# An Integrated Compilation and Performance Analysis Environment for Data Parallel Programs

*Vikram S. Adve  Jhu-Chun Wang*
*John Mellor-Crummey  Daniel A. Reed*
*Mark Anderson  Ken Kennedy*

**CRPC-TR94513-S**
**December, 1994**

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

# An Integrated Compilation and Performance Analysis Environment for Data Parallel Programs[*]

Vikram S. Adve John Mellor-Crummey
Mark Anderson Ken Kennedy

Center for Research on Parallel Computation
Rice University
Houston, Texas 77251-1892

Jhy-Chun Wang Daniel A. Reed

Department of Computer Science
University of Illinois
Urbana, Illinois 61801

**Abstract**

To support the transition from programming languages in which parallelism and communication are explicit to high-level languages that rely on compilers to infer such details from data decomposition directives, tools for performance analysis require increased sophistication and integration with other components in the programming system. We explore integration of performance tools with compilers for data parallel languages by integrating the Rice Fortran 77D compiler and the Illinois Pablo Environment. This integration permits analysis and correlation of the compiler-generated code's dynamic behavior and performance with the original data parallel source code. We expect that our strategy can serve as a model for integration of other data parallel compilers and performance tools.

## 1 Introduction

High-level, data parallel languages such as High Performance Fortran and Fortran D [4] have attracted considerable attention because they offer a simple and portable programming model for parallel, scientific programs. In such languages, programmers specify parallelism abstractly using data layout directives; a compiler uses these directives as the basis for synthesizing a program with explicit parallelism and interprocessor communication.

Although these languages offer a convenient model for parallel program development, the primary motivation for parallel processing is high performance. Constructing a high-performance parallel program requires a cycle of experimentation and refinement in which one first identifies the program components responsible for the bulk of the execution time and then modifies the program in the hope of improving its performance. For this cycle to be effective and unobtrusive to practicing scientists, not only must performance data be accurate, it must be directly related to the source program.

The adoption of high-level, data parallel languages and sophisticated parallelizing compilers means that an application software developer's mental model of a program and the actual code that executes on a particular parallel system are quite different, and the relation between execution dynamics and data parallel source code is often unclear. The advantages of a high-level programming model will be fully realized only if both compilers and performance analysis tools seamlessly support the cycle of code development and performance tuning by explaining performance at the level of the abstract, high-level source program.

[*] Supported in part by ARPA contracts DAVT63-91-C-0029, DABT63-93-C-0040 and DABT63-92-C-0038, and by NSF grant ASC92-12369 and NSF Cooperative Agreement Number CCR-9120008.

Unfortunately, current performance analysis and visualization tools, are targeted at the collection and presentation of program performance data when the parallelism and interprocessor communication are explicit (e.g. in message-passing codes). For high-level parallel languages, such tools would capture and present dynamic performance data in terms of primitive operations (e.g. communication library calls) in the compiler-generated code; clearly, this falls short of the ideal. Likewise, current data parallel compilers provide minimal information about the translation from data parallel source to explicitly parallel output, and it is rarely clear if poor performance is due to compiler limitations or to limited data parallelism.

To support source-level performance tuning of programs in high-level parallel languages, compilers and analysis tools must cooperate to integrate information about the program's dynamic behavior with compiler knowledge about the mapping from the high-level source to the low-level, explicitly parallel code. Furthermore, performance analysis tools can use other types of compiler knowledge (e.g., the impact of source code constructs like array layouts and data dependences on the parallelism and communication in the synthesized code) to provide more sophisticated explanatory feedback to the programmer. The D System group at Rice University and the Pablo group at the University of Illinois are jointly developing an integrated compilation and performance analysis system based on the Rice Fortran 77D compiler and the Illinois' Pablo performance environment.

The Fortran 77D compiler translates a Fortran 77 program annotated with data layout directives into an explicitly parallel *single-program multiple data* (SPMD) message-passing node program for a distributed memory machine. The compiler performs a number of complex optimizing transformations, including hoisting communication out of loops, aggregating messages between the same pair of processors, moving message sends above preceding code blocks that do not modify the data being sent, and introducing collective communication such as broadcasts or reductions when required. When data dependences prevent full parallelism, the compiler often can achieve partial parallelism by pipelining the computation.

The Pablo environment's principal components are an extensible data capture library [10], a data meta-format [3] for describing the structure of performance data records without constraining their contents, and a user-extensible graphical data analysis toolkit [8]. The data capture library can trace, count, or time code fragments as well as capture entry to and exit from procedures, loops, and message-passing library calls. A group of library extension interfaces allows instrumentation software developers to incrementally extend the library's functionality. The key source of the flexibility in Pablo's components is a flexible performance data meta-format that separates data semantics from data structure. The Pablo self-defining data format (SDDF) is a data description language that specifies both the structure of data records and data record instances. SDDF provides the ability to define and process new performance data records as needed.

This paper describes our approach in developing an integrated environment based on the Fortran 77D compiler and Pablo, and preliminary experiences from the integration. The paper first motivates the need for compiler support for performance analysis and outlines the limitations of current data parallel compilers and performance analysis tools. We then describe the design and initial implementation of the integrated environment which uses SDDF as an interchange format for sharing information between the Fortran D compiler and the Pablo performance analysis software. Finally, we describe preliminary experiments that illustrate the advantages of the integration, compare with related work, and conclude with a brief discussion and directions for future research.

## 2 Performance Correlation Example

The need for integrated compilation systems and performance tools is best illustrated by considering the steps needed to analyze the performance of a simple Fortran D code using decoupled compilation and performance analysis tools. As an example, we use the Erlebacher code, an 800 line, ten procedure benchmark written by Thomas Eidson at ICASE. Erlebacher performs 3-D vectorized tridiagonal solves using Alternating-Direction-Implicit (ADI) integration to solve partial differential equations. Figure 1 shows a portion of the Fortran D data declarations and one loop of the source code.

The Rice Fortran D compiler translates the code fragments in Figure 1 into the parallel, message-passing code for the Intel iPSC/860 shown in Figure 3. The compiler distributes the third dimension of array `duz` in equal blocks, and adds overlap areas to hold off-processor values ($z = 0$ and $z = 9$) in order to simplify the generated code. It then distributes the iterations of the $k$ loop across the processors using the owner-computes rule: processor $P$ $(0 \leq P \leq 7)$ executes iterations $8P + 1 \leq k \leq 8P + 8$. Communication is then required to satisfy references to non-local sections of the array, as depicted in Figure 2. To achieve partial parallelism without excessive communication overhead, the compiler strip-mines the $i$ loop in blocks of 8 iterations and then generates code to send and receive blocks of 8 elements at a time at the left boundary XY planes of each processor's local partition. It also inserts a broadcast to satisfy the source reference `duz(i,j,64)`, and hoists the broadcast outside the $i$ and $j$ loops.

Comparing Figure 1 to Figure 3 highlights the difficulty of the performance analysis task faced by a Fortran D application developer. There is a large semantic gap between the data parallel, global name space, programming model of Figure 1 and the SPMD, message-passing model of Figure 3. Because the goal of data-parallel languages is to insulate software developers from the idiosyncrasies of message passing, performance tuning should not require them to understand the details of the compiler-generated code.

Unfortunately, current performance instrumentation and analysis tools [9, 8, 6, 11] for distributed memory parallel systems only capture and present performance data from the generated Fortran 77 code of Figure 3. For example, Pablo performance displays for this program (shown in Figure 4) clearly show the performance characteristics of the message-passing program, including the volume of communication traffic, processor utilizations, and the staircase communication pattern arising from nearest neighbor communication. While these displays are useful, they fail to help the programmer understand how these performance characteristics relate to the original Fortran D source and the underlying causes of these performance characteristics. Without this understanding, the programmer will be unable to improve the program's performance without laboriously attempting to match compiler output to data parallel source code.

Many of the problems described above can be attributed to an unnecessary separation of compilation and performance analysis. Integrating the two has many advantages. First, the compiler can emit instrumented code, and can potentially use its extensive knowledge of parallelism and communication in the SPMD program to make the instrumentation less intrusive and less voluminous. Second, and most important, the compiler can provide extensive information to the performance tool about the mapping between source and SPMD code, as well as semantic analysis results about the impact of source code constructs on parallelism and communication. Finally, the compiler can exploit dynamic performance data to generate more efficient code, as described elsewhere [1].

# 3 Integrating Compilation and Performance Analysis

As illustrated in §2, a simple composition of compiler and performance tools is insufficient to support high-level performance analysis and software tuning of Fortran D programs. An integrated approach can provide two types of information not normally available to either traditional, stand-alone performance tools or compilers: (1) the mappping information describing how to assign measured performance costs to high-level source code objects such as data references, arrays, decompositions, loops and procedures; and (2) semantic information describing the impact of source code constructs such as array layouts and data dependences on parallelism and communication. The first is essential for presenting performance data to the programmer in terms of the data parallel source code, as do standard performance tools for sequential and explicitly parallel languages. The second can be used to provide more sophisticated, explanatory feedback to the programmer, describing the causes of performance bottlenecks in the program at a semantic level. Furthermore, as discussed elsewhere, in an integrated system a data parallel compiler could exploit dynamic performance measurements to optimize future compilations [1].The remainder of this section describes the integration of the Rice Fortran D compiler and the Illinois' Pablo performance environment to form a prototype programming environment that supports performance analysis of regular, data parallel programs written in Fortran D. After an overview of the system organization, the section describes the two major aspects of the integration: instrumentation of the explicitly parallel compiled code, and correlation of the instrumented information with the Fortran D source program. Preliminary examples of performance feedback and visualization provided in the tool are described in §4.

## 3.1 System Organization

Figure 5 shows an overview of the current integration. The system uses the Fortran D compiler to instrument the compiled code and to emit data on compiler analysis and transformations, uses Pablo SDDF as a flexible medium of data interchange between the compiler and Pablo, the Pablo instrumentation software's extension interfaces for capturing dynamic performance data, and a new software toolkit to combine the static and dynamic performance data and relate it to the Fortran D source.

To integrate compile-time and runtime performance information, three classes of information are recorded in SDDF data records: static, dynamic, and symbolic. The *static data* consists of information recorded by the Fortran D compiler during program compilation. The *dynamic data* is data known only during execution, such as the durations of procedure invocations and message-passing calls. Each dynamic record includes the tag of a static data record; this is a starting point for correlating the dynamic data with static information. Finally, the *symbolic data* is data that may be known either at compile-time or runtime (depending on the source code), e.g., the size of a message, or the distribution of an array in a phase of the program. If such a value is known at compile-time, the corresponding symbolic record is preserved in the static SDDF file; otherwise it is emitted at runtime. A common interface provides uniform access to this information regardless of where individual symbolic records are stored.

## 3.2 Program Instrumentation

The Pablo instrumentation library [10] can count or trace dynamic events and time intervals; specialized instrumentation supports tracing of procedure calls, loops, and message passing library calls. The library has been augmented, via its extension interfaces, to support integration with the Fortran D compiler. Key changes made to the library include: (1) information that may be available statically or symbolically (e.g., message size) is no longer captured during execution, (2) each dynamic record includes the tag of a static record that provides information for data correlation, and (3) a new dynamic record type is introduced to record symbolic values. In addition to tracing individual message passing events, the Pablo instrumentation can also generate a statistical summary of message passing activity within a selected code fragment, by analyzing the communication trace events as they are generated. Summarization is most commonly used to reduce instrumentation data volume when the compiler was unable to hoist message passing calls from a loop.

For each type of communication primitive (i.e., send, receive, or collective communication), the summary data includes the minimum, maximum, mean, sum, and standard deviation for message sizes and communication durations, as well as a histogram of these quantities. Compared with ordinary message tracing where one pair of records is generated for each call to the message passing library, summaries dramatically reduce the volume of captured performance data, albeit with some loss of information on dynamic behavior.

The Fortran D compiler decides where instrumentation must be inserted in the synthesized message-passing code, and inserts calls to the appropriate Pablo instrumentation routines. Instrumentation currently supported by the compiler includes tracing entry/exit events for procedure calls, loop nests, message library calls, and symbolic values unknown at compile-time. Summary tracing for messages is not yet invoked by the compiler.

## 3.3 Data Correlation

In addition to instrumenting the synthesized code, the Fortran D compiler records both mapping and semantic information relevant to performance analysis in a static SDDF file. Below, we describe the static information in more detail and show how it is combined with information in dynamic records to correlate dynamic performance data with the Fortran D source code.

### 3.3.1 Mapping Information

The mapping information in the static SDDF file specifies how the procedures, procedure calls, source loops, data distribution statements, data references, and data declarations in the source file relate to one another and to the corresponding constructs in the compiler-synthesized SPMD code.

For procedures, the static SDDF file describes the locations of procedure call sites in the source program, and the locations of the corresponding procedures. Separate static records are generated for each call site and each procedure, as illustrated in Figure 6. The dynamic records generated during an instrumented procedure call point to the corresponding call site record. Thus, the measured procedure call lifetimes can be attributed to individual call sites, averaged across all the processors, and also aggregated for the individual procedures across all its call sites, as typically provided by profiling tools for sequential programs.

For messages, the static SDDF records provide a variety of mapping information including (a) the set of non-local data references satisfied by each message, and the reverse mapping as well; (b) the data layout (decomposition, alignment and distribution) of each distributed array communicated by each message; and (c) the messages generated for each dynamic redistribution or realignment. [1] Dynamic records for each message send point to the corresponding static record for that message send, as shown in Figure 6. In general, one static record is generated for each message library call in the SPMD code, and the static records for receive and message-wait operations each point to the corresponding static message send record. Collectively, the static message records identify the source references, arrays, and layout directives that are responsible for the corresponding communication overhead in the parallel program.

The loop information in the static SDDF file describes the source loop (or loop nests) that are associated with a particular loop (or loop nest) in the SPMD program. The current design includes static records for loop nests as well as individual loops (at the source and SPMD level) because some transformations such as loop distribution and loop fusion operate on (pairs of) individual loops, while others such as iteration partitioning, loop interchange or fine- or coarse-grain pipelining operate on entire loop nests. The design of this portion of the static file is preliminary, however, and is yet to be implemented in the compiler.

### 3.3.2 Semantic Information

Perhaps the most important semantic information available with compiler support is information about the pattern of access to remote data in the program. The static message send records include tags for symbolic records that specify the lower bound, upper bound, and step for each dimension of each array section communicated in the associated message. If any of these symbolic values is unknown at compile-time, the compiler inserts instrumentation code to record it each time it changes during program execution. By correlating the static, symbolic, and dynamic records for each message, it is possible to compute the exact array sections communicated, along with the dynamic frequencies of communication.

The compiler also records the set of cross-processor dependences that cause each communication event. Again, the static message send record includes tags pointing to individual dependence records, each giving the locations of the source and sink data references of the dependence, as illustrated in Figure 6. This information about the data flow in the program (as determined by the compiler) is fundamental in understanding the causes of communication in an environment where the compiler transforms and optimizes the program based on its analysis of data dependences.

The static file also includes provisions to identify the additional calls to runtime buffering and unbuffering routines synthesized by the compiler for each message send and receive, such as in Figure 3. For this purpose, the static message send record includes two fields to point to the procedure call records for the corresponding calls to the buffering routines, as illustrated in Figure 6 for the buffering call. This information can allow the performance tool to present a more accurate picture of the overall overhead due to communication.

Finally, the static SDDF records provide support to determine the distribution or alignment of an array that is active in any interval of execution, in the presence of dynamic redistribution and realignment. In the future, this information will be used to compute the frequency and cost of array redistributions.

[1] The Fortran D compiler does not currently support ynamic redistribution or realignment, but a provision for this information is included in the static records for future use.

# 4 Preliminary Experiences

In §3 we described the mechanisms for data correlation but not the results of that correlation. Below, we describe our experiences combining compile-time data on program transformations with dynamic data on program execution dynamics.

## 4.1 Performance Data Correlation and Presentation

Our current data correlation software calculates many performance metrics for an execution of a Fortran D program. The specificity of these metrics ranges from aggregate (i.e., whole program) through procedure, procedure call site, and individual source code line and include both execution times and array reference data.

For each procedure call site and loop nest, the performance metrics include the inclusive and exclusive procedure and loop lifetimes (i.e., including and excluding descendent procedures and nested loops) and the inclusive and exclusive message passing overhead. For each metric, the statistics include the duration on each processor and the maximum, minimum, mean, and standard deviation across the processors. Using the measured communication costs, we map compiler-synthesized message passing back to array references in the original data parallel source code, showing the volume and cost of remote data references, and the overhead for message packing and unpacking.

Currently, our system support visualization of only a small subset of the performance information computed. Our preliminary efforts have focused on integrating static and dynamic performance data with the Rice D editor; however, performance information could be presented instead with a stand alone Each of these approaches has potential advantages and disadvantages. Below, we describe our experiences so far with our integration of performance information in the D editor; in §6 we describe integration plans for the Pablo performance environment.

Figure 7 shows the D Editor, which forms the primary user interface to the D System. The D Editor is a structured editor for Fortran that provides users feedback about the parallelism and communication in a program.[2] The `Overview` pane at the top of the figure shows a "compressed" view of the Erlebacher source code, with only the subroutines and loop nests shown. The other panes are used to show the dependences, communication and array layouts for the selected loop (the same as the loop shown in Figure 1). Not shown is a second window that displays the source code, highlights non-local references, and uses arrows to display cross-processor, loop-carried data dependences (or optionally other types of dependences).

One of the simplest and most useful displays of dynamic performance data is a profile of where time is spent during program execution. The overview pane in Figure 7 uses colored bars (shown here in black-and-white) to present the computation and communication cost of the individual procedures and loops, relative to the full program. The bars in the left column show procedure invocation lifetimes, where the light bars represent the inclusive procedure lifetime (i.e., including this procedure and other procedures it may call), and dark bars represent the exclusive procedure lifetime. The bars are normalized relative to the program's total computation time. The bars in the right column show similar metrics for the communication overhead

[2] The dynamic information in the editor is obtained and correlated with source code exclusively using the static mapping file and the dynamic trace files, exactly as would be done in an off-line tool.

due to compiler-synthesized message passing. In addition to this dynamic information, the editor also shows the nature and location of communication due to the selected loop, the dependences causing communication, the layouts of the key arrays in the loop, as well as the specific references that require the communication (the latter is shown by highlighting the references in the source window, not included in the figure).

Even with the limited correlation of static and dynamic data represented by the visualization of Figure 7, the system is able to convey to the programmer the location, relative impact, and causes of communication overhead, all in terms of the source program most familiar to the programmer. The system uses a subset of the mapping between messages and source references described in the static SDDF file to attribute the communication overhead to specific statements and (by aggregating these) to specific loops and procedures at the source level.

Using the static information from the compiler, the editor also directly shows the specific loop-carried, cross-processor dependence that inhibited parallelism and required the pipelined communication. In addition, it also shows the pattern of communication associated with that loop (in this case, the pipelined shift described in §2), the array sections involved in the communication, and the fraction of total execution time and of total communication time spent in this communication.

In addition to visualizing performance information in the D editor, we have used external visualization tools from NCSA to display array access locality patterns. Our system communicates with these tools by exporting remote array reference counts and costs in NCSA's HDF data format.

## 4.2 Data Correlation Overhead

By definition, software instrumentation perturbs program execution; one must insure that compiler-synthesized instrumentation does not so perturb nominal application behavior that salient behavioral features are lost. To quantify the overhead for our dynamic instrumentation and to assess the potential advantages of compiler support for instrumentation, we conducted a series of experiments with the Erlebacher benchmark with and without symbolic information and message summarization.

Analysis of the results showed that execution overhead for dynamic instrumentation, excluding symbolic records and excluding buffer extraction costs, ranged from about 35 percent on 4 processors to 140 percent on 32 processors of an Intel iPSC/860. The absolute magnitude of the overhead can in large part be attributed to the fine-grain pipelining (i.e., frequent, small messages) synthesized by the Fortran D compiler. However, because the Pablo data capture library records the cost of each type of instrumentation and the overhead for performance buffer extraction, in many, though not all, cases it is possible to recover relative event times even when the absolute instrumentation perturbation is large.

Ameliorating the effects of large numbers of dynamic records is the goal of message summarization. On 32 processors, message summarization reduced the instrumentation overhead by over 300 percent (from 225K records to 7K records). Moreover, by carefully applying message summarization to particular message library call sites, little information is lost.

Our instrumentation approach also enables us to collect rich information about remote data access patterns using symbolic records describing the bounds of array sections communicated in each message. However, naive collection of this information comes at a high run-time cost. For example, symbolic records for the loop-varying array section bounds associated with messages in Erlebacher's pipelined loop nests accounted for an additional 101K records. The added value of this symbolic information is nevertheless important; we are exploring approaches based on compile time and/or run time summarization to reduce the cost of collecting such array locality information.

## 5 Related Work

Few tools for dynamic performance analysis of parallel programs support abstract parallel languages, and few compilers export the requisite information needed to correlate dynamic performance data with the program source code. Notable exceptions include include Prism [12] and NV [5] for CM-Fortran, Forge90 [2] for Fortran 90 and HPF, and the MPP-Apprentice performance tool, which supports C, Fortran and Fortran 90 on the Cray T3D [7]. The CM Fortran compiler lacks aggressive inter-statement optimization, making the mapping of dynamic performance data to source lines for Prism and data objects for NV straightforward. Forge90 supports performance analysis at the level of compiler-generated SPMD code; its performance annotations show the cost of invocations of their communication library. MPP Apprentice is the most similar to our work as it uses compiler support to track the effect of complex optimizations and maintains a mapping of basic blocks in the source to basic blocks in the optimized code. The fundamental difference between the MPP Apprentice and our work is that the MPP Apprentice focuses on tracking scalar rather than parallel optimizations.

## 6 Observations and Future Directions

With high-level, data parallel languages and sophisticated parallelizing compilers, the compiled code that executes on a parallel system is usually quite different from the original, data parallel program. Because current performance evaluation tools capture performance data from the executing code and current compilers hide the details of source code transformations, relating dynamic performance data to high-level source code is extremely difficult.

In this paper we discussed the integration of the University of Illinois' Pablo performance environment and Rice University Fortran D compiler. The D compiler generates instrumented code and emits information on code transformations and compiler analyses in the Pablo performance data meta-format. During execution, the instrumented program generates dynamic performance data that is captured and reduced by the Pablo performance analysis software. The resulting dynamic data can be correlated with the original Fortran D code using the mapping information previously recorded at compile time.

To date, several key aspects of the integrated environment have been implemented, though much work remains. The mapping information relates dynamic performance data to high-level source code, providing the analog of detailed procedure and loop profiles, both for execution time and communication overhead incurred in the corresponding regions of the source code. By correlating interprocessor communication with array sections, the pattern of remote array references and their costs can be related to specific Fortran D code regions.

Integrating mapping support with the Fortran D compiler has proved instructive. The most basic and perhaps obvious lesson from our experience is that it is extremely important to consider performance support requirements throughout the design and development of the compiler. The performance support requires information from virtually all phases of the compiler and at many different levels. A pervasive difficulty we have encountered is that the lifetime of data needed for compiling is often different from that needed for performance analysis, thus necessitating new mechanisms to retrieve or preserve the data. Another difficulty is that the analysis or implementation of a single logical transformations is sometimes divided among different parts of the program, significantly complicating the task of recording the transformation and the corresponding mapping.

The current visualization support in the environment is extremely preliminary, and a comprehensive performance data browser is under development. The browser will exploit the Pablo environment's graphics displays to show performance metrics over user-specified execution intervals, support performance queries on the program, and in the future, handle queries about the performance of hypothetical program executions via performance prediction. Another priority in our future work is to implement the remaining static SDDF records within the compiler. These include correlation of buffer copying calls with the corresponding message calls, and the static records for describing mapping information between source and SPMD loops. We also are exploring techniques for further reduction of the dynamic instrumentation overhead for recording rapidly varying array section bounds inside loop nests.

The infrastructure integrating Pablo and the D compiler has been designed to be as independent of performance tool and compiler specifics as possible. We believe it is possible to transfer the integration infrastructure to other compilers or performance tools with modest effort.

# References

[1] Adve, V. S., Koelbel, C., and Mellor-Crummey, J. M. Performance Analysis of Data-Parallel Programs. Tech. Rep. CRPC-TR94405, Center for Research on Parallel Computation, Rice University, May 1994.

[2] Applied Parallel Research. *Forge 90 Distributed Memory Parallelizer: User's Guide*, version 8.0 ed. Placerville, CA, 1992.

[3] Aydt, R. A. SDDF: The Pablo Self-Describing Data Format. Tech. rep., Department of Computer Science, University of Illinois, Apr. 1994.

[4] Hiranandani, S., Kennedy, K., and Tseng, C.-W. Preliminary Experiences with the Fortran D Compiler. In *Proceedings of Supercomputing '93* (Nov. 1993), Association for Computing Machinery.

[5] Irvin, R. B., and Miller, B. P. A Performance Tool for High-Level Parallel Languages. Tech. Rep. 1204, University of Wisconsin-Madison, Jan. 1994.

[6] Miller, B. P., Clark, M., Hollingsworth, J., Kierstead, S., Lim, S.-S., and Torzewski, T. IPS-2: The Second Generation of a Parallel Program Measurement System. *IEEE Transactions on Computers 1*, 2 (Apr. 1990), 206–217.

[7] Pase, D. MPP Apprentice: A Non-Event Trace Performance Tool for the CRAY T3D. Presentation at the Workshop on Debugging and Performance Tuning for Parallel Computing Systems, Oct. 1994.

[8] Reed, D. A. Performance Instrumentation Techniques for Parallel Systems. In *Models and Techniques for Performance Evaluation of Computer and Communications Systems*, L. Donatiello and R. Nelson, Eds. Springer-Verlag Lecture Notes in Computer Science, 1993, pp. 463–490.

[9] Reed, D. A. Experimental Performance Analysis of Parallel Systems: Techniques and Open Problems. In *Proceedings of the 7th International Conference on Modelling Techniques and Tools for Computer Performance Evaluation* (May 1994), pp. 25–51.

[10] Reed, D. A., Aydt, R. A., Noe, R. J., Roth, P. C., Shields, K. A., Schwartz, B. W., and Tavera, L. F. Scalable Performance Analysis: The Pablo Performance Analysis Environment. In *Proceedings of the Scalable Parallel Libraries Conference*, A. Skjellum, Ed. IEEE Computer Society, 1993, pp. 104–113.

[11] Ries, B., Anderson, R., Auld, W., Breazeal, D., Callaghan, K., Richards, E., and Smith, W. The Paragon Performance Monitoring Environment. In *Proceedings of Supercomputing '93* (Nov. 1993), Association for Computing Machinery, pp. 850–859.

[12] TMC. *Prism User's Guide, V1.2.* Thinking Machines Corporation, Cambridge, Massachusetts, Mar. 1993.

```
      parameter (n$proc = 8)

      real duz(64, 64, 64)
c     decomposition d(64,64,64)
c     align duz with d
c     distribute d(:,:,block)

c     <CODE ELIDED>

      do 50 j=1,64
       do 50 i=1,64
        do 50 k=64-2,1,-1
          duz(i,j,k) = duz(i,j,k)
     *                 - c(k) * duz(i,j,k+1)
     *                 - e(k) * duz(i,j,64)
 50   continue
```

**Figure 1**: Erlebacher source code fragment

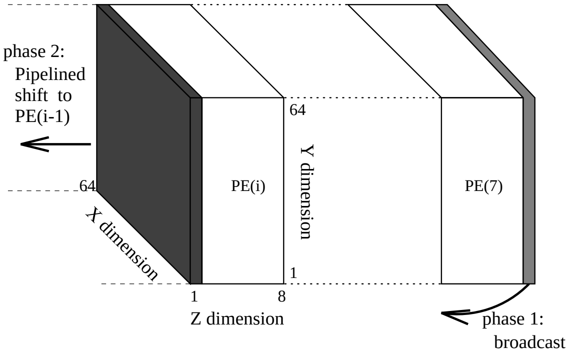


**Figure 2**: Array partitions and communication

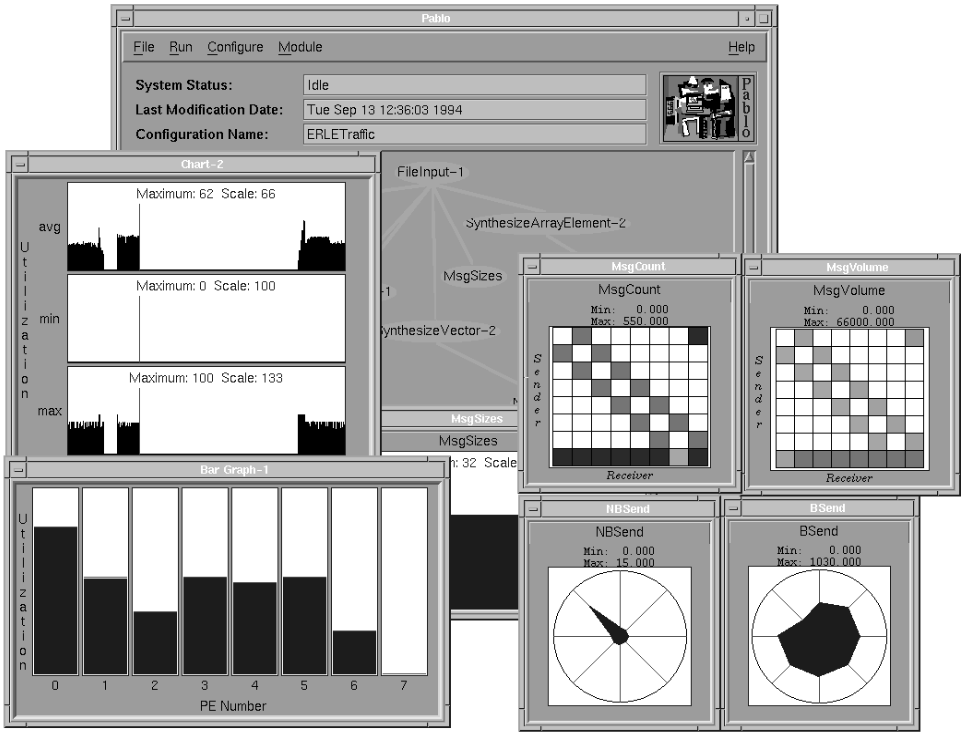
```
      real duz(64, 64, 0:9)

C     --<< Fortran D initializations >>--
      lb$1 = max((my$p * 8) + 1, 2) - (my$p * 8)
      ub$1 = min((my$p + 1) * 8, 64 - 2) - (my$p * 8)
      ub$2 = min((my$p + 1) * 8, 64 - 1) - (my$p * 8)
      off$0 = my$p * 8

C     <CODE ELIDED>

       if (my$p .eq. 7) then
C         <CODE FROM PREVIOUS LOOP NEST ELIDED>
C
C         PHASE ONE
C
C        --<< Broadcast duz(1:64, 1:64, 8) >>--
          call buf3D$r(f, 1, 64, 1, 64, 0, 9, 1, 64, 1,
     *                 1, 64, 1, 8, 8, 1, r$buf1(1))
          call csend(113, r$buf1, 4096 * 4, -1, my$pid)
       else
C         --<< Recv duz(1:64, 1:64, 64) >>--
          call crecv(113, r$buf1, 4096 * 4)
       endif
C
C      PHASE TWO
C
       do j = 1, 64
        do i$ = 1, 64, 8
          i$up = i$ + 7
C         --<< Recv duz(i$:i$up, j, 9) >>--
          if (my$p .lt. 7) then
            call crecv(112, duz(i$,j,9), 8*4)
          endif
          do i = i$, i$up
           do k = ub$1, 1, -1
              k$glo = k + off$0
              duz(i, j, k) = duz(i,j,k)
     *                       - c(k$glo) * duz(i,j,k+1)
     *                       - e(k$glo) * r$buf1(j*64 + i - 64)
           enddo
          enddo
C         --<< Send duz(i$:i$up, j, 1) >>--
          if (my$p .gt. 0) then
            call csend(112, duz(i$,j,1), 8*4,
     *                 log2phys(my$p-1), my$pid)
          endif
        enddo
       enddo
```

**Figure 3**: Generated Fortran 77 fragment



System Status: Idle

Last Modification Date: Tue Sep 13 12:36:03 1994

Configuration Name: ERLETraffic

**Figure 4**: Erlebacher Dynamic Behavior (Pablo Performance Environment)

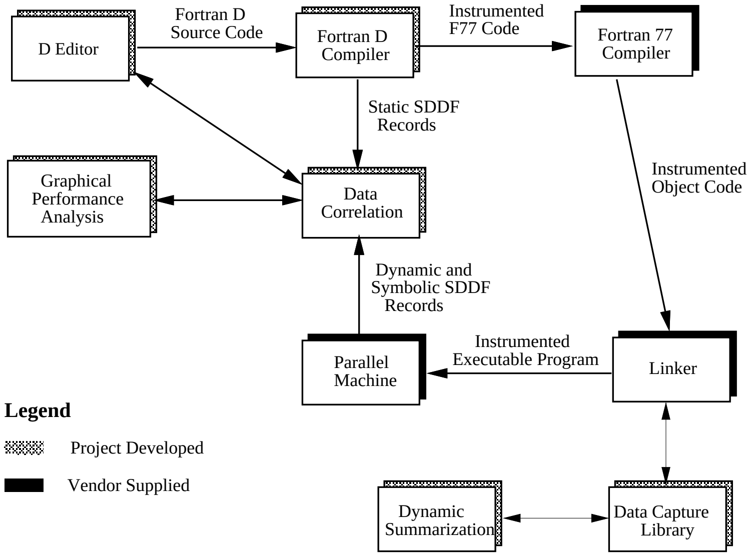


**Figure 5**: Fortran D Compiler and Pablo Performance Environment Integration

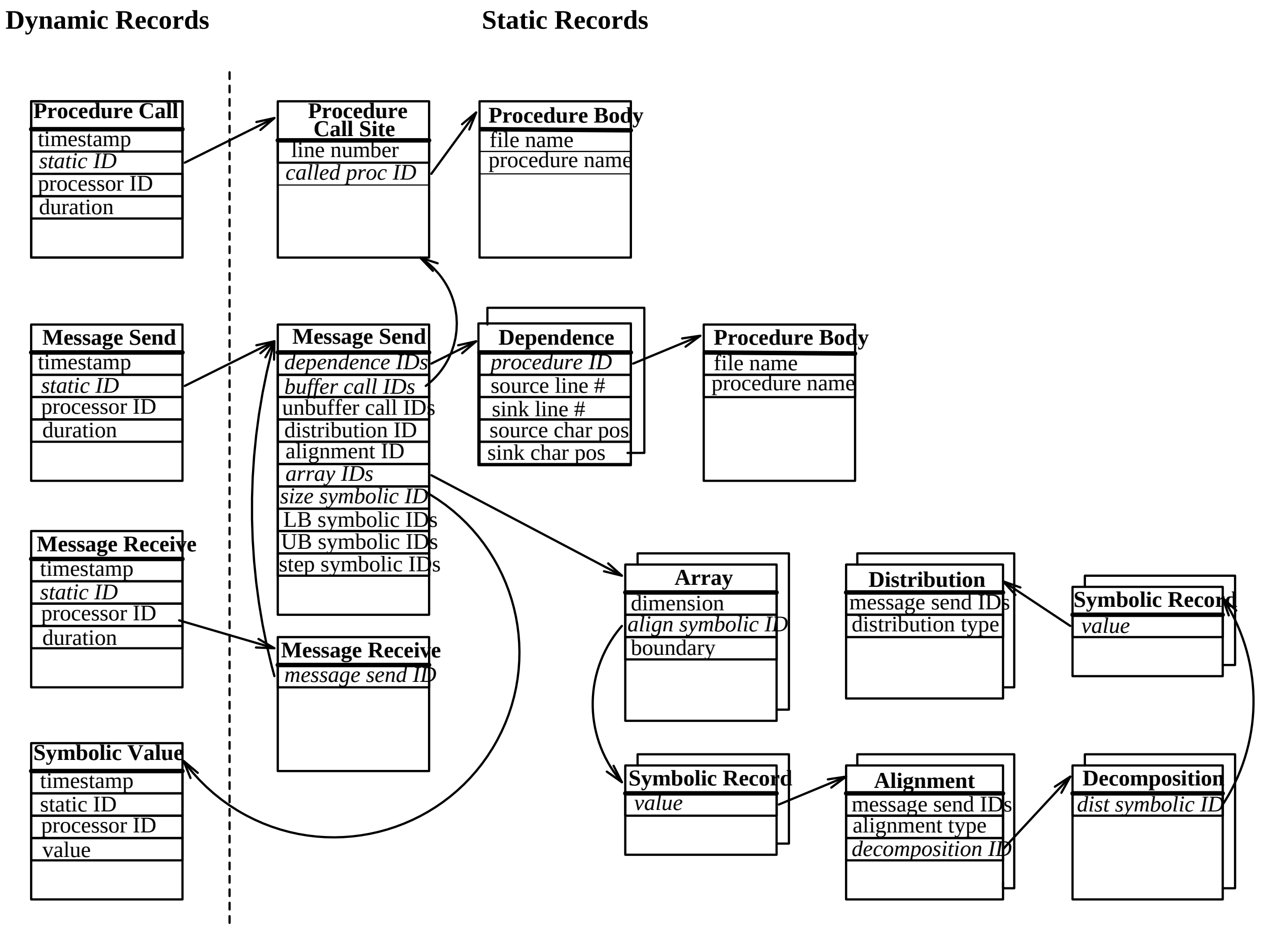
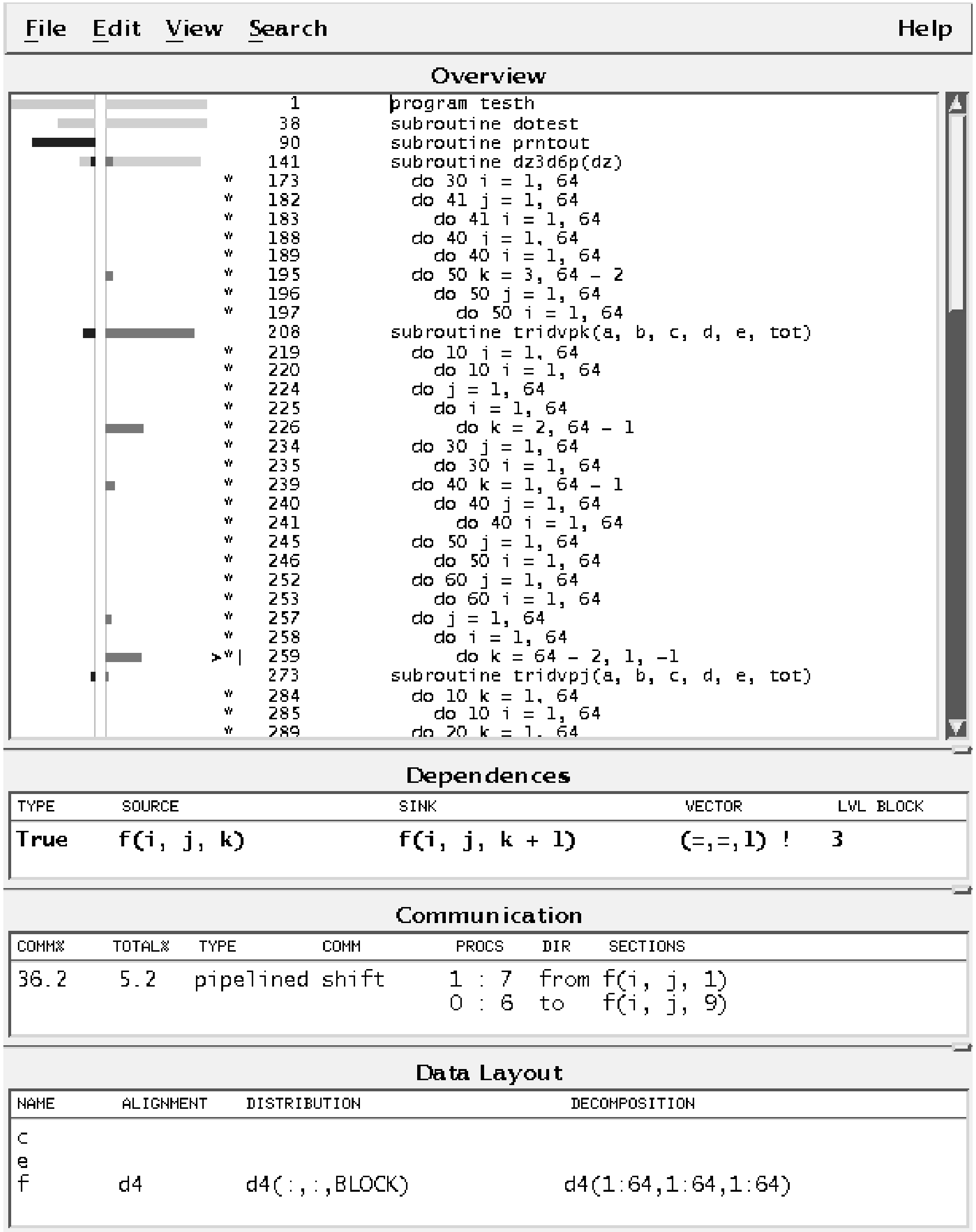


**Figure 6**: Fortran D Source Code Mapping



```
                 1        program testh
                38        subroutine dotest
                90        subroutine prntout
               141        subroutine dz3d6p(dz)
         *     173          do 30 i = 1, 64
         *     182          do 41 j = 1, 64
         *     183            do 41 i = 1, 64
         *     188          do 40 j = 1, 64
         *     189            do 40 i = 1, 64
         *     195          do 50 k = 3, 64 - 2
         *     196            do 50 j = 1, 64
         *     197              do 50 i = 1, 64
               208        subroutine tridvpk(a, b, c, d, e, tot)
         *     219          do 10 j = 1, 64
         *     220            do 10 i = 1, 64
         *     224          do j = 1, 64
         *     225            do i = 1, 64
         *     226              do k = 2, 64 - 1
         *     234          do 30 j = 1, 64
         *     235            do 30 i = 1, 64
         *     239          do 40 k = 1, 64 - 1
         *     240            do 40 j = 1, 64
         *     241              do 40 i = 1, 64
         *     245          do 50 j = 1, 64
         *     246            do 50 i = 1, 64
         *     252          do 60 j = 1, 64
         *     253            do 60 i = 1, 64
         *     257          do j = 1, 64
         *     258            do i = 1, 64
       >*|     259              do k = 64 - 2, 1, -1
               273        subroutine tridvpj(a, b, c, d, e, tot)
         *     284          do 10 k = 1, 64
         *     285            do 10 i = 1, 64
         *     289          do 20 k = 1, 64
```

Dependences

| TYPE | SOURCE | SINK | VECTOR | LVL BLOCK |
|---|---|---|---|---|
| **True** | **f(i, j, k)** | **f(i, j, k + 1)** | **(=,=,1) !** | **3** |

Communication

| COMM% | TOTAL% | TYPE | COMM | PROCS | DIR | SECTIONS |
|---|---|---|---|---|---|---|
| 36.2 | 5.2 | pipelined | shift | 1 : 7 | from | f(i, j, 1) |
|  |  |  |  | 0 : 6 | to | f(i, j, 9) |

Data Layout

| NAME | ALIGNMENT | DISTRIBUTION | DECOMPOSITION |
|---|---|---|---|
| c |  |  |  |
| e |  |  |  |
| f | d4 | d4(:,:,BLOCK) | d4(1:64,1:64,1:64) |

**Figure 7**: D Editor Performance Overview