Requirements for Data-Parallel Programming Environments

Vikram Adve, Alan Carle, Elana Granston, Seema Hiranandani, Ken Kennedy, Charles Koelbel, Ulrich Kremer, John Mellor-Crummey, Scott Warren

Center for Research on Parallel Computation, Rice University

Chau-Wen Tseng

Center for Integrated Systems, Stanford University

1.0 Introduction

Over the past decade, research in programming systems to support scalable parallel computation has sought ways to provide an efficient machine-independent programming model. Initial efforts concentrated on automatic detection of parallelism using extensions to compiler technology developed for automatic vectorization. Many advanced techniques, including interprocedural compilation, were tried. However, after over a half-decade of research, most investigators were ready to admit that fully automatic techniques would be insufficient by themselves to support general parallel programming, even in the limited domain of scientific computation. In other words, in an effective parallel programming system, the programmer would have to provide additional information to help the system parallelize applications. This realization led the research community to consider extensions to existing programming languages, such as Fortran and C, that could be used to help specify parallelism.

An important strategy for exploiting scalable parallelism is the use of *data parallelism*, in which the problem domain is subdivided into regions and each region is mapped onto a different processor. For example, if we wish to initialize the values in an array of 1000 elements on a parallel machine with 10 processors, we might map a 100-element section of the array to each processor and perform the initializations concurrently, speeding up the computation by a factor of 10. Data parallelism is *scalable* because we can use an array of 100 processors to initialize an array of

10,000 elements in approximately the same running time. In other words, we can increase the number of processors and the amount of data proportionately and expect the running time to stay roughly the same.

These factors have led to a widespread interest in *data-parallel languages* such as Fortran D, High Performance Fortran (HPF), and DataParallel C as a means of writing portable parallel software. In these languages, there is typically a mechanism for specifying the mapping of data elements to processors along with some way of specifying aggregate operations that can be performed on the array of processors in parallel. The compiler technology required to support a data-parallel language efficiently on a parallel machine is very sophisticated [15,16], so it is not surprising that most of the research to date has been focused on high-performance compilers [5,20,29,30,36].

Once a good compiler for a data-parallel language is available, another problem arises. Because data-parallel compilers perform aggressive program transformations, including some that affect more than one procedure, the relationship between the source program and the object program may be difficult for programmers to understand. To help the programmer make good design decisions, the programming system should include mechanisms that explain the behavior of object code in terms of the source program from which it was compiled. For sequential programs, the standard *symbolic debugger*, supporting single-step execution of the program source rather than the object program, provides such a facility. A more recent example is the *interactive vectorizer*, which provides the programmer with an indication of which statements in the program can be executed on the vector unit, along with explanations of why some statements fail to vectorize. Tools like these have proved essential for the use of conventional supercomputers and they have set a minimum standard for all scientific programming systems.

Because data-parallel computation is more complex than simple vector computation, even more sophisticated tools will be needed to help programmers understand the behavior of object programs compiled from data-parallel languages. The goal of this paper is to convey an understanding of the tools and strategies that will be needed to adequately support efficient, machine-independent data-parallel programming. To achieve our goal, we will examine the requirements for such tools and describe promising implementation strategies for meeting these requirements.

The remainder of this article is organized as follows. Section 2.0 introduces data-parallel languages and further motivates the special support they need. Section 3.0 gives a user-level view of a data-parallel programming environment, while Sections 4.0 through 7.0 describe several of its components. Finally, Section 8.0 gives our conclusions.

2.0 Data-Parallel Languages

Acceptance of current parallel machines has been hindered by their lack of software, a situation that was exacerbated by the lack of a portable programming model. In the past, machines were delivered with variants of Fortran and C extended to reflect the underlying hardware. For example, distributed-memory machines required the use of machine-specific message-passing libraries, while shared-memory machines required the use of parallel loops and dynamic tasks. Because these languages reflected the underlying hardware so closely, they tended to reduce portability. The current trend in language support is toward languages and compilers that support *machine-independent parallel programming*—that, for a given algorithm, can produce code that performs as well on each target machine as the best version of the same algorithm hand-coded in the target's standard machine-dependent programming interface. In other words, the algorithm should perform well if it is well-suited to the target architecture and should run as well as possible on an architecture for which it is ill-suited.

Data-parallel languages are one currently popular approach to supporting machine-independent programming for data-parallel problems, in which operations are applied to every element of a data domain and parallelism is achieved by assigning each processor to a piece of the whole domain. Fox [12] has reported that data parallelism is the most common form of parallelism in scientific calculations because it permits larger problems to be solved by increasing the number of processors. Moreover, several research projects [5,16,20,29,30,36] have shown how these programs can be efficiently compiled for a variety of parallel machines.

Data-parallel languages directly reflect the data-parallel paradigm. In this paper, we will use High Performance Fortran as our principal example. HPF expresses data-parallel operations using Fortran 90's array operations, including element-wise arithmetic operations on arrays, elemental intrinsic functions, and data reduction operations. The HPF `FORALL` statement extends array assignments to allow new array shapes and somewhat more general expressions. Such a repertoire of abstract parallel constructs is typical in data-parallel languages. Of course, HPF also includes

```
REAL A(6,6)
!HPF$ DISTRIBUTE A(*,BLOCK)
DO I = 2, 5
 A(I,:) = A(I-1,:) + A(I+1,:)
END DO
DO J = 2, 5
 A(:,J) = A(:,J-1) + A(:,J+1)
END DO
```

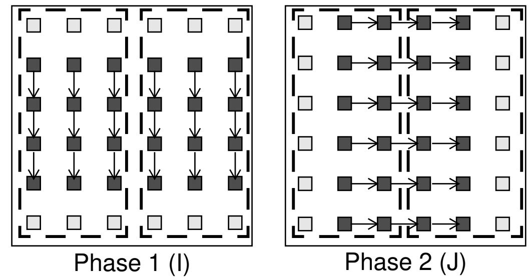

FIGURE 1. Line Relaxation in HPF

the standard Fortran sequential statements such as `DO` loops; we believe that high-quality compilers will detect parallelism in these constructs as well. Perhaps most importantly, HPF provides a rich set of data alignment and distribution options allowing array dimensions to be mapped independently in regular patterns. The compiler uses these annotations to partition data and computation among physical processors. The data mapping defined by the combination of alignment and distribution can have a substantial impact on the performance of the program. For a more detailed description of HPF, we refer the reader to Koelbel et al. [19]; we trust that the short examples in this article will be self-explanatory.

Notably absent from the above description are the detailed, machine-dependent synchronization operations such as message passing and semaphores common to task-parallel languages. An HPF compiler must generate these low-level operations in the object code, just as a scalar compiler must efficiently use registers. Moreover, it is extremely important that the compiler optimize the low-level operations if the compiled program is to be efficient. We demonstrate this in Figure 1 and Figure 2. Other papers in this special issue describe the necessary compiler optimizations in more detail.

Consider the simplified line relaxation program in Figure 1. The algorithm consists of two phases, each of which performs an ordered sequence of vector operations. As the figure suggests, the operations in one dimension are serialized, while operations in the other dimension are conceptually parallel. (Small squares represent array elements; dark squares are elements that receive new values and must be updated in the order shown by arrows.) This suggests that any implementation will suffer from some serialization. As Figure 2 shows, the compiled code can reduce this serialization on a two-processor message-passing system by using pipelining in the second phase. (Again, dark squares represent assignments to array elements; white squares are message-passing

```
REAL A(6, 0:4)
DO I = 2, 5
 DO K = 1, 3
   A(I,K) = A(I-1,K)+ A(I+1,K)
 END DO
END DO
IF (MY_NODE > 0)  SEND A(1:6, 1)
IF (MY_NODE < 1)  RECV A(1:6, 4)
LB_1 = Lower bound of J loop on MY_NODE
UB_1 = Upper bound of J loop on MY_NODE
DO K = 1, 6
 IF (MY_NODE > 0)  RECV A(K, 0)
 DO J = LB_1, UB_1
   A(K,J) = A(K,J-1)+ A(K,J+1)
 END DO
 IF (MY_NODE < 1)  SEND A(K, 3)
END DO
```

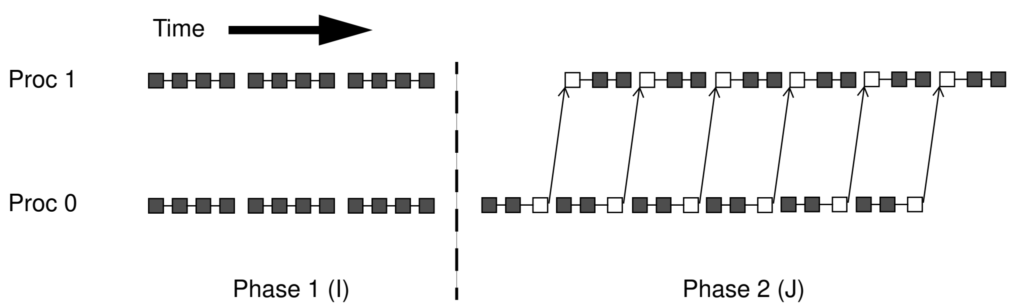

FIGURE 2. Translated Line Relaxation, pipelined execution

operations.) In effect, the compiler breaks the vector operations into segments, allowing computation to start on the second processor before the first completes its task. This transformation allows the phase to execute partially in parallel, at the cost of increased code complexity.

This example suggests several challenges for a data-parallel programming environment:

- Can programmers debug their programs when the actual order of execution is changed by the compiler?
- Can programmers predict and tune the performance of their codes when a complex compiler performs extensive transformations?
- Can the environment assist programmers in choosing good mappings for their data structures?

Addressing these concerns requires that the programming environment's tools know about the compiler's actions and have access to all of its information. The next section describes a system

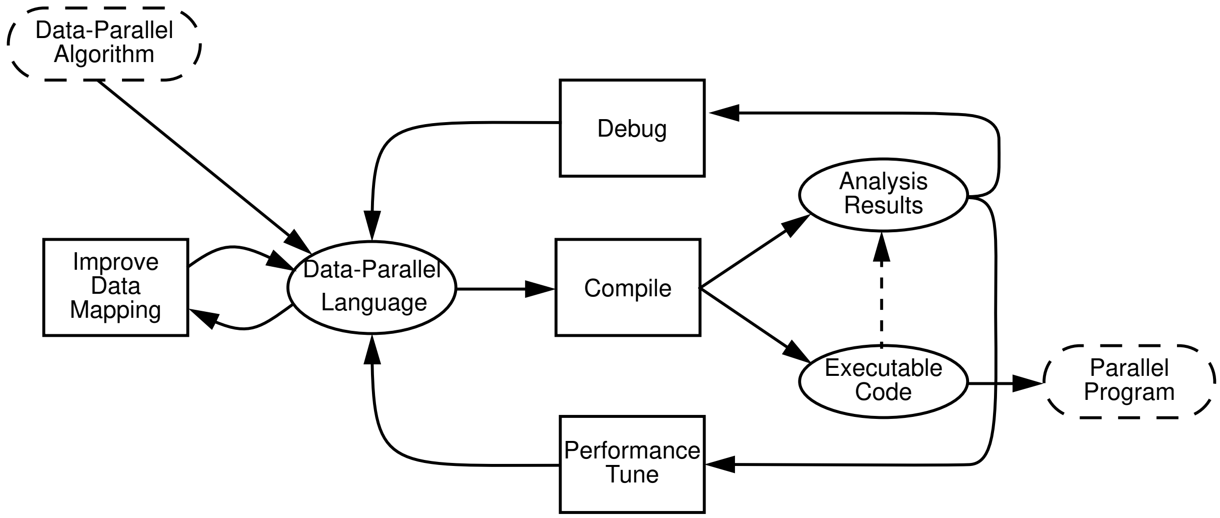

FIGURE 3. Developing a parallel program

architecture to achieve this, while Sections 4.0 through 7.0 describe individual tools in more detail.

3.0 A User's View of the Environment

Figure 3 shows how a programmer might use a data-parallel programming environment to build a correct and efficient HPF program through a converging series of edit-compile-test cycles. To identify program errors, the programmer would compile the program and invoke the debugger. The debugger would use both the executable code and records of the compiler transformations to interpret machine-level operations in terms of the data-parallel source program. After correcting the program, the programmer would attempt to tune it using performance information collected statically at compile time and dynamically at run time. Tuning might require changing the algorithm to a more parallel one, which would then probably require further debugging. Alternatively, performance information might indicate the programmer's choice of data mappings to be inappropriate, in which case the data mapping assistant could be invoked to suggest improvements. The code would be considered ready for production use only when the programmer was satisfied with its correctness and efficiency.

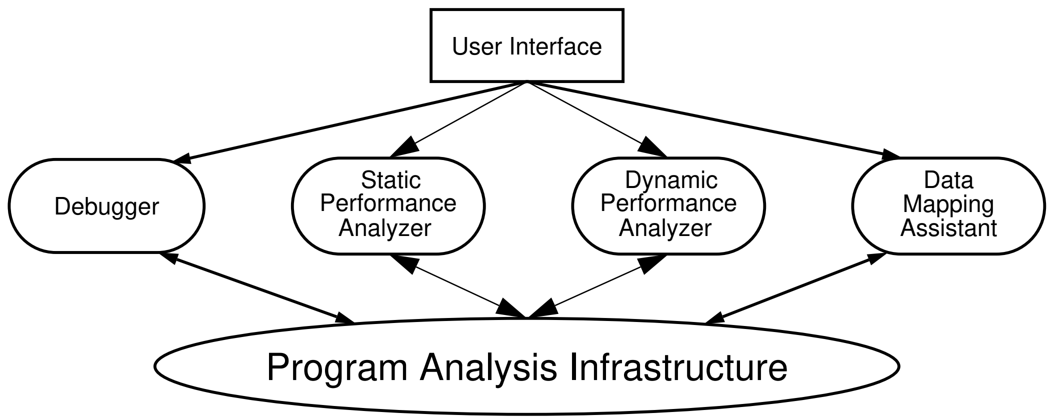

FIGURE 4. High-level environment overview

Figure 4 presents a schematic of a data-parallel programming environment designed to support debugging, performance analysis and visualization, and data mapping selection in terms of source-level programming language concepts. The focus of the environment is a program analysis infrastructure that computes and stores analysis results, applies and logs program transformations, and generates parallel code. Other data-parallel tools will rely on this infrastructure. A standard user interface will provide intuitive access to the entire tool set.

The design of the D System, a suite of integrated tools under construction at Rice University, closely resembles the figure. As Fortran D is a predecessor of HPF, we believe that most, if not all, of the features of the D System will carry over to environments for HPF and other data-parallel languages.

The program analysis infrastructure of the D System acts as a demand-driven Fortran D compiler and as a repository for information computed by other tools in the environment. Any tool that requires access to information about a program, such as the bidirectional mapping information between high-level source code and low-level executable code, will "ask" the infrastructure for the information. Tools will also store information they collect in the infrastructure for access by other tools.

The D System data-parallel debugger (described in Section 4.0) will provide the functionality of a sequential source-level debugger at the level of the data-parallel source program. Since compiler transformations change the order of operations in the program, the debugger will make use of transformation records provided by the program analysis infrastructure to describe program state and to explain program behavior. Performance analysis tools (described in Section 5.0 and Section 6.0) will help identify bottlenecks in the program, and may suggest solutions in some cases. Information provided by the program analysis infrastructure will be used to explain why particular transformations were (or were not) performed. A data mapping assistant (described in Section 7.0) will suggest data mappings for a program. The data mapping assistant may be thought of as high-level performance tuning tool, or as a tool to convert older codes to data-parallel form. In either case, rational suggestions can only be made using program analysis information placed in the infrastructure, gathered either statically by the compiler or dynamically by the performance tool.

The primary user interface to tools in the D System is the D Editor, a structured editor for Fortran. The D Editor is able to display information generated by the program analysis infrastructure or gathered by system tools. This information is displayed as either textual or graphical annotations associated with the data-parallel source code. Some examples of these annotations are discussed in Section 5.0. For some tools, the editor interface is optional; in those cases, we also provide stand-alone programs that can be invoked from the command line.

4.0 Debugging

A major advantage of the data-parallel programming paradigm is that details of communication and synchronization are handled by the compiler rather than the programmer. For example, the semantics of an HPF array assignment are specified by the language definition; compilers for different target machines must ensure that this definition is faithfully implemented regardless of the machine model or data mapping in effect. Most research to date on debugging parallel programs has focused on developing strategies to replay executions of parallel programs [4,21] and to pinpoint causes of non-deterministic behavior caused by improper use of synchronization primitives [8,23,26,31]. While these classes of problems are largely avoided by using a data-parallel language, there are still significant challenges for debugging data-parallel programs. These include verifying the correctness of programs on large data sets, exploring the intermediate execution states of radically-transformed programs, and verifying programmer assertions exploited by a data-parallel compiler.

A challenging problem facing developers of data-parallel programs is identifying when computed values in large data sets are incorrect and pinpointing the computational errors that caused them. Visualization techniques can be invaluable for discovering errors in large data sets. For example, Thinking Machines' Prism debugger [32] provides a rich set of facilities for supporting data visualization. Once a programmer has identified an incorrect data value in a program execution, a debugger must permit the programmer to isolate the fault in the program, typically by supporting the traversal and inspection (at the source level) of intermediate states in the program's execution.

Debuggers for data-parallel languages should strive to maximize source-level transparency — where possible, programmers should be unaware that the execution order of their programs may differ substantially from the execution order specified by their data-parallel source. For debugging optimized sequential code, there is a considerable body of work that addresses the issue of transparently supporting queries about variable values in the presence of optimizations such as common sub-expression elimination and register allocation [7,14]. In the presence of optimizing transformations, full transparency can be achieved by keeping multiple copies of each array data element [27]. For programs with large data sets, the space required for this approach will be completely unacceptable.

When a loop nest in a data-parallel program is partitioned for parallel execution, full transparency is feasible only in a limited set of circumstances. If each processor traverses its portion of the data domain in an order consistent with the order in the original program, then full transparency is possible without additional copies of data elements. If the actual computations respect the original program order within a loop nest, then the debugger can arrange to halt the computation at any intermediate state in the original loop nest computation through judicious setting of conditional breakpoints in the transformed code for each of the processors. In other cases, full transparency without copies is impossible. The traditional approach of not performing transformations that inhibit transparency may remove most or all of the parallelism from a program, which could make the debugging process painfully slow.

However, programmers may be willing to trade some loss of debugging transparency in return for the high performance that parallel execution of an aggressively restructured program can offer. For example, the Convex CXdb debugger pursues an approach in which the debugger interface helps programmers understand the effects of program transformations instead of striving for transparency [4]. The challenge for debugger design is to find an acceptable balance that offers transparency when possible, but provides programmers with enough information to understand the execution state of the transformed program when transparency is impossible. For example, although it may not be possible to provide full transparency everywhere, clean points in the execution may exist between loop nests where full transparency is possible.

In the D System debugger, we intend to provide programmers with mechanisms to advance an execution in terms of abstract events that will facilitate navigation of clean points. Another interesting idea that we will investigate is to try to wean programmers from expecting iterations in a loop nest to be executed in the standard sequential order. Suppose that, instead of setting unconditional breakpoints in the code and stepping through them the proper number of times until an iteration of interest is reached, programmers specify a set of data breakpoints that halt execution when particular variables are read or written or a set of conditional breakpoints that halt execution in particular loop iterations. Then programmers should be much less sensitive to changes in iteration execution order.

We are also considering providing the notion of a data-parallel array step as a primitive for the D Debugger. If we view the underlying machine as a multidimensional SIMD processor, then an array step is the computation that will be done in a single step on such a machine. This directly reflects such HPF statements as array assignments and `FORALL` loops. In compilers that detect implicit parallelism, however, the implications may not be as intuitive; for example, a single `DO` loop may correspond to many array operations. One possibility is to highlight loops involved in array steps; another is to display equivalent Fortran 90 code with explicit array steps. However the array step is displayed to the programmer, a second level of interpretation will also be necessary to translate low-level operations in terms of array steps. This could be quite complex. For instance, consider the example in Figure 2, where pipelining optimizations were applied to divide array assignments into smaller operations to improve performance. Presenting debugging information transparently to the programmer is nontrivial.

Another challenge for data-parallel debuggers is assertion checking. In previous compilers, assertions (known as compiler directives) demanded that the compiler perform certain actions, such as executing a loop in parallel. HPF introduces the `INDEPENDENT` directive which asserts that iterations of a particular loop do not interfere with each other. Although this information implies the loop can be run in parallel, it also means that the loop can be executed in vector mode or in several other ways; the compiler is free to use this information as it sees fit. Because the information may be used indirectly, a false assertion can cause unexpected and unexplained behavior, including non-determinacy or incorrect results. Such false statements are illegal in HPF, but programmers may accidentally make them. HPF debuggers will need to provide means to check such assertions. Checking assertions can be as simple as testing the value of a scalar or as complex as checking for data races that can occur in a loop parallelized on the basis of a false `INDEPENDENT` assertion.

It is clear that many issues remain unresolved for developing effective source-level debugging tools to work with restructuring compilers for data-parallel languages. Coordinating debuggers and data-parallel compilers to support efficient assertion checking, and exploring strategies for providing useful source-level debugging capabilities in the absence of full transparency will require considerable research and experimentation.

5.0 Static Performance Analysis

Many aspects of a data-parallel program's performance are implicit. For example, the quantity and frequency of interprocessor communication are implied by the data mappings, the program's data access patterns, and the compilation strategy. Therefore, it is important that the programming environment provide both *qualitative* and *quantitative* performance information to the programmer. Qualitative information might include indications that certain data references cause interprocessor communication, whereas quantitative information might include estimates of the amount of data transferred or the total time required for data movement. This information can naturally be provided as program annotations to the source program, viewed through a graphical user interface or a listing file such as those generated by vectorizing compilers. For example, the D System uses the D Editor to interactively provide qualitative annotations about data mapping, parallelism, and communication. Eventually, static performance estimates will be presented using the same interface. In this section, we discuss the types of static performance information that can be provided to the programmer, using the D Editor and the Vienna Fortran Parameter-based Performance Prediction Tool (PPPT) [10] to illustrate the main points. In the next section, we will discuss dynamic performance analysis.

Compile-time analysis can provide qualitative information about the three major factors affecting program performance on distributed-memory machines: data mapping, parallelism, and communication. In each case, the information can be provided at one or more levels: individual variable references, statements, loops, procedures, and the entire program. The system design of Figure 4 makes this information available to the static performance tool via the shared infrastructure.

Information about data mapping is fundamental to understanding the parallelism and communication characteristics of a program because the mapping typically determines the strategy for partitioning the computation (for example, using the “owner computes” rule), as well as the set of references that are non-local and thus require communication. The important features of data mapping, all of which can be derived directly from the results of compiler analysis, include the decomposition of each distributed array, the relative alignments of different arrays, and the set of variables that are replicated across the processors. For example, the D Editor uses static analysis to present data mapping information for all the arrays accessed in the currently selected loop.

The second important class of information is parallelism. By using compile-time analysis to calculate cross-processor data dependences and thus determine when synchronization is necessary, the tool can label individual portions of the computation (typically, individual loops) as sequential, pipelined, or fully parallel. In the first two cases, the tool can also extract and display information about the specific data dependences that inhibit full parallelism. Together, these annotations tell the programmer how much parallelism can be expected in individual phases of the program, and provide the compiler’s reasoning. For example, the D Editor requests the necessary information via the compiler interface and “labels” the code by coloring sequential sections red, pipelined sections yellow, and fully parallel sections green. A separate dependence pane displays the data dependences carried on the selected loop. Cross-processor dependences identify values that are defined and used on different processors; they are also marked red, yellow, or green depending on their effect on parallelism. Dependences selected in the dependence pane are displayed as colored arrows in the source pane.

Finally, the tool can provide qualitative information on the communication that will be generated by the compiler. For a selected code section, the compiler interface can be used to obtain the type, size, and location of the inter-processor data movement in the program. Furthermore, in languages where all communication is inserted by the compiler, all this information can be presented in terms of high-level communication patterns such as array shifts, pipelines, broadcasts, reductions, etc., rather than in terms of the individual messages that implement these communication patterns. Although communication is not explicit in the programming model of such a language, the need for non-local data accesses is implied by the program’s data mapping information and data access patterns. We believe that feedback at the level of these communication patterns is the appropriate mechanism for exposing the performance impact of these non-local data access. Thus, information about communication obtained via the compiler interface is presented in just this manner by the D Editor.

In addition to qualitative program annotations, compiler analysis could also be exploited to provide quantitative performance feedback. In general, static performance prediction assumes parameters such as loop iteration counts and branch frequencies are known or can be approximated, and provides static performance metrics specific to the input data set corresponding to these parameters. Some important metrics such as communication volume and frequency can then be computed accurately. Other metrics that can be estimated using approximate models of program and system behavior include the parallelism achievable in various code sections, the extent of load-imbalance, communication latency and overhead for specific communication operations. These metrics can be computed and presented selectively for individual code sections, as well as the entire program. By combining these detailed measures appropriately, overall execution time, speedup and efficiency of the program can be presented as well. Vienna Fortran's PPPT computes and presents a comprehensive set of such metrics to the programmer and also makes these metrics available to the compiler for automatic selection of optimization strategies.

The chief difficulty in deriving static performance estimates lies in obtaining key profiling parameters such as loop iteration counts, branch frequencies, and cache miss rates for the parallelized program. To compensate for this difficulty, PPPT relies on a single sequential profiling run to measure the necessary profiling parameters, but estimates cache miss rates from a simplified data access model. Because of the manner in which these miss rates are derived, they can only be used to rank kernels with respect to cache performance [10]. An alternative approach for static performance prediction taken by Balasundaram et al. [2] is to use *training sets* to estimate both the overall computation time of code sections as well as the communication costs of specific communication patterns.

Collectively, static quantitative and qualitative information allow the programmer to make appropriate algorithmic choices during code development. Static qualitative information can augment programmers' intuition about the performance impact of source program constructs. It can also help the programmer understand quantitative performance information either from static estimates or dynamic measurements. Static quantitative performance estimates can also be used to perform some tuning before executing the program, and can be used by other automatic tools.

6.0 Dynamic Performance Analysis

Because the major motivation for using parallel machines is to obtain fast execution times, support for extensive performance tuning must be a high priority in data-parallel programming environments. In the case of a data-parallel language supported by an optimizing compiler, deep analysis and transformation of the program present significant challenges as well as valuable opportunities for performance tuning. A dynamic performance analyzer for a data-parallel language should present information about parallelism and communication in the context of the source program, without requiring the programmer to understand the transformations or the details of the executable code.

We envision solving this problem in two steps: first, by presenting a high-level model of program performance to the programmer for use in tuning, and second, by translating static and dynamic performance information to this model. Detailed compiler information can be used to support these steps in a variety of ways. In addition to supporting static performance prediction as described in the previous section, compiler information can be used to improve the efficiency of runtime performance measurements, and to translate low-level dynamic information about other performance characteristics to the source-level model.

The task of presenting performance information about a data-parallel program to the programmer is challenging for a number of reasons:

1. The parallelism of a source-level construct may not be obvious in quantitative terms. This is particularly true when the compiler detects implicit parallelism, but also when the compiler replicates some computation (more generally, relaxes the so-called *owner-computes rule* to reduce communication) or introduces pipelining to obtain higher parallelism at the cost of more frequent communication.
2. The communication resulting from a construct and its quantitative impact on performance are often not obvious. This is a particular problem for data-parallel languages like HPF, in which all communication is inserted by the compiler.

3. Both of the above are exacerbated when a compiler optimization can be parameterized. For example, in coarse-grain pipelining [16], the compiler chooses a block size for data transfers to balance parallelism with communication overhead.
4. When reasoning about program parallelism and communication, characteristics of the compiler and the target architecture must be taken into account. For example, a simpler compiler might not perform pipelining. Similarly, on a machine with a fast data transpose operation redistributing arrays may be more efficient than pipelining the computation.

To insulate the programmer from these complexities as much as possible, the performance model and the performance tuning environment must meet five requirements. First, the model must quantify program performance characteristics using metrics whose meaning is independent of the underlying compiler and machine architecture. For example, such metrics for parallel overhead include communication volume and frequency, and the proportion of time spent in various activities such as waiting for remote data.

Second, the environment must provide system-independent visualization of these metrics relating them to the constructs in the data-parallel program. For example, in the D System, we plan to use a hierarchical visualization scheme that presents performance metrics at multiple levels of granularity, including information for an entire program, for individual arrays, and for individual loop-nests. We also plan to tailor visualizations for specific communication patterns such as shifts, pipelines, and broadcasts.

Third, the environment should minimize the need for a programmer's involvement in tuning parameterized constructs introduced by the compiler. For example, the programmer should not have to choose the block size for coarse-grain pipelining. The compiler itself should tune such constructs using static and dynamic performance information. In practice, however, programmers may desire greater control and predictability over program behavior, or their involvement may be required if the compiler's knowledge about program and system behavior proves insufficient.

Fourth, the environment should provide programmers with some understanding of the capabilities of the compiler. This allows the programmer to understand their program's performance characteristics and to optimize the data-parallel code to increase the effectiveness of the compiler. The challenge is to provide such an understanding without requiring deep knowledge of compilation techniques and without sacrificing the key advantages of the data-parallel programming model.

Finally, it is desirable for the compiler and the underlying parallel machine to ensure that the performance characteristics of the program are predictable. For example, system artifacts such as interrupts or contention for communication resources should not radically change a program's behavior. Unfortunately, such artifacts are often beyond the control of the performance tuning tools. However, careful use of system resources by the compiler and runtime system can sometimes minimize these effects. We believe that ensuring predictability is particularly important in the context of HPF and similar languages, where the connection between the original source and the actual executable is very indirect.

The dynamic performance data obtained from runtime measurements necessarily provides only low-level performance information about the explicitly parallel executing code. This information must be presented to the programmer in terms of the more abstract data-parallel model, if possible using a graphical performance visualization tool. Traditional symbol table information is insufficient if we wish to relate performance information to source-level constructs since a single low-level feature may correspond to several high-level operations and vice-versa. Instead, the compiler can record annotations describing the various optimizations and code transformations in the shared infrastructure; performance tools can read these annotations to translate low-level performance information to constructs in the data-parallel source. For example, the Fortran D compiler will create a file of annotations in the same data format as that used for dynamic traces in the Pablo performance analysis environment [28]. The Pablo visualization tool will use this file to interpret dynamic trace information and present it to the programmer in terms of the Fortran D source.

The process of measuring program performance at runtime can also be made more efficient by exploiting compiler information about program communication and parallelism. First, runtime measurements can be minimized by replacing dynamic tracing with static information wherever possible. Typically, only actual timings and symbolic values that cannot be resolved at compile-time need to be recorded in the dynamic trace. Second, in some cases summary statistics may suffice instead of detailed measurements. For example, in a pipelined loop, all pipeline stages on each processor except the first stage may have fairly uniform computation and communication behavior. The compiler can peel off the iterations of the first stage with little perturbation to the program, and collect only statistical summaries describing computation and communication in subsequent stages. This approach would provide almost complete information for subsequent iterations while reducing trace sizes by orders of magnitude. In the D System we plan to explore this technique and extend it to other situations.

A sophisticated parallelizing compiler can provide extensive support for performance tuning. This support can be used to make program instrumentation more efficient, and to present performance data to the programmer in intelligent ways that are more closely tied to the constructs in the source program.

7.0 Data Mapping Assistant

Choosing the data mapping is a key decision in data-parallel programming, as the data mapping often determines the performance of the entire data-parallel program. While performance analysis tools provide information about performance bottlenecks with a given data mapping, they provide little support for comparing alternative data mappings. The choice of a good data mapping depends on many factors, including the target machine architecture, the compilation system, the problem size and the number of processors available. The possibility of dynamically remapping arrays makes this choice even harder. Thus, a tool that can automatically and quantitatively compare a variety of data mappings and explain their performance impact would be invaluable.

The problem of finding an efficient data mapping for a distributed-memory multiprocessor has been recognized and addressed by many researchers [1,6,13,18,22,34]. The presented solutions differ significantly in the assumptions that are made about the input language, the possible set of data mappings, the compilation system, and the target distributed-memory machine. Most of the previously published work performs automatic data mapping as an optimization phase inside the compiler. Typically, these systems perform data mapping optimization in two steps, namely alignment analysis followed by distribution analysis.

We do not believe that an efficient data mapping can be determined fully automatically in all cases. Therefore, the programmer should be involved in the process of choosing an efficient data mapping. A data mapping assistant should allow the programmer to partially specify a data mapping. The tool should then extend the partial specification into an efficient total data mapping. To accomplish this task, the tool has to perform interprocedural analysis. A partial specification of a data mapping may consist of mapping information for only a subset of the program's data objects or mapping information restricted to particular regions of the program. Note that one possible partial data mapping specification is no data mapping specification at all. For example, passing a pure Fortran 90 or FORTRAN 77 code to the data mapping assistant would convert the program to HPF.

An automatic data mapping assistant should support the programmer's data mapping selection at different points in the program's tuning process. Given a sequential program, the tool could be used to generate a first approximation to an efficient data mapping. Using the performance analysis tools described in Section 5.0 and Section 6.0, the programmer could select more efficient mappings for program regions that dominate the program's overall execution time, thereby overriding the data mapping assistant's suggestions. The programmer could then invoke the data mapping assistant again to generate good data mappings for program regions that have not been hand tuned. Instead of generating a total data mapping specification, the data mapping assistant should also provide information that will give the programmer insights into the tool's trade-off decisions. Such information will improve the programmer's understanding of the performance characteristics of the selected data mapping and the key alternatives.

The programmer may tune the data mapping until a good one has been found and the overall performance of the program is satisfactory. For real applications, we do not expect many iterations of the tuning process. Because the data mapping assistant is not embedded in the compiler and will be run only a few times during tuning, it can use techniques that would be considered too computationally expensive for inclusion in compilers. However, the tool must be knowledgeable about the transformations and optimizations performed by that compiler.

The data mapping assistant that is being implemented as part of the D System will determine the data mapping for a program in several steps. In the initial step, the program will be partitioned into non-overlapping segments called "phases." Data mapping search spaces will then be constructed for each phase, based on alignment and distribution analyses. From each data mapping search space, a single candidate mapping will be selected to minimize the combined cost of the selected mappings, including the cost for necessary remappings. This selection process will be based on performance estimates of the candidate data mappings and the cost of remapping [2]. Some steps in the data mapping selection process may require the solution of NP-complete problems. Our current approach is to formulate these problems as 0-1 integer programming problems and solve them using a general-purpose integer programming tool; preliminary results indicate that the computational cost is reasonable. The design of the data mapping assistant and some preliminary experimental results are discussed in detail elsewhere [3]. We envision that the programmer will interact with the tool by inspecting and manipulating the candidate mapping search spaces of each program phase. The programmer will be able to insert or eliminate data mappings. For example, to force a particular data mapping for a phase, the programmer can specify the mapping as the only element in the mapping search space of the phase.

8.0 Summary and Conclusions

In this paper, we have presented our vision for the functionality and organization of a data-parallel programming environment. We believe the primary goal of, as well as the primary challenge for, such an environment is to support the development of efficient data-parallel programs while insulating the programmer from the intricacies of the explicitly parallel code. Such support is essential because a data-parallel program will be extensively transformed during compilation into an optimized, explicitly parallel program. Understanding these transformations along with the architecture-specific details of parallelism is an unnecessarily onerous requirement on the programmer. The central purpose of portable, data-parallel languages like HPF is to eliminate this burden.

In order to achieve this goal, the environment must allow a programmer to debug and tune a data-parallel program in the presence of sophisticated code optimizations by the compiler. To do this, the environment must present information about program behavior at a level as close to the abstract programming model of the language as possible. We have identified the key components of such an environment and the functionality they must provide. These include:

- A *source-level debugger* that provides support for understanding execution behavior in the presence of large volumes of data, and also provides as much of the functionality of a sequential debugger as possible. The principal challenge in the latter case is to provide sufficient support for transparent breakpoints so that the programmer can explore intermediate states of execution.
- A *static performance analysis tool* that provides the feedback necessary for the programmer to make appropriate algorithmic choices during code development. Qualitative feedback (for example, annotations that describe the extent of parallelism and communication in a program) is particularly important because many critical aspects of performance are only available implicitly.
- A *dynamic performance analysis tool* that can support extensive performance tuning based on dynamic performance information. The tool must be able to manage the potentially huge volume of data generated by runtime measurements and present it to the programmer in terms of

the constructs in the source program. As we have discussed, this process can be simplified in important ways by exploiting the program analysis infrastructure in the compiler.

- A *data mapping assistant t*hat provides guidance in choosing an efficient mapping of arrays to processors. The choice of data mapping is a fundamental determinant of performance. A tool that can quantitatively evaluate the large number of mapping choices should prove invaluable.
- Finally, a *common interface* that provides uniform access to these tools and presents information from these tools to the programmer.

While individual tools currently exist that can perform some subset of these tasks, none meet all these goals. To do so, the tools must have access to the program analysis infrastructure that performs the compile-time analysis and transformations of the source program. This shared infrastructure must allow the tools to view, create and modify data-parallel code, and to relate the behavior of the compiled code back to the original data-parallel source program. Maintaining such a representation requires that the compiler record analysis results as well as the code transformations, thus distinguishing it from traditional compilers. While such integration with the environment adds overhead to the compilation process, such annotations are vital for meeting the needs of programmers.

The D System, which is under development in the Center for Research on Parallel Computation, will serve as a testbed for the ideas presented here. Clearly, constructing a programming environment and compiler infrastructure that meets the above criteria is difficult, and will require addressing many challenging research problems. If successful, the system will demonstrate the feasibility of providing the extensive support that will be necessary for the success of portable data-parallel languages such as Fortran D and HPF.

9.0 Acknowledgments

We gratefully acknowledge the Fortran 77D compiler group at Rice University and the Fortran 90D compiler group at Syracuse University for their helpful discussions and advice. This research was supported 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.

10.0 References

[1] J. Anderson and M. Lam. Global Optimizations for Parallelism and Locality on Scalable Parallel Machines. *Proceedings of the SIGPLAN ‘93 Conference on Program Language Design and Implementation*, Albuquerque, NM, June, 1993.

[2] V. Balasundaram and G. Fox and K. Kennedy and U. Kremer. A Static Performance Estimator to Guide Data Partitioning Decisions. *Proceedings of the Third ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, Williamsburg, VA, April, 1991.

[3] R. Bixby, K. Kennedy, and U. Kremer, Automatic Data Layout Using 0-1 Integer Programming, *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, Montreal, Canada, August, 1994

[4] G. Brooks, G. Hansen, and S. Simmons. A New Approach to Debugging Optimized Code. *Proceedings of the SIGPLAN ’92 Conference on Programming Language Design and Implementation*, San Francisco, CA, June 1992, pp. 1-11.

[5] D. Callahan and K. Kennedy. Compiling Programs for Distributed-Memory Multiprocessors. *Journal of Supercomputing*, 2, October 1988 pp. 151–169.

[6] S. Chatterjee,J.R. Gilbert, R. Schreiber, and S-H. Teng. Automatic Array Alignment in Data-Parallel Programs. *Proceedings of the Twentieth Annual ACM Symposium on the Principles of Programming Languages*, Albuquerque, NM, January, 1993.

[7] D. Coutant, S. Meloy, and M. Ruscetta. DOC: A Practical Approach to Source-Level Debugging of Globally Optimized Code. *Proceedings of the SIGPLAN ’88 Conference on Programming Language Design and Implementation*, Atlanta, GA, June 1988, pp. 125-134.

[8] A. Dinning and E. Schonberg. An Empirical Comparison of Monitoring Algorithms for Access Anomaly Detection. *Proceedings of the 1990 Symposium on Principles and Practice of Parallel Programming*, Seattle, WA, March 1990, pp. 1-10.

[9] D. Eager, J. Zahorjan, and E. Lazowska. Speedup versus Efficiency in Parallel Systems, *IEEE Transactions on Computers*, C-38(2), March, 1989, pp. 408-423.

[10] T. Fahringer and H. Zima. A Static Parameter Based Performance Prediction Tool for Parallel Programs. *Proceedings of the 1993 International Conference on Supercomputing*, Tokyo, Japan, July, 1993.

[11] G. Fox, S. Hiranandani, K. Kennedy, C. Koelbel, U. Kremer, C. Tseng, and M. Wu. Fortran D language specification. Technical Report TR90-141, Dept. of Computer Science, Rice University, December, 1990.

[12] G. Fox, M. Johnson, G. Lyzenga, S. Otto, J. Salmon and D. Walker. *Solving Problems on Concurrent Processors,* Volume 1. Prentice-Hall, Englewood Cliffs, NJ, 1988.

[13] M. Gupta and P. Banerjee. Demonstration of Automatic Data Partitioning Techniques for Parallelizing Compilers on Multicomputers. *IEEE Transactions on Parallel and Distributed Systems,* 3(2). March, 1992, pp. 179-193.

[14] J. Hennessy. Symbolic Debugging of Optimized Code. *ACM Transactions on Programming Languages and Systems*, July, 1982.

[15] S. Hiranandani, K. Kennedy and C. Tseng, Evaluation of compiler optimizations for Fortran D on MIMD distributed-memory machines. *Proceedings of the ACM 1992 International Conference on Supercomputing*, Washington, DC, July, 1992, pp. 1-14.

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

[17] K. Kennedy, K. S. McKinley and C. Tseng. Analysis and transformation in an Interactive Parallel Programming Tool. *Concurrency: Practice and Experience,* 5(7), October 1993, pp. 575-602.

[18] K. Knobe, J. Lukas and G. Steele, Jr. Data Optimization: Allocation of arrays to Reduce Communication on SIMD Machines. *Journal of Parallel and Distributed Computing,* 8(2) February, 1990, pp. 102-118.

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

[20] C. Koelbel and P. Mehrotra. Compiling global name-space parallel loops for distributed execution. *IEEE Trans. on Parallel and Distributed Systems,* 2(4), October 1991, pp. 440-451.

[21] T. LeBlanc and J. Mellor-Crummey. Debugging Parallel Programs with Instant Replay. *IEEE Transactions on Computers,* C-36(4), April 1987, pp. 471-482.

[22] J. Li and M. Chen. The data alignment phase in compiling programs for distributed-memory machines. *Journal of Parallel and Distributed Computing,* 13(4) August, 1991, pp. 213-221.

[23] J. Mellor-Crummey. On-the-fly Detection of Data Races for Programs with Nested Fork-Join Parallelism. *Proceedings of Supercomputing '91,*Albuquerque, NM, November 1991, pp. 24-33.

[24] R. Mirchandaney, J. Saltz, R. Smith, D. Nicol and K. Crowley. Principles of run-time support for parallel processors. *Proceedings of the Second International Conference on Supercomputing,* St. Malo, France, July 1988.

[25] R. Netzer and B. Miller. Optimal Tracing and Replay for Debugging Message-Passing Parallel Programs. *Proceedings of Supercomputing '92,* Minneapolis, MN, November 1992, pp. 502-511.

[26] R. Netzer and S. Ghosh. Efficient Race Condition Detection for Shared-Memory Programs with Post/Wait Synchronization. *Proceedings of the 1992 Intl. Conference on Parallel Processing,* St. Charles, IL, August 1992.

[27] P. Pineo and M. Soffa. Debugging Parallelized Code Using Code Liberation Techniques. *Proceedings of the ACM/ONR Workshop on Parallel and Distributed Debugging*, Santa Cruz, CA, May, 1991, pp.108-119.

[28] D. Reed, R. Aydt, T. Madhyastha, R. Noe, K. Shields, and B. Schwartz. *An Overview of the Pablo Performance Analysis Environment*. UIUCCS Technical Report, University of Illinois, November, 1992.

[29] A. Rogers and K. Pingali. Process decomposition through locality of reference. *Proceedings of the SIGPLAN 89 Conference on Programming Language Design and Implementation*, Portland, Oregon, June 1989.

[30] M. Rosing, R. Schnabel and R. Weaver. The DINO Parallel Programming Language. *Journal of Parallel and Distributed Computing,* 13(1), September 1991, pp. 30-42.

[31] G. Steele. Making Asynchronous Parallelism Safe for the World. *Proc. of the 1990 Symposium on the Principles of Programming Languages,* San Francisco, CA, January 1990, pp. 218-231.

[32] Thinking Machines Corporation. *Prism Reference Manual*, 1992.

[33] C.-W. Tseng. *An Optimizing Fortran D Compiler for MIMD Distributed-Memory Machines.* Ph.D. Dissertation, Rice University, January 1993.

[34] S. Wholey. Automatic Data Mapping for Distributed-Memory Parallel Computers. *Proceedings of the 1992 ACM International Conference on Supercomputing* Washington, DC, July 1992.

[35] J. Wu, J. Saltz, S. Hiranandani and H. Berryman. Runtime compilation methods for multicomputers. *Proceedings of the 1991 International Conference on Parallel Processing,* St. Charles, Illinois, August 1991.

[36] H. Zima, H.-J. Bast and M. Gerndt. SUPERB: a tool for semi-automatic MIMD/SIMD parallelization. *Parallel Computing,* 6, 1988, pp. 1-18.