Mechanisms for Mapping High-Level Parallel Performance Data

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

A primary problem in the performance measurement of high-level parallel programming languages is to map low-level events to high-level programming constructs. We discuss several aspects of this problem and presents three methods with which performance tools can map performance data and provide accurate performance information to programmers. In particular, we discuss static mapping, dynamic mapping, and a new technique that uses a data structure called the set of active sentences. Because each of these methods requires cooperation between compilers and performance tools, we describe the nature and amount of cooperation required. The three mapping methods are orthogonal; we describe how they should be combined in a complete tool. Although we concentrate on mapping upward through layers of abstraction, our techniques are independent of mapping direction.

1 INTRODUCTION

When application programs are built on multiple layers of abstraction, performance tools must consider how the elements of one layer relate to the elements of the other layers. *Mapping* provides a way to represent the relations between abstraction levels for the performance characteristics of program elements. Any performance information that is measured for at one of abstraction is relevant not only to itself, but also to the other levels to which it maps.

To identify performance characteristics that are common across programming models, we have developed a framework within which we can discuss performance characteristics of programs written in these programming models. This framework is called the Noun-Verb (NV) model for parallel program performance explanation. In the NV model, *nouns* are any program elements for which performance measurements can be made, and *verbs* are any potential actions that might be taken by a noun or performed on a noun. For example, in CM Fortran[11] nouns include programs, subroutines, FORALL loops, arrays, and statements. Verbs in CM Fortran include statement *execution*, array *assignment* and *reduction*, subroutine *execution*, and file *I/O*.

An instance of a 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. *Performance information* consists of the aggregated costs measured from the execution of a collection of sentences.

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.

To build mappings layers of abstraction, performance tools must collect mapping information; such information can take many forms in real systems. Many compilers emit symbolic debugging information, which allows programming tools to map memory addresses to source code lines and data structures. However, common symbolic debugging information seldom provides the complete set of mapping data needed by performance tools. For example, a list of data structures used on each line of code (which is useful for mapping execution activity to data structures) is

. 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.

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 1: Types of Upward Mappings

typically not available. Other mapping information is stored only in application data-structures during execution. For example, a run-time system may determine data-to-processor mappings at run time after it has knowledge of available hardware resources; run-time systems usually keep this information in the program's address space. Traditionally, there has been no well-defined way for run-time systems and application programming libraries to communicate mapping information to performance tools.

Mappings can be one-to-one, one-to-many, many-to-one, and many-to-many, as shown in Figure 1. This figure shows examples of each type of mapping. One-to-one mappings (shown in the first row of the table in Figure 1) are relatively simple to handle in a performance tool. Any performance information measured for one sentence is associated with the one sentence to which it maps. However, when a sentence maps to several other sentences (one-to-many, shown in the second row), the correct assignment of performance data is more difficult. In this case, many tools split the measured data equally across all sentences to which the measured sentence maps [1,9]. However, such splitting assumes an equal distribution of low-level work to high-level code. It is often better to handle one-to-many mappings by merging the sentences to which the measured sentence maps [6]. The latter technique (used in the Paradyn Performance Tools[8]) makes no assumption about the distribution of performance data and helps to identify high-level programming constructs whose implementations have been merged by an optimizing compiler. It also avoids misleading the programmer with overly precise information.

Many-to-one and many-to-many mappings (shown in the third and fourth rows of Figure 1) can be reduced to the two types of mappings described above. In each case, we aggregate (either sum or average) the performance data for the low-level sentences and then treat the result as a one-to-one or one-to-many mapping. We show examples of each of these cases in Sections 3 and 4.

2 TYPES OF MAPPING INFORMATION

Mapping information may include noun and verb definitions as well as detailed descriptions of how particular nouns and verbs map to other nouns and verbs. In this section we describe a generic interface for communicating mapping information to performance tools. In following sections we describe how this information may be communicated from compilers to tools both prior to application execution (static information) and during execution (dynamic information)

The table in Figure 3 shows three components of mapping information. Noun and verb definitions describe to a performance tool the set of nouns, verbs, and levels of abstraction contained in an application. Mapping definitions are equivalence classes for performance data. Performance data collected for the source sentence can be presented in relation to either the source sentence or the destination sentence.

Our simple definition of mapping information can handle all the types mappings listed in Figure 1. For example, we can build a many-to-one mapping by defining many mappings from different source sentences to one destination sentence. We can build one-to-many and many-to-many mappings from similar combinations of our basic one-to-one mapping definition. The differences among the four types of mappings can then be exploited and interpreted by any performance tool that uses the mappings.

3 STATIC MAPPING INFORMATION

Static mapping information is any mapping information

NOUN name = line1160 abstraction = CM Fortran description = line #1160 in source file /usr/src/prog/main.fcm
NOUN name = line1161 abstraction = CM Fortran description = line #1161 in source file /usr/src/prog/main.fcm
VERB name = Executes abstraction = CM Fortran description = units are “% CPU”
NOUN name = cmpe_corr_6_() abstraction = Base description = compiler generated function, source code not available
VERB name = CPU Utilization abstraction = Base description = units are “% CPU”
MAPPING source = {cmpe_corr_6_(), CPU Utilization} destination = {line1160, Executes}
MAPPING source = {cmpe_corr_6_(), CPU Utilization} destination = {line1161, Executes}

Figure 2: Examples of Static Mapping Information

provided to a performance tool prior to the execution of an application program. To illustrate how we might use static mapping information, we present an example in Figure 2. This figure shows a subset of static mapping information for a CM Fortran program. The mapping information defines a mapping between a compiler generated function and two CM Fortran source code lines. The first three records define two source-level nouns (line1160 and line1161) and a source-level verb (Executes). The next two records defines a Base level noun (the compiler generated function cmpe_corr_6_()) and verb (CPU Utilization). Finally, the last two records define mappings between CPU Utilization in the base level function and execution of the source code lines.

The mapping information indicates that the two statements on lines 1160 and 1161 of the source code are implemented by a single low-level routine, and that if our performance measurement tool can measure CPU Utilization for cmpe_corr_6_(), then it can present that information as Execution of the corresponding source code lines. A performance tool may then split the execution costs between the two source code lines, merge the two lines into an inseparable unit, or make other interpretations of the mappings.

Static mapping information may be kept in an application program’s executable image, in a separate file, in an auxiliary database, or in some other static location. Regardless of its location, the mapping information must be communicated to performance tools before they can use mappings for high-level abstractions.

The method of communicating static mapping information discussed in this section provides a simple method with which compilers can describe important language-specific and program-specific information to performance tools. Because such information is defined statically, performance tools can process it before or after the execution of the application program and avoid competition for resources with the application program. However, static mapping information usually cannot provide information about mappings that are determined during application execution.

Type of Information	Description
Noun definition	name level of abstraction descriptive information
Verb definition	name level of abstraction descriptive information
Mapping definition	source sentence destination sentence

Figure 3: Types of Mapping Information

4 DYNAMIC MAPPING INFORMATION

Dynamic mapping information includes any mapping information that is generated during application execution. It includes the same types of information as static mapping information (see Figure 3), and differs with static mapping information only in that is communicated to performance tools 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 processing nodes, then the application must dynamically define a mapping between the object and processor nodes for the performance tool. The performance tool can use the dynamic mapping information during or after run time to relate performance measurements to abstract program constructs and activities.

In this section we discuss two important techniques for collecting dynamic mapping information. The first uses dynamic instrumentation [5] to reduce the perturbation effects of collecting dynamic mapping information, and the second uses a data structure called the Set of Active Sentences to discover verb mappings that are otherwise difficult to detect.

4.1 Using Dynamic Instrumentation

A *mapping point* is any function, procedure, or line of code in an application where dynamic mappings may be constructed. For example, if we have a run-time system routine that allocates parallel data objects and distributes them across processors, then the return point of the routine would be defined as a mapping point; the mapping of data objects to processor nodes will be determined just prior to that point. Our goal is to identify all such mapping points in an application, and instrument them with code that reports mapping information to our performance tool. We can instrument all such points by adding source code that calls our performance tool, or we can use dynamic instrumentation to insert the mapping instrumentation at run time.

Dynamic instrumentation[5] is a technique whereby an external tool changes the binary image of a running executable to collect performance data. The basic technique defines *points* at which instrumentation can be inserted, *predicates* that guard the firing of the instrumentation code, and *primitives* that implement counters and timers. Dynamic instrumentation provides an advantage over traditional static techniques because it allows performance tools to instrument only those points that are currently needed to provide performance data. Any point that does not contain instrumentation does not cause any execution perturbations.

For dynamic mapping instrumentation, we can define a subset of points consisting of all those points that generate mapping information. Typically, the subset is different for each language, or programming library and includes the return points for all subroutines in which data structures are allocated or in which distributions to parallel processors are determined. As an application executes, a performance tool can either insert mapping instrumentation once at the beginning of execution and leave it in, or it can insert and delete mapping instrumentation throughout execution. The latter technique reduces run-time perturbation but may miss mapping decisions or noun/verb definitions.

4.2 The Set of Active Sentences

Some dynamic mapping information is difficult to determine by simply 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. In this section we describe the Set of Active Sentences (SAS), a data structure that allows us to dynamically map concurrent sentences between layers of abstraction. We describe the SAS with an example taken from High Performance Fortran, describe the kinds of questions that might be asked and answered with the SAS, and describe limitations of the SAS approach.

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

Figure 4: Example HPF Code

4.2.1 Description of 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.

For example, consider the example HPF code fragment in Figure 4. In this code, we are concerned with the following problem: how to relate a low-level message to a high-level array reduction. The `SUM` reduction on line 1 and the `MAXVAL` reduction on line 2 of the code 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 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, what reduction is being performed on the array).

We 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`?

Although these questions are specific to data-parallel Fortran and in particular to the HPF code in Figure 4, they are representative of questions that we would like to ask for any language 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). .

HPF: line #1 executes

HPF: A sums

Base: Processor sends a message

(each line represents one active sentence)

Figure 5: The SAS When a Message is Sent

In the SAS approach to dynamic mapping, we defer the asking of performance questions until run time, and then only measure those sentences that help to 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. 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 5 shows the contents of a hypothetical SAS for our example HPF code

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 [2,3,4,7,10]. 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.

As defined, the SAS contains *all* sentences that are active. If we wish to reduce the size of the SAS, we can also take advantage of run-time requests for performance information [8] to eliminate uninteresting sentences from the SAS. For example, if we only ever request measurements for array `A`, then the SAS may avoid keeping sentences that do not contain `A`.

Performance Questions	Meaning
`{A Sum}`	Cost of summations of A?
`{Processor_P Send}`	Cost of sends by processor P?
`{A Sum}`, `{Processor_P Send}`	Cost of sends by P *while* A is being summed?
`{? Sum}`, `{Processor_P Send}`	Cost of sends by P *while* anything is being summed?

Figure 6: Example Performance Questions

4.2.2 Performance Questions

The SAS can also keep track of performance questions if they are asked using nouns and verbs. We define a performance question to be a vector of sentences. The meaning of a performance question is that performance measurements (of resource utilization) should be made only when all of the sentences of the question are active. Figure 6 shows a few of the possible performance questions (and their meanings) for our example HPF code. Although the questions in the figure consist of sentences that contain one noun and one verb, we can easily generalize questions to use more complex sentences without altering the operation of the SAS.

Monitoring code may use the SAS to answer the types of questions listed in Figure 6. Each component of a performance question represents a predicate that must be satisfied before monitoring code can measure CPU time, wall-clock time, channel bandwidth, or any other execution cost for the question.

We can make the SAS more flexible by extending our definition of performance questions. This extension would include boolean disjunction and negation incurring only the added cost of evaluating more complex expressions.

4.2.3 Distributed Memory

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. For example, all of the performance questions listed in Figure 6 can be answered without sharing any information between nodes.

Of course, some interesting performance questions can only be answered using information about sentence activity on more than one node. For example, in a distributed database system, if a server process performs disk reads on behalf of clients, then we may wish to measure server disk reads that correspond to a particular client or a particular query. The SAS information that is necessary to answer such a performance question (*server reads from disk*, *client query is active*) would be distributed between the SAS on the client and the SAS on the server. The client's SAS and the server's SAS would need to communicate before the performance question could be answered. In particular, the client's SAS would need to send one sentence (i.e., *client query is active*) to the server's SAS whenever that sentence became active or inactive.

4.2.4 Limitations of the SAS Approach

The SAS approach to relating low-level performance information to high-level activities has at least three limitations.

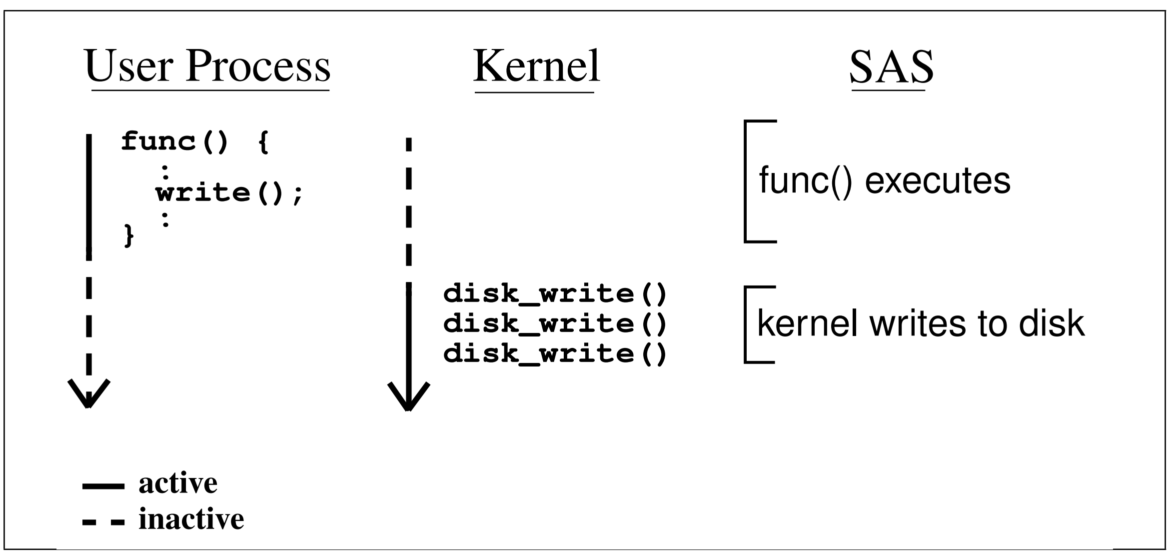

Figure 7: Asynchronous Sentence Activations and the SAS (time advances downward)

First, the SAS approach does not handle asynchronous activation of sentences. 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. Figure 7 shows time-lines for a hypothetical UNIX process and kernel. The user process makes a `write()` system call to the kernel and the kernel later writes the information to disk. The actual writes to disk do not occur until later. The third column of the figure shows how the SAS records each of these activities. As the figure shows, the SAS may not contain both the function execution sentence and the kernel disk write sentence at the same time, and therefore kernel disk writes on behalf of function `func()` could not be measured with the help of the SAS alone.

Second, sentence activity notifications that are ignored by the SAS cause unnecessary execution costs. For our example code from Figure 4, if we only ask performance questions about array `A`, then all activation notifications about array `B` are ignored by the SAS. But we must pay the run-time cost of the notification. We could eliminate this cost by dynamically removing such notifications from the executing code [5].

Third, sentences are not ordered in performance questions. For our current definition of performance question, the question "How many messages are sent for the summation of `A`?" is syntactically equivalent to the question "How many summations of `A` occur when messages are sent?" If we were to take advantage of sentence order in performance questions, then we could distinguish between these two very different performance questions.

5 STUDY: CM-FORTRAN AND PARADYN

Paradyn is a performance measurement tool that uses dynamic instrumentation to measure only the performance data requested by users. Paradyn starts an application executing, waits for user requests to measure performance metrics, instruments the running application (usually by rewriting the application's executing binary image), and then sends a stream of performance measurements back to the user. By limiting its instrumentation to only requested data, Paradyn can greatly reduce instrumentation intrusion and allows users to measure large, long-running applications on large-scale parallel computers. Paradyn includes performance display modules that allow users to view performance metric streams graphically during the execution of their applications. Paradyn also includes an automated module (called the Performance Consultant) to help users find performance problems in their applications.

Paradyn receives information about new levels of abstraction, new resources, and new metrics from two mapping information interfaces. Paradyn daemons import static mapping information via Paradyn Information Format (PIF) files just after they load each application executable. PIF files are emitted by compilers, programming environments, or other external sources that wish to define source-level language code and data objects that are contained in an application. PIF files allow such tools to explain to Paradyn how it should map requests for high-level language resources and metrics into requests for base resources and metrics such as functions and CPU time. The PIF format also allows external tools to communicate descriptive information about resources and metrics to Paradyn. In this way, language-dependent and application-dependent visualization modules can receive descriptive information to add meaning to visual displays.

The Paradyn dynamic instrumentation library sends dynamic mapping information to the Paradyn daemon process using the same communication channel used for performance data. The dynamic instrumentation library, linked into every application program that is measured by Paradyn, contains interface procedures that allow the application to describe mappings while it executes. The dynamic instrumentation library sends the mapping information to the Paradyn daemons, and the daemons forward the mapping information to the Data Manager. The Data Manager uses the dynamic mapping information in exactly the same way as it uses static mapping information. .

Paradyn uses dynamic performance instrumentation techniques to turn on or turn off the flow of dynamic mapping information. Dynamic instrumentation allows applications to avoid the cost of emitting mapping information when they are not run with Paradyn and allows Paradyn users to turn off mapping information collection when it is not needed. Currently, Paradyn allows users to turn on or turn off all dynamic mapping instrumentation points at once. Eventually, we could tie the enabling and disabling of individual mapping instrumentation points to requests for performance information.

6 CM FORTRAN-SPECIFIC RESOURCES AND METRICS

Using the static and dynamic mapping interfaces described in Sections 3 and 4, Paradyn measures important resources and metrics that are unique to CM Fortran and the CM Runtime System (CMRTS). In this section we describe the details of how Paradyn measures performance data for CM Fortran's parallel assignment statements and parallel arrays and measures CMRTS-specific activities such as Broadcast Messages, Point-to-Point Messages, Reductions, and Argument Processing.

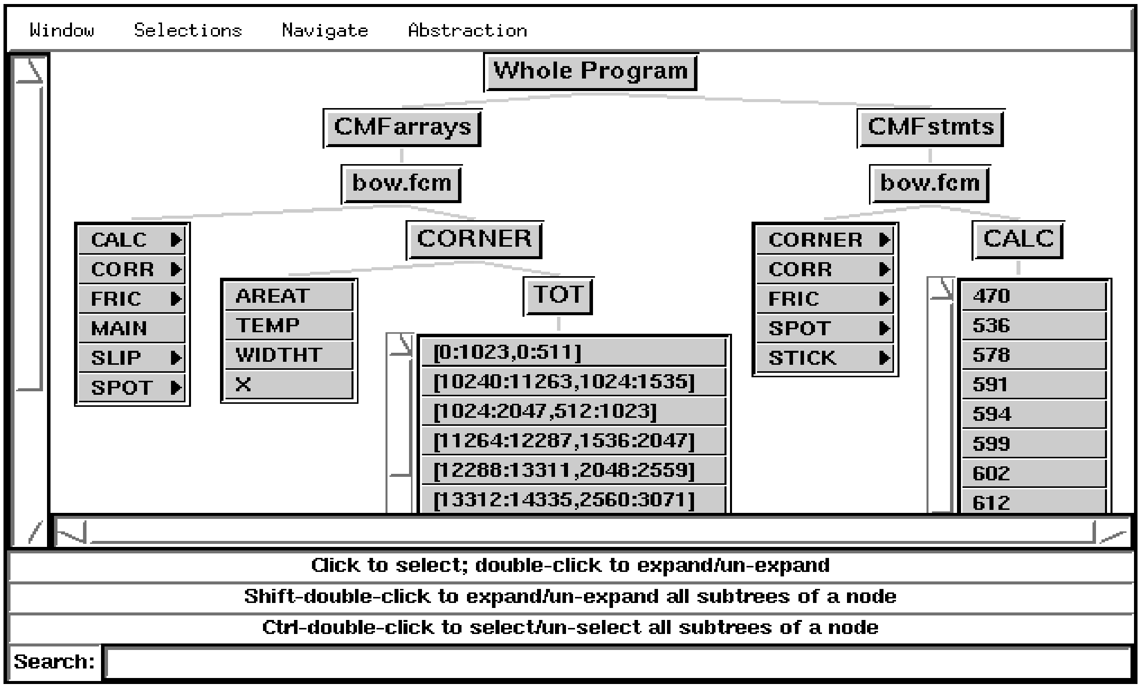

Figure 8: CMF-Level Where Axis

6.1 Performance Data for Parallel Arrays

Arrays are the fundamental source of parallelism in data-parallel CM Fortran. They are the only data objects that use memory on the nodes of a CM-5 system, and the performance of any particular CM Fortran program depends greatly on its efficiency of computation and communication of arrays.

Paradyn measures CM Fortran arrays in a two step process. First, Paradyn's dynamic instrumentation library detects array allocations (and deallocations) and forwards resource and mapping information to Paradyn. When an array is allocated (via a call to a particular CMRTS allocation routine) the dynamic instrumentation library notifies Paradyn of the new array, establishes a unique identifier for the array, and tells Paradyn (via the dynamic mapping interface described in Section 5) which subregion of the array is stored on which node of the system. Paradyn uses this information to build a CMFarrays hierarchy as shown in Figure 8. The figure shows that the module *bow.fcm* contains six functions, and one of those (*CORNER*) contains five arrays. One of the arrays within CORNER (called *TOT*) has been expanded to show its subregions

The second step occurs when the user requests a performance metric for a particular array. When the user chooses an array to measure, Paradyn's Data Manager maps the array to the proper CMRTS identifier and system node, and sends a message (via the same parallel debugging interface used for dynamic instrumentation) to a Set of Active Sentences (or SAS, described in Section 4.5) module located on the appropriate node of the system. The SAS module then sets a boolean variable to *true* whenever the requested array is active and sets the variable to *false* when the array becomes inactive. The CMRTS node code block dispatcher notifies the SAS of array activation/deactivation by sending the input arguments for each node code block to the SAS. The SAS only searches the arguments for those arrays that are requested by Paradyn.

To collect metrics, Paradyn dynamically inserts instrumentation code into node-level subroutines. If a metric is to be measured for an array, then the dynamically-inserted instrumentation code checks the array's node-global boolean variable (discussed above), before measuring the metric. Paradyn can thereby constrain any metric to an array of interest.

Paradyn can easily use its existing visualization modules (time plots, bar charts, and tables) for visual display of performance information for data objects. These visualization modules simply treat a data object as a resource like any other. However, Paradyn's visualization interface is open; we could build specialized visualization modules to take advantage of properties (such as geometric structure) that are unique to arrays.

6.2 Parallel Code Constructs

Parallel code constructs allow CM Fortran programmers to manipulate parallel arrays. Code constructs include parallel assignment statements, `FORALL` iterators, and intrinsic operations such as `SUM`, `MIN`, and `TRANSPOSE`.

Paradyn measures parallel code constructs by mapping each statement to the node code blocks that implement it. Paradyn receives this mapping information via PIF files as described in Section 5. We create CM Fortran PIF files with a simple utility that parses CM Fortran compiler output files. The utility scans the compiler output files for lists of parallel statements, parallel arrays, and node-code blocks. It then produces a PIF file that defines the statements and arrays for Paradyn and describes the mappings from statements to code blocks.

Paradyn's user interface displays statements in the *CMFstmts* hierarchy within the where axis display, as shown in Figure 8. Users may interact with the where axis display to choose resources from the CMFstmts hierarchy, from the CMFarrays hierarchy, or from a combination of the two hierarchies. Users may also choose resources from hierarchies for the CMRTS-level of abstraction, or the base level of abstraction.

6.3 CMRTS Metrics

Paradyn's dynamic instrumentation system includes a language for describing how to measure new metrics. This language (called Metric Description Language, or MDL) allows users to precisely specify when to turn on/off process-clock timers and wall-clock timers and when to increment and decrement counters. Paradyn compiles the descriptions into code that is inserted into running applications at precisely the moment when the particular metric is requested.

We have used MDL to define many new metrics that are specific to CM Fortran and CMRTS. Some of these are shown in the table in Figure 9. The table lists the name of each metric and a brief description of what the metric measures. Each of these metrics can be constrained to parallel arrays, subsections of arrays, parallel assignment statements, or combinations of assignment statements and arrays. Together, the metrics cover most of the activities (or verbs) necessary to understand the performance of CM Fortran applications.

7 SUMMARY

We have described the important problems of collecting, storing, communicating, and using mapping information in performance tools for high-level parallel programming languages. We have described a format and interface for static and dynamic mapping information, and we have presented the Set of Active Sentences as a method for identifying complex dynamic activity mappings.

8 BIBLIOGRAPHY

[1] V.S. Adve, J.-C. Wang, J. Mellor-Crummey, D.A. Reed, M. Anderson, and K. Kennedy. An integrated compilation and performance analysis environment for data parallel programming. Technical Report 94513-S, CRPC, 1994.

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

[3] 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.

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

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

[6] 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.

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

[8] 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.

[9] 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.

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

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

	Metric	Description
CM-Fortran (CMF) Level	Computations Computation Time	Count of computation operations. Time spent computing results.
	Reductions Reduction Time Summations Summation Time MAXVAL Count MAXVAL Time MINVAL Count MINVAL Time	Count of array reductions. Time spent reducing arrays. Count of array summations. Time spent summing arrays. Count of `MAXVAL` reductions. Time spent computing `MAXVAL`s. Count of `MINVAL` reductions. Time spent computing `MINVAL`s.
	Array Transformations Transformation Time Rotations Rotation Time Shifts Shift Time Transposes Transpose Time	Count of array transformations. Time spent transforming arrays. Count of array rotations. Time spent of rotations. Count of array shifts. Time spent shifting arrays. Count of array transposes. Time spent transposing arrays.
	Scans Scan Time	Count of array scans. Time spent scanning arrays.
	Sorts Sort Time	Count of array sorts. Time spent sorting arrays.
CM-Runtime (CMRTS) Level	Argument Processing Time	Time spent receiving arguments from CM-5 control processor.
	Broadcasts Broadcast Time	Count of broadcast operations. Time spent broadcasting.
	Cleanups Cleanup Time	Count of resets of node vector units. Time spent resetting node vector units.
	Idle Time	Time spent waiting for control processor.
	Node Activations	Count of node activations by control processor.
	Point-to-Point Operations Point-to-Point Time	Count of inter-node communication operations. Time spent sending data between parallel nodes.

Figure 9: Paradyn metrics for CM Fortran Applications