Compiler Support for Analysis and Tuning Data Parallel Programs

Vikram Adve
Charles Koelbel
John Mellor-Crummey

CRPC-TR95520
March, 1995

Center for Research on Parallel Computation
Rice University
6100 South Main Street
CRPC - MS 41
Houston, TX 77005

Compiler Support for Analysis and Tuning Data Parallel Programs*

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

Abstract

Data parallel languages such as High-Performance Fortran (HPF) and Fortran D simplify the task of parallel programming by enabling users to express parallel algorithms at a high level. Compilers for these languages are responsible for realizing parallelism and inserting all interprocessor communication. For this reason, these compilers have detailed knowledge of the the relationship between its parallelism and communication in the program. Here, we explore how this knowledge can be exploited to support the process of tuning programs for high performance.

1 Introduction Data parallel languages such as High-Performance Fortran (HPF) [9, 15] and Fortran D [11] support an abstract model of parallel programming. The principal advantage of these abstract languages is that they insulate programmers from the intricacies of concurrent programming and managing distributed data. Users write a single-threaded program augmented with data layout directives, which a sophisticated compiler uses to derive a single-program-multiple-data (SPMD) parallel program.

Since compilers for these data parallel languages are responsible for communication insertion and management of parallelism, they have deep knowledge of both and the relationship between them. This knowledge uniquely qualifies these compilers to provide assistance for performance evaluation and tuning.

First, performance tuning of parallel programs is a tedious process and ideally should be performed automatically to minimize the burden on users. Here, we present an overview of a performance model based on measurements of a program's dynamic behavior that a compiler can apply to automatically tune pipelined computations. Since such a *self-tuning* compiler necessarily relies on a a simplified model of program and system behavior that may be violated in practice, we describe how the model can be tested for applicability.

Second, measurements of a program's run-time behavior are important for understanding performance problems as well as for compiler self-tuning. We describe some code transformations that a compiler can apply that both increase the utility and reduce the data volume of dynamic performance traces.

Finally, we argue that for users to participate effectively in tuning programs written in data parallel languages, they must have an understanding of a compiler's optimization and transformation capabilities. Since compilers play such a central role in realizing high performance with such languages, it is imperative that users know how they can express data parallel algorithms in a form that will be understood by the compiler. Without realistic expectations about the compiler's abilities, attempting to restructure data parallel programs to achieve high performance can be a frustrating experience.

*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. Authors' address: Center for Research on Parallel Computation, Rice University, P.O. Box 1892, Houston, TX 77251. Email: {adve, chk, johnmc}@cs.rice.edu.

Throughout the paper, we illustrate our points using benchmark kernels 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 this 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.

In section 2 we discuss a self-tuning model for pipelined computations. Section 3 describes compiler support to improve the utility and compactness of dynamic performance traces. Section 4 elaborates on the need for programmers to understand the capabilities of data parallel compilers for performance tuning. Section 5 briefly places this work in the context of previous research with related goals. Section 6 summarizes our conclusions and our plans for future work.

2 Self-Tuning Compiler The communication patterns of a program written in a high-level data parallel language like HPF are inserted directly by the compiler, and the details of these communication patterns are often not directly visible to the programmer. With the appropriate static and dynamic performance information, the compiler itself could tune the performance of such communication patterns. We believe it is important to identify such opportunities for compiler "self-tuning" and minimize user involvement in tuning parts of a program where self-tuning strategies can be applied effectively. In the following, we use the example of pipelined communication to discuss how such self-tuning might be carried out in practice.

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. By deriving an expression for the optimal granularity in terms of system and program parameters, the compiler could estimate or measure the various parameters to optimize the execution time of the pipeline.

Previous authors [21, 12, 18, 19, ?] have recognized this trade-off and presented models for choosing the pipeline granularity. The most general model is that of King et al [?], who describe a Petri net based model to predict the execution time of a one or two-dimensional pipelined data-parallel computation, but the model is complex and not intended to be used directly in a parallelizing compiler. The special case of a one-dimensional pipeline under simplifying assumptions leads to a simple closed-form expression for the execution time, from which the optimal granularity can be directly derived. We separately derived a similar model based on similar assumptions. In particular, our model includes the influence of block sizes on communication time, and it accounts for non-overlapping communication and computation at the leading edge of the pipeline but perfect overlap in the rest of the pipeline, as motivated below.

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. Later in this section, we discuss a uniform approach for detecting and handling violations of model assumptions.

Consider a one-dimensional pipeline (as in Figure 1) operating on 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 bloc k 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. Under our assumptions that the pipeline stages are perfectly balanced, a number of critical paths with identical execution times exist, one of which is illustrated in the figure. The execution time of a critical path gives the total pipeline execution time, and can be computed as follows (where processors are numbered $0 \ldots P-1$):

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

We express these quantities in terms of the send and receive overhead per pipeline stage and the computation time per pipeline stage, where each of these terms is a linear function of B. Differentiating the resulting expression gives the value B_{opt} where $T_{pipe}(B)$ is minimized. The details are omitted and are available elsewhere **??**.

To apply the model in practice, the constant and linear coefficients for computation time and send and receive overhead must be predicted or measured. While the coefficients for communication overhead can be separately measured for a given system, the coefficients for computation time may be more difficult to predict accurately. Thus, the compiler might use an initial estimate to compute an approximate block size but subsequently use runtime measurements to obtain accurate parameter values and re-compile the program with an "optimal" block size.

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 **??**. [1] Using these parameter values, the model predicts the optimal block size to be $B_{opt} = 88.0$. Increasing the block-size to B=64 yielded substantial improvement in performance. (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

[1]In practice, we would not have to measure the coefficients of communication overhead because these are fixed system values for a given message size, as explained above. At present, we do not have measurements of these values available.

the i and j loops, which may not be implemented in a parallelizing compiler.) With B=64 on 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%. The specific improvement obtained in any particular case will depend on the the accuracy of the initially predicted parameter values (which determine the initially chosen block size), and on the relative fraction of the execution time devoted to pipelined computation. The experiment above shows that pipeline self-tuning has the potential to provide substantial improvements in performance without user intervention.

In an actual pipeline execution, some of the simplifying assumptions in the model may be violated. For example, the execution time per pipeline stage may vary within and across processors because of cache effects, conditionals and even floating-point operation timings. Since the model assumptions are "known" to 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 pipeline computation time, in addition to the mean. The compiler could also test if the mean values 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. Thus, a fairly careful approach to handling violations of assumptions made by a self-tuning performance model can be built in to the compiler. 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).

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 **??**, 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 **??**. 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 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 the ERLEBACHER and SOR benchmarks. In both cases we found that there was significant variation in the initial receive times because processors are not synchronized when entering the pipeline, but times for subsequent sends and receives were quite uniform, as demonstrated by a small COEFFICIENT OF VARIATION for operation timings. 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 3a, 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 3b.

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 4a. We traced the communication operations in the two versions of the program using the Pablo instrumentation library [20]. Table 5 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 3a. 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 4b, 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 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 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 6. 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.[2]

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.

5 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

[2]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.

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 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.

6 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.

7 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, Jan 1992, pp. 59–70.

[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, (1991), pp. 29–39.

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

[10] S. Hiranandani, K. Kennedy, and C. Tseng, *Compiler support for machine-independent parallel programming in Fortran D*, in Languages, Compilers, and Run-Time Environments for

Distributed Memory Machines, J. Saltz and P. Mehrotra, eds., North-Holland, Amsterdam, The Netherlands, 1992.

[11] ——, *Compiling Fortran D for MIMD distributed-memory machines*, Communications of the ACM, 35 (1992), pp. 66–80.

[12] ——, *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 (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*, tech. rep., 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.

```
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

```
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 3: Gaussian elimination with serial processing of pivot column

```
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-
   enddo
  endif
 enddo
 do row elimination for col NMAX with pivot(NMAX-1) ---
end
```

(b) Expressing the optimization in Fortran D.

Figure 4: Minimizing serialization in Gaussian elimination.

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

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