Automatic Generation of Parallel Programs with Dynamic Load Balancing for a Network of Workstations

Bruce S. Siegell

May 5, 1995

CMU-CS-95-168

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy in Electrical and Computer Engineering

Thesis Committee:

H. T. Kung, Chair
Allan Fisher
Peter Steenkiste
Jaspal Subhlok

Supported in part by the Defense Advanced Research Projects Agency, Information Science and Technology Office, under the title "Research on Parallel Computing," ARPA Order No. 7330. Work furnished in connection with this research is provided under prime contract MDA972-90-C-0035 issued by DARPA/CMO to Carnegie Mellon University and under its subcontract, No. 334918-58792 with Networks Systems Corporation.
The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of Network Systems Corporation, DARPA or the U. S. Government.

Keywords: dynamic load balancing, parallelizing compilers, network of workstations, self-scheduling, grain size, Nectar

Abstract

Because of their high availability and relatively low cost, networks of workstations are now often considered as platforms for applications that used to be relegated to dedicated multiprocessors. Parallelizing compilers have simplified the programming of shared and distributed memory multiprocessors. However, with networks of workstations, which are more loosely coupled, additional problems of heterogeneity, varying resource availability, and higher communication costs must be addressed in order to maximize utilization of system resources. Computational capabilities may vary with time due to other applications competing for resources, so dynamic load balancing is very important.

Our research explores issues in retargeting a parallelizing compiler for a network of workstations. In this dissertation, we describe a system that supports dynamic load balancing of distributed applications consisting of parallelized DOALL and DOACROSS loops. We outline the added compiler functionality needed to generate parallel programs with dynamic load balancing and demonstrate how parameters for dynamic load balancing can be selected and controlled automatically at run time with cooperation between the compiler and runtime system. We have implemented a prototype runtime system on the Nectar system at Carnegie Mellon University and have evaluated its performance using hand-parallelized applications running in various environments.

Key performance parameters under our control include the grain size of the application, the frequency of load balancing, and the amount and frequency of work movement. The optimal grain size is selected based on computation and communication costs of the application on the particular system on which it is run. Selecting an appropriate load balancing frequency requires information about communication costs and process scheduling by the operating system. The frequency must be adjusted as loads on the processors change, and controlling the frequency requires the cooperation of the compiler. Making correct decisions regarding work movement is a difficult problem because of high work movement costs and the unpredictable nature of the loads on the processors. Our measurements show that dynamic load performance improves system utilization and reduces execution times in some cases, but is ineffective for others, largely due to the costs of moving work.

Acknowledgements

I would like to thank my advisor, H. T. Kung, for his advice, support, and encouragement. I am grateful to him for allowing me to follow my interests in both hardware and software within the contexts of the Nectar and iWarp projects. I would also like to thank Peter Steenkiste for advising me after Kung left for Harvard. Peter is a good sounding board and encouraged me to write papers to help organize my ideas. He also read my manuscript several times, identifying problem areas and giving useful suggestions about ways to strengthen its presentation. Thanks also to the other members of my thesis committee—Allan Fisher and Jaspal Subhlok—for their invaluable comments and suggestions regarding my dissertation.

Many other people in the ECE[1] and SCS[2] communities have helped me with my research and helped to make my life enjoyable during my time at CMU. I am indebted to the people who listened to and gave constructive feedback on the practice talks for my defense and to the attendees of the iWarp/Nectar seminars who listened attentively to my other talks over the years. I thank the members of the Nectar project who developed and maintained the Nectar hardware and software. Michael Gillinov deserves special thanks for helping me keep the prototype Nectar system running after everyone else went off to do other things. I also thank the SCS Facilities staff for keeping everything else running. My officemates and the SCS Zephyr community[3] have also helped me with many day-to-day questions and problems. Finally, I'd like to thank my friends and family for being supportive during the ups and downs of my graduate studies.

Portions of the work described here have been published previously [54].

[1] Department of Electrical and Computer Engineering.
[2] School of Computer Science.
[3] The people who use the Zephyr Notification Service.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

There has been a lot of success in developing parallel languages [48, 51, 52, 66] and parallelizing compilers [25, 62, 67, 79] for MIMD distributed memory machines. These tools have simplified the distribution of applications on tightly-coupled machines, such as the Thinking Machines CM-5 [66], the Intel iWarp [7, 62], and the Cray T3D [1, 44]. *Workstation clusters*, in which independent workstations are connected by a high-speed network, are emerging as a new type of loosely-coupled multicomputer. However, the tools for managing the distributed resources on these *network-based multicomputers* are in a primitive state. Many message passing libraries exist for networks of workstations, such as PVM [63], Nectarine [57], and Express [18], but it is not straightforward to port tools such as parallelizing compilers to workstation clusters because of the much higher communication costs and the heterogeneity and variability of the available resources. On workstation clusters, both computation and communication capabilities may vary with time due to other applications competing for resources. High speed networks, such as FDDI [49], Nectar [3], Gigabit Nectar [59], or the more recent ATM networks (e.g., [13]), only partially address the high communication costs because throughput is limited by software overhead for protocol processing [58] and message assembly and disassembly on the sending and receiving hosts. Thus, dynamic load balancing and careful management of communication are essential for efficient parallel execution on workstation clusters. Our research explores these issues in retargeting parallelizing compilers for workstation clusters.

On networks of workstations, 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 [21, 39, 55]. More general load balancing packages must be developed so that a wider range of applications can be run efficiently on a range of systems. Switching between applications and systems should require minimal interaction with the programmer. Ideally, the programmer would only need to specify a small set of parameters for the system so that applications can use available resources efficiently. In this thesis, we show that it is possible for a parallelizing compiler to generate efficient code that can dynamically shift portions of an application's workload between processors to improve performance. By using a parallelizing compiler as our starting point, we can handle many load balancing decisions automatically for a large range of applications. A parallelizing compiler can also restructure programs to increase grain size and, thus, reduce communication overheads. Dynamic load balancing does not always improve application performance. The performance with dynamic load balancing is limited by incomplete knowledge in the load balancer, delay in responding to changes in processor performance, and costs of shifting work between processors. The thesis describes several optimizations which help to address these limiting factors and provides analysis identifying when load balancing can be profitable for certain types of applications.

We have developed a load balancing system for applications consisting of parallelized DOALL and DOACROSS loops [54]. The system involves both the compiler and runtime system in selecting load balancing parameters, with minimal involvement by the programmer. Key performance parameters that can be controlled at run time include the grain size of the application, the frequency of load balancing, and the amount and frequency of work movement.

The rest of this chapter is organized as follows. In Section 1.1, we describe in more detail our assumptions regarding our target system, a cluster of workstations. Then, in Section 1.2, we describe the application domain for our load balancing system and describe how it is parallelized. Section 1.2 also describes notation used in this thesis and presents several example applications. In Section 1.3, we introduce our approach to load balancing. Section 1.4 maps our approach into a control model and identifies areas where control theory provides insight regarding selection of load balancing parameters. Section 1.5 describes how the performance of parallel programs is evaluated in this thesis, and Section 1.6 summarizes our measurements taken on the Nectar system [3]. Section 1.7 discusses related load balancing research. We describe the organization of the remainder of the thesis in Section 1.8.

1.1 Features of target system

The environment targeted by our research is a set of workstations connected by a network. We do not assume that we have a dedicated set of workstations, but rather a possibly heterogeneous set of independent, personal workstations. The competing loads on the processors can not be determined until run time. The processors may be shared with other users, but we assume that there is no other load balanced application running on the system. Much of our analysis assumes that the workstations run operating systems where CPU scheduling is based on a fixed time quantum and the scheduling mechanism derives from a round-robin approach; this is typical of multitasking operating systems, such as Unix [4, 32]. For simplicity, we assume that a message passing library for the target system is provided that hides the underlying topology of the network.

Our research does not address fault tolerance, either for the processors or the network. Both network and processor performance are expected to vary, but we assume that communication is reliable and that all processors are available throughout the computation. We assume that communication costs are high relative to access to local memory (or access to shared memory on a shared memory machine) and, thus, direct much effort to reducing communication overhead. Our system attempts to hide communication latencies by overlapping communication costs with computation, but if the network is near saturation, latencies may be too high to be hidden.

The specific system targeted by our prototype implementation is the Nectar system [3] at Carnegie Mellon University. Nectar consists of a high-speed crossbar network connecting a set of Unix workstations. Details of the Nectar environment will be presented in Chapter 7.

1.2 Application domain and compiler model

Our target application domain is loop-based code operating on array and scalar data units. Numerical code operating on matrices (e.g., LINPACK [16]) often fits this description. The applications are parallelized by distributing the iterations of one or more loops in the loop nest among the processors in the target system; this thesis considers the case where only one loop is distributed. The application code is replicated on all processors, but loop bounds of the distributed loop are modified so that the processors operate on mutually exclusive subsets of the distributed iterations. Aggregate data structures, e.g., arrays, referenced by the iterations may be replicated on all processors or may be distributed in the same manner as the iterations, possibly with some overlap at the boundaries, according to the *owner computes* rule [24, 36]: each processor performs the computations that write to the data locations assigned to it (and vice versa). Figure 1.1 shows the distribution of a matrix by columns. In this example, the problem is partitioned so that each processor computes the same number of outputs. The distribution is called a *block distribution* because each processor is assigned a contiguous block of columns of the matrix.

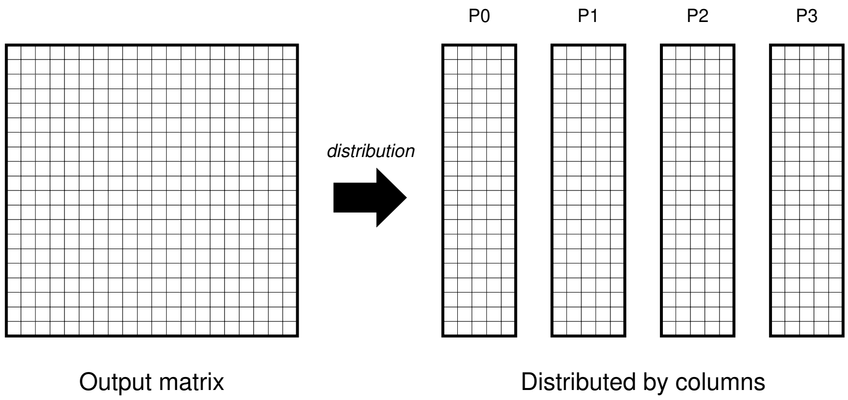


Figure 1.1: Distribution of output matrix by columns.

Two types of distributed loops are distinguished by the literature [15, 42, 43, 45]: DOALL and DOACROSS. DOALL loops have independent iterations (i.e., there is no data dependence between iterations) so their execution order is not constrained; all iterations of the loop may execute in parallel. DOACROSS loops have loop-carried flow dependences, and a partial ordering of the loop iterations must be maintained in executing the loop; parallelism can be obtained by pipelining multiple executions of the distributed loop. Sometimes, e.g., for loops with false output dependences, restructuring of code is necessary to convert loops to a form amenable to parallelization as DOALL or DOACROSS loops. For example, in some cases, a recurrence can be replaced with a parallel reduction operation to remove an output dependence from the loop; several researchers [8, 17, 31, 50] have addressed parallelization of loops with recurrences with certain properties.

There are many tools to aid in distributing DOALL and DOACROSS loops on distributed memory multiprocessors. Parallel languages [48, 51, 52] and parallelizing compilers [25, 62, 67, 79] relieve the programmer of the burden of managing tasks, distributed data structures and communication. With parallel languages, the programmer must think about how the application is to be parallelized and must explicitly specify the decomposition of the problem for parallelism. With parallelizing compilers, ideally, existing sequential code is translated into parallel code without user intervention. However, with existing parallelizing compilers, such as AL [67], Fx [62], or Fortran D [25], the programmer must help the compiler by inserting data alignment and distribution directives into the sequential code. Our research is concerned with changes that must be made to the parallel code produced by the compiler to support load balancing and is applicable whether the input is a parallel language or a sequential language. For simplicity, we assume that the input language has a FORTRAN-like syntax (e.g., Figure 1.3a).

1.2.1 Notation used in this thesis

Because numerical problems are in our target application domain, there will be much discussion of vectors and matrices in describing our example applications. These aggregate structures will be represented in several ways in the text, figures and program examples. In figures, equations, and accompanying text, entire matrices are represented by a bold capital letter, e.g., **A**, vectors are represented by a bold lower-case letter, e.g., **x**, and elements of matrices and vectors are represented by name of the aggregate structure in lower case (not bold) with the indices of the elements given as subscripts, e.g., $a_{i,j}$ or x_i. (See figure 1.2.) Scalar values are represented in italics in either upper or lower case and may have subscripts as well.

$$\mathbf{A} = \begin{bmatrix} a_{1,1} & \cdots & a_{1,n} \\ \vdots & \ddots & \vdots \\ a_{m,1} & \cdots & a_{m,n} \end{bmatrix} \qquad \mathbf{x} = \begin{bmatrix} x_1 \\ \vdots \\ x_m \end{bmatrix}$$

a) matrix notation b) vector notation

Figure 1.2: Matrix and vector notation for figures and equations.

In most cases, program examples will be shown in a Fortran-like language (Figure 1.3a) or Fortran-like pseudocode (Figure 1.3b) for sequential versions of the applications and in C (Figure 1.3c) for parallelized versions because our assumed input language is Fortran-like (e.g., AL [67] or Fortran D [25]) and the language of our parallel implementations is C. Matrices are represented as arrays in all cases, with matrix elements specified by lower case letters followed by individually bracketed subscripts, e.g, *a[i][j]*. The array name for an n-dimensional array followed by a single subscript represents an entire $n-1$-dimensional subarray. For example, for two-dimensional matrix A, with element *(row,col)* referenced as *a[row][col]*, *a[row]* is a pointer to a subarray containing an entire row of A. Note that this is not how arrays are defined in Fortran, although it is consistent with C. In some parallelized versions, the order of indices may be reversed from that of the sequential version; this representation indicates that the matrix has been transposed for reasons of locality or ease of work movement. Thus, *a[row][col]* is an element of a matrix stored in row-major form, and *a[col][row]* is an element of a matrix stored in column-major form.

```
DO i = 1, n
      a(i) = (a(i-1) + a(i+1)) / 2
END DO
```

```
DO i = 1, n
      compute a(i)
END DO
```

```
for (i = locallo; i < localhi; i++) {
      a(i) = (a(i-1) + a(i+1)) / 2;
}
```

a) Fortran-like language　　b) Fortran-like pseudocode　　c) C language

Figure 1.3: Representations for program examples.

1.2.2 DOALL loops

Loops with independent iterations (no flow dependences, output dependences, or anti-dependences) can be parallelized by assigning the iterations to processors in any fashion. To minimize communication requirements, the data on which the iterations operate is distributed in the same fashion, according to the owner computes rule. Also, to reduce communication at run time, input data required by multiple iterations of the distributed loop may be replicated on all processors that reference the data.

The easiest way to parallelize a DOALL loop is to assign blocks of consecutive iterations to processors. On a homogeneous, dedicated system, each processor is assigned the same number of iterations. Managing the parallelism with a block distribution simply requires adjusting loop bounds so that the appropriate subset of iterations is executed on each processor (Figure 1.4b).

In some applications, the DOALL loop is executed multiple times, but with monotonically increasing or decreasing loop bounds. For these applications, the initial equal distribution of work by a block distribution will cause load imbalance as the application executes on a homogeneous, dedicated system. In these cases, the loop iterations are distributed to the processors in a round-robin, or *cyclic*, fashion so that processors have equal workloads throughout the computation. The stride and offset for the loop bounds must be adjusted so that each processor gets the appropriate set of iterations (Figure 1.4c).

Sometimes, block and cyclic distribution methods are combined to create *block-cyclic* distributions.

```
for (i = 0; i < n; i++) {
      compute iteration;
}
```

a) Sequential loop.

```
blocksize = (n + P - 1) / P;
for (i = p * blocksize; i < (p + 1) * blocksize && i < n; i++) {
      compute iteration;
}
```

b) Block distribution

```
for (i = p; i < n; i += P) {
      compute iteration;
}
```

c) Cyclic distribution

Figure 1.4: Distribution of DOALL loop.

Also, for nested DOALL loops, where the distribution is multi-dimensional, the choice of distribution method may be made independently for each DOALL loop. Our presentation will not explicitly address block-cyclic distributions or nested DOALL loops.

1.2.3 DOACROSS loops

If the iterations of a loop to be distributed have flow dependences, e.g., in the inner loop in Figure 1.5a, the iterations can not be run independently. The partial order of execution of the iterations required by the dependences must be maintained. If the data for the loop is distributed and iterations are assigned to processors according to the owner computes rule, a single instance of the loop executes sequentially. However, when the distributed loop is nested inside another loop, parallelism can be obtained by pipelining the execution of the outer loop. The distributed loop retains the order required by its dependences, but portions of different instances of the loop are computed in parallel. When a processor finishes with its portion of the loop it sends its results to the processor handling the next portion of the loop. Then the sender proceeds with its portion of the next instance of the loop.

Again, the simplest distribution of the iterations is a block distribution as shown in Figure 1.5b, although other distributions are possible. The distribution is chosen to minimize communication costs. For short dependence distances, a block distribution is usually the most efficient because it only requires communication between logically adjacent processors at the boundaries of the blocks.

```
DO count = 0, 10
      DO i = 1, n-2
            a(i) = (a(i-1) + a(i+1)) / 2
      END DO
END DO
```

a) Sequential loop

```
for (count = 0; count <= 10; count++) {
      receive_left(a(locallo-1));
      for (i = locallo; i < localhi; i++) {
            a(i) = (a(i-1) + a(i+1)) / 2;
      }
      send_right(a(localhi));
}
```

b) Block distribution

Figure 1.5: Distribution of DOACROSS loop.

1.2.4 Dealing with recurrences

Loops that have output dependences between iterations, i.e., have recurrences, can not be parallelized as DOALL or as DOACROSS loops. However, in some cases, they can be restructured so that the dependence is removed from the loop. Much research has been directed towards recognizing parallelizable recurrences in sequential loops [8, 17, 50]. The basic process is to recognize recurrences (e.g., using pattern recognition) and then to test whether the recurrence operators have the required properties for parallelization [50]. For example, if the recurrence operation is associative, it can be replaced with a parallel reduction operation (Figure 1.6) with time complexity $\frac{n}{P} + \log P$ [8, 17, 50]. The global output variable is replaced by a local private variable on each processor, and each processor computes a portion of the recurrence using the data from the iterations it executes (in $O(\frac{n}{P})$ time if work is distributed to processors equally). Then, when the loop terminates, the partial results from the processors are combined to compute the output (in $O(\log P)$ time if a combining tree is used). When all recurrences have been removed from the loop, the loop can be treated as a DOALL or DOACROSS loop depending on the remaining dependences in the loop. If the recurrence is associative and commutative, a cyclic distribution can be used, but if the recurrence is just associative, a block distribution must be used. To avoid complicated analysis, some compilers (e.g., AL [67]) provide the programmer with ways to specify simple parallel reductions such as addition, multiplication, minimum, and maximum. However, automatic methods make it possible to parallelize many other types of recurrences if an efficient associative operator can be extracted from the source code [17].

1.2.5 Example applications

We discuss load balancing issues using three applications as examples: matrix multiplication (MM), successive overrelaxation (SOR), and LU decomposition (LU). These applications are commonly used

```
r = 0
DO i = 0, n-1
      r = r + x(i)
END DO
```

a) Sequential loop with recurrence

```
/* local computation on each processor.        */
r_local = 0;
for (i = locallo; i < localhi; i++) {
      r_local = r_local + x(i);
}
/* combine the values from all processors.    */
r = reduce_plus(r_local);
```

b) Parallelized loop with reduction

Figure 1.6: Replacement of recurrence with reduction operation.

routines in numerically intensive scientific codes and demonstrate the different types of parallelizable loops described above. In this section, we describe the three example applications and how they are parallelized.

Matrix multiplication (MM)

Our matrix multiplication routine multiplies two $n \times n$ matrices, **A** and **B**, to produce a third $n \times n$ matrix, **C**:

$$\mathbf{C} = \mathbf{A} \times \mathbf{B}$$

Sequential code for matrix multiplication is shown in Figure 1.7. Each element of the **C** matrix is the dot product of a row of the **A** matrix and a column of the **B** matrix. Because each iteration of the j loop computes independent results, the loop can be treated as a DOALL loop. To parallelize the application, we replicate the **A** matrix and distribute the columns of the **B** matrix. The output matrix has the same distribution as the **B** matrix. This parallelization is suggested to the compiler using distribution directives. Using this information, the compiler modifies the loop bounds for the j loop and generates code to distribute the data. Because the loop is a DOALL loop, the compiler is free to distribute the iterations in any fashion and, at run time, for load balancing, the iterations may be redistributed in any fashion as well. In this example, the distributed loop is executed many times, and the distributed input data is reused with each invocation of the loop. All iterations of the distributed loop do the same amount of work each time they are executed.

```
DO i = 1, n
      DO j = 1, n
            c(i)(j) = 0
            DO k = 1, n
                  c(i)(j) = c(i)(j) + a(i)(k) * b(k)(j)
            END DO
      END DO
END DO
```

Figure 1.7: Sequential code for matrix multiplication (MM).

Successive overrelaxation (SOR)

Successive overrelaxation (SOR), also called Simultaneous overrelaxation [47], is an iterative method used to solve Laplace's equation, the partial differential equation

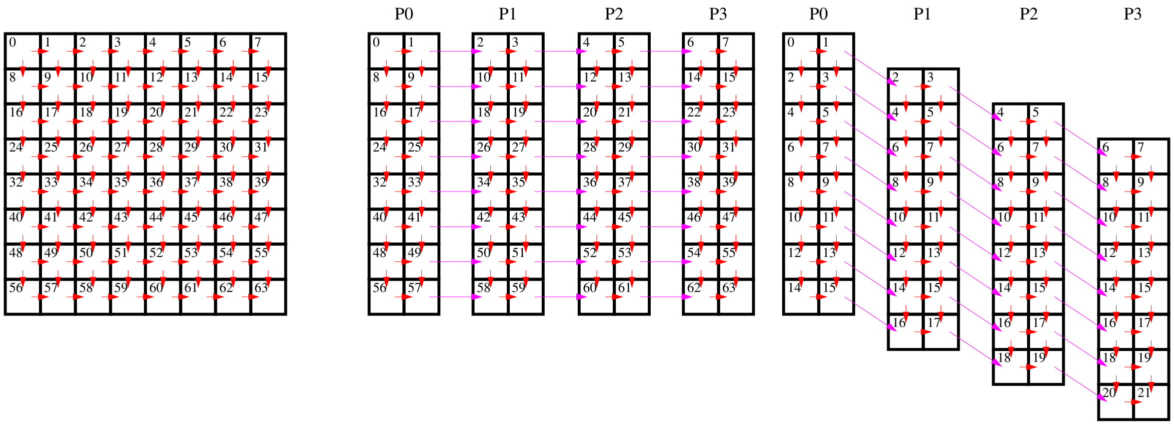
$$\frac{\partial^2 U(x,y)}{\partial^2 x} + \frac{\partial^2 U(x,y)}{\partial^2 y} = 0$$

on a square region with known boundary values. The region is described as an $n \times n$ mesh, represented as a matrix and initialized to an approximation of the solution. For each relaxation phase, each element of the mesh is recomputed as a weighted average of the element and its horizontal and vertical neighbors. The changes between the original and recomputed values are accumulated to compute an error value which is compared to the convergence condition. The order of computation of the elements of the mesh can affect the rate of convergence for the computation [30]. Depending on the order of computation of the mesh points, new mesh values may depend entirely on old mesh values (e.g. red/black SOR [30]), or may depend partially on values computed during the current relaxation phase. We have selected a version (from [67]) where each mesh point is computed as a weighted average of its old value, the new values of its left and upper neighbors, and the old values of its right and lower neighbors (Figure 1.8). The example demonstrates recurrences for the accumulation of the *error* and *norm* values. When the recurrence is replaced by a reduction operation, the loop still has loop-carried dependences, so it is treated as a DOACROSS loop. Figure 1.9a shows the dependences and order of execution for the sequential version of a single relaxation phase for an 8 $\times$ 8 matrix; each row of the matrix is computed from left to right.

In our parallelization of SOR, the input/output matrix is distributed to processors by columns (Figure 1.9b). Parallelism is extracted by pipelining the loop surrounding the distributed loop (Figure 1.9c). At the end of each stage of the pipelined loop, boundary data must be communicated between the adjacent processors. To minimize communication, contiguous blocks of columns are assigned to each processor so

```
iter = 0
error = 1.0
WHILE error < zeta && iter < maxiter
        DO i = 1, n
                DO j = 1, n
                        prev = b(i)(j)
                        b(i)(j) = 0.493 * (b(i)(j-1) + b(i-1)(j) + b(i)(j+1) + b(i+1)(j))
                                    + (-0.972) * prev
                        norm = norm + prev^2
                        error = error + (b(i)(j) - prev)^2
                END DO
        END DO
        error = error /  norm
        iter = iter + 1
END WHILE
```

Figure 1.8: Sequential code for successive overrelaxation (SOR).



(a) Sequential execution (b) Distributed (sequential) execution (c) Pipelined execution

Figure 1.9: Dependences and execution order for single successive overrelaxation (SOR) phase for 8×8 matrix. Arrows indicate dependences between iterations. Numbers indicate execution order.

that communication is only needed at block boundaries.

LU decomposition (LU)

Our third application example is LU decomposition, a type of Gaussian elimination. Gaussian elimination is the first step in solving the equation $\mathbf{A} \times \mathbf{x} = \mathbf{b}$ for $\mathbf{x}$. (The other step is backward substitution.) LU decomposition determines an upper triangular matrix $\mathbf{U}$ and a lower triangular matrix $\mathbf{L}$ such that $\mathbf{L} \times \mathbf{U} = \mathbf{A}$. Then,

$$\mathbf{A} \times \mathbf{x} = (\mathbf{L} \times \mathbf{U}) \times \mathbf{x} = \mathbf{L} \times (\mathbf{U} \times \mathbf{x}) = \mathbf{b}$$

can be replaced with two simpler equations: $\mathbf{L} \times \mathbf{y} = \mathbf{b}$ can be solved for $\mathbf{y}$ by forward substitution; then, $\mathbf{U} \times \mathbf{x} = \mathbf{y}$ can be solved for $\mathbf{x}$ by backward substitution. $\mathbf{L}$ and $\mathbf{U}$ need only be computed once when solving $\mathbf{A} \times \mathbf{x} = \mathbf{b}$ with multiple $\mathbf{b}$ input values. We have selected a version of LU decomposition (Figure 1.10) based on the SGEFA routine in the LINPACK benchmark set [16]. (In our version, the BLAS operations have been inlined and simplified.)

Our LU example is parallelized by distributing the rows of the $\mathbf{A}$ matrix. The "row elimination" loop, the most computation-intensive portion of the application, is split into two loops so that communication for interchanging values is isolated from the actual elimination computation. The resulting elimination loop is distributed as a DOALL loop; because it has no loop-carried dependences the execution order of the iterations is not a concern. However, communication is necessary between executions of the distributed loop, requiring the processors to synchronize. Also, because the loop bounds of the distributed loop and on the loop it contains depend on the indices of outer loops, the number of iterations of the distributed loop and the size of each iteration change with each execution of the loop.

1.3 Our load balancing approach

Load balancing attempts to minimize the execution time of an application by maximizing the utilization of available resources for productive work. When poor utilization is detected, our load balancing system redistributes work by redistributing the distributed aggregate data structures (Figure 1.11); by the owner computes rule, the distributed loop bounds are modified on each of the processors to correspond with the data local to the processor. This approach to load balancing is beneficial because it keeps communication costs for the application low: since data and loop iterations are assigned to processors according to the owner computes rule, most data accessed by the iterations is local; and, in cases where iterations share data (i.e., applications with DOACROSS loops), work movement can be constrained so that the iterations are usually assigned to the same processor. Also, the units of work for load balancing are loop iterations so, by maintaining the original loop structure of the application, the overhead of switching between tasks is kept to a minimum, i.e., just incrementing a loop counter.

Load balancing works as follows. At predetermined points in the parallelized application code, the processors performing the distributed computation—the *slave processors*—assess their recent performance and send the performance information to a central load balancing process on the *master processor* which

```
DO k = 1, n - 1
    /* find the pivot index.                                    */
    pidx = k
    dmax = abs(a(k)(k))
    DO i = k+1, n
        IF abs(a(i)(k)) > dmax THEN
            pidx = i
            dmax = abs(a(i)(k))
        END IF
    END DO
    ipvt(k) = pidx
    /* zero pivot implies this column already triangularized.   */
    IF dmax == 0.0 THEN
        info = k
    ELSE
        /* put the pivot in the pivoting position.              */
        w = a(pidx)(k)
        a(pidx)(k) = a(k)(k)
        a(k)(k) = w
        /* compute multipliers.                                 */
        t = -1.0 / w
        DO i = k+1, n
            a(i)(k) = t * a(i)(k)
        END DO
        /* row elimination with column indexing.                */
        DO j = k+1, n
            t = a(pidx)(j)
            a(pidx)(j) = a(k)(j)
            a(k)(j) = t
            DO i = k+1, n
                a(i)(j) = a(i)(j) + t * a(i)(k)
            END DO
        END DO
    END IF
END DO
ipvt(n) = n
IF a(n)(n) == 0.0 THEN
    info = n
END IF
```

Figure 1.10: Sequential code for LU decomposition (LU).

decides how to redistribute work. Performance assessment is based on measurements of rates of computation for recently computed work. The central load balancer computes a new distribution where work is allocated to the slaves in proportion to their relative capabilities and computes instructions for the slaves which specify the movement necessary to attain the new distribution. Slaves then shift work among themselves according to the instructions. The cost of interactions between the slaves and the load balancer are removed from the critical path of the application by pipelining the interactions to decouple sending of performance information from receiving of instructions. Several additional optimizations are performed at run time to

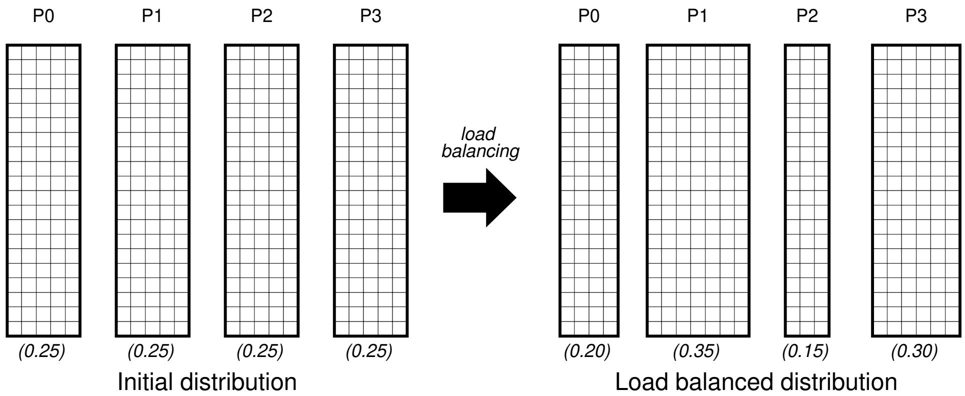


Figure 1.11: Redistribution of output matrix to balance load. Distribution must be changed when relative execution rates of processors (indicated in parentheses) change.

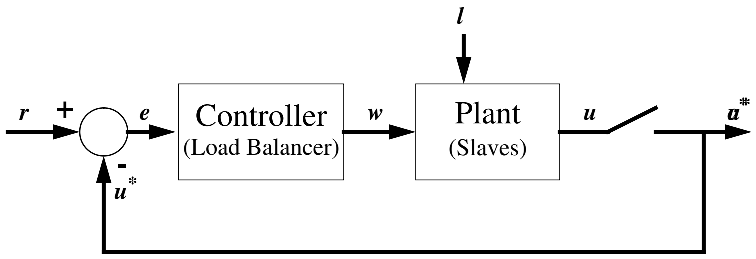
ensure that work movement results in reduced execution time.

The compiler is responsible for setting up the parallelized application code to facilitate load balancing and for supplying information needed for load balancing to the runtime system. The compiler and runtime system cooperate in several aspects of the load balancing decision-making process:

- The compiler notifies the load balancer regarding dependences in the loop nest so that the load balancer can restrict work movement to minimize communication.
- For applications with DOACROSS loops, the compiler and run-time system cooperate in selecting an appropriate grain size.
- The compiler places conditional calls to load balancing code at appropriate locations in the parallelized code so that the runtime system can control the frequency of load balancing.

1.4 Control theory model of dynamic load balancing

The load balancing system can be looked at as a digital feedback control system which uses the difference between the percent utilization of available resources and 1.0 as its *actuating* or *error* signal. A simplified block diagram of the system is shown in Figure 1.12. The central load balancer is the *controller* for the system, and the slaves are the *controlled system* or the *plant*. The central load balancer manipulates the work assignment, **w**, to control the utilization, **u**, of the slave processors. The *disturbances* in the system are the competing loads, **l**, on the slaves. The frequency of load balancing is the *sampling rate* for the system.



r	reference input (1.0)
e	actuating signal (1.0 - utilization of slaves)
w	manipulated variable (work assignment)
l	disturbance (competing load on slaves)
u	controlled variable (utilization of slaves)
u^*	sampled controlled variable (utilization of slaves)

Figure 1.12: Simplified model of load balancer as a digital feedback control system.

Figure 1.13 shows the control system in more detail. The total time for the computation performed by the slaves during one cycle of the control loop is the maximum time taken by any of the slaves. The utilization of each slave, u_i, is computed by dividing the computation time for the slave by the maximum time; thus, the slave (or slaves) with the maximum computation time is 100% utilized ($u_i = 1.0$) and the other slaves are underutilized ($u_i < 1.0$). The *error* that is input to the controller is e_i, the fraction of each slave that is underutilized ($1.0 - u_i$). The controller shifts work from more utilized processors to less utilized processors until all processors are fully utilized, i.e., load is balanced.

If we can create a simple control model for our system, we might be able to derive optimal parameters for the system (for some definition of optimality). Using control system techniques, we might also be able to accurately characterize the system. We might be able to develop proofs regarding the performance of the system, e.g., identifying the range of frequencies over which the system works well, how quickly the system responds to changes in performance, and how quickly the work distribution converges to the desired result.

However, several factors make modeling and analysis of our system difficult. The system is a multiple input multiple output (MIMO) system and can not be uncoupled into simpler systems. MIMO systems do not have unique solutions, and trial and error approaches must be used in their design due to the extra degrees of freedom. In addition, the system is non-linear, making finding exact solutions to the differential equations for the system unlikely; most methods for solution of nonlinear systems involve engineering approximations [20].

Several other factors complicate the modeling of our system. In our load balancing system, the sampling

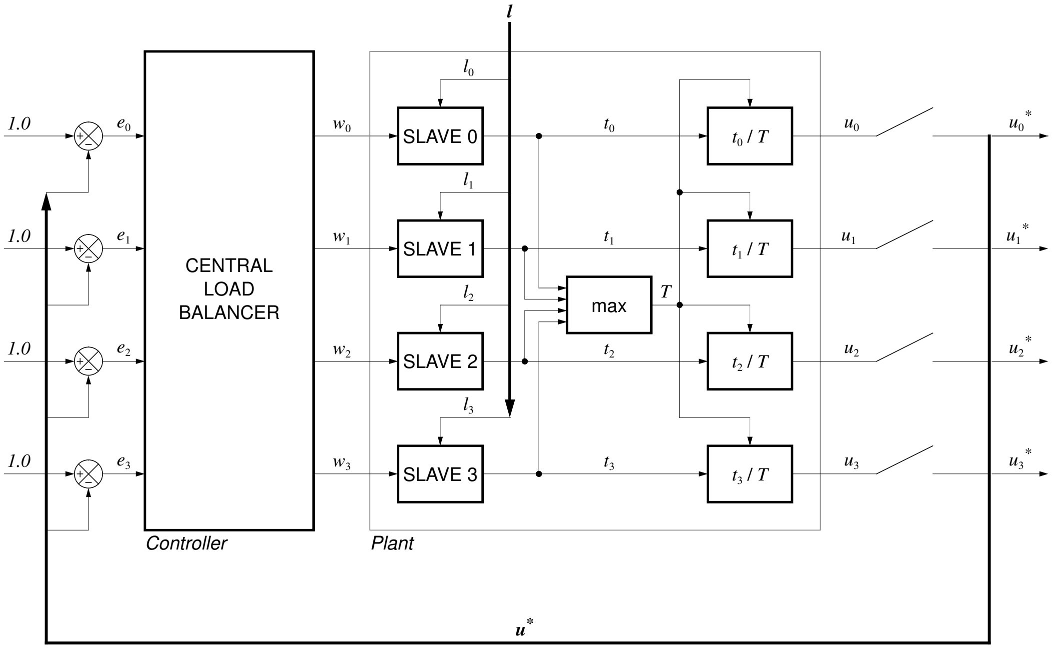


Figure 1.13: Dynamic load balancing control system.

period is based on amounts of work computed rather than on fixed time intervals. Thus, the differential equations describing the system should use derivatives with respect to the amount of work rather than with respect to time. The sampled data is the average performance over the most recent sampling period rather than a snapshot at the sampling time; this adds delay to the system and may have other ramifications. Also, in our actual load balancing system, the computation rates, rather than the utilizations, are used to compute the new work distribution so that quantities of work to move can be more easily computed. This makes the picture of the model much more complicated and makes identification of the components of the control system more difficult.

The modeling of our system is further complicated by the costs of sampling the data and of moving work, which are not included in the figures. Work movement costs have a large impact on performance of applications with load balancing. We have added several optimizations to our system to limit work movement costs. These optimizations should also be included in the system model.

Control theory does provide some insight into several aspects of the problem. The sampling rate of a control system determines which frequencies of change the system can identify and which frequencies may be aliased. The sampling theorem requires the sampling rate to be at least twice the frequency of the signal that we wish to track. The sampling rate also limits how quickly the system can respond to disturbances. If a control system's response time is too long, its inputs and outputs will be out of phase, and even greater deviations from the desired output (processor utilization in our case) can result [6]. Often, to reduce the response time, digital control systems sample at several times the frequencies that are of interest [20]; digital filters can be used to eliminate the undesired higher frequencies. In our system, performance is sampled as frequently as possible to reduce the response time, but the frequency is bounded by other factors, such as the sampling cost.

Another "benefit" of high sampling frequencies is that they make the response to changes smoother and reduce the magnitude of the control steps [20]. However, in our system, we wish to minimize work movement costs. Because the fixed, per message costs of sending work between processors are high, we wish to *decrease* the smoothness of the response and move work in fewer, *larger* messages/steps. These goals are considered in the selection of the sampling period for our system. Also, the sampled data is an average of the performance over the sampling period rather than the performance at the sampling times; the averaging results in implicit filtering that can reduce the amount of work moved by the system. Additional filtering is also used in our system to attenuate high-frequency disturbances and to minimize the impact of error. To further reduce the number of work movement messages, hysteresis has also been added to the system. Both filtering and hysteresis can cause the work assignment to lag behind changes in performance [56], increasing the response time of the system. The filters and hysteresis must be designed to minimize the added lag.

Although many aspects of the system can be modeled by control theory, because of the nonlinearity and complexity of the control system, we investigate components of the system separately in this thesis, and in some cases present only intuitive arguments and/or empirical results to describe the impact of the components on system performance.

1.5 Evaluating parallel performance

The goal of parallelization and of load balancing is to reduce the execution time—the *elapsed time*, rather than CPU time—of the application. We use several additional criteria to evaluate the performance of different versions of the parallelized application to get an idea of how close the observed performance is to the best possible performance given the available computing resources. *Speedup* relative to the sequential version of the application is often used to evaluate performance on different numbers of processors, but is most useful when the processors are homogeneous and are running no competing applications. To evaluate performance in dynamic and/or heterogeneous environments, we have designed an *efficiency* measure that measures how well a parallelized application is using available resources. In this section we'll describe these measures in detail.

Elapsed time measurements

The elapsed time ($t_{elapsed}$) of an application is used for comparing different versions of the application and in computing speedup and efficiency values. Since dynamic load balancing only addresses the computation portion of the application, the elapsed time measurements used in evaluating load balancing do not include times for initializing tasks, generating and distributing input data, or unloading output data, which are common to all parallelized versions of the application. These omitted times, while not a negligible portion of the parallel execution time, are determined by the distribution of the application and are only slightly affected by the addition of dynamic load balancing.

Speedup

The goal of parallelization is to reduce the computation time relative to the sequential time of the application. Thus, to evaluate parallelization, we use measures that compare the parallel execution time to the sequential execution time. The *speedup* of a parallel version is the elapsed time for the sequential version divided by the elapsed time for the parallel version:

$$speedup = \frac{t_{sequential}}{t_{elapsed}} \qquad (1.1)$$

Again, $t_{elapsed}$ only includes the computation portion of the application. For the experiments presented in this thesis, $t_{sequential}$ is determined by measuring the execution time of an efficient single-processor implementation of the same algorithm as the parallel version; however, the sequential version is *not* the parallel version run on one processor.

The speedup of an application run is often compared to the number of processors involved in the parallel computation to get an idea of how well the processing resources are being used. We count only the slave processors when making this comparison. For a dedicated homogeneous environment with each processor providing the same performance as that used for the sequential version, when graphing speedup vs. processors, a linear speedup with slope 1 is generally accepted as the best speedup parallelization can produce, not counting memory effects. The slope of the speedup curve is often used as an efficiency measure for parallel programs on dedicated homogeneous systems [36, 42]:

$$\textit{efficiency} = \frac{speedup}{P} = \frac{t_{sequential}}{P \times t_{elapsed}} \tag{1.2}$$

where P is the number of processors.

In a heterogeneous or dynamic environment, speedup and the efficiency measure based on it do not include enough information about the system to determine how well resources are being used. For example, if work is distributed equally to a heterogeneous set of processors, the best possible speedup curve will have a slope of 1 when calculated using the sequential time measured on the slowest processor, but will have a smaller slope when the sequential time is measured on a faster processor.

Parallelization efficiency

A deficiency of the efficiency measure based on speedup (Equation 1.2) is that it assumes that the parallel application can use all of the computing resources of the processors. This is not the case if the processors are shared with other users. When the processors are shared, Equation 1.2 produces an efficiency value that is too low because it does not account for the resources used by the competing applications. A better measure of efficiency would take into account only the resources actually available to the application:

$$\textit{efficiency} = \frac{c_{productive}}{c_{available}} \tag{1.3}$$

where $c_{productive}$ is the amount of computational resources (i.e., CPU resources) required to execute the application, and $c_{available}$ is the amount of computational resources that were available to the application during its execution.

In a homogeneous environment, computation times may be used in computing the measures of productive and available resources. The productive computation time, $t_{productive}$, is the time required to execute the application on a single, dedicated processor, i.e., the sequential execution time. The computation time available to the application, $t_{available}$, is more difficult to determine accurately. However, it can be estimated as the total of the elapsed times on all processors minus time spent on competing applications during the execution of the application. The elapsed time, $t_{elapsed}$, is the same on all processors. Thus,

$$\textit{efficiency}_{homo} = \frac{t_{productive}}{t_{available}} = \frac{t_{sequential}}{P \times t_{elapsed} - \sum_{i=1}^{P} compete_i} \tag{1.4}$$

In a heterogeneous environment, computing the efficiency is more complicated because times must be scaled by the relative processing capabilities of the different processors. For example, the processing capability of each processor might be the maximum computations per second (*cps*) measured in computing some benchmark program:

$$\textit{efficiency}_{hetero} = \frac{c_{productive}}{c_{available}} = \frac{t_{sequential,s} \times cps_s}{\sum_{i=1}^{p}(t_{elapsed} \times cps_i) - \sum_{i=1}^{p}(compete_i \times cps_i)} \tag{1.5}$$

where the additional subscripts on the time variables indicate the processor on which the measurement was taken. Selecting a universal measure of processing capabilities for different processors is difficult, especially if the architectures of the processors in the system vary greatly (e.g., some are RISC and some are CISC). The system on which we performed our measurements was homogeneous so we use Equation 1.4 to compute efficiencies. For our experiments on the Nectar system [3], competing processes were spawned by the parallelized application code, and their CPU usage was measured using the *getrusage* function [12] provided with Unix.

In the presence of competing loads, the measure of $t_{available}$ used in Equation 1.4 may be inaccurate because resources that are available to the application but not used by the application may be used productively by the competing loads. Thus, in some cases, Equation 1.4 may give a high estimate of the efficiency. Equation 1.4 can be treated conservatively as an upper bound on the efficiency, or, with sufficient knowledge about the interactions in the system, can be treated as an approximation of the efficiency. For the measurements and analysis in this thesis, competing loads are only added on one of the processors, and the competing loads use at most half of the resources of the processor. Thus, the error in $t_{available}$ should be small, and, since the competing load uses less of the total computation time as more processors are added to the system, the error decreases as the number of processors is increased. Therefore, the efficiency indicated by Equation 1.4 should closely approximate the actual efficiency. Also, measurements of the CPU usage of the artificial competing tasks used in our experiments indicate that, in most cases, the competing tasks do not consume more resources than expected given the artificially generated loads so the competing tasks are not consuming significant amounts of resources that should be included in $t_{available}$. The efficiency measure based on speedup, Equation 1.2, uses the most conservative measure of $t_{available}$, assuming that all computing resources are available to the parallelized application, so Equation 1.2 is a lower bound on the actual efficiency. For reference, efficiency values produced by Equation 1.2 are included along with the efficiency values produced by Equation 1.4 when presenting our data. For a dedicated homogeneous system, Equations 1.2 and 1.4 produce the same results because no time is spent on competing processes.

1.6 Summary of experimental results

To demonstrate the feasibility of our approach, we implemented a load balancing run-time system and two example applications on the Nectar system [3]. We measured performance in several controlled environments. In a dedicated homogeneous environment, we demonstrated that dynamic load balancing decisions do not add much overhead to the execution of the application. In an environment with a constant load added to one of the processors, we demonstrated that the load balancing system redistributes load correctly and improves application performance relative to the parallelized application running without load balancing. We added oscillating loads of varying frequency to one of the processors to give an indication of the performance of the system in more dynamic environments. Dynamic load balancing improved performance for slowly changing loads for applications with small work movement costs. In other cases, the performance of the load balanced applications in the dynamic environments was limited by the reaction time of the system and the costs of work movement. We created a model of the system's performance with an oscillating load to show the limits of the approach. In some cases the measured performance was better than that predicted by the model due to optimizations to prevent excessive work movement, included in the system, but not in the model. Our experiments will be explained in detail in Chapter 7.

1.7 Related work

General taxonomies for load balancing can be found in [10] and [71]. We focus on dynamic load balancing for distributed loops.

1.7.1 Compiler support for load balancing

Existing parallelizing compilers often assume a dedicated, homogeneous environment, and distribute work equally to all processors. Many compilers [11, 24, 67] support cyclic distribution of iterations so that when loop bounds vary, as in the LU decomposition example, each processor still gets approximately the same amount of work. For heterogeneous and dynamic environments, however, equal distribution of work does not balance the load. Some languages, such as Fortran-D [24], allow irregular distributions, which could be used for static load balancing in a heterogeneous environment if the characteristics of the environment are known when the program is written. Fortran-D [24] and Vienna Fortran [11] also include directives for redistribution so that data can be rearranged to balance load and reduce communication requirements as data access patterns change. However, these optimizations are performed at compile time according to annotations by the programmer and do not address load imbalances due to a dynamic processing environment.

Express [18], a message passing library and application toolkit that can be targeted by a parallelizing compiler, supports dynamic load balancing by providing routines that automatically distribute data according to specified weights for the processors. For static load balancing, the weight for each processor is a "figure of merit" provided by the user. For dynamic load balancing, Express includes a function that automatically determines weights for the processors; however, the user or compiler must explicitly place the code for recalibrating weights and redistributing data.

1.7.2 Self-scheduling

Many of the approaches for dynamic scheduling of iterations of distributed loops are *task queue* models, in which work units are kept in a logically central queue and are distributed to slave processors when the slave processors have (nearly) finished their previously allocated work. In these models, both control and work are centralized, and the measure of performance is task completion. Knowledge about the interactions between work units is often lost due to the desire to have a single list of tasks (e.g., [28, 46]), and most of the approaches assume that iterations are independent, requiring no communication, and target a homogeneous, shared memory target architecture.

The different task queue approaches differ mainly in the granularity of work movement. In *self-scheduling* [64], work is allocated to processors a single iteration at a time; this approach has high overhead due to the interaction between the processors and the queue for each iteration. *Chunk scheduling* addresses the overhead problem by allocating work a fixed number of iterations at a time, at the risk of increasing the skew in the finishing times of the processors [45]. *Guided Self Scheduling* (GSS) [46] attempts to minimize scheduling overhead and minimize the skew in execution times by allocating a fixed fraction of the remaining work to a processor when the processor requests more work; this reduces the size of the work allocation unit as the execution progresses. GSS still has the potential for execution time skew if too much work is allocated to processors early in the computation so that the remaining smaller chunks do not constitute enough work to smooth over the finishing times of the initial chunks [28]. *Factoring* [28] takes the number of processors into account as well as the amount of remaining work; it schedules a fixed fraction of the remaining work in batches of P equal-sized chunks (where P is the number of processors) and uses probabilistic analysis is used to select the optimal number of iterations per batch. *Trapezoid Self-Scheduling* (TSS) [69] is a simpler approach which linearly decreases the chunk size at run time; although and because GSS is more elaborate, TSS gets better speedups, due to its flexibility in selecting chunk sizes and its lower scheduling overhead. *Tapering* [36] is another variation on GSS that handles tasks with varying execution times. Tapering selects chunk sizes based on the mean and variance of the task execution times so that the inefficiency of execution has high probability of staying within a specified bound. All of these approaches were originally designed for shared-memory architectures.

Recent research [33, 37, 38] 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* [37, 38], data is moved to the local cache when first accessed, and the scheduling algorithm assigns iterations in blocks. In *Locality-based Dynamic Scheduling* [33], 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.

The Tapering approach [36] has also been implemented on a distributed memory machine. Because scheduling overhead is higher in a distributed memory environment, the data is initially distributed according to some data decomposition, and tasks are initially assigned to processors according to the owner computes rule; the data decomposition is refined as information is gained about the work distribution. (This is similar to our approach.) Communication locality is preserved by maintaining a minimum chunk size. For load balancing, the processors are logically connected as a binary tree, with each processor serving as a leaf node and some processors also serving as internal nodes; information about progress on the different processors is passed up through the tree, and instructions regarding redistribution of tasks are broadcast to all processors.

A hybrid approach that selects from among several load balancing algorithms is proposed by [41] for distributed memory machines. It uses a distributed version of Factoring [28] for independent, homogeneous tasks that works in a way similar to the distributed version of tapering.

1.7.3 Diffusion methods

Numerous other approaches have been proposed for scheduling loop iterations, especially if the iterations are independent. 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 [72]. Work movement may be initiated by the sender (*Sender Initiated Diffusion*) or by the receiver (*Receiver Initiated Diffusion*) [72]. The *Gradient Model* method [34] 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.

1.7.4 Use of prior performance as estimate of future performance

The approaches described so far use either workload or progress to determine how to allocate more work or redistribute workload. An alternative, used by our approach, is to use rates of computation of previous work to describe the performance capabilities of different processors [40, 41].

For the implementation of Dataparallel C on a network of workstations [40], loop iterations are mapped to virtual processors, and virtual processors are shifted between processors to balance load. As in our approach, relative computation rates are assessed periodically, and work is redistributed to processors in proportion to their rates. However, Dataparallel C requires the programmer to handle the program partitioning and communication explicitly; this makes pipelined execution of loops complicated to implement. Also, the virtual processor abstraction may add run-time overhead, and all processors communicate for load balancing so load balancing communication is in the critical path for the computation.

1.8 Organization of this dissertation

The remainder of this dissertation is organized as follows. In Chapter 2, we discuss the features of our application domain and run-time environment that have an impact on load balancing and describe the architecture of our load balancing system. Chapter 3 describes the effects of grain size on parallel performance and describes ways to control the grain size of an application. Chapter 4 describes selection of an appropriate load balancing frequency to minimize load balancing overhead and maximize the effectiveness of the load balancer. Chapter 5 presents the details of the load balancing decision making process. Chapter 6 describes the changes that must be made to a parallelizing compiler to support dynamic load balancing. In Chapter 7, we present performance results for an implementation of our load balancing system on the Nectar system [3]. Chapter 7 also includes a model of the performance of the load balancing system in an environment with an oscillating load on one of the processors. We conclude in Chapter 8.

Chapter 2

Load balancing architecture

This chapter presents the architecture of our load balancing system and the motivations for the design choices we made. We begin with discussion of the features of the application and execution environment that must be considered when designing a load balancing system. Then we describe major decisions that must be made to design a load balancing system. This is followed by a description of the high-level choices we made when designing our load balancing system. Load balancing decisions that can take advantage of information about the specific application being executed or the dynamic characteristics of the environment are delayed until compile time or run time, and are discussed in later chapters.

2.1 Application considerations

Several application features impose constraints on the design of an efficient load balancing system. Table 2.1 summarizes the presence or absence of several features affecting load balancing for the three applications described in the previous chapter. The goal of dynamic load balancing is to minimize the elapsed time of an application in a multiprocessor system with varying performance characteristics. This goal is attained by moving work to match the performance characteristics of the processors. This section discusses how the application features listed in Table 2.1 affect the load balancer's ability to attain its goal.

Property of distributed loop	MM	SOR	LU
repeated execution of loop	yes	yes	yes
loop-carried dependences	no	yes	no
dependences outside loop	no	yes	yes
index-dependent loop bounds	no	no	yes
data-dependent loop bounds	no	no	no
index-dependent iteration size	no	no	yes
data-dependent iteration size	no	no	no

Table 2.1: Application properties.

2.1.1 Position of distributed loop in loop structure

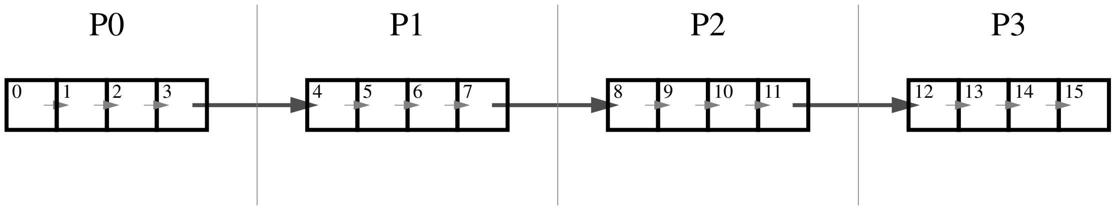
The position of distributed loops within the overall loop structure of the application determines the impact of work movement. If a distributed loop is an inner loop, then the repeated execution of the loop may result in reuse of locally stored data. Moving the distributed data (or data slices) referenced by a loop iteration to a different processor may actually move the corresponding iterations from all instances of the loop due to the owner computes rule. This allows the costs of moving work to be amortized over a longer period. However, as computation progresses, fewer loop instances remain over which to amortize the costs of moving the data and to gain benefits, so it can be more beneficial to move work earlier in the execution of the program than later. When the distributed loop is the outermost loop, the ratio of work movement to data movement is no more than one-to-one, and reuse of distributed data elements does not occur.

The position of the distributed loop in the loop nesting also affects which loop iterations can be moved to balance load. If a distributed loop is an outermost loop, then load balancing must be done during the execution of the loop, and the only loop iterations that can be moved to balance processing times are those that have not yet been executed. However, if the loop is an inner loop, load balancing may be done either during execution of the loop or between executions, and any of the loop iterations can be moved to balance future processing times.

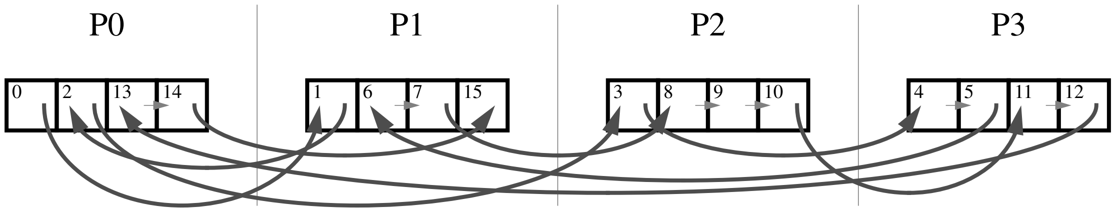
2.1.2 Loop carried dependences

Distributed loops with flow dependences between iterations (DOACROSS loops) are parallelized by pipelining the execution of an outer loop. Communication is needed to handle the dependences that cross processor boundaries. This communication adds overhead to program execution and synchronizes the processors at the ends of the pipeline stages. To minimize the execution time of the application, load balancing must balance the computation times of all the processors between the synchronization points. If the synchronizations in the pipelined application are infrequent, fluctuations in performance on the processors can average out so that the total skew between processing times is small, but with frequent synchronizations, skew times will accumulate.

Also, when there are loop-carried dependences in the distributed loop, the mapping of iterations to processors affects the amount of communication required by the parallelized application. For example, with a block distribution (Figure 2.1a), only the iterations near the boundaries of the blocks need to communicate with iterations on other processors. However, with other mappings (e.g., Figure 2.1b), all iterations may need to communicate with other processors, increasing the communication costs and making pipelined execution of the application unmanageable. When a load balancer makes work movement decisions, it should use the dependence information and attempt to maintain mappings that minimize communication.



(a) Block distribution.



(b) Random mapping.

Figure 2.1: Communication requirements for different mappings of loop iterations. Arrows indicate dependences and communication.

2.1.3 Dependences outside loop

Dependences involving distributed array elements may also exist outside the distributed loop, e.g., if there are assignment statements involving distributed array elements. Again, dependences crossing processor boundaries require communication. Work movement due to load balancing can complicate the handling of this communication in a distributed memory environment. With a regular distribution, such as a block or cyclic distribution, processors owning distributed values on the right hand side of an assignment statement can compute the owner of the destination value based on the array indices on the left hand side of the statement. In this case, only compile-time information is needed to determine the sending and receiving processors for the assignment statement and the data transfer can be done in one communication step. However, if the distribution changes at run time due to dynamic load balancing, each processor must use run-time information to determine its involvement in the communication, i.e., whether it is involved in the communication and whether it is the sender or receiver; and several communication steps may be required to implement the assignment statement. To handle dynamic distributions, a compiler must generate communication code to identify processors involved in moving data, as well as the communication code for the data movement itself.

2.1.4 Loop bounds

The number of iterations in a loop may be large or small and may change with the indices of outer loops or with other data values. The bounds of the distributed loop and the loops enclosing it affect load balancing in several ways.

For load balancing, we consider iterations of distributed loops to be atomic execution units. Thus, load can only be balanced to within one iteration. When the number of iterations in a distributed loop is small, one iteration can be a significant portion of the total execution time for the loop, and it may be difficult to control the skew in the execution times between processors. For matrix problems, the distributed loops often have iteration counts as large as the problem size, the size of the matrices. Since problem sizes must be large for parallelism to be practical, the loops that are load balanced will usually have enough iterations that the skew problem described above will not occur. If the number of iterations of the loops surrounding the distributed loop is small—i.e., the distributed loop is executed a small number of times—then work moved by moving data may be small, as in the case when the distributed loop is the outermost loop (Section 2.1.1).

We call distributed data for which there are no future references *inactive*. When the loop bounds of a distributed loop are not fixed, the load balancer must be careful not to move inactive data. Whether the bounds of the distributed loop are *index-dependent* (vary with indices of surrounding loops, as in the LU decomposition example) or *data-dependent* (depend on other data, e.g., WHILE loops), the load balancing system must keep track of which distributed data is active. Also, the distribution of work should not be such that data always becomes inactive on the same processor because work would have to be moved to that processor again and again to balance the load adding unnecessary overhead. Using a cyclic distribution addresses this problem.

Another concern is that if the number of iterations of a distributed loop or a surrounding loop is data-dependent, it may be necessary to pass around global information so that all processors handle the loop exit conditions properly. For example, the outer loop of the SOR example is a WHILE loop, and processors must communicate to determine the termination condition; this is implicit in the reduction operation.

2.1.5 Iteration sizes

The size of distributed loop iterations may be fixed, may be dependent on loop indices, or may be data-dependent. If load balancing incorrectly assumes equal sized work units when distributing work, variations in iteration sizes may prevent the load balancer from correctly balancing the load.

When there is a detectable trend in the iteration size, as when the size depends on loop indices, the load balancer may be able to correctly adjust its assumptions regarding equal-sized work units. In the LU decomposition example, the amount of work associated with the iterations of the distributed loop decreases each time the loop is invoked. The load balancer can still correctly handle this case because, for each invocation of the distributed loop, all of the iterations require the same amount of computation, and proportional allocation of iterations still works. However, as the computation progresses, the ratio of the cost of invoking the load balancer to the cost of executing a loop iteration increases; to compensate, the frequency of load balancing should be reduced as the size of work units decreases.

The amount of work associated with iterations of a distributed loop may also vary if the loop contains conditional code. In general, it will not be possible to predict the cost of different iterations and load balancing mechanisms that rely heavily on predicting the cost of future work are unlikely to do well. This difficulty also exists with static distributions on dedicated systems. However, if the number of iterations is large and the iteration size has a random distribution, variations may average out so that the load balancer's predictions are correct.

2.1.6 Data size

For efficient execution of a program with load balancing, the communication costs of the application and the communication and computing costs of the load balancer must be kept to a minimum. Maximum benefits of work movement are attained when the time spent moving data is small compared to the time spent computing the work associated with the data. The size of data slices and the amount of computation associated with the slices is determined by the distribution.

2.2 Environmental considerations

Features of the run-time environment determine the need for load balancing, and affect load balancing costs and the accuracy with which a load balancer can balance load. For example, dynamic load balancing is necessary if the available processing capabilities of the system or the processing requirements of the application vary with time; otherwise, efficient use of resources can be achieved with a static work distribution. This section discusses how the hardware configuration, communication costs, and dynamic properties of the environment impact load balancing decisions.

2.2.1 Hardware configuration

Several features of the hardware configuration affect the load balancer design and load balancing decisions, including the types of processors, the number of processors, and the topology of the interconnection network.

Types of processors. On a dedicated homogeneous system, work can be distributed equally to all processors. However, if the system is heterogeneous, then the work must be distributed in proportion to the relative capabilities of the processors. A measure of performance is needed for comparing the processors. Different application requirements regarding CPU time, memory, and I/O make use of measures such as MFLOPS or SPECmarks impractical. Task queue approaches work around this difficulty by not directly comparing processor performance; instead, completion of assigned work is used as the measure of performance. Our approach uses rate of completion of work as the relative measure.

Number of processors. If load balancing is based on global information, the cost of collecting the information increases with number of processors; interacting with the load balancer could become a bottleneck for the computation, especially if load balancing is in the critical path. Also, as the number of processors increases, work movement decisions can become more complicated, and more messages may be required to move work between the increased number of destinations and sources. However, with more processors, if data is distributed among the processors, less data may need to be shifted to balance the load because each processor will have a smaller share of the total distributed data.

Network topology. The way processors are connected in a distributed memory system affects communication costs. The latency and bandwidth of the physical connections is a strong indicator of communication costs, but the number of physical connections a message must traverse multiplies these factors. Communication costs increase with the number of intermediate processors, which is determined by the topology of the system. If the system is not fully connected, collection of global information for load balancing can become very costly. To avoid this problem, diffusion methods (Section 1.7.3) are often used for locally connected topologies such as arrays or hypercubes.

2.2.2 Communication costs

Several hardware factors in the cost of communication have already been mentioned. It is also necessary to select appropriate communication protocols so that unnecessary hand-shaking and data copies are avoided. The best way to reduce communication costs, however, is to reduce the amount of communication.

To track performance changes as closely as possible, load balancing should be done as frequently as possible. However, there is overhead associated with collecting the information to make the load balancing decisions, and there is overhead associated with moving work, mostly due to communication. The frequency of load balancing should be selected to keep the cost of interactions with the load balancer to an acceptable level and to limit the opportunities for work movement between processors so that the costs of movement do not exceed the benefits. Also, each decision regarding work movement should consider the costs of shifting the work between the processors.

2.2.3 Dynamicness of system

Dynamic load balancing is only necessary if either system performance or application requirements vary with time. Otherwise, static load balancing will suffice to balance load. From the point of view of the environment, the available performance only varies if there are competing loads on the system.

The effect of competing loads on resources available to the application depends largely on the scheduling granularity used by the operating system—the *time quantum*. The scheduling interacts with the synchronizations in the application and can interfere with performance measurements used for load balancing. For example, if application performance on a processor is measured over too short a period, the application may appear to be getting 100% of the CPU time because its use of the processor is uninterrupted; or it may appear to be getting a small fraction of the CPU time because control of the CPU is passed to other processes during the measurement period. However, if performance is evaluated over a longer period of time, then these cases will average out, giving a better view of the actual load on the system. If load balancing is based on measured performance, the frequency of measurement should be selected so that scheduling effects are averaged out.

2.3 Load balancing design space

This section describes major load balancing design choices that must be made based on the application and environment features described in the previous sections. Several researchers [71, 10] have presented more general taxonomies for load balancing, but here we emphasize choices applicable to our application domain where units of work are iterations of distributed loops.

Global vs. local information. The first decision that must be made is whether the entity making load balancing decisions uses global or local information. In most cases, use of global information implies a centralized controller which combines information from all slave processors, and use of local information implies distributed control. (Exceptions are possible, such as the Gradient Model [34] where global information is encoded in a distributed gradient surface.) Use of global information allows the system to respond quickly and accurately to changes in performance [70]. However, collection of information from all processors can be expensive, and a central load balancer may become a bottleneck. *Task queue* methods generally use global information; and *diffusion* methods use local information. For other approaches, such as hierarchical algorithms [72], the dichotomy is less clear.

Work location. Data for unfinished work may be stored in a central location, or it may be distributed among the processors. If the data is stored in a central location, as in many task queue approaches, the same data may need to be shifted back and forth between the central location and the processors many times. In the shared memory systems for which the task queue approaches were designed, the cost of moving data between the memory and the processors is low, but with distributed memory systems the cost can be much higher. However, if the data is kept distributed among the processors, assignment of work to processors according to the owner computes rule can improve locality, reducing communication costs.

Use of application knowledge. Another important decision is the degree to which application-specific information is used to control the load balancing. If the load balancing system takes advantage of information about reuse of data by the application and sharing of data by different tasks, it can place tasks on processors so that communication costs are minimized. For example, this distinguishes Affinity Scheduling [37, 38] and Locality-based Dynamic Scheduling [33] from earlier task queue approaches. In most other task queue approaches (e.g., [28]), loops are unrolled to create a single list of tasks, so this application-specific information is lost.

Load balancing strategy. Load balancers generally attempt to minimize execution time by maximizing productive utilization of processors. This can be done either by giving processors more work when they become idle or by attempting to predict how much work each processor can handle and distributing the work to the processors in advance. Thus, for dynamic load balancing, the load balancer can be invoked whenever a processor finishes its work, or the load balancer can be invoked periodically while the processors are computing to redistribute work based on assessments of processor capabilities. As discussed in Section 2.2.1, the latter case requires a performance measure that allows the load balancer to determine the relative processing capabilities of the slaves.

Control mechanism. The load balancing entities—the load balancer(s) and the computation processes—must interact to collect performance information, to distribute control information, and to move work. Decisions must be made regarding the conditions that trigger these interactions and the selection of the frequency of evaluating these conditions (e.g., to minimize costs or to minimize response time).

Some of these decisions may involve low-level details of the environment. For example, the decisions involved in collecting performance information include choosing when each computation process will evaluate its performance and how much information is necessary to trigger the load balancing computations. Distribution of control information involves deciding whether instructions will be sent and whether the computation processes will block waiting for instructions. Movement of work involves deciding when and how to follow received instructions; if instructions are inexact or are based on outdated information, the slave processors may modify or ignore received instructions.

Granularity of work movement. Granularity of work movement, i.e., the minimum and maximum amounts of work that may be moved between processors during a load balancing phase, has strong effects on the overhead added by load balancing. The granularity may be implied by the threshold used to decide whether a system needs to be rebalanced. Thus it can affect how sensitive load balancing is to fluctuations in loads in a dynamic system. Work movement granularity is the feature that distinguishes many of the task queue algorithms.

2.4 Load balancing architecture

The entities in our load balancing system are a central load balancer (the *master*) and the computation processors (the *slaves*). The computation processors run the application code, periodically send performance information to the load balancer, and follow instructions sent by the load balancer. The load balancer combines information from the slaves to generate instructions for work movement to balance load.

Both the master and the slaves alternate between a *computation phase* and a *load balancing phase*. The two phases compose a *load balancing cycle*. Slave code is similar to that generated by existing parallelizing compilers (e.g., AL [67]), except that load balancing code is added to collect performance information, send information to the load balancer, receive instructions, and move work. The master code imitates the structure of the slave code to the extent necessary to test loop termination conditions. Calls to the central load balancing code are inserted into the master code at points matching the insertion points in the slave code so that the load balancer is called the appropriate number of times. Details regarding code insertion are presented in Chapter 6.

2.4.1 Global information

Load balancing is based on *global information* since it allows the load balancer to respond to fluctuations in system performance more quickly than a load balancer based on local information [70]. The global information is collected by a *central load balancing process* which can communicate directly with each of the computation processes. The central load balancer can respond quickly to performance changes because it can instruct overloaded processors to move load directly to processors with surplus processing resources in a single step. The load balancing process, the *master*, periodically interacts with the computation processors, the *slaves*, but the frequency of interaction is controlled (Section 4.3) so that the central load balancing does not become a bottleneck. Also, load balancing can be taken out of the critical path of the application by overlapping the load balancing costs with the useful computation (Section 2.6). If necessary, the "central" load balancer could be distributed with minimal effect on the slaves' view of the system.

2.4.2 Work distributed among slave processors

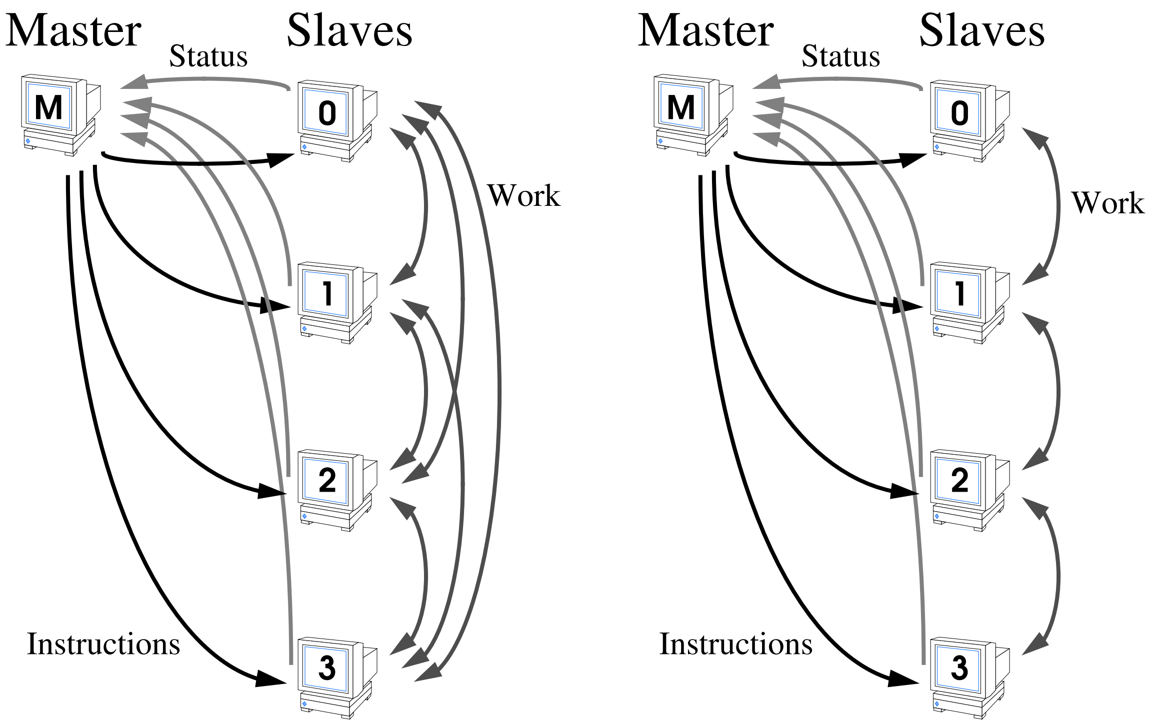
Because the target is a distributed memory system, the cost of moving work and its corresponding data back and forth from a central location to the slaves would make dynamic load balancing unprofitable. Thus, the *work is distributed among the slave processors*, and load balancing is done by shifting work directly between the slaves (Figure 2.2a). In some cases, work movement is constrained by characteristics of the application (Figure 2.2b).

The work units in our application domain are iterations of distributed loops. A set of distributed array elements is associated with each loop iteration. By the owner computes rule, each processor stores the distributed data elements referenced by the loop iterations assigned to it.

2.4.3 Use of application knowledge

In our approach, the loop structure of the sequential code is retained to take advantage of data reuse and data dependences and to minimize administrative costs. Loop bounds of the distributed loop are modified so that each processor computes its assigned subset of the iterations, and calls to load balancing code are inserted at appropriate locations. *Application information is preserved* implicitly in the loop structure.

Preserving application information in this manner pays off in a number of ways. First, data locality is maximized since iterations that operate on the same data, e.g., iterations of a loop that is executed



(a) Unrestricted (b) Restricted by dependences

Figure 2.2: Communication for load balancing. (The master is the central load balancer.)

multiple times, will be executed on the same processor. Second, knowledge about the loop structure and data dependences makes it possible to reduce communication since work movement can be restricted to minimize the number of data dependences that span processor boundaries (Figure 2.1). Also, for loops that are parallelized by pipelining, the synchronizations added by the restructuring caused by unrestricted work movement could reduce parallel execution so restricted work movement is more than just a communication optimization. (When there are no restrictions due to dependences, an alternate, unrestricted approach is used that attempts to minimize work movement costs.) Finally, we exploit the fact that tasks consist of loop iterations to minimize the cost of bookkeeping and task switching. Specifically, managing a task queue on a processor requires keeping track of a range of loop indices (i.e., two values), and task switching consists of incrementing a loop index. There is no real context switch since the entire context is captured in the loop structure and is automatically in place when proceeding from one iteration to the next.

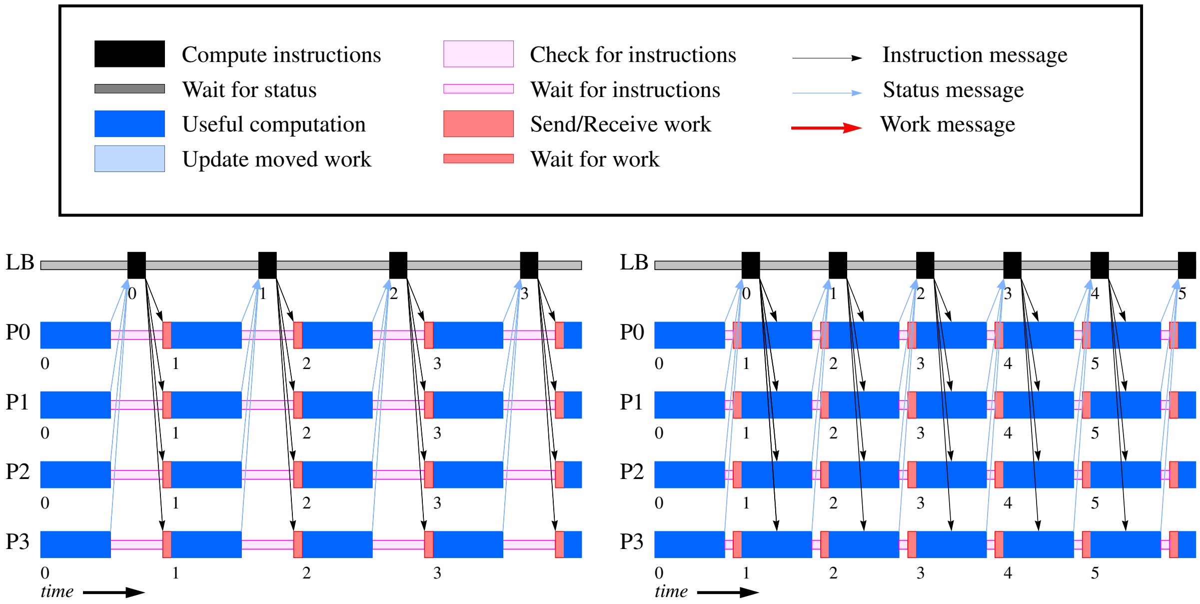
2.5 Load balancing strategy

In our load balancing system, the slave processors periodically exchange information with the load balancer at predetermined points in the application code. At these points, the slaves send information about their performance since the last information exchange and receive instructions on how to redistribute work. Slave performance is specified in work units executed per second, where the work units are iterations of the distributed loop; this *computation rate* is recomputed at each load balancing point. This provides the load balancer with a measure that implicitly takes into account both the relative static capabilities of the processors and the dynamic effects of competing loads on processor performance. However, the measure assumes temporal locality in the load on the slaves.

Using the rate 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. The load balancer then compares the new work distribution to the current work distribution and computes instructions for redistributing the work. For applications with loop-carried dependences, the instructions only move work between logically adjacent slaves so intermediate processors may be involved in a shifting of load (Figure 2.2b); this restriction minimizes the communication created by the loop-carried dependences. For applications without such restrictions, work may be moved directly between the source slave and the destination slave (Figure 2.2a). Each slave receives instructions specifying the slaves with which it must exchange work and the number of iterations it should execute before exchanging information with the load balancer again. After exchanging work, the slaves continue computing their assigned iterations until the next information exchange.

2.6 Master-slave interactions

It is important to minimize the cost of interactions between the load balancer and the slaves, since this overhead is incurred even if the system is well balanced. The simplest mechanism for the interactions between the load balancer and slaves is a *synchronous* mechanism in which all slaves send performance information, *status*, to the load balancer at each predetermined load balancing point and block waiting for instructions based on that information (Figure 2.3a). If necessary, work is moved upon receipt of the instructions (Figure 2.4a). This mechanism responds immediately to measured changes in performance but puts the load balancing costs in the critical path for the application. Moreover, the load balancing adds a barrier synchronization to the application if no synchronization was present in the original parallel code. The load balancing cost can be reduced by reducing the frequency of load balancing or by removing load balancing costs from the critical path. Load balancing costs may be hidden by doing the communication and computation for the load balancing in parallel with the computation required by the application. This can be accomplished by pipelining the load balancing interactions or by removing load balancing synchronizations.



(a) Synchronous load balancing (b) Pipelined load balancing

Figure 2.3: Interactions for load balancing in a stable balanced system. With pipelined load balancing, more useful computation is done in the same amount of time.

2.6.1 Pipelined Load Balancing

With *synchronous load balancing*, at each load balancing point, the slaves send performance information and wait for instructions based on that information. With *pipelined load balancing*, slaves still wait for instructions at each load balancing point, but the instructions received by the slaves are based on performance information sent a fixed number of load balancing cycles earlier, rather than on the performance information from the most recent computation phase. Work is still moved upon receipt of instructions. The *pipeline depth* is the number of load balancing cycles that the receipt of instructions follows the sending of the performance information on which the instructions are based.

Pipelining's advantage is that it removes load balancing latencies—transferring status from the slaves, computing instructions, and transferring instructions to the slaves—from the critical path. If load balancing is infrequent enough, a single load balancing cycle is greater than these latencies so a pipeline of depth 1 is sufficient to hide them (Figure 2.3b). The load balancing frequency should be low enough that fluctuations in the performance of the load balancing processor or in the latencies of the network are hidden as well. For example, without pipelining, a single competing load on the master processor can cause added delays of up to one time quantum (the period of processor allocation used by the operating system's scheduler) or more each time the load balancer is called. With pipelining and an appropriate load balancing frequency, the delays can be completely hidden.

The main disadvantage of pipelining is that it delays the effects of load balancing instructions: load remains unbalanced for an extra load balancing phase (Figure 2.4b), and loads on the slaves could change again before the instructions take effect. The delay is minimized by making the pipeline depth as small as possible, i.e., 1, and keeping the load balancing frequency as high as possible. (Frequency tradeoffs will be discussed in Section 4.3.)

An additional disadvantage of pipelining is that it requires the load balancer to keep track of more state. In synchronous load balancing, the amount of work assigned to each processor can be sent to the load balancer along with its rate information. This is not the case for pipelined load balancing because there are pending instructions on the slave, and the slave does not yet know its work assignment. Therefore, the load balancer must keep track of the work distribution valid at the time instructions take effect, based on the instructions already sent. Also, if there are limited communication resources for work movement, e.g., limited fan-in on receiving processors due to a limited number of ports, the load balancer must keep track of resources assigned to pending instructions so that new instructions do not interfere.

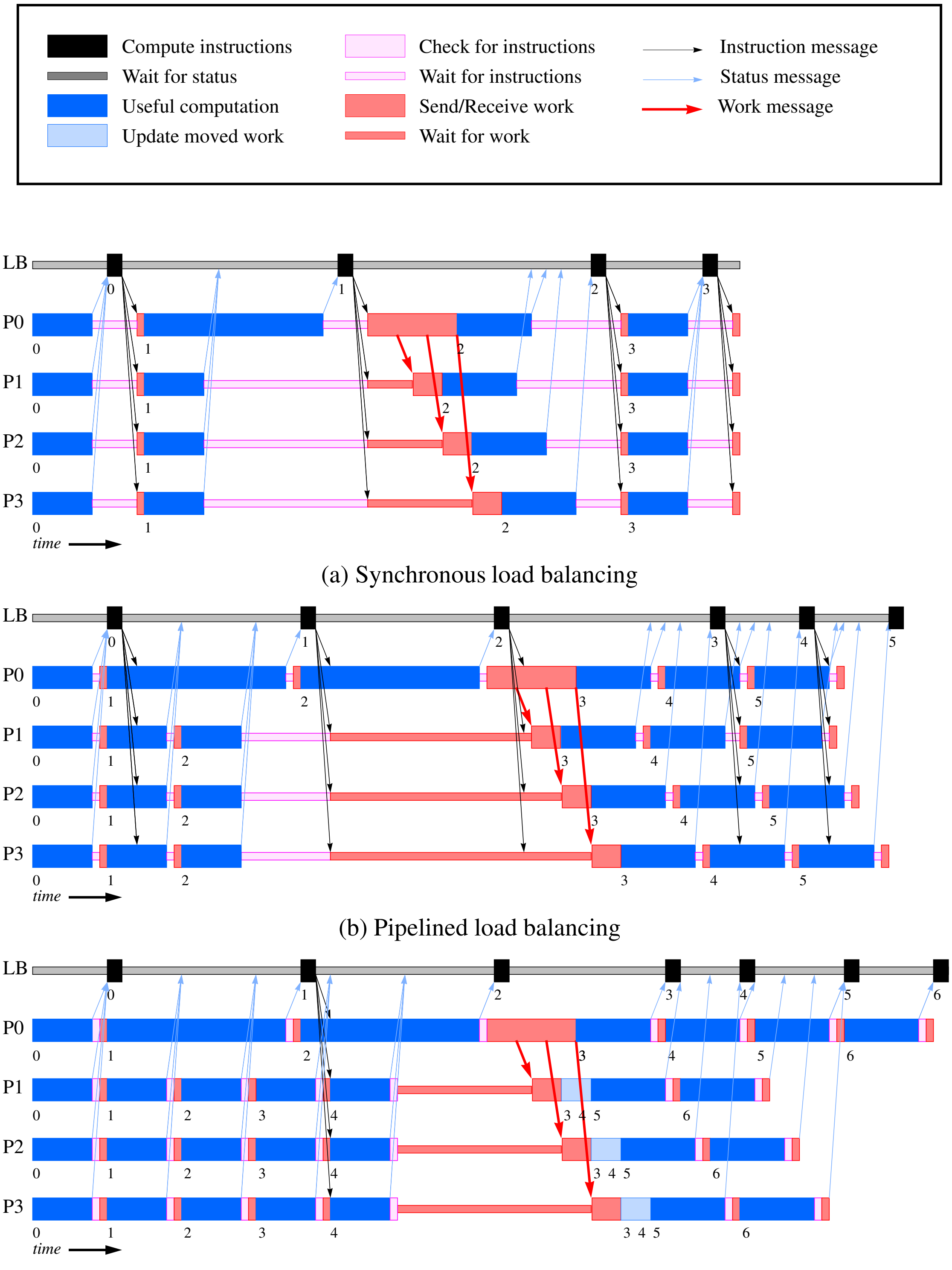
Pipelining is most beneficial in a static environment because once work has been distributed appropriately to balance the load, the response time of the load balancer is irrelevant. This was confirmed experimentally in a dedicated homogeneous environment; in that environment, pipelined load balancing produced higher efficiencies than load balancing without pipelining. In dynamic environments, if the load balancing frequency is too low, pipelining can be detrimental because of the delayed response to changes in performance. However, for the load balancing frequencies selected by our system (described in Chapter 4), pipelining did not hurt performance relative to load balancing without pipelining, although pipelining did not improve the performance either. Since a static or slowly changing environment is a common case—in effect, made to appear even more common by optimizations used in our system to avoid excessive work movement—pipelining is preferable over synchronous load balancing, in spite of pipelining's added implementation complexity.

2.6.2 Asynchronous load balancing

In the synchronous and pipelined mechanisms described above, load balancing synchronizes the slaves at the point where they receive instructions from the load balancer. Most of these synchronizations can be removed if the load balancer only sends instructions to the slaves when work needs to be moved. In this *asynchronous load balancing* mechanism, load balancing only causes the slaves to synchronize when they must shift work (Figure 2.4c). Like pipelined load balancing, asynchronous load balancing hides load balancing latencies, but asynchronous load balancing can be more efficient than synchronous approaches (pipelined or not) because fewer messages must be sent and processors can continue doing useful work until instructions to move work arrive. However, also like pipelining, asynchronous load balancing delays the reaction to changes in performance, but asynchronous load balancing has greater complexity. Also, for applications in which synchronizations occur frequently, independent of load balancing, there is little opportunity for processors to keep working if they complete their work sooner than other processors, so asynchronous load balancing can provide little additional benefit over pipelined load balancing in this case. Thus, asynchronous load balancing is applicable to fewer applications—i.e., only applications requiring no or infrequent synchronizations—than pipelined load balancing.

In addition to needing more state in the load balancer (like the pipelined case), asynchronous load balancing requires more state to be sent with moved work. The sender of work may not have proceeded as far into the computation as the receiver, so the data sent may not be in a state consistent with the data on the receiver. To manage inconsistent data, slaves must either keep track of the state of each distributed data slice or update data upon receipt so that it is in a state consistent with the data already resident. By the time work is received, data needed to update the received data may have been modified by later computation phases so updating received data may sometimes require storage of state information from earlier computations.

For asynchronous load balancing, redistribution of work in proportion to processing rates prevents processors from falling further behind other processors, but does not necessarily eliminate the lag that



(a) Synchronous load balancing

(b) Pipelined load balancing

(c) Asynchronous load balancing

Figure 2.4: Interactions for load balancing in a system where available computation resources decrease on one processor (P0).

already exists. The existing lag could be corrected by overcompensating when moving work, but this will eventually cause the processors that were ahead to lag behind. Also, if the faster processors continue to work ahead, the sending processor might be delayed waiting for the faster processors to be ready to receive the work movement messages, especially if the work movement messages are too large to be buffered. To avoid these problems, slaves block when instructed to receive work, as in Figure 2.4c. (Note that, even though processors P1, P2, and P3 stop computing after they receive instructions to receive work, they are still ahead of processor P0 when the work movement is completed.)

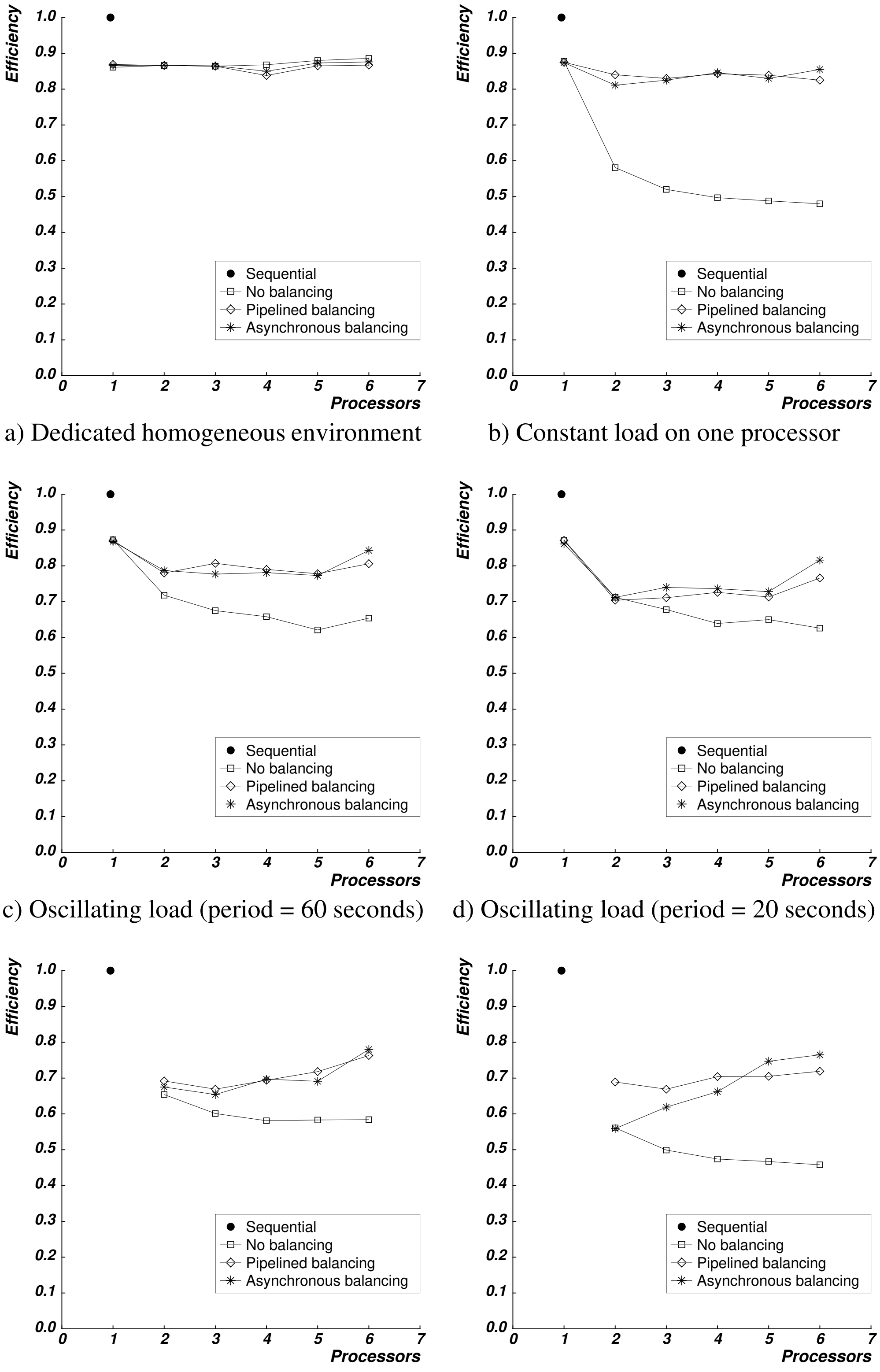
The applicability of asynchronous load balancing is limited by the synchronizations already present in the parallelized code. A processor can only work ahead of other processors until a synchronization point is reached; then it must wait for the other processors to catch up. Thus, for applications that synchronize frequently, such as SOR and LU decomposition, asynchronous load balancing is not practical.

Figure 2.5 shows performance measurements for pipelined load balancing and asynchronous load balancing for the matrix multiplication application executed in several environments—both static and dynamic—on the Nectar system.[1] Asynchronous load balancing is more efficient than pipelined load balancing in some cases, and less efficient in others. (Note that Equation 1.4 is used to compute efficiencies for the graphs. However, the ordering of the *no balancing*, *pipelined balancing*, and *asynchronous balancing* efficiency values is the same if Equation 1.2 is used.) Since asynchronous load balancing does not provide significant performance improvement and is only applicable for applications with infrequent synchronizations, its added complexity is not worth the additional effort. Therefore, our system uses pipelined load balancing.

2.6.3 Granularity of work movement

The granularity of work movement in our prototype system is determined by the distribution of the data. The unit of work in the system is a single loop iteration, but the unit of work movement is the work associated with an entire slice of the distributed data. Shifting entire data slices is advantageous in that future computation is balanced and future references to the slice remain local, but when slices are large, work movement can become very expensive. The load balancer (Chapter 5) includes several optimizations to ensure that work movement will be profitable.

[1] The load balancing parameters for the data presented are as follows: load balancing target period is 1 second; 10% predicted improvement is required for work movement; rate information is filtered using a state machine; cost-benefit analysis is enabled. The meanings of these parameters will be described later in the thesis.



a) Dedicated homogeneous environment

b) Constant load on one processor

c) Oscillating load (period = 60 seconds)

d) Oscillating load (period = 20 seconds)

e) Oscillating load (period = 6 seconds)

f) Oscillating load (period = 2 seconds)

Figure 2.5: Pipelined vs. asynchronous load balancing for 500 × 500 MM in different environments.[1]

2.7 Summary

In this chapter, we discussed application and environment features that affect load balancing decisions and described the major decisions that must be made in designing a dynamic load balancing system. Then we described the high-level features of our proposed load balancing system for automatically parallelized code running on a network of workstations:

- The tasks to be load balanced are distributed loop iterations, and the units of work movement are slices of the distributed data structures.
- The work and data are distributed on the processors to avoid the high cost of access to a centralized task queue.
- The sequential loop structure of the application is preserved to maximize data reuse, to minimize communication, and to minimize the overhead of switching between tasks.
- For rapid response to changes in performance, decisions are made by a central load balancer using global information.
- Work is allocated in proportion to measured processing rates.
- Periodic, pipelined interactions between the slave processors and the load balancer occur at preselected points in the application code so that all processors will be in a consistent state when load balancing occurs.

Experiments were conducted to compare the performance of pipelined load balancing with synchronous (unpipelined) and asynchronous load balancing. We showed that both the pipelined and asynchronous approaches hide the costs of interactions between the slaves and the load balancer. Also, in some cases, asynchronous load balancing allows fast processors to work ahead of slower (overloaded) processors. We found performance with pipelining to be as good as (in dynamic systems) or better than (in stable systems) performance with synchronous load balancing. Asynchronous load balancing had no clear performance benefits over pipelined load balancing. Because asynchronous load balancing is more complicated to implement and is applicable to fewer applications, we chose not to investigate it further. Thus, we selected the pipelined approach.

Chapter 3

Automatic selection of grain size

We define the *grain size* of a parallel application to be the amount of computation between the synchronizations required by the application. Selection of an appropriate grain size is a prerequisite for efficient execution of a parallel application, whether load balanced or not. Generally, a large grain size is considered desirable to minimize communication overhead [61, 65], but in some cases, parallelism is reduced by increasing the grain size, possibly increasing the execution time. Also, in the presence of competing loads, the grain size can interact with the scheduling of processes by the operating system, affecting the performance of the application in some cases. This chapter discusses the tradeoffs involved in the selection and control of the grain size of parallelized applications.

The synchronizations in a parallelized application result from its communication requirements, which are determined by the distribution of the loop iterations and the dependences and loop structure of the original sequential code. Often, the computation between synchronizations, i.e., the grain size, can be increased by increasing the problem size, but that is not a practical option when solving a problem of a particular size. In some cases, however, the loop structure of the parallelized code can be modified to change and control the grain size. The communication patterns in the parallelized code determine how easily the grain size can be controlled and how the grain size interacts with operating system scheduling. We distinguish different types of synchronizations based on the different communication patterns, and we distinguish parallelized applications by their most frequent synchronizations.

In the next section, we identify the types of synchronizations that may be present in a parallelized application. Then we describe loop restructuring transformations that can be used to control grain size.

The remainder of the chapter describes how an appropriate grain size is determined and, when possible, controlled for each type of synchronization. Analysis and simulation is used to predict performance with different grain sizes, using simplified models to estimate computation and communication costs.

In this chapter, performance with different grain sizes is compared using the efficiency measure defined by Equation 1.4. For the measurements and much of the analysis in this chapter, a dedicated environment is assumed (i.e., Equation 1.4 reduces to Equation 1.2) so there is no difficulty in determining the available computation time accurately. For the remainder of the analysis, we control the system model, so the available computation time can be accurately determined.

3.1 Synchronization types

We classify applications based on their most frequent synchronizations. The simplest case is applications that require no synchronizations, such as the matrix multiplication example.[1] For these applications, there is no interaction between the processors so grain size is not an issue. 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 synchronizations* between some or all of the processors.

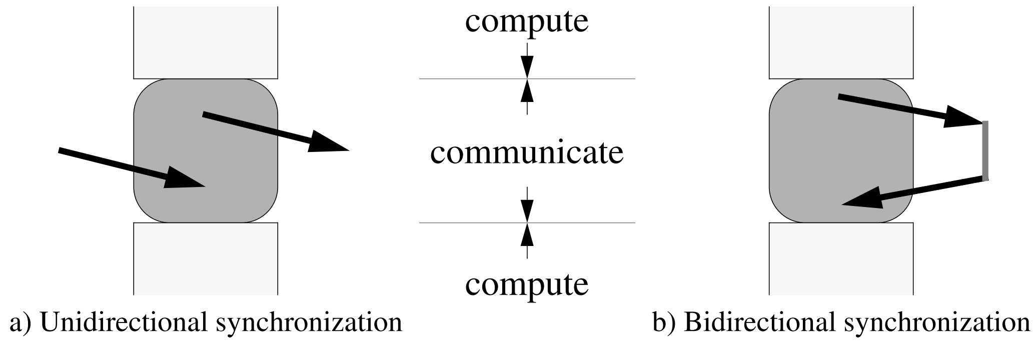


Figure 3.1: Communication pattern observed on one of the slave processors. The communication pattern determines the synchronization type.

We model applications as alternating between computation and communication/synchronization phases. In the case of unidirectional synchronizations (Figure 3.1a), i.e., pipelined applications, data is communicated

[1]Applications that can execute in parallel without requiring communication are sometimes called *embarrassingly parallel* applications because they are so easy to parallelize.

in only one direction through the processors during a communication phase of the application. The communication enforces a partial ordering on the computation: processors earlier in the pipeline must generate data before the later processors can proceed, but processors early in the pipeline can work ahead of processors later in the pipeline as much as buffering of intermediate data between the processors will allow. Often, this allows computation to occur in parallel with the communication. We take advantage of the flexibility of the partial ordering to control the grain size.

In the bidirectional case, a communication phase does not end until a message based on data sent during the same phase is received (Figure 3.1b). This does not permit much overlap between computation and communication for the processors involved in the communication. When all processors are involved in the synchronization, i.e., none of the processors can exit the synchronization point until all processors have reached it, the bidirectional synchronization is a *barrier synchronization*. Barrier synchronizations impose a total ordering on the computation phases, making control of grain size difficult. An example of a bidirectional/barrier synchronization is the *global combination* step [67] of a parallel reduction operation because each processor must contribute its portion of the result and then must receive the combined result. With dynamic load balancing, even a single assignment statement involving distributed data elements results in a barrier synchronization because global communication is needed to identify the processors owning the source and destination elements. (This will be described in detail in Section 6.6.) Bidirectional synchronizations involving only a subset of the processors are not likely to occur in automatically parallelized data parallel code. Therefore, in our analysis, we treat all bidirectional synchronizations as barrier synchronizations.

3.2 Compile-time control of grain size

Loop restructuring transformations can be used to increase parallelism [29, 43, 74], to increase data locality [29, 53, 73, 74, 77], and to reduce communication overhead [61, 65]. Here we limit our discussion to transformations used to control grain size and communication overhead, especially those that can be parameterized for control at run time. Grain size is increased by restructuring loops so that communication is moved out of inner loops. We do not discuss transformations such as loop interchange [2, 25, 43, 76, 78] and loop skewing [75, 78] because they are difficult to parameterize and are not always applicable. (E.g., loop interchange can be parameterized by conditionally selecting different copies of the loop nest [76] but does not provide a continuum of grain size choices.)

3.2.1 Loop splitting

Loop splitting [35] is a transformation that splits the loop body into two or more pieces, each in its own copy of the loop bounds. Loop splitting can move communication from a loop into a separate loop so that grain size is increased. It is useful for parallelization because it can be used to remove communication from the loop to be parallelized. For example, in the LU decomposition routine (Figure 1.10), the grain size is increased by splitting the "row elimination" loop (Figure 3.2a) into two loops (Figure 3.2b): in the first loop, the body references multiple columns of the matrix to be distributed so the loop requires communication when distributed; the body of the second loop (the BLAS SAXPY operation) accesses only one row of the matrix, so no communication or synchronization is required when the second loop is distributed. The inverse transformation of loop splitting is called *loop fusion* [35, 43]; for our purposes, loop fusion is usually not necessary because we usually wish to maximize the grain size, not decrease it.

```
DO j = k+1, n
        t = a(pidx)(j)
        a(pidx)(j) = a(k)(j)
        a(k)(j) = t
        DO i = k+1, n
                a(i)(j) = a(i)(j) + t * a(i)(k)
        END DO
END DO
```

a) Original loop

```
DO j = k+1, n
        t = a(pidx)(j)
        a(pidx)(j) = a(k)(j)
        a(k)(j) = t
END DO

DO j = k+1, n
        DO i = k+1, n
                a(i)(j) = a(i)(j) + t * a(i)(k)
        END DO
END DO
```

b) After loop splitting

```
t(k+1:n) = a(pidx)(k+1:n)
a(pidx)(k+1:n) = a(k)(k+1:n)
a(k)(k+1:n) = t(k+1:n)

DO j = k+1, n
        DO i = k+1, n
                a(i)(j) = a(i)(j) + t(j) * a(i)(k)
        END DO
END DO
```

c) Preparation for message aggregation

Figure 3.2: LU decomposition row elimination loop.

3.2.2 Message aggregation

Once communication has been grouped together by loop splitting, messages with common destinations can be combined into one or more larger messages to reduce the communication overhead [9, 25, 26, 27, 61, 65]. This optimization is called *message aggregation* [25, 26, 27]. For example, the first loop in Figure 3.2b could be replaced with array assignments (i.e., vector notation, Figure 3.2c) so that when parallelized, each communication moves a section of the row rather than just a single element. *Scalar expansion* [42, 43] of the t variable into an array of temporaries is necessary so that the assignments can be expressed in vector notation.

3.2.3 Strip mining

Strip mining [35] replaces a single loop with two nested loops, keeping the total number of iterations of the loop body constant (Figure 3.3). The number of iterations of the inner loop is the *block size* of the computation. Strip mining provides additional opportunities for loop splitting so that communication can be moved out of the innermost loop without moving it completely out of the loop and serializing the computation. For example, for applications such as SOR (Figure 3.4a) parallelized by pipelining multiple instances of DOACROSS loops (Figure 3.4b), strip mining is combined with loop splitting to increase the grain size and then messages are combined to reduce the communication overhead (Figure 3.4c). (A similar example is presented in [25].) The grain size is controlled by setting the *blocksize* variable.

```
DO i = 0, n
     . . .
END DO
```

a) Original loop

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

b) Strip-mined loop

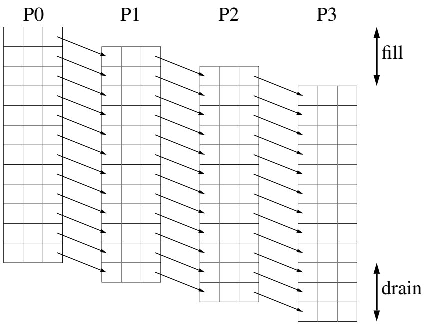
Figure 3.3: Strip mining transformation.

3.2.4 Loop tiling

Tiling the iteration space of loop nests [29, 76, 77] is another transformation that can be parameterized to control the grain size of DOACROSS loops. Tiling combines strip mining of one or more loops with loop

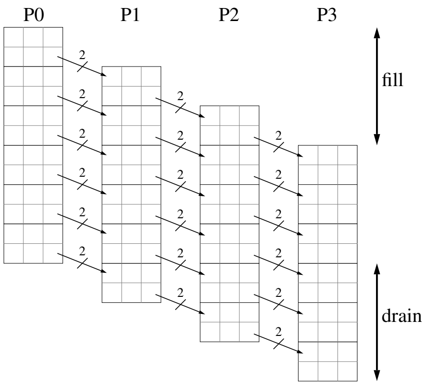
```
for (iter = 0; iter < maxiter; iter++) {
    for (i = 1; i < n - 1; i++) {
        for (j = 1; j < n - 1; j++) {
            b[j][i] = 0.493 * (b[j][i-1] + b[j-1][i] + b[j][i+1] + b[j+1][i]) + (-0.972) * b[j][i];
        }
    }
}
```

(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 3.4: 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.

interchange so that memory references by the resulting inner loops are localized. The size of the tiles is controlled by the block sizes of the strip-mined loops. Tiling is frequently used to increase data locality, especially on uniprocessors and shared-memory multiprocessors, so that most memory references are to data in the cache [29, 53, 73, 77]. However, on a distributed memory multiprocessor, when combined with loop splitting and message aggregation, tiling can also be used to control grain size and communication overhead [65]. In our system, where data is only distributed in one dimension and work is moved by shifting distributed data slices, the general tiling transformation makes data management more complicated, so we limit our discussion of controlling the grain size of DOACROSS loop nests to the less general case of strip mining, as in the SOR example in Section 3.2.3. Much of our analysis regarding grain size should be applicable to the more general case.

3.3 Unidirectional synchronizations

For applications with loop carried flow dependences, i.e., with DOACROSS loops, parallelism can be obtained by pipelining multiple instances of the distributed loop. There are two main factors influencing 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. For a given application, the minimum execution time is attained with a grain size that is a compromise between parallelism and communication overhead. We begin this section with a discussion of how grain size is controlled for applications with DOACROSS loops and then discuss how an appropriate grain size is selected.

3.3.1 Controlling grain size at run time

For applications with DOACROSS loops, the compiler strip mines the loop surrounding the distributed loop and moves communication out of the inner loop, combining messages when possible. The grain size can then be controlled at run time by setting the block size of the strip-mined loop. The grain size (t_{grain}) is related to the block size as follows:

$$t_{grain} = \mathit{blocksize} \times t_{iteration} \tag{3.1}$$

or

$$\mathit{blocksize} = \frac{t_{grain}}{t_{iteration}}$$

where $t_{iteration}$ is the longest of the execution times for the local portion of the distributed DOACROSS loop on all of the processors. Once execution of a strip-mined loop has begun, it is very complicated to modify the block size; thus, the desired block size or grain size must be selected before entering the loop. If the strip-mined loop is executed multiple times, the block size can be changed between the executions of the loop, based on measurements taken during previous executions of the loop. The next few sections describe how the optimal block size is selected, but first we describe how block size is selected given a desired fixed grain size.

Given a desired grain size, an accurate estimate of $t_{iteration}$, the cost of the portion of the loop body executed on each processor, is required to select the appropriate block size. Since this estimate is needed before entering the loop, the cost can not be measured as the loop is executed. If the compiler has an accurate model of execution times for the processor, the compiler can determine $t_{iteration}$. If not, $t_{iteration}$ can be estimated when the application is started by measuring the execution time of a copy of the loop body (operating on dummy data). If the strip-mined loop is executed multiple times, the estimates of the loop body costs can be updated between the executions of the loop, based on measurements of the actual times for executing the loop body during previous executions. In our implementation of SOR, we just use measurements from executing a copy of the loop body at startup time.

In our implementation of the SOR example, we initially computed an average $t_{iteration}$ from measurements of a fixed number of iterations of the dummy loop body. However, we found inconsistencies in the values determined with different numbers of slave processors. These inconsistencies are explained and corrected with some analysis. Assuming a dedicated, homogeneous system, the number of iterations, N_i, assigned to each processor is

$$N_i = \frac{n}{P} \tag{3.2}$$

where n is the problem size (the number of iterations in the distributed loop), and P is the number of slave processors. The $t_{iteration}$ measured on processor i is proportional to N_i, and, thus, varies with the problem size and number of processors. However, because the total amount of work is constant, we expect the total of the measurements of $t_{iteration}$ on all the slave processors to be constant, no matter how many slave processors there are:

$$\sum_{i=0}^{P-1} \underbrace{t_{iteration,i}}_{average\ measurement} \approx \underbrace{t_{iteration}}_{\text{max}\, t_{iteration,i}} \times P \approx \underbrace{k \times n}_{\text{constant}}$$

If $t_{iteration,i}$ is measured on each processor by averaging the time over a fixed number of iterations of the dummy loop body, there is a lot of variability in $t_{iteration} \times P$ because of the different ratios between loop overhead and computation for different numbers of processors. The variability is eliminated by averaging the execution time of the loop body for a number of iterations proportional to the number of processors so that each average involves the same amount of computation and has the same ratio of computation and loop overheads. Experiments verified that with the corrected approach, $t_{iteration} \times P$ was much more stable as the number of processors was varied. Another way to verify the accuracy of the $t_{iteration}$ measurements is to use them to extrapolate the sequential execution time for the problem and compare the predicted times with actual measurements. If there are m executions of the distributed loop and the problem is distributeed on P processors, the sequential time should be

$$t_{sequential} = m \times P \times t_{iteration} \tag{3.3}$$

For SOR, we found that the predictions of the sequential execution time using this approach are consistent over different numbers of processors and are quite close to measurements of the sequential time. Therefore, our measurements of $t_{iteration}$ must be accurate.

3.3.2 Communication costs

Communication is necessary when dependences exist between loop iterations that are assigned to different processors. If this communication is too frequent, communication costs can dominate the execution time, eliminating benefits due to parallelism. The frequency of communication depends on the computation associated with each iteration and the number of loop iterations, N_i, assigned to each processor. The latter value depends on the problem size, n, and the number of processors, P, for equal distribution of work (Equation 3.2). The former value can be changed by loop restructuring transformations, such as strip mining, performed during compilation. Strip mining the pipelined loop and moving communication out of the resulting inner loop provides a way to control the frequency of communication. Communication overhead can then be reduced using message aggregation. Increasing the block size reduces the frequency of communication and increases the number of messages that can be combined. Thus, the larger the block size, the smaller the communication overhead.

3.3.3 Pipeline fill and drain times

However, 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. From Figure 3.5, it can be observed 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}:

$$t_{elapsed} = (M + P - 1) * t_{block} \tag{3.4}$$

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

$$t_{sequential} = P \times M * t_{block} \tag{3.5}$$

We can now compute an upper bound on efficiency for a homogeneous environment with no competing loads, using Equation 1.4:

$$\begin{aligned} \textit{efficiency} &= \frac{P \times M \times t_{block}}{P \times (M+P-1) \times t_{block}} \\ &= \frac{M}{M+P-1} \end{aligned} \tag{3.6}$$

This upper bound on efficiency is graphed as a function of the number of blocks in Figure 3.6. 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.

3.3.4 Selecting the optimal block size

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 3.5). 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:

$$t_{total} = (M+P-1) \times t_{block} + (M+P-2) \times t_{shift} \tag{3.7}$$

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

M and t_{block} are related by Equation 3.5. 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, the sequential execution time can be estimated by measuring the cost of executing several iterations of a copy of the loop and extrapolating, as described in Section 3.3.1. Our implementation uses the latter approach. Thus, for a given number of processors, P, we can eliminate t_{block} as an unknown by replacing it with $\frac{t_{sequential}}{P \times M}$ and using the estimate of $t_{sequential}$.

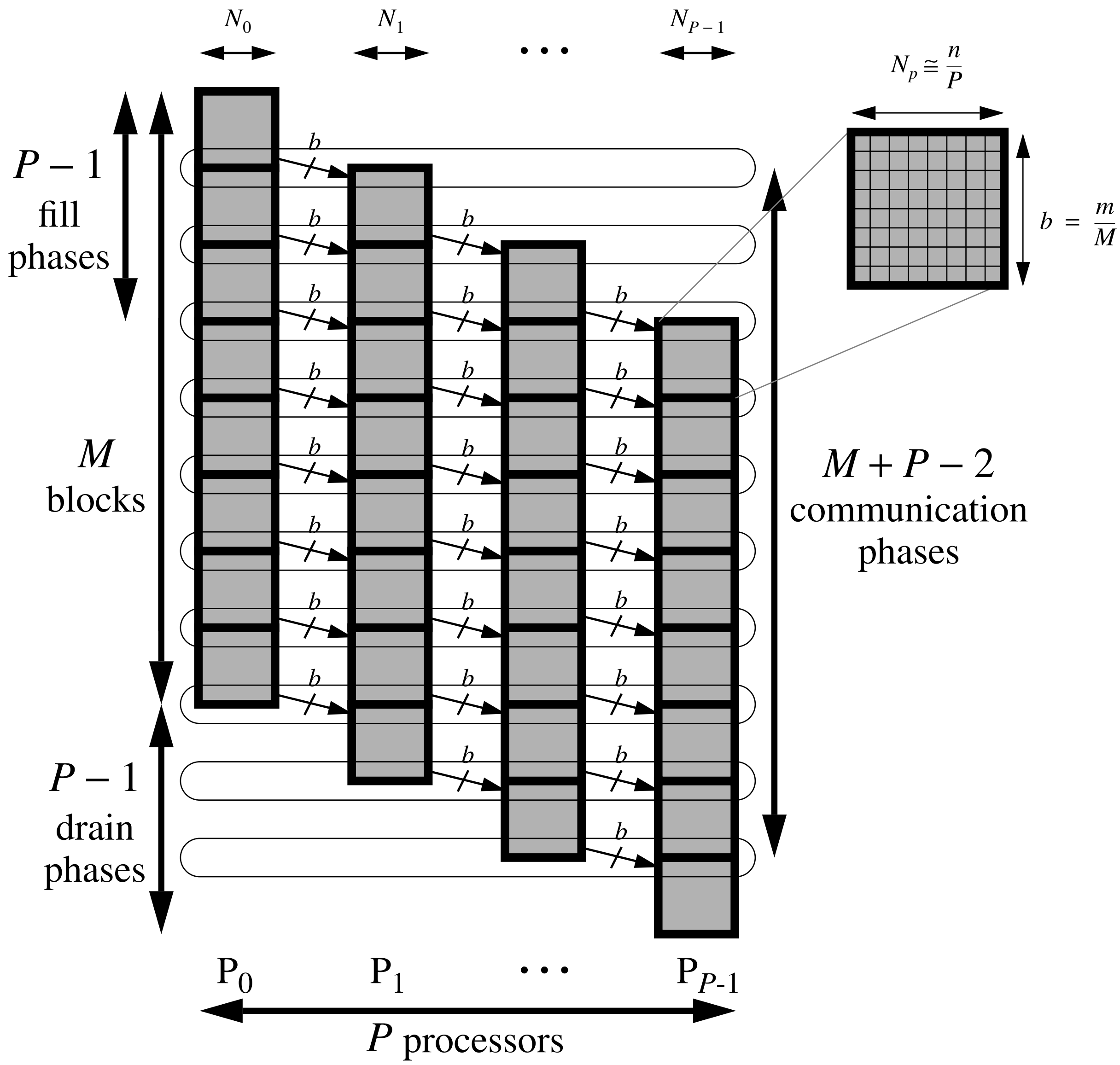


Figure 3.5: Pipelined execution of a distributed loop showing parameters for modeling execution time. Distributed loop has n iterations and pipelined loop (enclosing distributed loop) has m iterations.

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

$$t_{shift} = t_{fixed} + t_{incr} * \textit{elements} \tag{3.8}$$

where t_{fixed} is the fixed overhead of sending messages between processors, 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

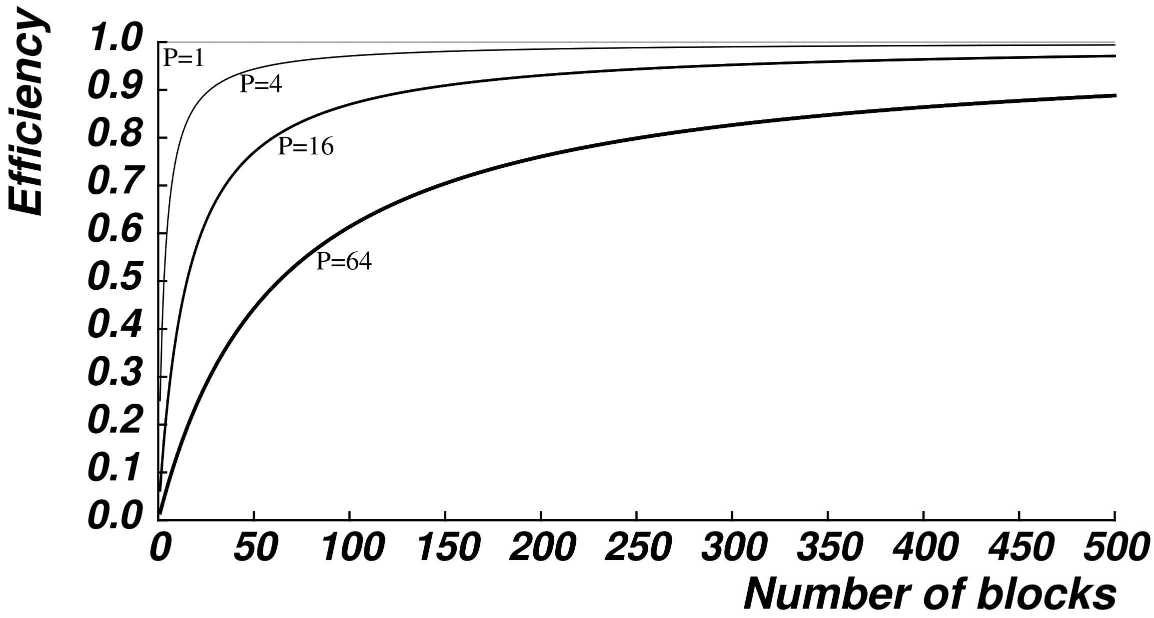


Figure 3.6: Upper bound on efficiency for the pipelined loop in the SOR example (1000x1000 matrix, 10 iterations) as a function of the number of blocks.

is equal to the block size, b:

$$elements = b = \frac{m}{M} \tag{3.9}$$

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, subtracting t_{fixed}, and dividing by the length of the messages.

Substituting for t_{block} and t_{shift} in Equation 3.7,

$$\begin{aligned}
t_{total} &= (M + P - 1) \times \frac{\frac{t_{sequential}}{P}}{M} \\
&\quad +(M + P - 2) \times (t_{fixed} + t_{incr} \times \frac{m}{M}) \\
&= \frac{t_{sequential}}{P} + \frac{t_{sequential}}{M} - \frac{t_{sequential}}{P \times M} \\
&\quad +M \times t_{fixed} + P \times t_{fixed} - 2 \times t_{fixed} \\
&\quad +t_{incr} \times m + P \times t_{incr} \times \frac{m}{M} - 2 \times t_{incr} \times \frac{m}{M}
\end{aligned} \tag{3.10}$$

All values on the right hand side of Equation 3.10 can be determined when the application is started except for M, the number of blocks. 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:

$$\begin{aligned}\frac{dt_{total}}{dM} &= 0 - \frac{t_{sequential}}{M^2} + \frac{t_{sequential}}{P \times M^2} \\ &\quad + t_{fixed} + 0 - 0 + 0 - \frac{P \times t_{incr} \times m}{M^2} + \frac{2 \times t_{incr} \times m}{M^2} \\ &= 0 \end{aligned} \tag{3.11}$$

Solving for M, the shortest execution time and the highest efficiency are attained when

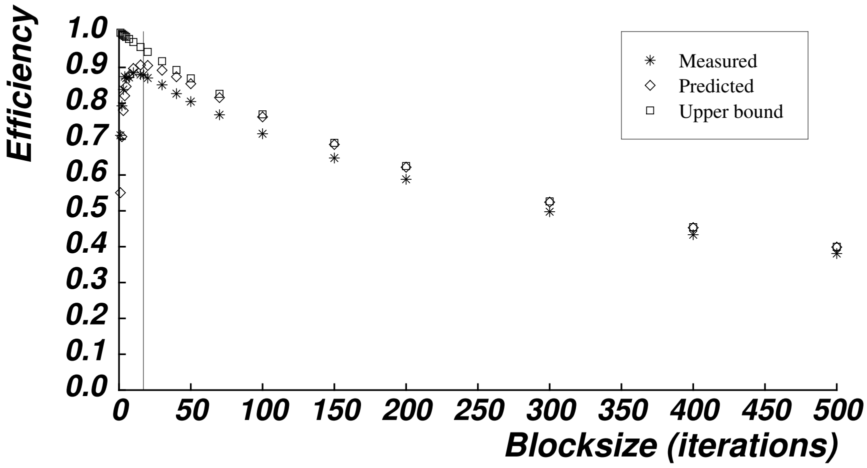
$$M = \sqrt{\frac{t_{sequential} \times (1 - \frac{1}{P}) + t_{incr} \times m \times (P - 2)}{t_{fixed}}} \tag{3.12}$$

The optimal M is computed at application startup time using the known values of P and m and the estimates of $t_{sequential}$, t_{fixed}, and t_{incr} determined by executing copies of small portions of the computation. From this, we determine the optimal block size, $b = \frac{m}{M}$ (Equation 3.9).

3.3.5 Evaluation of grain size model

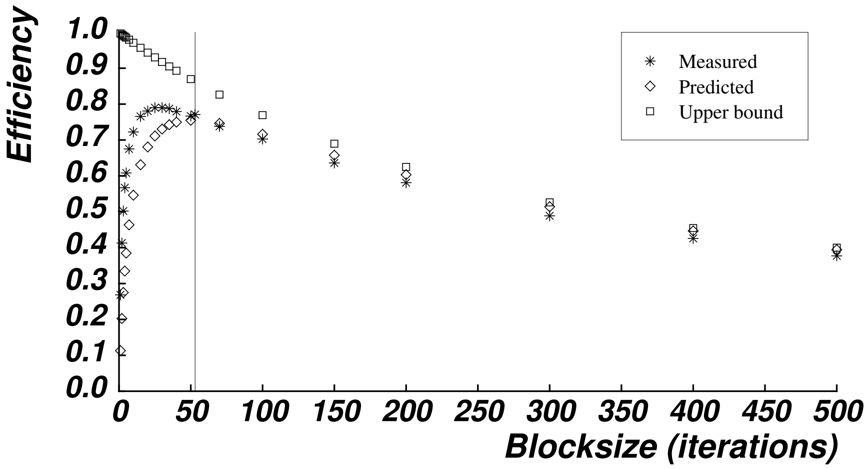
Figure 3.7 shows the efficiency of parallelization for the SOR example (1000x1000 matrix, 10 iterations) predicted by our model and measured on the Nectar system with 4 slave processors. We ran the SOR example with a range of block sizes and with the block size automatically selected using the computations described above; artificial delays were added in Figures 3.7b and 3.7c 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 (listed in Table 7.1). However, the communication costs are more variable and are more difficult to measure. To predict the efficiency and to compute the optimal block size, we use a conservative estimate of the communication costs: the time between the start of the first communication and the end of the last communication in a communication phase. This estimate tends to increase t_{fixed}, reduce t_{incr} and increase the optimal grain size prediction. (The predicted and measured efficiency curves move closer together if the t_{incr} estimate is increased.) Although this cost estimate may include some time spent on computation, a less conservative estimate, such as measuring the communication time from the point of view of a single processor, could result in shifting the optimal grain size prediction to the left where the slope of the efficiency curve is much greater.

Similar analysis will be done within the Fortran D compiler to select the appropriate block size for pipelined computations [26, 27]. As in our approach, the optimal block size is determined by setting



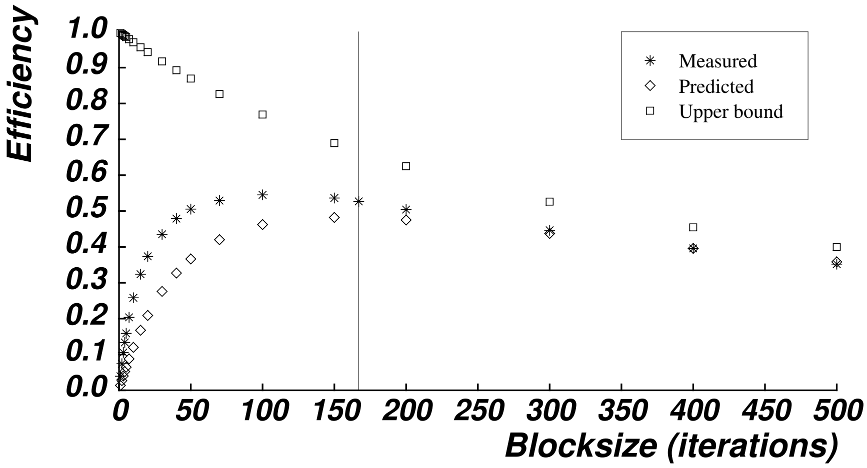
Values used in computing M:
$t_{sequential} = 8478720\mu$sec
$t_{fixed} = 1716\mu$sec
$t_{incr} = 4.8\mu$sec

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 millisecond 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 millisecond delay added to each message.

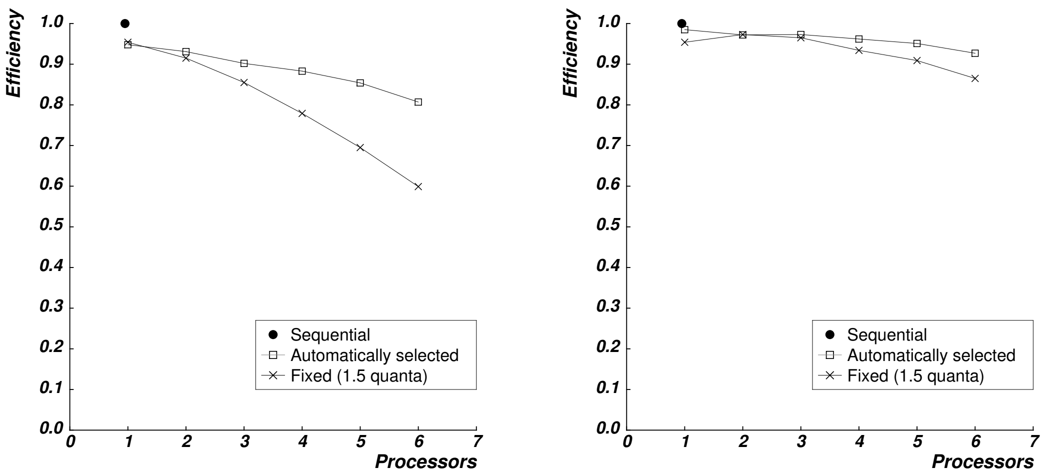
Figure 3.7: Efficiency of the pipelined loop in the SOR example (1000x1000 matrix, 10 iterations) on a 4 slave system as a function of the block size. Vertical lines indicate automatically selected block size and correspond with the peaks of the “Predicted” curves.

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 [23]. 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 in that it measures the costs for the specific application code being executed on the specific machine configuration being used and could be extended to update the costs as the application is executed. However, by delaying our characterization of costs until run time, we add the characterization time to the cost of executing the application.

3.3.6 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 3.3.1, with the performance of a version with the automatically-selected "optimal" grain size, selected using the method described in Section 3.3.4. Figure 3.8 shows efficiency measurements (see Section 1.5) taken on the Nectar system with homogeneous, dedicated processors for two different problem sizes. (In this case, since there are no competing processes on the system, the efficiency measure defined by Equation 1.4 is the same as the traditional, lower bound efficiency measure, Equation 1.2, so the measurements are neither optimistic nor pessimistic.) We selected a fixed grain size of 1.5 time quanta[2] (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.

[2] The *time quantum* or *time slice* is the unit of scheduling used by the operating system. For Unix systems, the time quantum is 100 milliseconds [32].



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

Figure 3.8: 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.

3.3.7 Effect of competing loads

For pipelined applications, when a competing load is added to one of the processors, intermediate results are delayed for all processors following that processor in the pipeline. A bubble of inactivity (idle waiting) passes through the pipeline each time the competing load is given the CPU (Figure 3.9a). However, if the 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 3.9b). 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.

We confirmed that grain size has little effect on the efficiency of a pipelined application in a load balanced environment with competing loads by simulating the interactions between the scheduling of processes by the operating system and the communication between the slave processors, as in Figure 3.9. 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 for a dedicated system is $\frac{M}{M+P-1}$. Figure 3.10 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 predicated upper bound, regardless of the grain size. On systems with competing loads, the

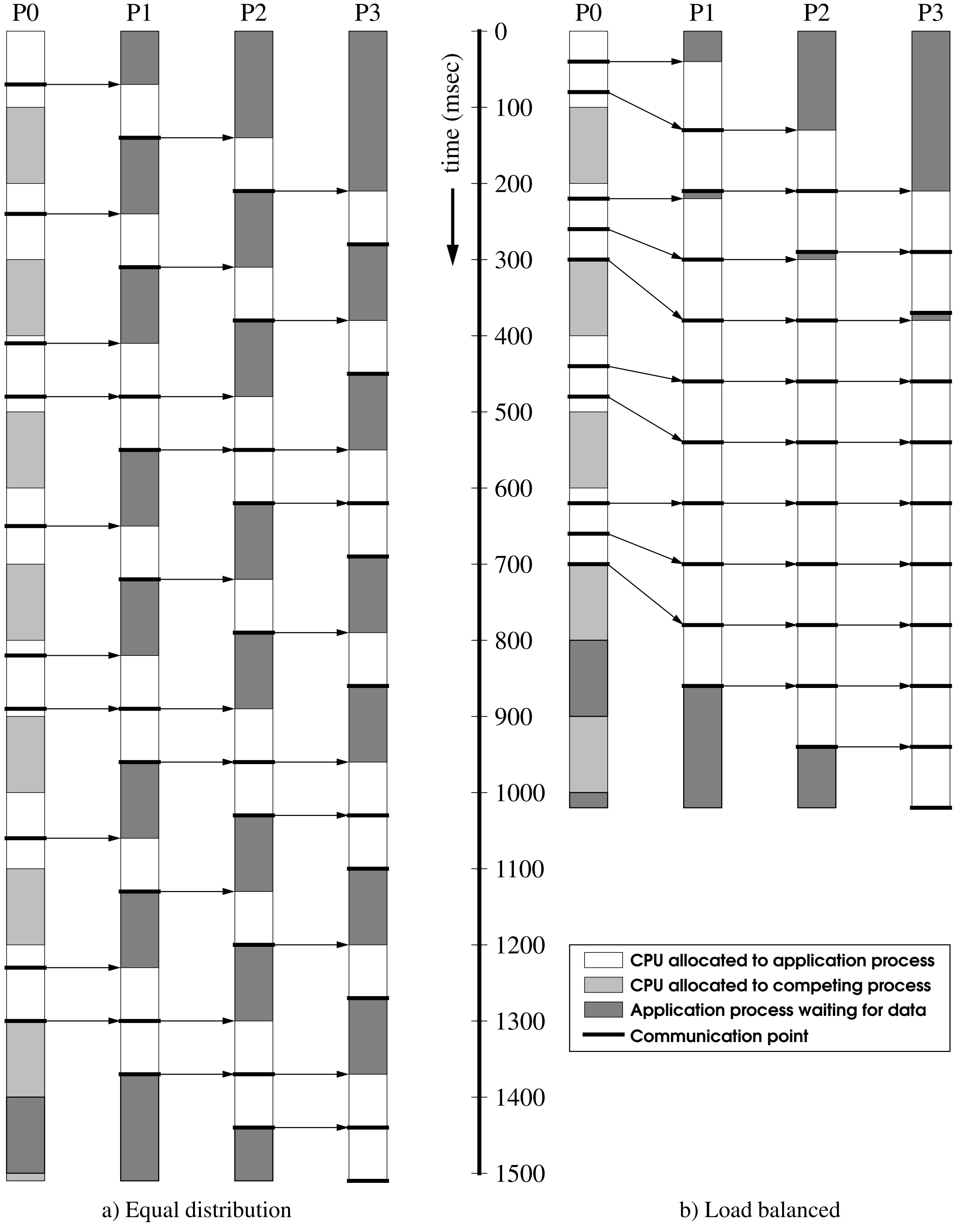
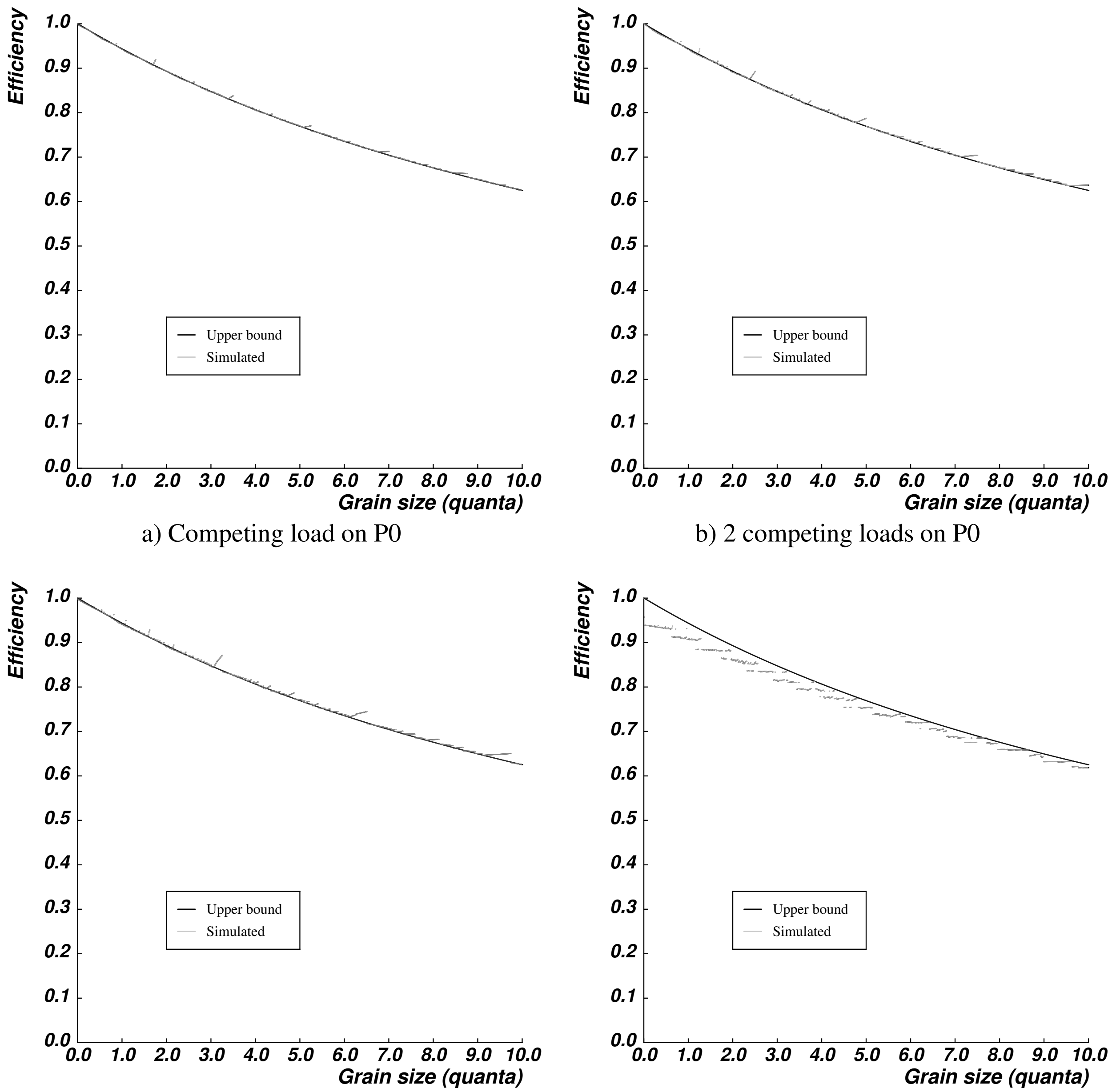


Figure 3.9: Pipelined execution with competing load on first processor.

efficiency sometimes exceeds the predicted upper bound because the length of the blocks varies with each pipeline stage as the phase difference between the start of the competing load and the start of the pipeline stage changes. There may be slight degradations in efficiency (most noticeable in Figure 3.10d) due to time spent by the slaves aligning themselves with each other in the early stages of the pipeline. In real systems, process 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 more than once; however, the natural tendency for communication to align the processors should prevent efficiency from being affected too adversely.

3.4 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*. Since DOALL loops require no communication, the grain size of applications with DOALL loops 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; the LU decomposition example (Figure 1.10) exhibits all of these features. Barrier synchronizations with no computation between them can be treated as a single barrier. With bidirectional synchronizations, grain size can be changed using transformations such as loop splitting (e.g., Figure 3.2), loop interchange, or loop skewing, but these transformations are difficult to parameterize. For limited control of the grain size, the compiler could generate several versions of the code which could be selected at run time [76], but a continuum of grain sizes is not possible as it was in the case of unidirectional synchronizations. Because of this limitation, our research does not investigate options for modifying the grain size of problems with bidirectional synchronizations; we leave it to the programmer or the compiler to decide on the best way to parallelize the application. In the remainder of this section, we analyze the overhead of bidirectional synchronizations and examine the effects of bidirectional synchronizations on program performance in the presence of competing loads.



a) Competing load on P0

b) 2 competing loads on P0

c) 3 competing loads on P0

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

Figure 3.10: Parallelization efficiency determined from simulation of pipelined execution on a 4 processor system. Upper bound ($\frac{M}{M+P-1}$) is included on all graphs. The sequential execution time of the simulated problem is 200 time quanta.

3.4.1 Synchronization overhead

Figure 3.11 shows the basic structure of parallelized code with barrier synchronizations. The barrier synchronizations are on the critical path of the application because they impose a total order on the processors. Thus the communication costs for the synchronizations add to the parallelization overhead for the program. Also, if there is any load imbalance between synchronizations, the processors that finish their computation first remain idle while the other processors finish, reducing the parallelization efficiency.

For a dedicated homogeneous system, we can model the total execution time for the program as follows:

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

where n is the total number of iterations of the distributed loop and m is the number of times the distributed loop is executed. 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

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

In this simple model, all parallelization overhead is due to barriers, and the number of barriers is irrelevant. The overhead can only be reduced by reducing the cost of each barrier. With a good implementation of barriers, this is not an option. Thus, to reduce synchronization overhead, the compiler must restructure the program so that it can be modeled differently; this requires distributing the problem in a different manner, a problem that we do not address.

```
DO i = 1, m
      DO j = locallo, localhi
            compute iteration
      END DO
      global synchronization
END DO
```

Figure 3.11: Parallelized version of DOALL loop followed by global operation.

3.4.2 Effect of competing loads

When multiple processors have competing loads, the scheduling of processes on different processors may not be synchronized, and the application may be inactive on different processors at different times. At each barrier synchronization, the elapsed time will be the worst case of the times on all the processors, and the barriers will cause the skews between execution times to accumulate and show up in the total execution time. Even when work is allocated to processors in proportion to their available resources, on a system with competing loads, the application may not be able to use its share of the processing resources productively due to the interactions between the grain size and the scheduling of processes by the operating system. The inefficiency is even worse if the work assignment is not proportional (i.e., the load is left unbalanced) or if the system is dynamic resulting in varying grain sizes. For effective utilization of resources, the computation assigned to each processor during the period between barrier synchronizations must correspond with the amount of CPU allocated to that processor during that period. This section identifies the grain sizes that make this match more likely.

Modeling scheduling interactions

To evaluate the effects of the barrier synchronizations on performance in the presence of competing loads, we model the scheduling of processes using the round-robin scheduling model described in Section 3.3.7. Barriers work as follows: each application process enters the barrier after completing a computation phase, and none of the process may exit the barrier until all processes have entered. Each process must be active, i.e., have control of its CPU, both when entering and when leaving a barrier, but not all processes must be active at the same time.[3]

Figures 3.12 and 3.13 show time lines for a four processor system with a single competing load on one processor, with different work assignments and grain sizes. The time lines identify the different CPU states (*working*, *waiting*, or *inactive* with respect to the load balanced application) and show the interactions for the barrier synchronizations. In the figures, the thick horizontal lines indicate the times when the application

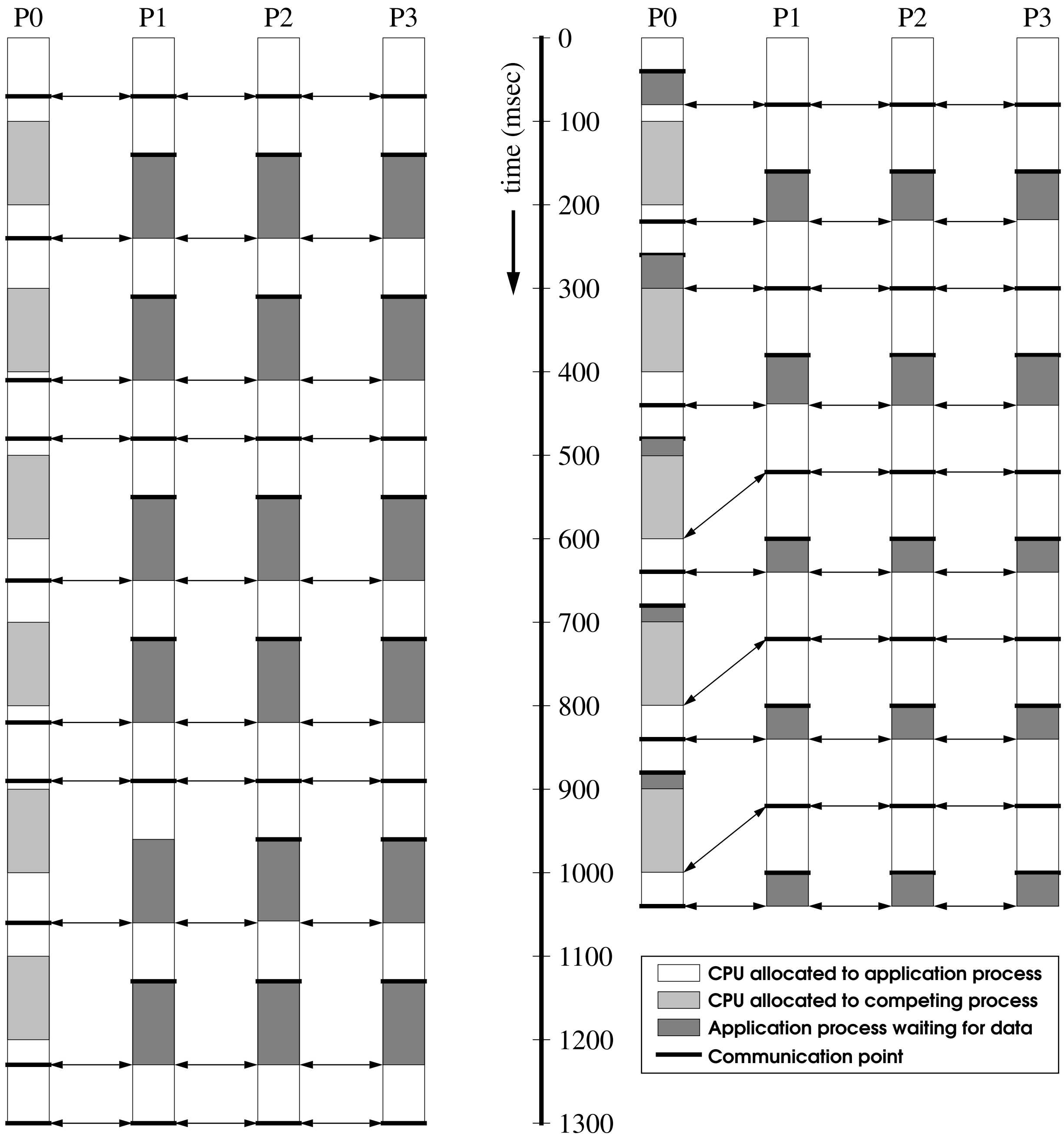
[3]This model for barrier synchronizations requires that the communication needed for the synchronization is buffered. We could use a more restrictive, possibly more realistic, model of the synchronizations, such as requiring all processes to be active at the synchronization point; however, with the simple, but inflexible, round-robin scheduling model, performance predictions would be too pessimistic. To compensate, the round-robin scheduling model would have to be replaced with a more complicated model, e.g., having the application process yield the CPU when waiting for communication and having incoming communication interrupt the competing processes to return the CPU to the application.

processes enter a barrier synchronization, and the arrows indicate the times when the processes exit the barrier. In Figure 3.12, the grain size for the case without load balancing (Figure 3.12a) is 0.7 time quanta. After load balancing (Figure 3.12b), the grain size on the loaded processor (P0) is 0.4 quanta, and on the dedicated processors (P1, P2, and P3) the grain size is 0.8 quanta. With load balancing, there is still quite a bit of time spent waiting at the synchronization points, and the execution time is not reduced much relative to the case without load balancing; load balancing only increases the efficiency from 60.9% to 76.5%. However, grain sizes (after balancing) closer to multiples of the time quantum result in higher utilization of the available CPU time because only occasional small corrections (i.e., small waiting periods) are needed to keep the synchronizations and scheduling in phase. In Figure 3.13 where the grain sizes after load balancing are closer to multiples of the time quantum (0.8 quanta on P0, and 1.6 on the other processors), the time reduction with load balancing is much greater than in Figure 3.12, where the grain size is 0.4 time quanta (on processor 0). The efficiency increases from 58.9% without load balancing to 94.6% with load balancing.

Better CPU utilization also results from increasing the grain size. With round-robin scheduling, after the application process executes for one time quantum, each competing process also executes for one time quantum. Between consecutive computation phases, the number of times the application is interrupted by the scheduler may differ by one due to the phase difference between the computation and the scheduling. Thus, on a processor with constant competing loads, execution times for consecutive computation phases may differ by a factor of $\lceil g \rceil / \lfloor g \rfloor$ where g is grain size (based on dedicated use of the CPU) measured in time quanta. When g is less than one time quantum, consecutive execution times can vary by any factor depending on the load on the system, but when g is greater than one time quantum, the variability factor is bounded, e.g., by 2 for $1 < g < 2$, by $\frac{3}{2}$ for $2 < g < 3$, etc. Thus, if grain size can be controlled, a grain size as large as possible, but at least one time quantum (on the loaded processors after balancing), should be selected.

Simulation of scheduling interactions

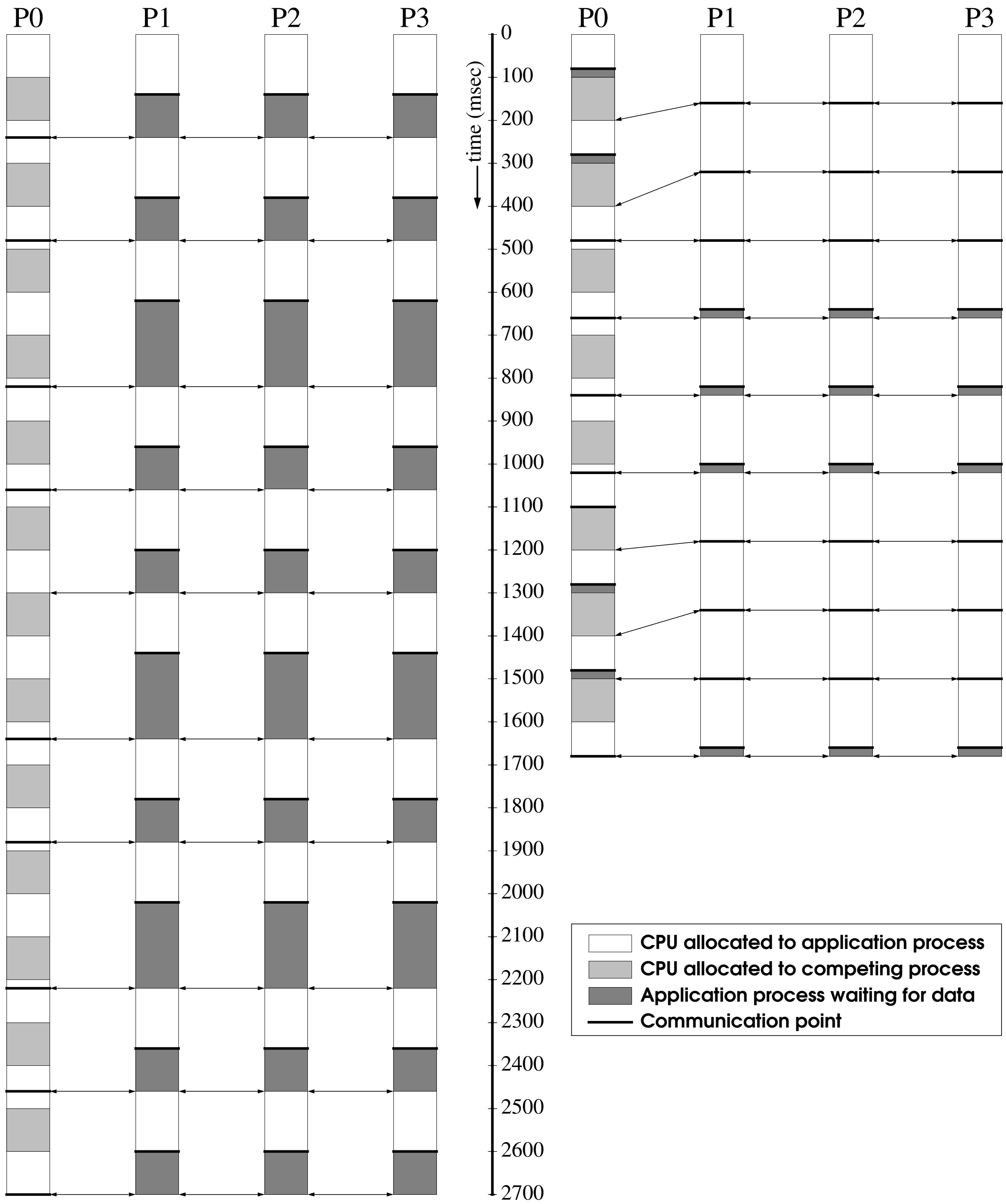
To show the effects of varying the grain size on performance, we simulated the interactions between different grain sizes and the scheduling of processes by the operating system. The simulations model the interactions between the grain size and scheduling in the same manner used in Figures 3.12 and 3.13, but run for 1000 synchronizations. At the start of the simulations, the parallel application is active and at the beginning



a) Equal distribution (*efficiency* $= 0.609$) b) Load balanced (*efficiency* $= 0.765$)

Figure 3.12: Application with bidirectional synchronizations executing with competing load on first processor. Grain size for equal distribution case is 70 milliseconds. Round robin scheduling ignoring communication costs.

of a time quantum on all processors. Figure 3.14 shows parallelization efficiencies attained with different grain sizes under different load conditions. The simulation results confirm our hypotheses: efficiency increases with grain size, but not monotonically; peaks with 100% efficiency occur where the grain sizes

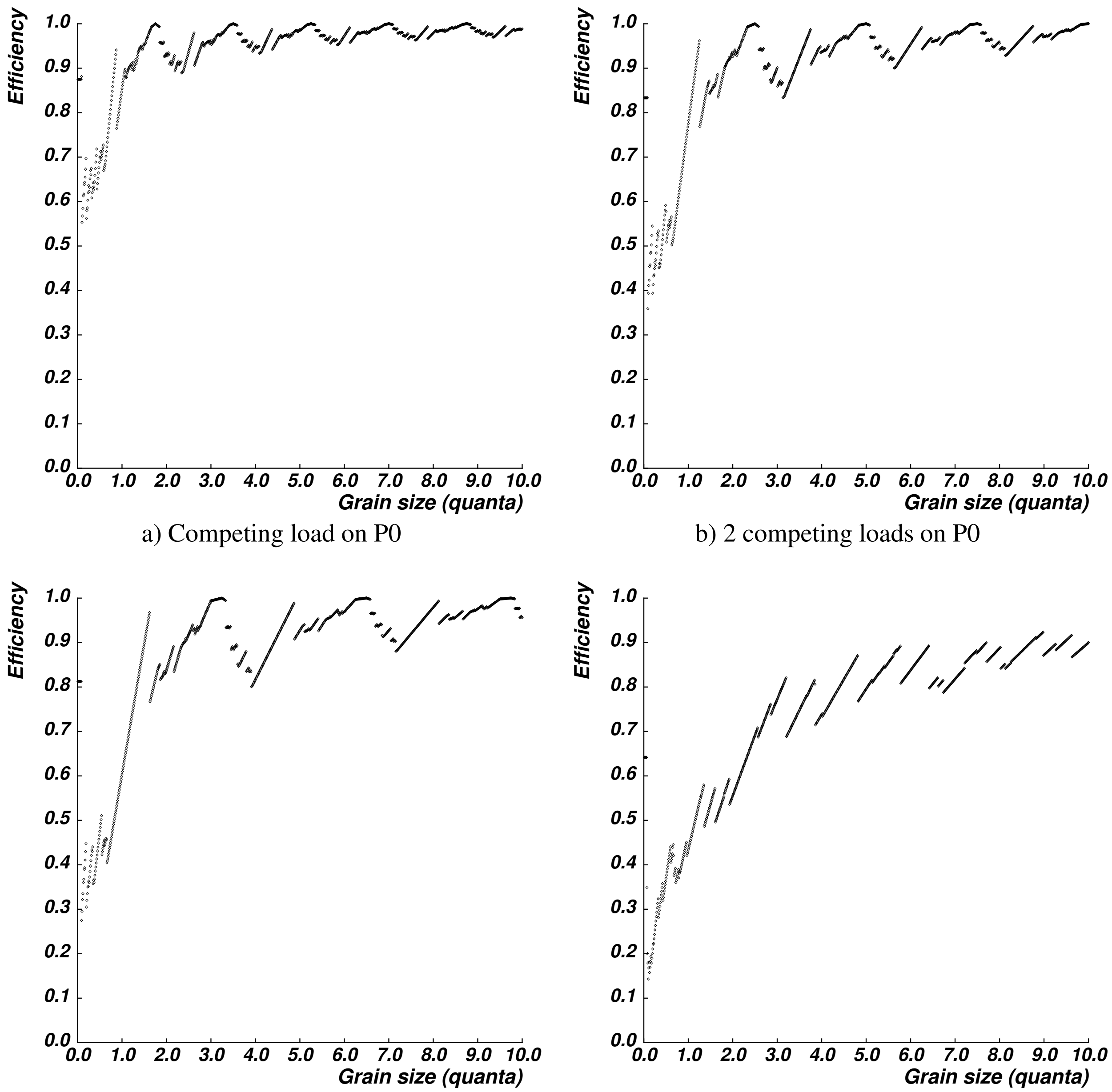


a) Equal distribution (*efficiency* $= 0.589$)   b) Load balanced (*efficiency* $= 0.946$)

Figure 3.13: Application with bidirectional synchronizations executing with competing load on first processor. Grain size for equal distribution case is 140 milliseconds. Round-robin scheduling ignoring communication costs.

and scheduling are in phase.

We can predict the locations of the peaks in the simulation results. With constant loads on the processors,



a) Competing load on P0

b) 2 competing loads on P0

c) 3 competing loads on P0

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

Figure 3.14: Parallelization efficiency for varying grain sizes on a 4 processor system. Simulation results for round-robin scheduling ignoring communication costs.

the CPU allocation schedule for the whole system repeats every lcm quanta, where lcm is the greatest common multiple of the loads ($load_i$) on each processor (including the load balanced application). For 100% utilization during one repetition of the schedule, each processor must be allocated $w_i = \frac{lcm}{load_i}$ quantum units of work. The total work during each lcm quanta period is thus $w_{total} = \sum_{i=0}^{i=P-1} w_i$. Since the x axes of the graphs in Figure 3.14 are the grain sizes before load balancing, the grain sizes with 100% efficiency are multiples of $grain_{peak} = \frac{w_{total}}{P}$. The first 100% efficiency peak on each of the graphs in Figure 3.14 occurs when the grain size on the most heavily loaded processor after balancing equals one time quantum: $grain_{peak} = 1.75$ for Figure 3.14a, 2.50 for 3.14b, 3.25 for 3.14c, and 19.25 for 3.14d. Additional peaks may occur if the schedule for the system can be divided into two or more equal segments for which each segment has the same amount of active time on each processor as each other segment. E.g., for the examples with competing loads on only one of the processors, peaks can also occur at multiples of $\frac{grain_{peak}}{2}$ (efficiencies near 100% can be observed in Figures 3.14a, 3.14b, and 3.14c. However, for 100% efficiency to result with these grain sizes, the computation and the scheduling of the processes must be in precisely the correct phase.

The simulation results confirm that efficiency improves as the grain size is increased, and also show that the sensitivity of the efficiency to fluctuations in grain size decreases as the grain size increases. In actual systems, scheduling algorithms more complicated than round-robin are used, making it more difficult to predict the grain sizes where efficiency peaks occur. Also, normal system activity may cause variations in the schedule so it is desirable to be out of the range of grain sizes where efficiency fluctuates greatly. Therefore, the grain size should be as large as possible for applications with barrier synchronizations if they are to be run in the presence of competing loads.

3.5 Summary

This chapter presented several ways grain size can be controlled using loop restructuring transformations and investigated the interactions between the grain size and scheduling of processes by the operating system. The selection of an appropriate grain size for applications distributed over networks of workstations requires both compile-time and run-time information. The ability to control grain size and the factors that must be considered in selecting an appropriate grain size are determined by the type of synchronizations in the application.

For applications with unidirectional synchronizations (i.e., with DOACROSS loops), we presented and evaluated a method for automatically selecting and controlling the grain size based on close cooperation between the compiler and the runtime system. For an application to use resources efficiently, its grain size must be selected by evaluating a tradeoff between communication overhead and parallelism. However, all the information needed to evaluate the tradeoff is not available until run time. Because our automatic selection approach takes into account the features of both the application (such as problem size) and the system on which it is run (such as the number of processors), it selects the optimal grain size for varied application and system parameters. Our experimental results demonstrate that our automatic selection approach is effective in selecting an appropriate grain size for an application. Using simulations, we also showed that, for applications with unidirectional synchronizations, interactions between the grain size and scheduling by the operating system do not significantly affect performance; therefore the interactions need not be considered in selecting grain size.

For applications with bidirectional (barrier) synchronizations, grain size is more difficult to control at run time. Simulations showed that, to reduce undesirable interactions between bidirectional synchronizations and scheduling by the operating system, the grain size should be made as large as possible.

Chapter 4

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. Load balancing must be performed often enough that fluctuations in performance on the processors can be tracked, but if load balancing occurs too often, the added overheads of interacting with the load balancer and moving work may exceed the benefits of balancing the load. Also, because the frequency of load balancing determines the period over which performance is measured, if the frequency is too high, measurements may fluctuate greatly due to interactions with the scheduling of processes by the operating system rather than due to actual changes in load on the processors; this could cause excessive, unnecessary work movement resulting in excessive overhead.

The load balancing system must be able to select an appropriate load balancing frequency and must be able to modify the frequency as system and application characteristics change. Control of the load balancing frequency involves both the compiler and the runtime system because the compiler must specify points in the code where work can be moved without disrupting the computation or corrupting data. This is most easily coordinated if load balancing interactions occur only at points in the parallelized code where the compiler has inserted *load balancing hooks*, conditional calls to the load balancing code. The runtime system determines when the hooks call the load balancing code.

The next section describes how the compiler and runtime system cooperate in controlling the load balancing frequency. Section 4.2 describes how the compiler places the load balancing hooks into the application code. Section 4.3 describes the factors considered when selecting the load balancing frequency and how the frequency is controlled at run time.

4.1 Cooperation between compiler and runtime system

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, a hook calls the load balancing code, which measures the time spent on the computation, sends performance information to the load balancer, receives instructions, and, if necessary, shifts work between the processors. The granularity at which frequency can be controlled at run time is determined by the placement of the load balancing hooks.

A simple implementation of a load balancing hook is shown in Figure 4.1a. Each time the hook is executed, a counter, *count*, is incremented. Load balancing is triggered when the counter reaches a specified value, *nexthook*. The counter is reset each time the load balancing code is called. 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, according to specifications contained in the instructions. The hook shown in Figure 4.1a implements synchronous load balancing. If the interactions between the slaves and load balancer are pipelined (Section 2.6.1), the hook is modified slightly so that two (or more) status messages are sent before the first set of instructions is received (Figure 4.1b); after the interaction pipeline has been filled (i.e., $phase > 1$), the modified hook code performs the same functions as the code in Figure 4.1a. For asynchronous load balancing (Section 2.6.2), the hook must be modified so that the hook only attempts to receive instructions if an instruction message has been detected (Figure 4.1c); otherwise, the hook just sends status to the load balancer and returns control to the application code.

The run time system determines when the hooks will call the load balancing code, based on a *target load balancing period* ($period_{target}$) and the length of the *computation periods* between the load balancing hooks ($period_{compute}$):

$$nexthook = \frac{period_{target}}{period_{compute}} \tag{4.1}$$

Both $period_{target}$ and $period_{compute}$ may vary at run time. $period_{target}$ is selected by the runtime system

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

a) Synchronous load balancing

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

b) Pipelined load balancing

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

c) Asynchronous load balancing

Figure 4.1: Code for load balancing hook.

based on tradeoffs between responsiveness and overhead. The value of $period_{compute}$ used to compute $nexthook$ is estimated based on measurements of previous computation periods and depends on the number of processors, the competing loads on the processors, the sizes of the distributed iterations, and the number of distributed iterations in the current execution of the distributed loop. The actual *load balancing period*, $period_{lb}$, defined to be the actual time between interactions between the slaves and the load balancer, may differ from the target load balancing period because of variations in the computation periods and due to the requirement that $nexthook$ have an integral value. Figure 4.2 shows the relationships between some of these periods.

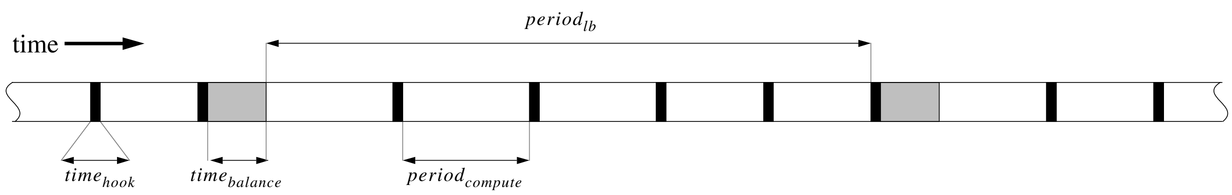


Figure 4.2: Time line showing computation and load balancing periods from point of view of single slave.

4.2 Compiler placement of load balancing code

Load balancing hooks must be inserted at appropriate points in the parallelized code so that load balancing can occur periodically during the distributed computation. 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 it can add too much overhead because 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 describe the hook placement decision in 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. We conclude this section with a hook placement algorithm that merges the two decision steps.

4.2.1 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. However, all possible hook locations at the same level of a loop nest are equivalent with regard to frequency of execution. Therefore, we only identify one potential hook location at each level of a loop nest. Our load balancing only addresses the distributed portion of the computation—i.e., only the execution time of the distributed portion of the code is measured—so, as a starting point, we select locations as closely following the distributed computation as possible. 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 4.3a). The outermost position is immediately eliminated from consideration because load balancing can not reduce execution time after the distributed computation has been completed. Thus, if the distributed loop is an outermost loop, the load balancing hook can only be placed at the end of the distributed loop body. Also, if the hook is placed at the innermost position, between iterations of the distributed loop (e.g., *lbhook0* in Figure 4.3a), controlling the frequency is more complicated because the number of iterations of the distributed loop on each processor may vary. In this case, the value of *nexthook* (Figure 4.1) sent to each processor must be different and must be based on the relative computation rates of the processors so that all slaves interact with the load balancer at the same frequency.

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 location at that level is shifted to the point immediately preceding the first communication following the distributed loop. If the first such location is the receive operation for a unidirectional synchronization

```
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 4.3: 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.

(an unlikely situation), the hook is shifted to the point immediately preceding the first send operation so that the load balancing communication does not interrupt the communication required by the application, possibly creating a deadlock situation.

4.2.2 Selecting from among possible hook locations

A compiler can use simple rules to select the location of the load balancing hook from the list of possible locations. The potential hook locations are evaluated from the most deeply nested location to the least deeply nested. To minimize the time for the load balancer to detect and respond to performance changes, the hook is inserted at the innermost location that adds a negligible amount of overhead (e.g., less than or equal to 1%) to the computation, assuming that the hook never calls the load balancing code. (Control of the overhead when load balancing code *is* called is handled at run time.) Each hook location is evaluated by comparing the estimated execution time for a hook (that does not call the load balancing code) with an estimate of the execution time for the computation 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 potential hook location, then the location is eliminated from consideration. Because this decision is only concerned with orders of magnitude, operation counts can be used to estimate execution time, or, if desired, more accurate methods, e.g., using training sets [5, 23], could be used. Since a hook consists of several operations, the computation between executions of the hook must consist of several hundred operations to keep hook overhead negligible. The innermost possible hook location that satisfies this criterion is selected as the location for the hook. If none of the possible hook locations satisfy the criterion, the outermost hook location is selected, although, in this case, dynamic load balancing is less likely to improve performance. If the code between hook executions includes loops with bounds that can not be determined at compile time, 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. Or, the compiler can generate multiple copies of the loop nest, each with the hook placed at a different nest level; at run time, the values of the loop bounds can be used to select the appropriate version of the loop nest.

In the SOR example in Figure 4.3a, *lbhook0* would create too much overhead because computing an element of the **B** matrix only requires several multiplication and addition operations. For problems of reasonably large size, *lbhook1* meets the overhead constraints, so the compiler would select it as the location for placing the hook. If *lbhook2* were selected, the hook would be executed much less frequently, so the system could not be very responsive to performance changes on the processors.

In addition to determining the maximum frequency at which load balancing can occur, the hook location determines the granularity at which frequency can be controlled at run time. For example, if the minimum computation period due to the hook location is larger than the target load balancing period, the load balancing period can not be controlled, and the system is likely to be unresponsive to variations in processor performance. If the minimum computation period is of the same order of magnitude as the target period, it may be possible to control the load balancing period, but not with much accuracy. Ideally, the minimum computation period should be a small fraction of the target load balancing period (e.g., 10% or less) so that the load balancing period can be controlled with reasonable accuracy.

A further consideration is that load balancing may add synchronizations to the application and may affect the grain size of the application. If load balancing code is executed more frequently than the communication inherent in the application, the grain size for the computation will be reduced. However, load balancing hooks can be placed so that they are executed more frequently than the inherent communication without affecting grain size, as long as the hooks are called less than once per computation phase (with respect to the application's grain size) and, preferably, are placed adjacent to existing communication. Because load balancing interactions are likely to be more expensive than the communication required by the application at each of its synchronization points, the control of the load balancing frequency to limit load balancing overheads will prevent the load balancing interactions from interfering with the grain size of the application in cases where grain size small enough to be critical to performance (e.g., the unidirectional cases where grain size is controlled by the system).

4.2.3 Code restructuring to create better hook locations

In some cases, due to large loop bounds, the compiler may have to choose between a possible hook location with high overhead and a possible hook location that is executed very infrequently. In the matrix multiplication example shown in Figure 4.4a, the two most promising choices are after each iteration of the distributed loop (*lbhook0*) and after the distributed loop (*lbhook1*). For some problem sizes, *lbhook0* has too much overhead, but many thousands of operations may be executed between executions of *lbhook1*. Thus, placing the hook at the deepest level of the loop nesting with low overhead does not always address responsiveness requirements. In situations where the compiler must choose between locations with high overhead and poor response time, the compiler can use strip mining (Section 3.2.3) to create an intermediate choice. When a large loop is split into two nested loops, 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 hook overhead, system responsiveness, and granularity in controlling the load balancing period at run time. (Again, note that it is complicated to control load balancing frequency using location *lbhook0* or *lbhook0a* because these hooks occur between the iterations of the distributed loop. In our implementation of MM, we placed the hook at *lbhook1*, in spite of the fact that it makes control of frequency less accurate and makes the system less responsive to performance fluctuations.) In the SOR case (Figure 4.3b), strip mining is already used to control grain size so, while the possible hook location added by strip mining (*lbhook1a*) may be the best location for the load balancing hook, the block size should not be used to control hook frequency at run time because controlling the grain size for the application is more important. To avoid dealing with this potential conflict, restructuring to improve grain size should precede placement of load balancing code.

Strip mining can also be beneficial in the case where loop bounds are unknown at compile time. The compiler can set the bounds of the inner loop so that it can count the number of statements in the inner loop and thus evaluate the new possible location for the load balancing hooks. When there are multiple loops with unknown bounds (Figure 4.5a), in some cases, each of the loops could be strip mined (Figure

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

4.5b), and loop interchange (Figure 4.5c) could be used so that all of the inner loops created by strip mining are inside all of the outer loops, allowing the compiler to evaluate several new locations by setting block sizes. In Figure 4.5c, the frequencies of possible hook locations *lbhook0a*, *lbhook1*, *lbhook1a*, *lbhook2*, and *lbhook2a* can all be controlled using the block sizes *xsize*, *ysize*, and *zsize*. At run time, if one of the loops has fewer iterations than expected, the block sizes of the other loops can be adjusted to compensate so that most of the compiler's estimates of statement counts remain correct. In figure 4.5b, where loops have been strip mined but not interchanged, only the frequencies of locations *lbhook0a*, *lbhook1a*, and *lbhook2a* can be controlled by the block sizes (*zsize*, *ysize*, and *xsize*, respectively) and with fewer degrees of freedom.

4.2.4 Hook placement algorithm

Algorithm 4.1 is a rudimentary algorithm for placing a load balancing hook in a loop nest, assuming known loop bounds. The algorithm selects the nest level where the hook will be placed, using strip mining to create an additional nest level when the given structure does not give a good balance of responsiveness and overhead. The algorithm assumes that the cost of the executing the load balancing hook and the costs of executing each level of the loop nest can be estimated reasonably accurately. (With known loop bounds, it should not be difficult for a compiler to calculate the number of operations executed at and below each level.) Once Algorithm 4.1 identifies the level for hook placement, it calls Algorithm 4.2 to place the hook at an appropriate point in the level so that load balancing does not create additional synchronization points unnecessarily. Algorithm 4.2 places the hook at the first location following the distributed loop (or loop

```
DO i = 0, x
      DO j = 0, y
            DO k = 0, z
                  compute
                  lbhook0
            END DO
            lbhook1
      END DO
      lbhook2
END DO
lbhook3
```

a) Original code

```
DO i0 = 0, x/xsize
      DO i = i0 * xsize, (i0 + 1) * xsize - 1
            DO j0 = 0, y/ysize
                  DO j = j0 * ysize, (j0 + 1) * ysize - 1
                        DO k0 = 0, z/zsize
                              DO k = k0 * zsize, (k0 + 1) * zsize - 1
                                    compute
                                    lbhook0
                              END DO
                              lbhook0a
                        END DO
                        lbhook1
                  END DO
                  lbhook1a
            END DO
            lbhook2
      END DO
      lbhook2a
END DO
lbhook3
```

b) Strip-mined code

```
DO i0 = 0, x/xsize
      DO j0 = 0, y/ysize
            DO k0 = 0, z/zsize
                  DO i = i0 * xsize, (i0 + 1) * xsize - 1
                        DO j = j0 * ysize, (j0 + 1) * ysize - 1
                              DO k = k0 * zsize, (k0 + 1) * zsize - 1
                                    compute
                                    lbhook0
                              END DO
                              lbhook0a
                        END DO
                        lbhook1a
                  END DO
                  lbhook2a
            END DO
            lbhook1
      END DO
      lbhook2
END DO
lbhook3
```

c) Strip-mined code with loops interchanged

Figure 4.5: Using strip mining and loop interchange to increase control of load balancing hook frequency.

body) in the given level, unless there is communication in the level. If the level has communication, the hook is placed immediately preceding the first send or barrier synchronization that follows the distributed loop. Algorithm 4.1 fails to place a hook if the problem is so small that the hook will create too much overhead if placed anywhere in the loop nest.

In Algorithm 4.1, strip mining is considered if a given nest level meets the overhead constraints but does not allow the load balancing period to be controlled to allow the desired responsiveness, as determined by the target load balancing period ($target$) and the responsiveness fraction (*respf*). However the strip mining is not performed unless the new hook location between the two loops resulting from strip mining can still meet the overhead constraints with at least two iterations of the resulting outer loop (the 0.5 in the algorithm). In making this additional restriction, we are giving the requirement to minimize overheads higher priority over the desire for responsiveness when the two goals are in conflict.

The input parameters for Algorithm 4.1 are chosen heuristically. *target* is a rough approximation of the target load balancing period based on a priori knowledge of the system on which the application is run; e.g., Algorithm 4.1: **InsertHook – Place load balancing hook in loop nest.**

Input:
Loop nest including distributed loop.
ovhf: overhead fraction allowed for hooks.
respf: responsiveness fraction for setting granularity of frequency control.
target: estimate of target load balancing frequency.

Output: Inserts hook in loop nest.

Assumptions: Loop bounds are known.

Method:

1. Number nested loops from distributed loop outwards. The distributed loop is loop 0. The outermost loop is loop $outermost - 1$.
2. Number nest levels from the distributed loop outwards. The body of the distributed loop is level 0. The outermost level is level $outermost$.
3. Estimate hook cost (*hookcost*) by counting operations in load balancing hook, assuming that the load balancer is not called.
4. Loop through nest levels selecting innermost level at which hook will add negligible overhead.

   Cost(*level_number***)** returns estimate of total cost of current level, including lower levels. Estimate cost by counting the operations executed at the current level and all lower levels.

   StripMine(*loop***)** strip mines specified loop creating outer loop *loop* and inner loop *loop*′. Block size (*blocksize*) of inner loop is set at compile time.

   InsertHookLevel(*level_number***)** inserts hook at appropriate location in specified level. (See Algorithm 4.2.)

```
placed = FALSE;
for (loop = 0; loop < outermost; loop++) {
    /* Decide whether to place hook at level loop+1.  Also,      */
    /* look at level loop+1 to decide whether to strip mine.     */
    level = loop + 1;
    /* check whether level meets overhead constraint.            */
    if (Cost(level) * ovhf > hookcost) {
        /* strip mine to maximize ability to control frequency  */
        /* at run time.  Body of outer loop of strip-mined loop */
        /* must meet overhead constraint.                        */
        if ((Cost(level) > respf * target)
                && (0.5 * target * ovhf > hookcost)) {
            StripMine(loop);
            blocksize = Max(hookcost / ovhf, respf * target)
                    / Cost(loop);
            InsertHookLevel(loop');
            placed = TRUE;
        }
        else {
            InsertHookLevel(level);
            placed = TRUE;
        }
        break;
    }
}
if (placed == FALSE) {
    Message("Problem too small");
}
```

Algorithm 4.2: **InsertHookLevel – Insert load balancing hook at given level.**

Input: Level number.

Output: Inserts hook at specified level.

Method:

1. Number statements at current level. Compound statements (e.g. conditionals and loops) are treated as single statements.
2. Identify boundaries of current level.

   $first$: first statement at current level.
   $precede$: last statement at current level preceding loop header for surrounded level.
   $follow$: first statement at current level following end of loop for surrounded level.
   $last$: last statement at current level.
3. Loop through statements at current level to pick hook location either as early as possible or as close as possible to existing communication code.

   IsSend(*statement_number***)** returns true if the statement is a send operation or a compound statement containing send operations, but no receive operations.

   IsReceive(*statement_number***)** returns true if the statement is a receive operation or a compound statement containing receive operations, but no send operations.

   IsBarrier(*statement_number***)** returns true if the statement is a barrier synchronization or a compound statement containing both send and receive operations.

   PlaceHook(*statement_number***)** places the hook code following the specified statement.

```
current = follow - 1;
stop = FALSE;
for (s = follow; s <= last; s++) {
    if (IsSend(s) || IsBarrier(s)) {
        current = s - 1;
        stop = TRUE;
        break;
    }
    else if (IsReceive(s)) {
        current = s;
    }
}
if (stop == FALSE) {
    for (s = first; s <= precede; s++) {
        if (IsSend(s) || IsBarrier(s)) {
            current = s - 1;
            stop = TRUE;
            break;
        }
        else if (IsReceive(s)) {
            current = s;
        }
    }
}
PlaceHook(current);
```

the scheduling quantum is 100 millisecond, so the target estimate is set to 1 second. (This estimate is made based only on information known at compile time, but its rationale is the same as that used in selecting the target load balancing period at run time, described in Section 4.3.3.) The value of *ovhf* should be around 1% so that executing the hooks does not result in excessive overhead. *respf* is set to 10% so that the load balancing period can be controlled within 10% of the target period.

For applications with infrequent synchronizations, load balancing interactions may be the most frequent communication in the application, and thus may determine the grain size. In placing load balancing hooks, Algorithm 4.1 does not explicitly consider the effects of communication added by the hooks on the grain size of the application. However, the selection of the load balancing frequency at run time considers some of the same constraints considered in selecting grain size; and these constraints should prevent the placement of load balancing code from interacting with the grain size of the application in cases where the application has a grain size that makes parallelization practical. Also, to make the implementation of the load balancer easier, we avoid placing the hooks between iterations of the distributed loop so that each slave can receive the same value of *nexthook*. Algorithm 4.1 must be modified if it is to consider this additional constraint.

4.2.5 Timing code

In addition to placing code for calling the load balancing routines, the compiler must place code for timing the computation so that rates of execution can be computed. Timing routines must be inserted before and after the portion of the code to be timed. If multiple segments of code are to be included in computing the rate of execution, each segment is surrounded by timing routines and the times from all segments are added together. Because timing routines can be expensive[1], the code should be divided into as few segments as possible to minimize the number of calls to the timing routines. Also, if too large a portion of the code is not timed, the timing measurements may not capture load information needed for load balancing because much of the process switching may occur when the timer is off.

Computing processing rates requires at least one timing measurement per load balancing period. The measurements should at least include the time spent on the distributed loop iterations because they are the work units used in computing the rate. Loop overheads can also be included in the measurements because they are added on a per iteration basis and do not change the relative rates of the different processors.

[1]The Unix *gettimeofday* routine takes 30–40 microseconds per call on a Sun 4/330 processor because the routine requires system calls. Using the Nectar system, the time can be read in 3–4 microseconds because the timer values can be read directly from memory mapped registers on the Communication Accelerator Board.

Communication times should not be included in the timing measurements for a processor because the communication costs depend on the performance of the network and on the performance of other processors. Thus, if the application requires communication between load balancings, the measured time must be the sum of measurements of two or more code segments. However, sequential portions of the code (see Figure 4.6), replicated on all slaves, should be included in the timing measurements used to compute the processing rates and, thus, do not further segment the timed code. As more work is allocated to a processor, the fraction of the total time between load balancings taken by the replicated portion of the code decreases for that processor, leaving more resources to the distributed portion of the code so that a larger fraction of the distributed work can be allocated to the processor. If the replicated portion of the code is not included in the timing measurements, this effect will not be evident in the measured rates so the load balancer can not take advantage of the added resources. When the replicated portion of the code is included in the measurements, the time per work unit decreases as work units are added, increasing the measured rate so that more work will be allocated to the processor, as desired. Therefore, the only code between interactions with the load balancer that should not be included in the timing measurements is communication code. For convenience, however, we insert timing code so that at least one segment of code is timed per computation period. If the load balancing period consists of more than one computation period, the measured times for the included computation periods are accumulated.

```
DO i = 0, n
      starttime
      DO j = locallo, localhi    /* distributed  */
            <compute iteration>
      END DO
      <replicated code>         /* sequential  */
      stoptime
      lbhook
END DO
```

Figure 4.6: Placement of timing code. Replicated sequential code is included in the timing measurements.

4.3 Selection of load balancing frequency at run time

To respond quickly to changes in performance on the processors, load balancing should occur as often as possible. However, load balancing overheads limit the frequency at which load balancing is practical. Several factors in the execution environment contribute to load balancing overhead. Communication costs are the main factor in the overhead, 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; process 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 4.7) place lower limits on the load balancing period and, thus, upper limits on the load balancing frequency. (The load balancing frequency is approximately the inverse of the load balancing period as defined in Section 4.1.) 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. The target period is used to set the count ($nexthook$ in Figure 4.1) that determines when the load balancing hook calls the load balancing code.

4.3.1 Interaction overhead

Collecting performance information and interacting with the load balancer adds overhead, even if the system is balanced. The cost of each interaction increases with the number of slaves due to increased computation on the load balancer, and the total 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, $k_{interact}$, 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. However, during

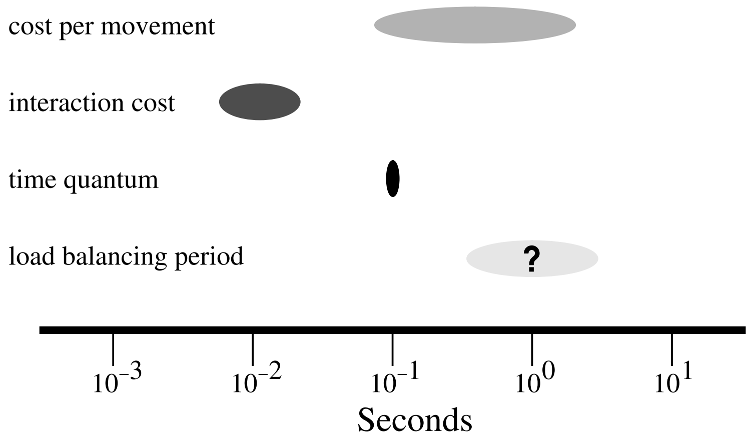


Figure 4.7: Periods affecting selection of load balancing period. The ovals show the approximate ranges (in seconds on a logarithmic scale) for the periods for the application examples when run on the Nectar system. Values are more likely toward the centers of the ovals.

the computation, the interaction time may vary with loads on the processors and delays in the communication network. With synchronous load balancing, it is convenient to update the estimate of interaction costs as the program executes. However, for pipelined or asynchronous 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 and asynchronous cases 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:

$$period_{interact} = \frac{t_{interact}}{k_{interact}} \tag{4.2}$$

4.3.2 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:

$$period_{movement} = \frac{t_{movement}}{workscale} \tag{4.3}$$

$t_{movement}$ 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.

4.3.3 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, if the time quantum used for scheduling is small and the loads on the 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. The load balancing period must be several times as large as the time quantum for performance measurements to appear stable on a processor with constant competing loads. Thus, the time quantum ($t_{quantum}$) sets another lower bound on the load balancing period:

$$period_{scheduling} = t_{quantum} \times quantumscale \tag{4.4}$$

To determine an appropriate value for *quantumscale*, we analyze the effects of the sampling period on the amplitude of fluctuations in performance. The amplitude of the fluctuations in computation rate due to scheduling by the operating system is the difference between the maximum rate for the application, hi, observed when the application has dedicated use of the CPU, and the minimum rate of execution, $lo = 0$, when other processes have control of the CPU:

$$amplitude = hi - lo \tag{4.5}$$

Sampling over periods higher than the time quantum results in averaging of periods including both the maximum and minimum rates. To determine the effect of sampling on the amplitude of observed fluctuations, we subtract an estimate of the minimum rate with sampling, $sample_{min}$, from an estimate of the maximum rate with sampling, $sample_{max}$, assuming simple, round-robin scheduling by the operating system. $sample_{max}$ is observed when the sampling period, s, begins with the end of the measured application's time quantum (Figure 4.8a), and $sample_{min}$ is observed when the sampling period begins with the start of the application's time quantum (Figure 4.8b).

$$sample_{min} = \begin{cases} \frac{1}{s} \times ((q \times d) \times hi + (q \times c + r) \times lo) & \text{if } r \leq c \\ \frac{1}{s} \times ((q \times d + (r - c)) \times hi + (q \times c + c) \times lo) & \text{if } r \geq c \end{cases} \tag{4.6}$$

$$sample_{max} = \begin{cases} \frac{1}{s} \times ((q \times d + r) \times hi + (q \times c) \times lo) & \text{if } r \leq d \\ \frac{1}{s} \times ((q \times d + d) \times hi + (q \times c + (r - d)) \times lo) & \text{if } r \geq d \end{cases} \tag{4.7}$$

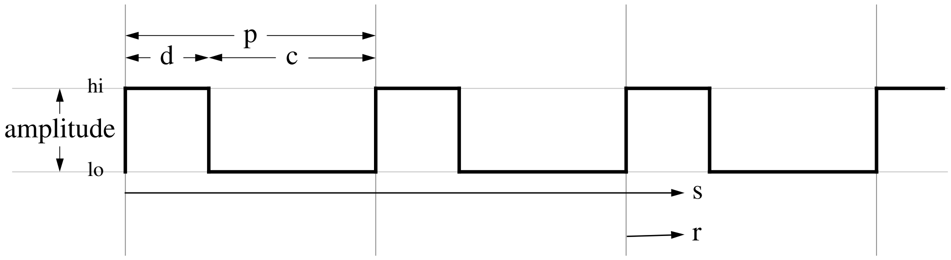
d is the duration of the portion of the oscillation period (p) dedicated to the measured application, and c is the remaining portion of the period:

$$p = d + c \tag{4.8}$$

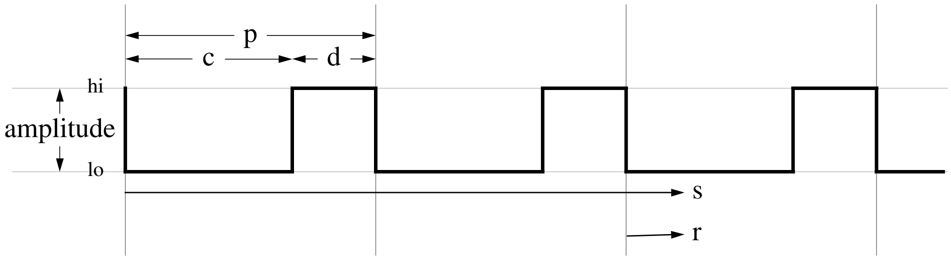
q is the number of full oscillations that are contained in the sampling period ($\lfloor \frac{s}{p} \rfloor$), and r is the remaining time in the sampling period ($s - q \times p$). Subtracting $sample_{min}$ from $sample_{max}$ and dividing by the oscillation amplitude ($hi - lo$) yields the scaling factor due to the sampling period:

$$scale = \begin{cases} \frac{r}{s} & \text{if } r \leq d \text{ and } r \leq c \\ \frac{d}{s} & \text{if } d \leq r \leq c \\ \frac{c}{s} & \text{if } c \leq r \leq d \\ \frac{p-r}{s} & \text{if } r \geq d \text{ and } r \geq c \end{cases} \tag{4.9}$$

Figure 4.9 shows the scale factor for different sampling periods with different numbers of competing loads. (We assume that each process is allocated the CPU in full time quantum units.) Independent of the



a) Maximum sample



b) Minimum sample

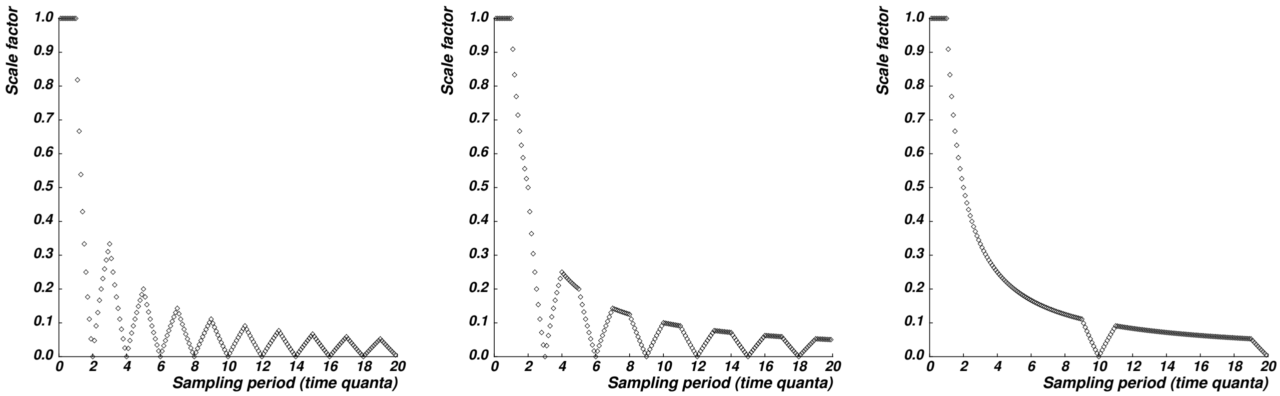
Figure 4.8: Sampling of oscillating performance information.

load on the system, the scale factor is bounded by

$$upper\ bound = \begin{cases} \frac{d}{s} & \text{if } d \leq c \\ \frac{c}{s} & \text{if } c \leq d \end{cases} \tag{4.10}$$

When there are one or more competing loads on the system, $d \leq c$, and the scale factor is at most $\frac{d}{s}$. With no competing loads, there is no oscillation, and sampling has no effect on the range of measurements. Because measured rates can fall anywhere between $sample_{min}$ and $sample_{max}$, observed oscillation amplitudes will usually be even smaller. We have chosen to limit the magnitude of oscillations in measured performance to at most 10% of the maximum amplitude so that variations in the measured rates due to scheduling by the operating system will not exceed the threshold level required for load balancing (described in Section 5.3). The desired scaling factor ($scale = 0.1$) is attained when the sampling period is at least 10 times the time quantum, i.e., $quantumscale = 10$. Figure 4.10 shows normalized performance measurements with different sampling periods for a system with one competing load. As expected, the magnitude of oscillations decreases as the sampling period is increased and is in agreement with our model.

If the time quantum is very large, it may not make sense for the load balancing period to be several times the length of the time quantum; applications may not run long enough for load balancing to have any positive effect. If, due to a large time quantum, the load balancing period must be about the same size as



a) 1 competing load b) 2 competing loads c) 9 competing loads

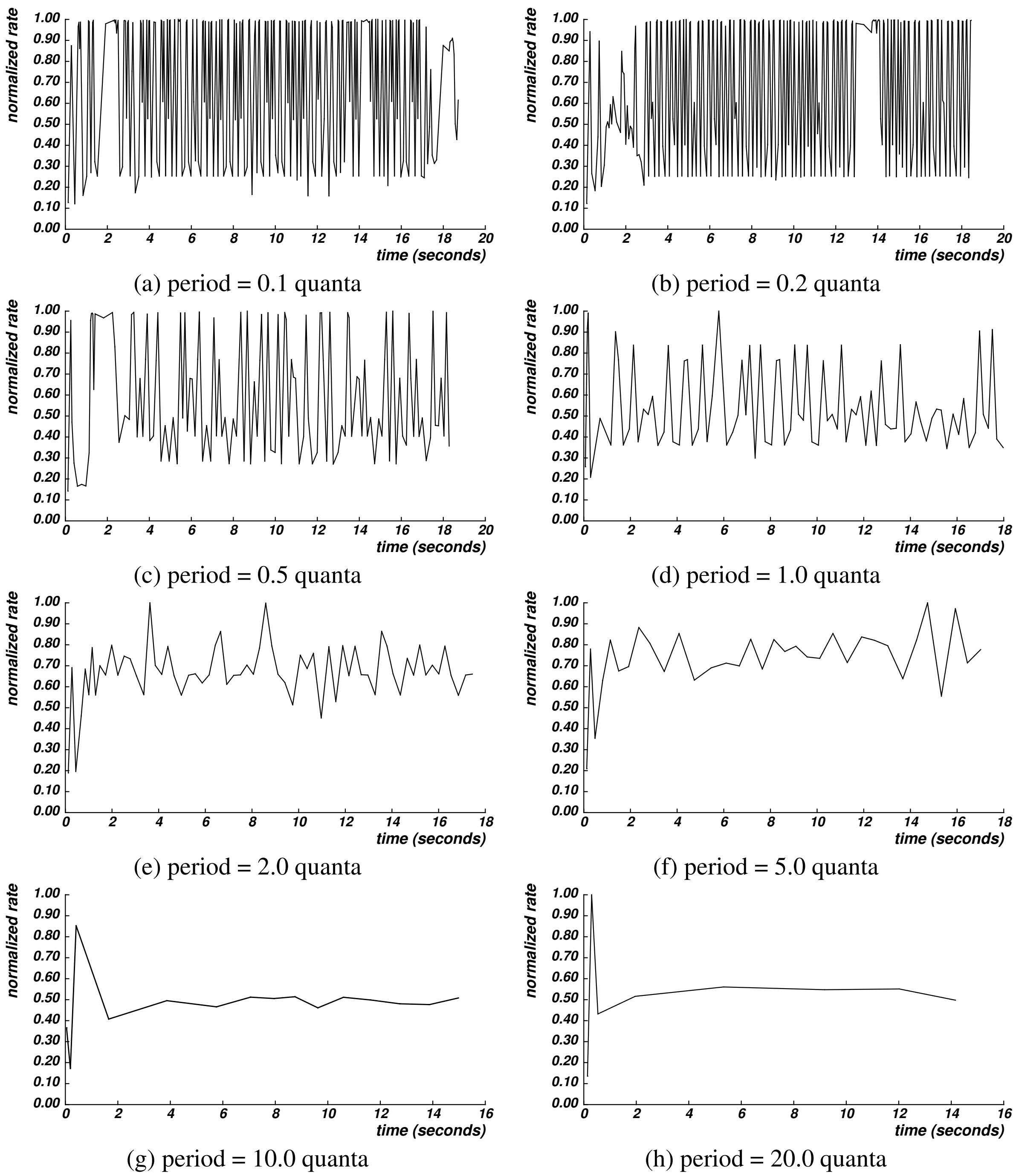
Figure 4.9: Scale factor for amplitude of oscillations for different sampling periods.

the time quantum or smaller, measured computation rates during one load balancing period will no longer be a useful measure for making estimates of the processing capabilities for later load balancing periods. The actual loads on the processors combined with intimate knowledge of how the loads will be scheduled by the operating system will give much better indications of performance in future load balancing periods. In this case, the load balancer will have to be aware of the context switching between processes and will have to do much more detailed analysis to determine what resources will be available at any given time. The load balancer will also need to be given measures of the relative computational capabilities of the processor hardware. To have access to the necessary information regarding scheduling, the load balancer may have to be integrated into the kernel of the operating system rather than running as a user process. This approach could be used on systems that use small time quanta as well, but the complexity of the approach is prohibitive. Fortunately, scheduling on Unix systems is based on small (100 millisecond) time quanta so that response times for interactive jobs will be acceptable [32]. The increasing trend in processor speeds indicates that if the length of time quanta will be changed in the future, it will be made smaller, not larger. Thus, we expect the lower bound on the period given by Equation 4.4 to be valid in most (if not all) workstation environments.

4.3.4 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:

$$period_{target} = \max(period_{interact}, period_{movework}, period_{scheduling}) \tag{4.11}$$



(a) period = 0.1 quanta
(b) period = 0.2 quanta
(c) period = 0.5 quanta
(d) period = 1.0 quanta
(e) period = 2.0 quanta
(f) period = 5.0 quanta
(g) period = 10.0 quanta
(h) period = 20.0 quanta

Figure 4.10: Effect of sampling period on stability of measurements. The rate of computation of matrix multiplication iterations is measured with a single competing computation intensive load on the 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.

For our measurements on the Nectar system, $period_{scheduling}$ was usually the maximum of the limits due to the small number of processors in the system. The processors in the system are Unix workstations with

100 millisecond time quanta. A measurement period of at least 10 time quanta (1.0 seconds) is necessary to reduce the apparent fluctuations on a stable loaded system to the extent that they will be ignored by the load balancer (Figure 4.10). Thus, a target load balancing period of approximately 1.0 seconds is usually selected, although the target period could increase if work movement costs become very high. Figure 4.11 shows the approximate values of each of the factors used in determining the load balancing period for our implementation on the Nectar system. The arrows indicate the approximate bounds on the load balancing period due to each factor: 5% overhead is allowed for interactions between the slaves and load balancer; work movement costs must be able to be 100% amortized over the period between detected fluctuations (which may be several load balancing periods); and the period must be at least 10 times the time quantum.

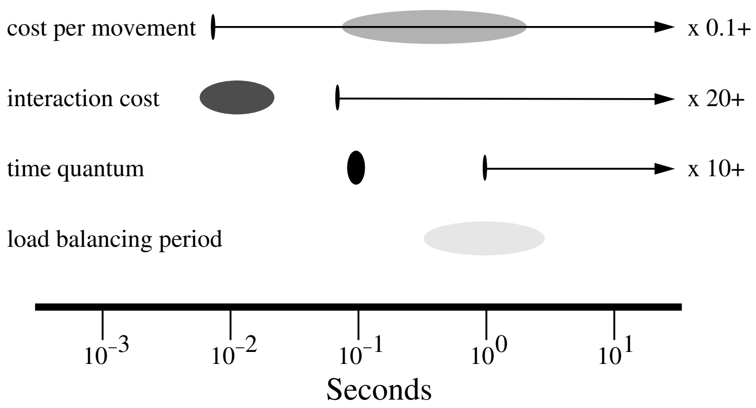


Figure 4.11: Lower bounds on load balancing period.

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. The amount of computation between executions of the load balancing hook may vary due to varying loop bounds in the application, and the elapsed time for the computations varies with the load on the processors. The number of computation phases per load balancing period must also change to track these variations. Thus, each time the load balancer is invoked, the average time for a computation phase, $period_{compute}$, is recomputed. Then the number of computation phases between calls to the load balancing code, *nexthook* (Figure 4.1), is computed using Equation 4.1:

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

If the iteration space is not rectangular, $period_{compute}$ will vary, even on a static system; if it is known how loop bounds vary (e.g., increasing or decreasing linearly), the computation of *nexthook* can be modified to compensate. New values of *nexthook* are sent to the slaves as part of their load balancing instructions. If hooks are placed between iterations of the distributed loop, $nexthook$ must be scaled appropriately for each processor so that all processors interact with the load balancer the same number of times and with the same frequency.

Because $period_{compute}$ varies, the actual load balancing period fluctuates around the target period. In our implementation on Nectar, to reduce the magnitude of the fluctuation, changes in $period_{compute}$ are damped using a recursive discrete-time filter called the *exponential smoothing forecast* [22]:

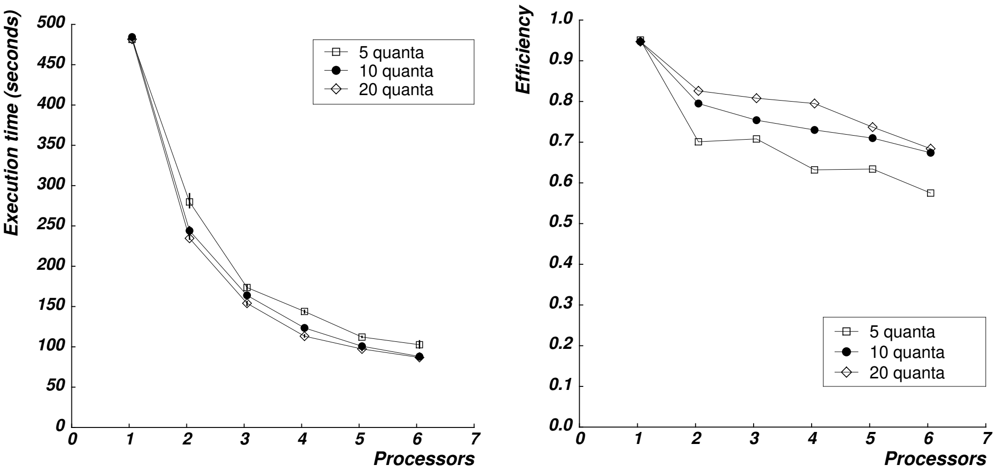
$$period'_{compute} = 0.5 \times period'_{compute} + 0.5 \times period_{compute} \tag{4.12}$$

The target and actual periods may also differ because *nexthook* must have an integral value, especially if $period_{compute}$ is a substantial fraction of or greater than $period_{target}$. Thus, it is important that the load balancing hook be placed so that it is executed as often as possible.

4.3.5 Effect of load balancing frequency on performance

The length of the period between load balancings has large effects on performance of applications with load balancing. Fluctuations in performance average out if the period is long enough, resulting in less work movement. Thus, increasing the load balancing period can improve performance as long as load balancing is still frequent enough to track significant changes in the computation rate. Figure 4.12 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.[2] The period is controlled by changing the value of *quantumscale* (Section 4.3.3), the dominant factor in selecting the period for our target environment. For most of the measurements presented in this thesis, a 1.0 second period (the middle curve) was used, to allow the system to be responsive to fluctuations in more dynamic environments; optimizations in the load balancing decision making process raise the efficiency close to the level attained with $quantumscale = 2.0$.

[2] 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. I.e., no optimizations are included over the basic load balancing system described in Section 2.5. The grain size for the application is selected automatically as described in Section 3.3.



a) Computation time b) Efficiency

Figure 4.12: Effect of load balancing period on efficiency for 1000 × 1000 SOR (40 iterations) with oscillating load (period = 60 seconds) on one processor.[2]

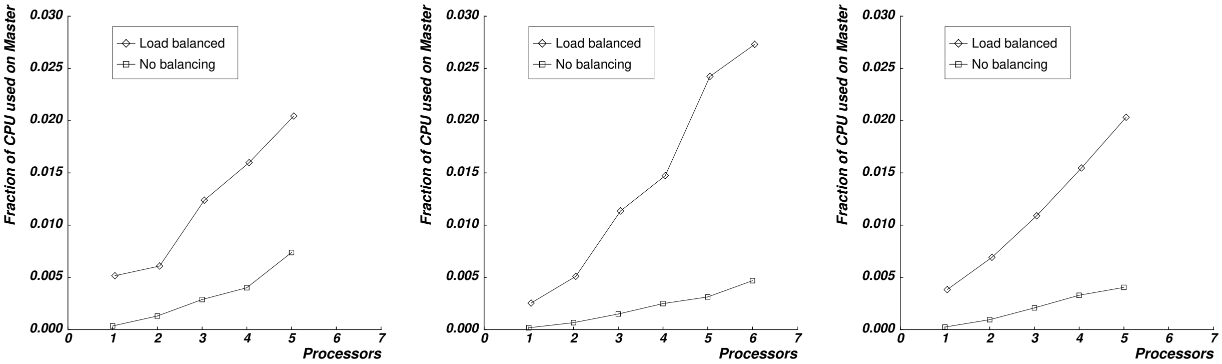
4.3.6 Effectiveness of frequency selection in limiting overhead

The cost of interaction with the central load balancer increases as the number of slave processors is increased. However, the frequency selection mechanism described above prevents the load balancer from becoming a bottleneck by limiting the interaction costs to a small percentage, 5%, of the total execution time. To evaluate the effectiveness of the frequency selection mechanism, we measure the CPU time used by the master process and divide it by the elapsed time for the application. This measure is a good indicator of the effectiveness of the frequency selection mechanism because it takes into account all of the load balancing computation costs and part of the communication costs for the load balancing interactions. (It does not include the transit time or the portions of the send and receive operations that take place on the slaves.) If the frequency selection mechanism is working correctly, the CPU used by the master process should be less than 5% of the elapsed time.

In Figure 4.13, the CPU usage by the master process is presented as a fraction of the elapsed time, with the lower limit on the load balancing period due to interactions with process scheduling set to 5 time quanta.[3] The CPU usage is measured using the *getrusage* function [12] provided with Unix. Receives on the master processor are done using interrupts, because, with polling, CPU usage by the master process

[3] 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. A high load balancing frequency is used and all portions of the load balancer are enabled, so the data should be on the conservative side.

expands to fill the time given.



a) dedicated system

b) constant load on one processor

c) oscillating load (period = 20 seconds)

Figure 4.13: Fraction of CPU used on master processor for 500 $\times$ 500 matrix multiplication.[3]

Figure 4.13 shows that load balancing uses only a small fraction of the available cycles—always less than 5%—for up to the maximum number of slaves in our system. In the graphs, the master CPU usage increases with the number of processors because interactions with the time quantum (Section 4.3.3) are the dominant factor in frequency selection for the small number of slaves used in the measurements. 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. Extrapolating from the data, a 5% overhead would be observed with about 10–12 slaves. At around that number of slaves, the load balancing interaction cost will become the dominant factor in frequency selection, and the period between load balancings will begin to increase as more slaves are added. The CPU usage will level off at about 5%, but eventually, as the number of processors increases, the frequency will become too low for the system to be responsive to trackable changes in the processing rates. At that point, distributing the load balancer will be necessary to keep the system responsive. However, Figure 4.12 indicates that the load balancing frequency can still be reduced substantially (e.g., from a period of 5 quanta to a period of 20 quanta) with beneficial results.

Also, the similarity of the three graphs in Figure 4.13 indicates that the fraction used by the master process is affected only slightly by the loads on the slave processors. In our Nectar implementation, used for the measurements in the graphs, the computation done by the load balancer is mostly independent of the loads on the slaves. The main exception is the case where no imbalance is detected and the generation of load balancing instructions can be skipped; however, the fixed portions of the load balancing computation (that depend on the number of slaves but not their loads) are the dominant cost.

As explained in Section 2.6.1, pipelining of the interactions takes most of the costs of interacting with the load balancer out of the critical path for the application. This does not affect the CPU usage by the load balancer on the master processor—in fact, the data in Figure 4.13 is for pipelined load balancing—but it can make the system immune to competing loads on the master processor, as long as the load balancing period is long enough that both the load balancing and the competing applications on the master all get a share of the CPU during the period. The lower bound on the load balancing period due to interactions with scheduling (Section 4.3.3) should ensure that this is true for reasonable loads (e.g., less than 5 competing processes for a 5 quantum lower bound) on the master processor.

4.4 Summary

This chapter described selection of an appropriate frequency for load balancing and described how the compiler and runtime system cooperate in controlling the load balancing frequency. The compiler places a load balancing hook as deep in the loop nest as possible without causing substantial overhead, and the runtime system selects which executions of the hook call the load balancer, based on a target load balancing period.

The target load balancing period is selected to minimize load balancing overhead and to minimize the effects of scheduling by the operating system on measurements of performance on the slaves. The target period is the maximum of lower bounds set by the cost of interactions between the slaves and load balancer, the cost of work movement, and the time quantum used by the operating system. Because work movement costs are distributed over several load balancing periods and often result in better resource utilization, the costs are scaled so that they rarely affect the selection of the load balancing period. Analysis of the interactions between scheduling by the operating system and observed computation rates was used in selecting the lower bound on the target period due to the time quantum. For an environment with a small number of slaves, the lower bound due to the time quantum determines the target period. As the number of slaves increases, the cost of interactions between the slaves and load balancer increases, and the target period is reduced to keep load balancing overhead at a desired level. This adaptation limits the interaction overhead and prevents the central load balancer from becoming a bottleneck, but increases the response time of the system; for systems with very large numbers of processors, the load balancer should be distributed to reduce the interaction costs. Measurements showed that the automatic frequency selection approach is effective in keeping the overhead of load balancing interactions low and in reducing the load balancing system's response to performance fluctuations due to scheduling of processes by the operating system.

Chapter 5

Load balancing process

The responsibility of the load balancer is to collect performance information from the slaves and to generate instructions for the slaves for redistributing remaining work so that computation times on the slaves are balanced. Each time the load balancer is invoked, it computes a new work distribution that allocates work to the processors in proportion to their available processing resources. The available processing resources for each processor are specified as a *computation rate*, in work units computed per second, so that heterogeneous processors with varying resource availability can be compared on an equal basis. Using the rate information, information about the amount of remaining work currently allocated to each slave, and information about constraints due to the application and environment, the load balancer generates an ordered set of work movement instructions for each slave. Each instruction specifies the slave to send to or receive from and the quantity of work to move.

5.1 High-level design

The decision making process used by the load balancer is shown in Figure 5.1. Upon receipt of performance information from the slaves, the load balancer evaluates the imbalance in the system using a threshold function. If the imbalance is significant enough to warrant work redistribution, the raw rate information from the slaves is filtered to reduce the effects of undesirable fluctuations in the measurements and a new, optimal distribution is computed. Work movement instructions are generated based on the new distribution. The current distribution is subtracted from the optimal distribution to determine the quantity of work

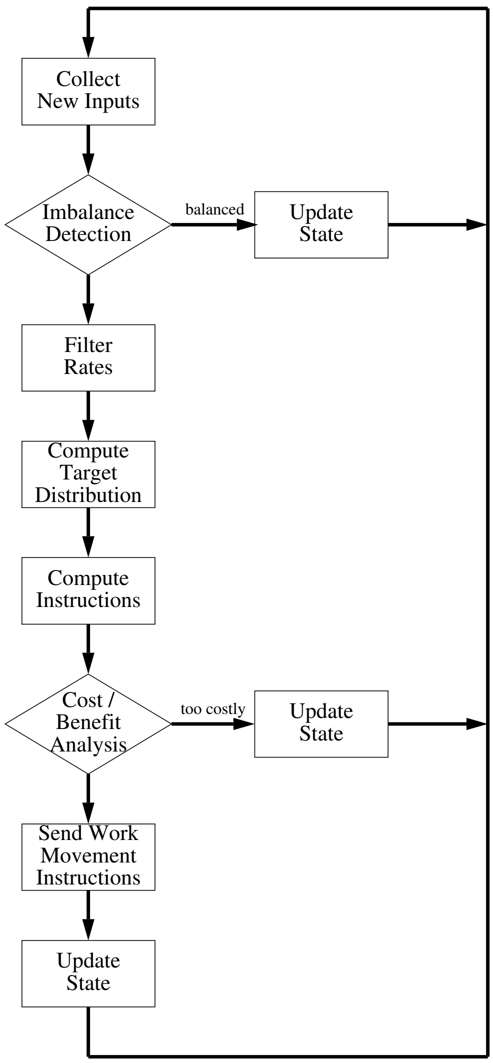


Figure 5.1: The load balancing decision process.

each slave should move, and instructions are generated by pairing senders of work with receivers using algorithms that take into account work movement constraints, while attempting to minimize communication costs. Finally, estimated costs of executing the instructions, i.e., the costs of moving the specified quantities of work, are compared to estimated benefits to determine whether the instructions should actually be sent to the slaves. The internal state of the load balancer includes history information regarding past computation rates and past work movement and the load balancer's idea of the current distribution of work.

5.2 Computing the optimal distribution

Given the measure of available resources—the computation rate—for each processor and the total number of work units distributed among the processors, we can compute the "optimal" distribution of work, where the work allocated to each slave is proportional to its contribution to the aggregate rate. This computation is used in quantifying the load imbalance in the system (Section 5.3) and in computing work movement instructions (Section 5.5).

To compute the optimal work distribution, w_i^{opt} ($0 \leq i < P$, where P is the number of slaves), the load balancer first sums the computation rates, r_i, provided by the slaves and divides the portion contributed by each slave by the sum, R, to determine the fraction of the total performance that each slave is expected to provide during the next load balancing period. Each slave's fraction is multiplied by the total number of iterations to be computed in the next computation phase to determine the number of iterations that should be allocated to that slave. The total number of iterations, W, is computed from the current distribution, w_i.

$$R = \sum_{i=0}^{P-1} r_i \tag{5.1}$$

$$W = \sum_{i=0}^{P-1} w_i \tag{5.2}$$

$$w_i^{opt} = \frac{r_i}{R} \times W \tag{5.3}$$

Because the work units, iterations of the distributed loop, are atomic, the results, w_i^{opt}, must be converted to positive integers, but the results still must be consistent with

$$W = \sum_{i=0}^{P-1} w_i^{opt} \tag{5.4}$$

5.3 Imbalance detection

The slaves are instructed not to move work if redistributing work into the optimal distribution can not reduce the projected execution time by a specified *threshold fraction*, *tfract* (e.g., 0.1, a 10% reduction). This provides the system with hysteresis so that small performance fluctuations do not cause work to be moved back and forth between processors.

The threshold check, which is done based on the raw performance measurements from the slaves, determines the fraction by which the elapsed time for the computation would be reduced if the assessments of performance for the slave processors match 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 (synchronous/pipelined load balancing) or no instructions (asynchronous load balancing) are sent to the slaves. Otherwise, the generation of instructions continues.

In our system, we set *tfract* to 0.1 (10%). Although the choice was somewhat arbitrary, there is also some intuition behind it. In Section 4.3.3, we set a lower bound on the target load balancing period so that amplitude of oscillations in performance due to scheduling would be less than 10% of the maximum possible amplitude; with the threshold fraction set at 10%, these limited amplitude oscillations are unlikely to result in work movement.

5.3.1 Quantifying load imbalance

To compute the reduction fraction, elapsed times for execution of the work units are estimated for the current distribution and for the optimal distribution. For either distribution, the last processor to finish determines the elapsed time. Thus, the elapsed time is the maximum of the execution times for the individual slaves. However, for the optimal distribution, the load is balanced so all processors should be fully utilized and should take the same amount of time (within the computation time for one work unit):

$$t_{opt} = \max_{i \in P} \frac{w_i^{opt}}{r_i} = \frac{W}{R} \pm \epsilon \tag{5.5}$$

When W is large and the computation time of a single unit is small, $\epsilon \to 0$, and computation of the optimal distribution can be omitted when determining the load imbalance. With the current distribution, the elapsed time is

$$t_{curr} = \max_{i \in P} \frac{w_i}{r_i} \tag{5.6}$$

The reduction fraction is computed as follows:

$$\textit{rfract} = \frac{t_{curr} - t_{opt}}{t_{curr}} \tag{5.7}$$

rfract is computed based on the raw performance measurements received from the slaves, rather than on the measurements after filtering. If, instead, the filtered measurements were used to compute *rfract*, the low-pass filtering used to reduce undesirable fluctuations would also tend to reduce *rfract*, and *tfract* would have to be reduced to compensate. We selected the former ordering because, when tuning our system, we found that performance was slightly better with *rfract* based on the raw measurements. In our system, both *tfract* and the filtering parameters were selected intuitively and empirically. Further analysis is needed to select ideal parameters for imbalance detection and filtering and to determine the ordering of filtering and imbalance detection that is most effective.

5.3.2 Effect of imbalance threshold on performance

Figure 5.2 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 5.2 demonstrates that the threshold (along with other optimizations[1]) prevents work movement from occurring in this case: after an initial period of instability, the work allocated to the processor remains constant.

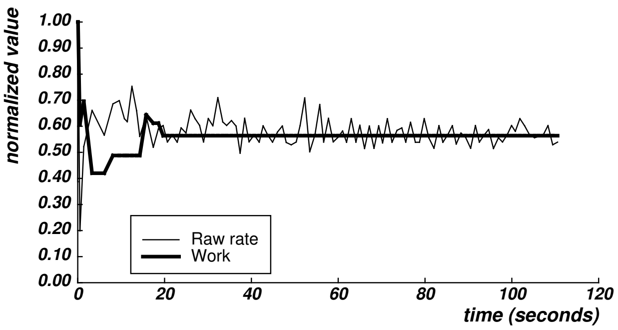
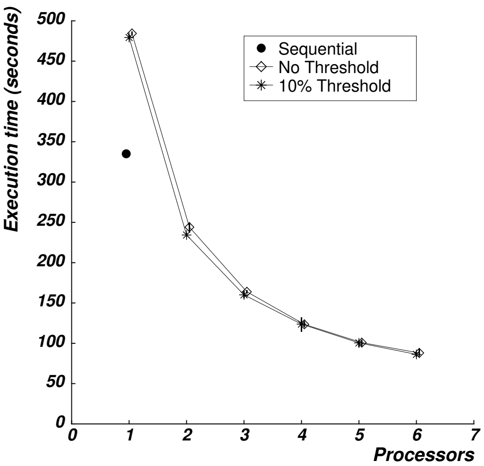


Figure 5.2: Measured performance and resulting work allocation on loaded slave for 1000 × 1000 SOR (40 iterations) running on a 4 slave system with a constant computation-intensive load on one slave.[1] The imbalance threshold helps reduce work movement in response to small fluctuations.

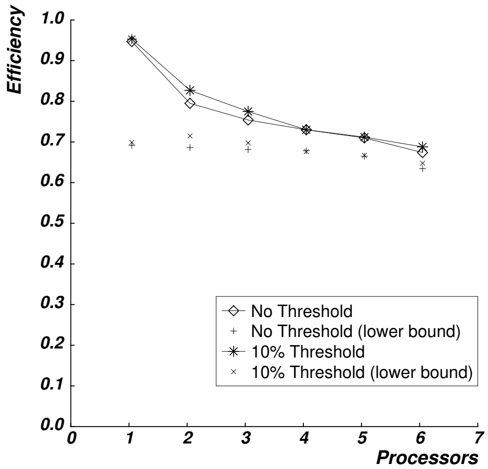
We executed the SOR example with an imbalance threshold of 10% (an anticipated 10% improvement is required for redistributing work) and without an imbalance threshold (work is redistributed whenever the

[1] 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.

observed computation rates change). The load balancing parameters were selected to isolate the effects of the imbalance threshold.[2] Figure 5.3 shows that using a threshold fraction to detect load imbalance improves efficiency in some cases, but not in others. Figure 5.4 shows that the goal of eliminating movement of small amounts of work is achieved: in Figure 5.4a the work allocation curve is smooth, but in Figure 5.4b work is moved in larger chunks. However using the threshold sometimes allows the system to remain unbalanced, as in Figure 5.4c.



a) Computation time



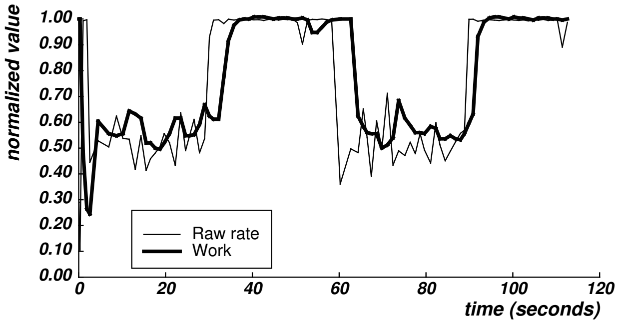
b) Efficiency

Figure 5.3: 1000 × 1000 SOR (40 iterations) running on 4 slave system with oscillating load (period = 60 seconds) on one slave. Effect of using threshold to detect load imbalance.[2]

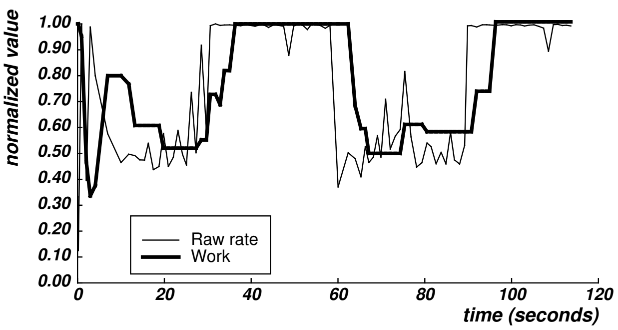
In some cases, using an imbalance threshold is not advantageous because imbalance at almost the threshold level may remain. In Figure 5.4b, the performance changes are such that work movement tracks performance well; the efficiency in this case was 77.7%. However, in Figure 5.4c, generated from a different run in the same environmental conditions, work movement does not track the computation rate as well. The 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 5.4c 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

[2] 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); raw (unfiltered) rate information is used; cost-benefit analysis is disabled. I.e., no optimizations are included over the basic load balancing system described in Section 2.5 other than the addition of the threshold check.

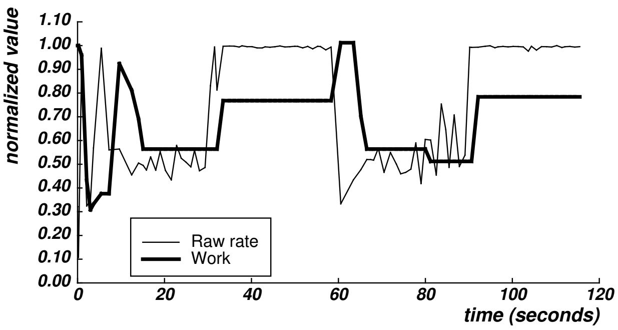
on one of the four processors in the system, so the total improvement would not exceed the 10% threshold. Figures 5.4b and 5.4c are based on data from two runs with the same parameters under the same conditions; the difference in how the load balancing system responded in the two runs is just due to random timing variations.



a) No improvement required.



b) 10% improvement required.



c) 10% improvement required.

Figure 5.4: 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.[1] The imbalance threshold reduces work movement but may result in suboptimal work distribution.

In Figure 5.3, the difference in performance is small between cases with and without imbalance detection, but there is still reason to believe that imbalance detection is beneficial. Figures 5.2 and 5.4b demonstrate that imbalance detection provides the desired hysteresis, preventing small fluctuations from resulting in work movement; but Figure 5.4c identifies a deficiency in the approach used. Also, in measurements (data no longer available) using a higher load balancing frequency, we observed more substantial improvements with imbalance detection. 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 5.4c, might improve performance.

5.4 Filtering rate information

The raw measure of available processing resources on each slave is the number of work units computed per unit time during the most recent load balancing period. Depending on the frequency of measurement, this measure may include several effects: the load on the system; the context switching between processes; and normal periodic system activity. Of these effects, work movement should track only the load on the system. Also, it is undesirable to try to track short-term fluctuations in the load because the cost of moving work could exceed the benefits. Decreasing the load balancing frequency reduces many of the undesirable effects, but the frequency still must be high enough to track the load on the system. Much of the remaining instability in the measurement of available resources can be eliminated by filtering out the high frequency component of the raw measure. Thus, in computing a new work distribution, the raw measure is replaced with a weighted average of the raw computation rate and previous computation rate measurements. There is still no guarantee that the system will not make errors in redistributing work, but the averaging reduces the degree and impact of bad predictions.

We replace the raw computation rate with a simple filtering function that combines recent and old information:

$$r'_i = (1 - h) \times r_i + h \times r'_{i-1} \tag{5.8}$$

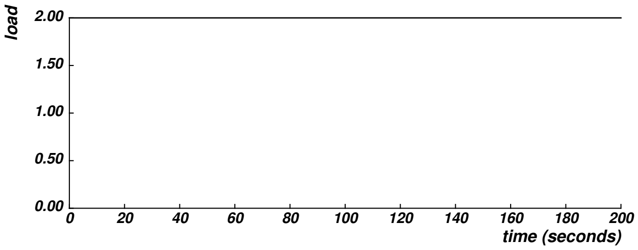
Like Equation 4.12, Equation 5.8 is a recursive discrete-time low pass filter called the *exponential smoothing forecast* [22]. r'_i is the filter output, the *adjusted rate* for the most recent computation phase, r_i is the raw rate for the most recent computation phase, and r'_{i-1} is the adjusted 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. A recursive filter was chosen over a nonrecursive filter because a recursive filter can include more history with fewer terms, thus requiring less computation time and less memory.

The selection of an appropriate value for h is a difficult task. We wish the filter to average out oscillations and short term fluctuations in the load, but we do not want the filter to delay the response to fluctuations for which tracking by the load balancer is profitable. Also, it is more important to respond quickly to decreases in performance than increases because a decrease in the performance of one processor causes all other processors to wait when the processors synchronize, while an increase in the performance of one processor has no effect on the productivity of the other processors. Thus, a constant value for h is inadequate to model the fluctuations that must be attenuated. Instead, a function that incorporates recent performance trends is used to compute the weights for the filter.

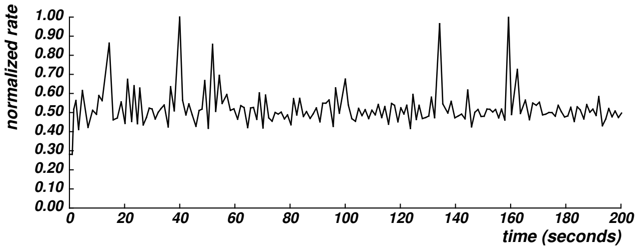
h can be arbitrarily complicated, ranging from a constant value to a function taking all previous measured values as inputs. With h equal to a fixed value, changes in performance are attenuated without regard to performance trends (Figures 5.5c and 5.6c); for some values of h, response is too slow, and for others, fluctuations in performance are not attenuated enough. A first step in addressing these problems is to use two values for h (Figures 5.5d and 5.6d): one for increasing performance, and one for decreasing performance. More weight is given to recent information if performance decreases than if performance increases because penalties are greater if the system moves work toward a processor erroneously, increasing idle time on all processors, compared with moving work away from a processor erroneously, only increasing the idle time on that processor. However, a single increase or decrease in performance does not constitute a trend; thus, we can not have much confidence in the two-valued h function. Therefore, we compute h using a state machine (Table 5.1) which encodes past trends in its state bits and takes the most recent information about changes in performance as input (Figures 5.5e and 5.6e):

$$(h_{next}, state_{next}) = f(input, state) \tag{5.9}$$

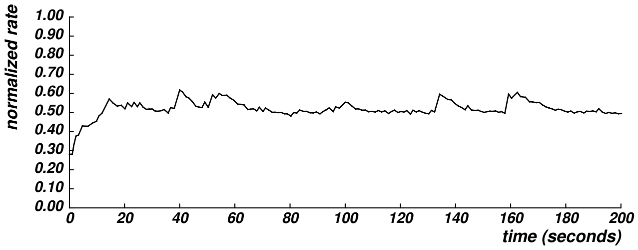
The state keeps track of the direction and duration of the performance trends. The state machine described in Table 5.1 uses 3 bits of state to store trend information from approximately 3 measurement periods. The amount of history incorporated in the trend information could be increased by increasing the number of state bits. As with the two-valued h function, in our state machine, we trust downward trends sooner than upward trends. New information gets greater weight when it is consistent with past trends and when there is confidence in the trends. The output values, h, in Table 5.1 are consistent with these specifications and were selected empirically. Figure 5.6 indicates that the filter based on the state machine responds more quickly to real load changes than the simpler filters, while still eliminating or attenuating most of the undesirable fluctuations. In the case of a processor with constant load, the attenuation of undesirable fluctuations is slightly less with the filter based on the state machine than with the other filters, but the state machine filter eliminates some fluctuations completely because it considers longer term trends (Figure 5.5).



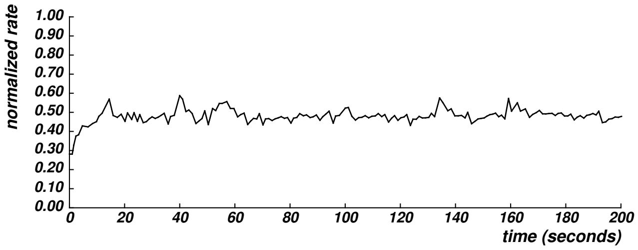
(a) total load on processor (including application)



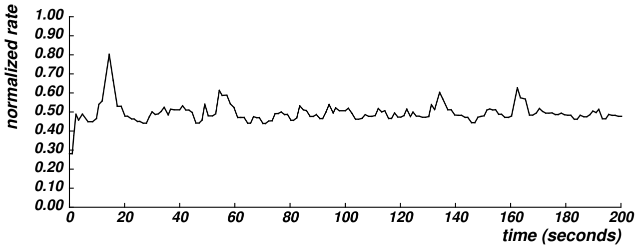
(b) raw rate



(c) adjusted rate: $h = 0.8$

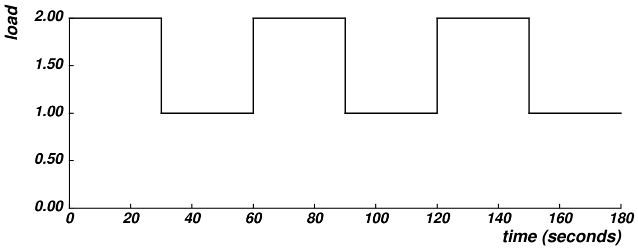


(d) adjusted rate: $h_{increase} = 0.8$, $h_{decrease} = 0.2$

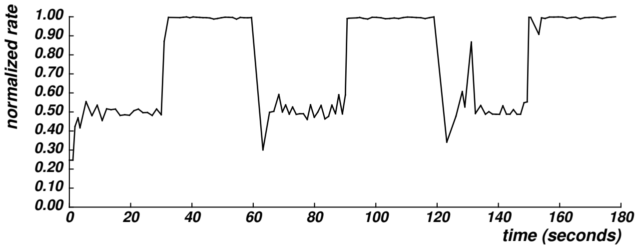


(e) adjusted rate: h determined by state machine

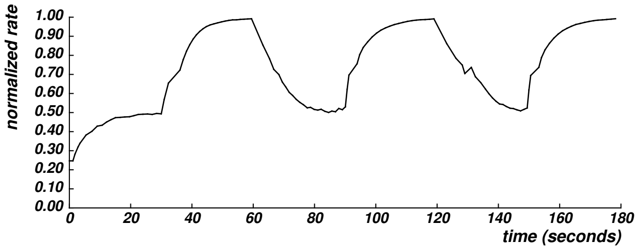
Figure 5.5: Performance assessment for a constant competing load. Target load balancing period is 1.0 seconds. Raw rate (b) is used as input to filters in (c), (d), and (e).



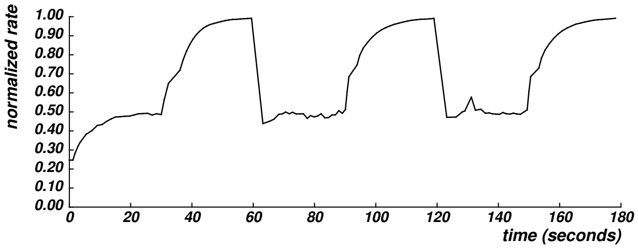
(a) total load on processor (including application)



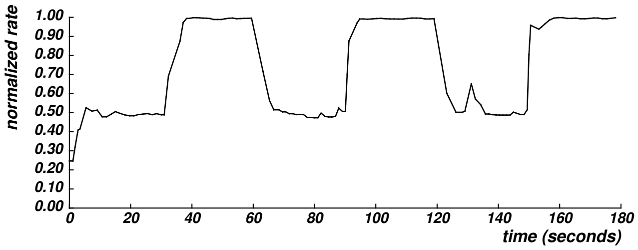
(b) raw rate



(c) adjusted rate: $h = 0.8$



(d) adjusted rate: $h_{increase} = 0.8$, $h_{decrease} = 0.2$



(e) adjusted rate: h determined by state machine

Figure 5.6: Performance assessment for an oscillating competing load with 60 second period. Target load balancing period is 1.0 seconds. Raw rate (b) is used as input to filters in (c), (d), and (e).

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 5.1: 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.

Because filtering of rate information is done independently for each slave, the filtering computations can be performed either on the slaves or on the master. If performed on the slaves, the computations are distributed, but remain in the critical path for the application. If performed on the master, the computations for the different slaves are performed sequentially, but can be removed from the critical path by pipelining the load balancing interactions. Since pipelining is used and the load balancer is not a bottleneck for the small number of processors in our target system, the filtering computations are performed on the master for our implementation on Nectar.

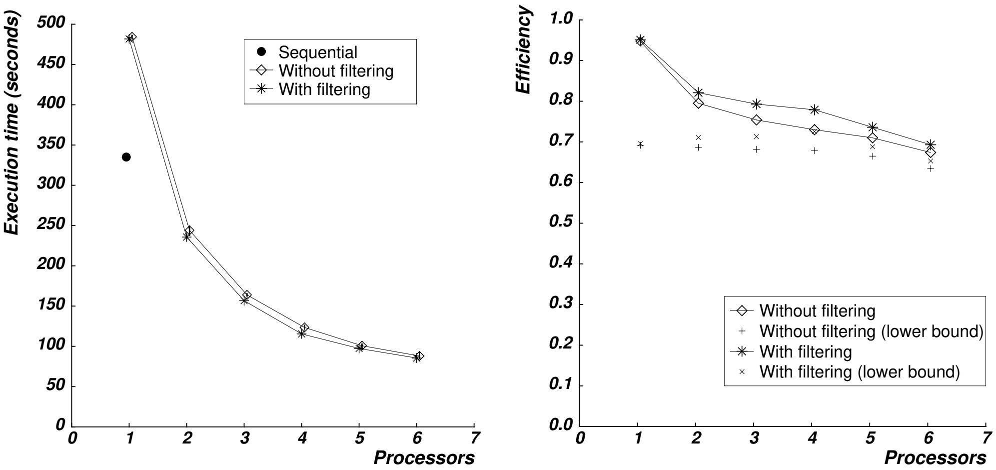
5.4.1 Effect of filtering on performance

Figure 5.7 shows how the filtering, based on Equation 5.8 and the state machine described in Table 5.1, improves the efficiency of a load balanced program. The load balancing parameters were selected to isolate the effects of filtering.[3] The run in Figure 5.8a 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 5.8b, a run without filtering, the system responds

[3] 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. I.e., no optimizations are included over the basic load balancing system described in Section 2.5 other than the addition of filtering.

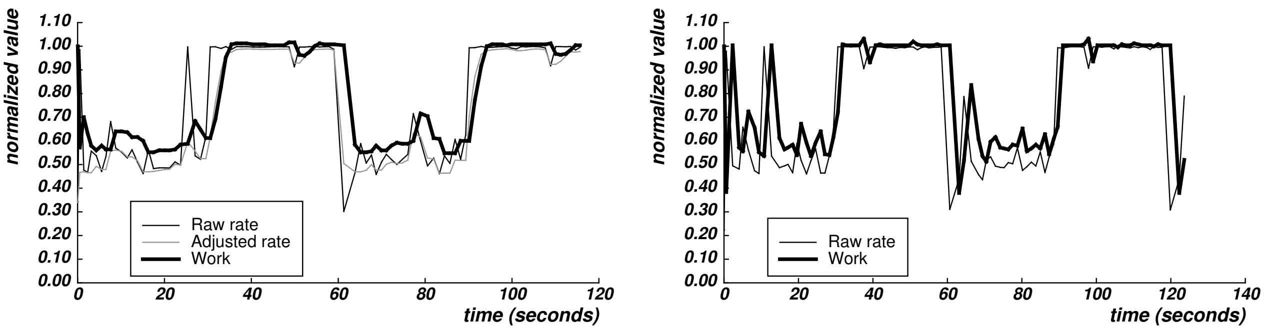
to all fluctuations on the loaded processor, and the work allocation curve has the same shape as the raw computation rate curve; the efficiency is lower due because there is more unnecessary work movement.

The effect of filtering on efficiency is very similar to that of increasing the load balancing period (see Figure 4.12) 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.



a) Computation time b) Efficiency

Figure 5.7: 1000 × 1000 SOR (40 iterations) running with oscillating load (period = 60 seconds) on one slave. Effect of filtering of rate information on efficiency.[3]



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

Figure 5.8: 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.[3] Low pass filtering reduces work movement in response to short term performance fluctuations.

5.5 Instruction generation

If load imbalance has been detected, the optimal distribution is computed using the filtered rates and is compared to the current distribution to determine which processors have too much work and which have too little. The difference between the optimal and current distribution is the amount of work that must be moved to and from the various processors. Instruction generation matches processors that need to offload work with processors that can handle more work. Instructions are generated for both sending and receiving processors. Separate instruction generation algorithms are needed for work movement restricted by dependences in the application and unrestricted work movement. Because the cost of the computation by the load balancer affects load balancing frequency selection and thus limits the responsiveness of the system (Sections 4.3.1 and 4.3.6), we include analysis of time complexity with respect to the number of processors in our discussion of the algorithms.

5.5.1 Unrestricted work movement

To minimize the cost of work movement, work should be transferred in as few messages as possible. A search could be used to determine how to move the data in the smallest number of instructions, but as the number of slaves is increased, searching would take too much time. (The problem of mapping work movement requirements to instructions is very similar to the *bin packing* problem, which is known to be NP-hard [14].) Also, the input information from which the optimal distribution is computed is not precise, so attempting to track it exactly is unlikely to be worth the added costs. Instead we use a greedy algorithm, Algorithm 5.1, that matches the processors that need to send the most work with the processors that need to receive the most work. To reduce communication costs, instructions for moving smaller amounts of work may be omitted. An example of application of the algorithm is shown in Figure 5.9.

Time complexity. In Algorithm 5.1, if the sender and receiver lists are implemented as a binary heap, the sorting of the lists can be performed in $O(P \log P)$ time; the **Next** routine, which extracts the maximum value, can be performed in $O(\log P)$ time; and the **Insert** routine can be performed in $O(\log P)$ time [14]. With each iteration of the **while** loop, **Next** is called twice and **Insert** is called at most once; either a sender or a receiver (or both) is removed from consideration. Therefore, since the total number of senders and receivers is P, the total number of iterations of the loop is at most P, and the time complexity of the whole

Algorithm 5.1: **Unrestricted work movement.**

Input: Set of P processors, numbered from 0 to $P-1$, with each processor's current allocation of work and the quantity of work to move to or from each processor.

Output: Ordered set of instructions for each processor. Each instruction specifies sender, receiver, quantity to send, and destination port.

Assumptions: Each originator has separate input port on receiver. Limited number of input ports on receivers.

Method:

1. Separate processors into *senders* and *receivers*. $O(P)$
2. Compute fraction of work each sender must move. $O(P)$
3. Sort *senders* by fraction of work to move (largest to smallest). $O(P \log P)$
4. Sort *receivers* by quantity of work to move (largest to smallest). $O(P \log P)$
5. Loop through receivers and match with senders. $O(P \log P)$

   Next(*structure***)** returns and removes the maximum element from the specified structure (*senders* or *receivers*).

   Insert(*structure***,** *processor***,** *quantity***)** inserts a processor and quantity of work to move into the appropriate location in the sorted structure.

   WorkToMove(*processor***)** returns the quantity (unsigned) of work the specified processor must move.

   CreateInstructionPair(*sender***,** *receiver***,** *quantity***)** adds an instruction to the instruction lists for the specified sender and receiver, allocating an input port on the receiver.

   FreePorts(*receiver***)** returns the number of unallocated input ports for the given receiver.

```
while ((r = Next(receivers)) != NULL) {
  if (FreePorts(r) == 0) continue;
  s = Next(senders);
  rqty = WorkToMove(r);
  sqty = WorkToMove(s);
  if (rqty > sqty) {
    CreateInstructionPair(s, r, sqty);
    Insert(receivers, r, rqty - sqty);
  }
  else if (sqty > rqty) {
    CreateInstructionPair(s, r, rqty);
    Insert(senders, s, sqty - rqty);
  }
  else {
    CreateInstructionPair(s, r, rqty);
  }
}
```

algorithm is $O(P \log P)$.

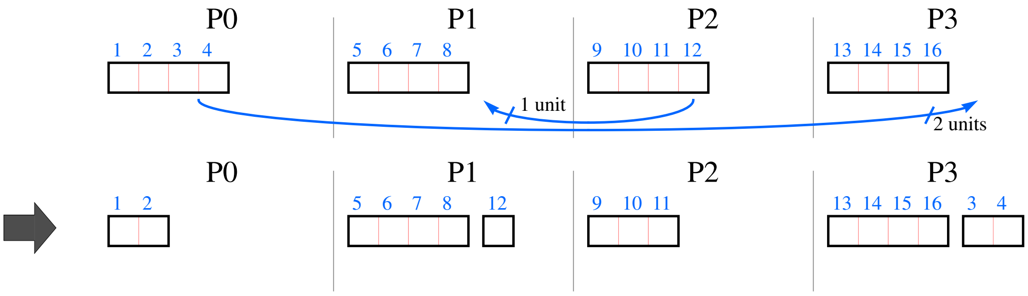


Figure 5.9: Unrestricted work movement using Algorithm 5.1.

Implementation concerns. To minimize the number of times data is copied, our implementation allocates an input buffer on the receiver for each instruction; this prevents work movement messages for carrying out different instructions from becoming interleaved. Due to limited memory on the communication processors, the number of input buffers is limited. Thus, in Algorithm 5.1, no additional instructions to receive work are generated for a processor once all its buffers have been allocated. Also, an acknowledgement must be sent to the load balancer when a receive buffer is no longer in use so that the load balancer can return the buffer to the free pool. The acknowledgements are received by the *FreePorts* routine. Fortunately, the instructions that are not generated due to the limited number of buffers are likely to be those that transfer the least work because the instructions that transfer the most work are generated first. However, if a processor that has very few work units slows down further, it may have difficulty offloading more units. The resulting distribution is still likely to approximate the optimal distribution.

5.5.2 Restricted work movement

Loop carried dependences in a distributed loop require communication between work units if the dependences cross processor boundaries. To minimize the amount of communication required for applications that have loop carried dependences (DOACROSS loops), a block partitioning is maintained by restricting work movement. Load balancing instructions are constructed so that work is only moved between logically adjacent processors, i.e., processors that store adjacent portions of the distributed data. Having a limited number of input buffers on each processor is not a problem because each processor receives work from a limited number of processors (for a one-dimensional distribution, at most 2 processors). However, if work must be moved between non-adjacent processors to balance load, *intermediate processors*, all the processors on the path between the non-adjacent processors, must be involved in the work movement.

Because work only moves between adjacent processors, no searches or heuristics are required to generate the most efficient instructions. Instructions are generated using Algorithm 5.2, a simple, straightforward, $O(P)$ complexity algorithm. Figure 5.10b shows the result of application of Algorithm 5.2 for an application with loop-carried dependences. Figure 5.10a demonstrates that use of Algorithm 5.1 for applications with dependences is not practical due to the added communication it may cause for the application.

Algorithm 5.2: **Restricted work movement.**

Input: Set of P processors, numbered from 0 to $P-1$, with quantity of work to move to or from processor.

Output: Ordered set of instructions for each processor. Each instruction specifies sender, receiver, quantity to send, and destination port.

Assumptions: Each processor can only exchange work with processors to its left and right.

Method: $O(P)$

WorkToMove(*processor***)** returns the quantity of work the specified processor must move (positive if receiving, negative if sending).

CreateInstructionPair(*sender***,** *receiver***,** *quantity***)** adds an instruction to the instruction lists for the specified sender and receiver. For each sender/receiver pair the same receiver input port is always used.

```
surplus = 0;
for (i = 0; i < P; i++) {
  delta = WorkToMove(i);
  delta -= surplus;
  if (delta > 0) {          /* receive from right.      */
    CreateInstructionPair(i+1, i, delta);
  }
  else if (delta < 0) {     /* send to right.           */
    CreateInstructionPair(i, i+1, delta);
  }
  surplus = -delta;
}
```

While instruction generation is simplified by adjacency constraints, instruction ordering is made more complicated by the fact that an intermediate processor can be both a sender and receiver of work. To minimize work movement time, parallel work movement must be maximized, but, at the same time, deadlock can not be allowed to occur. To maximize parallelism, each processor's instructions are ordered as follows:

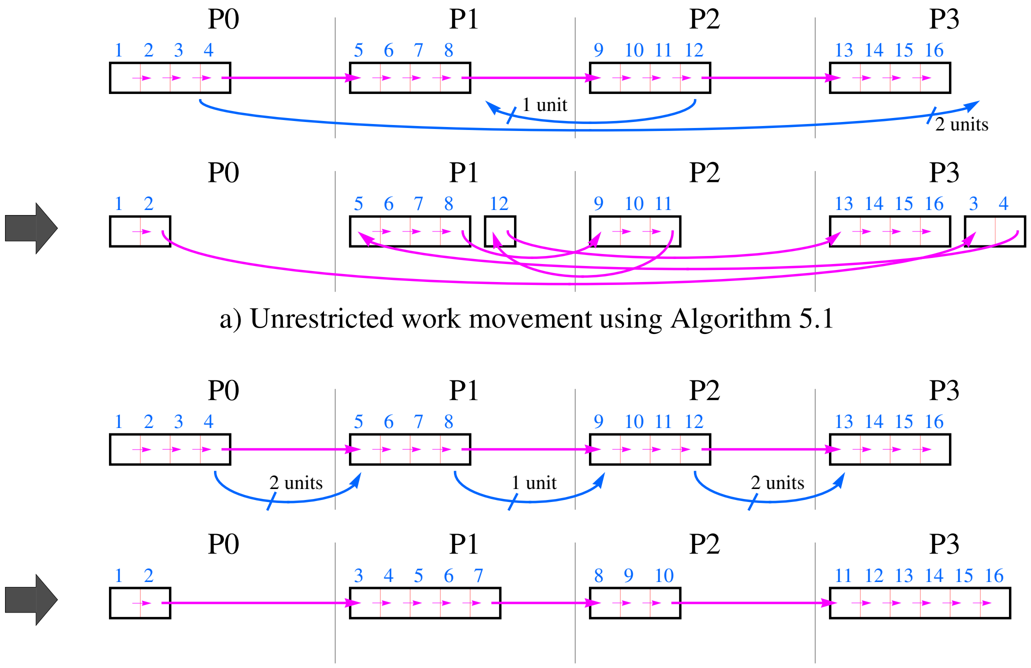
Instructions	Instruction 1	Instruction 2	net change
0	-	-	none
1	Send left	-	loss
1	Receive right	-	gain
1	Send right	-	loss
1	Receive left	-	gain
2	Send left	Receive right	gain/none/loss
2	Send left	Send right	loss
2	Receive right	Receive left	gain
2	Send right	Receive left	gain/none/loss

Table 5.2: All possible ordered sets of instructions sent to each slave for restricted work movement.

1. Send to left.
2. Receive from right.
3. Send to right.
4. Receive from left.

Each slave receives at most 2 instructions, and only the limited sets of instructions listed in Table 5.2 are possible. Parallelism is obtained in the case where work passes through intermediate processors (e.g., Figure 5.10b): the processors that must send work (e.g., to the right) can all send at the same time; then the processors that must receive the work (from the left) can all receive at the same time. Assuming that there is adequate buffering and flow control between processors, this ordering prevents deadlock because there is always at least one processor in the system that can proceed. If there is inadequate buffering, senders must block when the receive buffers are full, requiring flow control between the processors. If the system does not provide adequate flow control, explicit handshaking, possibly mandating serial execution, may be required to prevent loss of data. (Unfortunately, this was the case for Nectar, our target system.)

Another difficulty with the above ordering is that intermediate processors may be required to move more work than they initially own. In this case, the processors must receive work before they can send it. When a slave receives an instruction to send more than it owns, the slave delays execution of the instruction until the corresponding receive instruction (which must exist) has been executed. This may result in loss of parallelism, but deadlock is avoided.



a) Unrestricted work movement using Algorithm 5.1

b) Restricted work movement using Algorithm 5.2

Figure 5.10: Load balancing of loop with dependences. Unrestricted work movement results in much more communication for application.

5.6 Profitability determination

Work should not be moved if the costs of moving the work exceed the benefits. The instruction generation algorithms (Section 5.5) consider the imbalance in the system but do not consider work movement costs. Although the use of thresholding (Section 5.3) and the filtering of raw performance information (Section 5.4) have reduction of work movement costs as their goals, neither explicitly considers the actual costs of work movement. To further ensure that work movement results in improved performance, we add a *profitability determination phase* (motivated by [72]) that explicitly considers the cost of executing the generated instructions. The phase estimates the costs and benefits of work movement and cancels the instructions if the costs exceed the benefits.

If it were possible to predict the exact costs of work movement and to predict the amount by which the time spent on productive computation will be reduced as a result of the movement, we could create precise models of program execution to decide when work movement is profitable (although the computation time for the decision could be quite high). However, because we can not predict the future, the best we can do is to estimate work movement costs and benefits based on past information. Erroneously cancelling work movement instructions can delay the load balancer's reaction to real performance changes in the system and reduce the effectiveness of load balancing. Therefore, since the estimates of work movement costs and benefits are based on inaccurate information, profitability determination is used only as a sanity check for the work movement instructions. Instructions are only cancelled if their estimated costs, $c_{movement}$ are *several* times their projected benefits, $t_{benefit}$, i.e., if

$$c_{movement} > k \times t_{benefit}$$

where k is a small number greater than or equal to 1. For our experiments where profitability determination was enabled, k was set to 5. With more accurate projections for costs and benefits, k could be reduced.

5.6.1 Estimating costs of work movement

The time to transfer work between processors can be measured before starting the actual computation. Work movement messages are sent back and forth between processors several times to compute average transfer times. The cost of transferring empty work movement messages is measured to determine the fixed cost of work movement, c_{fixed}, and the cost of transferring known amounts of work is used to determine incremental work movement costs, c_{incr} (Section 3.3.4). The cost of an individual instruction can then be estimated based on the amount of work being transferred. An estimate is needed for $c_{movement}$, the cost of all work movement in a load balancing phase. If multiple processors move work, some work movement may occur sequentially and some may occur in parallel. For the restricted and unrestricted work movement cases, the total work movement cost must be estimated in different ways. Complete analysis could be used to determine the critical paths for each of these cases, but simple estimates for the costs are sufficient due to the even greater difficulty and inaccuracy in predicting the benefits of work movement. Below we describe the estimates used in our implementation on Nectar. For other systems, other estimates may be more appropriate, depending on the topology of the communication network and the sharing of communication resources.

For unrestricted work movement, instructions involving different sets of processors can be executed independently, and much of the movement can occur in parallel. The cost of work movement is estimated based on the cost of movement to and from the processor which moves the most work:

$$c_{unrestricted} = instructions \times c_{fixed} + workunits \times c_{incr} \tag{5.10}$$

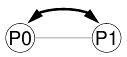
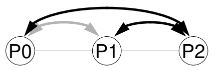
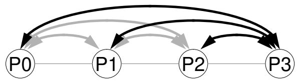
Processors		Added paths	Added hops	Total paths	Total hops	Hops per path
2		1	1	1	1	$1 = \frac{3}{3}$
3		2	3	3	4	$\frac{4}{3}$
4		3	6	6	10	$\frac{10}{6} = \frac{5}{3}$
P		$P-1$	$\sum_{i=1}^{P-1} i$	$\sum_{i=1}^{P}(i-1)$	$\sum_{j=1}^{P}\left(\sum_{i=1}^{j-1} i\right)$	$\frac{P+1}{3}$

Table 5.3: Derivation of average number of hops in a linear array of P processors. Each added processor, Pn, adds a path of length $n - i$ hops to each of the n other processors, Pi. ($n = P - 1$ because we label processors starting with 0.) The average number of hops between processors is computed by dividing the total number of hops for all paths between processors by the total number of paths.

For the restricted work movement case, we assume that no work movement occurs in parallel for the worst case estimate. (This is the case for our implementation on Nectar due to unreliable hardware flow control.) Thus, for restricted movement, the estimate for unrestricted movement is multiplied by an estimate of the number of intermediate processors that will be involved in the transfer. The average number of hops between two processors in a P-processor linear array is $\frac{P+1}{3}$. (The derivation of this result is outlined in Table 5.3.) Therefore,

$$c_{restricted} = \frac{P+1}{3} \times (instructions \times c_{fixed} + workunits \times c_{incr}) \tag{5.11}$$

$c_{movement}$ is either $c_{unrestricted}$ or $c_{restricted}$ depending on the instruction generation algorithm used for the application.

5.6.2 Estimating benefits of work movement

The benefit of redistributing work is the reduction in elapsed time for the computation due to elimination of idle time. If, after redistributing work, the rates of the processors relative to each other remain stable or continue to change in the same direction as the changes that prompted the redistribution, the benefits accrue over time. For example, if one processor slows down, the total computation rate is increased by moving work away from that processor; if the processor slows down further, the new distribution, while no longer optimal, is still an improvement over the original distribution. If the trends in computation rates continue long enough, the cost of work movement can be amortized, and there can be a reduction in the overall elapsed time. The potential benefits of work movement are limited by the end of program execution, but, in general, the time until the end of the program can not be predicted; fortunately, this limit can be ignored since the time to redistribute work one extra time is expected to be negligible relative to the total execution time of the program.

We estimate the time saved due to redistributing work as

$$t_{benefit} = bfract \times t_{stable} \tag{5.12}$$

where *bfract* is the expected benefit per unit time, and t_{stable} is the expected length of time over which benefits will accrue, i.e., the time period over which we expect system performance to remain stable. *bfract* is computed the same way as the reduction fraction, *rfract* in Section 5.3.1, but, in this case, the new distribution may not be optimal due to the constraints on instruction generation:

$$t_{orig} = \max_{i \in P} \frac{w_i^{orig}}{r_i} \tag{5.13}$$

$$t_{new} = \max_{i \in P} \frac{w_i^{new}}{r_i} \tag{5.14}$$

$$bfract = \frac{t_{orig} - t_{new}}{t_{orig}} \tag{5.15}$$

Estimating t_{stable} is more difficult because it is not possible to accurately predict future trends in system performance. However, by examining many traces of past performance it may be possible to recognize patterns in the loads on the system well enough to make a reasonable assessment of the system stability. For example, if we can recognize that load changes are periodic and identify the frequency of change, we can model the system to decide whether it is profitable to shift work in response to the changes. (We have modeled a system with an oscillating load on one processor in Section 7.5.) If the frequency of change is high, t_{stable} will be small. In this case, if the system attempts to move work to track each change, work movement costs will be high and will have little benefit. In this unstable system, better overall performance is likely if work movement does not track the performance fluctuations. However, if the frequency of change is low, t_{stable} will be larger, and moving work to track the changes will be less costly and more beneficial, resulting in a net improvement in performance.

Lacking extensive traces and with the unlikeliness of encountering truly periodic loads in real systems, we estimate system stability based on recent performance information collected during the current run of the application, assuming that there is temporal locality in the stability of the system. A record is kept of the number of times imbalance was detected over the last $windowsize$ load balancing phases. The total computation time for the $windowsize$ corresponding computation phases is divided by the number of times imbalance was detected (i.e., exceeded the imbalance threshold described in Section 5.3) to determine the period between significant changes in system performance. The period over which work movement accrues benefits, t_{stable}, is estimated at twice the period between changes, assuming that only half of the changes will be in unfavorable directions. This is a very rough estimate of stability, and the selection of $windowsize$ (to equal about 10 seconds in our implementation) is somewhat arbitrary.

5.6.3 Effect of Profitability Determination on Performance

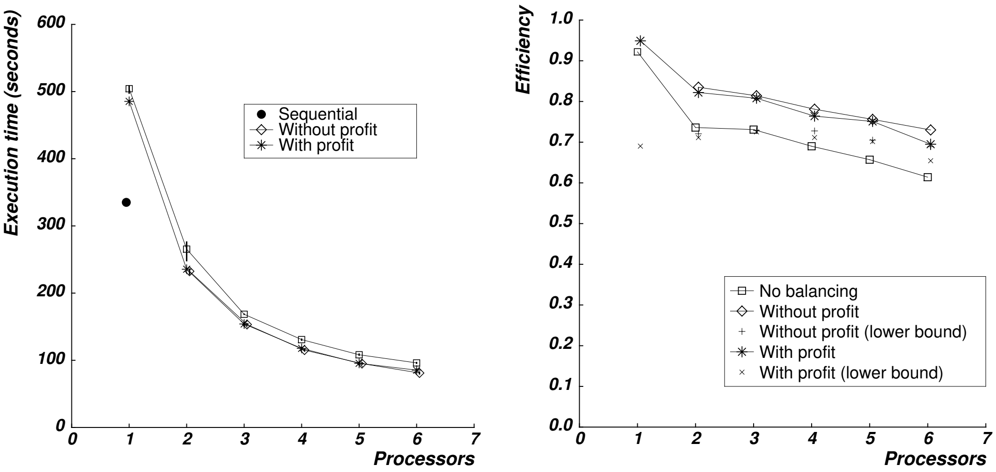
The profitability determination phase is designed to be most beneficial when work movement costs are high or when the loads on the processors are very unstable. For the 1000×1000 SOR example (Figure 5.11[4]), dynamic load balancing without the profitability determination phase results in improved performance relative to equal distribution of work so we know that tracking the loads on the processors is beneficial. Thus, we do not expect the profitability determination phase to improve performance. In fact, the cancelling of work movement instructions delays reaction to changes in load, resulting in a slight performance loss.

In Section 7.5.2, we will show a case where the profitability determination phase does substantially improve performance with dynamic load balancing.

5.7 Summary

In this chapter we described the operations that are performed each time the slaves interact with the load balancer. The load balancer generates instructions that redistribute work in proportion to measured computation rates on the slaves. For applications with loop-carried dependences, a block distribution must

[4] 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.



a) Computation time b) Efficiency

Figure 5.11: 1000 × 1000 SOR (40 iterations) running with oscillating load (period = 60 seconds) on one slave. Effect of addition of profitability determination phase on efficiency.[4]

be maintained to minimize the communication required for the application. Thus, for these applications, we use an instruction generation algorithm that restricts work movement so that work is only moved between logically adjacent slaves; to move work between non-adjacent slaves, intermediate slaves are involved. For applications without loop-carried dependences, communication required for the application is not a concern, so we attempt to minimize the cost of work movement using an algorithm that generates instructions to move work directly between the slaves that are overloaded and the slaves that can handle extra work.

Several optimizations are included in the load balancing process to prevent undesirable work movement. An imbalance detection phase, which compares a measure of the imbalance between the processors to a threshold, adds hysteresis to the system, eliminating the high per-message communication costs of moving small amounts of work back and forth between slaves. Before the new distribution is computed, the rate information from the slaves is filtered to reduce high frequency components, reducing the system's response to short term changes in load. Finally, after instructions are generated, a cost-benefit analysis is performed to check whether it is profitable to execute the generated instructions. All of these optimizations are effective in reducing work movement costs, but they also can delay the system's reaction to changes in performance, leaving the system unbalanced for longer than necessary; the inefficiency due to this imbalance sometimes outweighs the benefits of the optimizations. Selection of good parameters and implementations for the optimizations is difficult due to this tradeoff and due to the inability to accurately predict the loads on the processors; many of the parameters for the optimizations were selected empirically. More analysis is needed, and a good characterization of typical loads on the processors would be useful. Although good motivation for each of the optimizations has been presented, from the experimental results, it is difficult to make definitive claims regarding their performance benefits. However, for each optimization, we were able to demonstrate small performance benefits in some situations.

Chapter 6

Compiler support for load balancing

Our load balancing system consists of application-specific code generated by a parallelizing compiler and a run-time library that supports functions common to all applications, such as task creation, communication, and load balancing. In this chapter, we describe the changes that must be made to a typical parallelizing compiler to support dynamic load balancing. Other than the restructuring transformations needed to support control of grain size (Chapter 3) and load balancing frequency (Chapter 4) summarized in Table 6.1, the modifications needed to the compiler to support dynamic load balancing are in its code generation portion and require little analysis or restructuring of the program. These modifications will be the focus of this chapter. Although these changes have not yet been implemented in a compiler, hand-parallelized versions of the MM, SOR, and LU examples were used to motivate and to investigate the implementation details of the changes.

To support our dynamic load balancing system, the parallelizing compiler must generate code for both the master and the slave processes. The master process controls the central load balancer, and the slave

transformation	used for
strip mining	control of grain size control of frequency of load balancing hooks
loop splitting	removing communication code from distributed loops
message aggregation	reducing communication overhead
loop interchange	increasing grain size control of frequency of load balancing hooks

Table 6.1: Restructuring transformations.

processes perform the computation for the application and interact with the master process periodically for load balancing. For effective load balancing, the compiler must notify the runtime system which type of loop—DOALL or DOACROSS—has been distributed so that work movement can be restricted in the DOACROSS case (Section 5.5.2). The compiler is also responsible for generation of application-specific code to load input data, unload output data, and package and transfer portions of distributed data structures to shift work between the processors. In addition, to support work movement, modifications must be made to the distributed data structures, the distributed loop bounds, and the calls to the communication code.

The structure of the master and slave code is outlined in Section 6.1. Sections 6.2 and 6.3 describe changes to the distributed data structures and loop bounds needed to facilitate work movement. Section 6.4 discusses the routines that must be generated to transfer work between processors. Then we discuss how moved work must be handled so that it is consistent with the work on the destination processor. Finally, we discuss how the irregular data distributions resulting from work movement complicate location and communication of distributed data elements.

Parallelizing compilers usually translate a sequential source program into a parallel single-program multiple-data (SPMD) source program and use native compilers of the target machine to generate the actual object code [11, 62, 68]. In describing our system, we assume that the sequential source is written in a Fortran-like language with annotations to aid in parallelization (e.g., Fortran D [25] or AL [67]) and that the generated code has C syntax and semantics so that data structures can be moved around easily by manipulating pointers.

6.1 Code structure

The compiler must generate code for the master and slave processes. The structure of the slave code is the same as that of the sequential source code and of typical code generated by a parallelizing compiler: the parallelized slave code has the same loop structure as the source code, but loop bounds are modified to work on the subset of loop iterations allocated to the current processor. Additional code is added to support load balancing, but the loop structure is not changed significantly. Although the master process performs no useful computation, its loop structure must mimic that of the slave code to at least the depth of the load balancing hooks placed in the slave code so that the load balancing code is called the correct number of times and the application can terminate properly. The master code must have load balancing hooks placed in the same locations in the loop structure as in the slave's loop structure, although the hooks on the slave call client code and the hooks on the master call server code. Also, when there are loops with data dependent bounds, e.g., WHILE loops, above the nest level of the load balancing hooks, the master, like the slaves, must receive all data needed to test the loop conditions. This is not necessary at nest levels deeper than the load balancing hooks because the communication required by the application can not interfere with the load balancing communication.

In the SOR example, the outermost loop is a WHILE loop that tests error conditions and terminates when either the error is low enough or when a maximum iteration count has been reached. The master must keep track of the iteration count for the loop; and when a new error value is computed, it must be sent to the master as well as being broadcast to all slaves because the load balancing hook is in a lower nest level. Figure 6.1 outlines the master and slave code that would be generated for the SOR example.

```
iter = 0;
error = 1.0;
while ((error > zeta) && (iter < maxiter)) {



      for (i = 1; i < m - 1; i++) {


            master_hook();

      }

      receive error;
      iter = (iter + 1);
}
```

a) Master code

```
iter = 0;
error = 1.0;
while ((error > zeta) && (iter < maxiter)) {
      error = 0.0;
      if (pid != 0) send leftmost column to left;
      if (pid != pcount - 1) receive column from right;
      for (i = 1; i < m - 1; i++) {
            if (pid != 0) receive from left;
            compute local portion of row and error;
            slave_hook();
            if (pid != pcount - 1) send to right;
      }
      collect local error values and compute total error;
      if (pid == 0) send error to master;
      iter = (iter + 1);
}
```

b) Slave code

Figure 6.1: Code structure for master and slave processes for SOR.

6.2 Changes to distributed loop bounds and distributed data structures

Loops and data are distributed according to the owner-computes rule [24, 36]. That is, loop iterations are computed on the processor that owns (stores) the distributed data locations written to by the iterations. (Some data may be shared between loop iterations, and therefore must be replicated if read-only or communicated when updated if read-write.) Therefore, when we distribute loops, we also distribute the associated data, and vice versa.

Loops are distributed by modifying the bounds so that, on each processor, the bounds only include the iterations that are allocated to that processor. The most commonly used static distributions—*block*, *cyclic*, and *block-cyclic*—are regular, so the local set of iterations can be computed directly based on the processor number (p, assuming processors are numbered sequentially starting from zero), the number of processors (P), the original loop bounds (e.g., 0 to $n-1$), and a block size (Figure 6.2). (Note that block and cyclic distributions are equivalent to special cases of block-cyclic distributions where *blocksize* $= \lceil \frac{n}{P} \rceil$ and *blocksize* $= 1$, respectively. Sequential execution on a single processor is equivalent to a block-cyclic distribution with *blocksize* $= n$.) In all of these distributions, (approximately) equal amounts of work are allocated to all processors. If processors vary in performance, an irregular distribution is needed, and the local set of iterations can no longer be computed directly. The distributed data must be stored in a manner that makes both work movement operations (sending and receiving work units) and accesses to the data required by the application (Figure 6.3) inexpensive.

```
for (i = 0; i < n; i++) {
    compute iteration;
}
```

a) Original loop

```
blocksize = (n + P - 1) / P;
for (i = p * blocksize; i < (p + 1) * blocksize && i < n; i++) {
    compute iteration;
}
```

b) Block distribution

```
for (i = p; i < n; i += P) {
    compute iteration;
}
```

c) Cyclic distribution

```
for (i0 = p * blocksize; i0 < n; i0 += P * blocksize) {
    for (i = i0; i < i0 + blocksize && i < n; i++) {
        compute iteration;
    }
}
```

d) block-cyclic distribution

Figure 6.2: Common regular distributions.

```
a(k)(...) = ...
```

a) Access particular data slice

```
DO i = 0, n-1
    a(i)(...) = ...
END DO
```

b) Access each data slice

Figure 6.3: Sequential version of code used in comparing representations of irregular distributions.

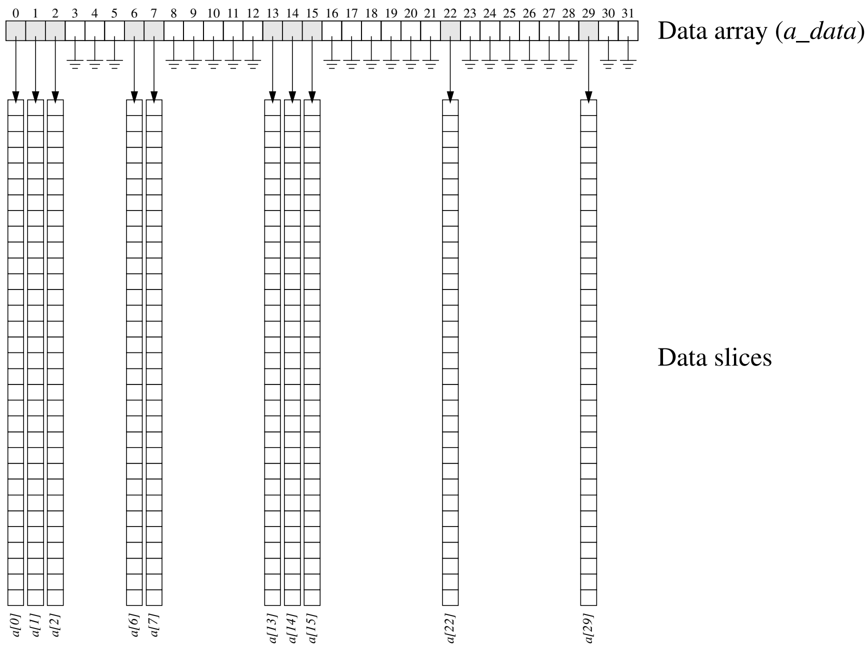
6.2.1 Basic data structure

There are many ways to implement irregular data distributions, such as arrays, linked lists, or tuples. In our application domain, since applications are parallelized by distributing portions of arrays, it is convenient to maintain the data in a structure based on arrays. To make shifting of distributed data slices easy and efficient, data is stored as arrays of pointers to slices of the distributed data structure (Figure 6.4a), rather than as contiguous areas in memory. The distributed data structure is stored in distribution-major order (e.g., row-major order or column-major order for a 2-dimensional matrix, depending on how the matrix is distributed). For example, since SOR is distributed by columns, the **B** matrix is stored in column-major order: in the sequential version of the program, an element of the matrix is referenced as $b[i][j]$, where i is the row number and j is the column number; in the parallel version, the same element is referenced as $b[j][i]$ because of the column major storage. Figure 6.4a shows the basic structure used to store a matrix. The array of pointers, which we call the *data array*, is large enough to point to all slices of the distributed data structure. Each processor has a local version of the data array, but since each processor only stores part of the distributed data structure, each element of the local version either points to a local *data slice* or is a null pointer.

The basic data structure described above facilitates work movement because each data slice is stored in a contiguous area of memory and can be copied efficiently for work movement. We call this data structure a scattered implementation because, on each processor, the local data slices are scattered throughout the data array. A slice of the data is local if its location in the data array contains a valid (non-null) pointer. Figures 6.4b and 6.4c show the operations required on a processor to send or receive a distributed data slice. Sending a slice stored in the scattered structure is inefficient because it requires searching through the data array for a valid pointer. Similarly, a loop that accesses all local data slices (Figure 6.4e) must step through the entire data array to locate the valid pointers. In the next section, we describe ways to augment and modify the basic data structure so that access to the structure is more efficient.

6.2.2 Efficient access to data

Because the distributions are irregular and may change at run time, it is not possible to determine which processors are responsible for particular loop iterations and distributed data at compile time. Therefore, we compare different data structures by how processors identify and access their local data slices. The data



a) Data structures

```
i = 0;
while (a_data(i) == NULL) i++;
send(a_data(i), len);
free_mem(a_data(i));
a_data(i) = NULL;
```

b) Send work

```
receive(idx);
a_data(idx) = alloc_mem(len);
receive(a_data(idx), len);
```

c) Receive work

```
if (a_data(k) != NULL) {
      a_data(k)(...) = ...
}
```

d) Access particular slice

```
for (i = 0; i < n; i++) {
      if (a_data(i) != NULL) {
            a_data(i)(...) = ...
      }
}
```

e) Access each slice

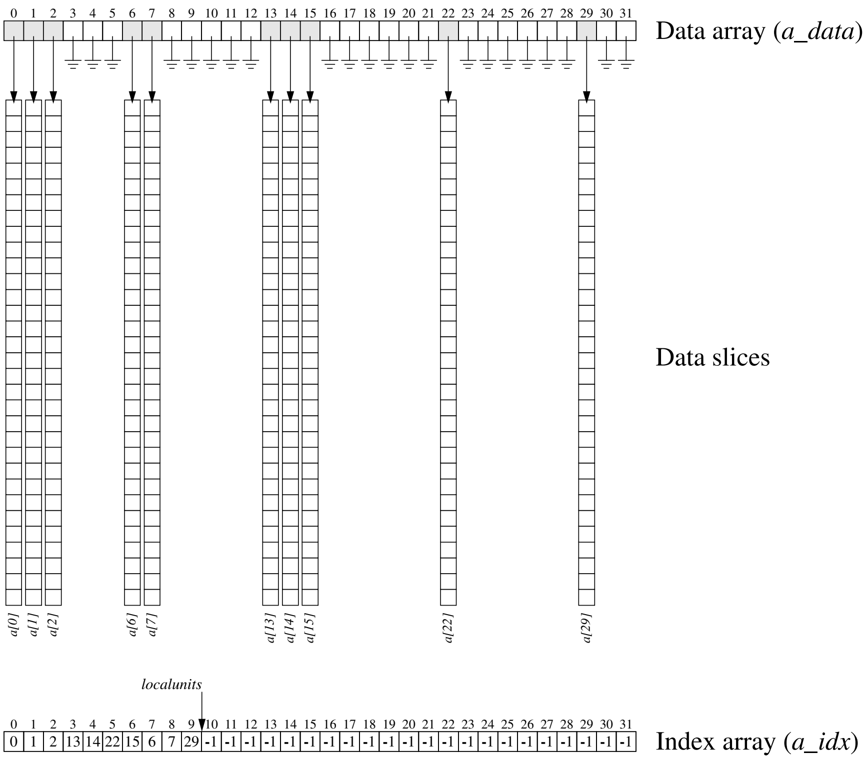
Figure 6.4: Basic (scattered) data structure for storing distributed data. Local data slices are stored in the corresponding locations of the data array. Other locations contain null pointers.

structures are modified by operations which transfer work between processors, i.e. sending and receiving work units. The application code may reference particular slices of the distributed data in the sequential portion of the code (e.g., Figure 6.3a) or may access many or all slices in the distributed loop (e.g., Figure 6.3b). The code in Figure 6.3 is used in comparing the efficiency of different data representations; when the sequential code is parallelized, a is distributed by rows.

In the basic scattered implementation (Figure 6.4) described earlier, sending work and stepping through all local data slices is inefficient because each processor may perform many unnecessary tests to identify local slices, especially when there is a large number of processors and the non-null elements of the data array are very sparse. These tests can be eliminated by augmenting the scattered implementation with an *index array* which stores only the set of local iterations (Figure 6.5a). Looping through the local indices can then be done by looping through the index array, using the values as indices into the data array (Figure 6.5e). Also, sending a data slice no longer requires a search. The index array is managed as a stack, with the stack pointer being the number of data slices stored on the processor; when work is moved (Figures 6.5b and 6.5c), indices for data slices are added to or deleted from the end of the index array. A disadvantage of the implementation using the index array is that an extra level of indirection is required to access the local data (Figure 6.5d).

The extra indirection can be avoided by using a *packed implementation* for the data: the local slices of the distributed data are packed into the beginning of the data array (Figure 6.6a). In this representation, when looping through the local indices, the slices of the data array can be referenced directly (Figure 6.6e). However, to access a particular slice of the data, it is necessary to search through the index array to determine whether the data is local and where the data is stored in the data array (Figure 6.6d). In the matrix multiplication example, this is not necessary since the index value is not important to the calculation. Other applications, such as the LU decomposition example, do reference particular data slices, and each processor must search to determine whether referenced slices are local. If memory is not a limitation, adding an additional *reverse index array* eliminates the need for searching (Figure 6.7).

In both scattered and packed representations, it is necessary to allocate enough memory to store the pointer and index arrays and the local data slices. When work is moved to a processor, the processor allocates more memory for the associated data slices, and when work is moved from a processor, the memory for the associated data slices is freed. This memory allocation and deallocation is an expensive part of work movement. The packed representations have an advantage in this regard because they can implicitly manage a free list for data slices so that system calls requesting more memory occur less frequently. When work is moved from a processor, the memory associated with the data slices is freed just by decrementing the count of slices on the processor and modifying the index and reverse index arrays; the data array still points to the memory. When more work is moved to the processor, the slice count is incremented. If the values of the data array for the new slices are non-null, then they point to memory that can be reused; otherwise, a request is made to the system for more memory. When memory is needed from the system, it can be allocated in



a) Data structures

```
idx = a_idx(--localunits);
send(idx);
send(a_data(idx), len);
free_mem(a_data(idx));
a_data(idx) = NULL;
```

b) Send work

```
receive(&idx);
a_data(idx) = alloc_mem(len);
receive(a_data(k), len);
a_idx(localunits++) = idx;
```

c) Receive work

```
if (a_data(i) != NULL) {
      a_data(i)(...) = ...
}
```

d) Access particular slice

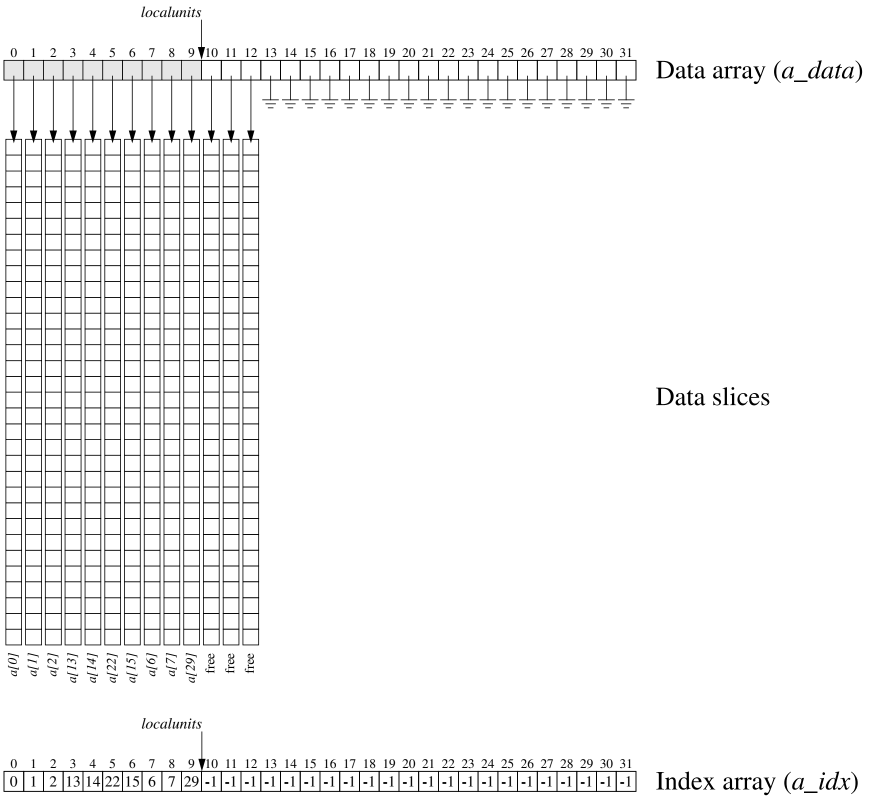
```
for (i = 0; i < localunits; i++) {
      a_data(a_idx(i))(...) = ...
}
```

e) Access each slice

Figure 6.5: Scattered data structure with index array. Local data slices are stored in the corresponding locations of the data array. Other locations contain null pointers.

large chunks, with excess put on the free list so that future system calls can be avoided. In the scattered representation with an index array, freed data can also be left in place, but the memory can only be reused if the same work unit is moved back to the processor.

For static load balancing, any initial distribution can be specified in the index array. At run time, the majority of the iterations remain in the original distribution because the index array is only changed at one



a) Data structures

```
localunits--;
send(a_idx(localunits));
send(a_data(localunits), len);
```

b) Send work

```
receive(&a_idx(localunits));
if (a_data(localunits) == NULL) {
      a_data(localunits) = alloc_mem(len);
}
receive(a_data(localunits), len);
localunits++;
```

c) Receive work

```
for (i = 0; i < localunits; i++) {
      if (a_idx(i) == k)
            a_data(i)(...) = ...
            break;
      }
}
```

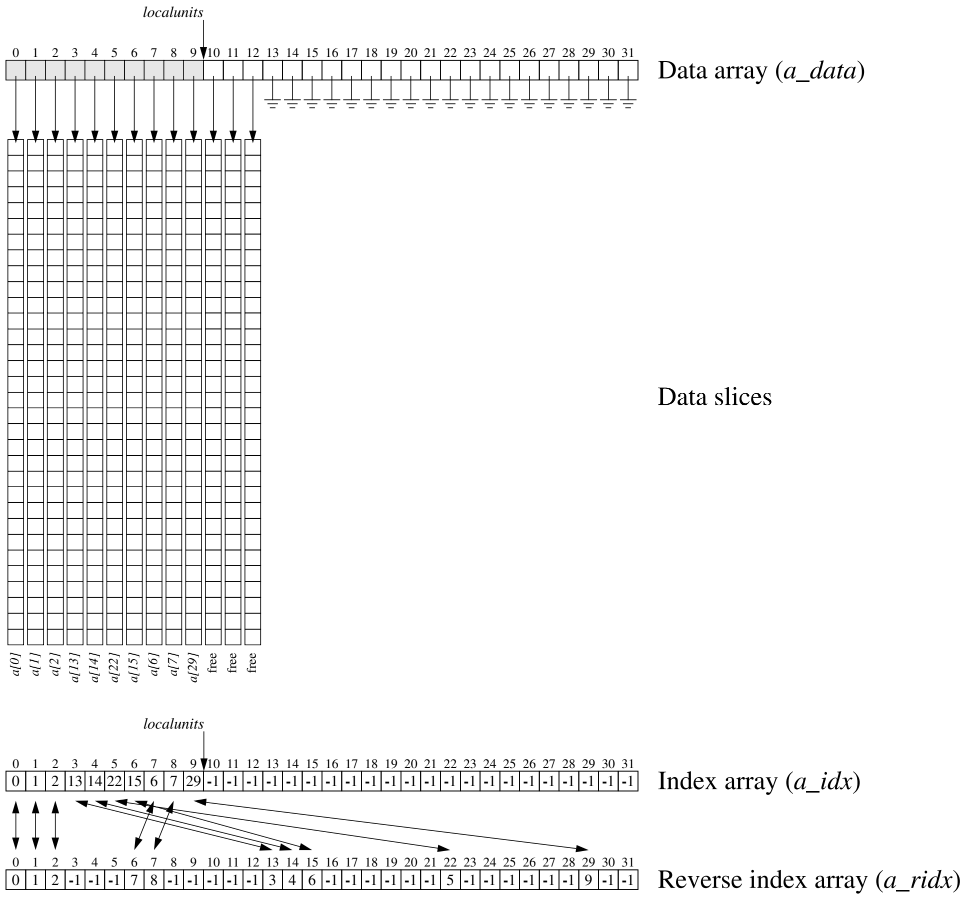
d) Access particular slice

```
for (i = 0; i < localunits; i++) {
      a_data(i)(...) = ...
}
```

e) Access each slice

Figure 6.6: Packed data structure. Local data slices are packed into the beginning of the data array. Other locations contain null pointers. Index array lists local slices.

end. For the LU decomposition example, the triangular iteration space for executions of the distributed loop is statically balanced (approximately) using a cyclic distribution so the initial distribution is cyclic; otherwise, data would constantly be shifted toward the leftmost processor.



a) Data structures

```
localunits--;
a_ridx(a_idx(localunits)) = -1;
send(a_idx(localunits));
send(a_data(localunits), len);
```

b) Send work

```
receive(&a_idx(localunits));
if (a_data(localunits) == NULL) {
      a_data(localunits) = alloc_mem(len);
}
receive(a_data(localunits), len);
a_ridx(a_idx(localunits)) = localunits;
localunits++;
```

c) Receive work

```
if (a_ridx(k) >= 0) {
      a_data(a_ridx(k))(...) = ...
}
```

d) Access particular slice

```
for (i = 0; i < localunits; i++) {
      a_data(i)(...) = ...
}
```

e) Access each slice

Figure 6.7: Packed data structure with reverse index array. Local data slices are packed into the beginning of the data array. Other locations contain null pointers. Index array lists local slices. Reverse index array makes determining whether a slice is local easier.

The data structures described so far all assume that work may be shifted between any processors.

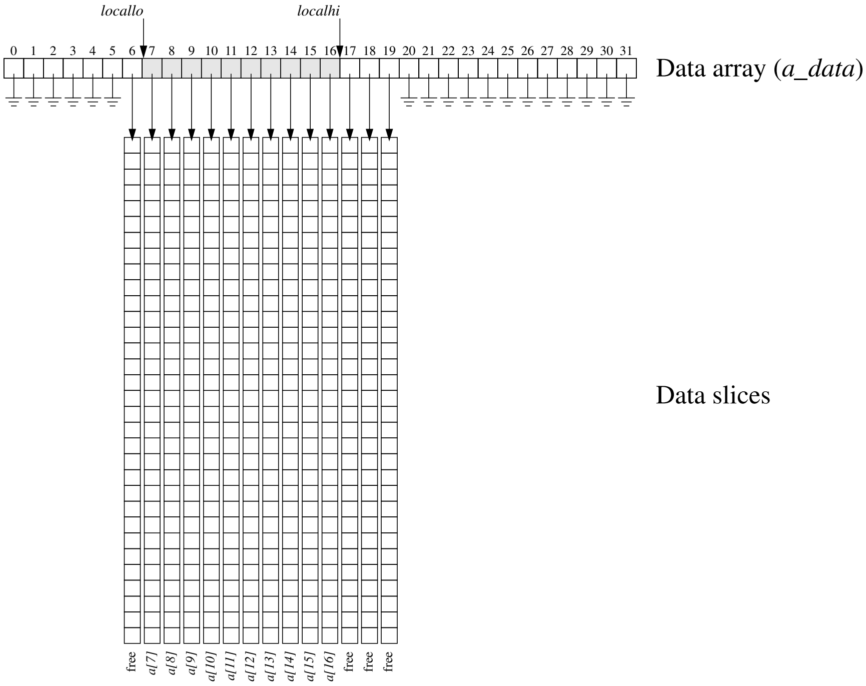
However, for applications with DOACROSS loops, where work movement is restricted (Section 5.5.2), adjacency of iterations is maintained using an irregular block distribution. Because all local slices with a block distribution are contiguous, index arrays are not needed to aid in identifying local data. Local slices of the distributed array are put in their actual locations in the data array, and the range of local slices is specified by the indices of the lowest and highest (+1) local slices (Figure 6.8a). These indices become the loop bounds when looping through all local slices of a distributed loop (Figure 6.8e). Testing whether a particular data slice is local (Figure 6.8d) just requires testing whether the slice number (i) is in the local range ($locallo \leq i < localhi$). The data array is managed as a bidirectional stack, with *locallo* and *localhi* as the stack pointers. As with the packed implementations for unrestricted work movement, the stack pointers also manage free lists at no extra cost.

6.2.3 Selecting the data structure

Table 6.2 summarizes the properties of the different data structures with respect to work movement and data access costs. (The highest cost in sending and receiving work—actually transferring the data—is not included in our assessment of costs because it is required for any implementation.) When generating code, the compiler must select the data structure most appropriate for the given application. The restricted data structure (Figure 6.8) is the most efficient for all the types of accesses. However, restricted work movement is not used for all applications because it requires more work movement involving intermediate processors and thus has a higher total cost. Thus, the restricted data structure is only used with applications with DOACROSS loops that require restricted work movement. For applications without restricted work movement, packed data structures should be used because of their memory management benefits. The reverse index array should only be included for applications that access particular slices of the distributed data because the extra array takes up space, and managing the extra array increases the processing required to send and receive data slices.

6.3 Dealing with varying loop bounds

In some applications, the number of loop iterations in the distributed loop changes each time it is executed. For example, in LU decomposition, the number of iterations decreases by one with each execution. When



a) Data structures

```
localhi--;
send(a_data(localhi), len);
```

b) Send work (to right)

```
if (a_data(localhi) == NULL) {
      a_data(localhi) = alloc_mem(len);
}
receive(a_data(localhi), len);
localhi++;
```

c) Receive work (from right)

```
if (locallo <= k < localhi) {
      a_data(k)(...) = ...
}
```

d) Access particular slice

```
for (i = locallo; i < localhi; i++) {
      a_data(i)(...) = ...
}
```

e) Access each slice

Figure 6.8: Data structure for applications with restricted work movement.

moving work to balance load in applications with varying loop bounds, it is only beneficial to move the iterations and data that will be executed in future executions of the distributed loop; and when computing new distributions, the load balancer should only consider the number of iterations in future executions of the loop, not the total number of distributed data slices. At run time, we must distinguish data slices that will be used in future executions of the distributed loop from those for which all work has been completed. This is done by labeling slices with future work as *active* and those without future work as *inactive*. Only active data slices are moved between slaves to move work. With decreasing loop bounds, data slices are initially active

Data Structure	send work	receive work	access each	access particular
Basic (Scattered)	expensive (*free_mem*)	expensive (*alloc_mem*)	expensive (search)	cheap
Scattered with index array	expensive (*free_mem*)	expensive (*alloc_mem*)	cheaper (indirection)	cheap
Packed	cheap	usually cheap	cheap	expensive (search)
Packed with reverse index array	cheap	usually cheap	cheap	cheaper (indirection)
Restricted	cheap	usually cheap	cheap	cheap

Table 6.2: Summary of data access costs for different data structures.

and are *deactivated* with each execution of the distributed loop; with increasing loop bounds, data slices are initially inactive and are *activated* with each execution of the loop. For applications with unrestricted work movement, active slices are stored in the data structures as described earlier (i.e., *a_data*, *a_idx*, *a_ridx*, *localunits*), and inactive slices are stored in separate, similar structures (*a_inact*, *a_idxinact*, *a_ridxinact*, *inactunits*). The compiler must generate slave code to move slices between the active and inactive data structures as the bounds of the distributed loop vary (Figure 6.9b); it must also generate master code that keeps the load balancer's notion of the total number of work units up-to-date (Figure 6.9c). Placement of this code can be done with the aid of directives from the programmer (Figure 6.9a) or can be determined by the compiler from analysis of loop bounds; in either case, the code to activate or inactivate data slices should be inserted in the same location in the generated code.

For simplicity, the load balancer does not keep track of the specific work units allocated to each slave. It only knows how many work units are allocated to each slave. When a work unit is activated or deactivated, the code inserted into the master informs the load balancer that the total number of work units has increased or decreased, but the load balancer does not know which slave the change occurred on. The status information sent by a slaves at each load balancing interaction includes the number of active work units currently allocated to the slave, but with pipelined or asynchronous load balancing, this information is delayed. The load balancer can only approximate the current distribution at a given time. Each time work is activated or deactivated, the load balancer modifies its view of the current distribution assuming that changes are distributed equally. Using its incomplete information, the load balancer provides the slaves with inexact instructions specifying the target number of active data slices each slave should have, from which the slaves

```
DO i = 0, n-1
      DO j = 0, i        /* distributed  */
            a(i)(j) = ...
      END DO
      DEACTIVATE(a, i)
END DO
```

a) Compiler directive showing when to deactivate a data slice

```
aidx = a_ridx(i)              /* current index            */
lidx = localunits - 1;        /* last active index        */
iidx = inactunits;            /* last inactive index + 1  */
/* move the slice to the inactive array.                  */
a_inact(iidx) = a_data(aidx);
a_idxinact(iidx) = i;
a_ridxinact(i) = iidx;
inactunits++;          /* increase inactive count.        */
/* repack the active array.                               */
a_data(aidx) = a_data(lidx);
a_idx(aidx) = a_idx(lidx);
a_ridx(a_idx(aidx)) = aidx;
a_data(lidx) = NULL;
a_idx(lidx) = -1;
a_ridx(i) = -1;
localunits--;          /* decrease active count.          */
```

b) Deactivation code on slave

```
/* decrease the number of active units.                   */
activeunits -= 1;
```

c) Deactivation code on master

Figure 6.9: Code for deactivating data slices when distributed loop bound decreases assuming packed data structures with reverse index arrays.

can determine amounts of work to move.

6.4 Work movement routines

The load balancer makes work movement decisions based on very abstract information: numbers of work units and rates of computation. It just generates instructions to move the abstract work units from one slave to another. However, from the point of view of the slaves, work units are iterations of a distributed loop, and the slaves must determine which iterations and which portions of the distributed data must be moved between the processors. The iterations moved are determined by the distributed data structures used and, for applications with restricted work movement, the direction of work movement; the data moved is determined by relations established at compile time between the distributed loop iterations and the distributed data slices. Once data associated with the work is identified, the senders of work must pack the data into messages, update the local data structures, and send the packaged data to the receiver specified by the load balancing instruction. The receiver must receive and unpack the data and store the data in its local data structures.

6.4.1 Identifying data to be moved

Distributed data slices are linked to distributed loop iterations by the owner computes rule. At compile time, possibly aided by hints from the programmer, a relation is established between the loop indices for the distributed loop and the array indices for the distributed data accessed by the loop. Given an assignment of iterations to processors, the relation defines the processors that own and store the master copy of the data slices. The data slices modified by an iteration are owned and stored by the same processor as the iteration; copies of data slices read, but not written to, by an iteration may also be stored on the same processor as the iteration, but they may be owned by other processors. Existing parallelizing compilers (e.g., AL [67] and Fortran D [24, 36]) already establish this type of ownership relation when they assign iterations and data to processors for static distributions.

When work is moved between two processors, ownership of loop iterations and data slices is shifted from the source processor to the destination processor. The data moved is identified using the relation established between the iterations and the data. In some cases, neighboring data slices may also be sent along with the slices changing ownership because the iterations related to the slices changing ownership also reference the neighboring slices; the neighboring slices are still owned and stored by the sending processor, but they are replicated to eliminate the need for communication when executing the moved iterations. The AL compiler [67] identifies distributed data slices referenced (i.e., read) in the same loop iteration with the aid of directives provided by the programmer: XREL (cross relation) declarations link slices from different distributed arrays, and WREL (window relation) declarations link slices from the same distributed array. The cross relations identify data which should be kept together; the window relations identify data slices for which replication might be beneficial. However, replicating slices is only beneficial if the slices are in the appropriate state for use by the moved iterations. In the SOR example, each iteration of the distributed loop depends on the result of the previous iteration and on the old value of the next iteration. When work is moved to a processor from the processor on its left, the new values of the data to the left of the moved work are not yet valid; however, when work is moved from the right, the slice to the right of the moved work contains the old values needed for the computation. Therefore, for SOR, neighboring data slices should only be copied when work is shifted to the left. Identifying the state of the data slices may be difficult for a compiler so, in some cases, data may be copied unnecessarily.

6.4.2 Moving distributed data between processors.

The compiler must generate application-specific routines for sending and receiving distributed data structures and for updating the local data structures on the sender and receiver. The compiler knows the layout of the data in memory and can generate code to gather the necessary data when sending work and to scatter the data across the data structures when receiving work. Work movement messages must contain the following information:

- Number of data slices sent.
- Indices of data slices sent.
- Data slices owned by the moved work.

Since data slices from multiple distributed arrays may be owned by the moved loop iterations, the different types of slices must also be indicated in the messages; this can be done implicitly by the ordering of the data in the messages. Neighboring data slices also may be included in the messages. Because the number of neighboring slices sent with work is known at compile time, receivers need no additional information to distinguish neighboring slices from those that are actually owned by the transferred work units.

If load balancing is asynchronous, there may be differences in the amount of progress on different slaves, so data transferred to move work may be in a different state from the data on the receiving slave. Thus, work movement messages must also contain state information for the data slices indicating how far the computation on the slices has progressed. This state can be specified by the values of the loop indices of all loops surrounding the distributed loop. With synchronous or pipelined load balancing, the additional state information is usually unnecessary because the load balancing keeps the slaves synchronized so that moved slices are in a state known to the receiver, except when the load balancing hooks have been inserted between iterations of the distributed loop.

Each work movement instruction specifies a sender, a receiver, and the amount of work to be moved. The sender and receiver each receive a copy of the instruction. Only active work units should be moved. For the unrestricted work movement case, the last data slices listed in the active index array on the sender are moved, and the number of active slices on the sender is decreased by the number of slices moved. On the receiver, the indices of the received data slices are added to the end of the active index array, the data is copied into the local data structures, and the number of active slices is increased by the number of slices moved.

With restricted work movement, the work movement instruction specifies work movement either to the left or to the right. Depending on the direction of work movement, data slices are moved to or from either the right side or left side of the active range of slices, and the corresponding loop bound is modified.

6.5 Work update routines

In some cases, when load balancing occurs, slaves have made different amounts of progress on their local computation. For applications with DOACROSS loops parallelized by pipelining, such as SOR, slaves early in the pipeline are ahead of slaves later in the pipeline. To maintain the pipelined execution of the loops, work movement is restricted so that work only moves between logically adjacent slaves. Synchronous or pipelined load balancing is used for pipelined applications because of the frequent synchronizations inherent in the applications. This constrains the progress on the slaves so that logically adjacent slaves differ by one execution of the distributed loop (Figure 6.10a). When work is moved between processors in these cases, the moved data slices are not consistent with data already resident on the destination, so the moved work and data can not be inserted immediately into the computation loop and data structures with the resident work and data. Work moved to a processor from its left is one pipeline phase ahead of its local work, and work moved from its right is one phase behind the local work (Figure 6.10b). The received data must be handled separately until it is made consistent with the rest of the data. Work shifted from the right is immediately updated upon receipt using a copy of the main loop nest with bounds adjusted to work only on the received work (Figure 6.10c). Work shifted from the left is set aside until the local work catches up (Figure 6.10d.

Since for DOACROSS loops, work movement is restricted to be between adjacent slaves, only these two cases need to be handled.

For DOALL loops, synchronous and pipelined load balancing keep the slaves synchronized so that all data is in a consistent state whenever work is moved; received work needs no special handling and can be merged into the local data structures immediately. However, with asynchronous load balancing, there can be great disparity between the progress on different slaves. In this case, work is moved away from processors whose computation rates have decreased relative to the rates of the other slaves so work transferred to a slave is always behind the work already local to the slave. Received work is updated so that it is consistent with local work using a copy of the main loop nest with adjusted bounds. With unrestricted work movement, a slave may receive work from several other slaves during a single load balancing phase. Fortunately, each work movement instruction can be handled independently as soon as it is received so there is no need to buffer work received from multiple slaves.

The compiler is responsible for generating code to set work aside or catch work up. The routine for catching work up has the same core computations as the routine for computing work. The loop body is copied, but loop bounds must be changed. Work can be set aside by inserting the data slices into the data structures, but not changing the loop bounds to work on the new data until the original data has reached the same state of progress.

6.6 Modifications to communication code

Adding dynamic load balancing to parallelized code has an impact on how data updates are exchanged at run time since the location of distributed data and work is no longer known at compile time. If statements outside of distributed loops reference distributed data, communication may be required to access the data. With a fixed data distribution, a compiler can generate code that can compute the location of any distributed data element using information local to each processor [60]. However, with a data distribution that changes at run time due to load balancing, processors cannot compute data locations using local information only; additional communication may be necessary. This section describes the handling of several cases where run-time information is necessary. The compiler must generate the described communication code.

If a replicated variable is to be copied into a location in a distributed data structure, each processor determines whether it owns the target location as described in Section 6.2, and the owner of the target

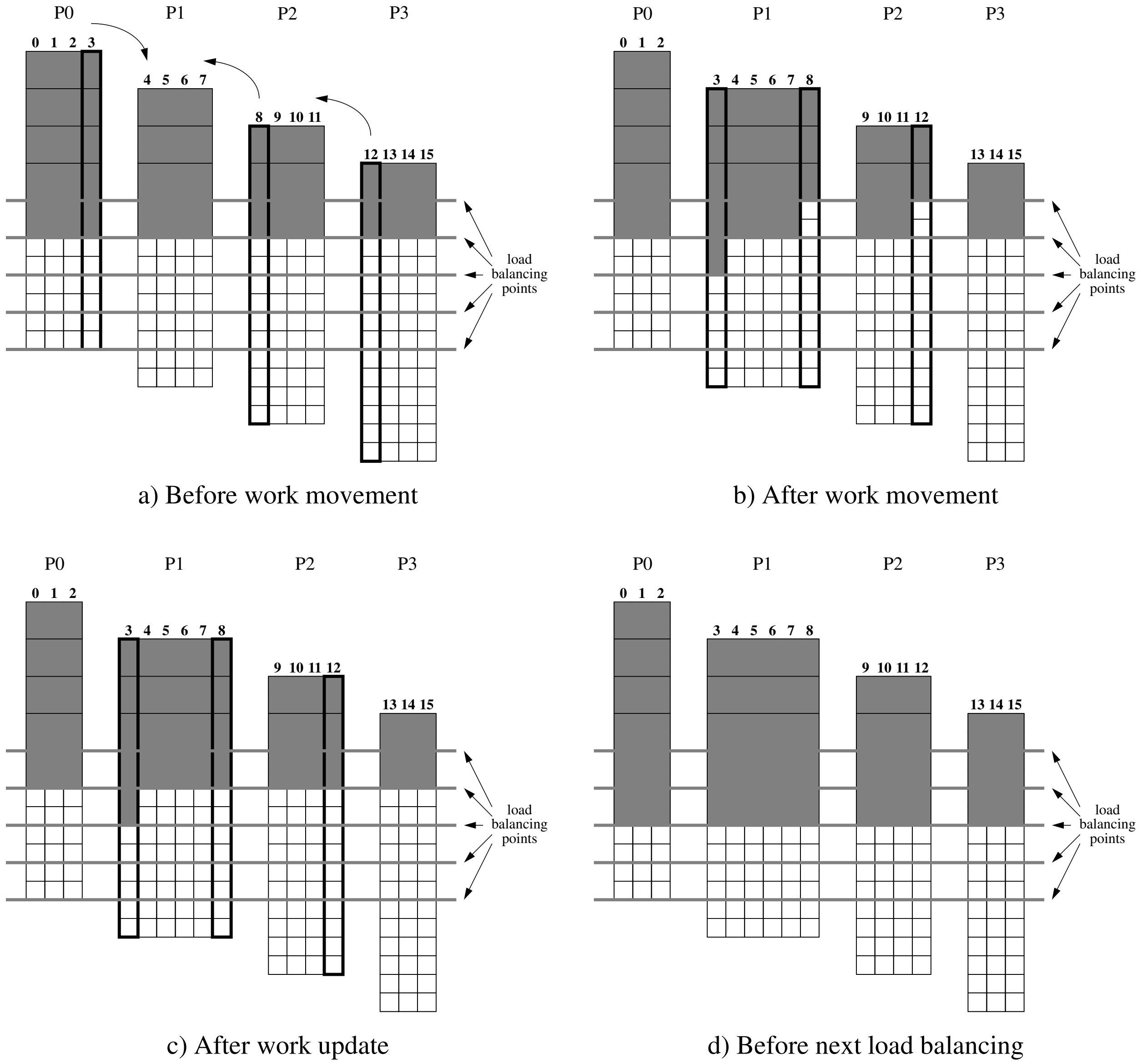


Figure 6.10: Steps in load balancing of the SOR example. Shading indicates progress on computation. (Communication and replicated data not shown.)

location does a local copy.

If an element of a distributed data structure is to be copied into a replicated variable, each processor determines whether it owns the element as described in Section 6.2, and the owner broadcasts the data to all other processors.

If an element of a distributed data structure is to be moved into another distributed data location, then the data must still be broadcast because the sender does not know which processor owns the target location.

The processors for which the data was not intended receive but discard the data. In a system with hardware support for multicast, all processors can receive the broadcast information in parallel so the size of the data to be transferred is usually not important. An alternative would be to have each slave determine whether it owns the target location and send its processor identification number (pid) to the sender; however, broadcast of the pid would be necessary because the processor owning the target location does not know which processor owns the data to be sent. In a system that does not support multicast, this alternative may perform better when the amount of data to be moved is large. Otherwise, the first approach is more efficient because it only requires a broadcast, while the second approach requires a broadcast and another send.

Because reduction operations are associative, the implementation of reduction operations is not affected by dynamic load balancing. Each processor's contribution to the total value is computed as usual, although each processor may operate on a different number of elements. The combination of the local portions is computed in the same way as usual (e.g., using a combining tree).

6.7 Summary

This chapter described changes that must be made to the code generation phase of a parallelizing compiler to support dynamic load balancing. The loop structure of the parallelized program does not have to be changed for the slave processes, but must be duplicated on the master. Loop bounds and distributed data structures must be changed to handle and facilitate work movement. We described several implementations for data structures for irregular distributions; the compiler must select the most efficient implementation based on the features of the given application. In addition, because of the dynamic, irregular data distributions resulting from load balancing, the communication code generated by the compiler must be modified to locate the distributed data slices involved in the communication. The compiler must also generate application-specific code to send and receive work and to deal with received work that is inconsistent with work already resident on the receiving slave. The compiler can make most decisions regarding code generation using simple rules based on features identified in the application code.

Chapter 7

Evaluation

To evaluate the mechanisms for supporting dynamic load balancing described in this thesis, we implemented a runtime system on the Nectar system [3], a set of workstations connected by a high-speed fiber optic network. We hand-parallelized the matrix multiplication (MM) and successive overrelaxation (SOR) examples described earlier (Section 1.2.5) and took measurements with varying parameters in several controlled environments.

This chapter begins with a description of our experimental setup on the Nectar system. Then we present measurements that show how dynamic load balancing affects performance in environments with different load characteristics. Performance is evaluated using the measures described in Section 1.5. The remainder of the chapter discusses models of the performance of applications with load balancing to give a frame of reference for the performance measurements.

7.1 Experimental setup

This section describes our experimental setup on the Nectar system and includes descriptions of the target system and its programming environment, the versions of the example applications used in the experiments, and the criteria used to compare the performance of the different versions of the applications.

7.1.1 Target environment

Our target platform is the Nectar system [3] developed at Carnegie Mellon University. The system consists of a fully-connected crossbar network connecting a set of workstations (Figure 7.1). Each workstation contains a memory mapped Communication Accelerator Board (CAB) which is connected to the crossbar by 100 megabit per second fiber optic links in each direction. For our measurements, the host processors were Sun 4/330 workstations running SunOS (Release 4.1.3), and no other users were active while the measurements were taken. Due to hardware limitations, the largest system used in our experiments consisted of a master and six slaves.

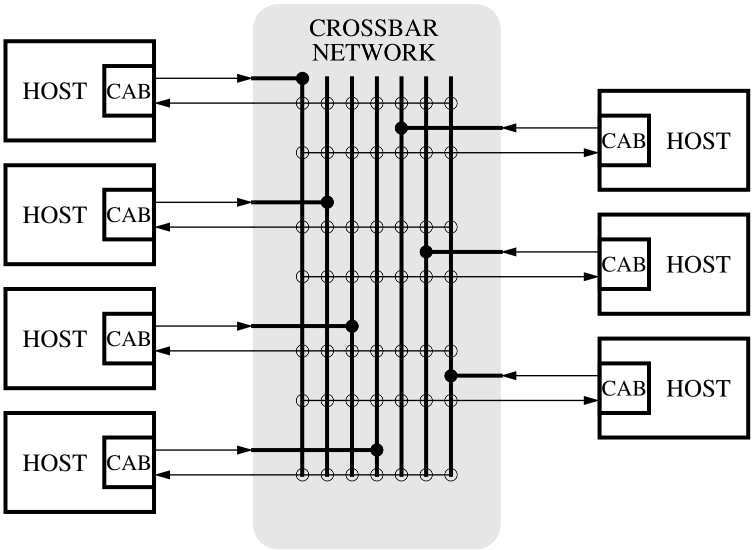


Figure 7.1: The Nectar system.

The programming interface provided for the Nectar system is called *Nectarine* [57]. The Nectarine library provides low-level routines for task management and for communication using several different protocols. Our implementation used Nectarine's *reliable message protocol* for communication. We spent a great deal of time optimizing the communication code in our load balancing runtime system and in the application-specific code. To reduce the number of times data is copied for communication, we selected Nectarine communication routines that allow data to be moved directly between user memory on the host workstation and system buffers on the CAB, eliminating the need to build the messages in user space. Also to reduce the need to copy data, messages to a processor from different sources were directed to different buffers so pieces of large messages would not be interleaved with other messages. In the cases where copying data was necessary, when possible, data was copied in blocks to minimize loop overheads (using the Unix *bcopy* routine [12]).

Measurements were taken in several controlled environments. Aside from processes created by the application being measured, no other processes were active on the processors. Artificial competing loads were generated by the application so that the computation resources used by the competing loads could be easily measured. If a competing load is to be generated on a processor, the application process on the processor forks off a new process that executes routines with desired load characteristics. The load generation routines used for the experiments generate either a constant computation intensive load or a discrete oscillating load of specified frequency. When the computation portion of the application program is completed, the application kills the generated processes and measures their CPU usage using the *getrusage* function [12] provided with Unix.

7.1.2 Application versions

The matrix multiplication (MM) and successive overrelaxation (SOR) examples used in our measurements are described in Section 1.2.5. For MM, the input matrices contain single-precision floating point values generated using simple functions. For SOR, elements of the input matrix are randomly generated, non-zero, single-precision floating point values. (The same seed is always used so that runs are reproducible.) For SOR, *zeta* is set to zero so that convergence conditions are never met and the WHILE loop always terminates after *maxiter* iterations (see Figure 1.8). Table 7.1 lists problem sizes used in our experiments (in this and previous chapters) and presents measurements of their sequential execution times on a single Sun 4/330 workstation running SunOS. Each sequential time presented is the minimum time measured over at least six runs. The problem sizes selected all fit into the real memory of the workstations. (Each workstation has at least 24 megabytes of memory.)

Each application has a sequential version, a parallel version with fixed, equal distribution of work, and a parallel version with dynamic load balancing. For the parallel versions, with or without load balancing, there is one master process—responsible for initialization, load balancing, and cleanup—and several slave processes which do the useful computation. Each of these processes runs on a separate processor. The parallel versions without load balancing use the same code as the parallel versions with load balancing, but the code is compiled with as much of the load balancing-related code disabled as possible. In all parallel versions,

Application	Problem size	Iterations	Time (seconds)
MM	100×100	1	1.79
MM	250×250	1	31.27
MM	500×500	1	252.71
MM	1000×1000	1	2068.84
SOR	500×500	10	20.77
SOR	1000×1000	10	84.19
SOR	200×2000	100	334.84
SOR	500×2000	40	334.40
SOR	1000×1000	40	335.07
SOR	2000×2000	10	353.24

Table 7.1: Elapsed time measurements for sequential versions of applications on Sun 4/330 workstation running SunOS.

a small amount of overhead may be attributed to code added for instrumentation. Our implementations of the applications and runtime system allow many application and load balancing parameters to be selected when invoking the program. These parameters are summarized in Table 7.2.

Parameter	Description
problemsize	Problem size for application
slaves	Number of slave processors
grainsize	Grain size in time quanta (when controllable)
type	Synchronous, pipelined, or asynchronous
depth	Pipeline depth for pipelined load balancing
threshold	Fractional improvement required for load balancing
filter	Filter used for rate measurements
quantumscale	Lower bound on load balancing period due to time quantum (in quanta)
overhead	Fraction of time allowed for interaction overhead; puts lower bound on load balancing period
movement	Restricted or unrestricted work movement
interrupts	Use interrupts instead of polling for communication
delay	Artificial delay added to communication

Table 7.2: Application and load balancing parameters selectable at startup time.

7.1.3 Performance with load balancing

We measured the execution times of the tuned MM and SOR applications in several environments for varying numbers of slave processors. These measurements were used to generate execution time (elapsed time), speedup, and efficiency graphs. The efficiency graphs include both the efficiency computed using Equation 1.4 and the lower bound on efficiency computed using Equation 1.2. Although the actual efficiency in using available resources may lie anywhere between the computed efficiency and the computed lower bound, the measured execution times and measurements of the time spent on the artificially generated competing processes indicate that the competing processes are not using more resources than expected and that the actual efficiency is close to that computed using Equation 1.4.

For the parallel versions of the programs, each data point presented is the average of at least 3 runs. For execution time and speedup graphs, vertical bars show the range of the raw measurements. In almost all cases, the variation in the measurements between different runs is small and significantly less than the difference between runs with and without dynamic load balancing. (Most of the vertical bars are so short that they do not extend outside the symbols for the average measured values.) There is a slight horizontal offset between the points for the sequential, parallel, and load balanced parallel versions so that the vertical bars can be distinguished. To make trends clearer, lines are drawn connecting the data points for the different numbers of processors, although, in reality, only integral numbers of processors are possible. Because of the slight horizontal offset between the different types of runs, the lines may appear to be shifted slightly upward or downward; closer examination of the data points is necessary in cases where the lines are close together.

The values of key load balancing parameters used for the measurements presented in this chapter are summarized in Table 7.3. (See Table 7.2 for a description of the parameters.) *movement* should be selected automatically by the compiler based on properties of the application. *grainsize* is selected automatically based on run-time measurements. The target load balancing period is selected at run time based on the values of *quantumscale* and *overhead* and on run-time measurements.

7.2 Load balancing overhead in a dedicated homogeneous environment

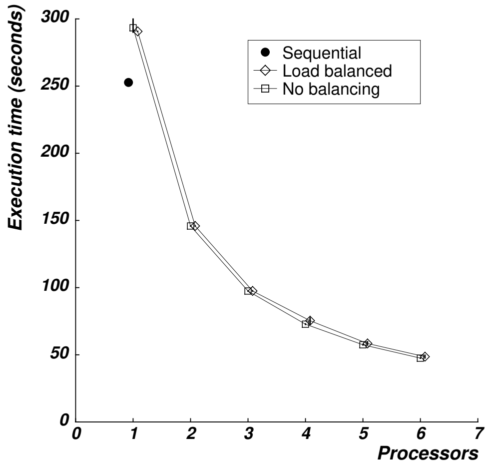
In an environment consisting of a homogeneous set of dedicated machines, load is balanced if work is distributed equally to the processors. Dynamic load balancing is not needed and can only add to the total execution time. Since we can not predict when there will be competing processes, we would like for the overhead added by dynamic load balancing in a dedicated homogeneous environment to be as small as possible. For applications with non-varying loop bounds, e.g., MM and SOR, the default distribution

Parameter	Value
grainsize	automatically selected at run time
type	pipelined
depth	1
threshold	10% improvement
filter	state machine described in Figure 5.1
quantumscale	10 quanta (1 second)
overhead	0.05
movement	unrestricted for MM, restricted for SOR

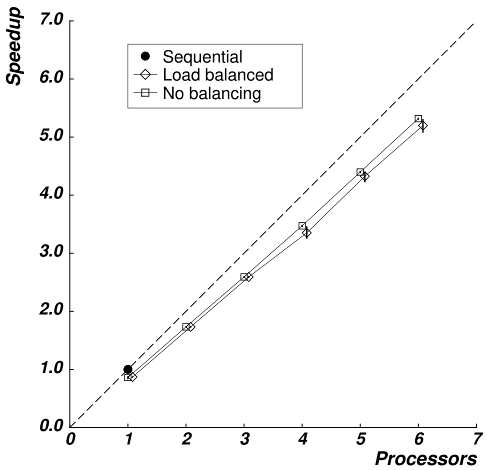
Table 7.3: Parameters used for load balanced versions of applications.

distributes work equally. Figures 7.2a, 7.3a, and 7.4a show the execution times for MM and SOR running in a dedicated homogeneous environment with and without dynamic load balancing. For both applications, dynamic load balancing adds little to the execution times of the applications. The speedup graphs (Figures 7.2b, 7.3b, and 7.4b) show that the speedup curves are close to the perfect linear speedup curve (the dashed line) for the number of processors used in our experiments, so our distributions of the problems get good parallelism. However, for SOR, which is parallelized by pipelining, the speedup curve drops off as the number of processors is increased because time spent filling the pipeline increases with the number of processors; communication costs for the application (independent of load balancing) also increase with the number of processors. MM requires no communication so its speedup remains linear. In the dedicated homogeneous case, because there are no competing loads, the efficiency graphs (Figures 7.2c, 7.3c and 7.4c) show the same information as the speedup graphs; the efficiency is just the speedup divided by the number of processors. Also, the efficiencies computed using Equations 1.4 and 1.2 are the same. All of the graphs in Figures 7.2, 7.3 and 7.4 indicate that the overhead added by dynamic load balancing in a dedicated homogeneous environment is small.

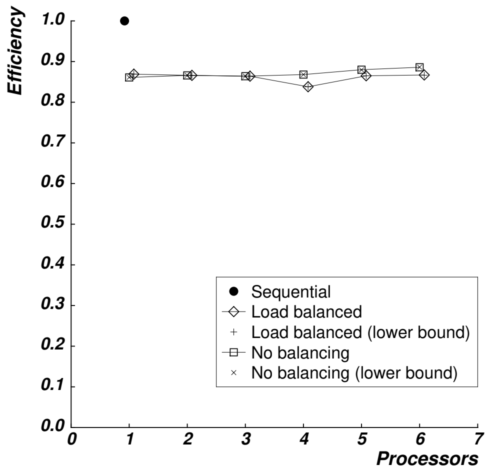
The highest efficiencies should be attained in dedicated homogeneous environments because the only load balancing overhead should be due to the interactions between the slaves and the load balancer, and most of this overhead is eliminated by pipelining the interactions. At best, we hope that the efficiencies measured in heterogeneous and dynamic environments will equal those measured in a dedicated homogeneous environment, and we therefore treat the dedicated homogeneous efficiencies as upper bounds for other environments. (It is possible to describe a case where efficiency in a dynamic environment could exceed that in a dedicated homogeneous case, e.g., if the competing loads used all of the pipeline fill and



a) Computation time



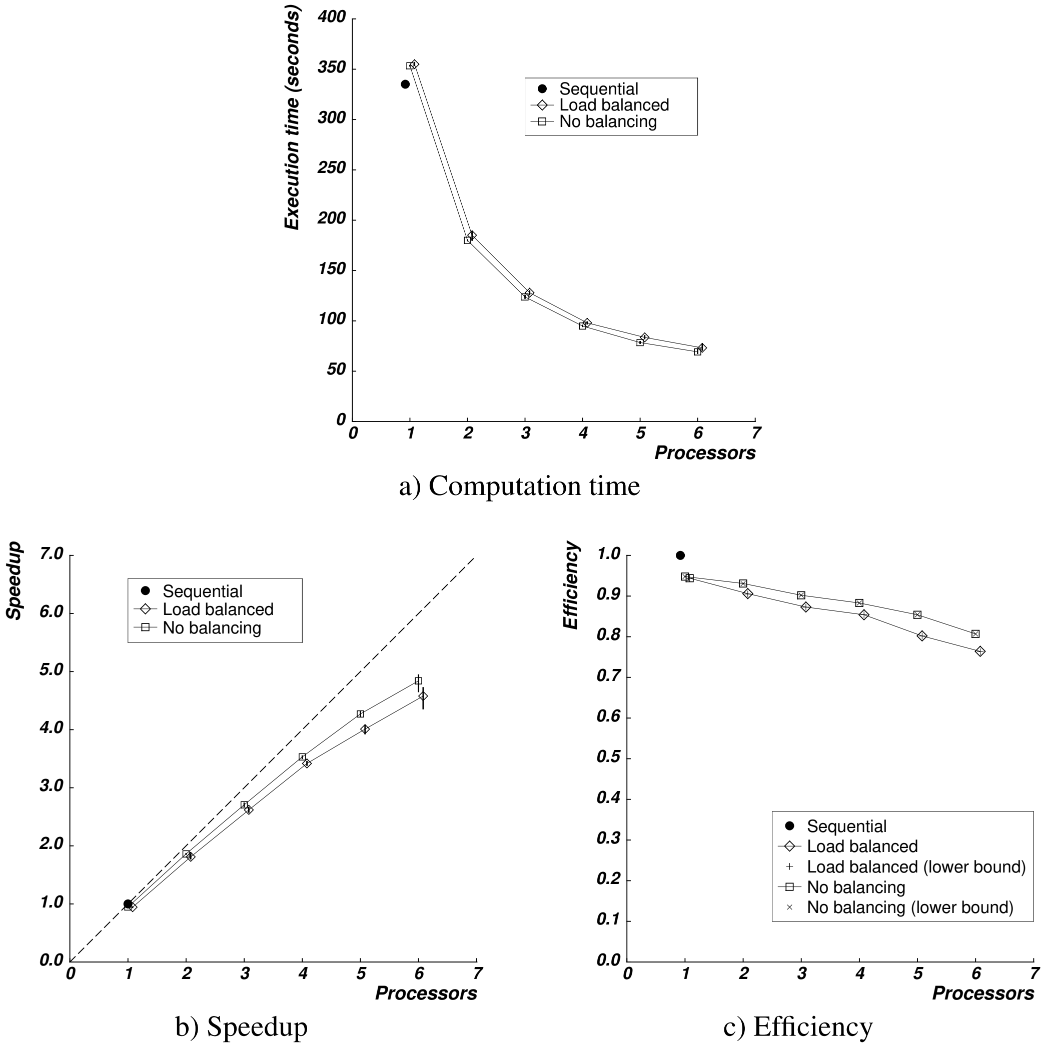
b) Speedup



c) Efficiency

Figure 7.2: 500×500 matrix multiplication running in dedicated homogeneous environment.

drain times productively for a pipelined application, but such a case is unlikely to occur with the simple loads used in our experiments.) For reference, we will show the dedicated homogeneous efficiencies (with smaller symbols and dashed lines) on the efficiency graphs for other environments.

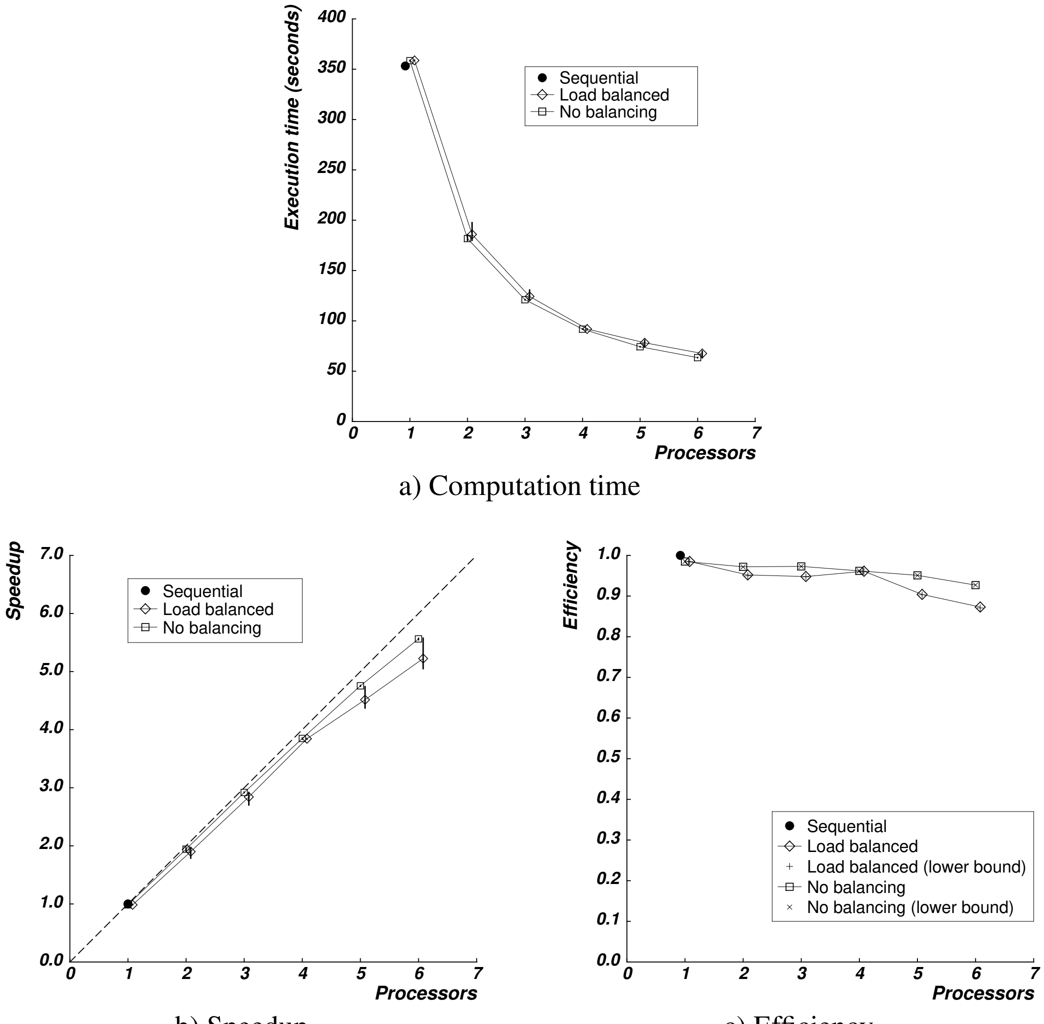


a) Computation time

b) Speedup

c) Efficiency

Figure 7.3: 1000 × 1000 successive overrelaxation (40 iterations) running in dedicated homogeneous environment.



a) Computation time

b) Speedup

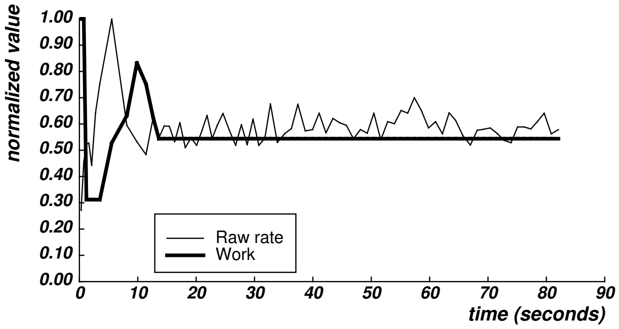
c) Efficiency

Figure 7.4: 2000 × 2000 successive overrelaxation (10 iterations) running in dedicated homogeneous environment.

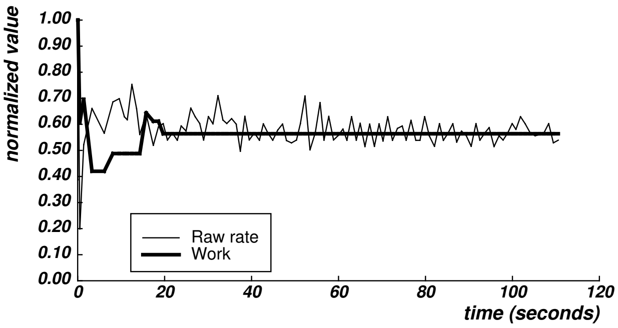
7.3 Load balancing with a constant competing load

When a constant competing load is added to one of the processors (processor 0), we expect the load balancer to redistribute work immediately and then for the distribution to remain stable. We create a constant load by running a computation-intensive loop as a competing process on one of the slaves. The load balanced application should find a static work distribution where that slave is allocated half as much work as the other slaves. Figure 7.5 shows how the work movement tracks the measured performance of the loaded processor, and confirms that such a distribution is reached; there is an initial period of instability while the competing application is starting up.[1] (In Figure 7.5, the raw measured rates are normalized against the maximum rate measured on the processor, and work allocated to the processor is normalized against the work that would be allocated if work was distributed equally to all processors.) Since work is only redistributed once, the efficiency of the load balanced version should be almost as high as in the dedicated homogeneous case. Figures 7.6b and 7.7b show this to be the case. However, for the 2000×2000 SOR example, in Figure 7.8b, the efficiency for the load balanced case is substantially lower than that in the dedicated environment because of the large amount of data (4 times as much as for the 1000×1000 SOR example, and 8 times as much as for the 500×500 example; see Table 7.5), especially for the smaller numbers of processors, that must be shifted to do the initial balancing. Also, because the size of the data for the problem is so large, the profitability determination phase is cancelling many more instructions than for the other examples, leaving the load unbalanced for a larger portion of the time.

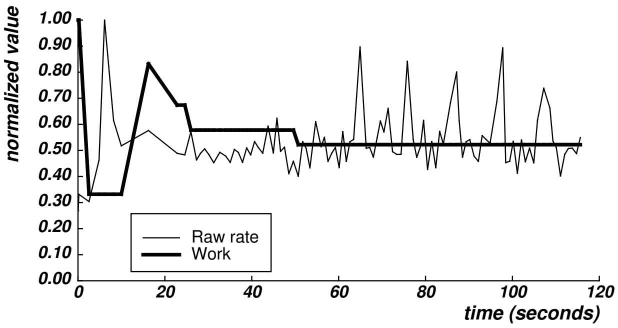
[1] It takes about 15 load balancing periods for the load to stabilize during the initial period of instability. However, for a transition in load after the initial period of instability, it typically takes about 5 load balancing periods for the load to reach a stable value.



a) 500 × 500 MM

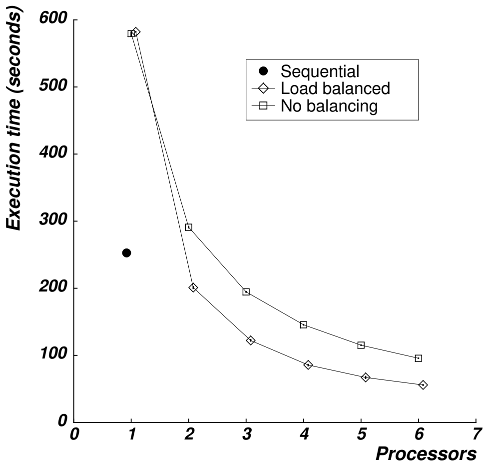


b) 1000 × 1000 SOR (40 iterations)

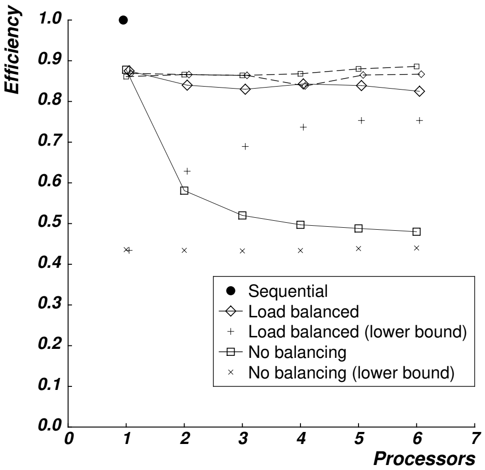


c) 2000 × 2000 SOR (10 iterations)

Figure 7.5: Measured performance and resulting work allocation on loaded slave on a 4 slave system with a constant computation-intensive load on one slave.

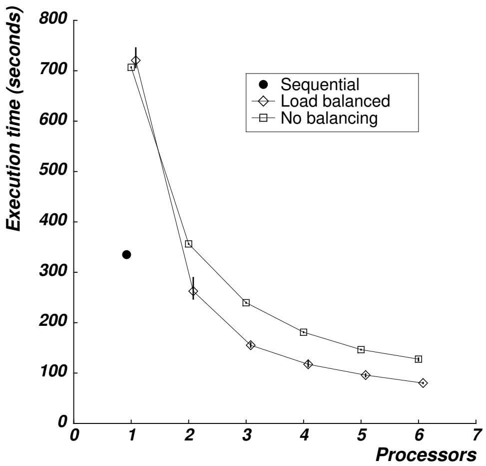


(a) Computation time

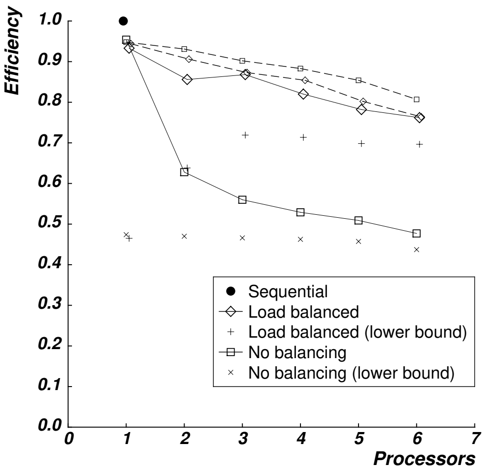


(b) Efficiency

Figure 7.6: 500 × 500 matrix multiplication running in homogeneous environment with constant load on first processor.

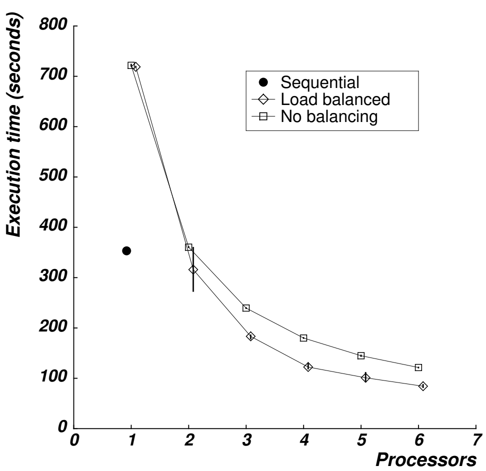


(a) Computation time

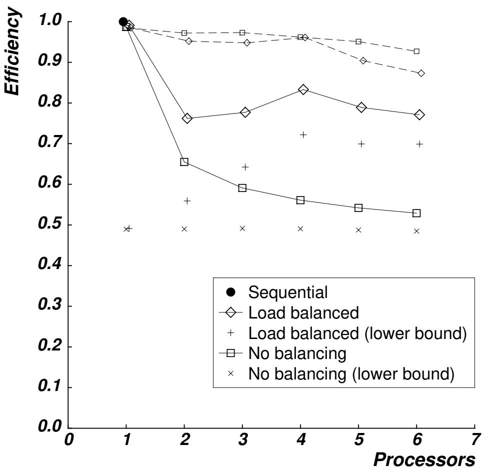


(b) Efficiency

Figure 7.7: 1000 × 1000 successive overrelaxation (40 iterations) running in homogeneous environment with constant load on first processor.



(a) Computation time



(b) Efficiency

Figure 7.8: 2000 × 2000 successive overrelaxation (10 iterations) running in homogeneous environment with constant load on first processor.

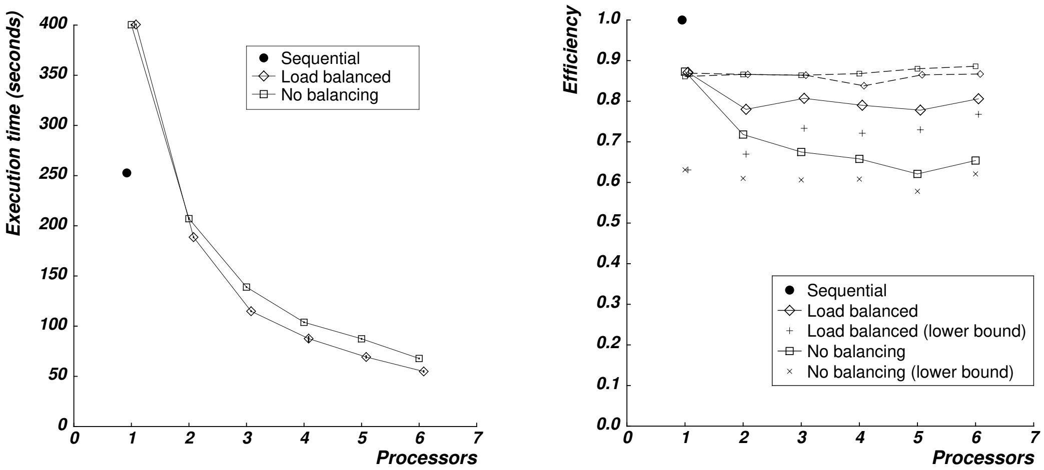
7.4 Load balancing in a dynamic system

To evaluate performance in a dynamic system, we added an oscillating load to one of the processors. With the oscillating load, the rate of computation on the processor alternates between the maximum rate attainable by the processor (i.e., the same rate attained in a dedicated system) and approximately half of the maximum attainable rate. Figures 7.9, 7.10, 7.11, and 7.12 show data for 4 oscillation frequencies ranging from once per minute to once every 2 seconds. Because of time constraints, data was not collected for the 2000×2000 SOR example for oscillation periods less than 20 seconds; it is presumed that performance for those periods would be as bad as or worse than the performance with the 20 second oscillation period.

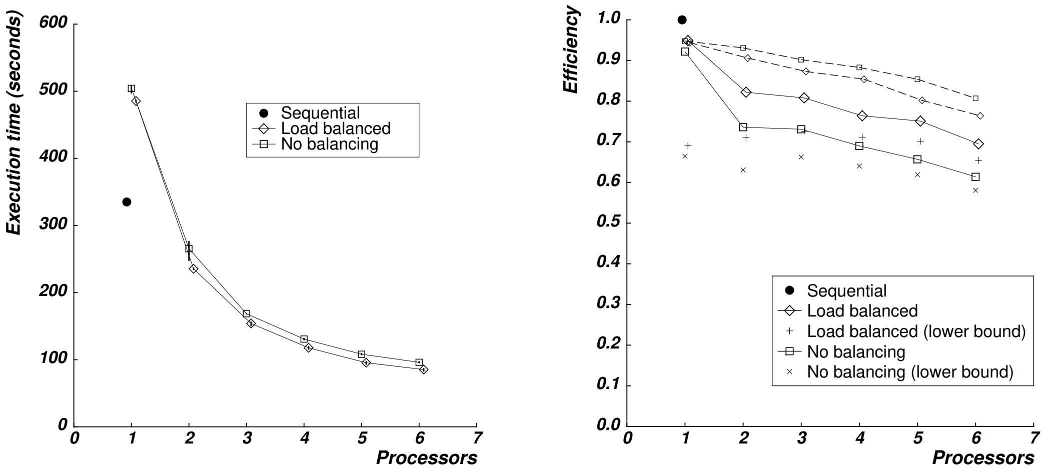
Without load balancing the expected efficiency (lower bound from Equation 1.2) is 0.75, independent of the number of processors, because all processors are limited by the performance of the loaded, slowest processor which oscillates between its maximum rate and half of its maximum rate. The measurements produce lower results, but much of the difference can be attributed to the same causes as the inefficiency in the dedicated environment. The measured efficiencies drop as the frequency of the oscillating load increases, possibly due to operating system costs for creating and killing processes not included in the *getrusage* measurements. The relatively high efficiencies without load balancing do not leave much margin for improvement with load balancing.

Figures 7.9a and 7.9b demonstrate that dynamic load balancing results in significant performance improvements when changes in load are infrequent and work movement costs are low. However, due to the work movement costs and the periods of imbalance, the performance is substantially lower than that in dedicated environments. As the oscillation frequency increases (Figures 7.10, 7.11, and 7.12), the efficiency with dynamic load balancing drops due to increased work movement costs (work is moved more times) and the increased significance of the lag between changes in rate and the redistribution of work. Figures 7.13a and 7.13b demonstrate that the period when load is balanced gets shorter as the oscillation frequency increases. Because of the large size of its distributed data, the 2000×2000 SOR example with dynamic load balancing performs poorly, even in an environment with loads that change very infrequently (Figure 7.9c); performance gets much worse when the oscillation frequency is increased (Figure 7.10c). On the other hand, because of its relatively small data size, the MM example is more immune to changes in the oscillation frequency. The effects of the work movement costs and the load imbalance will be modeled in the next section.

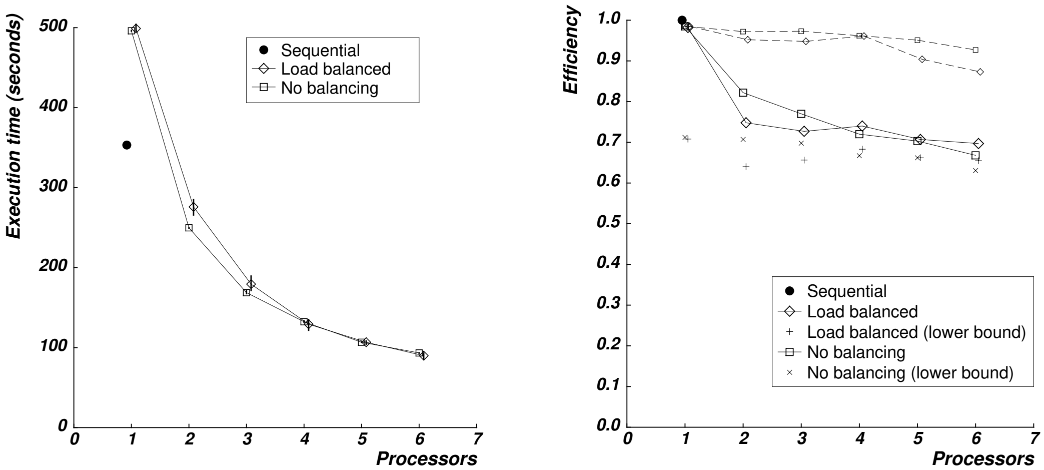
Dynamic load balancing is a difficult problem. The work movement costs and the response time of the load balancing system limit the system's ability to improve application performance, especially for applications with large data sizes. One way to improve performance is to reduce the amount of data that must be moved to shift work. To do this, the distributed data units must be decoupled from the problem size, e.g., by partitioning the data in more than one dimension. However, the benefits of partitioning the data in a different manner will be at least partially offset by the added complexity of managing the new data structures.



a) 500 × 500 MM

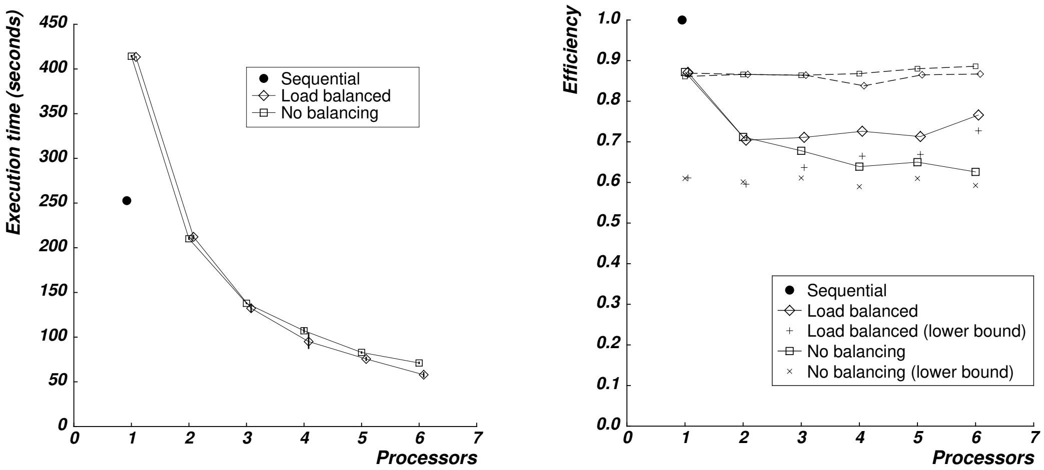


b) 1000 × 1000 SOR (40 iterations)

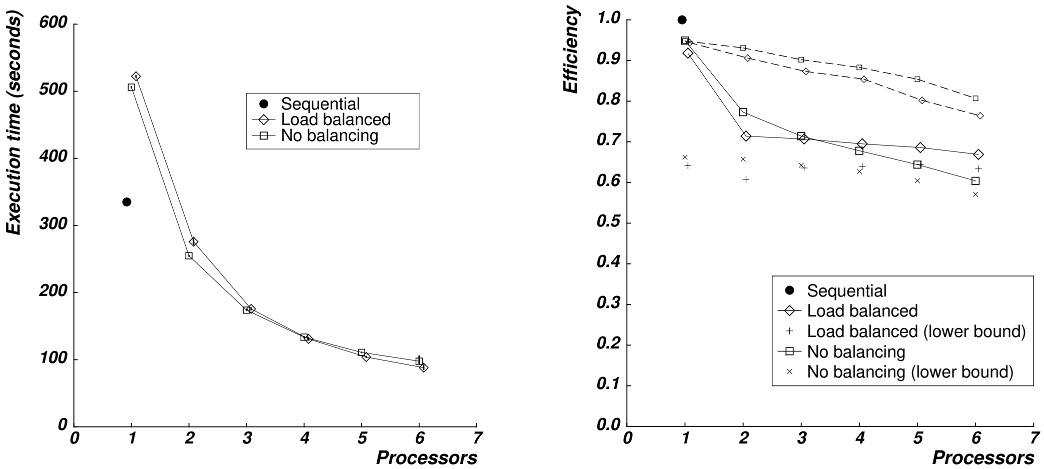


c) 2000 × 2000 SOR (10 iterations)

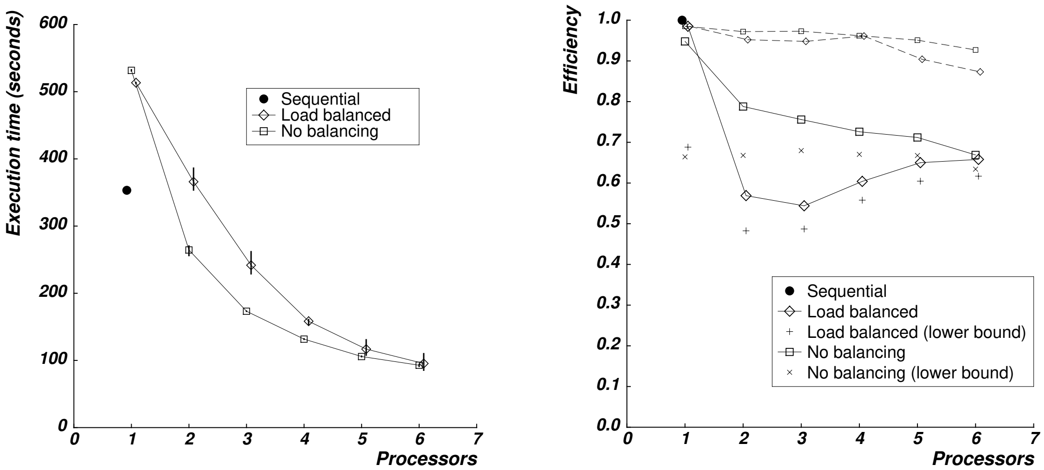
Figure 7.9: Execution time and efficiency in homogeneous environment with oscillating load (period = 60 sec, duration = 30 sec) on first processor.



a) 500 × 500 MM

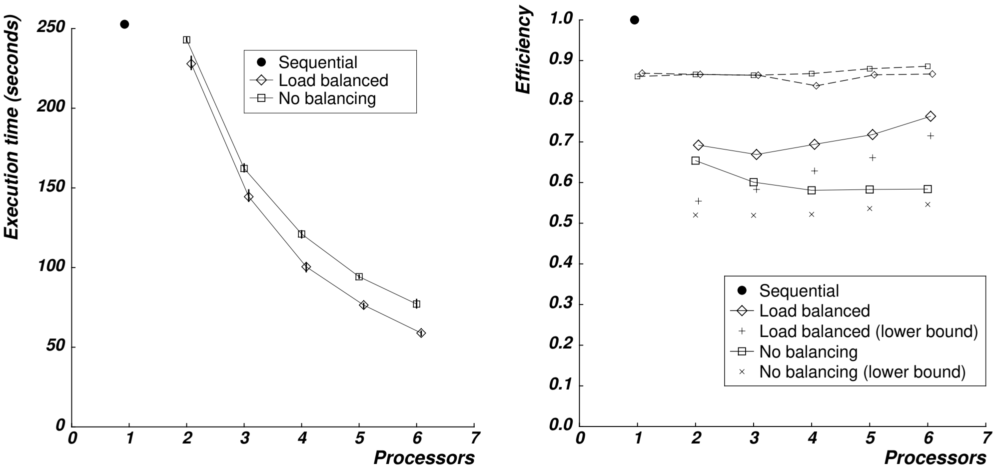


b) 1000 × 1000 SOR (40 iterations)

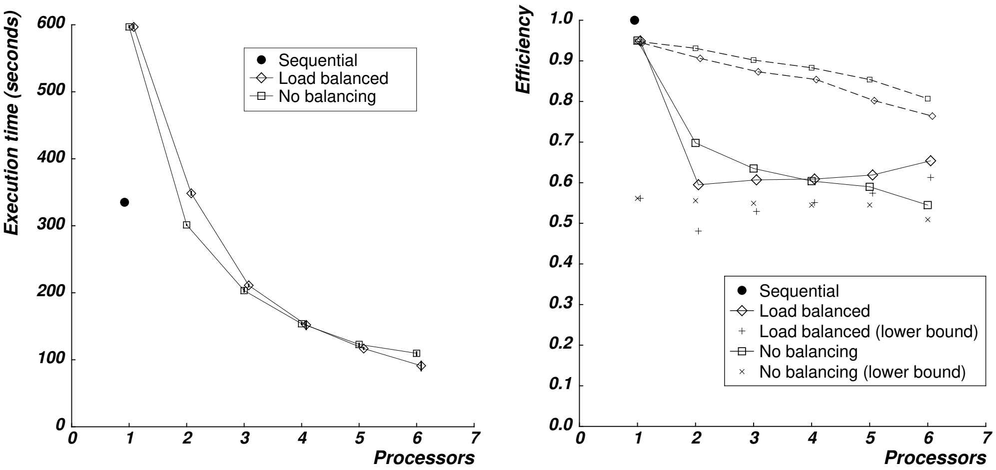


c) 2000 × 2000 SOR (10 iterations)

Figure 7.10: Execution time and efficiency in homogeneous environment with oscillating load (period = 20 sec, duration = 10 sec) on first processor.

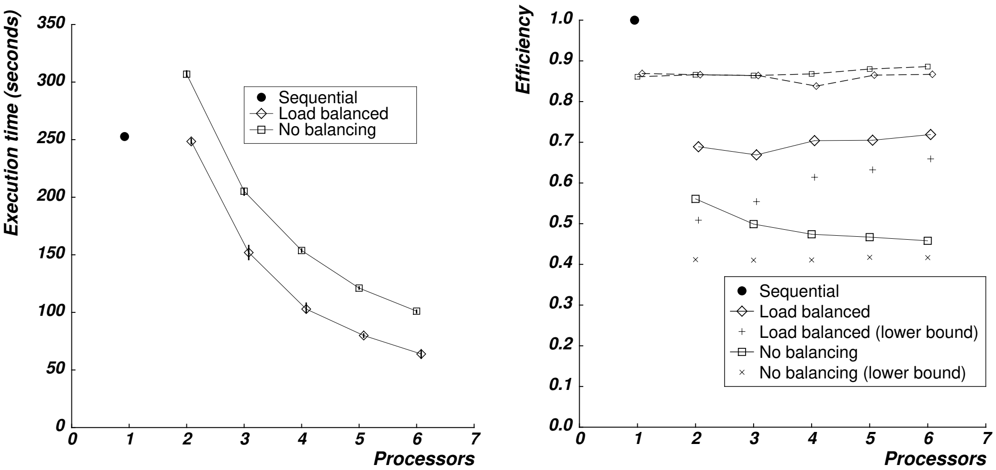


a) 500 × 500 MM

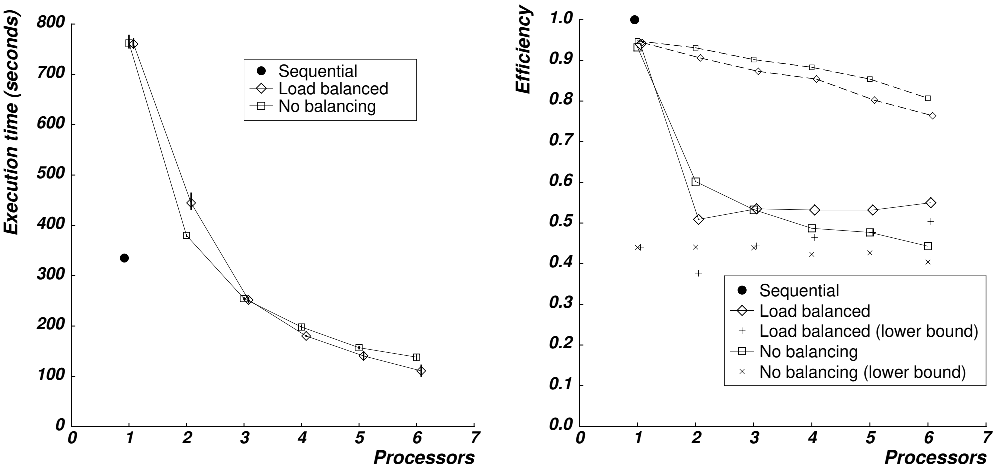


b) 1000 × 1000 SOR (40 iterations)

Figure 7.11: Execution time and efficiency in homogeneous environment with oscillating load (period = 6 sec, duration = 3 sec) on first processor.

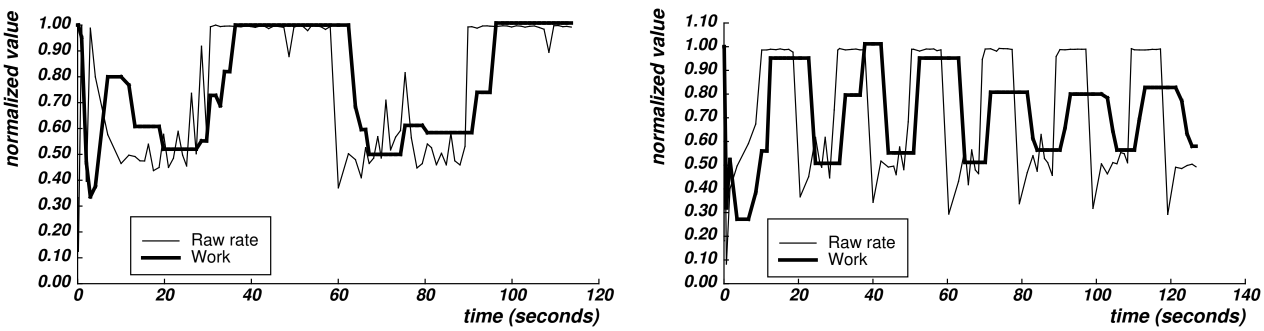


a) 500 × 500 MM



b) 1000 × 1000 SOR (40 iterations)

Figure 7.12: Execution time and efficiency in homogeneous environment with oscillating load (period = 2 sec, duration = 1 sec) on first processor.



a) Oscillation period = 60 seconds

b) Oscillation period = 20 seconds

Figure 7.13: Measured performance and work movement on loaded slave for 1000 × 1000 SOR (40 iterations) running on a 4 slave system with an oscillating load on one slave.

7.5 Modeling performance with oscillating loads

In this section we model the performance of several load balancing schemes with an oscillating load on one processor. We compare the performance predicted by the model for dynamic load balancing to the data presented in Section 7.4. We also compare the model to a model of the performance with equal distribution of work so that the conditions under which dynamic load balancing is beneficial can be determined. We also present performance models for other static load balancing schemes to demonstrate how performance can be improved with prior knowledge of the loads on the system.

To correspond with the data presented in this chapter, our models assume an oscillating competing load with equal on and off durations on the first processor. In Figure 7.14a, for example, the period of oscillation is $period_{osc}$, the competing load is off during t_a and on during t_b, and $t_a = t_b$. The observed computation rate on the loaded processor is proportional to the inverse of the total load on the processor. For example, when a single competing load is executing on a processor, the total load on the processor is 2 (including the load balanced application), and the computation rate is 0.5 times the rate that would be observed on the processor if it had no competing load. For convenience, we normalize rates against the rate that would be observed on a dedicated processor. Thus, the rate on the first processor oscillates between a minimum value, $rmin$, and the maximum value, 1.0. On the rest of the processors, which have no competing loads, the rate is always 1.0. For the measurements presented in this chapter, where there is at most one competing load, $rmin = 0.5$.

For all cases, the maximum amount of work that can be computed during one oscillation period is the same:

$$\begin{aligned} c_{available} &= \frac{t_a \times P + t_b \times (rmin + P - 1)}{period_{osc}} \\ &= \frac{0.5 \times period_{osc} \times P + 0.5 \times period_{osc} \times (rmin + P - 1)}{period_{osc}} \\ &= 0.5 \times rmin + P - 0.5 \end{aligned} \tag{7.1}$$

where the unit of measurement is the maximum performance provided by a single, dedicated processor during one oscillation period. For each type of load balancing, we estimate $c_{productive}$, and determine the efficiency using Equation 1.3:

$$efficiency = \frac{c_{productive}}{c_{available}}$$

Distribution	$c_{productive}$		Efficiency
	t_a portion	t_b portion	
Equal distribution of work	P	$P \times rmin$	$\frac{0.5 \times P \times (rmin + 1)}{0.5 \times rmin + P - 0.5}$
Avoid tracking changing load	$rmin + P - 1$	$rmin + P - 1$	$\frac{rmin + P - 1}{0.5 \times rmin + P - 0.5}$
Avoid loaded processor	$P - 1$	$P - 1$	$\frac{P - 1}{0.5 \times rmin + P - 0.5}$

Table 7.4: Modeling performance with static allocation of work.

The models presented in this section assume that the performance observed at any time is directly related to the current load on the system (i.e., no averaging of performance occurs). This assumption fits well with the SOR example because of its frequent synchronizations. For the MM example, which has no synchronization, the models should still be reasonably accurate if the oscillation period is several times the load balancing period so that the effects of performance averaging are minimized.

7.5.1 Static load balancing

There are many ways to allocate work to processors statically. Figure 7.14 shows boundary cases for the allocation of work on the processor with oscillating load. For each case, the remaining work is allocated equally to the other processors. The performance of these cases is summarized in Table 7.4 and graphed in Figure 7.15. In Table 7.4, the productive computation is estimated separately for the t_a and t_b time periods and combined to compute the efficiency during the whole oscillation period. The model for equal distribution of work gives slightly better results than the measurements, but matches the trends in the measurements quite well (Figures 7.19 and 7.20). Although static approaches such as those shown in Figures 7.14b and 7.14c are often more efficient than equal distribution of work, they are not generally applicable because they require prior knowledge of the loads that will be on the processors.

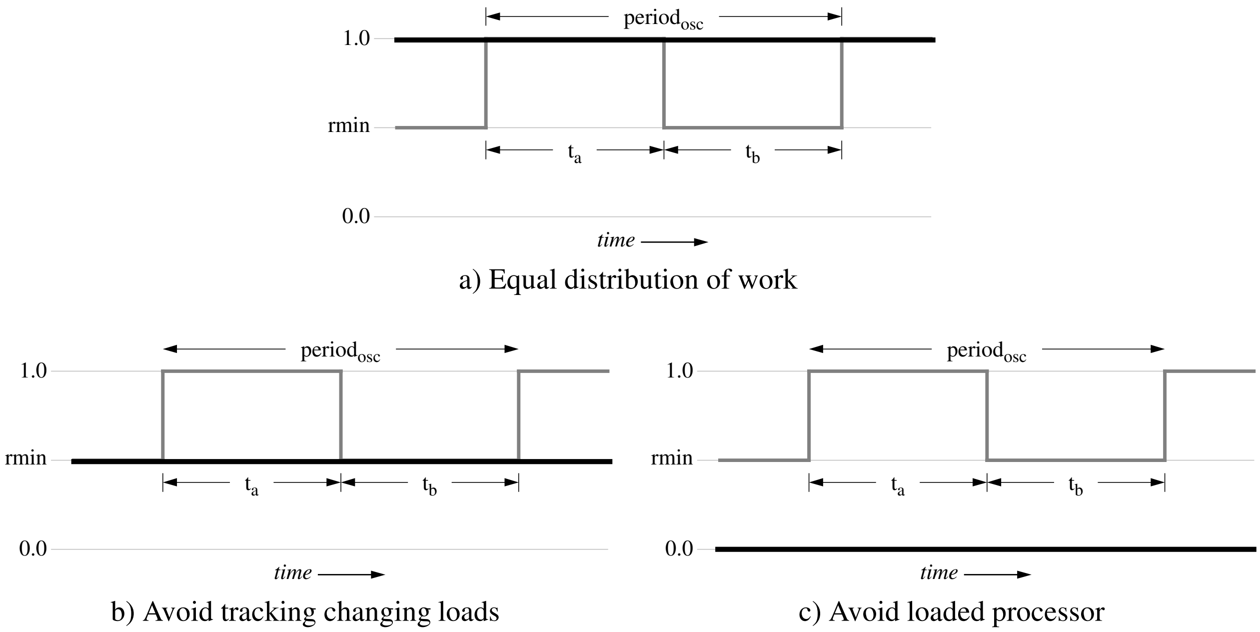


Figure 7.14: Static allocation of work. Heavy dotted line is computation rate on processor with competing load, normalized against rate on dedicated processor. Heavy solid line is work allocated to processor with competing load, normalized against work allocated with equal work distribution.

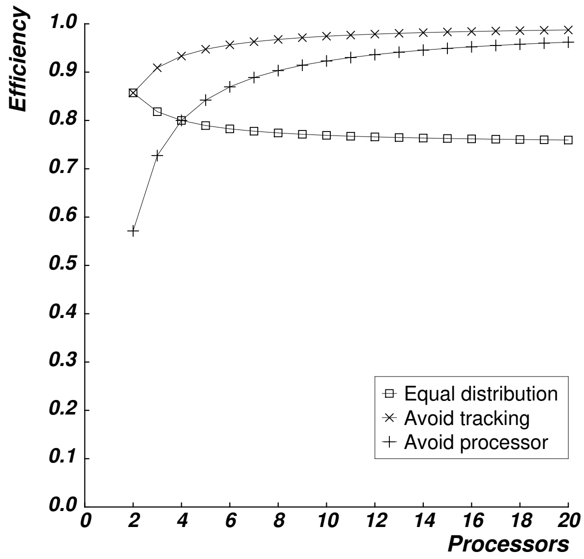


Figure 7.15: Performance of static load balancing approaches.

7.5.2 Dynamic load balancing

For dynamic load balancing, we divide the oscillation period up into several time segments, identified in Figure 7.16, to compute the overall productive use of resources. The time period when all processors are dedicated to the application, t_a, is divided into a period where load is not balanced, t_c, a period when work is being moved, t_d, and a period where load has been balanced, t_e. Similarly, the period when competing loads are active, t_b, is divided into t_f, t_g, and t_h. For each of these time segments, the amount of processing resources used productively will be determined so that the total productive use of resources can be computed:

$$c_{productive} = \frac{(t_c \times u_c) + (t_d \times u_d) + (t_e \times u_e) + (t_f \times u_f) + (t_g \times u_g) + (t_h \times u_h)}{t_c + t_d + t_e + t_f + t_g + t_h} \tag{7.2}$$

For each time segment, s, we must determine the length of the segment, t_s, and the resources used productively during the segment, u_s. The lengths of the segments are related as follows:

$$period_{osc} = \frac{1}{frequency_{osc}} = t_c + t_d + t_e + t_f + t_g + t_h \tag{7.3}$$

$$0.5 \times period_{osc} = t_c + t_d + t_e = t_f + t_g + t_h \tag{7.4}$$

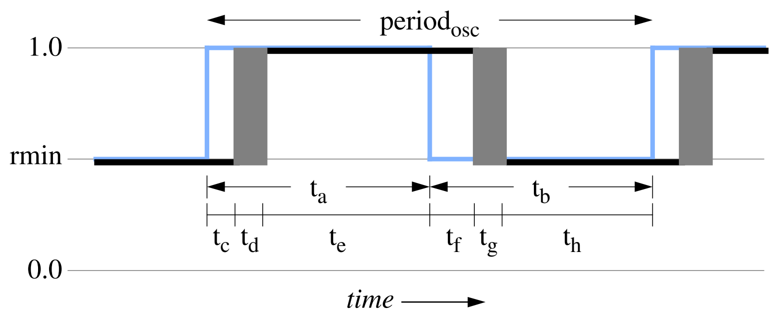


Figure 7.16: Dynamic load balancing model for predicting performance.

The performance model presented in this section breaks down when t_e or t_h is less than zero. Also, the model does not include the effects of filtering, use of a threshold for imbalance detection, or cancelling instructions based on a cost-benefit analysis. Because the model assumes an ideal input with clean transitions, imbalance detection is not an issue unless the amplitude of the oscillations is less than the imbalance threshold. At about the same oscillation frequencies where the model breaks down due to large delays or high work movement costs, the filtering and cost-benefit analysis start having significant effects on performance, so these optimizations make extending the frequency range covered by the model even more difficult. Our current model is most accurate when the oscillation period is at least an order of magnitude greater than the load balancing period.

Load imbalance (t_c and t_f)

On average, the load balancer is not notified of a change in performance until one half of a load balancing period after the change occurs; during this period, the load remains unbalanced, and one or more of the processors is not fully utilized. If the load balancing interactions are pipelined, there is an additional delay of $depth$ load balancing periods, where $depth$ is the pipeline depth ($= 1$ for the measurements presented in this chapter). Because the length of the load balancing period is determined by the amount of work computed during the period, $nexthook$ (Section 4.1), rather than clock time, the length changes when the rate changes. The instruction to correct the period is received on the slaves at the same time as the work movement instructions. Thus,

$$t_c = (depth + 0.5) \times rmin \times period_{target} \tag{7.5}$$

$$t_f = (depth + 0.5) \times \frac{1}{rmin} \times period_{target} \tag{7.6}$$

For our analysis, we assume $period_{target} = 1.0$ seconds (from Section 4.3.4).

During t_c, the processor with the oscillating load is underutilized and must wait for the other processors to finish computing:

$$u_c = rmin + P - 1 \tag{7.7}$$

During t_f, the processor with the oscillating load has slowed down. All other processors must wait for the loaded processor to finish computing and are thus limited to its slower rate:

$$u_f = rmin \times P \tag{7.8}$$

Work movement (t_d and t_g)

During work movement, no resources are used productively so $u_d = u_g = 0$. However, values for t_d and t_g are still needed so that t_e and t_h can be computed. The total size of the distributed data and the number of processors involved in the work movement are the main factors in the cost of work movement. An additional factor is restrictions on work movement which determine whether intermediate processors are involved when work is moved: for applications with restricted work movement, the estimated costs must be multiplied by the number of intermediate processors that must transfer the work.

Data size. Dynamic load balancing performance varies greatly between applications. The differences in the efficiencies between the applications are largely due to the amount of data that must be moved when shifting work. For the MM example, portions of two matrices (**B** and **C**) must be shifted, with the total size of each matrix $500 \times 500 \times 4 = 1$ megabyte. For SOR, there is only one matrix, but the 1000×1000 matrix is 4 megabytes, and the 2000×2000 matrix is 16 megabytes. Table 7.5 summarizes the work movement costs for the MM and SOR examples. $m_{wholearray}$ is an estimate of the time to move the whole distributed array between two processors, extrapolated from measurements of the time to move a single work unit. When shifting work, each slice is sent as a single message so the per unit cost, the total cost for sending one slice of the data, includes both the initial (per message) and incremental (per byte) costs (Section 5.6.1).

Application	total units	total size (MB)	per unit cost (msec)	$m_{wholearray}$ (msec)
500×500 MM	500	2	5.46	2730
1000×1000 SOR	1000	4	6.91	6910
2000×2000 SOR	2000	16	11.99	23980

Table 7.5: Work movement costs used in modeling performance. Average of measurements taken at startup time for greater than 50 runs.

Number of processors. Work movement costs also vary with the number of slave processors. For a small number of slaves, work movement costs are higher than with more processors because a larger fraction of the distributed data is shifted each time the rate of computation changes. For example, in a two slave system with a single oscillating load, the work allocated to the loaded processor alternates between one half and one third of the total work; thus, $\frac{1}{2} - \frac{1}{3} = \frac{1}{6}$ of the total work is shifted each time the rate changes. However, for a three slave system, work allocated to the loaded processor alternates between one third and one fifth of the total; only $\frac{1}{3} - \frac{1}{5} = \frac{2}{15}$ of the total is shifted each time. The amount of work shifted each time the rate changes continues to decrease as the number of processors is increased (Figure 7.17):

$$fractionmoved = \frac{1}{P} - \frac{1}{2 \times P - 1} \tag{7.9}$$

Unrestricted work movement. Because unrestricted work movement can occur in parallel, we estimate the time for unrestricted work movement based only on the processor that moves the most work (Section 5.6.1). Thus, each time the rate changes, the total cost of work movement for unrestricted work movement

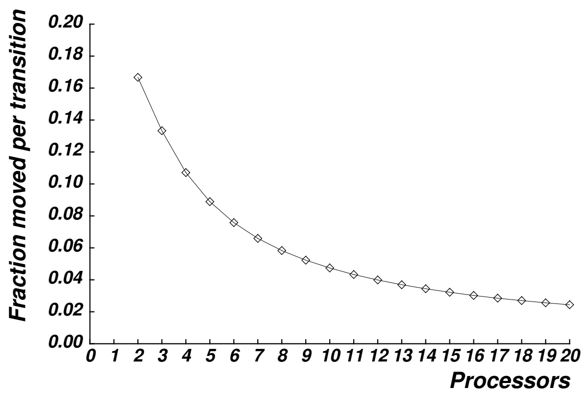


Figure 7.17: Fraction of total distributed matrix moved to or from the loaded processor (the first processor in the logical array) with each transition of the oscillating load.

is

$$t_d^{unrestricted} = t_g^{unrestricted} = (\frac{1}{P} - \frac{1}{2 \times P - 1}) \times m_{wholearray} \tag{7.10}$$

Restricted work movement. For applications with restricted work movement, the work movement costs must be adjusted to account for time spent shifting work through intermediate processors. Ideally, the work would be shifted through the processors in parallel, using the ordering of messages described in Section 5.5.2, and the work movement cost would be determined by the processor shifting the most work, as in the unrestricted case. Unfortunately, due to poor flow control in Nectar, each work movement message for the SOR example required an acknowledgement, forcing work to be transferred sequentially. In Section 5.6.1, a multiplier for the work movement cost was computed by average number of hops between arbitrary processors in a linear array ($\frac{P+1}{3}$). However, in this case, the actual work movement patterns are known due to our knowledge of the load patterns and the implementation. The patterns are shown in Figure 7.18. Accurately modeling the time spent on copying data is complicated because of the possibility for overlap between work movement and productive computation; thus, our estimates for the total work movement costs will be conservative, i.e., high. However, the cost estimates for unrestricted work movement can be considered as lower bounds on the costs for restricted work movement.

When the first processor ($P0$ in Figure 7.18) slows down, work is passed sequentially from the first processor through all the other processors, leaving $\frac{1}{P-1}$ of the total moved work on each intermediate

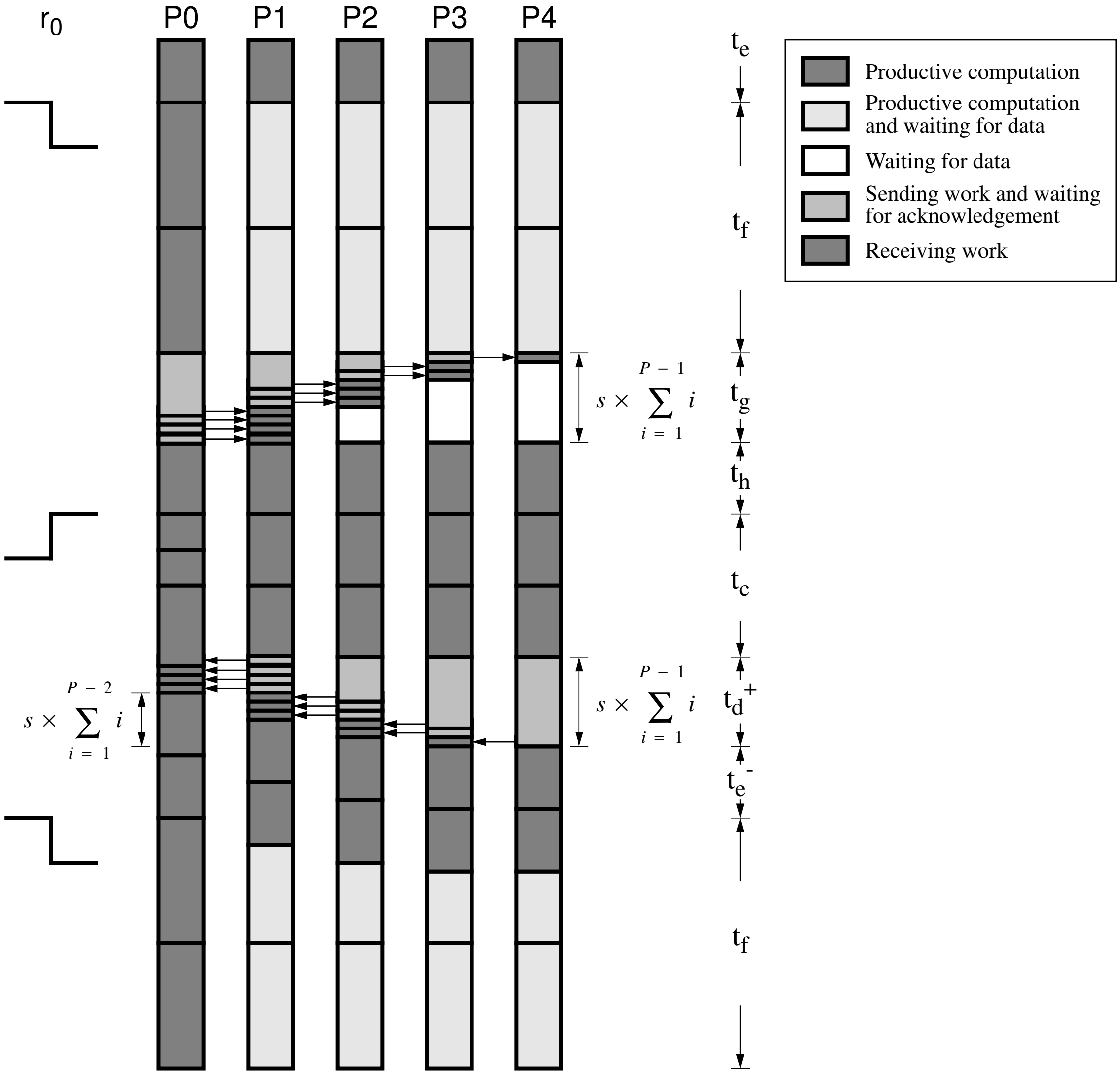


Figure 7.18: Work movement patterns for restricted work movement.

processor. Thus, the work movement takes

$$\sum_{i=1}^{P-1} i = \frac{(P-1) \times P}{2}$$

steps of length

$$s = \frac{t_d^{unrestricted}}{P-1} \tag{7.11}$$

Because there is an acknowledgement for each message, the work movement actually finishes first on the last processor in the processing pipeline. When work is passed to the right, very little computation can occur in parallel with the work movement because all processors must wait for $P0$, the processor at the beginning of the pipeline, to send intermediate data through the pipeline. (It may be possible for the beginning of t_g to overlap with t_f, but only during the last pipeline phase of t_f. The size of the overlap depends on the grain size for the application and is very complicated to model.) Therefore,

$$\begin{aligned} t_g^{restricted} &= \frac{(P-1)\times P}{2} \times \frac{t_d^{unrestricted}}{P-1} \\ &= \frac{P}{2} \times t_d^{unrestricted} \\ &= \frac{P}{2} \times (\frac{1}{P} - \frac{1}{2\times P - 1}) \times m_{wholearray} \end{aligned} \tag{7.12}$$

The work movement during $t_d^{restricted}$ takes the same number of steps and the same amount of time as during $t_g^{restricted}$. However, when work is passed to the left, there is overlap between the work movement and the following computation, t_e, because $P0$ is the first processor to finish with work movement. To compensate for the overlap, we scale the estimate of the work movement costs:

$$t_d^{restricted} = t_g^{restricted} \times overlapscale \tag{7.13}$$

overlapscale is computed by subtracting the total time spent on productive work (on all processors) from the total CPU time (on all processors) computed for work movement assuming no overlap:

$$overlapscale = \frac{cpu_{nooverlap} - cpu_{productive}}{cpu_{nooverlap}} \tag{7.14}$$

$$cpu_{nooverlap} = P \times t_g^{restricted} \tag{7.15}$$

$$cpu_{productive} = s \times \sum_{i=1}^{P-2} (\sum_{j=1}^{i} j) \tag{7.16}$$

where s is the time for each work movement step (Equation 7.11).

$t_d^{unrestricted}$ and $t_g^{unrestricted}$ are used in modeling the efficiency of MM with dynamic load balancing, and $t_d^{restricted}$ and $t_g^{restricted}$ are used in modeling the efficiency of SOR. $t_d^{unrestricted}$ and $t_g^{unrestricted}$ are also used in computing an upper bound on the efficiency of the SOR example.

Balanced load (t_e and t_h)

Given the oscillation period and the values of t_c, t_d, t_f, and t_g determined in the preceding paragraphs, the lengths of segments t_e and t_h are computed using Equation 7.4:

$$t_e = 0.5 \times period_{osc} - t_c - t_d \tag{7.17}$$

$$t_h = 0.5 \times \textit{period}_{osc} - t_f - t_g \tag{7.18}$$

Since the load is balanced during t_e and t_h, all available resources are used productively:

$$u_e = P \tag{7.19}$$

$$u_h = rmin + P - 1 \tag{7.20}$$

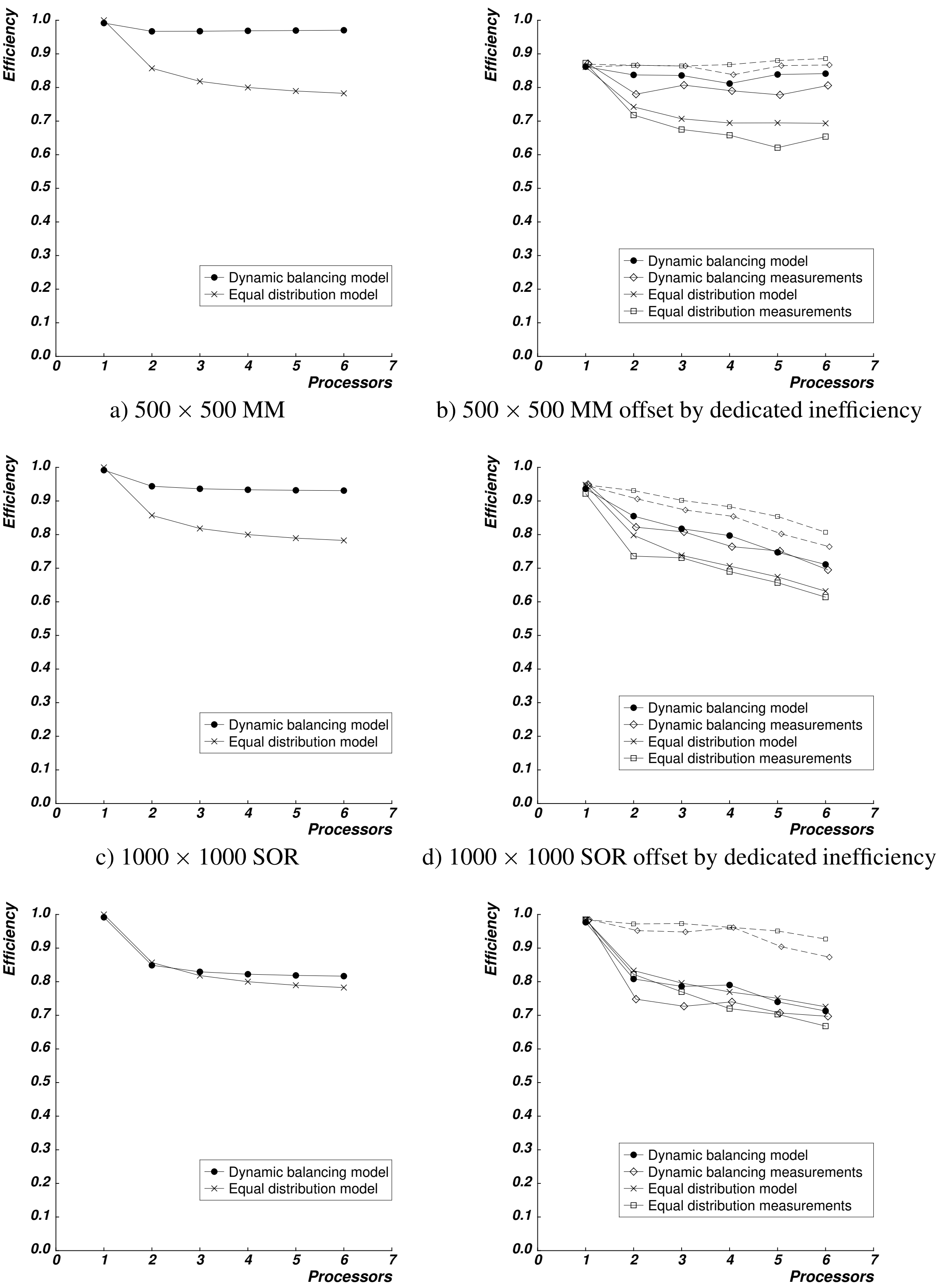
Efficiency

Once all components of Equation 7.2 have been determined, the efficiency can be computed. The only run-time information required for the model (so far) is estimates of the time to move data slices between processors (Table 7.5). Figures 7.19a,c,e and 7.20a,c,e show the computed efficiencies for two oscillation periods for the MM and SOR examples. The predicted efficiencies with equal distribution of work are also shown. (For the 2 and 6 second oscillation periods, t_e and/or t_h is less than zero, so the model is not applicable.)

Figures 7.2, 7.3, and 7.4 indicate that there is inefficiency even when load is balanced. This inefficiency is due to modifications to data structures, changes in access patterns, and communication required by the parallelized application and, for the load balanced versions, the load balancing interaction costs. The inefficiency only affects the times when the processors are actively working on the computation, i.e., not the work movement costs. Thus, to account for this inefficiency, we multiply the estimates of u_c, u_e, u_f, and u_h by the efficiencies measured for the parallel code with load balancing on the corresponding dedicated homogeneous systems (from Figures 7.2c, 7.3c, and 7.4c). The model for performance with equal distribution of work is multiplied by the dedicated homogeneous efficiencies measured for the parallel code without load balancing. The recomputed efficiency results based on the modified models are shown in Figures 7.19b,d,f and 7.20b,d,f. For comparison, these graphs also include the measured efficiency values collected in the same environments. For reference, the measurements from the dedicated homogeneous environment are also shown (with smaller symbols connected by dashed lines).

For all of the examples, the plots of the model for equal distribution of work parallel the plots of the corresponding measurements, although the model predicts slightly higher efficiencies.

For the MM example, the slopes of the efficiency curves for the dynamic load balancing model and the measurements are similar, but the model produces higher efficiencies. The efficiencies predicted by the



a) 500 × 500 MM

b) 500 × 500 MM offset by dedicated inefficiency

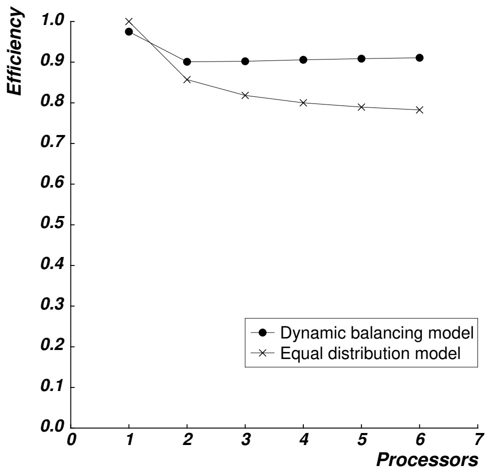
c) 1000 × 1000 SOR

d) 1000 × 1000 SOR offset by dedicated inefficiency

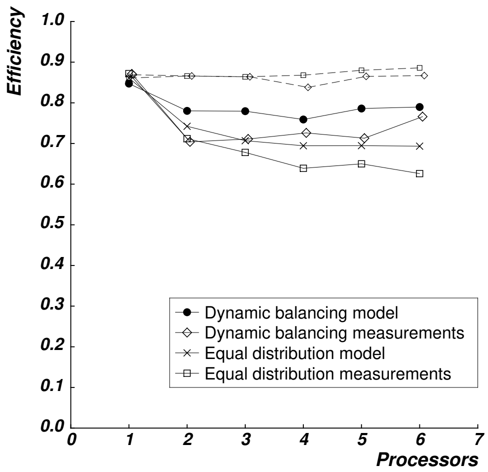
e) 2000 × 2000 SOR

f) 2000 × 2000 SOR offset by dedicated inefficiency

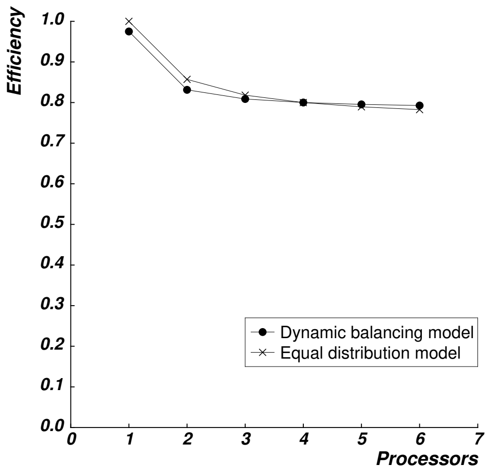
Figure 7.19: Efficiency predicted by dynamic load balancing model for homogeneous environment with oscillating load (period = 60 sec, duration = 30 sec) on first processor.



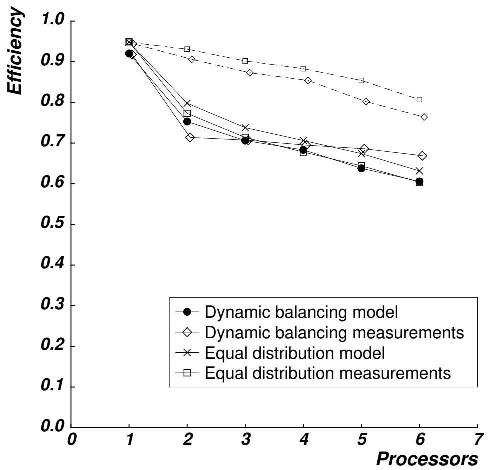
a) 500 × 500 MM



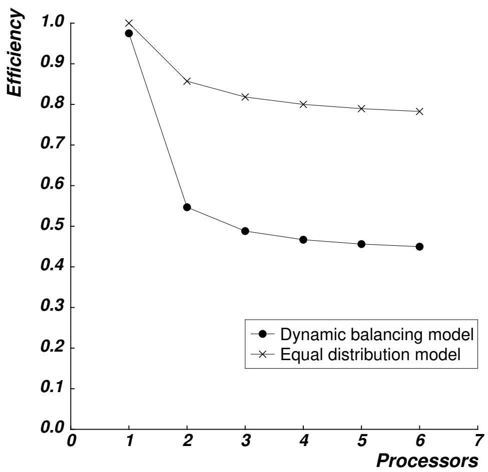
b) 500 × 500 MM offset by dedicated inefficiency



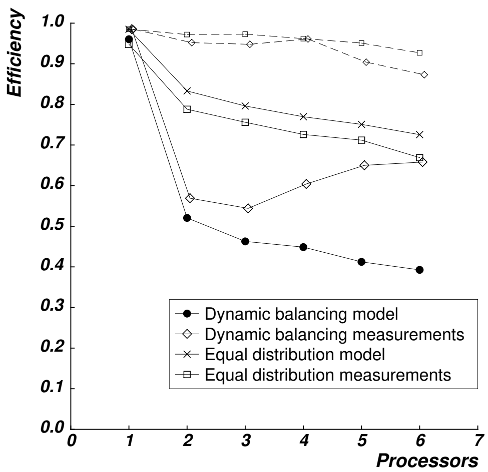
c) 1000 × 1000 SOR



d) 1000 × 1000 SOR offset by dedicated inefficiency

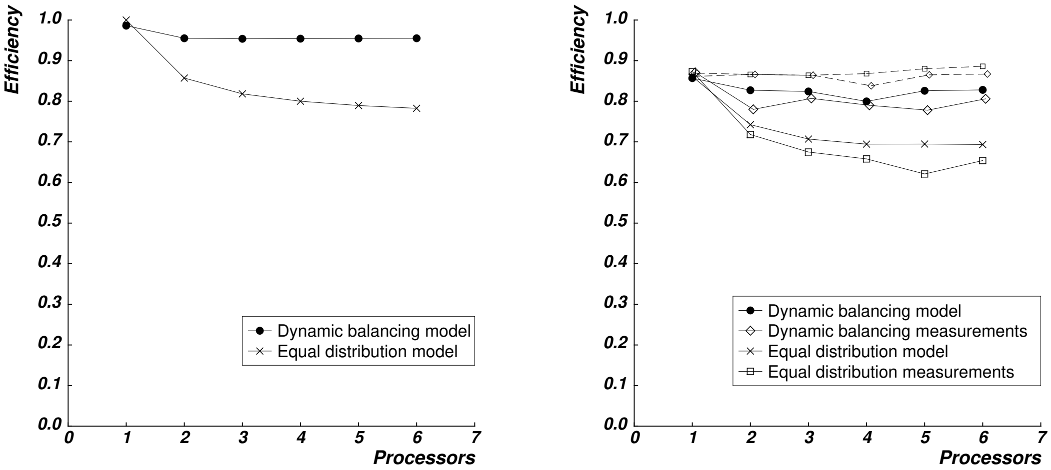


e) 2000 × 2000 SOR



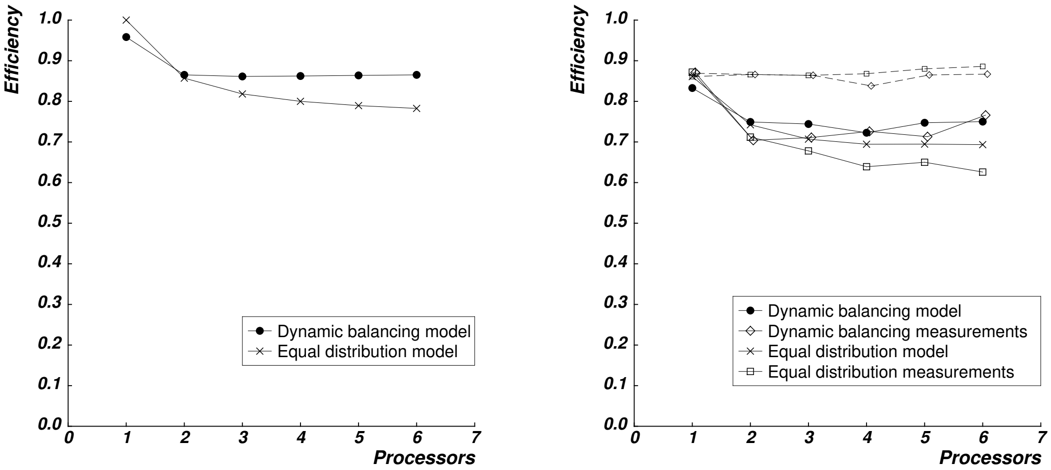
f) 2000 × 2000 SOR offset by dedicated inefficiency

Figure 7.20: Efficiency predicted by dynamic load balancing model for homogeneous environment with oscillating load (period = 20 sec, duration = 10 sec) on first processor.



a) 500 × 500 MM b) 500 × 500 MM offset by dedicated inefficiency

Figure 7.21: Efficiency predicted by dynamic load balancing model for homogeneous environment with oscillating load (period = 60 sec, duration = 30 sec) on first processor. $t_c = t_f = 2.5$ load balancing periods.

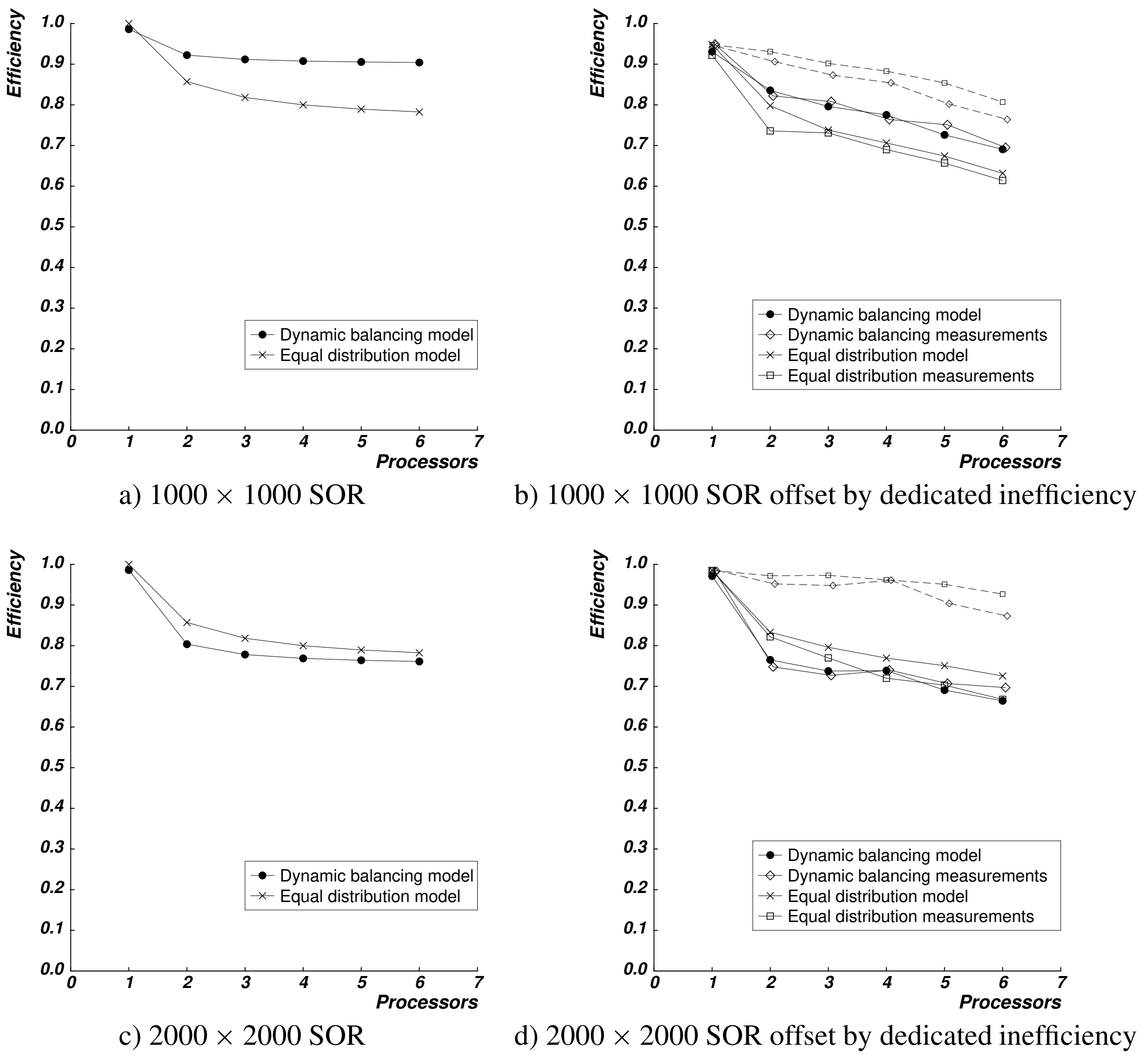


a) 500 × 500 MM b) 500 × 500 MM offset by dedicated inefficiency

Figure 7.22: Efficiency predicted by dynamic load balancing model for homogeneous environment with oscillating load (period = 20 sec, duration = 10 sec) on first processor. $t_c = t_f = 2.5$ load balancing periods.

model are higher partly because the model does not consider the filtering of the measured rate information. As a result of the filtering, work is not shifted all at once to reach the balanced distribution. It takes at least 2 or 3 load balancing periods (on average, about 5 periods) after a rate change for the balanced distribution to be reached. If the response times (t_c and t_f) in the model are increased by 1 load balancing period, the model and the measurements move closer (Figures 7.21 and 7.22). There may also be some error in the estimates of work movement costs.

For SOR, the accuracy of the efficiency model changes with the frequency of oscillation. For the 60



a) 1000 × 1000 SOR

b) 1000 × 1000 SOR offset by dedicated inefficiency

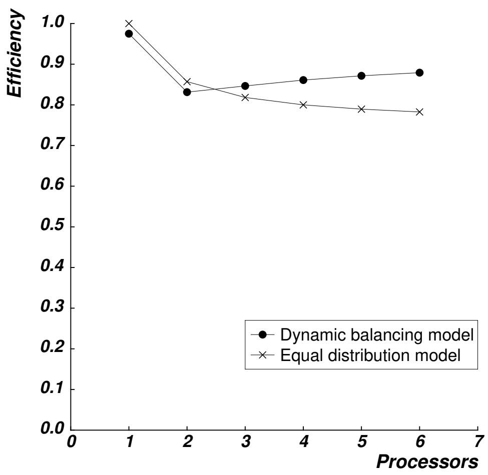
c) 2000 × 2000 SOR

d) 2000 × 2000 SOR offset by dedicated inefficiency

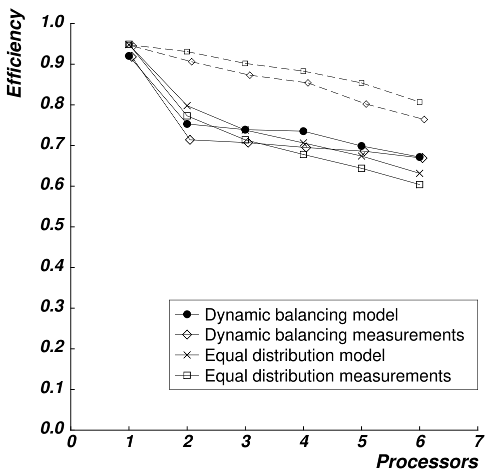
Figure 7.23: Efficiency predicted by dynamic load balancing model for homogeneous environment with oscillating load (period = 60 sec, duration = 30 sec) on first processor. $t_c = t_f = 2.5$ load balancing periods. Work movement cost estimates 25% higher than values listed in Table 7.5.

second period, the model produces efficiency curves that parallel the measured efficiency curves (Figure 7.19d,f). The curves are almost coincident if the response time is increased by 1 load balancing period and the estimates of the per slice work movement costs are increased by 25% (Figure 7.23). For the SOR examples, because work movement costs are high due to the large size of the distributed data, a 25% change in the cost estimates has a large effect on the efficiency predicted by the model. For the MM example, which has a smaller data set, scaling the work movement cost estimates has a much smaller effect.

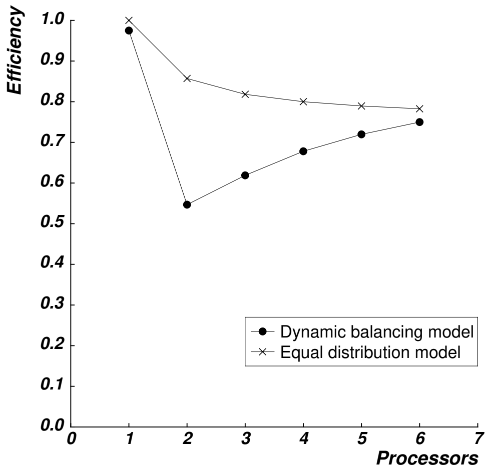
For the 20 second period, the slopes of the efficiency curves for the dynamic load balancing model and the measurements are noticeably different (Figure 7.20d,f), especially for the 2000 × 2000 problem



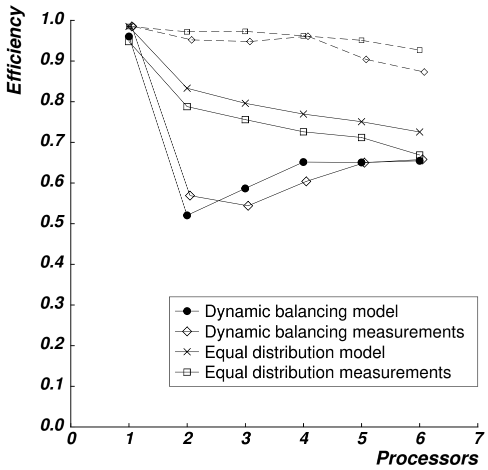
a) 1000 × 1000 SOR



b) 1000 × 1000 SOR offset by dedicated inefficiency



c) 2000 × 2000 SOR



d) 2000 × 2000 SOR offset by dedicated inefficiency

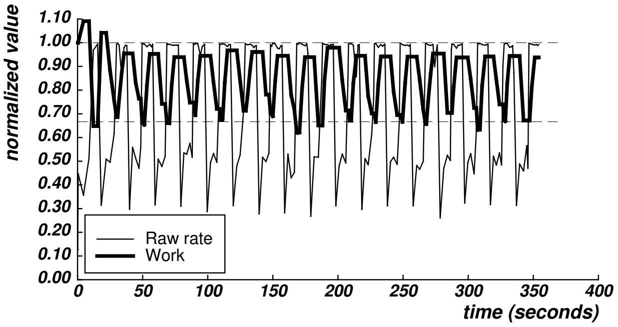
Figure 7.24: Efficiency predicted by dynamic load balancing model for homogeneous environment with oscillating load (period = 20 sec, duration = 10 sec) on first processor, assuming unrestricted work movement.

size. For the measurements, the efficiency curves stay level or slope upward as the number of processors is increased, while for the model, the curves always slope downward. Changing the response time or the estimate of the time to transfer one data slice do not correct the problem. In fact, the model for applications using unrestricted work movement produces better approximations of the measured efficiencies (Figure 7.24). This indicates that work movement costs are not as high as expected.

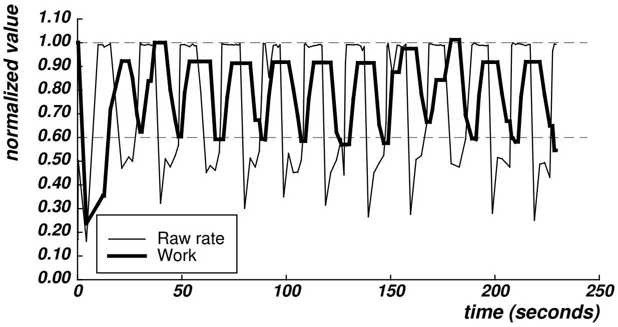
A probable reason for the lower work movement costs is that less work is being moved than predicted by the model. As the number of processors increases, because of the decreasing benefits of moving work onto the loaded processor, the imbalance detection and profitability determination phases of the load balancer start to have significant effects on the amount of work moved: the threshold for detecting load imbalance reduces work movement by allowing the load to remain in an unbalanced state, and the profitability determination phase cancels work movement instructions. The result of these optimizations is that only part of the work for balancing the load gets moved before the competing load changes again.

The reduction in work movement due to the threshold depends on the timing of events in the system, and is somewhat random. For cases where work movement costs are low, imbalance detection sometimes results in decreased performance because the system is left unbalanced (e.g., Figure 5.4c), but in this case, the results are beneficial because high work movement costs can be avoided.

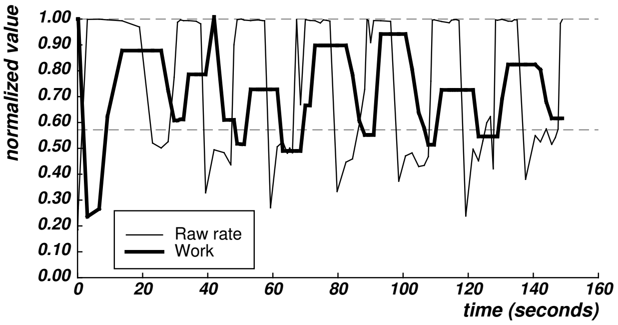
To compute projected benefits, the profitability determination phase is determining the stability of the system by estimating the frequency of substantial performance changes in the system. (This method of estimating stability corresponds well with the oscillating loads on the system, but may not work as well with systems with other load characteristics.) When the oscillation period is decreased from 60 seconds to 20 seconds, the estimate of system stability decreases, and the same high cost of shifting data slices may exceed the smaller projected benefits. As the number of processors is increased, the benefit of moving work back to the loaded processor decreases, and work movement instructions are more likely to be cancelled. With enough processors, the system stops tracking the changing portion of the performance, as in the "avoid tracking" static scheme. Figure 7.15 shows that with the "avoid tracking" static scheme, efficiency exceeds 90% for systems with greater than two processors, so if work movement costs are high, they are likely to outweigh the small benefit of balancing the load. Figure 7.25 demonstrates how the amount of work shifted decreases relative to the amounts predicted by Equation 7.9 as the number of processors is increased. The dashed lines indicate the expected allocation of work on the loaded processor during the periods when the competing load is active (lower line) and inactive (upper line). The difference between the actual allocation and the expected allocation during periods when the competing load is not active increases as the number of processors increases due to the threshold for detecting load imbalance. For the system with 6 slaves (Figure 7.25e), some transitions in load are ignored completely because work movement instructions are cancelled by the profitability determination phase. The reduction in work movement when benefits of movement are small helps increase the efficiency in situations where performance is dominated by the load balancing overhead, although not necessarily enough to result in a net improvement over the case with equal distribution.



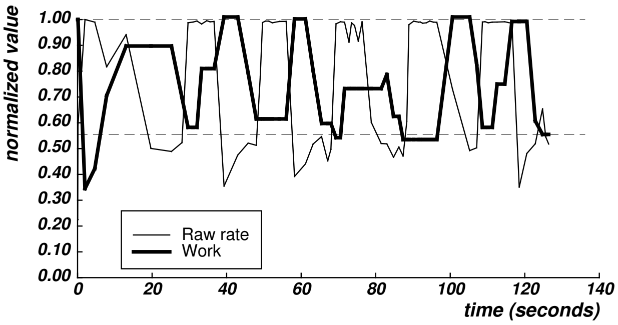
a) 2 slave system



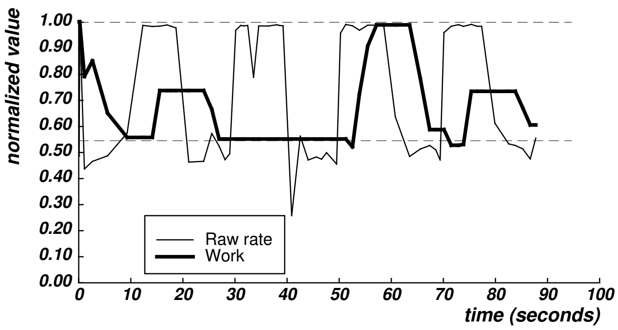
b) 3 slave system



c) 4 slave system



d) 5 slave system



e) 6 slave system

Figure 7.25: Measured performance and work movement on loaded slave for 2000×2000 SOR (10 iterations) running on systems with an oscillating load (period = 20 sec, duration = 10 sec) on one slave. Work movement decreases relative to expectations as the number of processors increases because of imbalance detection and profitability determination. Dashed lines show expected ranges for work allocation.

7.5.3 Improving the model/Improving the system

For more accurate modeling, the effects of imbalance detection, filtering, and the profitability determination phase should be taken into account. However, these optimizations are difficult to model because they affect both work movement costs and the degree of imbalance in the "unbalanced" and "balanced" portions of the oscillation period. Simulation is probably an easier approach for including the effects of these optimizations because the optimizations have some dependence on past events in the execution of the program. The model should also be extended to handle higher frequency oscillations, where work allocations and changes in performance are more than one half of the oscillation period out of phase, i.e., t_e or t_h is less than zero.

Work movement costs are very complicated and are implementation dependent, making them difficult to estimate accurately. Measurements of the costs at the start of program execution may not be accurate over the entire run of the program due to contention in the network or occasional unreliability (e.g., dropped packets) of the network. An additional factor that was not included in the model is the effect of the loads on the processors on the costs of work movement; to account for this, both the loads on the senders and the receivers must be considered in the costs.

The modeling helped identify some deficiencies in the system. We discovered that imbalance detection and profitability determination are having the desired effect of reducing work movement costs, but, in some cases, they do not reduce the costs enough for dynamic load balancing to result in an overall gain in performance. For applications with large data slices running on systems with rapidly changing performance, work movement instructions should be cancelled (or altered) more aggressively. Further investigation of the imbalance detection, filtering, and profitability determination optimizations and, possibly, other optimizations for reducing unnecessary work movement is necessary.

7.6 Limits of dynamic load balancing approach

The same factors that limit the range of frequencies for which performance can be easily modeled also limit the load balancing systems ability to deal well with high frequency changes in load. When t_e or t_h is less than zero, work movement is completely out of phase with the changes in load and is unlikely to result in good performance. Fortunately, selection of an appropriate load balancing frequency, adding hysteresis with imbalance detection, filtering raw measurements, and doing a cost-benefit analysis to limit work movement all help prevent the system from responding to high frequency changes. When the frequency of changes is high, the system sees the average performance over a computation period and need not respond to each change in load. These averaging effects are difficult to model and are limited by synchronizations required by the application. These techniques do not completely eliminate the problems with rapidly changing loads, as demonstrated in Figure 7.26, but Figures 7.11 and 7.12 indicate that in some cases, dynamic load balancing using the techniques still results in performance improvements in rapidly changing environments.

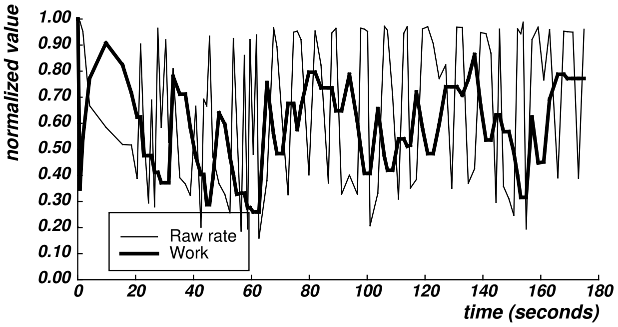


Figure 7.26: Measured performance and resulting work allocation on loaded slave for 1000 $\times$ 1000 SOR (40 iterations) running on a 4 slave system with an oscillating load (period = 2 seconds, duration = 1 second) on one slave.

7.7 Summary

In this chapter, we presented data collected for the SOR and MM applications run on the Nectar system under different load conditions. The efficiency in using available resources is the primary measure of performance used to evaluate and compare different runs. We compared performance with dynamic load balancing to performance with a static, equal distribution of work. Runs in a dedicated homogeneous environment showed that overhead added by load balancing is small and within the bounds expected with our method of frequency selection (i.e, approximately 5% overhead, from Chapter 4). Runs with a constant load on one of the processors showed that load is redistributed correctly and improves performance; the efficiency is approximately the same as in the dedicated homogeneous environment. In more dynamic environments, with an oscillating load on one of the processors, the performance improvements with dynamic load balancing were smaller, and in some cases the performance dropped. For the MM example, the parallelization efficiency was improved by load balancing in almost every case, but there still seems to be room for significant improvement. For the SOR examples, load balancing was much less effective and actually decreased performance in many of the environments.

To identify causes of inefficiency, we modeled the efficiency of our dynamic load balancing system for systems with an oscillating load on one processor. For the MM example, both the work movement costs and the delay in responding to changes in loads contributed to the inefficiency. However, for SOR, work movement costs were the dominant factor. In our load balancing system, the work movement costs increase with the size of the distributed data, so the larger data sizes used for the SOR example account for part of its higher work movement costs. In addition, for applications with restricted work movement, like

SOR, intermediate processors are involved in moving work. In Section 5.5.2, we presented an algorithm which maximized parallelism in work movement, but in our implementation, we were unable to obtain the parallelism due to poor flow control in the network. Because the work is not shifted through the processors in parallel, the work movement costs in the model must be scaled by a factor approximately proportional to the number of processors (i.e., $O(P)$), and performance drops as the number of processors is increased. With parallel work movement, however, performance increases as processors are added because the time for the work movement is approximately proportional to $\frac{1}{P}$. This highlights the importance of parallelism in moving work and the need for good flow control in the network to make implementing the parallel movement possible (without significant effort).

We compared the model of dynamic load balancing performance with the measurements on the Nectar system. The model matches the measurements when the competing load has a low oscillation period, but for higher frequency load changes, especially for applications with high work movement costs, the model produces lower efficiencies than the measurements due to the effects of the imbalance detection and profitability determination optimizations not included in the model. These optimizations were successful in reducing work movement costs, but did not always reduce the costs enough to result in net improvements in performance for dynamic load balancing compared with a static, equal distribution of work. Further investigation of optimizations to reduce work movement costs is needed.

Chapter 8

Conclusions

Parallelizing compilers for networks of processors that are shared with other users must generate efficient code that supports dynamic load balancing. In this thesis we presented an architecture for a system that supports the automatic generation of parallel programs with dynamic load balancing. In our system the parallelizing compiler generates code that includes calls to a run-time load balancer. The load balancer generates work movement instructions taking into account application-specific features so that data locality and data reuse are maximized to minimize communication costs. The target application domain for the compiler is applications consisting of parallelized DOALL and DOACROSS loops. We described how the compiler and runtime system cooperate to automatically select parameters for dynamic load balancing and control of grain size. We implemented a runtime system and described the additional code the compiler must generate to support load balancing. Performance measurements based on two hand-parallelized applications showed that dynamic load balancing can be effective in improving parallelization efficiency and reducing execution time of applications in an environment with an oscillating load on one processor. However, performance of dynamic load balancing is limited by the cost of work movement and the delay in reaction to changing loads, especially for applications with large distributed data sizes.

8.1 Contributions

This thesis demonstrated the feasibility of having a parallelizing compiler generate efficient code that supports dynamic load balancing on a network workstations. Application-specific knowledge provided to the runtime system by the compiler is used to aid in load balancing decisions so that load balancing can be effective and have low overhead.

Dynamic loop scheduling. This thesis described a new approach to load balancing for automatically parallelized applications on networks of workstations. Most prior approaches for scheduling of loop iterations do not fully exploit locality in application programs, but our approach, by retaining the structure of the application code and considering dependences in the code, improves data reuse and reduces the need for communication. Also, our load balancing system automatically adjusts to the characteristics of the application and target environment to maximize performance. In the thesis we identified and addressed several performance issues, including grain size, load balancing frequency, the load balancing interaction cost, and the cost of work movement. We demonstrated our approach with an implementation of a load balancing runtime system, and we evaluated the performance of two hand-coded applications with load balancing running in several environments. To evaluate the performance of programs with dynamic load balancing, we designed an efficiency measure appropriate for a dynamic, heterogeneous environment. We also presented a model for predicting the performance of a load balanced application in an environment with an oscillating load on one processor; the model is consistent with our measurements.

Grain size on a loosely-coupled, shared system. For good performance on a distributed system, an application must have an appropriate grain size. We described and demonstrated a method for selecting the optimal grain size for DOACROSS loops based on both communication costs and parallelism. The compiler and runtime system cooperate in selecting and controlling the grain size. At run time, grain size is controlled by setting the block size of a strip-mined loop. A similar method for selecting grain size is used in the Fortran D compiler [26], but with a less flexible method of estimating the computation and communication costs for the application [23]. However, most previous methods for controlling grain size [61, 65] only consider communication costs. Because our target system may have competing loads, grain size may interact with the scheduling of processes by the operating system. We simulated this interaction and found that it has little effect on the performance of pipelined, DOACROSS loops, but, for parallel, DOALL loops, grain size should be as large as possible to maximize and stabilize performance.

Load balancing algorithms. Several of the ideas and techniques developed in this work can be used in other load balancing systems. Our idea of using computation rate as a measure for comparing performance on a dynamic, heterogeneous set of processors has been proposed before [40, 41], but we have introduced ideas from control theory and signal processing, such as sampling frequency and filtering, to eliminate undesirable fluctuations in the performance measurements, resulting in better total performance for the load balanced application. We also described a method for quantifying load imbalance based on our performance measure (or any other measure that quantifies the relative capabilities of the different processors) and the current work distribution; this measure can be used to decide when load balancing should be performed.

Parallelizing compilers. We described the modifications to a parallelizing compiler needed to support dynamic load balancing. Many of the necessary changes are due to having to deal with dynamic, irregular data distributions. We described data structures that can be used to manage such distributions with various application requirements and in different environments. We also described how communication code must be modified to deal with the dynamic distributions.

8.2 Areas for future work

This work described the features that are needed in a parallelizing compiler to support dynamic load balancing. The next step is to incorporate the features into a parallelizing compiler so that many more applications can be run and evaluated with our load balancing system. Since the prototype Nectar system [3] has been decommissioned (R.I.P., JUNE 29, 1994), the runtime system should be reimplemented on a newer architecture, using a more portable message-passing interface, such as PVM [63] or MPI [19]. Because these interfaces are more portable, however, we expect communication latencies to be much higher than with Nectarine [57], which was designed specifically for Nectar. The automatic calibration features of our load balancing system, e.g., for grain size and frequency control, should be very helpful in moving between different machines once the runtime system has been made more portable.

A shortcoming of our current load balancing model is that the cost of work movement is proportional to the size of the distributed data structures; the unit of work movement is an entire slice of the distributed data. Because of this problem, with load balancing, the performance of 2000×2000 (10 iterations) SOR was much worse than that of 1000×1000 (40 iterations) SOR, even though both problems require the same amount of computation. This problem must be addressed for our load balancing approach to be more generally applicable. Tiling of the iteration space may help reduce the problem, although the data structures may become much more complicated and difficult to manage.

Our system relies heavily on its ability to predict future performance (i.e., computation rates) on each of the slaves. This is an area where there is much room for improvement. Further investigation of control theory and signal processing techniques may provide more effective ways of selecting sampling frequencies (load balancing frequencies) and of filtering raw performance data so that load balancing is more effective in reducing the execution time of parallelized applications. A more general model for predicting performance with dynamic load balancing should be derived, possibly based on techniques from control theory.

Automatic selection of grain size is a very useful technique for distributed systems. For DOACROSS loops, the technique we described for selecting the optimal grain size (also discussed in [26] and [27]) should be expanded to handle tiled loops. If the compiler provides the runtime system with simple models of the computation and communication in a DOACROSS loop nest and sets up code for calibrating costs, the runtime system should be able to select an appropriate grain size for the loop. Also, techniques for controlling grain size for DOALL loops, e.g., using loop interchange, should be investigated.

Our work in this thesis addressed load balancing of a single loop nest distributed in a single dimension. Although many applications consist of one or more phases, each containing a single loop nest that can be load balanced independently (e.g., Gaussian elimination can be performed using LU decomposition, forward substitution, and back substitution), some applications may also have sections where multiple loop nests interact and share distributed data. The execution of loop nests could alternate so that load balancing of one nest could undo the load balancing of the other. If applications with this type of structure are common, this problem should be investigated and addressed.

For some loop nests, more parallelism may be available with multidimensional distributions so incorporating load balancing approaches for multidimensional distributions into a parallelizing compiler should also be investigated. A dimensional exchange approach [70, 72], in which balancing is done successively in each dimension, is an obvious choice for balancing of multidimensional distributions.

We presented several techniques for preventing the central load balancer from becoming a bottleneck for the application: we use efficient algorithms in the load balancer, select the load balancing frequency to keep the load balancing overhead low, and pipeline the interactions between the slaves and the load balancer.

As the number of processors becomes very large, the load balancer will not become a bottleneck because the system will reduce the frequency of load balancing to compensate for the higher load balancing costs. However, reducing the load balancing frequency makes the load balancer less responsive. To address this problem, the load balancer should be distributed, and the cost of performance collection should be reduced. Since we have made the "central load balancer" an abstract entity from the point of view of the slaves, it should be possible to distribute the load balancer without changing the slaves' view of the system.

Bibliography

[1] D. Adams. *Cray T3D system architecture overview. Revision 1.C.* Cray Research Inc., September, 1993.

[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] Maurice J. Bach. *The Design of the Unix Operating System.* Prentice Hall Software Series. Prentice-Hall, Inc., Englewood Cliffs, New Jersey 07632, 1986.

[5] Vasanth Balasundaram, Geoffrey Fox, Ken Kennedy, and Ulrich Kremer. A Static Performance Estimator to Guide Data Partitioning Decisions. In *Third ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 213–223, Williamsburg, VA, April, 1991. ACM Press.

[6] Richard Ernest Bellman. *Adaptive Control Processes: a Guided Tour*. Princeton University Press, Princeton, NJ, 1961.

[7] Shekhar Borkar, Robert Cohn, George Cox, Sha Gleason, Thomas Gross, H. T. Kung, Monica Lam, Brian Moore, Craig Peterson, John Pieper, Linda Rankin, P. S. Tseng, Jim Sutton, John Urbanski, and Jon Webb. iWarp: An integrated solution to high-speed parallel computing. In *Proceedings of Supercomputing '88*, pages 330–339, Orlando, FL, November 14–18, 1988. IEEE Computer Society and ACM SIGARCH.

[8] D. Callahan. Recognizing and Parallelizing Bounded Recurrences. In U. Banerjee, D. Gelernter, A. Nicolau, and D. Padua, editors, *Languages and Compilers for Parallel Computing. Fourth International Workshop.*, pages 169–185, Santa Clara, CA, August 7–9, 1991. Springer-Verlag.

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

[10] 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.

[11] B. Chapman, H. Zima, and P. Mehrotra. Handling Distributed Data in Vienna Fortran Procedures. In *Languages and Compilers for Parallel Computing. 5th International Workshop Proceedings.*, pages 248–263, New Haven, CT, August, 1992. Springer-Verlag.

[12] 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.

[13] Eric Cooper, Onat Menzilcioglu, Robert Sansom, and Francois Bitz. Host Interface Design for ATM LANs. In *Proceedings of the 16th Conference on Local Computer Networks*, pages 247–258, Minneapolis, MN, October 14–17, 1991. IEEE Computer Society Press.

[14] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*. The MIT Electrical Engineering and Computer Science Series. The MIT Press (McGraw-Hill Book Company), Cambridge, MA, 1990.

[15] Ron Cytron. Doacross: Beyond Vectorization for Multiprocessors (Extended Abstract). In Kai Hwang, Steven M. Jacobs, and Earl E. Swartzlander, editors, *Proceedings of the 1986 International Conference on Parallel Processing*, pages 836–844, University Park, PA, August 19–22, 1986. IEEE Computer Society Press.

[16] J. J. Dongarra, J. R. Bunch, C. B. Moler, and G. W. Stewart. *LINPACK Users' Guide*. Society for Industrial and Applied Mathematics, Philadelphia, 1979.

[17] Allan L. Fisher and Anwar M. Ghuloum. Parallelizing Complex Scans and Reductions. In *Proceedings of the ACM SIGPLAN '94 Conference on Programming Language Design and Implementation*, pages 135–146, Orlando, FL, June 20–24, 1994. ACM Press.

[18] Jon Flower and Adam Kolawa. Express is not just a message passing system: Current and future directions in Express. *Parallel Computing*, 20(4):597–614, April, 1994.

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

[20] Gene F. Franklin, J. David Powell, and Michael L. Workman. *Digital Control of Dynamic Systems*. Addison-Wesley Series in Electrical and Computer Engineering: Control Engineering. Addison-Wesley Publishing Company, Reading, MA, 1990.

[21] Christopher Giertsen and Johnny Petersen. Parallel Volume Rendering on a Network of Workstations. *IEEE Computer Graphics and Applications*, 13(6):16–23, November, 1993.

[22] R. W. Hamming. *Digital Filters (Second Edition)*. Prentice-Hall Signal Processing Series. Prentice-Hall, Inc., Englewood Cliffs, New Jersey 07632, 1983.

[23] 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.

[24] Seema Hiranandani, Ken Kennedy, and Chau-Wen Tseng. Compiler Support for Machine-Independent Parallel Programming in Fortran D. In J. Saltz and P. Mehrotra, editors, *Languages, Compilers, and Run-Time Environments for Distributed Memory Machines*, Amsterdam, The Netherlands, 1992. Elsevier Science Publishers B. V. (North-Holland).

[25] 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.

[26] 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.

[27] Seema Hiranandani, Ken Kennedy, and Chau-Wen Tseng. Evaluating Compiler Optimizations for Fortran D. *Journal of Parallel and Distributed Computing*, 21(1):27–45, April, 1994.

[28] 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.

[29] Ken Kennedy and Kathryn S. McKinley. Optimizing for Parallelism and Data Locality. In *Proceedings of 1992 International Conference on Supercomputing*, pages 323–334, Washington, DC, July 19–23, 1992. ACM Press.

[30] C.-C.J. Kuo and T. F. Chan. Two-color Fourier analysis of iterative algorithms for elliptic problems with red/black ordering. *SIAM Journal on Scientific and Statistical Computing*, 11(4):767–793, July, 1990.

[31] Richard E. Ladner and Michael J. Fischer. Parallel Prefix Computation. *Journal of the Association for Computing Machinery*, 27(4):831–838, October, 1980.

[32] 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.

[33] 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.

[34] 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.

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

[36] 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.

[37] 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.

[38] Evangelos P. Markatos and Thomas J. LeBlanc. Using Processor Affinity in Loop Scheduling on Shared-Memory Multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 5(4):379–400, April, 1994.

[39] S. Mohan and Pinaki Mazumder. Wolverines: Standard Cell Placement on a Network of Workstations. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 12(9):1312–26, September, 1993.

[40] 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.

[41] 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.

[42] David A. Padua, David J. Kuck, and Duncan H. Lawrie. High-Speed Multiprocessors and Compilation Techniques. *IEEE Trans. on Computers*, C-29(9):763–776, September, 1980.

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

[44] Douglas M. Pase, Tom MacDonald, and Andrew Meltzer. MPP Fortran Programming Model. Technical report, Cray Research, Inc., 655F Lone Oak Drive, Eagan, Minnesota 55121, May 19, 1994. Internal document. Available on World Wide Web as "ftp://ftp.cray.com/product-info/program_env/program_model.html".

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

[46] 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.

[47] William H. Press, Brian P. Flannery, Saul A. Teukolsky, and William T. Vetterling. *Numerical Recipes in C – The Art of Scientific Computing*. Cambridge University Press, Cambridge, 1988.

[48] Michael J. Quinn and Philip J. Hatcher. Data-Parallel Programming on Multicomputers. *IEEE Software*, 7(5):69–76, September, 1990.

[49] K. K. Ramakrishnan. Performance Considerations in Designing Network Interfaces. *IEEE Journal on Selected Areas in Communications*, 11(2):203–219, February, 1993.

[50] Xavier Redon and Paul Feautrier. Detection of Recurrences in Sequential Programs with Loops. In Arndt Bode, Mike Reeve, and Gottfried Wolf, editors, *PARLE '93 Parallel Architectures and Languages Europe. 5th International PARLE Conference Proceedings*, pages 132–145, Munich, Germany, June 14–17, 1993. Springer-Verlag.

[51] John R. Rose and Guy L. Steele Jr. C∗: An Extended C Language for Data Parallel Programming. In *Proceedings for the Second International Conference on Supercomputing, Volume 2*, pages 2–16, May, 1987.

[52] M. Rosing, R. B. Schnabel, and R. P. Weaver. The Dino parallel programming language. *Journal of Parallel and Distributed Computing*, 13(1):30–42, September, 1991.

[53] Robert Schreiber and Jack J. Dongarra. Automatic Blocking of Nested Loops. Technical Report CS-90-108, Computer Science Department, University of Tennessee, 107 Ayres Hall, Knoxville, TN 37996-1301, May, 1990.

[54] 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.

[55] Steve Sistare and Mark Friedell. A Distributed System for Near-Real-time Display of Shaded Three-Dimensional Graphics. In *Proceedings. Graphics Interface '89*, pages 283–90, London, Ontario, Canada, June 19–23, 1989. Canadian Man-Computer Communication Society, Morgan Kaufmann Publishing, Palo Alto, CA.

[56] Otto Joseph Mitchell Smith. *Feedback Control Systems*. McGraw-Hill Series in Control Systems Engineering. McGraw-Hill, New York, NY, 1958.

[57] Peter Steenkiste. *Nectarine - A Nectar Interface*. School of Computer Science, Carnegie Mellon University, Pittsburgh, PA 15213, May 14, 1991. Internal Document.

[58] Peter A. Steenkiste. A Systematic Approach to Host Interface Design for High-Speed Networks. *Computer*, 27(3):47–57, March, 1994.

[59] Peter A. Steenkiste, Brian D. Zill, H. T. Kung, Steven J. Schlick, Jim Hughes, Bob Kowalski, and John Mullaney. A Host Interface Architecture for High-Speed Networks. *IFIP Transactions C (Communication Systems)*, C-14:31–46, 1993. 4th IFIP Conference on High Performance Networking, Liege, Belgium, December, 1992.

[60] James M. Stichnoth. Efficient Compilation of Array Statements for Private Memory Multicomputers. Technical Report CMU-CS-93-109, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA 15213, February, 1993.

[61] 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.

[62] Jaspal Subhlok, James M. Stichnoth, David R. O'Hallaron, and Thomas Gross. Exploiting Task and Data Parallelism on a Multicomputer. In *Fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 13–22, San Diego, CA, May, 1993.

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

[64] Peiyi Tang and Pen-Chung Yew. Processor Self-Scheduling for Multiple-Nested Parallel Loops. In Kai Hwang, Steven M. Jacobs, and Earl E. Swartzlander, editors, *Proceedings of the 1986 International Conference on Parallel Processing*, pages 528–535, University Park, Pennsylvania, August, 1986. IEEE Computer Society Press.

[65] 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.

[66] Thinking Machines Corporation, Cambridge, MA. *Connection Machine CM-5 Technical Summary*, November, 1992.

[67] Ping-Sheng Tseng. A Parallelizing Compiler for Distributed Memory Parallel Computers. Ph.D. Thesis CMU-CS-89-148, ECE Department, Carnegie Mellon University, May, 1989.

[68] Ping-Sheng Tseng. Compiling Programs for a Linear Systolic Array. In *Proceedings of the ACM SIGPLAN '90 Conference on Programming Language Design and Implementation*, pages 311–321, White Plains, NY, June, 1990. ACM Press.

[69] 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.

[70] Reinhard v. Hanxleden and L. Ridgway Scott. Load Balancing on Message Passing Architectures. *Journal of Parallel and Distributed Computing*, 13(3):312–324, November, 1991.

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

[72] 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.

[73] Michael E. Wolf and Monica S. Lam. A Data Locality Optimizing Algorithm. In *Proceedings of the ACM SIGPLAN '91 Conference on Programming Language Design and Implementation*, pages 30–44, Toronto, Ontario, Canada, June 26–28, 1991. ACM Press.

[74] 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.

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

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

[77] 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.

[78] 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.

[79] H. Zima, P. Brezany, B. Chapman, P. Mehrota, and A. Schwald. Vienna Fortran – A Language Specification Version 1.1. Technical Report ACPC/TR 92-4, Austrian Center for Parallel Computation, March, 1992.