Run-Time Selection of Block Size in Pipelined Parallel Programs*

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

Abstract

Parallelizing compiler technology has improved in recent years. One area in which compilers have made progress is in handling DOACROSS *loops, where cross-processor data dependencies can inhibit efficient parallelization. In regular* DOACROSS *loops, where dependencies can be determined at compile time, a useful parallelization technique is* pipelining*, where each processor (node) performs its computation in blocks; after each, it sends data to the next processor in the pipeline. The amount of computation before sending a message is called the* block size*; its choice, although difficult for a compiler to make, is critical to the efficiency of the program. Compilers typically use a static estimation of workload, which cannot always produce an effective block size. This paper describes a flexible run-time approach to choosing the block size. Our system takes measurements during the first iteration of the program and then uses the results to build an execution model and choose an appropriate block size which, unlike those chosen by compiler analysis, may be nonuniform. Performance on a network of workstations shows that programs using our run-time analysis outperform those that use static block sizes when the workload is either unbalanced or unanalyzable. On more regular programs, our programs are competitive with their static counterparts.*

1. Introduction

Distributed-memory parallel computing is widely used to speed up scientific applications. In practice, parallel programming has proven to be very difficult, so compilers have been developed to generate parallel programs. Most parallelizing compilers look for parallelism only in loops, because that is where most of the time is spent in typical scientific applications.

One key obstacle to automatic 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. Loops with data dependencies are often referred to as DOACROSS loops; if the dependencies can be precisely determined by the compiler, we call them *regular* DOACROSS loops [7]. These can often be executed efficiently using a technique called *pipelining*, where each 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. Sophisticated compilers such as SUIF [2] often can automatically transform sequential loops with data dependencies to pipelined loops.

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, compilers are not always successful at choosing an effective block size. There are several reasons for this. First, compilers use static estimates of workloads, which can be inexact [8]. Second, compilers generally divide the computation into equal-sized blocks, which may not be flexible enough to accommodate scientific applications with unbalanced workloads. Finally, the best block size

*Supported by NSF CAREER Grant CCR-9733063.

may be unknown until run time if the amount of work depends on input values. Fortunately, most scientific programs are iterative and many exhibit similar characteristics on each iteration. This makes it possible to monitor one iteration of the outermost loop and use the results to guide decisions on future ones.

This paper develops novel run-time analysis, to be used in conjunction with existing compiler analysis that detects pipelining, that finds the best block size for pipelined code. Specifically, our analysis:

- Monitors the first iteration of a pipelined computation, obtaining the time to update each column.
- 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 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 always execute within 10% of ones with the best static block size (and sometimes execute faster).

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.

2. Overview and Programming Model

There has been a significant amount of work on automatic parallelization. This includes, for example, SUIF [2] and PARADIGM [3]. All perform *dependence analysis* [1], which is compile-time analysis that inspects control and data flow to determine when parallel execution of code does not violate its sequential semantics. The two primary types of dependencies are data and control; this paper focuses on the former.

When a compiler detects a true data dependence inside a loop that it cannot remove, loop 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,

```
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 1. Outline of a sequential ADI program.

often referred to as `DOACROSS` loops, are common in practice, and, when dependencies can be inferred at compile time, lend themselves to pipelining.

Consider the Alternate Direction Implicit (ADI) code fragment in Figure 1. The row sweep can be executed without any internal communication if the data (array X) is distributed by rows ({`BLOCK,*`} in HPF [6]). 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 by the compiler.

Because of the cross-node dependence, there are four choices that the compiler can make for the second loop: (1) serialize it, (2) transpose the matrix before and after the loop, allowing the loop to be run in parallel with no communication, (3) schedule the iterations of the column sweep such that the dependencies are not violated, and (4) pipeline the loop, leaving the matrix in a row-wise distribution. The first three ways have drawbacks; this paper discusses ways to efficiently pipeline and so assumes the compiler makes the fourth choice.

Note that we assume that all dependencies are detectable at compile 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 [13, 4, 9].

After the compiler determines the dependencies, it must generate pipelined code. It does this by "strip-mining" [10], code motion, and synchronization insertion. The resulting program is shown in Figure 2.

Once the code has been generated, an appropriate block size must be found. The compiler-generated pipelined code in Figure 2 does not specify the block size; if the compiler alone is doing the complete trans-

```
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 2. 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.

formation, it must choose one. 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. Compiler-only methods suffer from several drawbacks. First, compiler algorithms for static workload estimation are still maturing [8]. Second, compilers pick a single block size, which is restrictive unless the workload is completely uniform. Finally, when the workload depends on values that cannot be determined until run time, the compiler cannot infer the amount of work.

Our programming model is general enough to encompass many important scientific applications, though 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. Second, we support regular `DOACROSS` loops, which allows the compiler to detect opportunities for pipelining. Third, we require that a loop be at least doubly nested so that pipelining can be efficient. Fourth, we support only `BLOCK` data distributions, which means that each node operates on a contiguous set of rows. Finally, we only pipeline in a single dimension, as opposed to in multiple dimensions.

3. Run-Time Analysis

We assume that the compiler transforms `DOACROSS` loops to pipelined loops as in Figure 2. However, instead of choosing the best block size by statically estimating work at compile time, we make use of run-time measurements taken during the first iteration of a pipelined computation. At the end of the first iteration, we determine an effective block size on each node and use it for the rest of the computation. 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 $p-1$.

On the first iteration, each node times each column it updates. We also 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).

Note that our current model does not take interrupt time into account as in [14]; it considers only send and receive overheads, as in [12]. We are investigating the effect of interrupts in our current research.

At the end of the first iteration, each node sends all of its column times to node 0. Thus, 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. We denote $T^k_{i,j}$ as the time to update block j on node i using a block size of k. This includes $f_{send}(k)$ if necessary; so, $T^k_{i,j} = \sum_{l=j\cdot k}^{(j+1)\cdot k-1} t_{i,l} + f_{send}(k)$ 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 $p-1$. In addition, 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.

Our algorithm works in three 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.

The first step is to choose an initial block size; we will consider all block sizes that are a power of two. 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 $S^k_{1,0} = S^k_{0,0} + T^k_{0,0} + f_{net}(k) + f_{recv}(k)$. Node 0 can start updating its second block at time $S^k_{0,1} = S^k_{0,0} + T^k_{0,0}$.

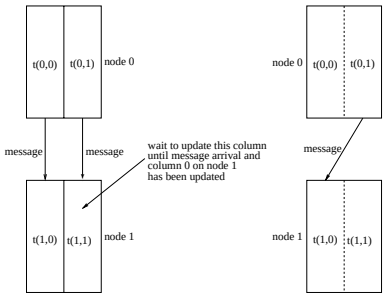

Figure 3. The difference between block sizes of 1 (left) and 2 (right). On the left, node 0 sends a message after updating each column, which means that node 1 must wait to start its second column until it has finished its first column and it has received the necessary data from node 0. On the right, 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.

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: $S^k_{0,0} = 0$ and $S^k_{i,0} = S^k_{i-1,0} + T^k_{i-1,0} + f_{net}(k) + f_{recv}(k)$ and, for $1 \leq j \leq n/k - 1$, $S^k_{i,j} = \max\{S^k_{i-1,j} + T^k_{i-1,j} + f_{net}(k), S^k_{i,j-1} + T^k_{i,j-1}\} + f_{recv}(k)$. 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. 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 3 illustrates the tradeoff (for two columns) between using a block size of 1 and 2.

It it is possible that the most effective block size found above is actually an *ineffective* choice. Consider the case where most of the work is clustered at the right part of the matrix. A choice must be made between a finer granularity, where there is less idle time at the right part of the matrix but excess messages when updating the rest of the matrix, and a coarser granularity, where there are fewer messages sent but significant idle time at the right part of the matrix.

Our analysis handles this using the following heuristic. We start by using the best block size (denoted k) 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 $S^k_{i,j}$; in doing so, we know the time that node i must wait for a message from $i-1$, if any. 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 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.

After we have reduced waiting time by subdividing blocks, we look for opportunities to eliminate 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.

4. Performance

This section reports the performance of three programs. The first two are ADI integration and an implicit hydrodynamics kernel, where an effective block size might be inferred statically by a compiler. The third is an adaptation of an airshed simulation where the workload is not uniform, which represents applica-

tions that need to use run-time information and variable block sizes to achieve good performance.

For each application we developed a program that includes the 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.

Below, we first discuss the overheads incurred by our run-time analysis. Then, we briefly describe the three applications and present the results of runs on 2, 4, and 8 nodes. The sequential times are also reported. All tests were run on a network of 8 Pentium Pros connected by a 100 Mbs Fast Ethernet, using Solaris and the `cc` compiler. 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.

4.1. Overhead of Run-Time Analysis

In general, the overheads of our run-time analysis are small. They are primarily executing the program with fine-grain pipelining on the first iteration, taking measurements (`gethrtime`), estimating completion time for the (possibly nonuniform) block sizes, adding extra code to allow nonuniform block sizes, and choosing an ineffective block size due to timing inaccuracies. Most of these overheads are small or negligible; however, potentially the most serious problem is choosing an ineffective block size. Because we the only user on the machine, this has not happened in our tests.

4.2. ADI

The execution times for two versions of ADI, size 1024, are shown in Table 4. The program that uses our run-time analysis starts with a block size of 1 (fine-grain pipelining) on the first iteration and then switches to a block size of 16 (on 2 nodes) or 8 (on 4 and 8 nodes). The best static block size on 2, 4, and 8 nodes was 16; the overhead of the run-time program was at most 8.1%, on 8 nodes.

4.3. Hydro

Hydro is adapted from kernel number 23 from the Livermore Loops [11]. 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.

Program	Time (2/4/8)	Block (2/4/8)
ADI Run-Time	48.6/25.9/13.3	16/8/8
ADI Best Static	48.3/24.3/12.3	16/16/16
ADI Sequential	95.7	
Hydro Run-Time	57.9/30.1/19.9	32/32/32 (*)
Hydro Best Static	54.7/31.0/20.1	64/64/64
Hydro Sequential	98.9	
Airshed Run-Time	44.8/26.0/16.6	64/1
Airshed Best Static	53.2/36.4/28.9	8/8/8
Airshed Prior Knowl.	42.4/24.6/15.6	64/1
Airshed Sequential	81.6	

Figure 4. Performance in seconds of our three test programs: ADI, Hydro, and Airshed. The first two use a size of 1024×1024; the latter uses $1024 \times 1024 \times 4$. 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. The 64/1 indicates that the block size is 64 on the first 1000 columns and 1 on the last 24. The "prior knowledge" version statically chooses a block size of 64/1.

The execution times for two versions of Hydro are shown in Figure 4. The run-time version first finds a global block size of 32; then, interestingly, it subdivides only the first block (due to the initial wait). This results in a better completion time on four and eight nodes than the best static program, which was with a block size of 64. Because the workload is uniform, this is an application for which a compiler *might* be able to choose an effective block size. Still, our run-time analysis can sometimes produce a *better* one, even though this is a program for which static analysis is possible.

4.4. Airshed Simulation

Our third test program is an airshed simulation, which is adapted from the `tpsuite` from Carnegie Mellon [5]. The program does two transport calculations with a chemistry phase in between. The chemistry phase has an unbalanced workload, and pipelining is required between the transport and chemistry phases. A compiler cannot determine the amount of work in this case, because it does not know which points will be updated. This program is intended to represent less regular programs, where the workload is nonuniform.

The execution times for three versions of the airshed simulation are shown in Table 4. The work was clustered at the right end of the matrix, so an effective composition of blocks is to use larger blocks (size 64) 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. 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 of the run-time program. The prior knowledge test measures the overhead of our system; it implements the effective block size by hand.

5. Summary and Future Work

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 compiler analysis. Furthermore, our system allows the choice of block size to be nonuniform, which allows increased flexibility. Our analysis monitors the program for a single iteration, builds an execution model, 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; in a few cases, they even exceeded the performance of the the programs that used statically determined block sizes. We believe that when combined with a compiler, our system is a viable way to efficiently execute a larger class of pipelined programs than previously possible.

The work described in this paper is a first step; there are many avenues still to be explored. These include recomputing the block size whenever application characteristics change, integrating our run-time analysis more tightly with compile-time analysis, integrating interrupts into our model, investigating better (graph-theoretic) algorithms to choose the block size, allowing more general dependencies, and implementing efficient pipelining in distributed shared memory systems.

6. Acknowledgements

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

References

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

[2] S. P. Amarasinghe, J. M. Anderson, M. S. Lam, and C.-W. Tseng. The SUIF compiler for scalable parallel machines. In *Proc. of the Seventh SIAM Conf. on Parallel Processing for Scientific Computing*, Feb. 1995.

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

[4] D.-K. Chen and P.-C. Yew. On effective execution of nonuniform DOACROSS loops. *IEEE Trans. on Parallel and Distributed Systems*, 7(5):463–476, May 1996.

[5] P. Dinda, T. Gross, D. O'Hallaron, E. Segall, J. Stichnoth, J. Subhlok, J. Webb, and B. Yang. The CMU task parallel program suite. Technical Report CMU-CS-94-131, School of Computer Science, Carnegie Mellon University, Mar. 1994.

[6] HPF–2 scope of activities and motivating applications. Nov. 1994.

[7] A. Hurson, J. T. Lim, K. M. Kavi, and B. Lee. *Parallelization of DOALL and DOACROSS Loops — A Survey*, volume 45, pages 53–103. Academic Press, Ltd.

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

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

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

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

[12] D. Palermo, E. Su, J. 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, Aug. 1994.

[13] J. H. Saltz, R. Mirchandaney, and K. Crowley. Run-time parallelization and scheduling of loops. *IEEE Transactions on Computers*, 40(5):603–612, May 1991.

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