Accurately Selecting Block Size at Run Time in Pipelined Parallel Programs*[†]

David K. Lowenthal
Department of Computer Science
The University of Georgia
Athens, GA 30602–7404
dkl@cs.uga.edu

January 11, 2000

Abstract

Loops that contain cross-processor data dependencies, known as DOACROSS loops, are often found in scientific programs. Efficiently parallelizing such loops is important yet nontrivial. One useful parallelization technique for DOACROSS loops is *pipelining*, where each processor (node) performs its computation in blocks; after each, it sends data to the next node in the pipeline. The amount of computation before sending a message is called the *block size*; its choice, although difficult to make statically, is important for efficient execution. This paper describes a flexible run-time approach to choosing the block size. Rather than rely on static estimation of workload, our system takes measurements during the first two iterations of a program and then uses the results to build an execution model and choose an appropriate block size which, unlike a static choice, may be nonuniform. To increase accuracy of the chosen block size, our execution model takes intra- *and* inter-node performance into account. It is important to note that our system finds an effective block size automatically, *without experimentation* that is necessary when using a statically chosen block size. Performance on a network of workstations shows that programs that use our run-time analysis outperform those that use static block sizes by as much as 18% when the workload is unbalanced. When the workload is balanced, competitive performance is achieved as long as the initial overhead is sufficiently amortized.

Keywords: pipelining, block size, compiler, run-time system

1 Introduction

Distributed-memory parallel computing is widely used to speed up scientific applications. One key obstacle to parallelization is the presence of data dependencies, which enforce a certain ordering; their existence in loops may result in sequential execution of the entire loop. A significant number of scientific applications contain loops whose data dependencies prevent all iterations from being executed in parallel, including alternate direction implicit (ADI) integration, implicit hydrodynamics codes [9], three-dimensional Multigrid methods [13], and air pollution simulations [14]. Loops with data dependencies are often referred to as DOACROSS loops; if the dependencies can be precisely determined by inspecting the program, we call them *regular* DOACROSS loops [15]. These can often

[†] An earlier, abbreviated version of this paper entitled "Run-Time Selection of Block Size in Pipelined Parallel Programs" appeared in the Proceedings of the Second Merged IPPS/SPDP, 1999.

[*] This work was supported by National Science Foundation CAREER Grant CCR-9733063.

be executed efficiently using a technique called *pipelining*, where each processor (node) computes its work in blocks; after each, it passes the necessary results (via an explicit message) to the next node in the pipeline. Once the pipe is full, all nodes execute in parallel; the data dependencies are satisfied by forcing a node to block awaiting data from the node that precedes it in the pipeline.

One important parameter in a pipelined program is the amount of work performed by each node before communicating. This is often referred to as the tile or *block size* and is critical to the efficiency of the program. A small block size decreases node idle time but increases the amount of communication, while a large block size decreases the amount of communication but increases node idle time. Unfortunately, an effective block size cannot always be successfully chosen purely by program inspection. There are several reasons for this. First, it is difficult to reason about workload statically for both programmers and compilers [10]. For example, including cache performance in overall workload analysis is an entire area of active research. In fact, the increasing complexity of architectures (e.g., out-of-order execution, branch predication, etc.) makes it ever more difficult to statically model a computation. Second, pipelined programs generally divide the computation into equal-sized blocks, which may not be flexible enough to accommodate scientific applications with unbalanced workloads; for example, an update may be guarded by a conditional (or, the machines themselves might have heterogeneous CPUs). Finally, the best block size may be unknown until run time if the amount of work depends on input values. Fortunately, most scientific programs are iterative (i.e., contain a loop that encloses the entire computation), and many exhibit similar characteristics on each iteration. This makes it possible to monitor a small number of iterations of the outermost loop and use the results to guide decisions on future ones.

This paper develops novel run-time analysis that, given a pipelined parallel program without a hard-coded block size, finds an effective block size. Specifically, our analysis:

- monitors the first two iterations of a pipelined computation, obtaining the times to update (1) each column and (2) blocks of two columns. Monitoring for two (instead of one) iterations allows us to estimate the effect of caching on workload.
- uses the results of the monitoring to select an effective block size. Our system uses an efficient heuristic that first chooses an initial (uniform) block size, then subdivides blocks that incur a large waiting penalty, and finally eliminates excess messages. Our choice of block size adjusts to the application and can be nonuniform.
- uses the computed block size during the rest of the computation.

Performance results are such that programs that apply our run-time analysis to choose the block size outperform (by as much as 18%) those that make use of *the best* statically chosen block size on an application with an unbalanced workload. When the workload is balanced, our programs are competitive. After the first two iterations, when there no longer is monitoring (and sequential execution), the run-time versions are sometimes even slightly faster than their static counterparts. While the initial overhead is nontrivial, it is amortized to a small percentage of total execution time as long as there are a sufficient number of iterations (which is common in scientific computing). It is important to note that we are comparing our run-time versions to *the best* static version; the latter must be found be executing each program many times with different block sizes for each configuration of nodes. On the other hand, the run-time version works well with no programmer experimentation.

The rest of this paper is organized as follows. Section 2 discusses the pipelining problem, our programming model, and related work. Section 3 describes the run-time analysis to choose the best block size, and Section 4 gives the performance results. Finally, Section 5 summarizes and discusses future work.

```
S1: x := 5;
S2: y := x;
S3: x := 3;
```

Figure 1 Three statements that illustrate three kinds of data dependencies. There is a true dependence between S1 and S2, an anti dependence between S2 and S3, and an output dependence between S1 and S3.

2 Overview and Programming Model

Our run-time analysis is designed so that it could be integrated with a compiler or run-time system that determines data dependencies and outputs a pipelined parallel program. Details on this integration are given in Section 3.3. At present we transform a sequential program to a pipelined parallel program by hand *without* specifying the block size. Our system automatically chooses this block size.

Whether a programmer, compiler, or run-time system (or a combination thereof) transforms a sequential program to a pipelined parallel program, three tasks must be performed: data dependencies must be detected, pipelined code must be generated, and a block size must be chosen. The next three subsections discuss each of these in turn. The last subsection discusses our programming model.

2.1 Detecting and Handling Data Dependencies

Dependence analysis [3] is static analysis that inspects control flow and data flow to determine when parallel execution of code does not violate its original (sequential) semantics. The two primary types of dependencies are data and control; this paper focuses on the former. A data dependence occurs when two references read or write a common memory location. There are four types of data dependencies. A true dependence occurs when a statement contains a write to a location that is later read. An anti dependence occurs when a statement contains a read of a location that is later written, and an output dependence occurs when two statements both contain a write to the same location. These three dependencies restrict program order. (The fourth, an input dependence, does not; it means that two statements contain a read of the same location.) Examples are shown in Figure 1. It should be noted that there has been a significant amount of work on automatic detection of data dependencies, including the Fortran D compiler [6], SUIF, Polaris [16], [17], SUPERB [18], and PARADIGM [19].

When a loop contains (1) a true data dependence, or (2) an anti or output data dependence that it does not remove[1], parallelization with no communication is not possible. This paper focuses on loops with true dependencies where it is possible to extract some parallelism. Such loops, often referred to as `DOACROSS` loops, are quite common in practice, and, when dependencies can be determined statically, lend themselves to pipelining.

The basic idea behind discovering opportunities for pipeline parallelism is finding loops that contain true dependencies. Consider the Alternate Direction Implicit (ADI) code fragment in Figure 2. The row sweep can be executed without any internal communication if the data (array X) is distributed by rows ({`BLOCK,*`} in HPF [1]). So, a programmer or parallelizing compiler

[1] Anti and output dependencies can be removed, but this requires extra memory [4]; hence, it is not always practical to remove them.

```
for iter := 1 to NUMITERS {
  /* row sweep */
  for i := 0 to N-1
    for j := 0 to N-1
      X[i][j] := F(X[i][j], X[i][j-1], A[i], B[i])
  /* column sweep */
  for i := 0 to N-1
    for j := 0 to N-1
      X[i][j] := F(X[i][j], X[i-1][j], A[i], B[i])
}
```

Figure 2 Outline of a sequential ADI program.

would likely choose this distribution. However, if this distribution is chosen for the first loop, then the column sweep will have a cross-processor (cross-node) dependence because $X[i][j]$ depends on $X[i-1][j]$; the distribution of X by rows results in node k requiring access to the last row of node $k-1$ in order to update its own first row. Note that the exact dependence can be determined statically.

Because of the cross-node dependence, there are four choices that can be made for the second loop:

- Serialize it.
- Transpose the matrix before the loop, allowing the loop to be run in parallel with no internal communication. This requires another transpose after the loop.
- Schedule the iterations of the column sweep such that the dependencies are not violated.
- Pipeline the loop, leaving the matrix in a row-wise distribution.

If the loop is time consuming, serializing it will be detrimental to speedup. The second choice will be best if each loop has a significant amount of computation, and communication is extremely fast, so that the transpose doesn't dominate the computation. However, for large problem sizes, a transpose, which does not typically speed up well, becomes very expensive. Several researchers have noted the inferiority of transposing to pipelining on larger problems [13, 6]. The applications that we tested in this paper do not perform an extremely large amount of computation per iteration, so a transpose would dominate the execution time[2]. The third choice includes such methods as pre-synchronized scheduling [12], staggered distribution [20], and cyclic staggered distribution [5]. Pre-synchronized scheduling splits a loop into two loops at the point where the dependence occurs. The loops are then executed by having processors remove iterations from a queue. If the iteration is from the first loop, all iterations from the second loop that are then eligible to execute are added to the queue. The staggered distribution scheme is the same except that it assigns iterations from the first loop to processors such that later processors receive more iterations. This is done so that later processors perform useful work while awaiting data necessary to begin iterations from the second loop. Cyclic staggered distribution differs from staggered distribution in that it attempts to relieve the load imbalance from the first loop by assigning iterations cyclically. All three scheduling schemes will either incur more communication or more load imbalance than pipelining, although

[2] We implemented a parallel transpose routine to verify this.

```
for iter := 1 to NUMITERS {
  /* row sweep */
  for i := start to end
    for j := 0 to N-1
      X[i][j] := F(X[i][j],X[i][j-1],A[i],B[i])
  /* (pipelined) column sweep */
  for jj := 0 to N-1 by blocksize
    if (myId != 0)
      recv X[start-1][jj:jj+blocksize] from myId-1
    for i := start to end
      for j := jj to jj+blocksize-1
        X[i][j] := F(X[i][j],X[i-1][j],A[i],B[i])
    if (myId != p-1)
      send X[end][jj:jj+blocksize] to myId+1
}
```

Figure 3 Pipelined ADI program. Variables `start` and `end` are local to each node and based on the number of participating nodes such that the work is divided evenly. A specific block size must be chosen at some point. Note this assumes that the blocksize divides N evenly.

the communication delay will most likely be smaller. (Section 2.4 explains why for our applications, improvements in delay do not outweigh the increased communication.) This paper discusses ways to efficiently pipeline and so assumes the fourth choice.

Note that we assume that all dependencies are detected outside of our system; this can be at compile or run time. In general it can be very difficult to determine the exact dependencies. Many have studied compile- and run-time methods to detect arbitrary dependencies, such as [21, 22, 7, 23].

2.2 Generating Pipelined Code

After the dependencies are determined, pipelined code must be generated. This is done by first "strip-mining" [2] the inner (j) loop, meaning that the step size is increased (by `blocksize`) and renamed (jj). A new inner loop is then added (j) that iterates over the "chunks" of the jj loop. Next, a check is made to see if it is legal to move the jj loop outside of the i loop. If so, the loops are interchanged; if not, pipelining cannot be performed. Finally, the necessary synchronization must be inserted (by using `send` and `receive` on a distributed-memory machine). This is because it is not legal for a node to start updating a block until it has received the appropriate values from the previous node. The resulting program is shown in Figure 3. The execution of this program is shown pictorially in Figure 4.

2.3 Choosing the Block Size

Once the code has been generated, an appropriate block size must be found; note that the pipelined code in Figure 3 leaves the block size open. However, it is necessary to select a block size. A smaller block size will decrease node idle time but increase the amount of communication. Conversely, a larger block size will decrease the amount of communication but increase the node idle time. The best size depends on the ratio of computation to communication; specifically, this includes the cost of updating each element of the matrix, the cost of copying a message from the application program to and from the network, and the cost of the network latency of the message itself. The latter three

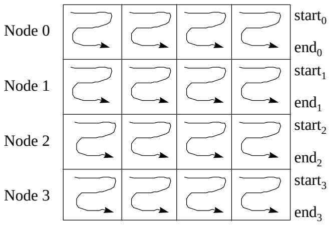

Figure 4 Picture of the general execution model of a pipelined parallel program. An individual block, which can start when the blocks above and to the left are complete, is updated in the direction of of the arrows (by subrows).

```
F(n1, n2, n3, n4) {
  if (n1 > 0)
    update point
}
```

Figure 5 A function with a nonuniform amount of work.

costs can be determined through *training sets* [24]; that is, tests are run for each new architecture and the results are made accessible to a programmer or compiler. However, static estimation of the cost of computation is problematic for two primary reasons. First and most importantly, compiler algorithms for static workload estimation are currently immature [10], and the increasing complexity of new architectures makes improvement in this area difficult. For example, current sequential architectures have some or all of the following: deep pipelines, multiple levels of cache memory, out-of-order execution, delay slots, and dynamic memory disambiguation. Future architectures will likely have branch predication. It is becoming nearly impossible to reason statically about dynamic performance of programs. Furthermore, if compilers have difficulty estimating workload, programmers will have an even more difficult time. Second, when the workload depends on values that cannot be determined until run time, the amount of work cannot in general be determined statically.

Figure 5 shows possible code that calls for a variable block size and almost certainly cannot be inferred statically. To find the best block size in general, it is necessary to use run-time analysis.

2.4 Programming Model

Our programming model encompasses many important scientific applications. In general, we support parallel programs that read from and write to arbitrary-dimensional arrays. However, there are some restrictions.

First, we support long-running, iterative scientific applications, which allows for the possibility of using run-time information to improve future iterations, as well as the amortization of our run-time overhead on the first two iterations. If the computation is not long running, the choice of block size becomes less critical (and the program may not even be parallelized in the first place).

Second, we support regular DOACROSS loops, which allows static detection of pipelining opportunities. On the other hand, it is not possible to determine statically if irregular DOACROSS loops can be pipelined; this requires an inspector/executor method [21] just to determine if pipelining is legal (and advantageous). One could then presumably choose a block size using information gathered by the inspector, although we know of no such work.

Third, we require that a loop be at least doubly nested. This allows pipelining to be efficient because the first node is able to continually produce values into the pipeline; if the loop is singly nested, the first node must wait for the last node's data, which greatly degrades the efficiency of pipelining.

Fourth, we support only BLOCK data distributions, which means that each node operates on a contiguous set of rows. (Note this is independent of the block size.) The potential advantage of BLOCKCYCLIC distributions, where each node can have several distinct sets of rows, is that it is possible to decrease waiting time even further; in particular, using BLOCK, fine-grain pipelining (block size of 1) still causes each node to wait for n/p data points. With a BLOCKCYCLIC distribution, this could be decreased to just a single point at the cost of more messages. In our experiments, fine-grain pipelining was never the best way to pipeline; in fact, the smallest block size that was ever the most effective was 8. Based on these tests, we decided that supporting BLOCKCYCLIC distributions was not worth the added overhead (more timing measurements, more complex algorithm to choose a block size, etc.).

Finally, we only pipeline in a single dimension. If an application has a triply nested loop over a three-dimensional array (as our airshed code does), we can actually pipeline it in either one or two dimensions. With two dimensional pipelining, each block can require several messages instead of one; as with BLOCKCYCLIC distributions, this can decrease waiting time but increases the number of messages. Integrating two-dimensional pipelining into our run-time execution model would add significant overhead because the search space is much larger. More work is needed to determine how well a run-time approach will perform in this more general model.

3 Run-Time Analysis

We assume that the a DOACROSS loop is transformed to a pipelined loop as in Figure 3. However, instead of choosing the best block size statically, we use the following approach:

- During the first two iterations of a pipelined computation, we measure the time to compute each column as well as each pair of columns[3].
- At the end of the second iteration, the column times are sent to a master node.
- The master node (1) chooses an initial (uniform) block size, (2) looks for blocks that cause excessive waiting and subdivides them recursively, and (3) eliminates excess messages by re-executing the algorithm on adjacent blocks that are not subdivided.
- The block size is no longer necessarily uniform (as shown in Figure 4); it could, for example, look like that of Figure 6. The resulting pipeline *schedule* is sent to each node, and it is used for the rest of the computation.

[3] Taking measurements on the first iteration can sometimes be inaccurate due to initialization effects such as cold caches. We experimented with delaying measurements until the second and third iterations, but found that there was no significant difference. This is because in our applications, all arrays are initialized before the main computation loop.

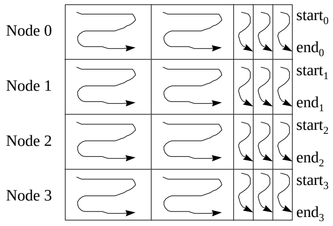

Figure 6 Picture of a pipelining schedule with a nonuniform block size.

For the following explanation, we will assume a two-dimensional problem of size $n \times n$ (all indices are assumed to start at 0) with p nodes numbered 0 through $p-1$.

There are two main components of developing a run-time model for pipelined parallel programs. First, a model for sequential performance on each node must be developed. Second, a model for pipelined parallel performance involving all nodes must be developed. The next two sections discuss each of these in turn. The last section discusses issues involved with potential integration of our analysis with a compiler.

3.1 Sequential Model

As described earlier, in a pipelined computation points of a data structure are grouped into *blocks*; when each point in a block is updated, we say the block itself is updated. (For our work a block is always a subarray.) The core of modeling sequential performance is developing a formula for the time taken per node to update each of its blocks. We denote $T^k_{i,j}$ as the time to update the j^{th} block on node i using a block size of k. (As will be discussed in Section 3.2, we will limit consideration to those k that are powers of 2.) The quantity $T^k_{i,j}$ includes $f_{send}(k)$, the time to copy the boundary points in block j (needed by the next node in the pipeline) to the network, if necessary (node $p-1$ need not copy). (We will discuss $f_{send}(k)$ further in Section 3.2.)

To obtain $T^k_{i,j}$ for each node, we measure the time to compute each column on each node during the first iteration. We must measure each column because one extreme that we will consider—fine-grain pipelining—uses blocks that consist of single columns. By timing columns, we limit the number of timer calls (`gethrtime`) to $O(n)$ (twice per column), while the workload is at least $O(n^2)$. (Otherwise, to recover the times for individual columns it might be necessary to time each *point*, which would increase the overhead to a significant $O(n^2)$.)

There are two nontrivial issues with sequential performance. First, we must measure all subcolumns on all nodes. Second, we must incorporate caching into our model.

3.1.1 Measuring Column Times

The first issue that arises is how to acquire column times for each node. We choose to execute the first iteration *sequentially* on node 0 (the master node). Node 0 performs all computation and measures the times for the subcolumns that each node will perform once the data is distributed. The primary disadvantage to this scheme is that the first iteration is executed sequentially rather than in parallel, but this overhead is amortized over a large number of iterations. The other possibility is to execute the first iteration in parallel, have all nodes measure their own computation, and have each node send all of its column times to node 0 (this was the algorithm we previously used in [25]). The primary problem with this is that inaccurate measurements may result on all but the first node; for example, while the second node is updating its first column, the first node may send *several* columns, interrupting the second node's computation and perturbing the measurements.

After the first iteration, node 0 will have a vector t of size p, where each vector element is itself a vector of n column times; $t_{i,j}$ denotes the time to update column j on node i. It might appear that we can compute the time for all blocks in a straightforward manner, i.e., $T^k_{i,j} = \sum_{l=j\cdot k}^{(j+1)\cdot k-1} t_{i,l}$ + $f_{send}(k)$ for nodes 0 through $p-2$, and $T^k_{i,j} = \sum_{l=j\cdot k}^{(j+1)\cdot k-1} t_{i,l}$ for node $p-1$.

However, pipelining changes the iteration space of a loop from a row sweep to a column-oriented sweep (blocks consist of groups of columns). For reasonably large problems, this can have an adverse effect on cache behavior. If fine-grain pipelining is used, the computation proceeds column-wise on each node; if the column size exceeds the cache size, every array access that varies with the enclosing loop indices will result in a memory access. (We use C, which is row major; the reverse problem will occur in Fortran.) We verified this phenomenon with the Shade cache simulator [11] that we parameterized to model the first-level data cache of the Pentium Pro architecture (which is 8K with a 32 byte line size). We executed a sequential (pipeline-style) program that simply writes to each element of a two-dimensional array of double-precision numbers. With a block size of one, every write resulted in a cache miss; with a block size of two, every other resulted in a miss. With block sizes 4 and higher, every fourth write resulted in a cache miss. Similar observations have recently been noted in [26]. Therefore, estimating larger block times by simply summing the columns within that block will overestimate execution times for all blocks with size greater than one.

3.1.2 Incorporating Caching

The second issue is how to incorporate caching into our model. Determining its effect statically is difficult. We have devised a method to include caching with a reasonable overhead: we measure an additional (sequential) iteration using a block size of *two* instead of one. This will mean that there is the potential to reuse each cache block once, assuming the block holds data for array accesses that vary with the enclosing loop indices. Whether this actually results in a significant difference between a block of size 2 and the sum of its two corresponding consecutive columns depends on several complicated factors (the number of distinct array accesses and conflict misses, the total size, line size, and associativity of the cache, number of non-array instructions in the loop, etc). With knowledge of only the line size of the cache, we can develop an effective run-time model, which is discussed below. On the other hand, a purely static model must take all of these factors into account; a good representative paper is [27]. However, besides being limited by what can be inferred, research on choosing block sizes by statically modeling the cache generally focuses on choosing block size to improve locality in *sequential* programs; as we will see in Section 3.2, we must trade off locality with a number of other factors, including message send and receive time as well as wait time. Therefore, any analysis to choose block size for pipelined parallel programs must include accurate workload analysis, which we have already argued is problematic.

We will explain the inclusion of caching into our run-time model with the help of the example in Figure 7; in particular, assume that for node 0, measuring the first two iterations finds that columns 0-7 each take 4, 4, 5, 5, 5, 5, 6, and 6 time units respectively, and blocks [0,1], [2,3], [4,5], and [6,7] take 6, 6, 6, and 9 time units. First, we compute the difference between the sum of two consecutive columns with their corresponding block of size 2 and store this in a vector called *cacheOverhead.*

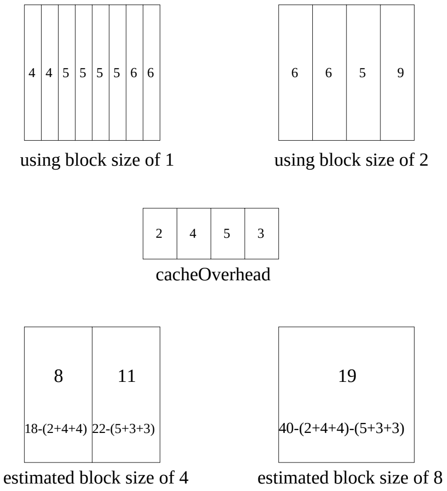

Figure 7 Estimating block execution times taking caching into account. At the top are the two measurements our system takes (block sizes of one and two). The cache overhead is then the difference of columns i and $i + 1$ with block $i/2$. We then compute the effective times for larger blocks by summing the individual columns in the block and subtracting to take into account that only a single memory access per cache block is necessary. For example, with a block size of 4, the first block will incur one memory access (and take time 4) but then hit in the cache 3 times (which takes time 2, 1, and 1). This example assumes that four array elements fit into a cache block.

(This vector is of size $n/2$.) In our example, the differences would be 2, 4, 5, and 3, respectively ($4 + 4 - 6$ for the first entry, $5 + 5 - 6$ for the next, and $5 + 5 - 5$ and $6 + 6 - 9$ for the last two). Then, starting with $T_{i,j}^k$ as computed above (in isolation, without considering cache behavior), we subtract from each block of 4, 8, $\ldots$, n the cache overhead, unless we know that a cache miss will result. We assume that a cache miss must occur every four elements, because on the Pentium Pro the line size is 32 bytes and our arrays contain doubles. In our example, the initial block times for [0,1,2,3] and [4,5,6,7] would be 18 and 22 (the respective sums over columns). Then, assuming that first column of each block results in a cache miss, we subtract 2, 4, and 4 from the first block and 5, 3, and 3 from the second block, resulting in 8 and 11 as the effective block times. Note that the first and third elements of *cacheOverhead* are subtracted only once each, because the first and fifth columns require a memory access. For a block size of 8, note that the effective time is the sum of the effective times of the two blocks of 4. This is expected because no additional caching benefit exists with a block size of 8—a cache line can only hold four doubles. The same principle holds for block sizes of 16, 32, $\ldots$, n. This gives us accurate estimates of $T_{i,j}^k$ for all k.

3.2 Pipelined Parallel Model

Now that we have accurate times for the computation of each block, we can develop an accurate model for a pipelined parallel program. We define, for a message of size x, $f_{send}(x)$ and $f_{recv}(x)$ as the overhead due to copying a message to (from) the network, and $f_{net}(x)$ as the latency. They are computed from separate experiments (training sets) where several test runs are performed and averaged.

Note that our current model does not take interrupt time into account as in [28]. Interrupt time is the operating system time taken to copy an arriving message to an operating system queue. We consider only send and receive overheads, as in [29]. (The work in [28] does not select block sizes at run time.) We are investigating the effect of interrupts in our current research; this will be discussed further in Section 5.

Our algorithm to choose an effective pipelining schedule works in four steps. First, choose an initial (uniform) block size. Next, look for blocks that cause excessive waiting and subdivide them recursively. Finally, eliminate excess messages by re-executing the algorithm on adjacent blocks that are not subdivided. Finally, send the schedule describing when messages are sent and received to each node.

3.2.1 Choosing an Initial Block Size

The first step is to choose an initial block size; we will consider all block sizes that are a power of two, as these allow us a reasonably wide range of choices at a small cost. We denote $S^k_{i,j}$ as the time that node i can *start* its computation of block j using a block size of k. The basic idea is: for each block size, estimate its completion time using the block update times from each node along with the message overheads. For a given block size, node 0 commences the computation at time 0 (i.e., $S^k_{0,0} = 0$) by updating its first block, which takes time $T^k_{0,0}$; node 1 must wait to start updating its first block until it receives a message from node 0 (due to the data dependence). Hence, the time that node 1 can start updating its first block is given by

$$S^k_{1,0} = S^k_{0,0} + T^k_{0,0} + f_{net}(k) + f_{recv}(k).$$

(Recall that $T^k_{i,j}$ includes $f_{send}(k)$ if necessary.) Node 0 can start updating its second block at time

$$S^k_{0,1} = S^k_{0,0} + T^k_{0,0}.$$

Note that the second message will arrive at node 1 at time $S^k_{0,1} + T^k_{0,1} + f_{net}(k)$. Node 1 will be ready for the message after it updates its first block, which occurs at time $S^k_{1,0} + T^k_{1,0}$. The message may arrive before node 1 has finished updating its first block, in which case it can start as soon as it finishes updating $T^k_{1,0}$. Otherwise, node 1 must wait until this message arrives. Note that either way, node 1 must read the message from the network before starting its second block. Hence, the time at which node 1 can start updating its second block is

$$S^k_{1,1} = \max\{S^k_{0,1} + T^k_{0,1} + f_{net}(k), S^k_{1,0} + T^k_{1,0}\} + f_{recv}(k).$$

In general, a formula for $S^k_{i,j}$ is given as follows:

$$S^k_{0,0} = 0$$

$$S^k_{i,0} = S^k_{i-1,0} + T^k_{i-1,0} + f_{net}(k) + f_{recv}(k)$$

and, for $1 \le j \le n/k - 1$,

$$S^k_{i,j} = \max\{S^k_{i-1,j} + T^k_{i-1,j} + f_{net}(k), S^k_{i,j-1} + T^k_{i,j-1}\} + f_{recv}(k).$$

This indicates that node i can start block j at the *later* of two times: (1) when it has finished updating all of its own previous blocks and (2) it has received the message for the same numbered block on node $i-1$. The completion time is then $S^k_{p-1,n/k-1} + T^k_{p-1,n/k-1}$, which is the time that it takes node $p-1$ to start updating its last block plus the update time for that block. We estimate the completion time for $k = 1, 2, 4, \ldots n$, and take the smallest such time; the algorithm has a complexity of $O(pn \log n)$. Figure 8 illustrates the tradeoff (for two columns) between using a block size of 1 and 2.

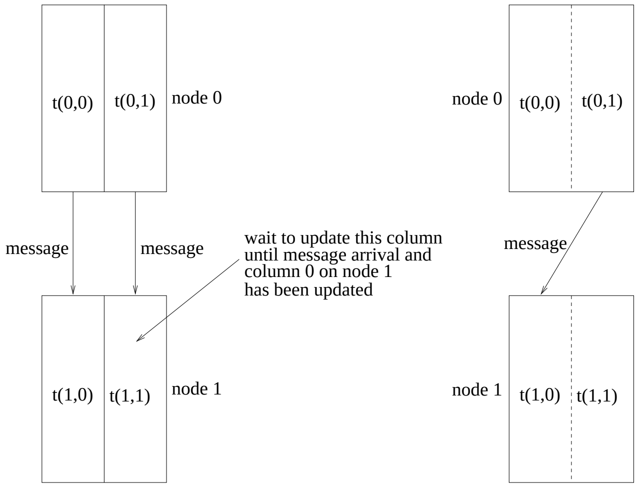

Figure 8 The difference between block sizes of 1 (left) and 2 (right). On the left, node 0 sends a message after updating each column, which means that node 1 must wait to start its second column until it has finished its first column and it has received the necessary data from node 0. On the right, node 0 updates both columns before sending a message; after receiving it, node 1 can update both of its columns. Recall that $t_{i,j}$ is the time to compute column j on node i.

3.2.2 Subdividing Blocks

The next step is to improve upon the initial block size by subdividing blocks. Consider the case where most of the work is clustered in one part of the matrix. A compromise is made in our first step (above) between a fine granularity, where there is less idle time in the part of the matrix where a lot of work is done but excess messages when updating the rest of the matrix, and a coarse granularity, where there are fewer messages sent but significant idle time where the work is done. Without simply decreasing the initial block size (which can substantially increase the message overhead), we would like to reduce the idle time.

Our analysis handles this using the following heuristic. We start by using the best block size (denoted *best*) determined with the above analysis and attempt to improve upon it by potentially subdividing blocks. We have already computed the start time for each block on each node ($S_{i,j}^{best}$); in doing so, we know the time that node i must wait for a message from node $i-1$, if any. Recall that $S_{i,j}^{best}$ is computed by taking the maximum of (1) the time to compute the first $j-1$ blocks on node i and (2) the time to compute the first j blocks on node $i-1$ and transfer a message to node i. When the time given by (1) is *smaller* than the time given by (2), node i must wait for a time equal to the difference between (2) and (1). On the other hand, if the time given by (1) is greater, there is no waiting time. The key waiting time is that of the *last* node ($p-1$), because the pipeline cannot be completely full while the last node is waiting for data. So, for each block on node $p-1$ we compute the waiting time; any block that has a relatively large waiting time might benefit from being split into smaller blocks, which may reduce the waiting time. Hence, we invoke the first step of our algorithm recursively on the first block whose waiting time exceeds our prespecified threshold (we used 10% of the total wait time). This will in fact effect a finer block size if profitable, because the message overhead incurred by finer-grain pipelining will not be nearly as significant on a smaller block (there are fewer points). After subdividing this block, we recompute the total wait time as well as the wait time for each block, taking into account the subdivided

```
for iter := 1 to NUMITERS {
  // row sweep omitted
  while not at end of schedule {
    last := getLastReceivePoint(schedule)
    next := getNextReceivePoint(schedule)
    recv X[start-1][last:next] from myId-1
    for i := start to end
      for j := last to next
        X[i][j] := F(X[i][j],X[i-1][j],A[i],B[i])
    send X[end][last:next] to myId+1
  }
}
```

Figure 9 Pipelined ADI program with our system. The *schedule* is generated by our run-time analysis. It supports operations `getLastReceivePoint` and `getNextReceivePoint`, which are used to determine what points should be received. They allow for a nonuniform block size.

block. We then look for a (different) block to subdivide. When no blocks need to be subdivided, we are done. We found this simple heuristic to perform well in practice.

3.2.3 Eliminating Excess Messages

The third step improves upon the first step without modifying the regions that were subdivided in the second step. We do this by eliminating excess messages. In the example above we use fine-grain pipelining on the parts of the matrix with significant computation; similarly, we want to use coarse-grain pipelining on the parts of the matrix where there is little work. To realize this, we re-run the original (global) algorithm on consecutive groups of blocks that were *not* subdivided. This eliminates unnecessary messages when part of the matrix contains very little work, but the initial global block size was relatively small due to significant work on a different part of the matrix.

3.2.4 Distributing the Schedule

After the completed schedule has been determined by node 0, it sends each node its schedule before execution of the *third* iteration. For the rest of the computation, each node executes and communicates according to this schedule; an outline of the code is shown in Figure 9. Note that executing the first two iterations sequentially means that all the data resides on node 0 during that time; data is also distributed to the nodes before the third iteration.

3.3 Integration with a Compiler

Although our analysis is currently stand-alone, it is designed so that it can be integrated with a compiler that determines data dependencies and possibly transforms a sequential program to a pipelined parallel one. This section discusses the basic implementation ideas.

First, to integrate our system with a compiler that transforms a sequential program to a distributed-memory pipelined parallel one, such as PARADIGM [19], would be conceptually straightforward. The compiler subsystem that determines block size would be disabled; furthermore, instead of generating code that used a static block size as in Figure 3, code would be generated to use the schedule generated by our system, shown in Figure 9. This would certainly require a modification of the code generator. Code would also need to be inserted to (1) sequentialize the first two iterations, (2) obtain the times on those iterations in a similar way that our system, and (3) insert a call to invoke our analysis after the second iteration. While the implementation effort for these items using a compiler like PARADIGM is unknown at this time, it appears possible.

Integrating our system with a shared-memory parallelizing compiler such as SUIF presents an additional hurdle. This is because we require explicit message passing code. In this case we have two options. First, the compiler could be retargeted to generate distributed-memory code; however, this is an extremely time-consuming project. The other possibility is to integrate the compiler with a distributed shared memory system, which several researchers have investigated in the context of SUIF [30, 31]. Unfortunately, DSM systems are ill-equipped to handle pipelined programs. In separate work we have added explicit support to DSM systems to support pipelining [32]. This means that all the pieces necessary to integrate our analysis with SUIF exist. Still, the total implementation effort involved to make such an integrated system work would be significant and surely more involved than retargeting a distributed-memory compiler.

4 Performance

This section reports the performance of four programs. The first three are ADI integration, Hydro (an implicit hydrodynamics kernel), and Gauss-Seidel iteration. For each, an effective block size might be able to be inferred statically. The fourth is an adaptation of an airshed simulation where the workload is not uniform; it represents applications that need to use run-time information and nonuniform block sizes to achieve good performance. Although the programs are relatively small in size, they are indicative of larger programs (or may be called as subroutines in them).

For each application we developed a program that uses our run-time analysis. For an accurate comparison, we also developed a program with a (parameterizable) static block size that does not perform any run-time analysis. In addition, we implemented a separate sequential program. For each application we present the results of the program with the *best* static block size and compare it to the program that uses our run-time analysis. It is important to note that finding the best static block size requires experimenting with many block sizes for 2, 4, and 8 nodes, whereas the run-time version is run once for each set of nodes.

Below, we first discuss the overheads incurred by our run-time analysis. Next, we discuss the accuracy of the predicted execution times of our model. Then, we briefly describe the four applications and present the results of runs on 2, 4, and 8 nodes, along with (1) a comparison between the best and worst static block size and (2) a measure of the time per iteration excluding the first two (where monitoring occurs). The sequential program times are also reported; we chose problem sizes such that sequential programs took around 80-150 seconds[4].

All tests were run on a network of 8 Pentium Pros connected by a 100 Mbs Fast Ethernet, using Solaris and the `cc` compiler with the `-O` flag. The execution times reported are the median of at least three test runs, as reported by `gethrtime`. The tests were performed when the only other active processes were daemons.

[4] Note that the longer the tests run, the better the programs using run-time analysis perform relative to the programs that use a statically determined block size. This is because the overhead is amortized over more iterations.

Nodes	2	4	8
ADI	1	5	11
Hydro	4	7	13
Gauss-Seidel	2	3	6
Airshed	3	7	13

Table 1 Total overhead, to the nearest percent, of our run-time analysis on each of our applications. This percentage is computed by determining how much longer the first two iterations take than with the best statically chosen block size. Note that the larger the number of iterations, the smaller the overhead percentage will be. We chose a moderate number of iterations (300 at most), and the moderate overhead on 8 nodes reflects that.

4.1 Overhead of Run-Time Analysis

In general, we found the overheads of the run-time analysis in our system to be small. They consist of the following quantities:

- executing the program sequentially on the first two iterations,
- taking measurements (`gethrtime`),
- estimating completion time for the (possibly nonuniform) block sizes,
- adding extra code to allow nonuniform block sizes according to the schedule, and
- choosing an ineffective block size (when an effective size can be statically determined) due to timing inaccuracies.

The most significant overhead arises from executing the program sequentially on the first two iterations; however, it is manageable as long as it is amortized over a reasonably large number of iterations. However, sequential execution has a higher relative cost as the number of nodes increases. It is important to note that executing the first two iterations in parallel has its own overheads (for example, sending the times to node 0 can become a bottleneck for larger number of nodes). The time to take measurements involves 2 system calls per column; a call to `gethrtime` takes about 4.5 microseconds, so total timing overhead is on the order of milliseconds. The time to choose the best block size increases with the number of nodes, but even on our eight-node tests, it was small (a few tenths of a second). The extra code to allow nonuniform block sizes (which involves using the generated schedule to decide when to send and receive) is outside of the main computation loops and so is negligible.

A potentially serious problem is choosing an ineffective block size. In the tests we performed, our run-time analysis did not always choose the global block size that was the best (see the 2-node Hydro test as well as Gauss-Seidel); however, the one it did choose had performance that was practically identical to the best block size. In general, though, it is possible to choose a poor block size if wallclock timing is inaccurate due to the existence of other user processes. However, because we are operating as the only user on the machine, this has not happened in our tests. Table 1 shows the overall overhead percentage on 2, 4, and 8 nodes for our applications. Note that we use a moderate number of iterations; for a reasonably large number, the overhead would be very small.

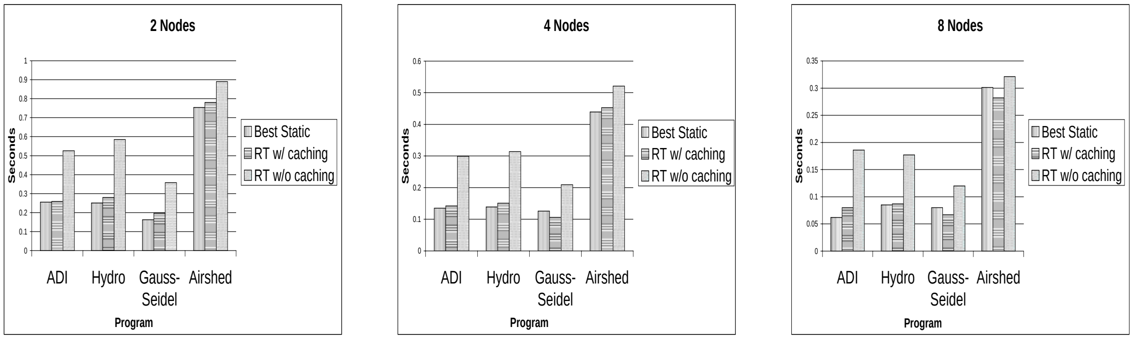

Figure 10 Accuracy of the model on the four appplications for two, four, and eight nodes, respectively. The time shown, in seconds, is the time for *just the pipelining portion* on a single iteration. (ADI and Airshed have non-pipelining phases, so the total iteration time is greater than that shown here.) The run-time version that incorporates caching is much more accurate than the one without.

4.2 Model Accuracy

It is important to inspect the accuracy of the predicted execution times of our model. Figure 10 shows the execution times of a single iteration for 2, 4, and 8 nodes on each of our programs. Data is shown for the best static block size and the estimation of the completion time using that block size by our run-time model both with and without taking caching into consideration. (Only the pipelined portion of the iteration is measured.) Clearly our model is much more accurate when caching is taken into account; the estimated iteration times often are around double the actual times. This is because the intra-node block times for block sizes of 4, 8, . . . , n are overestimated; what really are cache hits are modeled as memory accesses. Our model is usually within 10% of the actual time per iteration. The figure does not show how well the model performs on block sizes other than the best; it is most inaccurate in estimating performance of a block size of 1. The difference in that case can be as much as 20% from actual. We tend to underpredict the actual time in this case (and to a lesser extent with a block size of 2); this is most likely because the large number of messages causes many interrupts on all but the first node, and our current model does not take this into account.

4.3 ADI

ADI (Alternate Direction Implicit) integration is one way to implement numerical integration. It can be parallelized exactly as shown in Figure 3.

The execution times for two versions of ADI, size 1024, are shown in Table 2. The program that uses our run-time analysis first finds a global block size of 32; then, interestingly, it subdivides only the first block (due to the initial waiting time) into blocks of size 4. The best static block size on 2, 4, and 8 nodes was 32. As can be seen from Figure 11, on 2 nodes, the run-time version was in fact slightly better than the static version on a per-iteration basis after the second iteration. This appears to be because the waiting time is reduced by the subdivision of the first block. However on 8 nodes, the run-time version performs slightly worse; there is less work, so the waiting time is not as severe. The overhead of the first two iterations accounts for almost all of the difference between

Nodes	Time (2/4/8)	Blocksize (2/4/8)
Run-Time	76.1/41.1/23.7	32/32/32 (*)
Best Static	74.5/38.8/21.5	32/32/32
Sequential Time	135	

Table 2 ADI integration, 1024×1024, 100 iterations (times in seconds). Run-Time refers to the program that makes a run-time choice of block size, and Best Static refers to the best out of the static programs. The (*) indicates that a size of 32 was chosen for all blocks except the first, which was subdivided into blocks of 4 by our algorithm due to its waiting time.

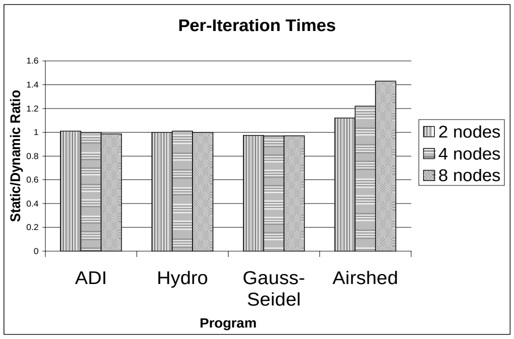

Figure 11 Ratio of time per iteration of static version to run-time version for each application on two, four, and eight nodes. A ratio greater than one indicates that the run-time version is superior.

the static and run-time version. Note also that a poor choice of block size can be detrimental; for example, on 2 nodes, a static block size of 1 is 39% slower overall on each iteration than a block size of 32 (the best); furthermore, the pipelining part of the computation is three times slower. Figure 12 indicates this disparity in performance. Finally, because the workload is uniform in ADI, an effective block size *might* be able to be chosen statically. Still, our run-time analysis can sometimes produce a slightly *better* one, even though this is a program for which static analysis is possible.

4.4 Hydro

Hydro is kernel number 23 from the Livermore Loops [9]; it is an excerpt from an implicit hydrodynamics code. It consists of a prespecified number of iterations, where each updates every point on a two-dimensional matrix. Because each update is based on a four point stencil on the *same* matrix, there is a data dependence that prevents full parallelization of the loop.

The execution times for two versions of Hydro are shown in Table 3. The run-time version first finds a global block size of 32; as with ADI, it then subdivides only the first block. However, although this improves the per-iteration time, the improvement is extremely small because the work per iteration of Hydro is much less than that of ADI. The per-iteration times found by our analysis are virtually identical or just slightly better than their static counterparts (see Figure 11).

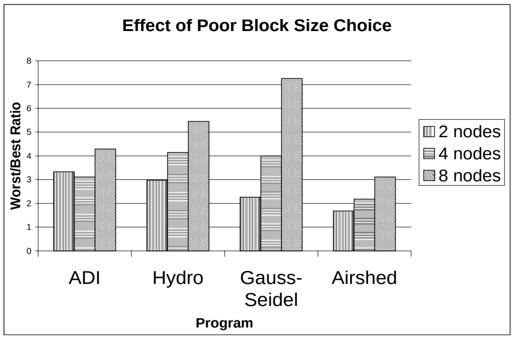

Figure 12 Ratio of worst static block size to best static block size for each application on two, four, and eight nodes. The worst block size was 1 for each application except airshed, where it was 512. (A block size of 1024 was not considered because that sequentializes program execution.)

Nodes	Time (2/4/8)	Blocksize (2/4/8)
Run-Time	49.6/30.0/18.8	32/32/32 (*)
Best Static	48.0/28.3/16.9	64/32/32
Sequential Time	83.1	

Table 3 Hydro kernel, 1024 × 1024, 200 iterations (times in seconds). Run-Time refers to the program that makes a run-time choice of block size, and Best Static refers to the best out of the static programs. The (*) indicates that a size of 32 was chosen for all blocks except the first, which was subdivided into blocks of size 4 by our algorithm due to its waiting time.

As with ADI, the overhead of the first two iterations causes the run-time version to be slower than the static version; on 8 nodes the overhead is 11.2%. The best static block size for Hydro was 64 on two nodes and 32 on four and eight nodes. (On two nodes, performance was nearly identical with block sizes of 32 and 64.) A poor choice of block size has a similar effect as that seen with ADI.

4.5 Gauss-Seidel

Laplace's equation in two dimensions is the partial differential equation $\bigtriangledown^2(\Phi) = 0$. Given boundary values for a region, its solution is the steady-state values of interior points. These values can be approximated numerically by using a finite difference method such as Gauss-Seidel iteration, which repeatedly computes new values for each point for a specified number of iterations [8]. Our program uses an eight point stencil.

Gauss-Seidel iteration is efficient compared to explicit iterative techniques such as Jacobi iteration, but hard to parallelize because the computation of each point depends on both old and new values in the array. This prevents parallelization with communication only on the boundaries of each node's subarrays. A "wavefront" parallelization method may be used, because parallelism is possible on each diagonal. Another way to solve it is to use pipelining, which is the method we employ. Computation proceeds sequentially within a block, which preserves the dependencies in

Nodes	Time (2/4/8)	Blocksize (2/4/8)
Run-Time	69.9/40.6/26.5	16/16/16 (*)
Best Static	67.2/38.3/23.9	32/32/32
Sequential Time	109	

Table 4 Gauss-Seidel, 1024 × 1024, 300 iterations (times in seconds). Run-Time refers to the program that makes a run-time choice of block size, and Best Static refers to the best out of the static programs. The (*) indicates that a size of 16 was chosen for all blocks except the first, which was subdivided into blocks of size 4 by our algorithm due to its waiting time.

the application.

The results of our Gauss-Seidel experiments are shown in Table 4. One notable difference between these tests and the Hydro tests are that the run-time version found a smaller block size (16), although the time is only slightly worse than with a size of 32. It also subdivided the first block; however, the time per iteration is inferior to the program with the best static block size, as can be seen from Figure 11. We believe that this is due to slight timer variations caused by the very fine-grain nature of this program. Also, the fine granularity causes the extremely poor performance of the worst static block size, which was 1; almost of all of the time is spent sending and receiving messages.

4.6 Airshed Simulation

Our fourth test program is an airshed simulation, which models the formation, reaction and transport of atmospheric pollutants and related chemical species[5]. Our benchmark program is adapted from the `tpsuite` from Carnegie Mellon [33]. Our version performs only the main computational loop of the calculation, which consists of a transport calculation, followed a chemistry phase, followed by another transport calculation. For our purposes, the important computational aspect of the transport calculation is to perform a row-wise (across the first dimension) update to every element of a three-dimensional array. The chemistry phase also updates the array, but in a column-wise manner, making pipelining a reasonable way to parallelize this program. The chemistry phase has an unbalanced workload, making run-time analysis especially important for this program. This program represents less regular applications, where the workload is nonuniform.

The execution times for two versions of the airshed simulation are shown in Table 5. The work was clustered at the right end of the matrix, so an effective composition of blocks is to use larger blocks (size 32) that encompass the part of the matrix that has little work and then fine-grain pipelining (block size of 1) in the part where there is significant work. The program using our run-time analysis found this block size by finding an initial static block size of 4 (step 1 of our algorithm), subdividing the right part of the schedule, on which a block size of 1 was obtained (step 2 of our algorithm), and eliminating excess messages from the rest of the schedule, on which a block size of 32 was obtained (step 3 of our algorithm). The best static block size in our experiments was 8; this is a compromise—it avoids the severe message overhead of fine-grain pipelining where there is little work and severe load imbalance where there is significant work. Still, the static program incurs more overhead than necessary on all parts of the matrix, which accounts for the superiority

[5] The general airshed program allows both task and data parallelism; we only exploit data parallelism.

Nodes	Time (2/4/8)	Blocksize (2/4/8)
Run-Time	80.3/45.2/26.9	{32/1}/{32/1}/{32/1}
Best Static	86.4/50.8/32.8	8/8/8
Sequential Time	135	

Table 5 Airshed simulation, 1024 × 1024 × 4, 100 iterations (times in seconds). Run-Time refers to the program that makes a run-time choice of block size. It uses a block size of 4 on the first 1000 columns and a block size of 1 (fine-grain pipelining) on the last 24. Best Static refers to the best out of the static programs.

of the run-time version. This can be seen from Figure 11 on a per-iteration basis. The disparity in this application is enough that the run-time version is superior even given the overhead of the first two iterations; overall, it is 7%, 11%, and 18% faster on two, four, and eight nodes, respectively.

5 Summary

We have presented a run-time approach to selecting block sizes in pipelined parallel programs. This allows us to choose an effective block size even when a source program is not amenable to static analysis. Furthermore, our system allows the choice of block size to be nonuniform, which allows increased flexibility. Our analysis monitors the program for two iterations, builds an execution model that includes the effect of caching, and takes a three-step procedure to find an effective block size: choose an initial block size, subdivide blocks that incur a large waiting penalty, and then eliminate excess messages.

We implemented our system on a cluster of 8 Pentium Pros. The programs that made use of run-time information to select block sizes had faster execution times than those that make a static choice when the workload is unbalanced. For programs that are amenable to static analysis, the programs that use our system are competitive. Our run-time system provides a potentially attractive target for a parallelizing compiler; in particular, a sequential program need only be translated to a pipelined parallel program. No workload analysis needs to be done to find an effective block size. We believe that our system is a viable way to efficiently execute a larger class of pipelined programs than previously possible.

Future Work

We intend to continue work on using run-time analysis to choose block sizes in pipelined parallel programs; there are many avenues that can still be explored. These include integrating interrupts into our model, investigating better (graph-theoretic) algorithms to choose the block size, and implementing efficient pipelining in distributed shared memory systems.

Including interrupts into the model (as in [28]) will be a challenge. It means that new recurrences will need to be developed; the work in [28] does not represent the problem with a discrete formulation as we do. How the new formulation will interact with our analysis to select block sizes is unknown.

Although our heuristic worked well, we are looking at finding even better algorithms to choose block sizes as well as allowing a larger search space. We have discussed formulating the problem using an acyclic directed graph and finding the longest path, which would then represent the completion time. The goal is then to improve this path. We would also like to determine how far our solution is from optimal.

6 Acknowledgements

John Kececioglu provided us with invaluable assistance on all aspects of this work.

References

[1] HPF–2 scope of activities and motivating applications. November 1994.

[2] D.B. Loveman. Program improvement by source-to-source transformation. *Journal of the ACM*, 24(1):129–138, January 1977.

[3] J.R. Allen and K. Kennedy. Automatic translation of Fortran programs to vector form. *TOPLAS*, 9(4):491–542, October 1987.

[4] David Padua and Michael Wolfe. Advanced compiler optimizations for supercomputers. *Communications of the ACM*, 29(12):1184–1201, 12 1986.

[5] A.R. Hurson, J.T. Lim, and B. Lee. Extended staggered scheme: a loop allocation policy. In *World IMACS Conference*, pages 1321–1325, 1994.

[6] Chau-Wen Tseng. *An Optimizing Fortran D Compiler for MIMD Distributed-Memory Machines*. PhD thesis, Rice University, January 1993.

[7] Ding-Kai Chen and Pen-Chung Yew. On effective execution of nonuniform DOACROSS loops. *IEEE Transactions on Parallel and distributed systems*, 7(5):463–476, May 1996.

[8] Gregory R. Andrews. *Foundations of Multithreaded, Parallel, and Distributed Programming*. Addison/Wesley, 2000.

[9] F. McMahon. The Livermore Fortran Kernels: A computer test of the numerical performance range. Technical Report UCRL-53745, Lawrence Livermore National Laboratory, 1986.

[10] Ken Kennedy. Compiling a software bridge to the 21st century—invited talk at PPOPP 97. June 1997.

[11] Robert F. Cmelik and David Keppel. A fast instruction-set simulator for execution profiling. TR SMLI 93-12, Sun Microsystems Labs, 1993.

[12] V.P. Krothapalli and P. Sadayappan. Dynamic scheduling of DOACROSS loops for multiprocessors. In *Proceedings of Parbase-90: International Conference on Databases and Parallel Architectures*, pages 66–75, 1990.

[13] Daryl Olander and Robert B. Schnabel. Preliminary experience in developing a parallel thin-layer navier stokes code and implications for parallel language design. In *Proceedings of the 1992 Scalable High Performance Computing Conference*, 1992.

[14] G. McRae, W. Goodin, and J. Seinfeld. Development of a second-generation model for urban air polution - 1. model formulation. *Atmospheric Environment*, 16(4):679–696, 1982.

[15] A.R. Hurson, Joford T. Lim, Krishna M. Kavi, and Ben Lee. *Parallelization of DOALL and DOACROSS Loops — A Survey*, volume 45, pages 53–103. Academic Press Ltd., 1997.

[16] William Blume, Ramon Doallo, Rudolf Eigenmann, John Grout, Jay Hoeflinger, Thomas Lawrence, Jaejin Lee, David Padua, Yunheung Paek, Bill Pottenger, Lawrence Rauchwerger, and Peng Tu. Parallel programming with Polaris. *IEEE Computer*, 29(12):78–82, December 1996.

[17] Mary W. Hall, Jennifer M. Anderson, Saman P. Amarasinghe, Brian R. Murphy, Shih-Wei Liao, Edouard Bugnion, and Monica S. Lam. Maximizing multiprocessor performance with the SUIF compiler. *IEEE Computer*, 29(12):84–89, December 1996.

[18] H.P. Zima, H.J. Bast, and M. Gerndt. SUPERB: A tool for semi-automatic MIMD/SIMD parallelization. *Parallel Computing*, 6(6):1–18, January 1988.

[19] P. Banerjee, J.A. Chandy, M. Gupta, E.W. Hodges IV, J.G. Holm, A. Lain, D.J. Palermo, S. Ramaswamy, and E. Su. The PARADIGM compiler for distributed-memory multicomputers. *IEEE Computer*, 28(10):37–47, October 1995.

[20] A.R. Hurson, J.T. Lim, K. Kavi, and B. Shirazi. Loop allocation scheme for multithreaded dataflow computers. In *Proceedings of the 8th International Parallel Processing Symposium*, pages 316–322, 1994.

[21] Joel H. Saltz, Ravi Mirchandaney, and Kay Crowley. Run-time parallelization and scheduling of loops. *IEEE Transactions on Computers*, 40(5):603–612, May 1991.

[22] Peiyi Tang and John N. Zigman. Reducing data communication overhead for doacross loop nests. In *Proceedings of the 1994 ACM International Conference on Supercomputing*, pages 44–53, 1994.

[23] V.P. Krothapalli, J. Thulasiraman, and M. Giesbrecht. Run-time parallelization of irregular DOACROSS loops. In *Proceedings of Irregular '95*, pages 75–80, 1995.

[24] V. Balasundaram, G. Fox, K. Kennedy, and U. Kremer. An static performance estimator to guide data partitioning decisions. In *Proceedings of the Third ACM SIGPLAN Symposium on Principles and Practices of Parallel Programming*, pages 213–223, April 1991.

[25] David K. Lowenthal and Michael James. Run-time selection of block size in pipelined parallel programs. In *Proceedings of the 2nd Merged IPPS/SPDP*, pages 82–87, April 1999.

[26] David Sundaram-Stukel and Mary K. Vernon. Predictive analysis of a wavefront application using LogGP. In *Proceedings of the Seventh ACM Symposium on Principles and Practice of Parallel Programming*, pages 141–150, May 1999.

[27] Stephanie Coleman and Kathryn S. McKinley. Tile size selection using cache organization and data layout. In *SIGPLAN '95: Conference on Programming Language Design and Implementation*, 1995.

[28] Rob F. Van der Wijngaart, Sekhar R. Sarukkai, and Pankaj Mehra. The effect of interrupts on software pipeline execution on message-passing architectures. In *Proceedings of ACM Int'l. Conference on Supercomputing*, May 1996.

[29] D.J. Palermo, E. Su, J.A. Chandy, and P. Banerjee. Compiler optimizations for distributed memory multicomputers used in the PARADIGM compiler. In *Proceedings of the 23rd International Conference on Parallel Processing*, pages II:1–10, August 1994.

[30] Pete Keleher and Chau-Wen Tseng. Enhancing software DSM for compiler-parallelized applications. In *Proceedings of the 11th International Parallel Processing Symposium*, April 1997.

[31] Honghui Lu, Alan L. Cox, Sandhya Dwarkadas, Ramakrishnan Rajamony, and Willy Zwaenepoel. Compiler and distributed shared memory support for irregular applications. In *Sixth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 48–56, June 1997.

[32] Karthikeyan Balasubramanian and David K. Lowenthal. Efficient support for pipelining in distributed shared memory systems (submitted to Parallel and Distributed Computing Practices). August 1999.

[33] Peter Dinda, Thomas Gross, David O'Hallaron, Edward Segall, James Stichnoth, Jaspal Subhlok, Jon Webb, and Bwolen Yang. The CMU task parallel program suite. Technical Report CMU-CS-94-131, School of Computer Science, Carnegie Mellon University, March 1994.