Mapping Performance Data for High-Level and Data Views of Parallel Program Performance[1]

R. Bruce Irvin
rbi@informix.com

Informix Software, Inc.
921 SW Washington St.
Portland, OR 97205

Barton P. Miller
bart@cs.wisc.edu

Computer Sciences Department
University of Wisconsin
1210 W. Dayton Street
Madison, WI 53706-1685

Abstract

Programs written in high-level parallel languages need profiling tools that provide performance data in terms of the semantics of the high-level language. But high-level performance data can be incomplete when the cause of a performance problem cannot be explained in terms of the semantics of the language. We also need the ability to view the performance of the underlying mechanisms used by the language and correlate the underlying activity to the language source code. The key techniques for providing these performance views is the ability to map low-level performance data up to the language abstractions.

We identify the various kinds of mapping information that needs to be gathered to support multiple views of performance data and describe how we can mine mapping information from the compiler and run-time environment. We also describe how we use this information to produce performance data at the higher levels, and how we present this data in terms of both the code and parallel data structures.

We have developed an implementation of these mapping techniques for the data parallel CM Fortran language running on the TMC CM-5. We have augmented the Paradyn Parallel Performance Tools with these mapping and high-level language facilities and used them to study several real data parallel Fortran (CM Fortran) applications. Our mapping and high-level language techniques allowed us to quickly understand these applications and modify them to obtain significant performance improvements.

1 INTRODUCTION

High-level parallel languages offer portable, concise notations for specifying parallel programs, and their compilers automatically map programs onto complex parallel machines. These languages can free programmers from the difficult, error-prone, and sometimes ineffective task of specifying parallel computations explicitly. We describe a tool for profiling the performance of parallel programs written with a high-level parallel language or library. This paper concentrates on the mechanisms for *mapping* low-level performance data to the high-level language control and data

1. This work is supported in part by Wright Laboratory Avionics Directorate, Air Force Material Command, USAF, under grant F33615-94-1-1525 (ARPA order no. B550), and CDA-9024618, and Department of Energy Grant DE-FG02-93ER25176. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the Wright Laboratory Avionics Directorate or the U.S. Government.

structures. We show how these techniques can be combined with *dynamic instrumentation* [13,19] (a technique for on-the-fly instrumentation of executable programs) from the Paradyn Parallel Performance Tools. We present two of our case studies of the use of this tool to profile and improve the performance of real-world parallel applications.

Our approach to tools for high-level parallel languages emphasizes three crucial features that have contributed to the usefulness of our system. First, we present performance data in terms of the semantics of the programming language. The language provides abstractions that insulate the programmer from many of the details of communication, synchronization, and parallelization. If we present performance data in terms of low-level node computing and message passing, for example, it would be difficult for the programmer to relate this information to their program.

There are several existing tools that do source-level profiling. Some commercial examples of these tools include the MPP Apprentice[29] for the Cray T3D, Prism[27] for the TMC CM-5 and MPPE[18] from Maspar. In the research world, examples include the Pablo system from the Universities of Illinois[24,1] that can trace Fortran D programs and present this information in terms of the source program, and TAU from University of Oregon[20] that can do similar operations for pC++ programs[2]. While presenting performance data at the source code level of a high-level parallel language is crucial, it is not always sufficient. The language abstractions can insulate a programmer from the need to specify the low-level details, but they can also hide the cause of a performance problem.

We address this problem with the second feature of our tool: we allow the programmer to *peel back* the layers and examine the lower level details of program performance. We can map the lower-level operations to the source language statements that caused them, and map the source language statements to the lower-level operations that they caused. Performance information can be available for each of the programming layers on which the parallel program is built. The programmer can view performance data at any of these layers, and map the data between layers. *This mapping is the crucial mechanism for providing high-level performance data, while simultaneously being able to view lower-level performance data.*

The third feature of our tool is that we can associate performance data with data structures, as well as control structures. Most tools associate various performance metrics with control structures such as procedures, loops, or statements, but in many parallel languages it is the data structure that is the source of the parallelism. For example, consider arrays in data parallel Fortran and objects in parallel C++ dialects. Associating performance information with the data structures being used can provide useful intuitions into the cause of a performance problem. For example, a bottleneck may be associated with operations on one (or a few) parallel arrays, while control profiling may show the main work diffused over many parts of the code. Our facility for mapping low-level performance data makes it relatively easy to provide both control and data views.

Data views of performance are not new; they have been used to study memory system behavior. For example, Cprof [17] and MemSpy[8] can relate cache hits and misses to data structures in sequential C and Fortran programs. Several algorithm animation and visualization tools have been developed as part of sequential object-oriented programming systems [9,16,5]. Each of these object-oriented systems draws a representation of an object hierarchy and then animates the view with execution information such as recordings of object activations and method invocations.

We generalize these ideas to provide a complete control and data view of performance.

In the next section, we briefly describe our model of high-level performance data. Section 3 then describes how we provide the mappings for our layered and multi-view presentation of performance data. These ideas have been built into the Paradyn Parallel Performance Tools. Section 4 reports on some of our experiences using these tools to study real parallel applications. We conclude in Section 5.

2 THE NV MODEL OF HIGH-LEVEL PERFORMANCE DATA

We have developed a framework within which we can discuss performance characteristics of programs written in many different programming models. This framework is called the Noun-Verb (NV) model [14]. In NV, a *noun* is any program element for which performance measurements can be made, and a *verb* is any potential action that might be taken by a noun or performed on a noun. The collection of nouns and verbs of a particular software or hardware layer defines a *level of abstraction*. Nouns and verbs from one level of abstraction are related to nouns and verbs from other levels of abstraction with *mappings*. A mapping expresses how high-level language constructs are implemented by low-level software and hardware. With mappings, performance information collected at arbitrary levels of abstraction can be related to language level nouns and verbs. Mappings are discussed in detail in Section 3.

We will describe the NV model using examples from the data-parallel language CM Fortran [28]. CM Fortran (and its implementation on CM-5 computers) is representative of many high-level parallel programming languages, including HPF [11]. The NV model, however, is applicable to many other parallel programming models.

```
1     PROGRAM EXAMPLE
2     PARAMETER (N=1000)
3     INTEGER A(N+1,N+1), B(N,N), ASUM
4
5     A = 0
6     DO K = 1,10
7     FORALL (I = 2:N+1, J = 2:N+1) A(J,I) = K*(I+J)
8     ASUM = SUM(A)
9     FORALL (I = 1:N/2, J = 1:N/2) B(J,I) = A(J,I) + A(J+1,I+1)
10    END DO
11    END
```

Figure 1: Example CM Fortran Program

We will use the example CM Fortran program in Figure 1 to describe some of the nouns and verbs of the CM Fortran language. The example program declares two multi-dimensional arrays (line 3), initializes all elements of array `A` with a parallel assignment statement (line 5), assigns values to a subsection of array `A` (line 7), computes the sum of the array (line 8), and computes a function of the upper left quadrant of array `A` and assigns it to array `B` (line 9).

NV nouns for the CM Fortran include the program (line 1), subroutines, FORALL loops, arrays (`A` and `B` on line 3), and statements (lines 5, and 7-9). Verbs in CM Fortran include statement *execution* (lines 5-10), array *assignment* (lines 5, 7, and 9) and *reduction* (line 8), subroutine *execution*, and file *IO*.

A particular execution instance of the program construct described by a verb is called a *sentence*. A sentence consists of a verb, a set of participating nouns, and a cost. The cost of a sentence may be measured in terms of such resources as time, memory, or channel bandwidth. Finally, *performance information* consists of the aggregated costs measured from the execution of a collection of sentences. For example, performance information for array A might include measurements of the assignments of lines 5, 7, and 9, and the reduction on line 8. Performance information for array B, however, would include only measurements of the assignment on line 9.

The method of expression of a verb is either *explicit*, meaning that the verb was directly requested by the programmer through the use of a program language construct, or *implicit*, meaning that the programmer did not explicitly request the verb but that it occurred to maintain the semantics of the computational model.

High-level language programs are usually built on several levels of abstraction, including the source code language, runtime libraries, operating system, and hardware. In well constructed systems, each level is self-contained; levels interact through well-defined interfaces. It is possible to measure performance at any level, but measurements may not be useful if they are not related to constructs that are understood by the programmer. In the NV model, each level of abstraction for which performance may be measured is represented by a distinct set of nouns and verbs. Nouns and verbs of one level may be mapped to nouns and verbs in other levels.

For CM-5 systems, a CM Fortran program is compiled into a sequential program and a set of node routines. The sequential program executes on the CM-5 Control Processor and makes calls to the parallel node routines and to parallel system routines through the CM Runtime System (CMRTS). The CMRTS creates arrays, maps arrays to processors, implements CM Fortran intrinsic functions (e.g., SUM, MAX, MIN, SHIFT, and ROTATE), and coordinates the compute nodes. Each parallel CM Fortran array is divided into sections, and each section is assigned to a separate node. Each node is responsible for computations involving its local array sections; if array data from non-local sections are needed, then the non-local data must be transferred before computation can proceed.

Our NV mappings for CM Fortran divide the CM-5 system into three levels of abstraction. The highest level, called the CMF level, contains the nouns and verbs from the CM Fortran language. The middle level is the RTS level. RTS level nouns include all of the arrays allocated during the course of execution. This set of arrays includes all of the arrays found in the CMF level as well as all arrays generated by the compiler for holding intermediate values during the evaluation of complex expressions. Verbs of the RTS level include array manipulations such as *Shift*, *Rotate*, *Get*, *Put*, and *Copy*. The lowest level of abstraction is the node level. Node level nouns include all of the compute nodes. Node level verbs include *Compute*, *Wait*, *Broadcast Communication*, and *Point-to-Point Communication*.

A mapping relates nouns (verbs) from one level of abstraction to nouns (verbs) of another level. A mapping may be top-down or bottom-up. For example, a top-down mapping from arrays to nodes might relate a particular array (or subsection of an array) to a particular set of compute nodes. A bottom-up mapping from node routines to code lines might relate CPU time recorded for a particular node routine to the CM Fortran statement from which it was compiled.

3 MAPPING PERFORMANCE DATA

Performance tools for parallel language applications must consider how performance information maps across layers of abstraction. As we described in Section 2, mappings in the NV model provide a way to represent the relations between abstraction levels for nouns or verbs. Any performance information that is measured for a sentence is relevant not only to itself, but also to all nouns or verbs to which it maps. We now describe several types of mapping, techniques that we have used to mine mappings from real systems, and a method for tracking dynamic verb mappings.

3.1 Kinds of Mapping

Noun and verb mappings in a parallel programming system can take many forms. The simplest form is static mapping information. Static mapping information is any mapping information provided to a performance tool prior to the execution of an application program. Traditionally, performance tools have exploited static information found in debugging symbol tables to explain the runtime performance of applications. Symbol tables help performance tools map from memory locations at the processor level of abstraction to procedures and statements at the source code level.

However, sometimes tools must look beyond traditional symbol tables to identify static mappings and lists of program code and data nouns. For example, a key piece of mapping information for the CM Fortran programming system is the mapping between node-level code blocks and CM Fortran-level parallel assignment statements. The compiler determines these mappings when it compiles assignment statements into node-level code, and it often merges separate node-level code blocks together to improve performance.

Mapping information can also be dynamic. Dynamic mapping information includes any mapping information that is generated during application execution, and differs with static mapping information only in that it must be gathered during program execution. For example, if an application dynamically allocates parallel data objects, then the application must dynamically communicate the definition of the corresponding noun to the performance tool. If the application dynamically distributes the data object across parallel compute nodes, then the application must dynamically define a mapping between the object and the nodes for the performance tool. The performance tool can use the dynamic mapping information during or after execution to relate performance measurements to abstract program constructs and activities.

Mappings for performance information can be one-to-one, one-to-many, many-to-one, or many-to-many. All of these types arise in actual systems and must be handled properly by performance tools. Theses types of mappings are shown in Figure 2 along with a brief explanation of how a tool could handle them. The most interesting cases, one-to-many and many-to-many, arise when a compiler or runtime system has merged the activities of two or more source-level constructs and generated a single low-level sentence to implement them. In this case, the optimization has obscured the boundary between the source-level sentences and performance tools must either split or merge the measured data. For example, compiler optimizations may cause several statements to be combined into one statement or communication optimizations may cause several messages to be combined and sent together. Our view is that performance data for a merged operation should be reported as a single value for the merged set of statements; e.g., “Statements 4-7 had 10 seconds of synchronization blocking time." If you try to split the cost between the merged source statements, you must be able to assign a proportion to each statement; this is not always possible. Simply making an even split (such as is done with recursive procedure call chains in gprof [7]) can produce misleading results.

Type of Mapping	Example	How to assign low-level costs to high-level structure
One-to-One	Low-level message send S implements high-level reduction R.	Measurements of S are equivalent to measurements for R.
One-to-Many	Low-level function F implements reductions R1, R2, ...	(1) Cost of F is split evenly over all R, or (2) Merge all R into one set and assign cost of F to entire set.
Many-to-One	Low-level functions (F1, F2, ...) implement one source line L.	First aggregate costs of F1,F2,... then assign cost to line L.
Many-to-Many	Many source code lines L1, L2, ... are implemented by an overlapping set of low-level functions F1, F2,...	First aggregate costs of F1, F2, ..., then treat as a one-to-many mapping to L1, L2, ...

Figure 2: Types of Upward Mappings

3.2 Mining Mapping Data

Mining noun, verb, and mapping data from a parallel programming system is sometimes difficult because the programming systems either do not provide information or the information is provided in non-standard ways. With some programming languages, naming particular nouns can be difficult, and performance tools must generate unique names for them. For example, languages such as Lisp and C++ do not explicitly identify dynamically created data objects, and we might name such nouns in a variety of ways. The choice of naming scheme depends on the parallel language and the functionality of the profiling tool. The goal of these naming schemes is to provide names that are intelligible to the programmer and are sufficiently disambiguous (if a name is overly disambiguous, desired cumulative effects may not be detectable).

1. We can name the object by the function and line number of the statement that created the object. For example, we might have the name `foo.c:foo:75` for an object allocated on line 75 of function foo() within source file foo.c. Multiple objects created in the same place will have the same name; this may or may not be a desired effect.

2. We can name the object after the control structure within which it was allocated. For example, `(retval foo)` might name the list returned by function foo() in a Lisp program. In this case, all lists returned by foo() will have the same name. If you are interested in the cumulative behavior related to the return values from this function, then this naming scheme is appropriate. For other uses, this scheme might create undesired ambiguity.

3. For cases in which we want to differentiate between various calls to a memory allocation routine, we may add the names of functions that are currently on the dynamic call stack. For example, `(retval(foo bar))` is the list

returned by function foo() when called by function bar(). This is similar to the scheme used in the Cprof memory profiler [17]. This technique can also be applied to case 1.

4. We could name the object by the memory location at which it was allocated. For example, `noun:0x5000` might represent the data object allocated at memory location 0x5000. If memory space is re-used, there is a chance of ambiguity. This scheme also provides little semantic meaning to the programmer.

5. We could give unique global or local names to dynamically allocated objects. For example `noun:#56` might indicate the 56th object allocated during program execution. This scheme provides information about the order in which objects are created. While it provides a unique name to each object, the name provides little semantic meaning to the programmer.

6. We could ask the programmer to supply a name. For example, we could ask C programmers to supply an extra argument to calls to the memory allocator `malloc()`. While this scheme can produce meaningful names, it puts an extra burden on the programmer and (probably more important) creates a non-standard interface to a system-supplied function.

7. Languages such as C++, can name dynamically create instances of a class using the class name and an identifier. For example, `Queue:#34` might indicate the 34th instance of class Queue. This scheme is similar to case 5 above, but with added type information.

A combination of the above schemes is likely to be most useful. For example, combining cases 1, 4, 5, and 7 would produce a name like `foo.c:foo:Queue:34:0x5000`, a Queue object at memory address 0x5000 created in function foo on line 34 of file foo.c.

Code structures can be simple to name when the language gives them an explicit name, such as a function. But even this simple case can be complicated by such things as inlining, inheritance, overloading, and type templates. Objects without explicit names, such as loops, iterators, switches, and simple statements require names to be generated for them.

1. The simplest (and most common) approach is to name code structures after the location at which they are defined. For example `foo.f/doloop:53` might identify the do loop that begins at line 53 of the Fortran program contained in the file foo.f.

2. We can ask the programmer to insert compiler pragmas into their code to name particular control structures. For example, the programmer could give a name such as *innerLoop* to a particularly important loop.

3. In some languages, programmers can create control structures dynamically, and we can name them with one of the schemes proposed above for dynamic data objects.

Static mapping information is sometimes available from programming systems, but often times we must deduce mapping information from other explicit channels of information. For example, CM Fortran does not provide explicit mapping information between the RTS-level code blocks that are executed on the compute nodes and CMF-level parallel assignment statements to which they correspond. Instead, we must parse compiler-generated listing files and compiler-generated assembly code to deduce mappings between node code blocks and statements. Our parser generates tuples containing the compiler-generated names of the code blocks and the file and line number of the statement. These tuples are entered into a Paradyn information file as one-to-one mappings and Paradyn later merges statements that have node-level code blocks in common. Then Paradyn automatically aggregates all performance information for the node code blocks that map to each merged group. In this way, Paradyn treats a many-to-many mapping by first merging the high-level nouns, thus forming a many-to-one mapping.

Mapping points are usually unique to each programming system. A mapping point is any function, block, or line of code in an application where dynamic mappings may be constructed. For example, in the CM Fortran system, we identified the run-time functions that allocate parallel arrays and distribute them to the compute nodes. The return point of the routine is instrumented by Paradyn at runtime such that the instrumentation code examines the return value of the allocation function, reports the name of the array and the compute node memory location to Paradyn, and reports CM Fortran-specific mapping parameters to Paradyn. Paradyn later interprets these parameters and generates mappings between memory locations (and compute nodes) to CM Fortran-level array subgrids. When a Paradyn user requests performance information for a particular array, Paradyn maps the request to instrumentation on the compute nodes that monitors the appropriate address range (every node-level code block and runtime function is called with a memory location parameter). If a user requests performance information for a particular subgrid of an array, then Paradyn translates not only to a memory location, but also to a subset of the compute nodes.

3.3 Tracking Mappings: Active Sentences

Some dynamic mapping information is difficult to determine simply by instrumenting mapping points in an application. Verb mappings between layers of abstraction are often difficult to detect because the implementation of one layer is usually hidden from other layers for software engineering reasons. We describe the Set of Active Sentences (SAS), a data structure that allows us to dynamically map which sentences are concurrently in the layers of abstraction. We describe the SAS with an example taken from High Performance Fortran and describe the kinds of questions that might be asked and answered with the SAS.

The Set of Active Sentences (SAS) is a data structure that records the current execution state of each level of abstraction, similar to the way a procedure call stack keeps track of active functions.Whenever a sentence at any level of abstraction becomes active, it adds itself to the SAS, and when any sentence becomes inactive, it deletes itself from the SAS. Any two sentences contained in the SAS concurrently are considered to dynamically map to one another.

```
        ASUM = SUM(A)
        BMAX = MAXVAL(B)
```

Figure 3: Example HPF Code Fragment

For example, consider the example HPF code fragment in Figure 3. With this code, we might be concerned with the following problem: how to relate a low-level message to a high-level array reduction. The SUM reduction and the MAXVAL reduction imply that messages must be sent between processors on a distributed memory parallel computer. We assume that each node of a parallel computer holds subsections of arrays A and B, and each node reduces its subsections before sending its local results to other nodes to compute the global reductions. We also assume that a performance tool can measure the low-level mechanisms for message transfer (e.g., message send and receive routines), and can monitor the execution of the high-level code (e.g., which line of code is active, which array is active, or what operation is being performed on the array).

We might want to answer such questions as: How many messages are sent for summations of A? For finding the MAXVAL of B? How much time is spent sending messages for summations of A?

These questions are representative of questions that we would like to ask for any language system built on multiple layers of abstraction. In any such system, we want to explain low-level performance measurements in terms of high-level programming constructs (and vice versa).

In the SAS approach to dynamic mapping, we defer the asking of performance questions until run time, and then only measure those sentences that satisfy at least one performance question. As explained above, the SAS keeps track of all sentences that are active at any level of abstraction. Whenever any sentence becomes active, monitoring code notifies the SAS, and the SAS remembers all such active sentences if they could satisfy an outstanding performance question. When a low-level sentence is to be measured (whether by a counter, timer, or any other means), monitoring code queries the SAS to determine what sentences are currently active and thereby relates low-level sentences to active sentences at higher levels. Figure 4 shows the contents of a hypothetical SAS for our example HPF code.

HPF: line #1 executes

(each line represents one active sentence)

HPF: A sums

Base: Processor sends a message

Figure 4: The SAS at the moment when a message is sent.

The figure represents a snapshot of the SAS at the moment when a message is sent as part of the computation of the sum of array A. It shows that three sentences are active, two at the HPF level of abstraction, and one at the base level. Any part of an application (e.g., user code, programming libraries, or system level code) may add and remove sentences from the SAS and need not know about the existence of other layers to do so.

Our use of the SAS resembles the way in which some performance tools for sequential programs make use of a monitored program's function call stack [6,7,8,17,23]. A program's function call stack records the functions that are active at any given point in time. By exploring the call stack, a performance tool can relate performance measurements for a function to each of its ancestors in the program's call graph. Users of such a performance tool can then understand how function activity relates to the dynamic structure of their programs. The SAS, however, may record any active sentence, regardless of whether the sentence could be discovered by examining the call stack.

We have defined the SAS to be a global data structure. If our target hardware systems support shared global memory, then we can use globally shared memory to store the SAS. However, many of today's parallel systems do not use globally shared memory, and even for those that do, we may not want to pay the synchronization cost of contention for such a globally shared data structure. Fortunately, we can still use the SAS approach if we duplicate the SAS on each node of a parallel computer, just as application code is duplicated for Single Program Multiple Data (SPMD) programs. Each individual SAS can operate independently of others as long performance questions are not asked that require information from several SASs.

The SAS approach is not appropriate for tracking mappings between asynchronous activities. For example, in a UNIX system we may want to measure kernel disk writes that occur on behalf of a particular function in a user process. In this system, writes to disk occur asynchronously with process execution. As a result, the SAS may not contain both the function execution sentence and the kernel disk write sentence at the same time, and therefore mappings between kernel disk writes and user function execution could not be tracked with the SAS alone.

4 CASE STUDIES OF CM FORTRAN APPLICATIONS

We have used NV in Paradyn in three performance experiments with real CM Fortran application codes. In this section, we describe the applications, our measurements of the applications, and our use of Paradyn's performance information to improve the applications. Our experiences with these applications show us that performance information about Node-level verbs helps to explain performance problems that can be difficult to fully understand at the CMF level of abstraction. The NV mapping mechanism was the key to providing these multiple views. We also found that data views of performance add a useful perspective to performance studies of CM Fortran applications, and that, in some cases, data views can localize performance problems to a few parallel arrays, even when the problems are diffuse in code views. The NV mappings made it easy to provide both the control and data views.

Note that we had no previous experience with each of these applications and that we spent less than a week on our study of each of them, from first study to final measurements on the modified application.

4.1 Vibration Analysis (Bow)

Our first application (called Bow) simulates the oscillations of a bowed violin string [25,26]. This application has been used to study such variables as initial transients in bowed strings, frictional interaction at the bow-hair string interface, and stiff strings. The code consists of approximately 1200 lines of CM Fortran code contained in a single source file. By varying input parameters, we can control the resolution of simulation, the length of simulation, and the simulation's initial conditions. The code was provided by Robert Schumacher, Professor of Physics, Carnegie Mellon University.

We began our study of Bow by examining the overall performance characteristics of the application as shown in Figure 5. The figure shows a time plot of three primary CMF-level metrics and CMF Overhead for a run on a 32 node partition of a CM-5 system. The metrics are not constrained to any particular module, subroutine, statement, or array, but instead are collected for the whole program. The total available time as measured by Paradyn is 16 CPUs[2].

The display in Figure 5 shows that RTS-level and Node-level overhead costs dominate the execution of Bow.

Only Computation Time is significant among the CMF-level metrics and it accounts for only a fraction of the CPU resources available to the application. The display also shows that the Computation Time metric is relatively uniform over the entire execution. We concluded that the majority of the application's execution must be spent in the lower level, RTS-level and Node-level, activities.

To identify the lower-level activities that consumed the majority of the CPU time available to Bow, we peeled back layers of abstraction and examined RTS-level and Node-level performance data, as shown in Figure 6. This figure shows the CMF-level Computation Time metric, along with measurements of Idle Time (time spent waiting for the CM-5 control processor), Argument Processing Time (time spent receiving parameters for node-level code blocks), and Node Activations (frequency of activation of the parallel nodes by the control processor).

We can see from the display that the cost of processing arguments and idling are higher than the cost of computing, and that the rate of node activation is also high (over 100,000 per second or almost once per 150 microseconds per node). These measurements suggest that each computation activated on the nodes is a small computation.

The performance data shown in Figure 6 led us to believe that, although the application had been partitioned properly, the application's computational grain-size was too small for the CM-5. We knew that the program's authors efficiently distributed and aligned the program's data because there are virtually no costs associated with node-to-node messages or broadcasts (not shown). However, the large amounts of idle time, argument processing time, and node activations indicate that the control processor spends a large amount of time deciding what the compute nodes should do, but the nodes do not spend much time finishing their tasks. Therefore, most of the machine waits for the control processor most of the time.

We then used a data view of performance to determine which arrays were involved in computations. We displayed Computation Time for the arrays shown in Figures 7 and 8. The time plots in Figures 7 and 8 show the Computation Time metric constrained to four different arrays that account for almost all of the computation activity in the application. To find these four arrays we examined the Computation Time metric for several dozen parallel arrays and statements, and the four shown in the figures stood out as the top users of computation resources. We also discovered (not shown) that the time spent per computation on each of the arrays was very small. This concurred with our earlier hypothesis that the computational grain-size was small.

When we looked at the code that manipulated these four arrays, we found several DO loops like the one shown in Figure 9. We immediately saw why the code was not achieving full parallelism. The code fragment is part of a convolution in which a small matrix is multiplied, row by row, over a larger matrix. Each column of a given row is calculated in parallel but each row is computed sequentially and immediately added to a running vector sum. The rows are rarely very long, and computing each one separately does not provide enough work per computation to outweigh the

2. The CM-5 allocates a full time quantum to every process that is ready to run on the CM-5's compute nodes, regardless of whether the process requires a full time quantum. Paradyn's daemon process is always allocated a full time quantum, even though it rarely requires a full time quantum to complete its work. Therefore, any application measured by Paradyn on a CM-5 is only allocated half of the total available CPU time. The application's execution is not affected, it is only scheduled half as often as it might.

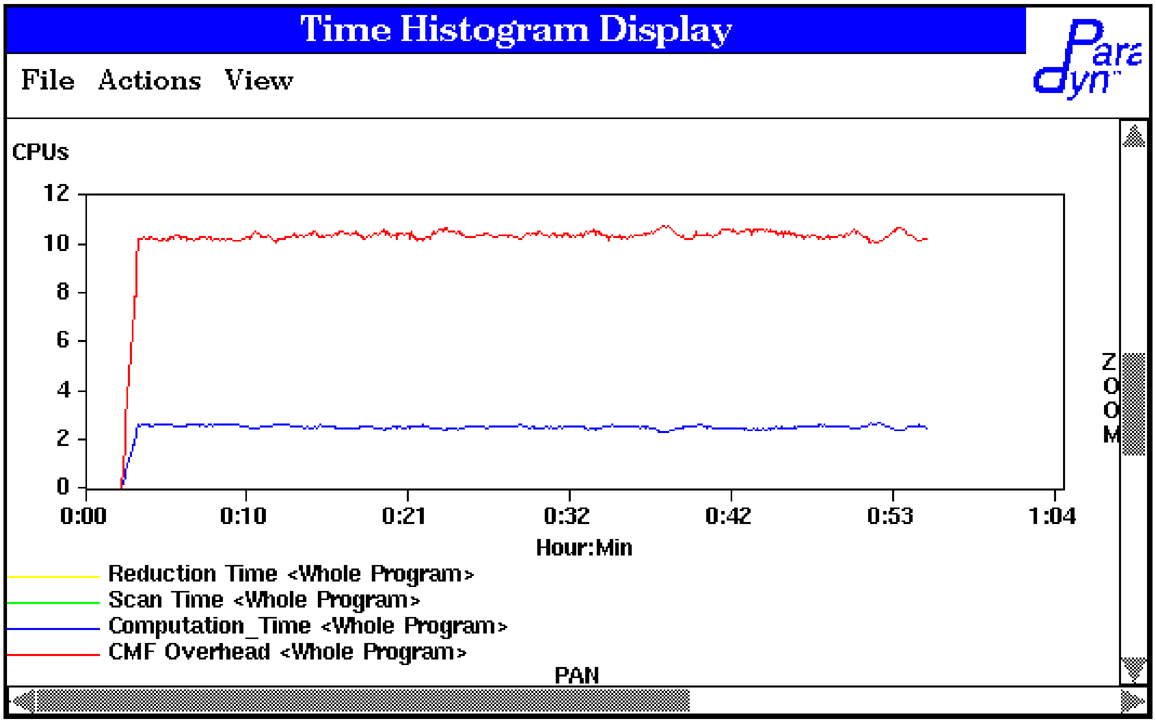

Figure 5: CMF-level Performance Data for Bow (Scan_time and Reduction_time are each at or near zero during entire execution)

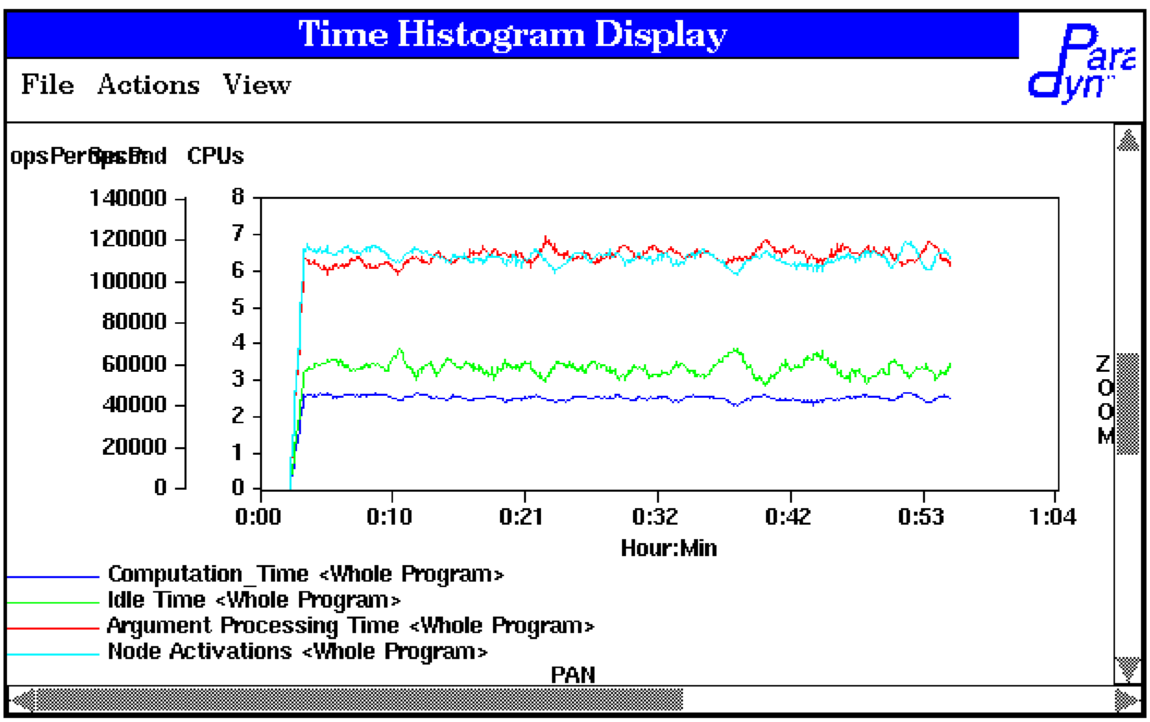

Figure 6: Node-level Performance Data for Bow

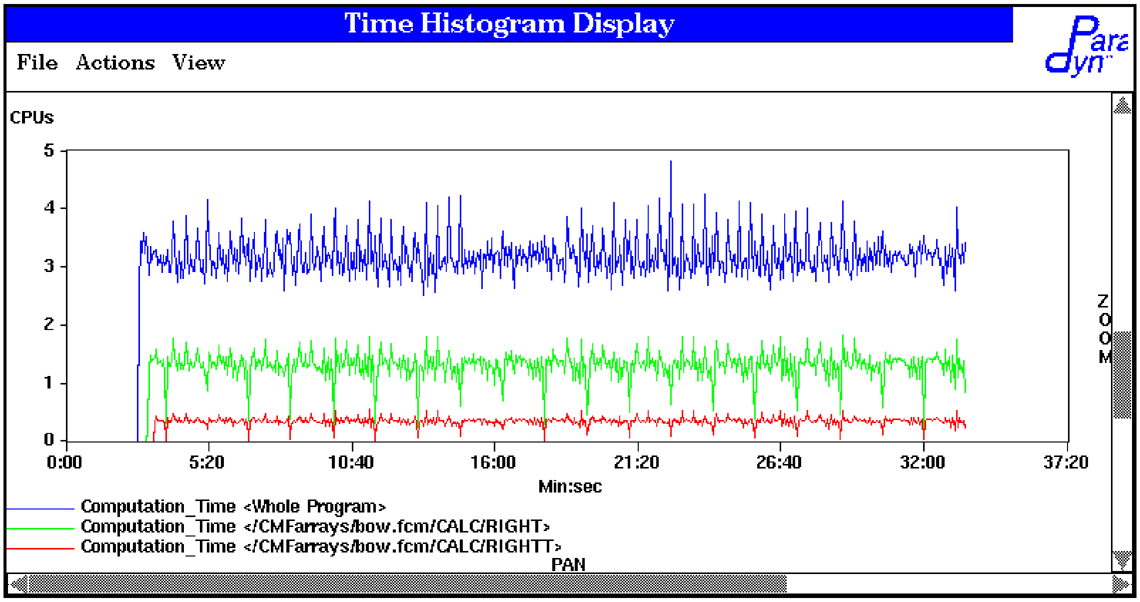

Figure 7: Performance for Righthand Convolution Matrices for Bow
"/CMFarrays/bow.fcm/CALC/RIGHT" means CM Fortran parallel array "RIGHT" declared in function CALC in source file bow.fcm.

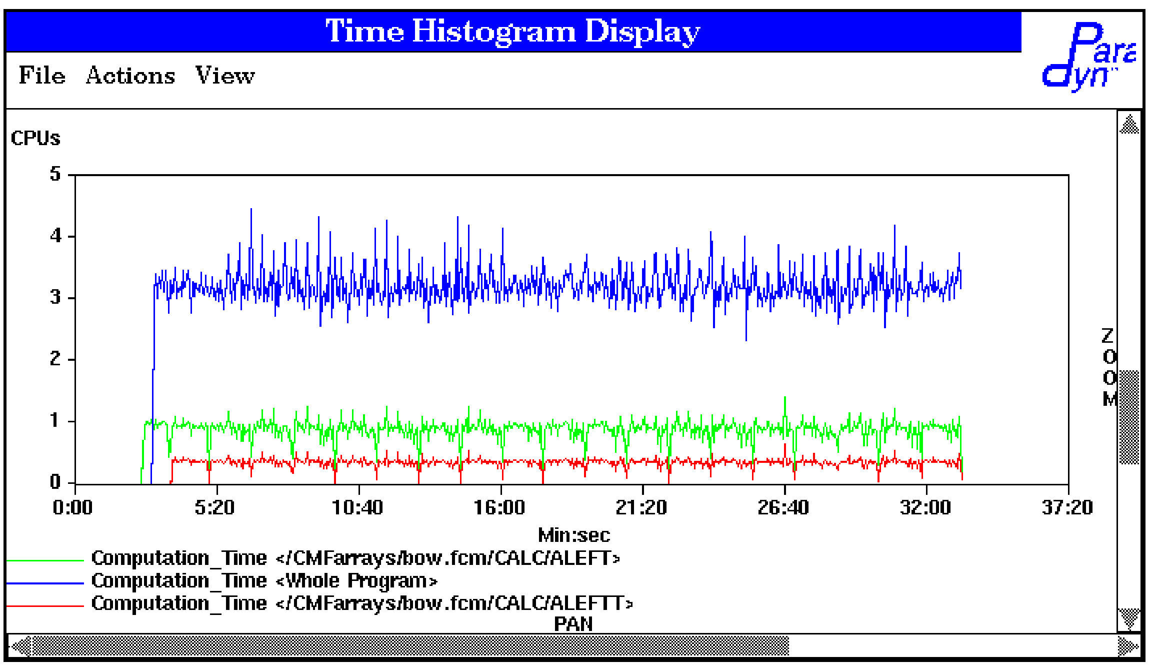

Figure 8: Performance for Lefthand Convolution Matrices for Bow
"/CMFarrays/bow.fcm/CALC/ALEFT" means CM Fortran parallel array "ALEFT" declared in function CALC in source file bow.fcm.

```
ALEFT = 0.0
DO I=1,LENL
  ALEFT = ALEFT + (HISTL(IBACKL+I,:) * CRLEFT(I,:))
END DO
```

Figure 9: Example of Original Convolution Code (taken from lines 635-638 of the code)

implicit costs of starting the compute nodes and sending them arguments for the computation. Therefore, the application does not execute efficiently on the CM-5 hardware.

To increase grain-size, we altered the code so that both the rows and the columns of the convolution matrix are executed in parallel, as shown in Figure 10. As the code fragment shows, we allocate a temporary array (`LT`) to store intermediate values and we use an intrinsic parallel sum operation to calculate the final result vector. The storage required for the new temporary vector is relatively small because it need only be as large as the size of the convolution matrix. The results of these changes may be seen in the time plot shown in Figure 11.

```
LT(1:LENL, :) = HISTL(B+1:B+LENL, :) * CRLEFT(1:LENL, :)
ALEFT = SUM(LT, DIM=1)
```

Figure 10: Improved Convolution Code

Figure 11 shows that the execution time of the application has been reduced substantially as a result of our changes. We can now see that the Idle Time, and Argument Processing Time have both been reduced substantially and that Computation Time has increased substantially. These curves indicate that the compute nodes spend more of their time doing useful work and less time waiting for the control processor. Furthermore, the Sum Time curve in the figure shows the portion of Computation Time due to Summation operations.

To determine whether our improvements to Bow were useful across a range of various input parameters, we ran both versions of the application through five input sets (without Paradyn attached). The results are shown in Figure 12. The table shows that the actual reduction in execution time for the uninstrumented application improves slightly with increased iteration size and that the overall reduction was quite close to the change predicted by examining before (Figure 5) and after (Figure 11) runs using Paradyn.

Our measurements of the Bow application suggest that source-level views can be more useful when augmented with lower level views of performance. We demonstrated that a perfectly parallel application can have high Node-level initialization costs if it is spread thinly across compute nodes. In the CM Fortran execution model on the CM-5, small grain-size causes frequent node activations, large amounts of idling while nodes wait for the control processor, large proportions of time spent receiving code-block arguments from the control processor, and very brief computation periods on the nodes. Paradyn allowed us to describe, measure, and display the performance of these Node-level implicit activities.

Data views of performance were especially useful (relative to code views) in focusing on the parallel data struc-

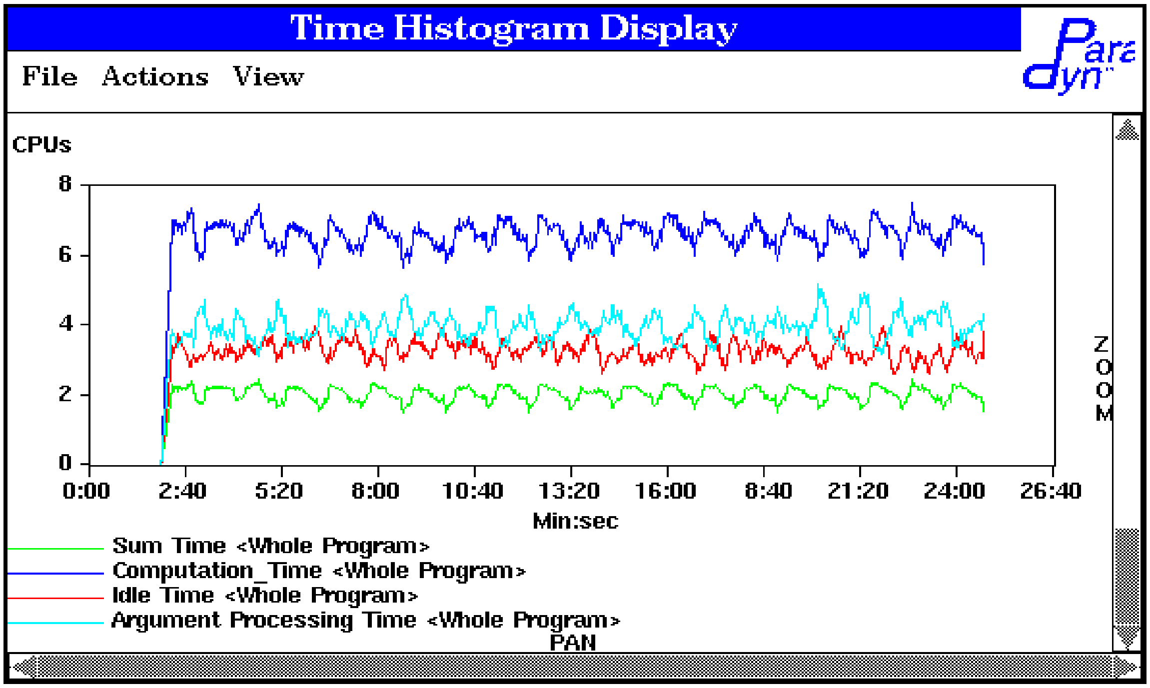

Figure 11: Paradyn Display of Improved Bow

Iteration Count	Original	Improved	Percent Change
5x5	01 min 35 sec	01 min 09 sec	-27%
10x10	05 min 49 sec	04 min 09 sec	-29%
15x15	12 min 47 sec	09 min 02 sec	-29%
20x24	26 min 51 sec	18 min 57 sec	-29%
32x30	53 min 24 sec	37 min 31 sec	-30%

Figure 12: Execution Times for Original and Improved Bow (all times are for uninstrumented executions of the program)

tures used in several fine-grained computations. Although the fine-grained computations were diffused among many statements, they were localized to just a few parallel arrays.

There may be other performance problems and potential improvements for the Bow application. The overall costs of implicit activities as shown by the Idle Time and Argument Processing Time curves in Figure 11 indicate that there are still some significant costs related to implicit activities.

4.2 Peaceman-Rachford PDE Solver (Peace)

Our second application, called Peace, is a partial differential equation solver that uses the Peaceman-Rachford iteration method [21,22]. The Peaceman-Rachford method is sometimes used to factor penta-diagonal equations. This particular implementation was written by Vincent Ervin, Associate Professor of Mathematics at Clemson University. The code consists of approximately 1100 lines of CM Fortran code spread over five files. We can control the execution of the program by varying the number of variables in the equation, by altering the convergence coefficient, and varying the number of iterations.

We began our analysis of Peace by examining the costs of CMF-level verbs in a time plot, as shown in Figure 15. The display shows that the application has two distinct phases of execution with a substantial change in Computation Time and Array Transformation Time at the phase boundary (approximately ten minutes, forty seconds into execution). The first phase exhibits relatively low costs associated with CMF-level verbs and high overhead costs, while the second phase shows much more compute node utilization for CMF-level verbs.

We investigated implicit, Node-level costs in the first phase by moving to the RTS level of abstraction and displaying metrics as shown in Figure 16. The display shows relatively high costs for implicit verbs such as processing parallel code block arguments, broadcasting, and activating parallel code blocks. Little time is spent doing actual computations during the first phase. During the second phase, the application spends a much larger percentage of time computing with no communication between compute nodes.

We investigated why the first phase of Peace incurred high costs for initializing the compute nodes of the CM-5. We used visualization displays (not shown) to find that most of the node activations in the phase were due to broadcasts, and that small amounts of data were broadcast per activation. This information indicated that most of the costs of waiting for the control processor, activating the nodes, and receiving code block arguments from the control processor were artifacts of very brief broadcasts.

We examined code views and data views to explain the cause of the broadcasts. To identify the data being broadcast, we constrained our measurements of broadcast activity to data structures at the CMF level. The data view in Figure 13 shows us which parallel arrays are broadcast and in what order. Almost all of the broadcast costs are associated with three parallel arrays. The first array (`CONODE`) is broadcast exclusively for over one minute and is then broadcast concurrently with each of the other two arrays (`RCONODE` and `BCONODE`). We examined the code to find that `CONODE` was computed in a special routine that used parallel assignments within sequential iteration loops and that `RCONODE` and `BCONODE` were then computed using elements of `CONODE`. The code to perform the computations of `CONODE`, `RCONODE`, and `BCONODE` included forty lines of code spread over two functions in two files.

We improved Peace by replacing all of the code that computed `CONODE`, `RCONODE`, and `BCONODE` with a set of two parallel loops that compute all three arrays simultaneously. The new code eliminates all broadcast activity and greatly reduces the execution time of the first phase of the application. Time plots of the improved application (not shown) are similar to the second phase of the displays shown in Figures 15 and 16.

To measure the effect of our changes, we ran the original and improved versions of Peace over five input sets of varied size. The results, shown in Figure 14, show that the improvements help to reduce execution time over a range of input sizes.

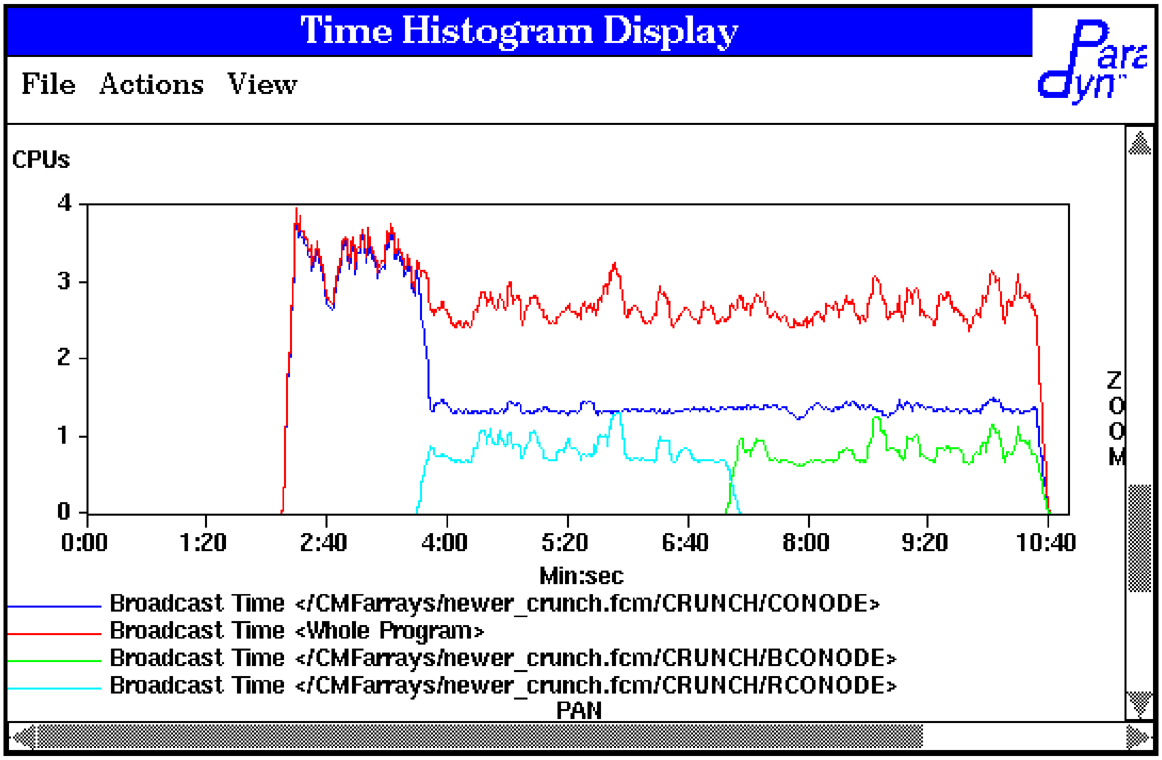

Figure 13: Node-level Broadcasts Constrained by CMF-level Arrays for Peace

"/CMFarrays/newer_crunch.fcm/CCRUNCH/RIGHT" means CM Fortran parallel array "CONODE" declared in function CRUNCH in source file bow.fcm.

Input Size	Original	Improved	Percent Change
66K	3 min 14 sec	1 min 44 sec	-46%
148K	6 min 39 sec	3 min 13 sec	-51%
333K	14 min 35 sec	6 min 53 sec	-53%
491K	22 min 1 sec	10 min 48 sec	-51%
1051K	44 min 55 sec	20 min 56 sec	-54%

Figure 14: Execution Times for Original and Improved Peace

The second phase of Peace seems to be limited by RTS-level and Node-level implicit costs, but as we scale up the problem set sizes, the costs of implicit activity decrease in comparison with the costs of CMF-level explicit activity. We used a simple bar graph visualization to show that computations and array shifting now account for most of the processing time available to the application. We saw that two arrays, `SHFTER` and `AB`, account for most of these costs. These two arrays are accessed in many different places in the code (with nearly equal cost for most of the code locations), and we have not yet attempted to improve those sections of the code.

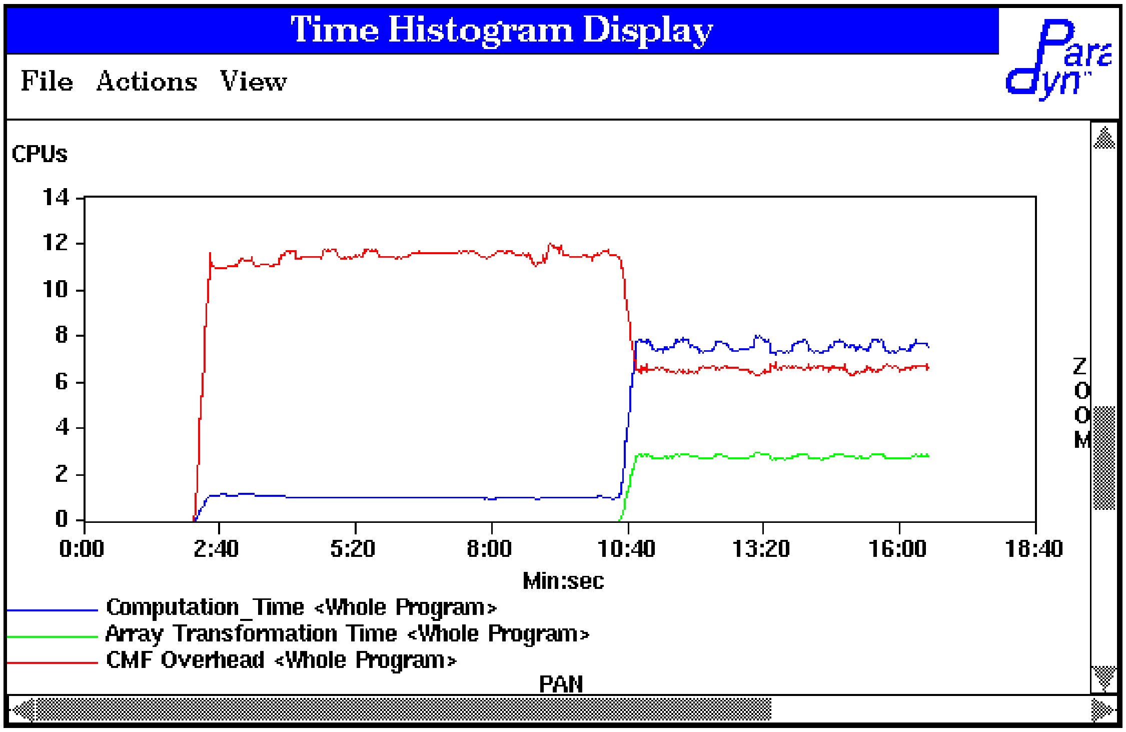

Figure 15: CMF-level Performance for Peace

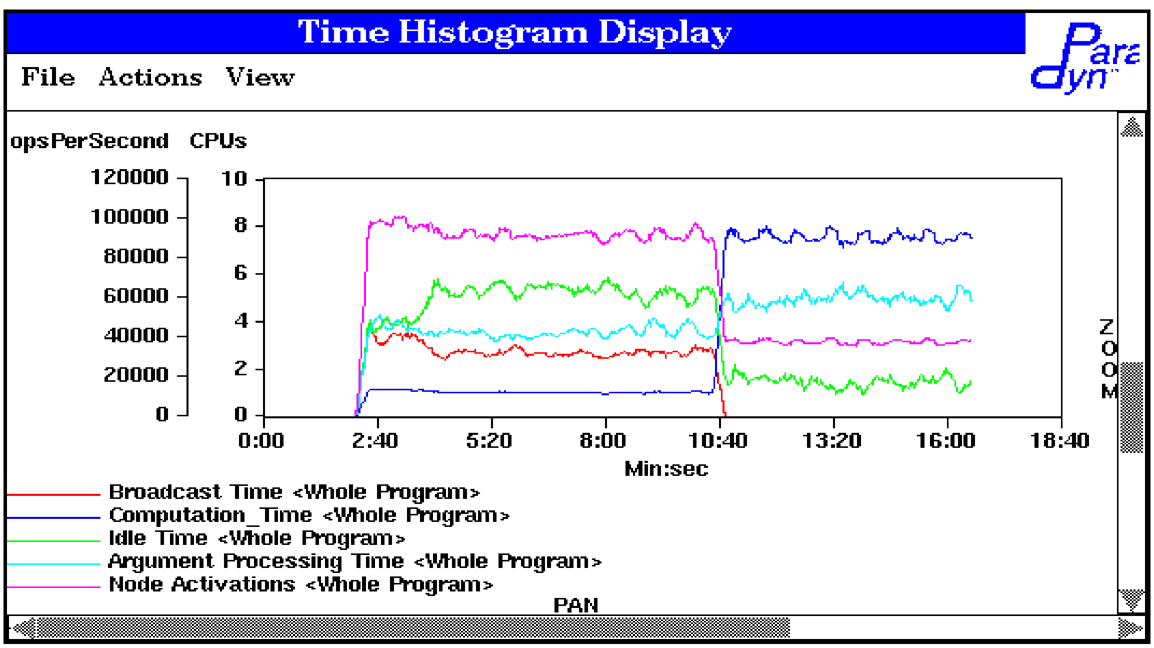

Figure 16: Node-level View of Performance for Peace

In this experiment, source-level views of performance, lower-level metrics, and data views of performance helped us to find and fix a significant performance problem in the Peace application. All three techniques helped to point us to a region of execution that performed poorly, a Node-level symptom of poor performance, and a CMF-level cause of Node-level activities. As in the Bow application, data views of performance helped us more than code views as the performance problem was localized to a few arrays but diffused over many statements of code.

4.3 Particle Simulation (PSICM)

We also studied a third application, called PSICM, a three-dimensional particle simulation code used in computational fluid dynamics studies of hypersonic flow and studies of plume flowfields [4,3]. The code consists of over 6500 lines of code in eleven source files. Input sets specify the geometry of the flow field, the number of particles simulated, and other factors. The code was written by a group of computational scientists at the Numerical Aerodynamic Simulation Laboratory of NASA Ames Research Center. We briefly summarize our experience with PSICM; a full description of this study can be found elsewhere [15].

Our analysis of PSICM showed that combinations of code and data views of performance could be very useful for the performance analysis of data-parallel programs. In particular, we identified a single statement that caused large amounts of implicit Node-level activity and explained the cause of the activity. This activity was caused by a check point operation that caused excessive communication on the slow path from compute nodes to Control Processor. The combination of code and data views showed us that the array elements were transferred one at a time. While the control view showed that the communication was expensive, the data view showed that the different arrays involved in the communication were being serialized. As a result of this information we reallocated the data structures to block-transfer them as a single unit and thereby reduce the Node-level implicit communication costs associated with the I/O operations.

5 CONCLUSIONS

Our work on NV has convinced us that multiple levels of performance data, combined with control and data views are important in identifying performance bottlenecks in programs written in these high-level parallel languages. The mechanisms and techniques for mapping low-level performance information to higher-level control and data views are the crucial enabling technologies.

The mechanisms for mapping are crucial. We must insure that each lower-level activity can be mapped to a high-level (source language) operation. We must also be able to map high-level behavior can be mapped to its component operations. The Set of Active Sentences is a useful first technique in performing these mappings. More work is needed to make these techniques more widely applicable.

A difficult problem in any high-level language or library environment is finding the mapping data. Different compilers encode similar information in different ways. For example, array geometry and distribution information in CM Fortran is contained in array descriptors (which can be read at run-time), while the same information in Fortran D is contained in the compiler-generated code. Other compiler-generated information (such as the location of statements or variable use/def information), if available at all, is in a variety of places in a variety of formats. Some effort to standardize access to this information is crucial to the development of general purpose debugging or profiling tools.

Dynamic instrumentation in the Paradyn tool has proven to be a useful vehicle for implementing ideas of the NV model. We have incorporated the ideas of source-level performance data, mapping performance data between layers of abstraction, and data views of performance into the Paradyn performance tool and have used the tool to gain substantial improvements in real parallel applications.

Through our experiments with the Paradyn tool and CM Fortran applications, we have found that CMF-level views are important but that they must be augmented with RTS-level and Node-level views of performance when applications exhibit performance problems in the runtime system or on the compute nodes. The experiments with the Bow and Peace applications demonstrate that sometimes only a small fraction of total available resources are used for CMF-level computations. In these cases, we studied lower levels to better understand the activities of the entire system.

However, information about lower-level activity may not be useful unless programmers can relate the activity to their source code. In our experiments, we have found that RTS-level and Node-level activity can be very useful when constrained to CM Fortran code and data structures. In each of our applications, code and data views of performance helped to identify the CMF-level causes of RTS-level and Node-level activities.

Traditionally, performance data has been explained in terms of code constructs, but we have shown that data views of performance can lead to more focused explanations of performance. In our experiments we found performance problems to be localized among a few parallel arrays while being diffused among many code statements. We believe that data views of performance will often be helpful in understanding data-parallel programs because data is the primary source of parallelism and synchronization in such programs. In the particular cases that we have studied, the applications allocated large data structures at the beginning of execution and then used many routines to transform the data structures. The result is that performance problems often can be found and fixed by concentrating on a program's use of data structures as well as code structures.

6 BIBLIOGRAPHY

[1] Vikram S. Adve, Jhu-Chun Wang, John Mellor-Crummey, Daniel A. Reed, Mark Anderson, and Ken Kennedy. An integrated compilation and performance analysis environment for data parallel programming. Technical Report 94513-S, CR-PC, 1994.

[2] Francois Bodin, P. Beckman, Dennis Gannon, J. Gotwals, S. Narayana, S. Srinivas, and B. Winnicka. Sage++: An object oriented toolkit and class library for building fortran and C++ restructuring tools. In *OONSKI 1994*, 1994.

[3] Leonardo Dagum. Three-dimensional direct particle simulation on the connection machine. Technical Report RNR-91-022, NASA Ames Research Center, 1991.

[4] Leonardo Dagum and S.H. Konrad Zhuh. Three-dimensional particle simulation of high altitude rocket plumes. Technical Report RNR-92-026, NASA Ames Research Center, 1992.

[5] William DePauw, Richard Helm, Doug Kimelman, and John Vlissades. Visualizing the behavior of object-oriented systems. In *Object-Oriented Programming Systems, Languages, and Applications Conference*, pages 326–337, May 1993.

[6] A. J. Goldberg and John Hennessy. Performance debugging shared memory multiprocessor programs with mtool. In *Supercomputing 1991*, pages 481–490, November 1991.

[7] S. L. Graham, P. B. Kessler, and M. K. McKusick. Gprof: A call graph execution profiler. In *ACM SIGPLAN Symposium on Compiler Construction*, June 1982.

[8] Anoop Gupta, Margaret Martonosi, and Tom Anderson. Memspy: Analyzing memory system bottlenecks in programs. *Performance Evaluation Review*, 20(1):1–12, June 1992.

[9] V. Haarslev and R. Moller. A framework for visualizing object-oriented systems. In *Object-Oriented Programming Systems, Languages, and Applications Conference*, pages 237–244, May 1990.

[10] Michael T. Heath and Jennifer A. Etheridge. Visualizing the performance of parallel programs. *IEEE Software*, 8(5):29–39, September 1991.

[11] High Performance Fortran Forum. *High Performance Fortran Language Specification - Version 1.0*, January 1993.

[12] Jeffrey K. Hollingsworth and Barton P. Miller. Dynamic control of performance monitoring on large scale parallel systems. In *7th ACM International Conference on Supercomputing*, pages 185–194, July 1993.

[13] Jeffrey K. Hollingsworth, Barton P. Miller, and Jon Cargille. Dynamic program instrumentation for scalable performance tools. In *Scalable High Performance Computing Conference*, May 1994.

[14] R. Bruce Irvin and Barton P. Miller. A performance tool for high-level parallel programming languages. In Karsten M. Decker and Rene M. Rehmann, editors, *Programming Environments for Massively Parallel Distributed Systems*, pages 299–314. Birkhauser Verlag, 1994.

[15] R. Bruce Irvin. Performance tool for high-level parallel programming languages. *Ph.D. Dissertation*, University of Wisconsin-Madison, November 1995.

[16] M. F. Kleyn and P. C. Gingrich. Graphtrace - understanding object-oriented systems using concurrently animated views. In *Object-Oriented Programming Systems, Languages, and Applications Conference*, pages 191–205, May 1988.

[17] Alvin R. Lebeck and David A. Wood. Cache profiling and the spec benchmarks: A case study. *IEEE Computer*, 27(10):15–26, October 1994.

[18] MasPar Computer Corporation, 749 North Mary Avenue, Sunnyvale, CA. *MPPE Reference Manual*, 1991.

[19] Barton P. Miller, Mark D. Callaghan, Jonathan M. Cargille, Jeffrey K. Hollingsworth, R. Bruce Irvin, Karen L. Karavanic, Krishna Kunchithapadam, and Tia Newhall. The Paradyn parallel performance measurement tools. *IEEE Computer*, 28(11), November 1995.

[20] Bernd Mohr, Darryl Brown, and Allen Malony. Tau: A portable parallel program analysis environment for pC++. In *International Conference on Parallel Systems*, pages 29–40. Springer Verlag, September 1994.

[21] D. W. Peaceman and H. H. Rachford. The numerical solution of parabolic and elliptic differential equations. *Journal of the Society of Industrial Applied Mathematics*, 3(1):28–33, 1955.

[22] Donald W. Peaceman. *Fundamentals of Numerical Reservoir Simulation*. Elsevier Scientific Publishing Company, 1977.

[23] Pure Software Incorporated, Menlo Park, CA. *Quantify User's Guide*, 1993.

[24] Daniel A. Reed, Robert D. Olson, Ruth A. Aydt, Tara M. Madhyastha, Thomas Birkett, David W. Jensen, Bobby A. Nazief, and Brian K. Totty. Scalable performance analysis: The Pablo performance analysis environment. In A. Skjellum, editor, *Scalable Parallel Libraries Conference*. IEEE Computer Society, 1993.

[25] Robert T. Schumacher. Self-sustaining oscillations of the bowed string. *Acustica*, 43:109, 1979.

[26] Robert T. Schumacher. Analysis of aperiodicities in nearly periodic waveforms. *Journal of Acoustic Society of America*, 91:438, 1992.

[27] Steve Sistare, Don Allen, Rich Bowker, Karen Jourdenais, Josh Simons, and Rich Title. Data visualization and performance analysis in the Prism programming environment. In *Programming Environments for Parallel Computing*, pages 37–52. North-Holland, 1992.

[28] Thinking Machines Corporation, Cambridge MA. *CM Fortran Reference Manual*, January 1991.

[29] Winifred Williams, Timothy Hoel, and Douglas Pase. The MPP Apprentice performance tool: Delivering the performance of the Cray T3D. In Karsten M. Decker and Rene M. Rehmann, editors, *Programming Environments for Massively Parallel Distributed Systems*. Birkhauser Verlag, 1994.