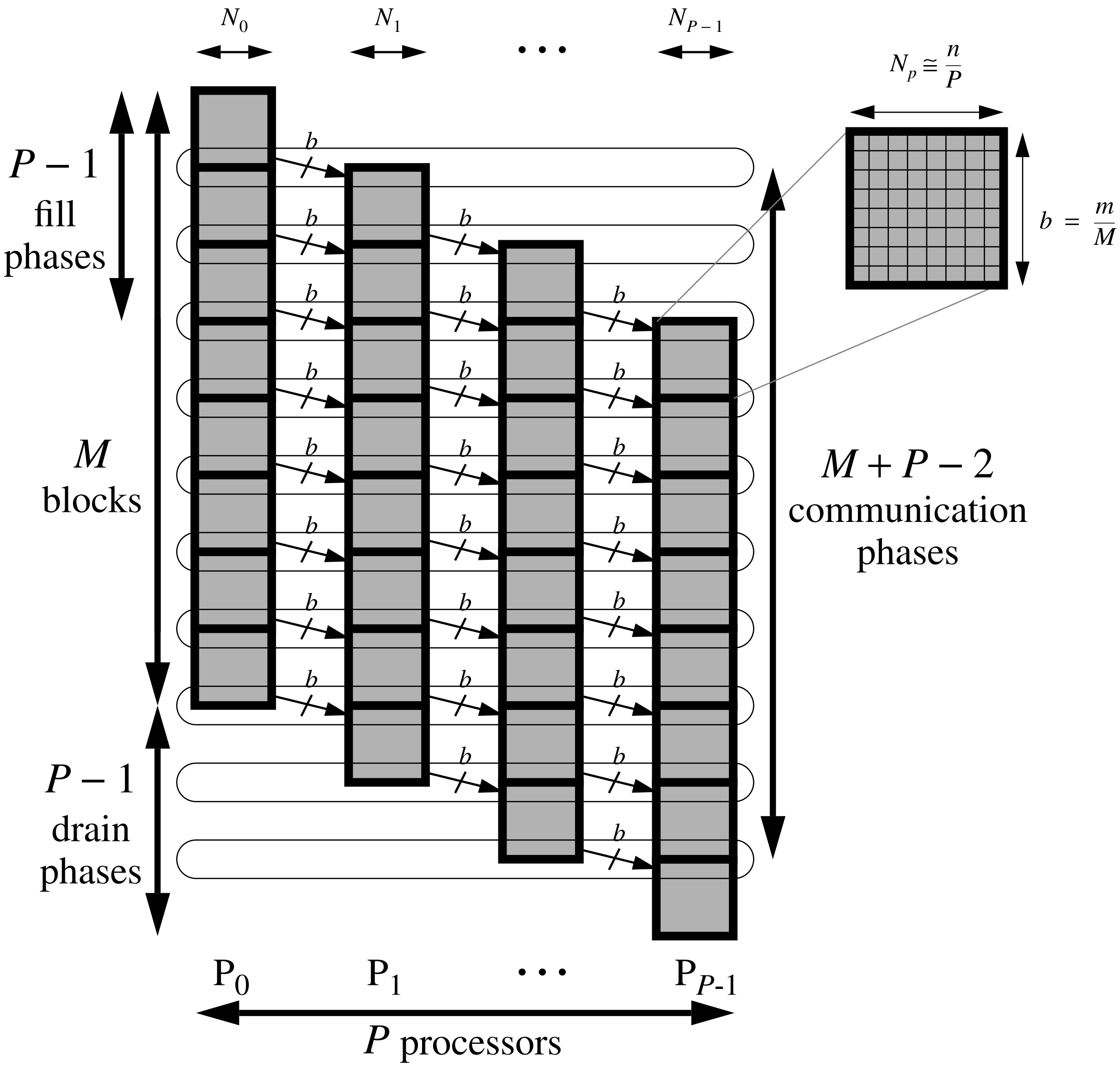


Figure 7: Pipelined execution of a distributed loop showing parameters for modeling execution costs. Distributed loop has n iterations and pipelined loop (enclosing distributed loop) has m iterations.

Thus, substituting into Equation 7,

$$t_{total} = (M+P-1) \times \frac{\frac{t_{sequential}}{P}}{M} + (M+P-2) \times (t_{fixed} + t_{incr} \times \frac{m}{M}) \quad (9)$$

We wish to select M to minimize the total execution time. The value of M that minimizes t_{total} is computed by setting the derivative of t_{total} with respect to M equal to zero. Then, solving for M, the shortest execution time and the highest efficiency are attained when

$$M = \sqrt{\frac{t_{sequential} \times (1 - \frac{1}{P}) + t_{incr} \times m \times (P - 2)}{t_{fixed}}} \tag{10}$$

The optimal M can be computed at application startup time because P is known, and $t_{sequential}$, t_{fixed}, and t_{incr} can be estimated by executing copies of small portions of the computation. The optimal block size is then computed using Equation 4.

Figure 8 shows the efficiency of parallelization for the SOR example (1000x1000 matrix, 10 iterations) predicted by our model and measured on the Nectar system with 4 slave processors. We ran the SOR example with a range of block sizes and with the block size automatically selected using the computations described above; artificial delays were added in Figures 8b and 8c to show how our execution model responds to different communication costs. Estimates of $t_{sequential}$ based on measurements of several iterations of the pipelined loop are consistent over several measurements and are quite close to measurements of the actual time for the application running on a single processor. We estimate the communication cost using the time between the start of the first communication and the end of the last communication in a communication phase. This estimate is conservative (i.e. might be high), but given the shape of the graphs, the estimate errs in the right direction.

Our estimates of $t_{sequential}$ at startup time (extrapolated from measurements of $t_{iteration}$) are based on dedicated use of the system. If there are competing loads on one or more processors, the actual grain size will be larger than expected. In general, we expect these grain size variations to be small enough that the efficiency remains near the maximum value. In the curves shown in Figure 8, it can be observed that changing the block size by as much as a factor of two in either direction does not reduce the efficiency very much. In addition, to take into account changing loads, if the strip mined loop is executed multiple times, the estimate of $t_{sequential}$ can be updated between executions of the loop.

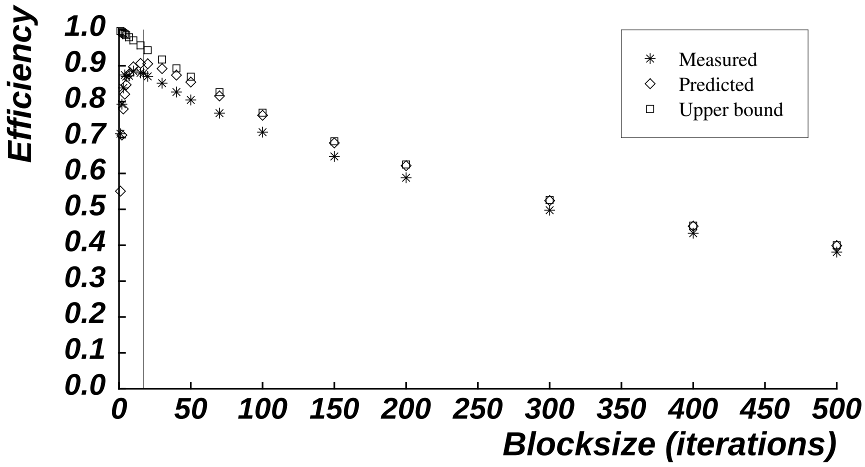
4.2.5 Optimal grain size vs. fixed grain size.

To show the effectiveness of considering both communication overhead and parallelism in selecting grain size, we compared the performance of a version of SOR with a fixed grain size, controlled as described in Section 4.2.1, with the performance of a version with the automatically selected "optimal" grain size, selected using the method described in Section 4.2.4. Figure 9 shows efficiency measurements taken on the Nectar system with homogeneous, dedicated processors for two different problem sizes. We selected a fixed grain size of 1.5 time quanta[3] (150 milliseconds) so that the communication overhead would be small. The efficiency with the fixed grain size was approximately the same as that with automatically selected grain size when the number of processors was small, but as the number of processors was increased, the total execution time for the problems decreased, increasing the effect of filling and draining the pipeline, so the automatically selected grain size, which takes both communication costs and parallelism into account, resulted in higher efficiency.

4.2.6 Effect of competing loads

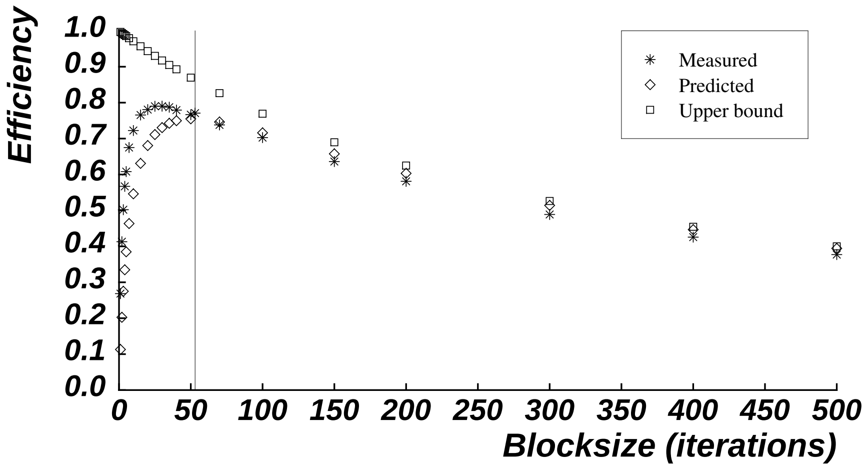
When a competing load is added to one of the processors, intermediate results are delayed for all processors following that processor in the pipeline, and all other processors are idle part of the time. However, if the

[3]The *time quantum* or *time slice* is the unit of scheduling used by the operating system. For Unix systems, the time quantum is typically 100 milliseconds [15].



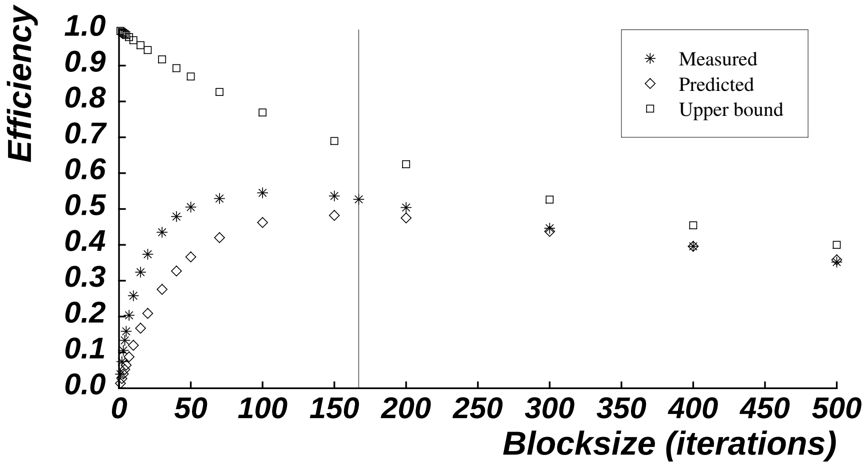
Values used in computing M:
$t_{sequential} = 8478720\mu$sec
$t_{fixed} = 1716\mu$sec
$t_{incr} = 4.8\mu$sec

a) No added delays.



Values used in computing M:
$t_{sequential} = 8496640\mu$sec
$t_{fixed} = 16633\mu$sec
$t_{incr} = 4.8\mu$sec

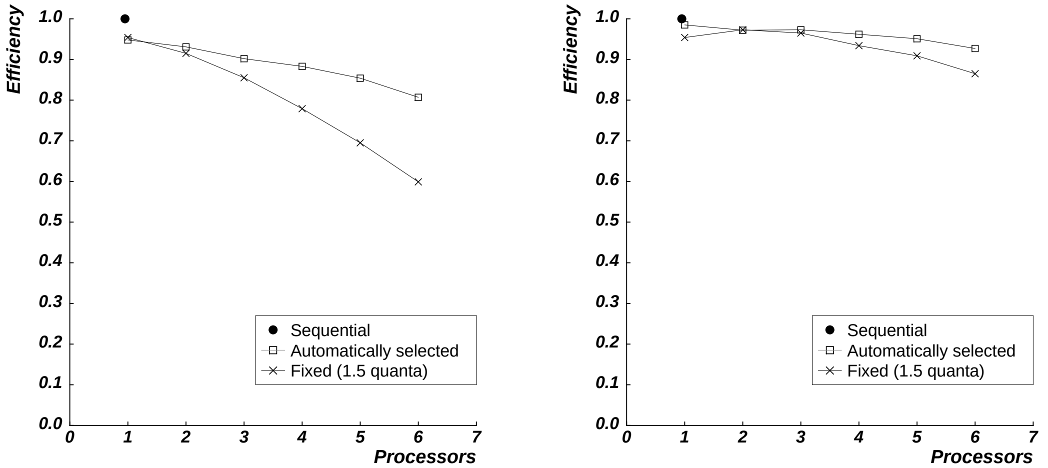
b) 5 msec delay added to each message.



Values used in computing M:
$t_{sequential} = 8500480\mu$sec
$t_{fixed} = 152360\mu$sec
$t_{incr} = 4.8\mu$sec

c) 50 msec delay added to each message.

Figure 8: Efficiency of the pipelined loop in the SOR example (1000x1000 matrix, 10 iterations) as a function of the block size. Vertical lines indicate automatically selected block size and correspond with the peaks of the "Predicted" curves.



a) 1000×1000 (40 iterations) b) 2000×2000 (10 iterations)

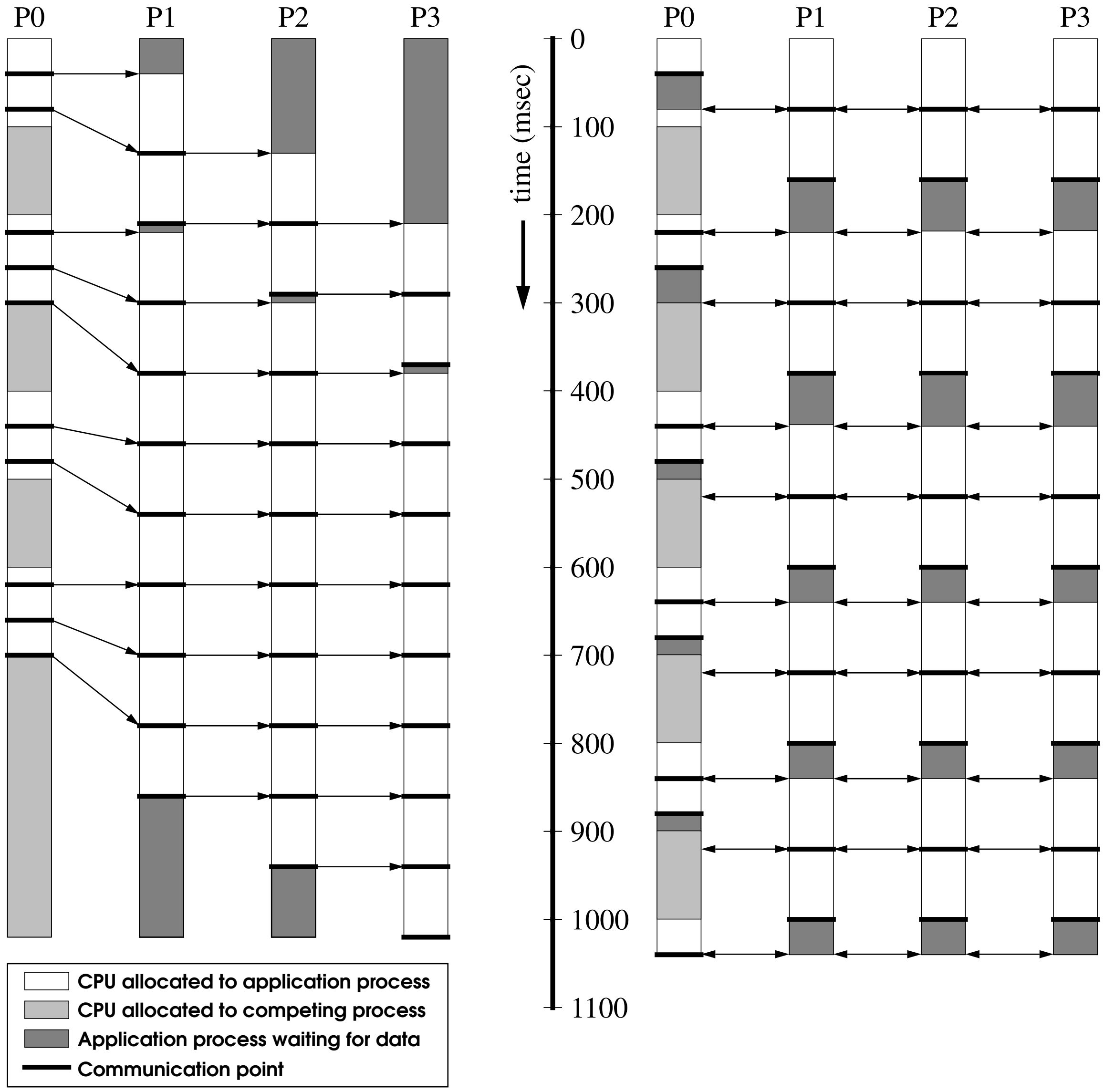
Figure 9: Parallel versions of SOR without load balancing in a dedicated homogeneous environment. Fixed grain size (1.5 quanta = 150 milliseconds) vs. automatically selected grain size.

load is balanced, the processor with the competing load is allocated less work so that during its allocation of the CPU, it generates enough data to keep the processors that follow it busy when the competing load has control of the CPU (Figure 10a). The communication required by the application aligns the processors so that efficiency is not affected adversely by competing loads and pipelined execution can continue without stalling. This is true for any grain size, as long as there is enough buffer space to store the intermediate data.

To verify that grain size has little effect on the efficiency of a pipelined application in a load balanced environment with competing loads we simulated the interactions between the scheduling of processes by the operating system and the communication between the slave processors, as in Figure 10a. Our model of the system assumes that the operating system allocates equal portions of the CPU time to all running processes in a round-robin fashion with a fixed time quantum. The simulations do not consider communication costs, but do model time spent filling and draining the pipeline; therefore the predicted upper bound on efficiency is $\frac{M}{M+P-1}$ (from Equation 6). Figure 11 shows the parallelization efficiencies resulting from simulating different grain sizes under different conditions. In all of the environments simulated, the efficiencies stay very close to the upper bound, regardless of the grain size. In real systems, CPU scheduling is more complicated than round-robin, and competing loads may vary over the course of the application, so the slaves may have to realign themselves occasionally; however, the natural tendency for communication to align the processors should prevent efficiency from being affected too adversely.

4.3 Bidirectional (barrier) synchronizations

If at a communication point, data must be exchanged rather than just shifted in a single direction, a barrier is created; none of the processors involved in the communication may continue executing until all processors have reached the barrier. We call these synchronization points *bidirectional* or *barrier synchronizations*. For DOALL loops, which require no communication inside the loop body, the grain size is determined by the barrier synchronizations outside the distributed loop. Barrier synchronizations may be caused by reduction operations, by distributed loops that just shift data between processors, or by assignment statements that involve data on multiple processors. Adjacent barrier synchronizations can be treated as a single barrier. If the synchronizations cannot be shifted outward by loop splitting, modifying the grain size is complicated as it involves loop transformations such as loop interchange or loop skewing; these transformations have been



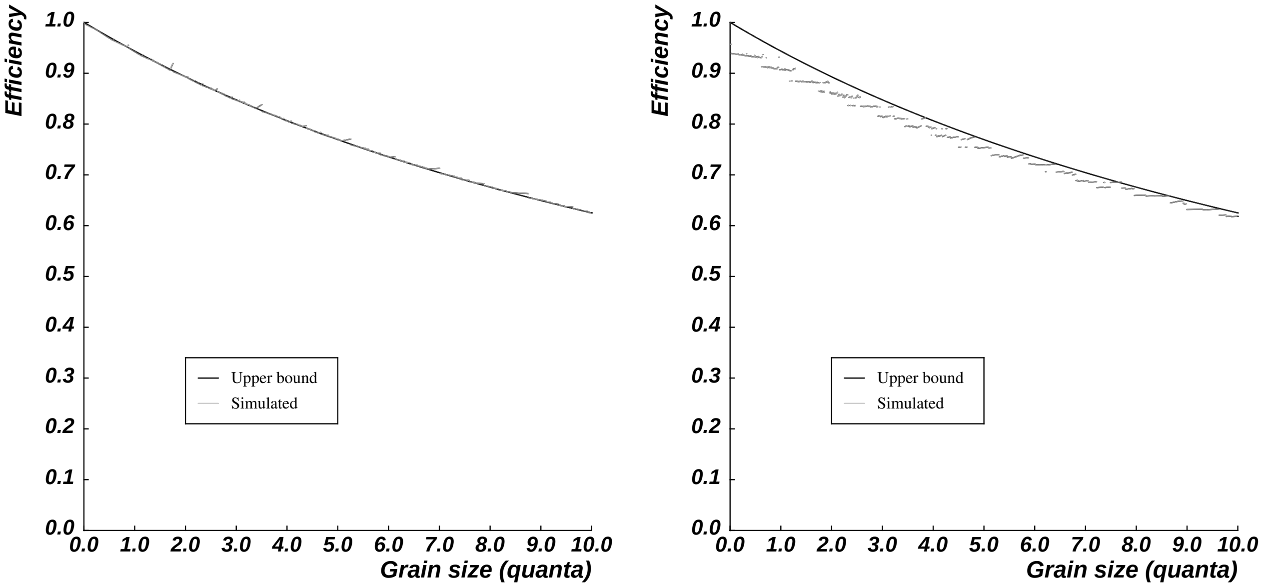
a) Pipelined execution b) Bidirectional synchronization

Figure 10: Applications with pipelined (a) and bidirectional (b) synchronizations executing with competing load on first processor and correctly balanced load. Round robin scheduling ignoring communication costs.

studied elsewhere [41]. For this section, we assume that the application has already been restructured to make the grain size as large as possible, and that the most frequently executed (i.e., innermost) synchronizations are barrier synchronizations. In this section, we analyze the effects of barrier synchronizations on program performance first in a dedicated environment and then in the presence of competing loads.

4.3.1 Synchronization overhead

Figure 12 shows our model for an application in which barrier synchronizations are the most frequent synchronization. For a homogeneous system, we model the total execution time for the application as



a) Competing load on P0

b) Competing loads: 1 on P0, 2 on P1, 3 on P2, 4 on P3

Figure 11: Parallelization efficiency determined from simulation of pipelined execution on a 4 processor system. The sequential execution time of the simulated problem is 200 time quanta.

follows:

$$t_{total} = t_{barrier} \times m + \frac{n}{P} \times t_{iteration} \times m$$

In the sequential version of the program, the execution time is approximately

$$t_{sequential} = n \times t_{iteration} \times m$$

Thus, the efficiency of parallelization in a dedicated homogeneous system is

$$\begin{aligned} \mathit{efficiency} &= \frac{t_{sequential}}{P \times t_{total}} \\ &= \frac{n \times t_{iteration} \times m}{P \times (t_{barrier} \times m + \frac{n}{P} \times t_{iteration} \times m)} \\ &= \frac{n \times t_{iteration}}{P \times t_{barrier} + n \times t_{iteration}} \end{aligned}$$

Since the m terms in the efficiency formulation cancel each other out, we conclude that on a homogeneous (and dedicated) system, the number of barriers is irrelevant, and the overhead can only be reduced by reducing the cost of each barrier.

4.3.2 Effect of competing loads

As we did in Section 4.2.6 for applications with unidirectional synchronization, we now look at the effect of competing loads on efficiency. Figure 10b shows that on a system with competing loads, even when load is balanced, an application with bidirectional synchronization may not be able to use its share of the CPU productively because processes may waste portions of their allocated CPU time waiting at synchronization

```
DO i = 0, m
      DO j = locallo, localhi
            compute iteration
      END DO
      global synchronization
END DO
```

Figure 12: Parallelized version of DOALL loop followed by global barrier synchronization.

points. Since at each synchronization point the elapsed time will be the worst case of the times on all the processors, the skews in execution times will accumulate. However, larger grain sizes (after balancing) guarantee a higher percentage of useful CPU time because the skews are a smaller portion of the total time. Thus, we conclude that for applications with bidirectional synchronization the largest possible grain size should be selected.

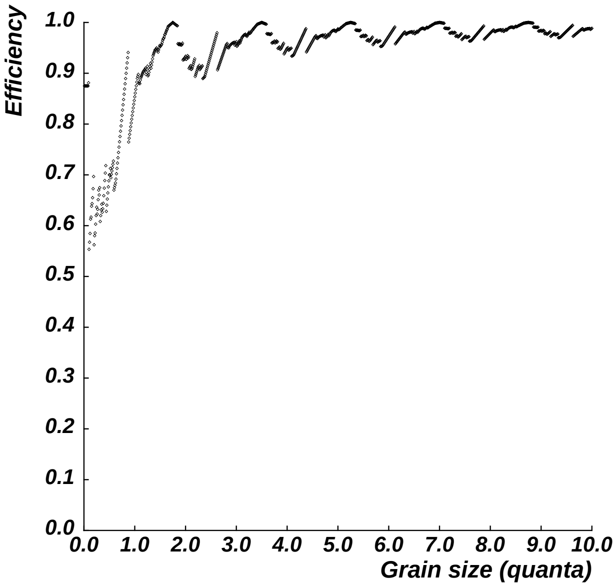
To confirm our hypotheses we simulated the interactions between a load balanced application and the OS scheduling of processes (e.g. Figure 10b) in the same way as we did for pipelined applications in Section 4.2.6. Figure 13 shows parallelization efficiencies attained with different grain sizes. The simulation results confirm the need for a large grain size when there are competing loads on the system. The overall trend is that efficiency increases with grain size, but the increase is not monotonic due to the mismatch between the scheduling periods and the periods between the synchronizations required by the application; 100% efficiency can be obtained when the period between synchronizations is an integral multiple of the scheduling period (Figure 13a). Since actual systems use scheduling algorithms more complicated than round-robin scheduling, and normal system activity may cause variations in the schedule, it is desirable to be out of the range of grain sizes where efficiency fluctuates greatly. Thus, the grain size should be made as large as possible when there are barrier synchronizations in the parallelized application.

5 Automatic selection of load balancing frequency

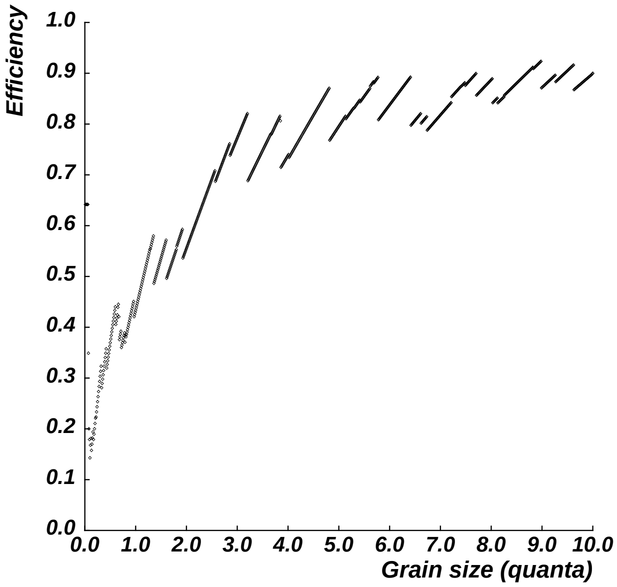
The frequency at which the slaves evaluate their performance and interact with the load balancer affects the overhead of load balancing and the responsiveness of the system to fluctuations in performance on the slave processors. The appropriate load balancing frequency depends both on the system and application characteristics, and controlling the frequency involves both the compiler and the run-time system. The compiler inserts code, called *load balancing hooks*, at points in the application code where the application can interact with the load balancer and work can be moved without disrupting the computation or corrupting data. The load balancing hooks are conditional calls to the load balancing code so the runtime system can control how frequently the the load balancer is actually invoked based on run-time conditions. In this section we discuss both the compile-time and run-time components of automatic load balancing frequency selection.

5.1 Compiler placement of load balancing code

The compiler places conditional calls to the load balancer into the application code in the form of *load balancing hooks*. The hooks test conditions for calling the load balancing code so that the frequency of load balancing can be controlled at run time. When conditions for load balancing are met, the hook calls the load balancing code, which measures the time spent in the computation, sends performance information



a) Competing load on P0



b) Competing loads: 1 on P0, 2 on P1, 3 on P2, 4 on P3

Figure 13: Parallelization efficiency for varying grain sizes on a 4 processor system. Simulation results for round robin scheduling ignoring communication costs.

```
count = count + 1
IF (count > nexthook) THEN
     send_status()
     nexthook = receive_instructions()
     count = 0
END IF
```

Figure 14: Code for load balancing hook.

to the load balancer, receives instructions, and, if necessary, shifts work between the processors. A simple implementation of a hook is shown in Figure 14. To control the load balancing frequency, the count that triggers load balancing, $nexthook$, can be changed each time instructions are received from the load balancer.

For the system to be able to respond to performance fluctuations quickly, hooks must be placed as deep in the loop nest as possible so that they are executed frequently. However, if a hook is placed too deep in the loop nest, the time spent executing the hook may be of the same order of magnitude or greater than the time to execute the computation between two executions of the hook, even if the hook never calls the load balancing code. Thus, placement of load balancing hooks must consider both responsiveness and overhead. A load balancing hook need only be placed at one level of the loop nest because the deepest hook determines the frequency of hook execution. Therefore, we divide the hook placement decision into two steps: identification of possible hook locations, and selection of a single location from among the possible locations. If none of the possible hook locations meets the selection criteria, code may be restructured to create new hook locations.

```
DO iter=1, maxiter
      update boundary information
      DO i=1, n
            receive from left
            DO j=firstcol, lastcol       /* distributed  */
                  compute b(j)(i)
                  lbhook0          /* high overhead  */
            END DO
            lbhook1                /* ok             */
            send to right
      END DO
      lbhook2                      /* poor response  */
END DO
lbhook3                            /* useless        */
```

(a) Original code

```
DO iter=1,maxiter
      update boundary information
      DO i0=1, n/blocksize
            receive block from left
            DO i=BKLO(i0), BKHI(i0)
                  DO j=firstcol, lastcol       /* distributed  */
                        compute b(j)(i)
                        lbhook0          /* high overhead  */
                  END DO
                  lbhook1                /* ok             */
            END DO
            lbhook1a                     /* controllable   */
            send block to right
      END DO
      lbhook2                            /* poor response  */
END DO
lbhook3                                  /* useless        */
```

(b) Strip-mined code

Figure 15: Pseudocode for SOR showing possible locations for load balancing hook. The comments indicate the evaluation of each of the hook locations for SOR, knowing that computing $b[j][i]$ only requires a few operations.

5.1.1 Possible hook locations

The iterations of the distributed loop are treated as atomic units of execution. In the parallelized code, hooks can be placed anywhere outside the body of the distributed loop. Since all possible hook locations at the same level of a loop nest are equivalent with regard to frequency of execution, we identify only one hook location at each level of a loop nest. Therefore, the initial set of possible hook locations is at the end of the body of the distributed loop, immediately following the distributed loop, and immediately following each loop enclosing the distributed loop (e.g., Figure 15a). The outermost position is immediately removed from consideration because load balancing can not reduce execution time after the distributed computation has been completed.

Because interacting with the load balancer requires communication, when possible, we shift the potential hook locations to points next to existing communication at the same nest level so that additional synchronization points are not created. Thus, when the application requires communication at some nest level, the hook at that level is shifted to the point immediately preceding the first communication following the distributed loop.

5.1.2 Selecting from among possible hook locations

A compiler evaluates the potential hook locations from the most deeply nested location to the least deeply nested. Each hook location is evaluated by comparing the estimated execution time for a hook that does not call the load balancing code (control of the overhead when load balancing code is called is handled at run time) with an estimate of the execution time for the code between executions of a hook at that location. If the cost of a hook is a significant fraction (e.g., greater than 1%) of the cost of the code executed between run-time instances of the location, then the location is eliminated from consideration. The first hook that has a low enough overhead is selected, since it corresponds to the highest load balancing frequency that can be selected without excessive overhead (Figure 15a). Because this decision is only concerned with orders of magnitude, operation counts can be used for estimating execution time, so since a hook consists of several operations, code between executions of the hook must require several hundred operations. If the code between hook executions includes loops with bounds that can not be determined at compile time,

```
DO i = 1, n
      DO j = locallo, localhi
            c(j)(i) = 0
            DO k = 1, n
                  c(j)(i) = c(j)(i) + a(i)(k) * b(j)(k)
            END DO
            lbhook0           /* high overhead   */
      END DO
      lbhook1                 /* poor response   */
END DO
lbhook2                       /* useless         */
```

a) Original code

```
DO i = 1, n
      DO j0 = locallo, (localhi - locallo) / blocksize
            DO j = BKLO(j0), BKHI(j0)
                  c(j)(i) = 0
                  DO k = 1, n
                        c(j)(i) = c(j)(i) + a(i)(k) * b(j)(k)
                  ENDDO
                  lbhook0           /* high overhead   */
            END DO
            lbhook0a                /* controllable    */
      END DO
      lbhook1                       /* poor response   */
END DO
lbhook2                             /* useless         */
```

b) Strip-mined code

Figure 16: Pseudocode for MM showing possible locations for load balancing hook. The comments indicate the evaluation of each of the possible locations for MM.

the compiler can make assumptions about the number of iterations in the loop and/or use hints from the programmer to estimate the operation count for the code. Alternatively, the compiler can generate multiple versions of the loop nest, each with the hook placed at a different nest level; the appropriate version of the loop nest can be selected at run time based on the actual loop bounds.

In some cases, due to large loop bounds, none of the available locations may be appropriate. An example is shown in Figure 16a for matrix multiplication: for some problem sizes neither location is acceptable. In such situations, the compiler can use strip mining to create an intermediate choice. When a large loop is split into two nested loops (e.g., Figure 16b), the bounds of the inner of the loops and thus the frequency of execution of the new location (*lbhook0a*) can be controlled by the block size at run time to give the proper balance of overhead and responsiveness. In the SOR case (Figure 15b), strip-mining is already used to control grain size so, while the possible hook location (*lbhook1a*) added by strip-mining may be the best location for the load balancing hook, the block size can not be used to control hook frequency because controlling the grain size for the application is more important. Strip mining can also be useful when loop bounds are unknown at compile time [29].

5.2 Selection of load balancing frequency at run time

Several factors in the execution environment contribute to load balancing overhead, and thus limit the load balancing frequency. Communication costs are the main factor, as they influence the cost of interactions between the slaves and load balancer and the cost of work movement. Frequent interactions between the slaves and load balancer can make the overhead unacceptable, so their cost puts an upper limit on the load balancing frequency. Moreover, the overhead associated with moving work means that it is impractical to trace load changes that happen very quickly, and trying to do so will result in unnecessary overhead. Another factor influencing the overhead is the scheduling granularity—the *time quantum*— used by the operating system; CPU scheduling by the operating system interacts with the measurements performed by the slaves for performance evaluation and with the synchronizations performed by the application. All of these factors (summarized in Figure 17) place upper limits on the load balancing frequency and lower limits on the load balancing period. In this section, we will discuss each of these factors separately, and then describe how they are combined in selecting a target load balancing period and controlling the period at run time.

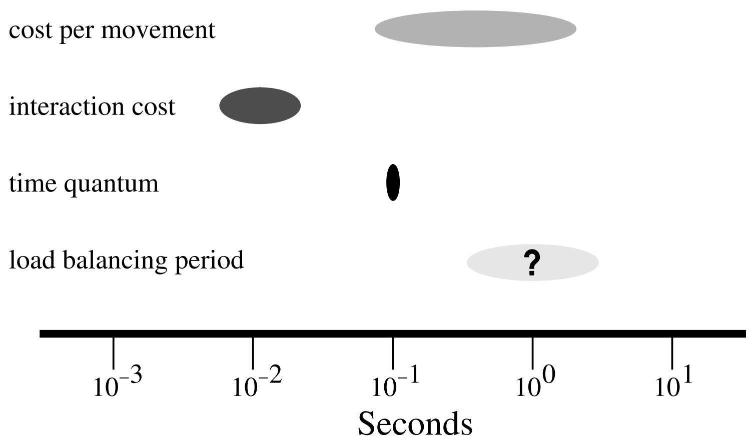


Figure 17: Periods affecting selection of load balancing period.

5.2.1 Interaction overhead

Collecting performance information and interacting with the load balancer adds overhead, even if the system is balanced. This overhead is proportional to the number of times the interactions occur. The load balancing period should be long enough that the total interaction costs are a small fraction, $f_{interact}$, e.g., less than 5 percent, of the total computation time. For synchronous load balancing, the time for a load balancing interaction is the sum of the times to collect and send performance information to the load balancer, to compute instructions, and to deliver the instructions to the slave processors. The average time for an interaction with the load balancer, $t_{interact}$, can be determined at the start of the computation by passing dummy load balancing information back and forth between the slaves and load balancer. During the computation, the interaction time may vary with loads on the processors and delays in the communication network. With synchronous load balancing, interaction costs can be updated as the program executes. However, for pipelined load balancing, much of the interaction cost is hidden so it is difficult to measure. Fortunately, because the costs are hidden, the $t_{interact}$ measured at startup time is actually a high estimate for the pipelined case so variations in loads and delays are unlikely to affect the overhead. The lower limit on the load balancing period due to the interaction costs is computed as follows:

$$period_{interact} = \frac{t_{interact}}{f_{interact}} \tag{11}$$

5.2.2 Cost of work movement

The cost of work movement should also be considered in selecting the frequency of interaction with the load balancer. However, for responsiveness, it is useful to track performance more frequently than it is profitable to move work, assuming that work will not be moved every time load balancing interactions occur. The work movement costs can be distributed over several load balancing periods. Also, the average work movement cost per load balancing period need not be limited to be a small fraction of the load balancing period because the work movement has benefits—load balancing resulting in improved utilization of resources—as well as costs. Therefore, the selection of the load balancing period allows the typical work movement cost to be several times the load balancing period:

$$period_{movement} = \frac{t_{movement}}{workscale} \tag{12}$$

$t_{movement}$ is an estimate of the average work movement costs determined by averaging the costs of the previous few work movements. ($t_{movement}$ can not be estimated at startup time because the work movement costs depend on the load imbalance in the system.) $workscale$ is a scaling factor which accounts for the typical period over which work movement costs are distributed and is determined by averaging the measured times between recent work movements. Because rapid response to performance changes is important and because work movement costs are difficult to determine accurately, $workscale$ is chosen so that the work movement costs are rarely the critical factor in selection of the target load balancing period; the work movement costs should only affect the target load balancing period when the costs are so high that work should not be moved at all.

5.2.3 Interaction with time quantum

Finally, the load balancing period determines the period over which performance is measured and should be selected so that the scheduling mechanism used by the operating system does not interfere with performance measurements. In particular, when loads on processors are stable, work should not be redistributed in response to the context switching between processes. For example, if the load balancing period is smaller than the time quantum and a processor has competing loads, some measurements on that processor will show the load balanced application getting the full performance of the CPU and others will show the application getting a fraction of the CPU. Thus, performance will appear to oscillate, resulting in work being moved back and forth between processors. To avoid oscillations in the measurements, the load balancing period must be large enough that performance variations due to context switching average out. Figure 18 indicates that the load balancing period must be several times (e.g., at least 5–10 times) as large as the time quantum for performance measurements to appear stable on a processor with constant competing loads; Siegell [29] presents analysis that agrees with these measurements. Thus, the time quantum sets another lower bound on the load balancing period:

$$period_{scheduling} = t_{quantum} \times quantumscale \tag{13}$$

5.2.4 Target load balancing period

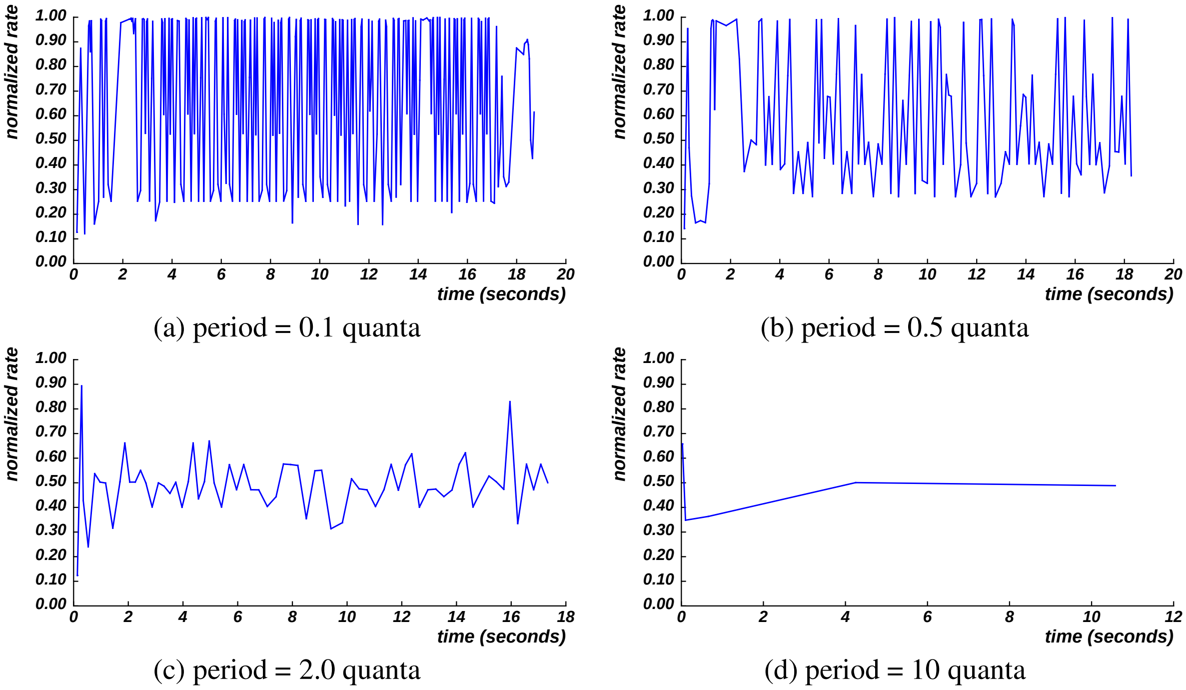
To minimize the time to respond to changes in performance, we set the target load balancing period, $period_{target}$ to be the maximum of the lower limits:

$$period_{target} = \max(period_{interact}, period_{movework}, period_{scheduling}) \tag{14}$$

With our implementation on Nectar, $period_{scheduling}$ is usually the maximum of the limits. The processors in the system are Unix workstations with 100 millisecond time quantums. A measurement period of at least 10 time quanta (i.e., $quantumscale = 10$) is necessary to eliminate the apparent fluctuations on a stable loaded system to the extent that they will be ignored by the load balancer (Figure 18). Thus, the target load balancing period is set at approximately 1.0 seconds.

Achieving the target load balancing period requires converting the period from seconds to the appropriate number of computation phases to execute between calls to the load balancing routines. Thus, each time the load balancer is invoked, the average time for a computation phase, $period_{compute}$, is computed and used to calculate $nexthook$ (Figure 14):

$$nexthook = \frac{period_{target}}{period_{compute}} \tag{15}$$



(a) period = 0.1 quanta

(b) period = 0.5 quanta

(c) period = 2.0 quanta

(d) period = 10 quanta

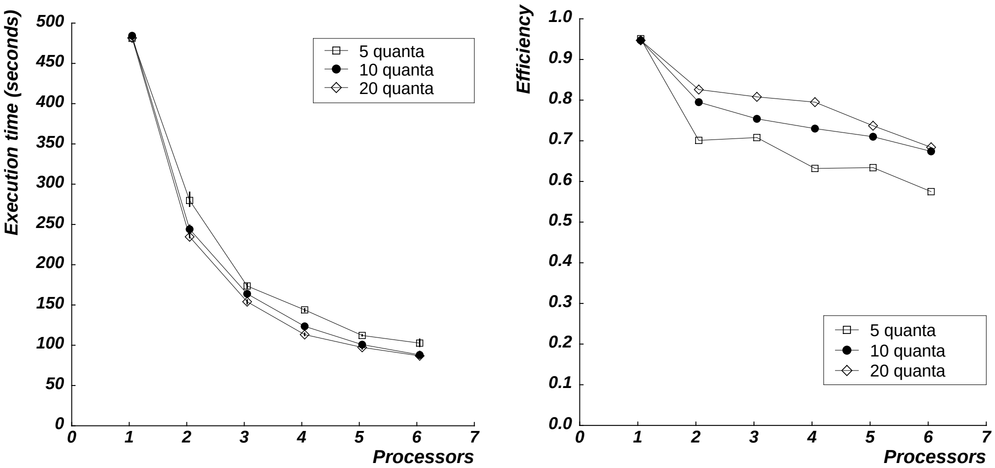
Figure 18: Effect of measurement period on stability of measurements. The rate of computation of matrix multiplication iterations is measured with a single competing computation intensive load on processor. Rates are normalized against the maximum computation rate for the application on the processor. The application should receive approximately 50% of maximum computation rate.

Because $period_{compute}$ varies, the actual load balancing period fluctuates around the target period. Part of the load balancing instruction sent to each slave is the number of computation phases to execute before interacting with the load balancer again.

5.3 Effect of load balancing frequency on performance

Load balancing frequency has a significant effect on application performance. Figure 19 shows how increasing the load balancing period improves parallelization efficiency for the SOR example. The load balancing parameters were selected to isolate the effect of changing the load balancing period.[4] The period is controlled by changing the value of $quantumscale$ (Section 5.2.3), the dominant factor in selecting the period for our target environment. For the measurements presented in this paper (Section 7), a 1.0 second period (the middle curve) was used. We found this to be a good compromise between responsiveness and overhead across a range of environments. Moreover, optimizations in the load balancing decision making process, as discussed in the next section, raise the efficiency close to the level attained with $quantumscale = 2.0$.

[4] The load balancing parameters for the data presented are as follows: load balancing interactions are not pipelined; 0% predicted improvement is required for work movement; raw (unfiltered) rate information is used; cost-benefit analysis is disabled. The grain size for the application is selected automatically as described in Section 4.2.

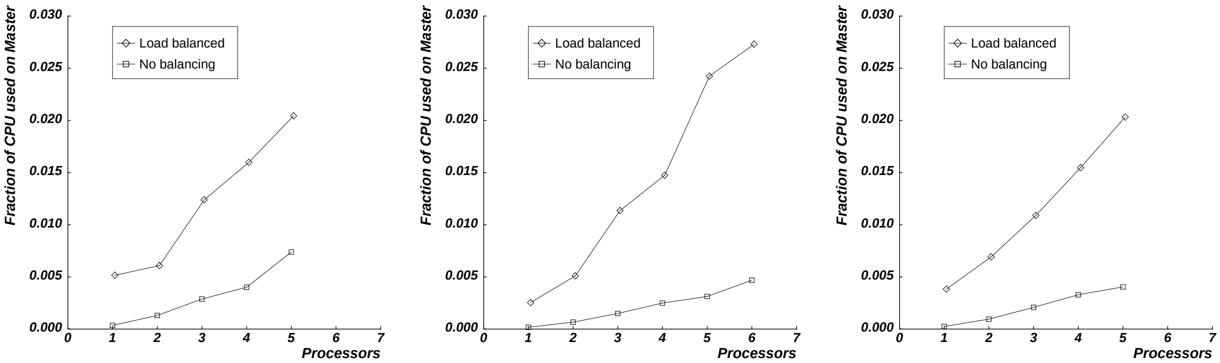


a) Computation time b) Efficiency

Figure 19: Effect of load balancing period on efficiency for 1000×1000 SOR (40 iterations) with oscillating load (period = 60 seconds) on one processor.[4]

5.4 Effectiveness of frequency selection in limiting overhead

The cost of interaction with the central load balancer increases as the number of slave processors is increased. However, for a fixed number of nodes, the frequency selection mechanism naturally limits the load on the load balancer, and pipelining the interactions between the slaves and the load balancer (Section 2.1) removes most of the interaction time from the critical path for the application. To evaluate the effectiveness of the frequency selection mechanism, we measure the CPU time used by the master process using the *getrusage* function [7] provided with Unix.



a) dedicated system b) constant load on one processor c) oscillating load (period = 20 seconds)

Figure 20: Fraction of CPU used on master processor for 500×500 matrix multiplication.[5]

Figure 20 shows the CPU usage by the master process as a fraction of the elapsed time.[5] We observe

[5] The load balancing parameters for the data presented are as follows: pipelined load balancing interactions; load balancing target period is 0.5 second; 10% predicted improvement required for work movement; rate information filtered with simple filter ($h = 0.8$); cost-benefit analysis enabled. Since a high load balancing frequency (5 quanta) is used and all stages of the load balancer are enabled, the data can be viewed as an upperbound on utilization.

that load balancing uses only a small fraction of the available cycles, for up to to the maximum number of slaves in our system. The trends in the data indicate that the master process could handle many more slaves before the central load balancer would become a limiting factor in system performance. Note that although the load balancing algorithm is O($P \log P$) due to a sorting operation, the majority of the time is due to the O(P) portions of the algorithm. Also, the similarity of the three graphs in Figure 20 indicates that the fraction used by the master process is affected only slightly by the loads on the slave processors. The only task that is load-dependent is the generation of load balancing instructions, which is only performed when an imbalance is detected.

6 Control of the work movement

Ideally, one would like to make use of every possible computing cycle on each node, i.e. redistribute the load instantaneously in response to every change in load. In practice, the cost of work movement makes this impractical, and overly aggressive work movement results in slowdown instead of speedup. More specifically, there are two areas of concern:

- Changes in load that happen too quickly should not be tracked, because the cost will outweigh the benefit. Another way of looking at this is that one has to have some confidence that a load change will last before adapting to it.
- Small changes in load should not be tracked. Tracking small changes will result in a slowdown, because of the high cost associated with even the smallest work movement. Plus, one should expect to see small load fluctuations even in a stable and balanced system.

Figure 21 outlines the instruction generation process used by our central load balancer. The load balancer uses the following three mechanisms to decide whether load balancing is appropriate:

- The load balancer averages new measurements with older data to filter out short-term load changes.
- The load balancer only considers work movement if the load imbalance is above a certain threshold.
- Before moving work, the load balancer does a cost-benefit analysis.

In the remainder of this section we describe how the parameters for each of these mechanisms are selected based on compile-time and run-time information. We observed that each of the above mechanisms results in small improvements [29], but that all three mechanisms have to be enable to achieve good load balancing. For this reason, we will limit the performance evaluation of the individual mechanisms. Performance results for the complete system are presented in the next section.

6.1 Filtering load measurements

Each time the load balancer is invoked, the load balancer computes a new work distribution that allocates work to the processors in proportion to their reported processing resources represented by their *computation rate* expressed in work units computed per second. To filter out short-term load fluctuations and noise components, the raw measurement is replaced for the purpose of computing a new work distribution, with a weighted average of the raw computation rate and previous computation rate measurements using a simple filtering function:

$$r'_i = (1 - h) \times r_i + h \times r'_{i-1} \tag{16}$$

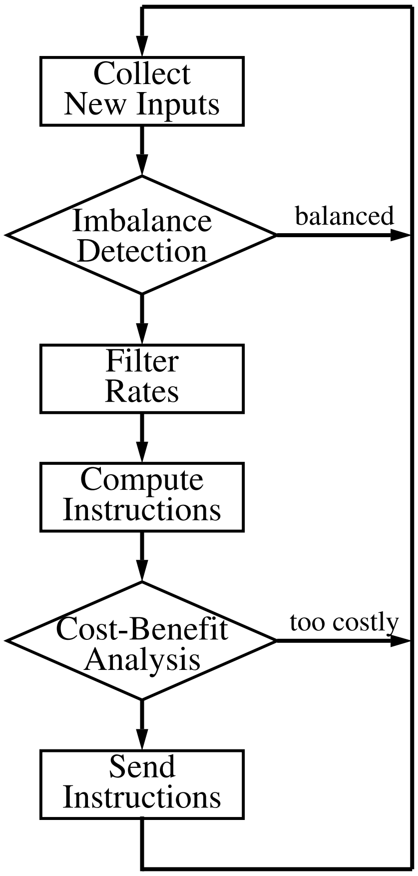


Figure 21: Decision process used by load balancer.

Equation 16 is a recursive discrete-time lowpass filter. r'_i is the filter output, the *adjusted rate* for the most recent computation phase, r_i is the raw rate for the most recent computation phase, and r'_{i-1} is the adjusted rate for the previous computation phase and incorporates all previous measurements. h ($0 \leq h < 1$), the *history fraction*, is the contribution of the old information to the new adjusted rate.

The selection of an appropriate value for h is a difficult task. h can range from a constant to an arbitrary function that takes all previously measured values as inputs. With h equal to a fixed value, changes in performance are attenuated without regard to performance trends; for some values, response is too slow, and for others, fluctuations in performance are not attenuated enough. The main shortcoming of a filter based on a fixed h is that it treats load increases and decreases in the same way. In practice it is more important to respond quickly to decreases in performance than increases because a decrease in performance from one processor causes all other processors to wait when the processors synchronize, while an increase in performance on one processor has no effect on the productivity of the other processors. A constant value for h is inadequate to achieve this. The weight of the filter has to be calculated using a function that incorporates recent performance trends.

Appropriate values for h can be calculated using a state machine:

$$(h_{next}, state_{next}) = f(measurements, state_{current}) \tag{17}$$

The state basically keeps track of the direction and duration of the performance trends. The amount of history incorporated in the trend information can be increased by increasing the number of state bits. In the state machine we implemented, shown in Table 1, we trust downward trends in the computation rate sooner (i.e., smaller h) than upward trends because the penalties are greater if one processor is overloaded compared with the case that one processor does not have enough work assigned to it. The specific values for the output of the state machine were selected empirically.

Input	State	Next State	History fraction (h)
increase	DOWN3	DOWN1	1.0 (all history)
increase	DOWN2	CONSTANT	1.0 (all history)
increase	DOWN1	UP1	1.0 (all history)
increase	CONSTANT	UP1	0.8
increase	UP1	UP2	0.6
increase	UP2	UP3	0.4
increase	UP3	UP3	0.2
decrease	DOWN3	DOWN3	0.1
decrease	DOWN2	DOWN3	0.1
decrease	DOWN1	DOWN2	0.2
decrease	CONSTANT	DOWN1	0.3
decrease	UP1	DOWN1	0.4
decrease	UP2	DOWN1	0.5
decrease	UP3	CONSTANT	0.6

Table 1: State table for computing h. The input is *increase* if raw performance increases or stays the same relative to the previous adjusted performance. The input is *decrease* if raw performance decreases relative to the previous adjusted performance.

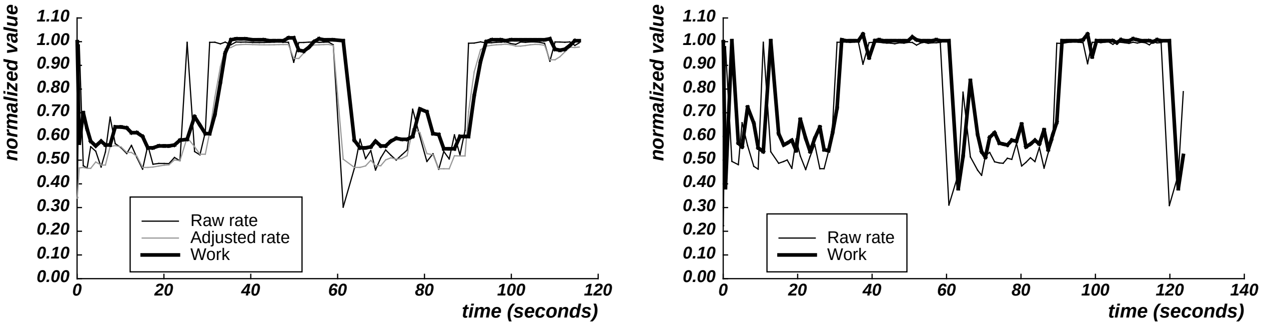
Effect of filtering on performance. Figure 22 illustrates the effect of filtering; the load balancing parameters were selected to isolate the effects of filtering.[6] The run in Figure 22a uses the filtered rate information; the work allocation curve has the same shape as the filtered rate curve (except for minor differences due to small rate fluctuations on other processors), but it is shifted to the right. In Figure 22b, a run without filtering, the system responds to all fluctuations on the loaded processor, and the work allocation curve has the same shape as the raw computation rate curve; the efficiency is lower due because there is more unnecessary work movement.

The effect of filtering on efficiency is very similar to that of increasing the load balancing period (see Figure 19) because both filtering and increasing the period cause performance to be averaged over a longer period of time (although, in this case, the filtering is a weighted average) so that fluctuations in performance can cancel each other out. Thus, with filtering, a shorter load balancing period can be used to allow the load balancing system to respond to changes in performance more quickly.

6.2 Imbalance threshold

The slaves are instructed not to move work if redistributing work into the optimal distribution cannot reduce the projected execution time by a specified *threshold fraction* (e.g., 0.1). This provides the system with

[6] The load balancing parameters for the data presented are as follows: load balancing interactions are not pipelined; target load balancing period is 10 quanta (1 second); 0% predicted improvement is required for work movement; cost-benefit analysis is disabled.



a) With filtering (*efficiency* $= 0.769$) b) Without filtering (*efficiency* $= 0.715$)

Figure 22: Measured performance and resulting work allocation on loaded slave for 1000×1000 SOR (40 iterations) running on a 4 slave system with an oscillating load (period = 60 seconds) on one slave.[6] Low pass filtering reduces work movement in response to short term performance fluctuations.

hysteresis so that small performance fluctuations do not cause work to be moved back and forth between processors.

The threshold check, which is done as soon as status is received, determines the fraction by which the elapsed time for the computation would be reduced if the assessments of performance for the slave processors matched the actual performance for the next computation phase. This *reduction fraction*, $rfract$, is compared to the threshold fraction, $tfract$, to determine whether load balancing should be attempted. If $rfract < tfract$, load is considered to be balanced, and null instructions or no instructions are sent to the slaves. Otherwise, the computation of instructions continues.

Effect of imbalance threshold on performance. Figure 23 shows the motivation for using an imbalance threshold to decide when to redistribute work. In the figure, the raw rate is normalized against the maximum rate measured on the processor, and the allocated work is normalized against the work that would be allocated to the processor if work were distributed equally to the processors. The competing load on the processor is constant, consuming about half of the computing resources of the processor, but small variations in performance (the raw rate) are still observed. Redistributing work in response to these small fluctuations would not be beneficial due to high fixed costs of moving work. Figure 23 demonstrates that the threshold (along with other optimizations[7]) prevents work movement from occurring in this case: after an initial period of instability, the work allocated to the processor remains constant.

We executed the SOR example with an imbalance threshold of 10% (an anticipated 10% improvement is required for redistributing work) and without an imbalance threshold (work is redistributed whenever the observed computation rates change). Figure 24 shows that the goal of eliminating movement of small amounts of work is achieved: in Figure 24a the work allocation curve is smooth, but in Figure 24b work is moved in larger chunks. However using the threshold sometimes allows the system to remain unbalanced, as in Figure 24c.

In some cases, using an imbalance threshold is not advantageous because imbalance at almost the threshold level may remain. In Figure 24b, the performance changes are such that work movement tracks performance well; the efficiency in this case was 77.7%. However, in Figure 24c, generated from a different run in the same environmental conditions, work movement does not track the computation rate as well. The

[7] The load balancing parameters for the data presented are as follows: load balancing interactions are pipelined; target load balancing period is 10 quanta (1 second); 10% predicted improvement is required for work movement; filtering is enabled; cost-benefit analysis is enabled.

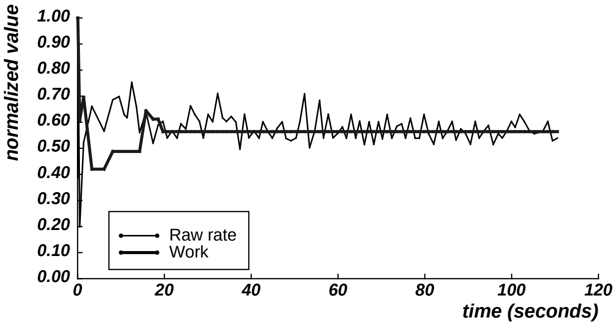


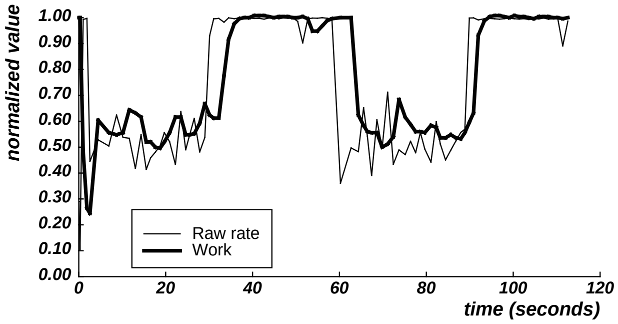
Figure 23: Measured performance and resulting work allocation on loaded slave for 1000×1000 SOR (40 iterations) running on a 4 slave system with a constant computation-intensive load on one slave.[7] The imbalance threshold helps reduce work movement in response to small fluctuations.

measured performance jumped to a point just within the threshold fraction of the maximum performance before reaching the maximum performance, so load was not redistributed when the performance later reached the maximum performance level, and imbalance led to a reduction in efficiency, to 75.6%. Note that although in Figure 24c it appears that a 20–25% improvement in throughput might be attained by shifting more work to the processor at about the 35 second point, the improvement would only be observed on one of the four processors in the system, so the total improvement would not exceed the 10% threshold. Figures 24b and 24c are based on data from two runs with the same parameters under the same conditions; the difference in how the load balancing system responded in the two runs is just due to random timing variations. Further research and analysis of the tradeoffs between responsiveness and work movement costs is needed to maximize the performance improvements attained when an imbalance detection phase is included. A dynamic threshold that periodically allows the system to make minor adjustments to the work distribution, preventing the unresponsiveness demonstrated in Figure 24c, might for example improve performance.

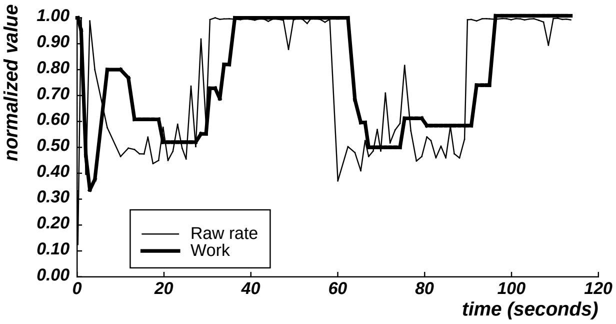
6.3 Profitability determination

Work should not be moved if the costs of moving the work exceed the benefits. The optimizations described in the previous sections attempt to eliminate unnecessary work movement but are performed before instructions are computed so they do not consider the actual work movement costs. Since work movement instructions are generated without considering the cost of work movement, a *profitability determination phase* [38] is added to verify that work movement makes sense. If the estimated costs of movement exceed the estimated benefits then the work movement instructions are cancelled. However, we give more weight to the benefits because cancelling instructions can delay the load balancer's reaction to real changes in the system and reduce the effectiveness of load balancing.

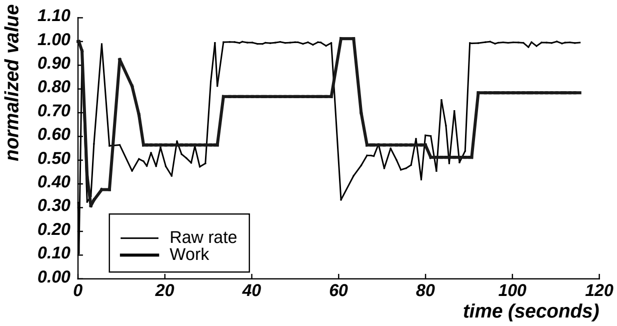
Once instructions have been generated, predictions of work movement costs are made based on the amount of work to be moved and measurements of past work movement times. Benefits are estimated using the reduction fraction (Section 6.2) and a projection of the time that the system will remain stable in the future. The latter value is determined based on the frequency of performance changes in the past, but the value is a very gross estimate because it is not really possible to predict how the loads on the processors will change in the future. Therefore, we leave a wide margin of error for the benefit estimate and only cancel instructions if the cost estimate greatly exceeds the projected benefits.



a) No improvement required.



b) 10% improvement required.



c) 10% improvement required.

Figure 24: Measured performance and resulting work allocation on loaded slave for 1000×1000 SOR (40 iterations) running on a 4 slave system with an oscillating load (period = 60 seconds) on one slave.[7] The imbalance threshold reduces work movement but may result in suboptimal work distribution.

Effect of Profitability Determination on Performance. Our measurements show that the profitability determination phase is mainly needed to prevent work movement when the work movement costs are very high or when the loads on the processors are very unstable. On the Nectar system with the oscillating competing loads, the profitability determination phase typically had little effect. In Section 8, we will show an example where the profitability determination phase does substantially improve performance with dynamic load balancing.

7 Evaluation

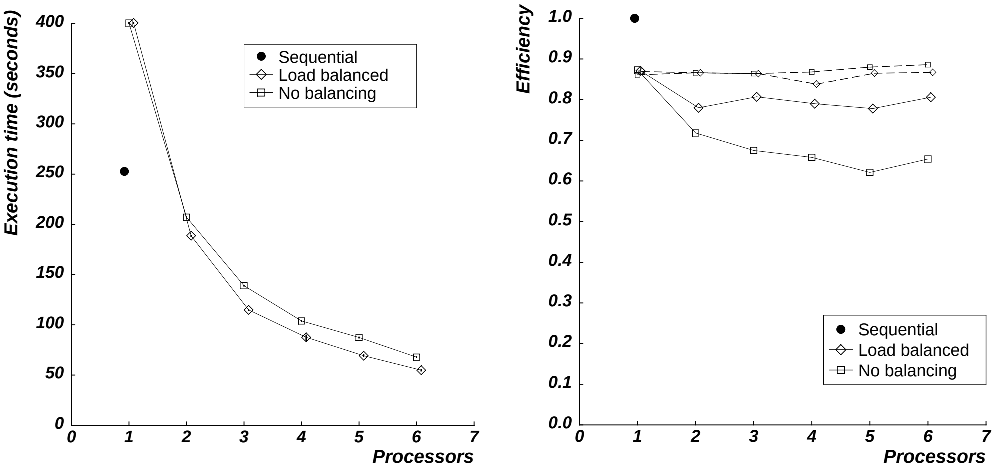
To evaluate the mechanisms for supporting dynamic load balancing described in this paper, we implemented a run-time system on the Nectar system [3], a set of workstations connected by a high-speed fiber optic network. We hand-parallelized versions of matrix multiplication (MM) and successive overrelaxation (SOR) and took measurements in controlled environments. For our experiments, we used a homogeneous set of Sun 4/330 processors. Artificial competing loads were started along with the application so that the computation resources used by the competing loads could be easily measured; CPU usage of the competing processes was measured by the *getrusage* function [7] provided with Unix.

In environments with no competing loads, i.e., on a dedicated system, the initial equal distribution of work should balance the load, and dynamic load balancing should not improve performance. For our example applications, performance with load balancing was only slightly worse than performance without load balancing due to the added overhead of interactions with the load balancer. However, neither case attains 100% efficiency due to inherent inefficiencies in the parallelization, e.g., due to communication overhead for the SOR example. We treat the performance measured on a dedicated system as an upper bound on performance for environments with competing loads, and, for reference, present the data from the dedicated system (indicated by dashed lines) along with the data measured in the other environments.

With a constant competing load on one of the processors, work is initially shifted to compensate for the competing load, and then the work distribution remains constant. The measured efficiencies are very similar to those in the dedicated environment, except when the size of the data moved to initially balance the load is very large [29].

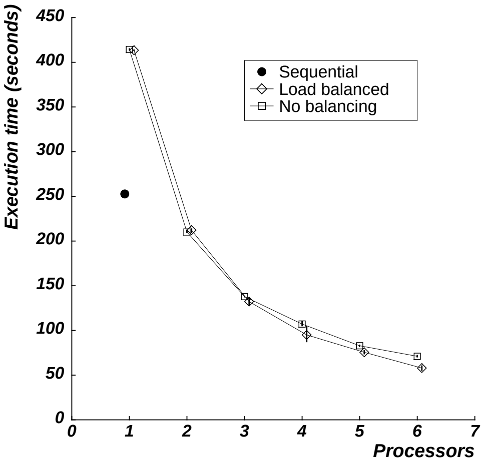
To characterize the responsiveness of the load balancer to dynamic loads, we applied an oscillating load with two different periods to our applications. Note when dynamic load balancing is used, there is always the risk of reducing performance. For example, competing loads can change right after work has been moved, thus making the work movement counter productive. One can only guarantee that work movements will improve performance if one has knowledge of how competing loads will behave in the future, something that is typically not the case. A performance loss is especially likely for oscillating loads with a period that roughly matches the response time of the load balancer, which is one of the cases that we will show.

Figures 25–30 compare the performance of the application versions with and without load balancing. Each point is the average of at least three measurements, and the range of measured values is shown in the execution time graphs. For the lower oscillation frequency (period = 60 seconds), the load balancer was able to track the changes in performance and redistribute work in a profitable manner . However, for the higher oscillation frequency (period = 20 seconds), the increased total work movement costs and increased significance of the delay in the load balancer made load balancing less profitable and for the 2000×2000 SOR example, dynamic load balancing even hurts performance considerably. As explained above, the reason is that fairly soon after work has been moved, the competing load changes, and the system does not have a chance of benefiting from the investment made in work movement. Load balancing becomes unproductive for lower frequencies for the 2000×2000 SOR example than for the other examples, because of the higher cost of work movement.

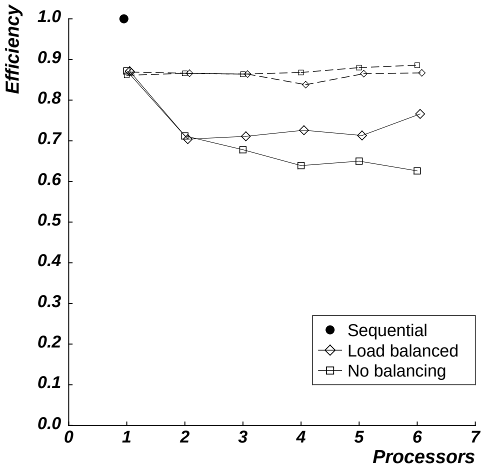


a) Execution time b) Efficiency

Figure 25: Execution time and efficiency for 500×500 matrix multiplication running in homogeneous environment with oscillating load (period = 60 sec, duration = 30 sec) on first processor.

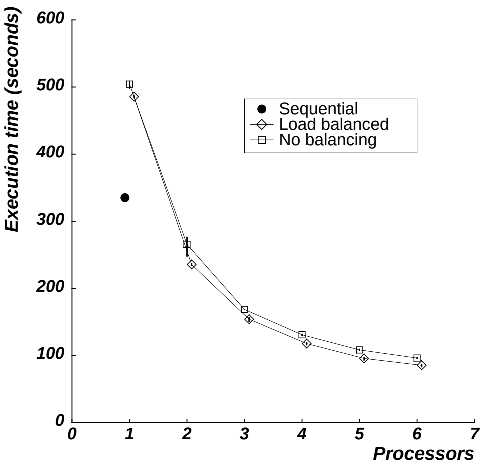


a) Execution time

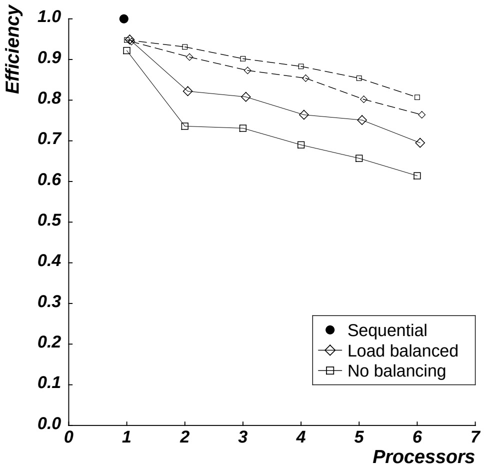


b) Efficiency

Figure 26: Execution time and efficiency for 500 $\times$ 500 matrix multiplication running in homogeneous environment with oscillating load (period = 20 sec, duration = 10 sec) on first processor.



a) Execution time

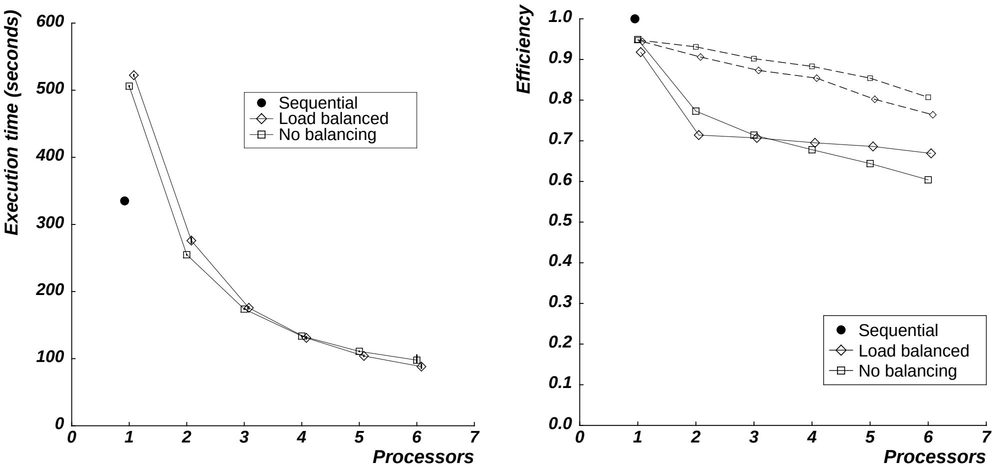


b) Efficiency

Figure 27: Execution time and efficiency for 1000 $\times$ 1000 SOR (40 iterations) running in homogeneous environment with oscillating load (period = 60 sec, duration = 30 sec) on first processor.

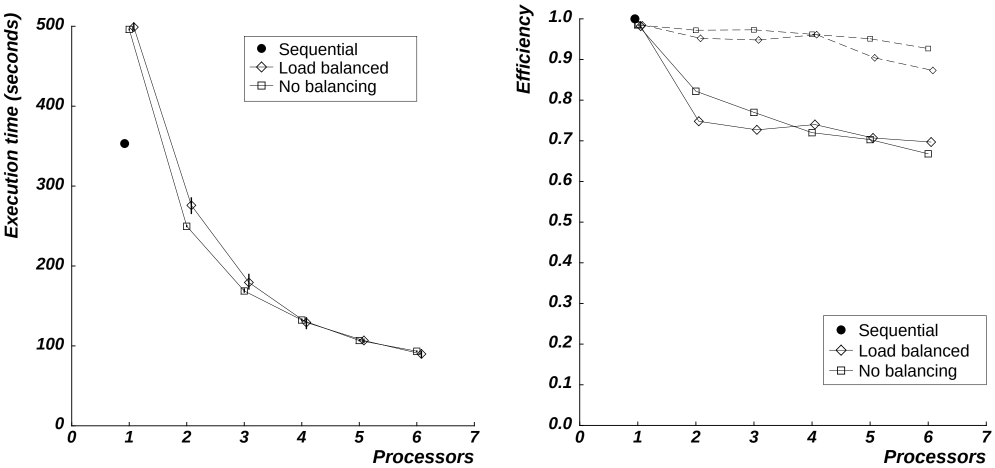
8 Modeling load balancing performance

To verify and quantify the sources of efficiency loss in a dynamic system, we modeled dynamic load balancing performance for a system with an oscillating load on one processor. Our model, shown in Figure 31, considers work movement costs and the delay between a change in performance on a processor and the change in allocation of work to that processor. However, the model does not include the effects of filtering, the imbalance threshold, or the cost-benefit analysis. Note the similarity between the model and the graphs shown in Figures 22 and 24. In Figure 31, the thick dashed lines shows the fraction of the CPU that the application should have, under perfect conditions; t_a and t_b are the time intervals that the competing load is off and on respectively. The black curve shows how much work is assigned to the node, and the grey areas indicate work movement. During t_c and t_f, the load balancer has not yet responded to the change in



a) Execution time   b) Efficiency

Figure 28: Execution time and efficiency for 1000 $\times$ 1000 SOR (40 iterations) running in homogeneous environment with oscillating load (period = 20 sec, duration = 10 sec) on first processor.

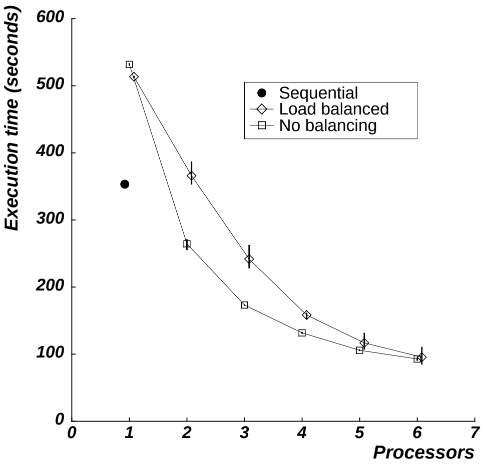


a) Execution time   b) Efficiency

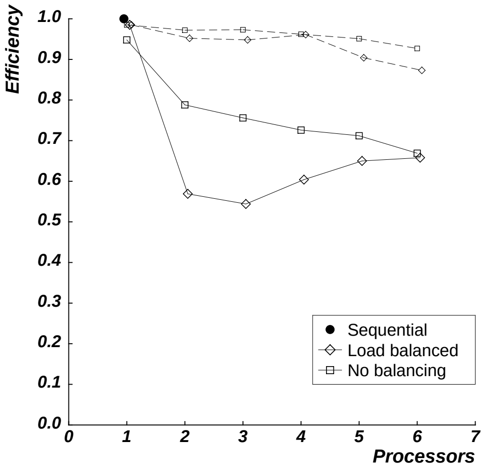
Figure 29: Execution time and efficiency for 2000 $\times$ 2000 SOR (10 iterations) running in homogeneous environment with oscillating load (period = 60 sec, duration = 30 sec) on first processor.

performance so only part of the available resources are used productively. During t_d and t_g, work is being moved between processors, and none of the resources are being used productively for the computation. Finally, during t_e and t_h, load is balanced and all of the available resources can be used productively. However, the efficiency during t_e and t_h is limited by the parallelization efficiency for the application, i.e., the efficiency measured in the dedicated system. By substituting estimates of the work movement costs into the model we can estimate the efficiency (see citeSiegell95 for details).

We found that for most scenarios, the model predicts the measured performance reasonable well. Examples are the matrix multiply and 1000 $\times$ 1000 SOR shown in Figures 32 and 33. However for the 2000 $\times$ 2000 SOR example, the measured efficiencies are actually better than those predicted by the model. A careful analysis of 2000 $\times$ 2000 SOR shows that the difference is caused by the profitability analysis and imbalance threshold, which are not captured by the model. Because the data that must be moved is large, the



a) Execution time



b) Efficiency

Figure 30: Execution time and efficiency for 2000 × 2000 SOR (10 iterations) running in homogeneous environment with oscillating load (period = 20 sec, duration = 10 sec) on first processor.

cost benefit analysis determines that work movement is not profitable and often cancels work movement. Also, as the number of processors increases, the imbalance detection phase allows greater imbalance to exist, again resulting in less work movement than predicted by the model.

The model allows us to estimate what overheads are responsible for the reduction in efficiency relative to the perfectly balanced application without competing loads (dashed curves in the graphs). The main sources of inefficiency are work movement and the delay with which the load balancer responds to changes in load. While the results depend on the specific scenario, i.e. frequency of competing load and size of the data to be moved, we found that in most cases the work movement contributed most to the inefficiency, certainly for the larger data sizes.

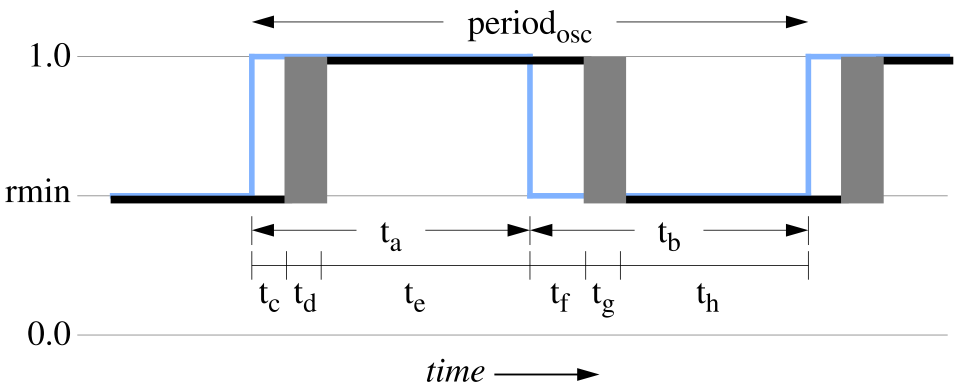
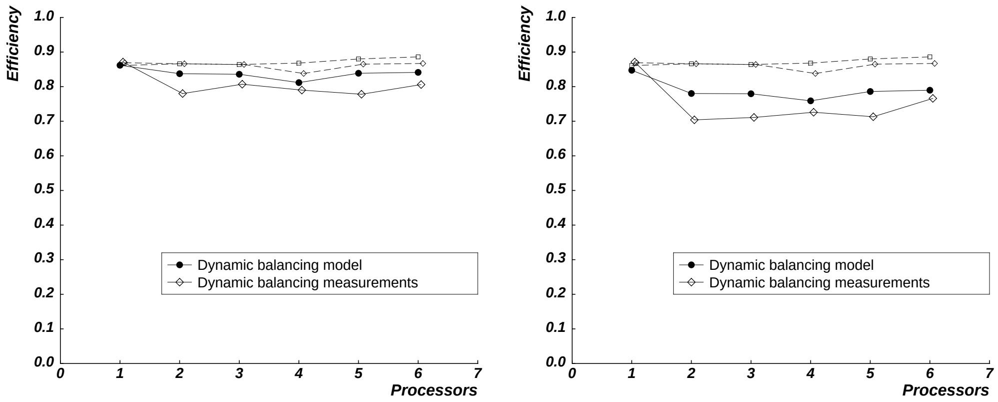


Figure 31: Dynamic load balancing model for predicting performance.

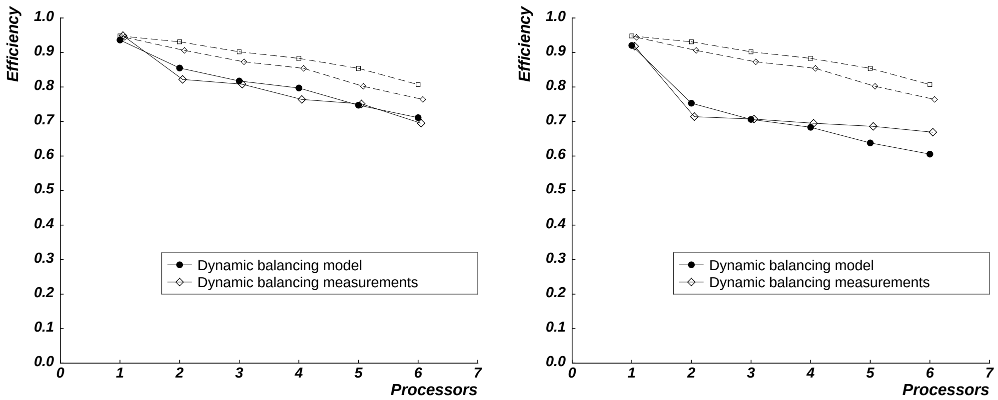
9 Other load balancing models

Our centralized load balancing system is only one of the many load balancing models that are applicable to distributed loops. While these models often differ substantially in features overhead, scalability, rate of



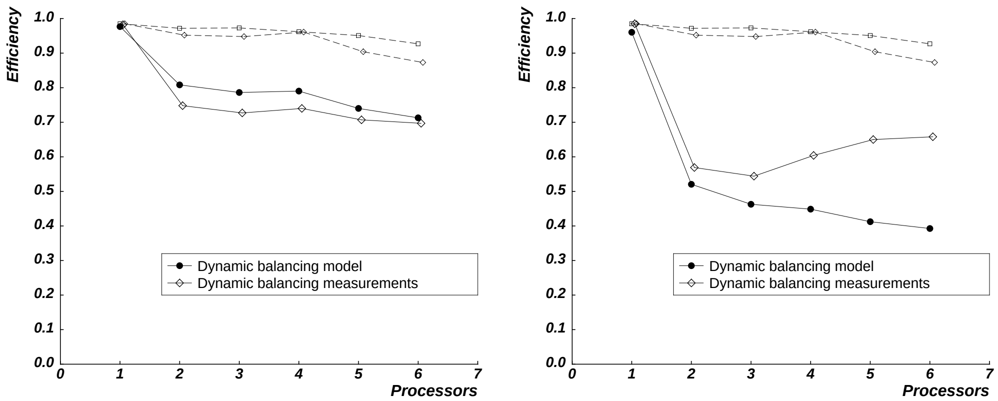
a) 60 second oscillation period b) 20 second oscillation period

Figure 32: Efficiency predicted by model for 500 $\times$ 500 MM running in homogeneous environment with oscillating load on first processor.



a) 60 second oscillation period b) 20 second oscillation period

Figure 33: Efficiency predicted by model for 1000 $\times$ 1000 SOR running in homogeneous environment with oscillating load on first processor.



a) 60 second oscillation period b) 20 second oscillation period

Figure 34: Efficiency predicted by model for 2000 $\times$ 2000 SOR running in homogeneous environment with oscillating load on first processor.

convergence, and application domain, all of them have to select an appropriate grain size, load balancing frequency and work movement parameters. In general, appropriate values will depend on both static and dynamic application and system parameters, and the general approach described in this paper will be applicable.

We expect the determination of the load balancing parameters (Sections 5 and 6) to readily apply to other models. For example, in a load balancing system based on diffusion, the load balancing frequency will be constrained by the same factors as in our centralized environment, even though the specific nature of the overheads can be different. One of the advantages of relying mainly on the direct measurement of overheads and parameters, is that it makes the approach more robust since measurements are less sensitive to variations in the exact nature of the overheads than model based estimates.

The result that is hardest to generalize is the grain size determination (Section 4). It relies on the same grain size being used by all nodes in the computation, and thus requires a global view of the computation. We expect the result to apply to load balancing systems that have access to global information, but not to those that rely exclusively on local information. Note that our approach depends heavily on a specific model of the computation, although this model does not include the load balancer. Note also that many of the load balancing systems described in the literature cannot support the pipelined applications considered in Section 4.

10 Related Work

General taxonomies for load balancing can be found in [37] [5]. In this section we will focus on dynamic load balancing of distributed loops.

Many of the approaches for scheduling of iterations of distributed loops are *task queue models* targeting shared memory architectures. Work units are kept in a logically central queue and are distributed to slave processors when the slave processors have finished their previously allocated work. Most of the research in this area is concerned with minimizing the overhead of accessing this central queue while minimizing the skew between the finishing times of the processors, and the approaches differ mainly in how they determine the granularity of the work movement [26] [27] [14] [36]. Information regarding interactions between the iterations is often lost due to the desire to have a single list of tasks (e.g., [14] [27]) and many of the approaches assume that iterations are independent, requiring no communication, and/or target a shared memory target architecture.

Recent research [20] [16] has added consideration for *processor affinity* to the task queue models so that locality and data reuse are taken into account: iterations that use the same data are assigned to the same processor unless they need to be moved to balance load. In *Affinity Scheduling* [20], data is moved to the local cache when first accessed, and the scheduling algorithm assigns iterations in blocks. In *Locality-based Dynamic Scheduling* [16], data is initially distributed in block, cyclic, etc. fashion, and each processor first executes the iterations which access local data. Both of these approaches still assume a shared memory environment.

Numerous other approaches have been proposed for scheduling loop iterations, especially if the iterations are independent. Some systems rely on the length or status of the local task queues to balance load [28][44]. In diffusion models, all work is distributed to the processors, and work is shifted between adjacent processors when processors detect an imbalance between their load and their neighbors'. In these models, control is based on near-neighbor information only [38]. The *Gradient Model* method [17] also passes information and work by communication between nearest neighbors, but uses a gradient surface which stores information about proximity to lightly loaded processors so that work can be propagated through intermediate processors, from heavily loaded processors to lightly loaded ones; global balancing is achieved by propagation and successive refinement of local load information.

The load balancing techniques we used are in part based on previous work in load balancing. Several groups [21][23][19] make load balancing decisions using relative computation rates instead indirect measures such as CPU utilization and load. A number of papers consider the stability of load balancing systems [32][6]. We use use filtering and an imbalance threshold to stabilize the load balanced application. There is some similar between our filtering based on system state (Table 1) and the statistical pattern recognition used in [10]. The use of cost-benefit analysis in load balancing of dynamic computations is discussed in [22]. However, our work differs in a number of aspects. First, we explicitly consider application data dependencies and loop structure when making load balancing decisions. Second, we support automatically generated code, i.e. we use both compiler-provided and runtime information to automatically select load balancing parameters.

The work that is closest to ours is the Fortran D compiler, which performs an analysis similar to ours to select the appropriate block size for pipelined computations [13]. As in our approach, the optimal block size is determined by setting the derivative of a model of the execution time for the application equal to zero. However, unlike our approach, their estimates of computation and communication times for the program are determined by a *static performance estimator* which runs a training set of kernel routines to characterize costs in the system [11]. The static performance estimator matches computations in the given application with kernels from the training set. Their approach requires a separate characterization for each machine configuration that might be used when running the application. Our approach is more flexible: since we rely on run-time measurements, we automatically adjust to the specific application (nature of computation and input size) and machine configuration (communication costs, number of processors, and processor speeds). However, by delaying our characterization of costs until run time, we add the characterization time to the cost of executing the application. Other groups have looked at grain size selection for other application domains, e.g. [1].

11 Conclusions

In this paper we describe the automatic selection of parameters for dynamic load balancing for applications consisting of parallelized DOALL and DOACROSS loops. We showed how the automatic selection of the parameters—based on parameters characterizing application and system features—usually involves cooperation between the compiler and runtime system:

- The compiler must restructure loops to increase grain size. For DOACROSS loops, strip mining provides a blocksize parameter that can be set at run time to control the grain size. We presented a method for selecting the optimal grain size for DOACROSS loops at run time.

- Load balancing frequency is controlled at run-time by load balancing hooks placed by the compiler. At run-time, the target frequency is selected based on the time quantum used by the operating system and measurements of load balancing interaction costs and work movement costs.

- Parameters that control work movement are calculated: filtering of the raw performance information; requiring that imbalance exceed a threshold for load balancing; and a cost–benefit analysis of the work movement instructions.

Measurements on the Nectar system show that dynamic load balancing with automatically selected parameters can improve the performance of parallelized applications in a dynamic environment.

References

[1] Gad Aharoni, Dror G. Feitelson, and Amnon Barak. A run-time algorithm for managing the granularity of parallel functional programs. *Journal of Functional Programming*, 2(4):387–405, October 1992.

[2] John R. Allen and Ken Kennedy. Automatic Loop Interchange. In *Proceedings of the ACM SIGPLAN '84 Symposium on Compiler Construction*, pages 233–246, Montreal, Canada, June 17–22 1984. ACM Special Interest Group on Programming Languages.

[3] Emmanuel Arnould, Francois Bitz, Eric Cooper, H. T. Kung, Robert Sansom, and Peter Steenkiste. The Design of Nectar: A Network Backplane for Heterogeneous Multicomputers. In *ASPLOS-III Proceedings*, pages 205–216. ACM/IEEE, April 1989.

[4] David Callahan and Ken Kennedy. Compiling Programs for Distributed-Memory Multiprocessors. *Journal of Supercomputing*, 2(2):151–169, October 1988.

[5] Thomas L. Casavant and Jon G. Kuhl. A Taxonomy of Scheduling in General-Purpose Distributed Computing Systems. *IEEE Trans. on Software Engineering*, 14(2):141–154, February 1988.

[6] Thomas L. Casavant and Jon G. Kuhl. Effects of Response and Stability on Scheduling in Distributed Computing Systems. *IEEE Transactions on Software Engineering*, 14(11):1578–1588, November 1988.

[7] Computer Systems Research Group, Computer Science Division, Department of Electrical Engineering and Computer Science, University of California, Berkeley, CA 94720. *UNIX Programmer's Reference Manual (PRM)*, 4.3 berkeley software distribution edition, April 1986.

[8] Michael J. Feeley, Brian N. Bershad, Jeffrey S. Chase, and Henry M. Levy. Dynamic Node Reconfiguration in a Parallel-Distributed Environment. In *Third ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 114–121, Williamsburg, VA, April 21–24 1991. Special Interest Group on Programming Languages, Association for Computing Machinery, ACM Press.

[9] Message Passing Interface Forum. MPI: A Message-Passing Interface Standard. Technical report, University of Tennessee, Knoxville, TN, May 1994.

[10] Kumar K. Goswami, Murthy Devarakonda, and Ravishankar K. Iyer. Prediction-Based Dynamic Load-Sharing Heuristics. *IEEE Transactions on Parallel and Distributed Systems*, 4(6):638–648, June 1993.

[11] S. Hiranandani, K. Kennedy, C. Koelbel, U. Kremer, and C.-W. Tseng. An Overview of the Fortran D Programming System. In U. Banerjee, D. Gelernter, A. Nicolau, and D. Padua, editors, *Languages and Compilers for Parallel Computing. Fourth International Workshop.*, pages 18–34, Santa Clara, CA, August 7–9 1991. Springer-Verlag.

[12] Seema Hiranandani, Ken Kennedy, and Chau-Wen Tseng. Compiling Fortran D for MIMD Distributed-Memory Machines. *Communications of the ACM*, 35(8):66–80, August 1992.

[13] Seema Hiranandani, Ken Kennedy, and Chau-Wen Tseng. Evaluation of Compiler Optimizations for Fortran D on MIMD Distributed-Memory Machines. In *Proceedings of the 1992 International Conference on Supercomputing*, pages 1–14, Washington, DC, July 19–23 1992.

[14] Susan Flynn Hummel, Edith Schonberg, and Lawrence E. Flynn. Factoring: A Practical and Robust Method for Scheduling Parallel Loops. In *Supercomputing '91 Proceedings*, pages 610–619, Albuquerque, NM, November 18–22 1991. IEEE Computer Society Press.

[15] Samuel J. Leffler, Marshall Kirk McKusick, Michael J. Karels, and John S. Quarterman. *The Design and Implementation of the 4.3BSD UNIX Operating System.* Addison-Wesley Series in Computer Science. Addison-Wesley Publishing Company, Inc., Reading, Massachusetts, 1989.

[16] Hui Li, Sudarsan Tandri, Michael Stumm, and Kenneth C. Sevcik. Locality and Loop Scheduling on NUMA Multiprocessors. In *Proceedings of the 1993 International Conference on Parallel Processing*, pages II-140–II-147. CRC Press, Inc., August 1993.

[17] Frank C. H. Lin and Robert M. Keller. The Gradient Model Load Balancing Method. *IEEE Trans. on Software Engineering*, SE-13(1):32–38, January 1987.

[18] David B. Loveman. Program Improvement by Source-to-Source Transformation. *Journal of the Association for Computing Machinery*, 24(1):121–145, January 1977.

[19] Steven Lucco. A Dynamic Scheduling Method for Irregular Parallel Programs. In *Proceedings of the ACM SIGPLAN '92 Conference on Programming Language Design and Implementation*, pages 200–211, San Francisco, CA, June 1992. ACM Press.

[20] Evangelos P. Markatos and Thomas J. LeBlanc. Using Processor Affinity in Loop Scheduling on Shared-Memory Multiprocessors. In *Proceedings of Supercomputing '92*, pages 104–113, Minneapolis, MN, November 16–20 1992. IEEE Computer Society Press.

[21] Nenad Nedeljković and Michael J. Quinn. Data-Parallel Programming on a Network of Heterogeneous Workstations. In *Proc. of the First Int'l Symposium on High-Performance Distribution Computing*, pages 28–36. IEEE Computer Society Press, September 1992.

[22] David M. Nicol and Joel H. Saltz. Dynamic Remapping of Parallel Computations with Varying Resource Demands. *IEEE Transactions on Computers*, 37(9):1073–1087, September 1988.

[23] Hiroshi Nishikawa and Peter Steenkiste. A General Architecture for Load Balancing in a Distributed-Memory Environment. In *Proceedings of the 13th International Conference on Distributed Computing Systems*, pages 47–54, Pittsburgh, PA, May 1993. IEEE, IEEE Computer Society Press.

[24] Hiroshi Nishikawa and Peter Stenkiste. Aroma: Language Support for Distributed Objects. In *Proc. of the Sixth Int'l Parallel Processing Symposium*, pages 686–690. IEEE Computer Society Press, March 1992.

[25] David A. Padua and Michael J. Wolfe. Advanced Compiler Optimizations for Supercomputers. *Communications of the ACM*, 29(12):1184–1201, December 1986.

[26] Constantine D. Polychronopoulos. Toward Auto-scheduling Compilers. *The Journal of Supercomputing*, 2(3):297–330, 1988.

[27] Constantine D. Polychronopoulos and David J. Kuck. Guided Self-Scheduling: A Practical Scheduling Scheme for Parallel Supercomputers. *IEEE Trans. on Computers*, C-36(12):1425–1439, December 1987.

[28] Larry Rudolph, Miriam Slivkin-Allalouf, and Eli Upfal. A Simple Load Balancing Scheme for Task Allocation in Parallel Machines. In *SPAA '91. 3rd Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 237–245, Hilton Head, SC, July 21–24 1991. ACM Press.

[29] Bruce S. Siegell. *Automatic Generation of Parallel Programs with Dynamic Load Balancing for a Network of Workstations*. PhD thesis, Department of Electrical and Computer Engineering, Carnegie Mellon University, May 5 1995.

[30] Bruce S. Siegell and Peter Steenkiste. Automatic Generation of Parallel Programs with Dynamic Load Balancing. In *Proceedings of the Third IEEE Symposium on High Performance Distributed Computing*, pages 166–175, San Francisco, CA, August 2–5 1994. IEEE Computer Society Press.

[31] Bruce S. Siegell and Peter Steenkiste. Controlling application grain size on a network of workstations. In *Supercomputing '95*, page To appear, San Diego, CA, December 1995. IEEE Computer Society Press.

[32] John A. Stankovic. Stability and Distributed Scheduling Algorithms. *IEEE Transactions on Software Engineering*, SE-11(10):1141–?, October 1985.

[33] Pieter Struik. Techniques for Designing Efficient Parallel Programs. In Wouter Joosen and Elie Milgrom, editors, *Parallel Computing: From Theory to Sound Practice*, pages 208–211. IOS Press, 1992.

[34] V. S. Sunderam. PVM: A Framework for Parallel Distributed Computing. *Concurrency - Practice and Experience*, 2(4):315–339, December 1990.

[35] Peiyi Tang and John N. Zigman. Reducing Data Communication Overhead for DOACROSS Loop Nests. In *1994 International Conference on Supercomputing Conference Proceedings*, pages 44–53. ACM SIGARCH, ACM Press, July 1994.

[36] Ten H. Tzen and Lionel M. Ni. Trapezoid Self-Scheduling: A Practical Scheduling Scheme for Parallel Compilers. *IEEE Trans. on Parallel and Distributed Systems*, 4(1):87–98, January 1993.

[37] Yung-Terng Wang and Robert J. T. Morris. Load Sharing in Distributed Systems. *IEEE Trans. on Computers*, C-34(3):204–217, March 1985.

[38] Marc Willebeek-LeMair and Anthony P. Reeves. Dynamic Load Balancing Strategies for Highly Parallel Multicomputer Systems. Technical Report EE-CEG-89-14, Cornell Univ. Computer Engineering Group, December 1989.

[39] Michael E. Wolf and Monica S. Lam. A Loop Transformation Theory and an Algorithm to Maximize Parallelism. *IEEE Transactions on Parallel and Distributed Systems*, 2(4):452–471, October 1991.

[40] Michael Wolfe. Loop Skewing: The Wavefront Method Revisited. *International Journal of Parallel Programming*, 15(4):279–293, August 1986.

[41] Michael Wolfe. Vector Optimization vs. Vectorization. *Journal of Parallel and Distributed Computing*, 5:551–567, 1988.

[42] Michael Wolfe. More Iteration Space Tiling. Technical Report CS/E 89-003, Oregon Graduate Center Department of Computer Science and Engineering, 19600 N. W. von Neumann Drive, Beaverton, OR 97006-1999 USA, 1989.

[43] Michael Wolfe. Massive Parallelism through Program Restructuring. In Joseph JaJa, editor, *The 3rd Symposium on the Frontiers of Massively Parallel Computation*, pages 407–415, University of Maryland, College Park, MD, October 8–10 1990. IEEE Computer Society Press.

[44] I-Chen Wu. *Multilist scheduling: A New Parallel Programming Model*. PhD thesis, School of Computer Science, Carnegie Mellon University, 1993. Also appeared as technical report CMU-CS-93-184.