Controlling Application Grain Size on a Network of Workstations

Bruce S. Siegell
Department of Electrical and Computer Engineering
Carnegie Mellon University

Peter A. Steenkiste
School of Computer Science
Carnegie Mellon University

Keywords

Automatic parallelization, grain size, network of workstations, dynamic load balancing.

Abstract

An important challenge in the area of distributed computing is to automate the selection of the parameters that control the distributed computation. A performance-critical parameter is the grain size of the computation, i.e., the interval between successive synchronization points in the application. This parameter is hard to select since it depends both on compile time (loop structure and data dependences, computational complexity) and run time components (speed of compute nodes and network). On networks of workstations that are shared with other users, the run-time parameters can change over time. As a result, it is also necessary to consider the interactions with dynamic load balancing, which is needed to achieve good performance in this environment.

In this paper we present a method for automatically selecting the grain size of the computation consisting of nested DO loops. The method is based on close cooperation between the compiler and the runtime system. We evaluate the method using both simulation and measurements for an implementation on the Nectar multicomputer.

1 Introduction

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. Parallel languages and parallelizing compilers have simplified the programming of shared and distributed memory multiprocessors. However, modifications to these tools are needed if they are to be targeted for networks of workstations because of the independent nature of the machines and the higher, more variable costs of communication over a network. Our research investigates these issues in the context of parallelizing compilers.

Generally, parallelizing compilers assume a *specific*, *homogeneous* target system that is *dedicated* to the application, and allocate the same amount of work to each processor. On networks of workstations, processors may be heterogeneous and may have performance that varies at run time due to competing users, and different types of networks with different loads may be encountered. As a result, static, equal distribution of work does not result in good utilization of the available resources. To address this problem, we have designed a system that supports dynamic load balancing of parallelized code by periodically adjusting the amount of work allocated to each processor at run time [11, 12]. At each load balancing point, the system attempts to allocate work units in numbers proportional to the relative processing capabilities of the processors.

The *grain size* of an application—the amount of computation between successive synchronization points—is a very important performance parameter for parallelized applications because it affects both communication costs and load balancing effectiveness. Because communication costs and load balancing depend on the configuration and dynamic aspects of the system, an appropriate grain size can not be selected using just compile-time information. Cooperation between the compiler and runtime system is necessary for selection and control of the grain size of an application distributed over a network. This paper investigates the impact of grain size on parallelism, communication costs, and load balancing on a network of workstations, and presents methods for selecting and controlling grain size for nested DO loops.

In the next section, we outline our parallel compilation model. In Section 3, we describe parameters that must be considered to select an appropriate grain size and the loop transformations useful in controlling grain size. Sections 4 and 5 describe how an appropriate grain size is determined and, when possible, controlled for different types of synchronization. Analysis and simulation is used to predict performance with different grain sizes, using simplified models to estimate computation and communication costs. We summarize in Section 6.

2 Parallelization model

In our compiler model, computations are parallelized by distributing loop iterations and slices of aggregate data structures using the owner computes rule: data slices modified by an iteration are assigned to the same processor as the iteration. The compiler modifies the loop bounds of the sequential program so that each processor only executes the iterations assigned to it. The aggregate data structures may be distributed along one or more dimensions, although, in this paper, we only discuss applications distributed in a single dimension. Load balancing is performed by adjusting the number of iterations and data slices allocated to each processor.

Our goal in parallelization is to minimize the execution time, $t_{elapsed}$, of the parallelized application. The minimum execution time is attained when the application uses all computing resources available to it productively. On a dedicated, homogeneous set of processors, we use the traditional efficiency measure [8, 9] to evaluate productive use of resources:

$$efficiency = \frac{t_{productive}}{t_{available}} = \frac{speedup}{P} = \frac{t_{sequential}}{P \times t_{elapsed}} \tag{1}$$

where P is the number of processors, and the sequential time, $t_{sequential}$, is measured on a dedicated processor. In this paper, all experiments are performed assuming a homogeneous set of processors, but, in some cases, competing processes use some of the processing resources. For experiments with competing loads, we modify the efficiency formulation to account for resources used by the competing processes:

$$efficiency = \frac{t_{productive}}{t_{available}} = \frac{t_{sequential}}{P \times t_{elapsed} - \sum_{i=0}^{P-1} compete_i} \tag{2}$$

where $compete_i$ is the amount of resources (i.e., CPU time) used by competing processes on processor i.

3 Grain size

This section discusses factors that influence the grain size and how the grain size can be controlled. The grain size is determined by the most frequently executed synchronizations in the application. The nature of these synchronizations determines how easily grain size can be controlled.

3.1 Compile time control of grain size

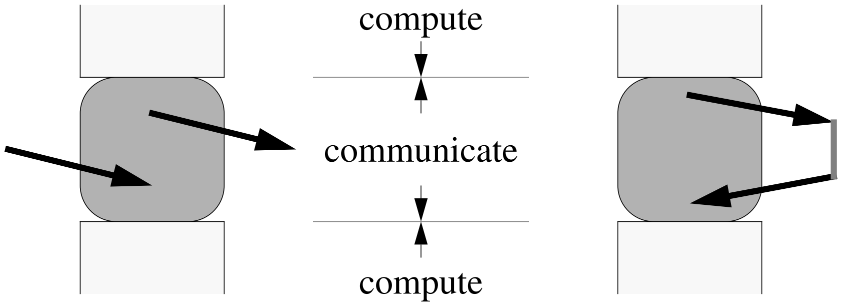
The synchronizations in a parallelized application result from its communication requirements, which are in turn determined by the distribution of the loop iterations and the dependences and loop structure of the original sequential code. In some cases, the amount of computation between synchronizations, i.e., the grain size, can be changed and controlled by modifying the loop structure of the parallelized code. Of greatest interest are loop restructuring transformations such as *strip mining* [7] that allow the grain size to be parameterized with a continuum of grain size choices. Other transformations, such as loop interchange [1, 10, 17, 19] and loop skewing [16, 19], also can be used to modify the grain size, but their application is more difficult to parameterize. Basic techniques for manipulating communication code, such as *loop splitting* [7] and *message aggregation* [3, 5, 6, 13, 14] are also necessary so that the above techniques can be effective. For a given application, the communication patterns in the parallelized code determine which of the above transformations are useful for controlling the grain size. Because of the dynamic nature of computation and communication costs on a network of workstations, the actual selection of the parameters for the transformations requires run-time information.

3.2 Applications requiring no communication

The simplest loop nests to parallelize are those that require no communication, such as matrix multiplication. For these applications, there is no interaction between the processors so grain size is not an issue on dedicated machines. However, when load balancing is added to these applications, the communication needed for load balancing determines the grain size; i.e., the load balancing frequency determines the frequency of this communication. Selection of the load balancing frequency is based on several factors, including the overhead of interactions between the application processes and the load balancer, and the overheads of work movement. The latter typically limits the load balancing frequency, and, in practice, communication is so infrequent that grain size is not a performance concern. Load balancing frequency selection is described in detail by Siegell [11].

3.3 Applications with communication

For applications with communication, 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.



a) Unidirectional synchronization

b) Bidirectional synchronization

Figure 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 1a), i.e., pipelined applications, data is communicated 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 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. With dynamic load balancing, even a single assignment statement involving distributed data elements can result in a barrier synchronization because global communication may be needed to identify the processors owning the source and destination elements. In our analysis, we treat all bidirectional synchronizations as barrier synchronizations, even if they only involve a subset of the processors.

4 Unidirectional synchronizations

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

4.1 Controlling grain size at run time

For applications with DOACROSS loops, the grain size can be parameterized using techniques such as *strip mining* [7]. Strip mining replaces a single loop with two nested loops. Communication is moved out of the inner loop and *message aggregation* [3, 5, 6, 13, 14] is used to combine messages with common destinations. The number of iterations of the resulting inner loop is the *block size* of the computation. Figure 2 demonstrates the use of these techniques for a successive overrelaxation (SOR) example. To control the grain size at run time, the compiler strip mines the loop, but the block size is a variable, e.g., *blocksize* in Figure 2c, that is set at run time. The grain size, t_{grain}, and the block size are related as follows:

$$blocksize = \frac{t_{grain}}{t_{iteration}} \qquad (3)$$

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

Loop tiling [14, 15, 18] is a more complicated approach that combines strip mining and loop interchange, usually to localize memory references by the resulting inner loops. Loop tiling also can be used to control grain size and communication overhead in a manner similar to strip mining [14]. However, compared with strip mining, tiling significantly complicates data management in a system with load balancing, and it will not be addressed in this paper.

4.2 Grain size selection

Figure 3 outlines the model of a pipelined computation. The distributed loop has n iterations distributed across P processors, and

```
for (iter = 0; iter < maxiter; iter++) {
    for (i = 1; i < n - 1; i++) {
        for (j = 1; j < n - 1; j++) {    /* distributed */
            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 code

```
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 2: Parallelization options for SOR (simplified version; error computation not shown). Code portions affected by strip mining and message aggregation are shaded.

the strip mined loop has a total of m iterations, divided into M blocks. The size of each block is the block size, b:

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

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

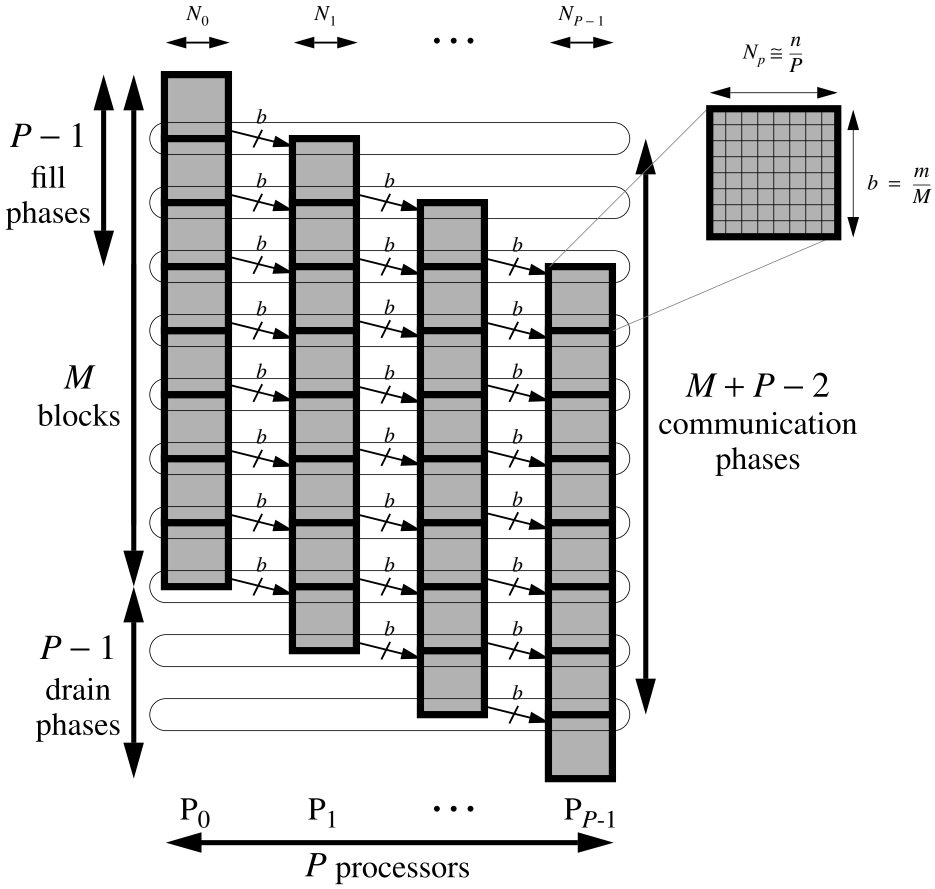
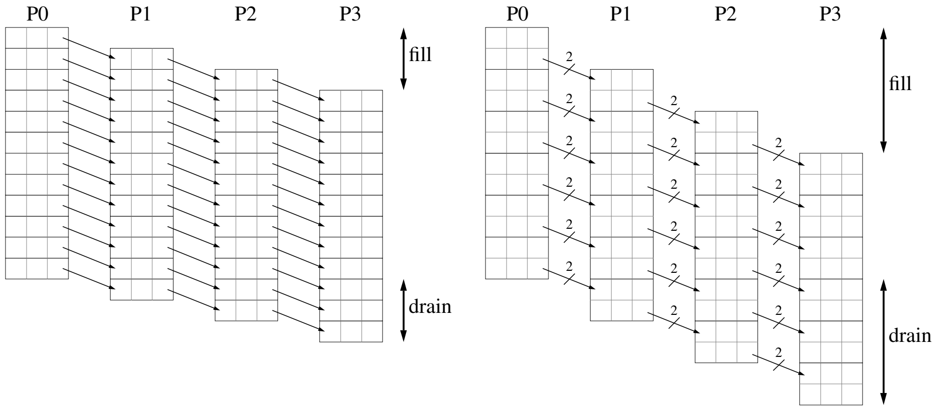


Figure 3: Pipelined execution of a distributed loop showing parameters for modeling execution time. Distributed loop has n iterations and pipelined loop (enclosing distributed loop) has m iterations.

For pipelined applications, a tradeoff between parallelism and communication costs must be considered to select an appropriate block size for the strip mined loop. Communication costs are minimized when the block size is made as large as possible. However, due to time required to fill and drain the pipeline, increasing the block size reduces the parallelism in the application; e.g., compare Figures 4b versus 4a for the SOR example in Figure 2. These two conflicting effects can be modeled to estimate the total computation time for the application. Using this model, we select a block size for the strip mined loop that maximizes performance.



a) Pipelined b) Blocked and pipelined

Figure 4: Pipelined execution of a single relaxation phase for the SOR example in Figure 2.

4.2.1 Pipeline fill and drain times

From Figure 3, it can be observed that the elapsed time for the application, ignoring communication costs, is $M + P - 1$ times the time to execute one block, t_{block}:

$$t_{elapsed} = (M + P - 1) \times t_{block} \tag{5}$$

Given that:

$$t_{sequential} = P \times M \times t_{block} \tag{6}$$

an upper bound on efficiency for a homogeneous environment, ignoring communication costs, is:

$$\begin{aligned} \mathit{efficiency} &= \frac{P \times M \times t_{block}}{P \times (M + P - 1) \times t_{block}} \\ &= \frac{M}{M + P - 1} \end{aligned} \tag{7}$$

4.2.2 Communication costs

Communication of boundary values occurs between the pipeline phases. Since there are $M + P - 1$ computation phases, there are $M + P - 2$ communication phases. Thus, the total communication cost is

$$t_{comm} = (M + P - 2) \times t_{shift} \tag{8}$$

where t_{shift} is the cost of a single communication phase. Each communication phase can be modeled as follows:

$$t_{shift} = t_{fixed} + t_{incr} \times elements \tag{9}$$

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 is equal to the block size, b. t_{fixed} and t_{incr} can be estimated by measuring communication costs—by measuring times for passing messages between the processors—when the application is started [11]. These measurements can be repeated periodically (e.g., between executions of the pipelined loop) to take into account changing conditions in the network.

4.2.3 Selecting the optimal block size

A model of the total execution time can be created by combining Equations 5 and 8:

$$t_{total} = (M + P - 1) \times t_{block} + (M + P - 2) \times t_{shift} \tag{10}$$

t_{shift} can be replaced using Equation 9, and t_{block} can be replaced using Equation 6. $t_{sequential}$ can be estimated by extrapolating from measurements at startup time of several iterations of a copy of the loop body. This results in a representation of t_{total} with only one unknown, M, the number of blocks:

$$\begin{aligned} t_{total} &= (M + P - 1) \times \frac{t_{sequential}}{P \times M} \\ &\quad + (M + P - 2) \times (t_{fixed} + t_{incr} \times \frac{m}{M}) \end{aligned} \tag{11}$$

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:

$$\frac{dt_{total}}{dM} = 0$$

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

$$M = \sqrt{\frac{t_{sequential} \times (1 - \frac{1}{P}) + t_{incr} \times m \times (P - 2)}{t_{fixed}}} \tag{12}$$

Once M is known, the block size for the optimal grain size can be calculated using Equation 4.

The optimal M is computed at application startup time using the known values of P and m and estimates of $t_{sequential}$, t_{fixed}, and t_{incr} determined by measuring the execution time of copies of small portions of the computation. The code that collects these measurements can be generated by the compiler, i.e., without intervention by the programmer. The initial measurements typically take several hundred milliseconds. Throughout the execution, the estimates can be updated using new measured times, thus allowing the application to adjust to changes on the processors and in the network. Updating the estimates takes less time than making the initial estimates because measurements from the actual computation can be used.

4.2.4 Related work

The Fortran D compiler performs a similar analysis to select the appropriate block size for pipelined computations [6]. As in our approach, the optimal block size is determined by setting the derivative of a model of the execution time for the application equal to zero. However, unlike our approach, their estimates of computation and communication times for the program are determined by a *static performance estimator* which runs a training set of kernel routines to characterize costs in the system [4]. The static performance estimator matches computations in the given application with kernels from the training set. Their approach requires a separate characterization for each machine configuration that might be used when running the application. Our approach is more flexible: since we rely on run-time measurements, we automatically adjust to the specific application (nature of computation and input size) and machine configuration (communication costs, number of processors, and processor speeds). However, by delaying our characterization of costs until run time, we add the characterization time to the cost of executing the application.

4.3 Evaluation

We implemented the SOR example (Figure 2c) on the Nectar system [2], a high-speed fiber optic network developed at Carnegie Mellon University connecting several Sun workstations. We use the implementation to evaluate both the accuracy of the model and the effect of having the grain size adjust to the system and application characteristics.

4.3.1 Model evaluation

Figure 5 compares the efficiency of parallelization for the SOR example (1000x1000 matrix, 10 iterations) predicted by our model with the measured efficiencies on Nectar for 4 slave processors and using different block sizes. In each graph, the peak of the efficiency predicted by the model—the optimal block size—is marked by a vertical

Version	$t_{sequential}$	t_{fixed}	t_{incr}
a) No added delays	8.479 sec	1.716 msec	4.8μsec
b) 5 msec added delay	8.497 sec	16.63 msec	4.8μsec
c) 50 msec added delay	8.500 sec	152.4 msec	4.8μsec

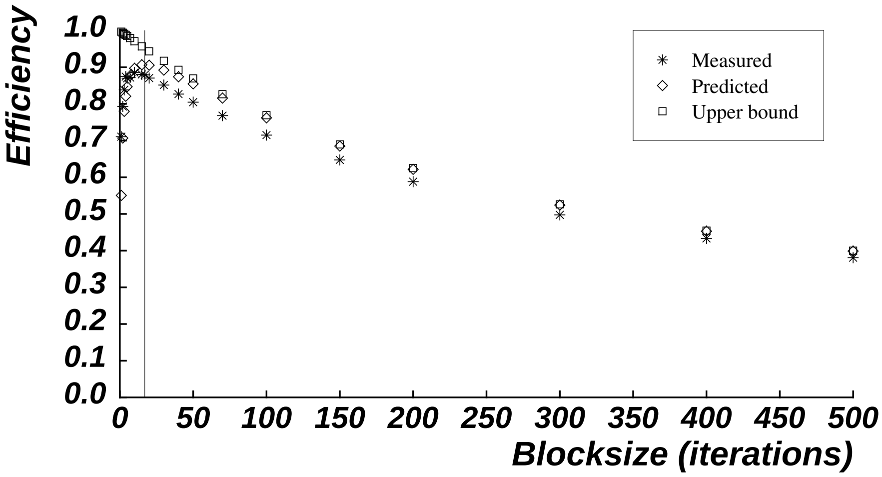
Table 1: Measured values used in calculating optimal grain size for Figure 5.

line. Table 1 shows the measured times used to compute the optimal block size in Figure 5. The times used in computing the efficiencies do not include the time spent characterizing the system or computing the optimal block size. We see in Figure 5 that there is a close match between the model and the measured values, and that the two curves reach their maximum at about the same block size; i.e., our model fairly accurately predicts the optimal block size. To show how our execution model responds to different communication costs, we did the same experiment, but with artificial delays added to the communication functions. The results are shown in Figures 5b and 5c. Again we observe a good fit between the model and measured values; i.e., our model adjusts well to different networks and changing network conditions.

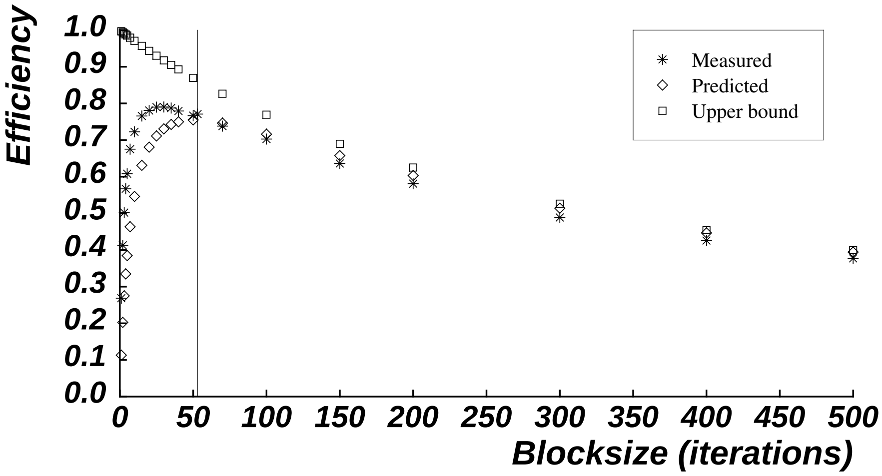
Since the values of $t_{sequential}$, t_{fixed}, and t_{incr} are only estimates, an important question is how sensitive the results are to errors in these estimates. The curves in Figure 5 are relatively flat around the maximum, indicating that changing the optimal block size by as much as a factor of two in either direction does not reduce the efficiency much. In addition, given the shape of the curves, it is better to select block sizes that are too high than too low.

Estimates of $t_{sequential}$ based on measurements of several iterations of the pipelined loop were consistent over several measurements (e.g., the measurements presented in Table 1) and were quite close to measurements of the actual execution time on one dedicated processor; i.e., estimates for $t_{sequential}$ were very accurate. In our implementation, we estimated $t_{sequential}$ for a *dedicated* system (by taking the smallest of several measurements of execution times for iterations of the loop). If during actual execution there are competing loads on one or more processors, the actual grain size in the distributed application will be larger than expected based on this estimate. In general, we expect these grain size variations to be small enough (e.g., within a factor of two) that the efficiency remains near the maximum value. In addition, to take into account changing loads, if the strip mined loop is executed multiple times, the estimate of $t_{sequential}$ can be updated between executions of the loop based on measurements that include the effects of competing loads.

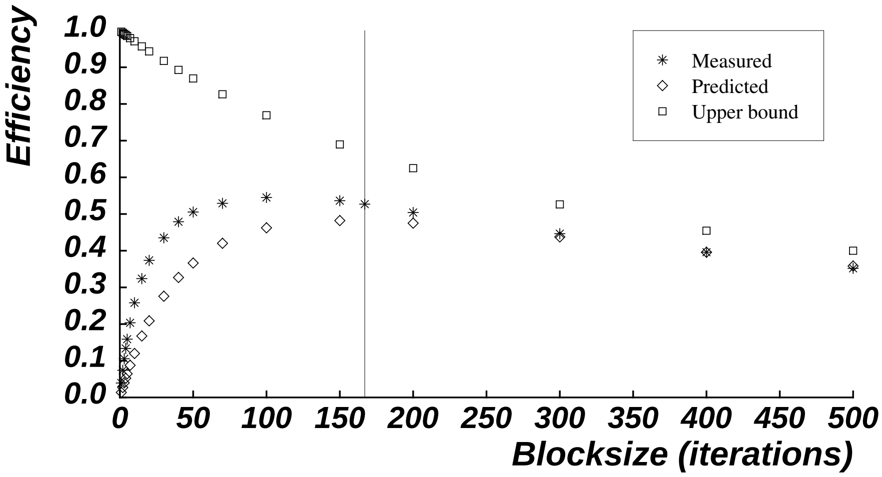
The communication costs are more variable and more difficult to measure. To predict the efficiency and to compute the optimal block size, we use a conservative estimate of the communication costs: the



a) No added delays.



b) 5 millisecond delay added to each message.



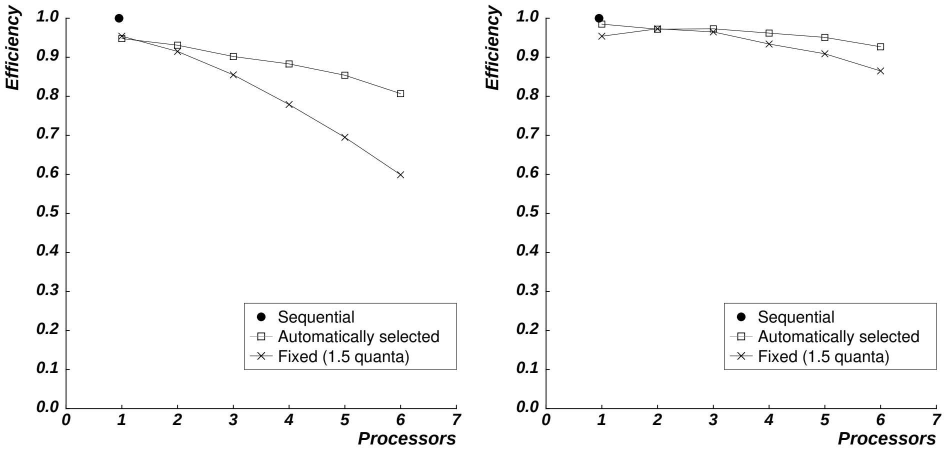
c) 50 millisecond delay added to each message.

Figure 5: Efficiency of the pipelined loop in the SOR example (1000x1000 matrix, 10 iterations) as a function of the block size.

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. 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.

4.3.2 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 of 150 milliseconds with the performance of a version with the automatically selected "optimal" grain size, selected using the method described in the previous section. Figure 6 shows the efficiency measurements taken on the Nectar system with homogeneous, dedicated processors for two different problem sizes. 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.



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

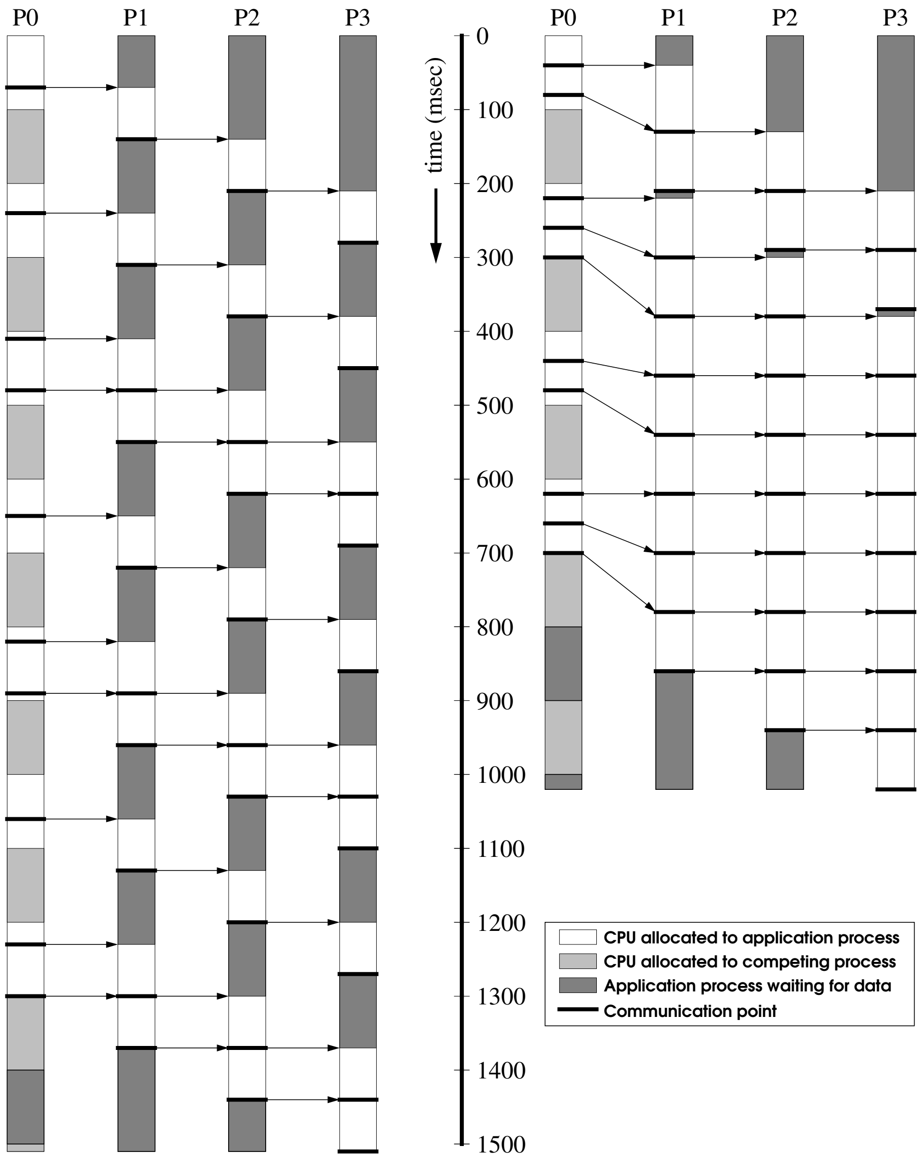
Figure 6: Parallel versions of SOR without load balancing in a dedicated homogeneous environment. Fixed grain size vs. automatically selected grain size.

4.4 Effect of competing loads

When a competing load is added to one of the processors executing a pipelined application, 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 7a). Dynamic load balancing can keep the load balanced by periodically redistributing work in proportion to the processing capabilities of each processor. Siegell [11], for example, uses measured computation rates specific to the application—loop iterations computed per unit time—to characterize the available capabilities of each processor. When 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 7b). The communication required by the application aligns the processors so that efficiency is not affected adversely by competing loads and pipelined execution can continue without stalling. This is true for any grain size, as long as there is enough buffer space to store the intermediate data.

To confirm that grain size has little effect on the efficiency of a pipelined application in a load balanced environment with competing loads, we simulated the interactions between the scheduling of processes by the operating system and the communication between the slave processors. 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 (100 milliseconds). The simulations do not consider communication costs, but do model time spent filling and draining the pipeline.

Figure 8 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 predicted upper bound (the solid line, computed using Equation 7), regardless of the grain size. On systems with competing loads, the 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 8d) 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 can 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.



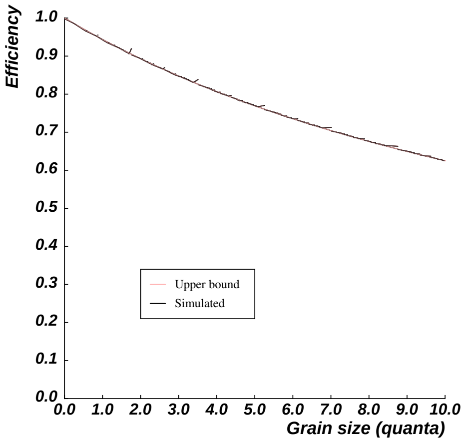
a) Equal distribution

b) Load balanced

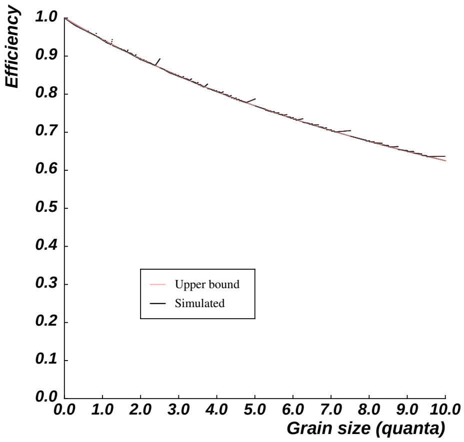
Figure 7: Pipelined execution with competing load on first processor. Grain size for equal distribution is 70 milliseconds. Round robin scheduling with 100 millisecond time quantum, ignoring communication costs.

5 Bidirectional (barrier) synchronizations

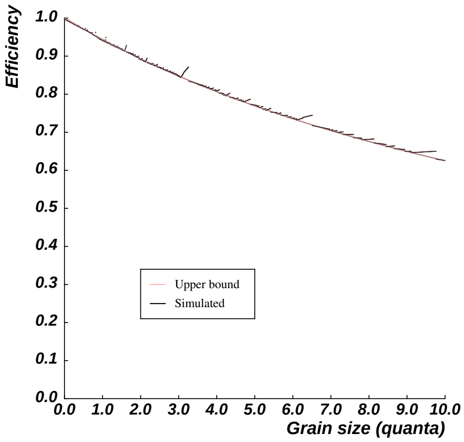
Barrier synchronizations may be caused by reduction operations, by distributed loops that just shift data between processors, or by assignment statements that involve data on multiple processors. With bidirectional synchronizations, grain size can be changed using transformations such as loop splitting [7], loop interchange [1, 5, 10, 17, 19], or loop skewing [16, 19], but these transformations are difficult to parameterize. A continuum of grain sizes as we had in the case of



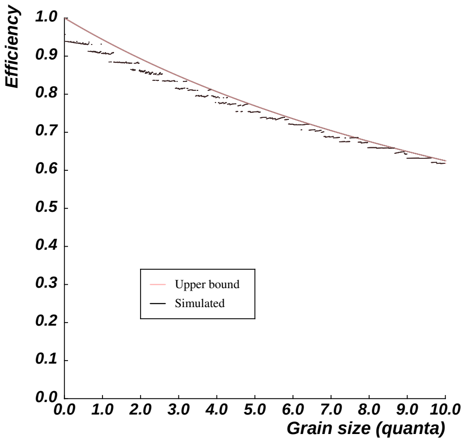
a) Competing load on P0



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 8: 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.

unidirectional synchronizations is not possible. It is possible to generate several versions of the code and to select the most appropriate on at run time [17], but this provides only limited run-time control of the grain size. Because of this limitation, we do not investigate options for modifying the grain size of problems with bidirectional synchronizations, but we analyze the overhead of bidirectional synchronizations and examine the effects of bidirectional synchronizations on program performance in the presence of competing loads.

5.1 Synchronization overhead

Figure 9 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 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

$$\begin{aligned} \mathit{efficiency} &= \frac{t_{sequential}}{P \times t_{total}} \\ &= \frac{n \times t_{iteration} \times m}{P \times (t_{barrier} \times m + \frac{n}{P} \times t_{iteration} \times m)} \\ &= \frac{n \times t_{iteration}}{P \times t_{barrier} + n \times t_{iteration}} \end{aligned} \tag{13}$$

The efficiency formulation shows that all parallelization overhead is due to barriers (i.e., the $P \times t_{barrier}$ term in the denominator) and that the efficiency is determined by the cost of the barrier relative to the amount of computation between barriers. Efficiency can only be improved by reducing the cost of each barrier, or by having the compiler or programmer restructure the program so that it can be modeled differently.

5.2 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 of the times on all the processors, and the barriers will cause the

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

Figure 9: Parallelized version of DOALL loop followed by global operation.

skews between execution times to accumulate and increase 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. 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.

5.2.1 Model

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 4.4. 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. This model applies in systems where communication is buffered.

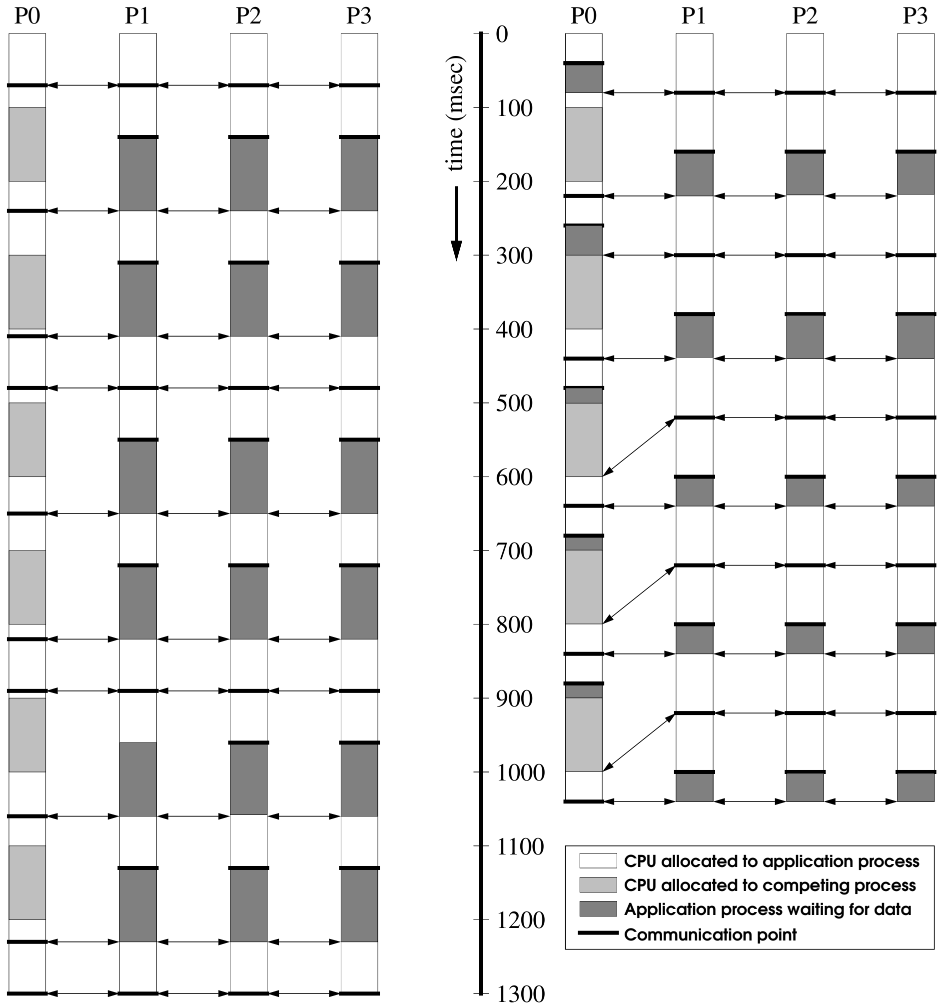
Figures 10 and 11 show time lines for a four processor system with a single competing load on one processor, with different work assignments and grain sizes. The time lines identify the different CPU states (*working*, *waiting*, or *inactive* with respect to the load balanced application) and show the interactions for the barrier synchronizations. In the figures, the thick horizontal lines indicate the times when the application processes enter a barrier synchronization, and the arrows indicate the times when the processes exit the barrier. In Figure 10, the grain size for the case without load balancing (Figure 10a) is 0.7 time quanta. After load balancing (Figure 10b), 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 11 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 10, 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. The scheduling of competing processes causes variation between execution times of consecutive computation phases, causing temporary skews between the processors, resulting in idle time. This variation increases with the load on the processors, but the significance of the variations decreases as the grain size is increased. If grain size can be controlled, a grain size as large as possible—at least one time quantum on the loaded processors after balancing—should be selected to minimize the effects of the variations [11].

5.2.2 Simulation

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 10 and 11, but run for 1000 synchronizations. Figure 12 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 and scheduling are in phase, e.g., grain sizes that are multiples of 1.75 quanta for Figure 12a, 2.5 for 12b, 3.25 for 12c, and 19.25 for 12d [11]. The simulation results 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 and normal system activity may cause variations in the schedule, so it is not practical to control the grain size very accurately, and 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, both to achieve a higher efficiency, and to be less sensitive to changes in competing loads.



a) Equal distribution ($\mathit{efficiency} = 0.609$)

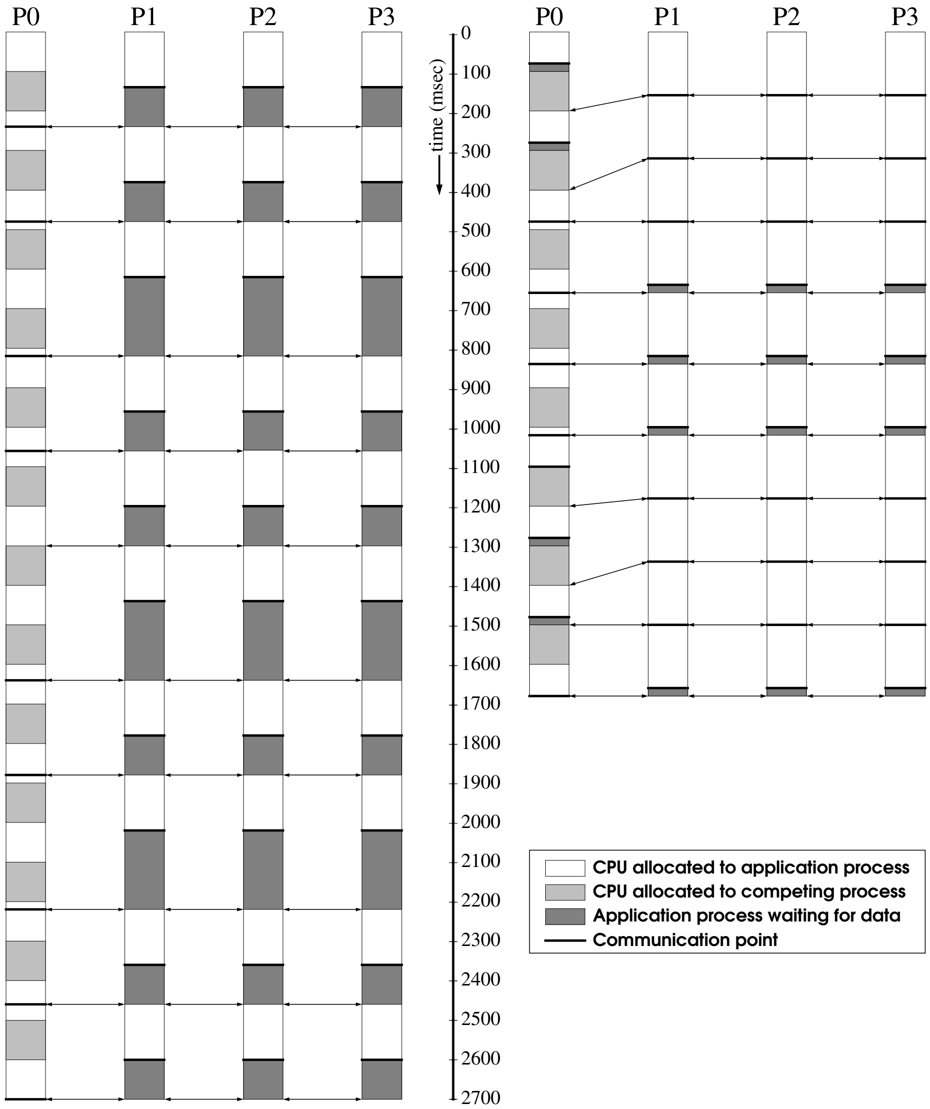
b) Load balanced ($\mathit{efficiency} = 0.765$)

Figure 10: Application with bidirectional synchronizations executing with competing load on first processor. Grain size for equal distribution case is 70 milliseconds. Round robin scheduling with 100 millisecond time quantum, ignoring communication costs.

6 Conclusion

The selection of the optimal grain size for applications distributed over networks of workstations requires both compile-time and run-time information. We presented and evaluated a method for automatically selecting the grain size. The method is based on close cooperation between the compiler and the runtime system.

For applications with unidirectional synchronizations (i.e., with DOACROSS loops), we showed how the grain size can be con-

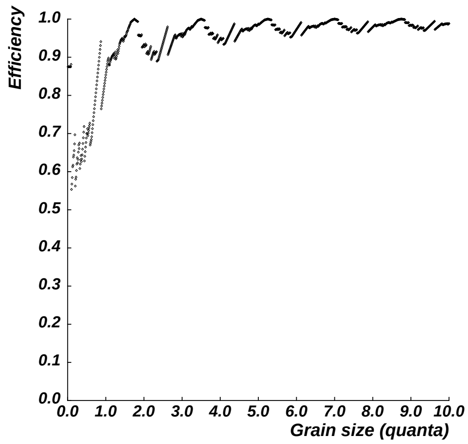


a) Equal distribution
(*efficiency* = 0.589)

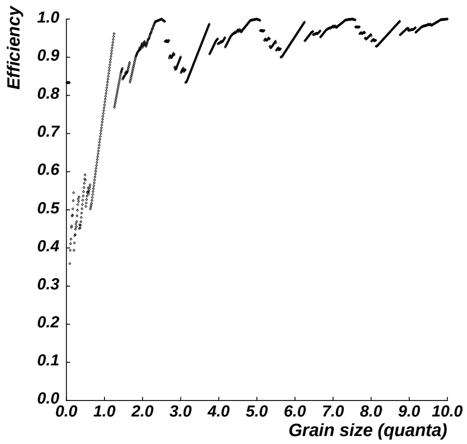
b) Load balanced
(*efficiency* = 0.946)

Figure 11: Application with bidirectional synchronizations executing with competing load on first processor. Grain size for equal distribution case is 140 milliseconds. Round-robin scheduling with 100 millisecond time quantum, ignoring communication costs.

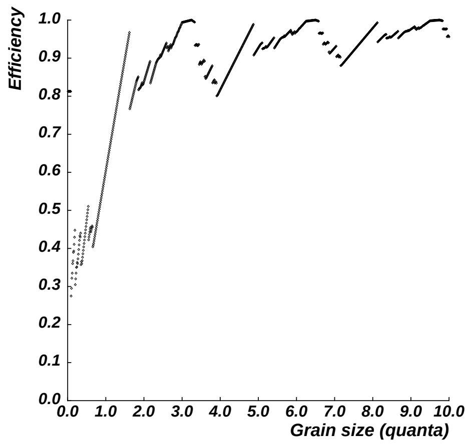
trolled, and showed how to select the optimal grain size considering both communication overhead and parallelism. Simulations and measurements verified the effectiveness of our method for selecting the optimal grain size. Simulations showed that for applications with unidirectional synchronizations, interactions between the grain size and scheduling by the operating system do not significantly affect performance; therefore the interactions need not be considered in



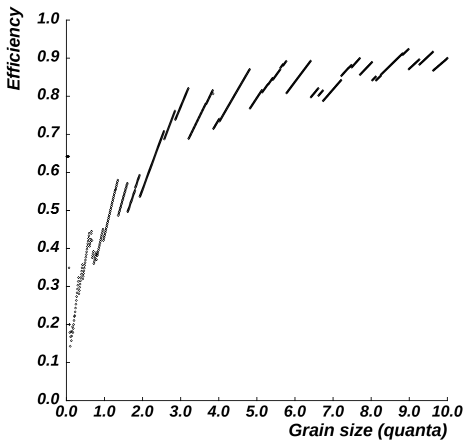
a) Competing load on P0



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 12: Parallelization efficiency for varying grain sizes on a 4 processor system. Simulation results for round-robin scheduling with 100 millisecond time quantum, ignoring communication costs.

selecting grain size.

For applications with bidirectional synchronizations, grain size is more difficult to control. 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.

References

[1] 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.

[2] 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.

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

[4] 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.

[5] 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.

[6] 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.

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

[8] 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.

[9] 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.

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

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

[12] 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.

[13] 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.

[14] 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.

[15] 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.

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

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

[18] 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.

[19] 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.

Bruce S. Siegell

Bell Communications Research
NVC 3X301
331 Newman Springs Road
Red Bank, New Jersey 07701
(908) 758-3211
e-mail: bss@bellcore.com

Bruce Siegell is a Research Scientist at Bell Communications Research. He received his Ph.D. degree from Carnegie Mellon University in May, 1995. He received his M.S. degree from CMU in 1986 and received the Sc.B. degree from Brown University in 1984. His research interests include parallel and distributed computing, network applications, and compilers and programming tools. He is a member of Tau Beta Pi, Sigma Xi, the IEEE and the ACM.

Peter A. Steenkiste

School of Computer Science
Carnegie Mellon University
5000 Forbes Avenue
Pittsburgh, PA 15213
(412) 268-3261
e-mail: prs@cs.cmu.edu
WWW: http://www.cs.cmu.edu/~prs/

Peter Steenkiste is a Senior Research Computer Scientist in the School of Computer Science at Carnegie Mellon University. Before coming to CMU in 1987, he received his M.S. and Ph.D. degrees in Electrical Engineering from Stanford University. He received the degree of Electrical Engineer from the State University of Ghent in Belgium in 1982. He is a member of the IEEE computer society and the ACM.

Peter Steenkiste's research interests are in the areas of high speed networking and distributed computing. While at CMU, he has been involved in the Nectar, Gigabit Nectar and Credit Net projects. Nectar was one of the first workstation clusters used for high-performance computing. It was built around a switch-based fiber-optic local area network. Gigabit Nectar uses a HIPPI network and optimized host-network interfaces for communication and it supports both applications distributed across workstations and across heterogeneous supercomputers. It forms the basis of the Nectar gigabit testbed. The Credit Net project is developing a 622 Mbit/second local area ATM network. The research focuses on ATM flow control and support for both distributed computing applications and applications that mix network traffic types (data, voice, video). More information on the these projects can be found on the author's home page.