Performance Analysis of Data-Parallel Programs

Vikram S. Adve
Charles Koelbel
John M. Mellor-Crummey

CRPC-TR94405
May 1994

Center for Research on Parallel Computation
Rice University
P.O. Box 1892
Houston, TX 77251-1892

Performance Analysis of Data Parallel Programs*

Vikram S. Adve Charles Koelbel John M. Mellor-Crummey

Center for Research on Parallel Computation, Rice University
Houston, TX 77251-1892
{adve, chk, johnmc}@cs.rice.edu

Abstract

Effective strategies for performance analysis and tuning will be essential for the success of data parallel languages such as High-Performance Fortran (HPF) and Fortran D. Since compilers for these languages insert all communication, they have considerable knowledge about a program's dynamic structure and the relationship between its parallelism and communication. This paper explores how this compiler knowledge can be exploited to support performance evaluation and tuning. First, the compiler itself can use parameterized models to tune the performance of individual program phases; this approach can be effective provided that the compiler can test and handle violations of the model assumptions. Second, by exploiting compiler knowledge and introducing code transformations to improve monitorability, we can collect dynamic performance information that is far more compact than full communication traces, but well suited to the needs of tuning specific communication patterns. Third, we discuss why an understanding of the compiler's capabilities can be important for effective performance tuning. We use several Fortran 77D benchmark kernels to illustrate these points. Finally, through our studies of these benchmarks, we identify the need for several generally applicable compiler optimizations that improve communication and computation overlap.

1 Introduction

Data parallel languages such as High-Performance Fortran (HPF) [9, 15] and Fortran D [11] have attracted considerable attention as promising languages for writing portable parallel programs. These languages support an abstract model of parallel programming in which users annotate a single-threaded program with data layout directives and a sophisticated compiler uses the directives to derive a single-program-multiple-data (SPMD) parallel program. The principal benefit of such abstract parallel languages is that they shield programmers from the intricacies of concurrent programming and managing distributed data.

Since the motivation for exploiting parallel architectures is to solve problems quickly, effective techniques for performance analysis and tuning of data parallel programs will be essential for these languages to achieve widespread acceptance. The fact that compilers for these languages manage both the physical parallelism and the communication presents both challenges and opportunities for performance evaluation. Because performance tools must relate information regarding the compiler-generated SPMD parallel code back to the original single-threaded source in a meaningful way, performance analysis of these languages will be

*Submitted to *Supercomputing '94*, April 1994. This work was supported in part by ARPA contract DABT63-92-C-0038. and NSF Cooperative Agreement Number CCR-9120008. The content of this paper does not necessarily reflect the position or the policy of the Government and no official endorsement should be inferred.

challenging. However, compilers for these languages also have considerable knowledge about a program's dynamic structure and this knowledge presents a unique opportunity for providing compiler assistance in the performance tuning process.

This paper explores how a compiler for a data parallel language can play an important role in supporting performance evaluation and tuning. First, whenever possible, performance problems that can be ameliorated automatically by the compiler (using a combination of static analysis and runtime measurements) should be identified and treated as such, so that user involvement in tuning these parts of a program can be minimized. However, support for such *self-tuning* necessarily relies on a simplified model of program and system behavior which may be violated in practice. We present a performance model for automatically tuning pipelined computations, and we use the results of preliminary experiments with the model to identify a few important steps that we believe should be followed when developing a self-tuning component of an optimizing compiler for a data-parallel language.

When compiler self-tuning is insufficient or impractical, a programmer must first understand the causes of the performance problem and then modify the source code to improve performance. Measurements of dynamic (i.e., runtime) performance information are important for understanding performance problems as well as for compiler self-tuning. We use examples to show that we can exploit the compiler's knowledge, both of program structure and of what information is important for understanding and tuning specific communication patterns, to reduce dynamic performance measurement to collection of summary statistics about related communication events on each processor. In addition, we describe some useful code transformations whose sole purpose is to provide better performance information.

Once a potential performance optimization has been identified by the programmer, this optimization must be "expressed" in the data parallel source code. We compare unoptimized and optimized versions of two simple numerical kernels to argue that, in order to do this successfully in the abstract programming model provided by the language, a user must have some understanding of a compiler's optimization and transformation capabilities. Providing such an understanding can help ensure, first, that a programmer's expectations reflect the compiler's true abilities, and second, that a programmer has sufficient understanding of the effects (or lack thereof) of program changes to express specific optimizations in a form that can be understood by the compiler. An important caveat is that such an understanding should not come at the expense of the abstract model provided by the language.

Throughout the paper, we illustrate our points with references to benchmark kernels written in Fortran 77D that have been compiled using the Rice University Fortran 77D compiler. Although our insights into the compiler's role in performance analysis and tuning apply most directly to the Rice Fortran 77D compiler, the similarity of the Fortran D and HPF language models leads us to believe that these results will be applicable to HPF compilers as well.

The structure of the paper is as follows. Section 2 discusses our experiences with the self-tuning model for pipelined computations. Section 3 discusses compiler support for more effective performance measurement.

Section 4 illustrates the need for compiler-aware tuning, i.e., providing an understanding of the compiler's abilities to the programmer. Section 5 discusses a few compiler enhancements suggested by the above performance studies. Section 6 briefly places this work in the context of previous research with related goals. Section 7 summarizes our conclusions and our plans for future work.

2 Self-Tuning Compiler

A sophisticated compiler for a data parallel language provides a number of opportunities for self-tuning, i.e., alleviating performance bottlenecks by tailoring the executable using a combination of static and dynamic information, with little or no user intervention. We believe it is important to identify such opportunities and minimize user involvement in tuning parts of a program where self-tuning strategies can be applied effectively. Preliminary experiences with the development and use of a performance model for tuning pipelined computations, however, reveal some interesting considerations that arise when putting this approach into practice. In the following, we present a pipeline performance model and then discuss our experiences using the model to tune a pipelined computation in a program. From these experiences we identify a few important steps that we believe should be followed when designing a self-tuning component of an optimizing compiler for data parallel languages.

2.1 A Model for Tuning Pipelined Computations

Pipelining is a strategy for realizing parallelism in computations that are only partially parallel. Pipelining is useful, for example, for parallel execution of a nest of loops in a data parallel program when each iteration of the innermost loop requires results from one or more previous iterations of each of the loops in the nest. If the data is partitioned among the processors, pipelined parallelism can be realized by having each processor compute values for a subset of its data and send partial results to waiting processors before continuing further. The amount of computation for each message sent is referred to as the granularity of the pipeline.

Tuning pipelined computations requires adjusting the granularity to balance the loss of parallelism in the initial and final phases of a pipeline when only a few processors are busy, against the communication overhead in the middle interval which is fully parallel. The length of the initial and final phases is directly proportional to the pipeline granularity, whereas the communication overhead in the middle interval is inversely proportional to the pipeline granularity. In a study of a pipelined Gauss-Seidel relaxation, Rogers and Pingali [21] recognized this trade-off but did not present a model for choosing the pipeline granularity. Hiranandani et. al.[12] present a model for determining optimal granularity in terms of a loop blocking factor based on the simplifying assumptions that communication cost is a constant independent of block size, and that no part of communication latency is overlapped with computation during the execution of the pipeline. Ramanujam and Sadayappan [18, 19] present a related model for choosing the optimal tile size for a wavefront computation; they too assume that communication latency is not overlapped with computation.

Our measurements of ERLEBACHER indicate that the influence of block sizes on communication time is signficant and therefore we incorporate block size in our model. Furthermore, as we motivate below, our model accounts for non-overlapping communication and computation at the leading edge of the pipeline and perfect overlap in the rest of the pipeline.

In a perfect pipeline, every processor performs identical computation, and communication time is identical between every pair of adjacent processors. Under these conditions, as long as the processing time (i.e., computation time plus send and receive overhead) in each pipeline stage is longer than the wire latency for values to propagate between processors, a processor will never block for an incoming message after the first (see Figure 1). In practice, imbalance or variability in computation time, communication overhead, or communication latency can cause pipeline delays. However, such delays are often data or timing dependent, so we choose to ignore their effects in our model. In section 2.3, we discuss a uniform approach for detecting and handling violations of model assumptions.

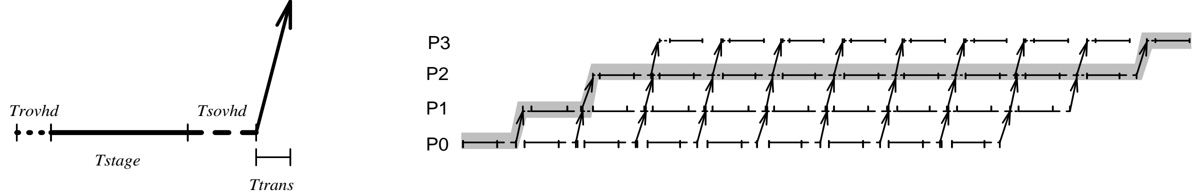
Here, we present a pipeline model applicable to one-dimensional pipelines in which each processor sends to only one processor and receives from only one other processor. As the basis for our model, consider an $N_1 \times N_2$ matrix partitioned block-wise in the first dimension among P processors. The pipeline granularity is measured in terms of the block size B, $1 \leq B \leq N_2$, which is the number of columns to compute in a pipeline stage. The total number of stages in the pipeline is N_2/B. The total number of bytes of data transmitted per stage is $m_0 B$. Using our model, we determine the optimal value of B, denoted B_{opt}, for which the total execution time of the pipeline is minimized. The total pipeline execution time can be computed as follows (where processors are numbered $0 \ldots P-1$):

$$\begin{aligned} T_{pipe} \;=\; & \text{time until processor } P-2 \text{ starts computing} + \\ & \text{time for processor } P-2 \text{ to finish } N_2/B \text{ pipeline stages} + \\ & \text{time for processor } P-1 \text{ to receive its last message and complete its last pipeline stage} \end{aligned}$$

We express these quantities as functions of B and differentiate the expression to find the value B_{opt} where $T_{pipe}(B)$ is minimized. Let $T_{stage}(B)$, $T_{sovhd}(B)$, $T_{rovhd}(B)$ and $T_{comm}(B)$ respectively denote the computation time per stage, the overhead incurred by a processor to send or receive $m_0 B$ bytes of data, and the total time to communicate $m_0 B$ bytes of data from the start of the send operation until the receive operation completes.[1] As Figure 1 shows, the message wire latency can be ignored in the second term because we assume it is completely overlapped with processing (i.e., computation and send/receive overhead). Thus,

$$\begin{aligned} T_{pipe}(B) \;=\; & (P-2)(T_{stage}(B) + T_{comm}(B)) + \\ & T_{stage} + (\frac{N_2}{B} - 1)(T_{sovhd} + T_{rovhd} + T_{stage}) + \\ & T_{comm}(B) + T_{stage}(B) \end{aligned} \tag{1}$$

[1] In the model presented, we assume there is no buffering or unbuffering of data other than by the communication routines themselves. Two additional terms can be added to the model to account for separate buffering and unbuffering costs within the program, but we omit them here in the interest of brevity.



Computation and communication times

Pipeline structure

Figure 1 Pipeline cost model

For each $x \in \{comm, sovhd, rovhd\}$, we model T_x as a linear function of B: $T_x(B) = C_{l,x}B + C_{f,x}$. We model $T_{stage} = C_{l,stage}(\frac{BN_1}{P}) + C_{f,stage}$, where $C_{l,stage}$ denotes the cost of the computation per element. A key aspect of the model is that $T_{comm}(B)$, $T_{sovhd}(B)$ and $T_{rovhd}(B)$ can be treated as fixed functions for a particular system, and in particular should not depend on variations in program input size or number of processors. Thus, only $C_{l,stage}$ and $C_{f,stage}$ have to be estimated for a particular program. Clearly, $C_{l,stage}$ and $C_{f,stage}$ could be estimated from two executions of the program that use different block sizes. In section 3, we describe how we could estimate these parameters from a single execution. When applying the model, we assume that these constants are independent of B and also of the input size and the number of processors. In practice, however, these parameters can vary due to cache effects, conditionals and even floating-point operation timings. We discuss how to identify and handle violations of this assumption in section 2.3.

Solving equation (1) to obtain B_{opt}, the value of B that minimizes T_{pipe} (B), results in: [2]

$$B_{opt} = \sqrt{\frac{N_2(C_{f,stage} + C_{f,rovhd} + C_{f,sovhd})}{(P-1)(\frac{C_{l,stage}N_1}{P} + C_{l,comm}) - C_{l,rovhd} - C_{l,sovhd}}} \tag{2}$$

Note that B_{opt} increases roughly as $1/\sqrt{C_{l,comm}}$, meaning that as bandwidth increases, larger block sizes are more attractive, and B_{opt} increases as the fixed overhead for sends or receives increases, agreeing with intuition that larger block sizes are better able to amortize fixed communication overhead. Also note that the model is not restricted to two-dimensional matrix computations: N_2 merely denotes the total number of units of work that must be completed per processor, where a unit of work is the smallest available granularity of a pipeline stage. Thus, for an $N_x \times N_y \times N_z$ matrix block-distributed in the Z dimension, the model can be directly applied with $N_2 = N_x \times N_y$.

2.2 Applying the pipeline model to Erlebacher

[2] When differentiating (1), we make two simplifications that must be adjusted for: (a) we assume B is a continuous variable, and (b) we ignore the discontinuities in T_{pipe} (B) that arise because T_{comm}, T_{sovhd} and T_{rovhd} typically are piecewise linear functions of B. In practice, we must find the local minimum for each interval of continuity and then compare T_{pipe} (B) at all adjacent integer values of these local minima to choose the globally optimal B_{opt}. This approach is valid because T_{pipe} (B) is convex in each interval of continuity. The approach should also be efficient if there are only a small number of discontinuities (e.g., the iPSC/860 has a single one at $m_0B = 100$ [2]).

```
subroutine tridvpk(a,b,c,d,e,tot)
 real a(64),b(64),c(64),d(64),e(64),tot(64,64)
 common /dvars/ ..., f(64,64,64)
 parameter (n_proc = 8)
 decomposition d(64,64,64)
 align f with d
 distribute d(:,:,block)
 ...
 ! *** forward substitution ***
 do 20 k=2,64-1
  do 20 j=1,64
   do 20 i=1,64
    f(i,j,k)=(f(i,j,k)-a(k)*f(i,j,k-1))*b(k)
20 continue
 ...
 end
```

(a) Fortran D code

```
subroutine tridvpk(a, b, c, d, e, tot)
 real a(64), b(64), c(64), d(64), e(64)
 common /dvars/ ..., f(64,64,0:9)
  ...
  off_0 = ...
  ...
  !*** forward substitution ***
  do j = 1, 64
   do i_ = 1, 64, 8
    i_up = i_ + 7
    if (my_p .gt. 0) then
     call crecv(114, f(i_, j, 0), 8 * 4)
    endif
    do k = lb_1, ub_2
     do i = i_, i_up
      k_glo = k + off_0
      f(i,j,k)=(f(i,j,k)-a(k_glo)*f(i,j,k-1))*b(k_glo)
     enddo
    enddo
    if (my_p .lt. 7) then
     call csend(114,f(i_,j,8),8*4,GRAY(my_p+1),my_pid)
    endif
   enddo
  enddo
  ...
end
```

(b) Compiler-generated pipelined code (block size $B = 8$)

Figure 2 Forward substitution phase in Erlebacher

We applied the pipeline model to tune the performance of pipelined phases in ERLEBACHER, a 13 procedure, 800 line benchmark program written by Thomas Eidson at the Institute for Computer Applications in Science and Engineering (ICASE). ERLEBACHER performs 3D tridiagonal solves using Alternating-Direction-Implicit (ADI) integration. The tridiagonal solve in each dimension consists of a computation phase followed by a forward and a backward substitution phase. Each of the substitution phases results in a computation wavefront across that dimension of the data. Figure 2 shows the Fortran D source code as well as compiler output for the forward substitution phase in the Z dimension. (Under the data distributions shown, the X and Y dimensional solves compute exclusively with local values, except for a single global sum reduction in each phase.) The compiler employs coarse-grain pipelining (with a default block size of $B = 8$) to parallelize both the the forward and backward (not shown) substitution phases; each pipeline stage consists of 8 iterations of the `i` loop.

From a single execution of the program on 8 processors with an input matrix of $64 \times 64 \times 64$, we measured the model coefficients described in section 2.1. [3] The function T_{pipe} (B) parameterized with the measured coefficients is shown in Figure 3 for $P = 8$ and $P = 32$. These curves show that substantial improvement

[3] In practice, we would not have to measure $C_{f,sovhd}$, $C_{l,sovhd}$, $C_{f,rovhd}$ and $C_{l,rovhd}$ because these are fixed system values for a given message size. At present, we do not have measurements of these values available. Instead we measured T_{sovhd} and T_{rovhd} and assumed $C_{f,sovhd} \approx T_{sovhd}$ and $C_{f,rovhd} \approx T_{rovhd}$, i.e., we assumed the components proportional to block size T_{sovhd} and T_{rovhd} for $B = 8$ ($m_0 B = 32$) are small. Values of $C_{l,comm}$ and $C_{f,comm}$ were taken from Bokhari [2]. We also assumed *a priori* that $C_{f,stage} = 0$; see the discussion in section 2.3.

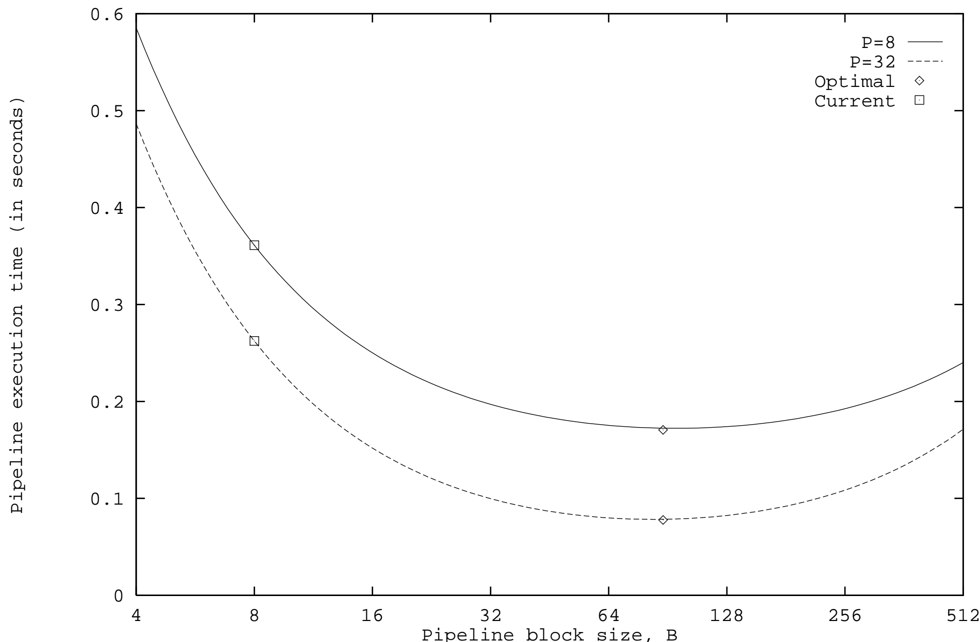


Figure 3 Predicted impact of pipeline granularity on execution time in ERLEBACHER

in execution time is possible by adjusting the block size. In fact, a space-time diagram of the pipeline execution viewed using ParaGraph [8] showed that, at the default block size of 8, pipeline stages are very small (resulting in frequent communication) and thus the startup phase is small as well. The small start-up cost indicates that it would be profitable to increase the block size to reduce the overall communication overhead for the pipeline. The model predicts the optimal block size to be $B_{opt} = 88.0$, as shown in Figure 3. Increasing the block-size to B=64 yielded substantial improvement in performance. On 8 processors, the measured execution time of each pipeline improves by an average of 34.9% while that of the overall program improves by 15%. (We did not use the actual value of $B_{opt} = 88$ because any value other than a multiple or submultiple of 64 would have required collapsing and linearizing the i and j loops, an arduous task by hand.)

2.3 A strategy for handling violations of model assumptions

In an actual pipeline execution, some of the simplifying assumptions in the model may be violated. For example, in the forward pipeline of ERLEBACHER, our *a priori* assumption that $C_{f,stage} = 0$ proved inaccurate, leading to an inaccurate estimate for $C_{l,stage}$. With this assumption, we estimated $C_{l,stage} = 3.92\ \mu s.$ per element for $B = 8$ on 8 processors; under the same assumptions with $B = 64$, $C_{l,stage}$ appeared to be 1.68 $\mu s.$ per element. Actually, $C_{f,stage} > 0$ in part because the accesses to `a(k)` and `b(k)` on the right-hand-side of the recurrence each miss only once every B iterations (see Figure 2b). This implies that

the value of B_{opt} computed from (2) using $C_{l,stage} = 3.92\ \mu s.$ is likely too low.

As a second example, in our experiments on an iPSC/860 with ERLEBACHER and SOR, we observed that interrupts caused by arriving messages introduce bubbles in an otherwise balanced pipelined computation. Both computations are forward pipelines. In each pipeline, processor 0 does not do any receive operations, so it sends messages at a higher rate than the other processors. Processor 1 is interrupted at a higher frequency which slows its computation, thus slowing the rest of the pipeline to the same degree. After the last message from processor 0 has been received by processor 1, processor 1 sees no more interrupts (though it still executes receive operations) and the entire pipeline speeds up to its "natural frequency" determined by the values of T_{rovhd}, T_{stage} and T_{sovhd} at processors $2 \dots P-2$. This effect is exacerbated with smaller block sizes which cause more frequent communication, implying again that it would be worthwhile to use a larger block size than predicted by the model.

In the above two cases, given that we have access to the compiler under development, we can attempt to account for these effects within the compiled code. The systematic cache effects described above only require a simple additional term ($C_{f,stage}$) in the model. The high rate of interrupts by processor 0 can be slowed down artificially by having processor 1 occasionally send back synchronization messages to processor 0. Alternatively, we can artificially overestimate B_{opt} by using values of T_{stage} measured near the end of the pipeline on each processor. More generally, however, system artifacts that cause less systematic variations in T_{stage}, e.g., external system interrupts, less systematic cache effects or floating-point operation timings, cannot be modeled explicitly or adjusted for by improving the generated code. Furthermore, modifying the compiler may itself not be an available option. Thus, a more general approach to handling violations of the tuning model is essential.

Since the assumptions about a parameter like T_{stage} are known in the compiler, the compiler could easily add instrumentation to test these assumptions. For example, the compiler could measure the variability in the measured values of T_{stage}, in addition to the mean. The compiler could also test if the mean values of T_{stage} do not fit the assumed linear function by comparing the new mean value obtained after recompiling with the mean value measured in the first execution. When such variations are detected, at a minimum the compiler/performance tool should convey to the user that an assumption in the model has been violated and that the block size chosen for the pipeline may be suboptimal. The user could also be told how much the observed pipeline execution time after recompiling differed from the predicted value, viz. $T_{pipe}(B_{opt})$, and perhaps also given a hint (inferred by the compiler from the observed discrepancy in T_{stage}) such as "further increasing the block size might reduce pipeline execution time". Finally, the compiler should provide a command-line option or directive to control the pipeline block size manually in case the user wants to try to improve the block size by trial and error.

More generally, the above example suggests a general approach to handling violations of assumptions made by a self-tuning performance model. The approach consists of carrying out the following steps for each self-tuning component (i.e., for each self-tuning model):

1. Identify the significant model assumptions, and the qualitative impact on performance when each of the assumptions is violated.

2. Include instrumentation to test the validity of each assumption; this may be in addition to the instrumentation required to apply the model.

3. If an assumption is observed to fail during the self-tuning process, warn the user of the failure and its potential impact on performance.

4. If requested and feasible, suggest "which way" the relevant tuning parameter (e.g., the block size in the pipeline model) might be adjusted to overcome the shortfall.

5. Provide the necessary hooks to allow a sophisticated user to manually tune the relevant portion of the program by trial-and-error (i.e., by repeated recompilation and execution).

These steps provide a uniform approach for handling violations of other assumptions in the pipeline model as well. In particular, simple summary statistics (mean and variance) of the send and receive overheads on each processor enable comparison of observed values of $C_{f,sovhd}$, $C_{l,sovhd}$, $C_{f,rovhd}$ and $C_{l,rovhd}$ with the assumed system-specific values, and to check for variability in these overhead terms. (Recall that obtaining two data points from a single run enable precise estimation of both the fixed and linear coefficients for each equation.) Accurately synchronized clocks on the different processors would allow us to compare $C_{l,comm}$ as well. Discrepancies or variability in these factors could arise from contention for network links or processor-network interfaces, architectural details that are not reflected in the model, or interference from other concurrently executing programs. Higher observed latencies ($C_{l,comm}$) would suggest using smaller block sizes to reduce the pipeline startup cost, while higher processing overheads for communication would suggest using larger block sizes to better amortize fixed overhead costs. As before, a user should have the option to tune the block size by trial and error.

A key aspect of the process of tuning pipeline performance discussed above is that the only runtime parameter values that must be measured are simple summary statistics about per-stage computation times and send and receive overheads. In section 3, we discuss how the compiler's knowledge of the pipeline structure can be exploited to increase the efficiency of pipeline instrumentation with little loss of information.

3 Compiling for observability

While compile-time estimates of parameters such as pipeline block size using only static information may lead to good performance, the impact of run-time effects as cache utilization can be difficult to predict and thus dynamic information can be critical for effective performance tuning. For programs written in data parallel languages such as Fortran D, however, the compiler's knowledge of a program's structure can be exploited to direct collection of dynamic information. Here, we describe two ways to exploit compiler knowledge: first,

to maximize what can be learned from a single execution, and second, to dramatically reduce the cost of gathering dynamic information for performance tuning.

In section 2.1, we described a model for computing the optimal block size for a pipeline that includes a number of linear terms dependent on the block size. Each term includes both a linear coefficient and a contant coefficient. A single execution of the program using a particular block size is insufficient to measure both of the coefficients for any term. However, there is no reason why we are limited to only a single block size in a pipelined execution. The compiler could split the iteration space of a pipelined loop nest into two halves and arrange to execute each half of the pipelined computation using a different block size. Separate measurements of the different halves of the pipelined computation will provide two data points for each of the terms enabling determination of both of each term's coefficients from a single execution. The use of two different block sizes would not disrupt the execution other than to introduce some additional waiting due to differences in the appropriate processor skew for the different pipeline granularities. A similar strategy might be useful for evaluating the performance implications of multiple alternative data distributions with a single execution.

The most common strategy for collecting dynamic information about message-passing programs involves tracing each communication event. Such a tracing methodology is embodied in packages such as the PICL communication primitives [7]. However, for understanding and tuning the performance of programs written in data parallel languages, this level of detail often appears unnecessary. For example, to tune a computational loop that exploits coarse-grain pipelining (such as the pipelined forward solve in ERLEBACHER), we are interested in collecting dynamic information to enable computation of the optimal block size based on the model we described in section 2.1. The dynamic information needed for this purpose consists only of estimates for the various parameters of the pipeline model, information that will enable us to test that none of the model assumptions are being violated, and statistics that describe the overall performance characteristics of the pipelined computation.

Examining these issues in a bit more detail, communication events in pipelined computations can be grouped into two distinct classes: those in the leading edge of the pipeline (the first pipeline stage on each processor), and those in the steady (the remaining stages on each processor). To compute the coefficients for T_{sovhd} and T_{rovhd} for the pipeline model, we want to collect information about pipeline stages in the steady state. We can collect statistics about the steady state of the pipeline on each processor merely by excluding communication events the first trip through the pipelined loop nest on each processor. This can be accomplished by having the compiler peel the first iteration of the loop before inserting communication instrumentation in the remaining iterations, which causes almost no perturbation to the execution of the program. Rather than collecting full traces of all communication events in the steady state of a pipelined computation, collecting summary statistics on each processor about overhead observed for each send and receive in the pipeline stage is effective. For most pipelined computations, the length of each pipeline stage is very regular and there all sends or receives will have similar characteristics. MIN and MEAN times for

FORWARD PIPELINE

PROC	FIRST RECV	SUBSEQUENT RECEIVES MEAN	C.V.	MIN	MAX	FIRST SEND	SUBSEQUENT SENDS MEAN	C.V.	MIN	MAX
0	—	—	—	—	—	0.204	0.162	0.009	0.150	0.278
1	0.241	0.176	0.045	0.144	0.323	0.271	0.186	0.047	0.152	0.327
2	13.430	0.254	0.158	0.148	0.522	0.171	0.182	0.036	0.156	0.329
3	3.844	0.302	0.059	0.153	0.539	0.160	0.168	0.013	0.157	0.281
4	15.233	0.316	0.047	0.155	0.564	0.166	0.164	0.010	0.152	0.276
5	4.943	0.315	0.050	0.185	0.568	0.163	0.164	0.007	0.153	0.257
6	15.512	0.309	0.057	0.165	0.573	0.165	0.165	0.008	0.155	0.279
7	22.795	0.637	0.017	0.464	0.916	—	—	—	—	—

Table 1 ERLEBACHER pipeline SEND and RECEIVE statistics for 64x64x64 data size distributed in Z dimension among 8 processors. (All times in milliseconds.)

PROC	FIRST RECV	SUBSEQUENT RECEIVES MEAN	C.V.	MIN	MAX	FIRST SEND	SUBSEQUENT SENDS MEAN	C.V.	MIN	MAX
0	—	—	—	—	—	0.223	0.206	0.008	0.189	0.334
1	0.248	0.177	0.031	0.158	0.377	0.205	0.220	0.025	0.191	0.402
2	0.216	0.177	0.025	0.159	0.301	0.217	0.220	0.021	0.197	0.346
3	3.625	0.280	0.019	0.164	0.514	0.197	0.210	0.010	0.194	0.328
16	34.017	0.268	0.125	0.162	1.736	0.204	0.211	0.013	0.189	0.324
17	35.840	0.294	0.093	0.196	1.924	0.207	0.207	0.009	0.190	0.326
18	37.794	0.263	0.159	0.162	2.022	0.210	0.211	0.015	0.194	0.401
19	51.324	0.284	0.112	0.162	2.081	0.212	0.209	0.009	0.190	0.335
28	75.046	0.301	0.219	0.197	3.120	0.198	0.206	0.012	0.191	0.355
29	65.634	0.302	0.241	0.191	3.266	0.202	0.206	0.008	0.188	0.327
30	67.519	0.297	0.280	0.167	3.467	0.203	0.206	0.009	0.190	0.331
31	97.102	0.714	0.051	0.589	3.862	—	—	—	—	—

Table 2 SOR pipeline SEND and RECEIVE statistics for selected processors, with 4096×4096 data size distributed in X dimension among 32 processors. (All times in milliseconds.)

communication operations provide the information needed to calculate aggressive or conservative model parameters. The coefficients of variation are useful for verifying the uniformity of activity across the processors and pipeline stages.

We explored using this data collection strategy in the context of the pipelines in ERLEBACHER. Table 1 shows summary statistics synthesized from dynamic traces of the forward 512-stage pipeline of ERLEBACHER. The column with the heading FIRST shows the measured time of SEND (RECEIVE) events in the first trip through the pipelined loop. The remaining columns show the MEAN, COEFFICIENT OF VARIATION, MIN, and MAX timings of communication events in subsequent iterations of the pipelined loop. The table shows that there is significant variation in the receive times in the leading edge of the pipeline. This variation is because processors are not synchronized when entering the pipeline. In subsequent stages of the pipeline, however, the delays due to receive operations on each processor are quite uniform, as shown by the similarity among the mean values for the different processors and the low coefficient of variation for each individual processor.

Table 2 shows the corresponding statistics for another pipelined application kernel, SOR, executing on 32 processors (data for representative processors were selected for the table). Again, the data show significant variation in the receive times in the first pipeline stage, but uniform send and receive costs in the subsequent stages. Thus, in both these cases, it would be effective to record summary statistics instead of the individual send and receive costs of the pipeline stages, thus yielding substantial savings in dynamic trace volume for the pipelined phase.

Here, we have described two novel ways of exploiting compiler knowledge to help harness dynamic information for performance analysis and tuning. While our running example in this section focuses on pipelined computations, the strategy of dynamic trace reduction through collection of only summary statistics is equally applicable to loosely-synchronous computations, e.g. JACOBI and RED-BLACK SOR, as well. We are confident that other opportunities for applying compiler knowledge will become apparent as we gain more experience in this area.

4 Understanding compiler capabilities for performance tuning

Even in an abstract data-parallel programming language such as Fortran D, it is clearly important for a programmer to understand the performance characteristics of the underlying parallel architecture in order to tune the performance of a parallel program effectively. In fact, when working with such a language, it can be equally important for a user to be aware of the broad capabilities of the compiler. In this section we use examples to illustrate the need for *compiler-aware tuning* and briefly discuss how a user might be provided the requisite knowledge without requiring any understanding of compilation techniques and without unduly corrupting the abstract programming model provided by the language.

DGEFA is a LINPACK subroutine and a principal computational kernel in the LINPACKD benchmark developed by Dongarra et al [5]. It performs LU decomposition using Gaussian elimination with partial pivoting. The algorithm, shown in Figure 4a, consists of selecting the largest element as the pivot for a particular column k, computing the row reduction factors for step k by dividing the column elements by the pivot value, and then reducing every column on the right using these factors. For linear algebra codes such as DGEFA, a cyclic or block-cyclic distribution helps maintain load-balance. In compiling DGEFA, the Fortran D compiler inserts two broadcasts for each step, one to broadcast the pivot element and location, and another to broadcast the row reduction factors. The code generated by the compiler is shown in Figure 4b.

The principal source of performance loss in this program is that other processors have to wait for the pivot processor to compute and broadcast the row-reduction factors for an elimination step. As Hiranandani et al. [12] have shown, an effective hand optimization is to overlap the processing and broadcast of factors for column $k+1$ with the row-reduction operations that use the factors for column k. This can be achieved by computing and broadcasting the factors for column $k+1$ as soon as factors for column k are received, as shown by the hand-optimized code in Figure 5a. We traced the communication operations in the two versions

```
program DGEFA
  double precision a(NMAX,NMAX)
  parameter (n_proc = 16)
C decomposition d(1024, 1024)
C align a with d
C distribute d(:, cyclic)

 do k = 1, NMAX-1
  find pivot(k) = max of a(k,k)...a(NMAX,k)
  let pl = row index of pivot

  if ( pivot(k) .ne. 0.0d0 ) then
   exchange a(k,k) and a(pl,k)
   !*** row reduction factors of col k
   do i = k+1, NMAX
    a(i,k) = a(i,k) / a(k,k)
   enddo
   do j = k+1, NMAX
    !*** reduction of col j with factors of col k
    exchange a(k,j) and a(pl,j)
    do i = k+1, NMAX
     a(i,j) = a(i,j) + p(k,j) * a(i,k)
    enddo
   enddo
  endif
 enddo
end
```

(a) Fortran D code.

```
program DGEFA
 double precision a(NMAX,NMAX/NPROC)

 do k = 1, NMAX-1
  pivot_proc = MOD(k-1,n_proc)
  if (my_p .eq. pivot_proc) then
   find pivot(k) = max of a(k,k)...a(NMAX,k)
   let pl = row index of pivot
   broadcast [pivot(k),pl]
  else
   receive [pivot(k),pl]
  endif

  if (pivot(k) .ne. 0.0d0) then
   if (my_p .eq. pivot_proc) then
    Compute reduction factors for col k
    broadcast [a(k+1,k)...a(NMAX,k)]
   else
    receive [a(k+1,k)...a(NMAX,k)]
   endif
   L1 = k/n_proc +1
   if (my_p .lt. pivot_proc) L1 = L1 + 1
   do j = L1, NMAX/n_proc
    Compute row reduction for col j with factors of col k
   enddo
  endif
 enddo
end
```

(b) Compiler-generated code.

Figure 4 Gaussian elimination with serial processing of pivot column

of the program using the Pablo instrumentation library [20]. Table 6 shows the distribution of idle time by cause in the two versions of the program for a 1024×1024 matrix on 32 processors. The measurements show that the fraction of total execution time spent waiting for the broadcast values is reduced from 39% to 9%, while total execution time is reduced by 24%. Thus, computing and broadcasting the column of factors for a pivot can largely be overlapped with the elimination step in DGEFA.

Unfortunately, it is difficult for the compiler to deduce and implement this optimization from the original Fortran D source of Figure 4a. The compiler would have to discern the potential gain from moving the broadcast operation from iteration $k+1$ to iteration k, detect that the pivot computation of iteration $k+1$ can indeed be performed before the third do loop of iteration k, and rearrange the body of the k loop accordingly (including splitting off the first iteration and the leftover work of the last iteration). Such aggressive transformations are outside the scope of the Fortran D compiler and, to our knowledge, any other data parallel compiler as well. Thus, once the source of the performance loss has been found and understood, the user must attempt to convey the optimization to the compiler by modifying the Fortran D source.

To reduce the trial-and-error involved in rewriting the code to express this optimization, an understanding of the capabilities of the compiler is extremely useful. For example, from the "natural" expression of this optimization in Fortran D, shown in Figure 5b, it would still require more sophisticated analysis than presently available to determine that the pivot columns and pivot location computed in loops 1 and 3 are

```
program DGEFA
 double precision a(NMAX, NMAX/n_proc)

 pivot_proc = 0
 if (my_p .eq. pivot_proc) then
  find pivot(1) = max of a(1,1)...a(NMAX,1)
  let pl = row index of pivot(1)
  broadcast [pivot(1),pl]
  if (pivot(1) .ne. 0.0d0) then
   Compute row reduction for col 1 with factors of col 1
   broadcast [pl, pivot(1), a(2,1)...a(NMAX,1)]
  endif
 endif

 do k = 2, NMAX - 1
  pivot_proc = MOD(k,n_proc)
  last_pivot_proc = MOD(k-1,n_proc)
  if (my_p .ne. last_pivot_proc) then
   receive [pl, pivot(k-1), a(k,k-1)...a(NMAX,k-1)]
  endif

  if (my_p .eq. pivot_proc) then
   Compute row reduction for col k with factors of col k-1
   find pivot(k) = max of a(k,k)...a(NMAX,k)
   let pl2 = row index of pivot(k)
   Compute reduction factors for col k
   broadcast [pl2, pivot(k), a(k+1,k)...a(NMAX,k)]
  endif

  if (pivot(k-1) .ne. 0.0d0) then
   L1 = k/n_proc + 1
   if (my_p .lt. pivot_proc) L1 = L1 + 1

   do j = L1, NMAX/n_proc
    Compute row reduction for col j with factors of col k-1
   enddo
  endif
 enddo
 Compute reduction for col NMAX with factors of col NMAX-1
end
```

(a) Hand-optimization to overlap computation and broadcasts of reduction factors with row reductions of previous iteration.

```
program DGEFA
  double precision a(NMAX, NMAX)
C decomposition d(1024, 1024)
C align a with d
C distribute d(:, cyclic)

 find pivot(1) = max of a(1,1)...a(NMAX,1)
 let pl2 = row index of pivot
 broadcast [pivot(1),pl2]
 if (pivot(1) .ne. 0.0d0) then
  !*** Loop 1:
  Compute row reduction for col 1 with factors of col 1
 endif

 do k = 2, NMAX - 1
  pl = pl2

  !*** Loop 2:
  Compute row reduction for col k with factors of col k-1
  find pivot(k) = max of a(k,k)...a(NMAX,k)
  let pl2 = row index of pivot(k)

  !*** Loop 3:
  Compute reduction factors for col k

  if (pivot(k-1) .ne. 0.0d0) then
   do j = k+1, NMAX
    !*** Loop 4:
    Compute row reduction for col j with factors of col k-1
   enddo
  endif
 enddo
 do row elimination for col NMAX with pivot(NMAX-1) ---
end
```

(b) Expressing the optimization in Fortran D.

Figure 5 Minimizing serialization in Gaussian elimination.

required on the following iteration of the k loop by loops 2 and 4, and to place the broadcast receive and send operations appropriately (above Loop 2 and after Loop 3 respectively). An alternative approach would be to avoid the sequential bottleneck of the computation and broadcast of the pivot column by distributing the array in row-cyclic instead of column-cyclic fashion. Unlike the above optimized version, this now requires extra communication in the form of a global reduction to compute the pivot value and location and a broadcast of the pivot row.

Another example of an optimization that requires somewhat sophisticated code-reordering transformations arises in non-pipelined stencil computations such as Red-Black SOR or Jacobi. In a column-block distribution of Red-Black SOR for example, each processor must receive data from its left and right

	TOTAL (sec)	BUSY	IDLE		
			Total	Send	Recv
Unopt	25.821	60.99%	39.01%	0.92%	32.87%
Opt	20.838	89.55%	10.45%	1.29%	9.16%

Figure 6 Improvement with overlapping broadcasts in DGEFA: 1024x1024,column cyclic data partition, 32 processors.

neighbors for computing the new values of its own boundary column. However, the non-boundary iterations in the body of each loop do not require non-local values and also do not have to wait for the boundary columns *in the same loop* to be computed (there are no loop-carried dependences in each of the red and black loops). This optimization has been implemented for the restricted case of FORALL statements in the Kali compiler [16] and previously suggested in [10]. The reduction in waiting time for message receives in the case of RED-BLACK SOR is small but significant since it reduces overhead idle time by nearly half, as shown by the measurements of the send and receive overhead given in Table 7. Exactly the same optimization is possible in Jacobi, and also in shift operations that occur at the start of the distributed phase in Erlebacher.

In general, a sophisticated programmer will have some idea of what a hand-optimized version of the program would look like. Providing an understanding of the compiler's capabilities can help ensure, first, that the programmer's expectations about compiler-generated code reflect reality, and second, when certain transformations are beyond the capabilities of the compiler, will guide the programmer in "expressing" these optimizations in the Fortran D source code.[4]

The problem of describing a compiler's capabilities to a user presents significant challenges, and we only briefly address it here. An important consideration in developing such a description will be to provide a concise and relatively general description of the transformations that are within or beyond the scope of the compiler, without requiring an understanding of compilation techniques and without sacrificing the key advantages of the data-parallel programming model provided by the language. For example, two key transformations the current Fortran D compiler provides is to combine individual array elements required by a particular loop nest in a single message (when permissible by the data dependences in the program) and to reorder the the iteration space of a loop nest to pipeline the computation as implemented in Erlebacher. Some transformations that are beyond the scope of the compiler at present are to distribute a loop so that iterations that require remote memory accesses can be executed separately from those that do not; to reorder the code of one iteration relative to code *within* the preceding iteration; or to perform code transformations across procedure boundaries. (The latter two transformations are beyond the scope of any data-parallel compiler we are aware of.) In the next subsection, we discuss the various compiler enhancements suggested by the applications studied above.

[4] While the additional option of modifying the compiler generated code might be available, this defeats the very purpose of high-level languages like HPF and Fortran D and we do not advocate this approach.

	TOTAL (sec)	BUSY	IDLE		
			Total	Send	Recv
Unopt	10.29	92.98%	7.02%	0.88%	5.3%
Opt	10.09	96.33%	3.67%	0.85%	1.49%

Figure 7 Improvement with loop distribution in red-black SOR: 4096x4096, block data partition in the first array dimension, 32 processors.

5 Compiler enhancements

The experiments and results of the last three sections suggest a number of enhancements to the compiler. Significant additional functionality is required to support self-tuning in general, and pipeline tuning in particular. In addition, the performance studies of ERLEBACHER, DGEFA, RED-BLACK SOR, and JACOBI suggest compiler optimizations that can be broadly characterized as increasing the overlap between computation and communication, and one goal of our research is to automate these optimizations in the compiler. We briefly outline these compiler enhancements here.

The additional support required for automatically tuning pipelined loops are the software infrastructure for measuring parameter values and associating them with the corresponding source code sections during subsequent recompilation, and the techniques necessary to give the programmer feedback on the results of self-tuning and appropriate control over the tuning process when model assumptions are observed to be violated. Candidate loop nests for pipelining can be easily identified by data dependence analysis. We plan to implement the model of Section 2.1 as our basic method, although new performance measurements may require us to modify it in the future. (To our knowledge no distributed memory compiler has implemented an explicit model for pipeline optimization, although simple pipelining of loops has been implemented by several compilers, including Vienna Fortran [3] and Id Nouveau [21].) If the performance data is unreliable, compiler switches and directives will allow the programmer to specify the parameters directly. If user input is required, however, it may be simpler to specify the block size directly. In fact, this is the only method that our current compiler allows.

Of the communication optimizations suggested by our performance studies, delaying the computations that access nonlocal data (as discussed for DGEFA and RED-BLACK SOR) requires a relatively straightforward enhancement, and has been previous discussed by Hiranandani et al. [12]. The Kali compiler previously implemented this optimization in the restricted case of FORALL statements embodying shift operations [16]. Of the enhancements required to the Fortran D compiler, relatively simple expressions are available to identify the nonlocal iterations for many common cases [4]. The details of the implementation are also relatively straightforward because the loop bodies can be kept intact.

To our knowledge, no parallelizing compiler has attempted the more difficult analysis and reordering of computations needed in order to deduce the optimization for DGEFAfrom the original source code of Figure 4. Van de Geijn's implementation of the LINPACK benchmark on the Intel Delta applied this technique by

hand, as well as using a blocked form of Gaussian elimination to further vectorize communication [23]. In order to be able to carry out the optimizations from the rewritten source code of Figure 5b, the compiler enhancements required are somewhat more straightforward, although, to our knowledge, no parallelizing compiler has implemented this optimization either. First, we must test that it is permissible to promote the send from the top of a loop iteration to a point immediately after the computation of the pivot in the *previous* iteration. This requires testing that the last loop in the iteration does not modify these pivot values, which is relatively straightforward with the available dependence analysis capabilities of the compiler. The major enhancement required is the ability to separate sends from receives in a general fashion, so as to promote the send as described above. In our compilation framework, collective communications such as broadcasts are implemented with a single call containing both the send and receive. Thus, we must also make some minor modifications to our runtime library.[5]

6 Related Work

Many previous performance tools for parallel systems support performance monitoring of parallel programs with explicit parallelism, communication and synchronization [17, 20, 8], while some more recent tools support performance debugging of data-parallel programs at the language level [22, 14]. However, there is relatively little work that on integrating performance analysis and compilation, partly because of the lack of availability of sophisticated parallelizing compilers to the performance evaluation community. One such project is the Vienna Fortran Compilation System, which integrates a static performance prediction tool called PPPT into the compilation process [6]. In particular, PPPT uses sequential profiling to obtain iteration counts and branch frequencies and combines this information with static analysis of a parallelized program to predict a number of parallel performance metrics. These metrics can be used by the programmer to tune program performance, or by the compiler to automatically select efficient data distribution strategies. A key difference in the approach described here is that we also propose to integrate compiler information into the process of performance analysis, in order to make dynamic performance instrumentation more efficient and in order to tune portions of the program automatically based on dynamically measured performance information.

Some previous performance tools have attempted to use program information to reduce the volume of information measured at runtime. Sequential profiling tools such as QPT [1] use control-flow analysis to reduce the volume of profiling or tracing data. Dynamic parallel instrumentation in the W^3 Search Model [13] reduces the instrumentation data volume by using sampled performance information to selectively insert instrumentation in interesting parts of a program at runtime, in order to answer specific performance queries. In contrast to these general-purpose approaches, we can exploit compiler information about specific

[5] If the collective communication requires significant computation or routing, the modifications may not be so minor. Fortunately, many parallel machines use communications coprocessors that can perform these tasks.

communication idioms (e.g., pipelining) to carefully choose the metrics that are most useful as well as to extract only the appropriate summary information needed to obtain those metrics.

7 Conclusions

For languages such as Fortran D and HPF to achieve widespread acceptance, effective techniques for performance analysis and tuning of data parallel programs will be essential. Compilers for these languages play a central role in translating abstract programs written in these languages to explicitly parallel SPMD programs. In this paper, we have provided evidence to show that much can be gained from involving the compiler in the performance tuning process.

Compilers for data parallel languages can be involved in the performance analysis and tuning process at several levels. In the best case, compilers can tune the program without user intervention. We described a preliminary investigation of the potential for automatically tuning pipelined computations. Important aspects of this work are development of an accurate model, and presenting a strategy for collecting the parameters needed to use the model. Our experiences show that if self-tuning approaches such as the one we propose are to be accepted, then we must also collect information that (1) provides feedback about the overall performance to verify effectiveness, and (2) helps recognize when the model may fail to provide accurate guidance because of assumptions that are not valid for a particular program.

Dynamic performance measurement is an important component of performance tuning, whether by the compiler or the user. We used examples to show that we can significantly reduce the cost of dynamic performance measurement by exploiting the deep knowledge about a program available in data parallel compilers like the Rice Fortran 77D compiler. For example, we showed how a compiler could arrange to use multiple blocking factors for different intervals of a pipelined computation in the same execution to determine both two coefficients of linear cost models within the pipeline self-tuning model. As a second example, we can use compiler knowledge of a program's structure as well as a built-in knowledge of what information is important for understanding and tuning specific communication patterns to collect only representative summary statistics to describe related communication events for each processor, thus potentially reducing dynamic trace sizes by orders of magnitude. As an example of this principle, we can separate the steady-state of a pipeline from the leading edge to obtain concise summary statistics of steady-state pipeline timings that provide almost complete information about the performance impact of these events. Even with varying communication behavior, such due to contention, summary statistics can provide substantial information without requiring detailed traces. From these statistical summaries, we can derive parameters for self-tuning models as well as validate the applicability of these models by verifying model assumptions about the homogeneity of work, etc.

When the compiler cannot automatically restructure the code to obtain high performance, it is important that a user be able to (1) understand the source of the problem and (2) improve the performance, overcoming

compiler limitations by supplying additional information. Such additional information could be in the form of directives, or rewriting critical parts of the application in a style where the desired optimizations are more apparent to the compiler, As illustrated with DGEFA, an understanding of the compiler's code-transformation and optimization capabilities can be just as important for performance tuning of data parallel applications as an understanding of the underlying parallel system.

Finally, in our work so far we have considered strategies for automatically tuning program phases in isolation, e.g. individual pipelined computations. However, individually tuning each phase for optimal performance in isolation may not result in optimal performance for the program as a whole, even for programs for which a single static data distribution is appropriate. For example, if a program has two adjacent pipelined computations, tuning each to be of minimal length may not provide best overall performance; rather tuning them together so that the skew in the final phase of the first pipeline matches the skew induced by the initial phase of the second pipeline will minimize total waiting. In future work, we hope to explore the effects of context on tuning as well.

8 Acknowledgments

The authors wish to thank Seema Hiranandani for clarifying details of the Fortran D compiler internals, and Dan Reed and the Pablo group at the University of Illinois for their involvement in a collaborative effort focused on performance analysis of data parallel programs. The iPSC/860 that was used for the experimental studies in the paper was purchased with funds from NSF Institutional Infrastructure Grant CDA-86198393 and support from the Keck Foundation.

References

[1] T. Ball and J. R. Larus. Optimally profiling and tracing programs. In *POPL92*, pages 59–70, Jan 1992.

[2] S. Bokhari. Communication overhead on the Intel iPSC/860 Hypercube. ICASE Report 10, Institute for Computer Application in Science and Engineering, Hampton, VA, May 1990.

[3] P. Brezany, M. Gerndt, V. Sipkova, and H. Zima. SUPERB support for irregular scientific computations. In *Proceedings of the 1992 Scalable High Performance Computing Conference*, Williamsburg, VA, Apr. 1992.

[4] S. Chatterjee, J. Gilbert, F. Long, R. Schreiber, and S. Teng. Generating local addresses and communication sets for data-parallel programs. In *Proceedings of the Fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, San Diego, CA, May 1993.

[5] J. Dongarra, J. Bunch, C. Moler, and G. Stewart. *LINPACK User's Guide.* SIAM Publications, Philadelphia, PA, 1979.

[6] T. Fahringer and H. Zima. A static parameter based performance prediction tool for parallel programs. In *Proceedings of the 1993 ACM International Conference on Supercomputing*, Tokyo, Japan, July 1993.

[7] G. A. Geist, M. T. Heath, B. W. Peyton, and P. H. Worley. PICL: A portable instrumented communication library, C reference manual. Technical Report ORNL/TM-11130, Oak Ridge National Laboratories, Oak Ridge, TN, July 1990.

[8] M. T. Heath and J. E. Finger. Visualizing performance of parallel programs. *IEEE Software*, pages 29–39, Sept. 1991.

[9] High Performance Fortran Forum. High Performance Fortran language specification. *Scientific Programming*, 2(1-2):1–170, 1993.

[10] S. Hiranandani, K. Kennedy, and C. Tseng. Compiler support for machine-independent parallel programming in Fortran D. In J. Saltz and P. Mehrotra, editors, *Languages, Compilers, and Run-Time Environments for Distributed Memory Machines*. North-Holland, Amsterdam, The Netherlands, 1992.

[11] S. Hiranandani, K. Kennedy, and C. Tseng. Compiling Fortran D for MIMD distributed-memory machines. *Communications of the ACM*, 35(8):66–80, Aug. 1992.

[12] S. Hiranandani, K. Kennedy, and C. Tseng. Preliminary experiences with the Fortran D compiler. In *Proceedings of Supercomputing '93*, Portland, OR, Nov. 1993.

[13] J. K. Hollingsworth and B. P. Miller. Dynamic control of performance monitoring on large scale parallel systems. In *International Conference on Supercomputing*, Tokyo, Jul 1993.

[14] R. B. Irvin and B. P. Miller. A performance tool for high-level parallel languages. Technical Report 1204, WISCONSIN, Jan. 1994.

[15] C. Koelbel, D. Loveman, R. Schreiber, G. Steele, Jr., and M. Zosel. *The High Performance Fortran Handbook*. The MIT Press, Cambridge, MA, 1994.

[16] C. Koelbel, P. Mehrotra, and J. Van Rosendale. Supporting shared data structures on distributed memory machines. In *Proceedings of the Second ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, Seattle, WA, Mar. 1990.

[17] B. P. Miller, M. Clark, J. K. Hollingsworth, S. Kierstead, S. Lim, and T. Torzewski. Ips-2: The second generation of a parallel program measurement system. *TOPDS*, 1(2), Apr. 1990.

[18] J. Ramanujam. *Compile-time Techniques for Parallel Execution of Loops on Distributed Memory Multiprocessors*. PhD thesis, Department of Computer and Information Science, Ohio State University, Columbus, OH, 1990.

[19] J. Ramanujam and P. Sadayappan. Tiling multidimensional iteration spaces for nonshared memory machines. In *Proceedings of Supercomputing '91*, Albuquerque, NM, Nov. 1991.

[20] D. A. Reed, R. A. Aydt, T. M. Madhyastha, R. J. Noe, K. A. Shields, and B. W. Schwartz. An overview of the pablo performance analysis environment. Technical report, UIUCCS, Nov. 1992.

[21] A. Rogers and K. Pingali. Process decomposition through locality of reference. In *Proceedings of the SIGPLAN '89 Conference on Program Language Design and Implementation*, Portland, OR, June 1989.

[22] M. Thinking Machines Corp., Cambridge. Prism reference manual. Technical report, 1992.

[23] R. van de Geijn. Massively parallel LINPACK benchmark on the Intel Touchstone Delta and iPSC/860 systems. In *1991 Annual Users' Conference Proceedings*, Dallas, Oct. 1991. Intel Supercomputer Users' Group.