A Performance Tool for High-Level Parallel Programming Languages

R. Bruce Irvin
rbi@cs.wisc.edu

Barton P. Miller
bart@cs.wisc.edu

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

Abstract

Users of high-level parallel programming languages require accurate performance information that is relevant to their source code. Furthermore, when their programs cause performance problems at the lowest levels of their hardware and software systems, programmers need to be able to peel back layers of abstraction to examine low-level problems while maintaining references to the high-level source code that ultimately caused the problem. In this paper, we present NV, a model for the explanation of performance information for programs built on multiple levels of abstraction. In NV, a level of abstraction includes a collection of nouns (code and data objects), verbs (activities), and performance information measured for the nouns and verbs. Performance information is mapped from level to level to maintain the relationships between low-level activities and high-level code, even when such relationships are implicit.

We have used the NV model to build ParaMap, a performance tool for the CM Fortran language that has, in practice, guided us to substantial improvements in real CM Fortran applications. We describe the design and implementation of our tool and show how its simple tabular and graphical performance displays helped us to find performance problems in two applications. In each case, we found that performance information at all levels was most useful when related to parallel CM Fortran arrays, and that we could subsequently reduce each application's execution time by more than half.

This research was supported in part by Department of Energy grant DE-FG02-93ER25176, Office of Naval Research grant N00014-89-J-1222, and National Science Foundation grants CCR-9100968 and CDA-9024618.

1. Introduction

High-level parallel programming languages promise to make programmers' lives easier. They offer portable, conceptually compact notations for specifying parallel programs, and their compilers automatically map programs onto complex parallel machines, freeing programmers from the difficult, error-prone, and sometimes ineffective task of specifying parallel computations explicitly. Unfortunately, effective performance measurement tools for high-level parallel programming languages are difficult to build because they must account for implicit low-level activities created by compilers and must present performance information about those activities in terms of the source code language.

In this paper we present NV, a new model for the explanation of performance measurements of high-level parallel applications. With NV, we organize a high-level parallel application into a set of abstraction layers that corresponds to the set of layers of software on which the application is built. Each layer of abstraction consists of a collection of nouns (representing code and data elements) and verbs (runtime actions). In NV, mapping functions relate nouns and verbs of a given layer of abstraction to nouns and verbs of other layers of abstraction. Mapping functions preserve the relationships between low-level system-dependent activities that are performed on behalf of high-level system-independent code constructs. These mapping functions allow us to get performance information not easily obtained from current tools.

Using NV as a guide, we have designed and implemented ParaMap, a performance tool for CM Fortran programs running on CM-5 systems. ParaMap has allowed us to study large and long running applications and significantly improve their execution times (73% in one case). ParaMap summarizes system performance at the source code level in terms of CM Fortran arrays, array subsections, and statements. However, if performance problems cannot be fully understood at the source code level, ParaMap allows the user to peel back layers of abstraction to view the runtime system and processor activity while retaining mappings to the appropriate CM Fortran constructs.

Section 2 of this paper describes the NV model using CM Fortran as an example language, and discusses how NV may be applied to parallel languages other than CM Fortran. Section 3 describes the design of ParaMap, and Section 4 demonstrates ParaMap with example CM Fortran programs. Section 5 discusses related approaches to the problem of providing performance information to users of high-level languages, and Section 6 summarizes and concludes the paper with a discussion of how NV based tools can utilize dynamic instrumentation and automated bottleneck searching within the context of next generation performance tools [11].

2. The NV Model

A major goal of our research is to identify performance characteristics that are common across programming models. To help achieve this goal, we have developed a framework within which we can discuss performance characteristics of programming models. This section provides an informal description of the Noun-Verb (NV) model for parallel program performance explanation. In the NV model, *nouns* are the structural elements of a particular program, and *verbs* are the actions taken by the nouns or performed on the nouns.†

A collection of nouns and verbs from a particular software or hardware layer is called 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 the notion that high-level language constructs are implemented with low-level software and hardware. With mappings, performance information collected at lower levels can be related to language level nouns and verbs. A mapping may be *static*, meaning that it is determined before runtime, or *dynamic*, meaning that it is determined at runtime and possibly changes over the course of program execution.

In describing the NV model, we use CM Fortran [34] as our example language because its characteristics are representative of many high-level parallel programming languages. CM Fortran is a data-parallel language that permits parallel, elemental operations across multi-dimensional arrays. CM Fortran extends Fortran 77 with Fortran 90 array features, and is similar to HPF [9]. The NV model, however, is applicable to many other parallel programming models including parallel object-oriented languages [5, 17, 24], functional languages [26], logic programming languages, parallel runtime systems [22, 29], coordination languages [4], and high-level application libraries [35]. We briefly describe how to model a few of these languages in Section 2.3.

2.1. Nouns and Verbs

A *noun* is any program element about 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. 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

† An alternate terminology could be *objects* and *methods*. However, we feel that these terms have been overused, so we have chosen *nouns* and *verbs*.

(line 9).

Nouns of CM Fortran include programs (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*.

```
 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

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 time, memory, channel bandwidth, etc. Finally, *performance information* consists of the aggregated costs measured from the execution of a collection of sentences. For example, performance information for array A in Figure 1 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.

2.2. Levels of Abstraction and Mapping

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. In general, 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. Furthermore, 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 (see Figure 2).† The sequential program executes on the CM-5 Control Processor and makes calls

† In this discussion, we do not consider the nodal style of execution for CM Fortran programs in which each CM-5 node executes a separate CM Fortran program.

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 processor nodes. Each parallel CM Fortran array is divided into subgrids, and each subgrid is assigned to a separate node. Each node is responsible for computations involving its local array subgrids; if array data from non-local subgrids are needed, then the non-local data must be transferred before computation can proceed.

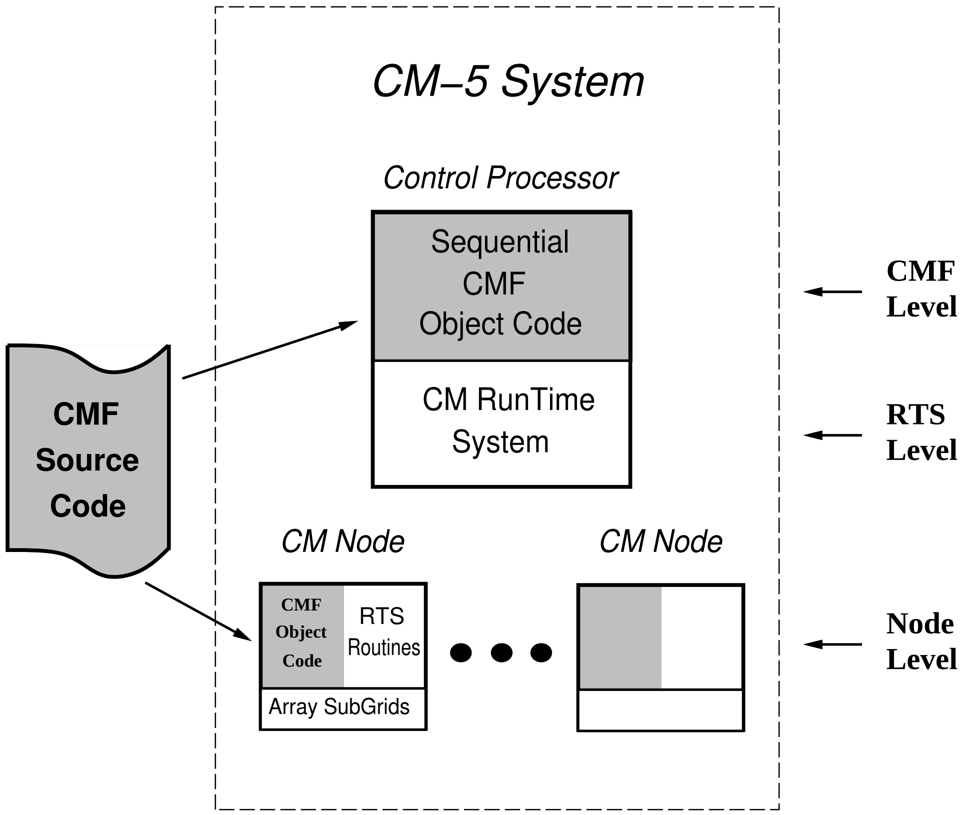

Figure 2.

CM-5 Execution of a CM Fortran Program. Shaded areas indicate user code. The labels along the right edge indicate the levels of abstraction in the NV representation of CM Fortran.

Figure 2 shows the division of the CM-5 system into three levels of abstraction for the NV model. The highest level, called the CMF level, contains the nouns and verbs from the CM Fortran language as discussed in Section 2.1. 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 processor nodes. Node level verbs include *Compute*, *Wait*, *Broadcast Communication*, and *Point-to-Point Communication*.

A *mapping* is a function relating 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 processor 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.

NV mappings are either *static* or *dynamic*. Static mappings are independent of time or context. For example, the mapping between node level object routines and CMF level statements is a static mapping that is determined at compile time. Dynamic mappings are time varying relationships. For example, CM Fortran arrays are mapped to processor nodes when they are allocated, so mappings between nodes and array sub-regions must be recorded at runtime.

Verb mappings are either *explicit* or *implicit*. An explicit mapping indicates that a high level verb is directly implemented by a lower level verb. For example, a SUM reduction (line 8 of Figure 1) is directly implemented by node level additions, so the mapping from CMF level SUM operations to node level additions is explicit. Implicit mappings indicate that a lower level verb helps to maintain the semantics of a higher level verb. For example, a SUM reduction is implemented on a CM-5 system with partial reductions on each processor node and a final reduction of the partial results using the CM-5 broadcast network. The creation of parallelism and broadcast communication caused by such a reduction is implicit because neither is specified directly by the CMF level SUM statement. Therefore, the NV mapping from CMF level SUM operations to creation of parallelism and broadcast communication is an implicit mapping.

2.3. Other Parallel Programming Models

Hundreds of parallel programming languages have been proposed and several have been implemented and even distributed for general use. We find that the NV model is useful for describing the performance aspects of particular languages as well as entire classes of languages. In this section, we briefly describe how nouns, verbs, levels of abstraction, and mappings could be used with several example languages.

Jade [16] is a task-parallel language that uses accesses to shared data structures to synchronize tasks. A Jade `withonly` block is a coarse grained task that specifies which shared objects it needs to access before executing. The Jade runtime system ensures that a `withonly` block has the appropriate access before allowing the block to execute. An NV representation of Jade might model `withonly` blocks and shared data objects as nouns and accesses to shared data objects as verbs. The execution of a `withonly` block would explicitly map to the execution of the statements within the block and accesses of the shared memory objects would implicitly map to time spent waiting for the objects.

SISAL [26] is a parallel functional language that has achieved performance comparable to Fortran on several applications. A SISAL programmer creates functional loops that expose code to parallel optimization techniques so that independent loops may be automatically scheduled on parallel processors. Nouns in a SISAL program might include functions, loops, and expressions. As with any functional language, the primary verb in SISAL is evaluation. Lower level implicitly mapped verbs include the scheduling and synchronization of the parallel loop iterations.

Portable runtime systems [22, 29] are not normally considered to be languages, but from the performance measurement point of view they share many characteristics with more traditional languages. A portable runtime system implements a particular level of abstraction and may itself be composed of multiple levels. For example, PVM uses a daemon on each workstation to set up connections and route messages. Therefore, PVM operations potentially map to multiple daemon operations, each of which effects the performance of a PVM operation. To measure a PVM system, we might measure both PVM level verbs and daemon level verbs and map between them to achieve a better understanding of performance.

3. ParaMap: A CM Fortran Performance Measurement Tool

The NV model can be used as a guide for the design and implementation of performance measurement systems for high-level parallel languages. To evaluate the NV model in this role and to gain experience with mapping in an actual parallel language system, we have built a performance measurement tool for CM Fortran. The tool, called ParaMap, measures performance information for nouns and verbs at three levels of abstraction (CMF, RTS, and node as discussed in Section 2.2) and maps performance information between levels. This section describes the design and implementation of ParaMap. In Section 4, we demonstrate the tool with actual CM Fortran applications.

3.1. Overview

The design of ParaMap follows the NV description of CM Fortran given in Section 2. Measured nouns at the CMF level of abstraction include subroutines, statements, and arrays, while verbs include array assignment and reduction, subroutine execution, and statement execution. At the RTS level, ParaMap measures the cost of array manipulations such as Shift, Copy, and Move. At the node level, ParaMap measures computation (CPU time), waiting, and broadcast communication. Point-to-point communication at the node level is difficult to measure directly, so we approximate it by measuring process time in node-level routines that perform point-to-point communications.

ParaMap uses a simple Tcl [30] interface to give users post mortem access to performance information. The user constructs sentences from nouns and verbs and asks the tool for the measured cost of the constructed sentences. Costs are provided in three formats: a count of the number of times the sentence was recorded, the total time cost of the sentence, and a time histogram showing the cost of the sentence graphed over time. Each level of abstraction is separated in the interface; the current level must be selected by the user. The user can construct sentences using only nouns and verbs at the current level.

ParaMap uses NV mappings to compute the costs of high-level sentences. When a user asks for the cost of a sentence, ParaMap automatically maps the noun and verb to lower levels and aggregates the lower level costs before returning the cost to the user. For example, if the user asks for the cost of using a particular array in assignment statements, ParaMap will map the request to explicit computations in PN code blocks as well as implicit runtime system activities such as shifting or broadcasting the array. If the user selects a sub-region of the array, then ParaMap will only provide information from the processors on which the elements of the selected subregion are stored.

ParaMap also uses NV mappings to implement contexts. A context is a set of nouns and verbs selected by the user for the purpose of constraining performance information at lower levels. For example, a ParaMap user may select the noun *Array A* and the verb *Reduction* at the CMF level and ask the tool to create a context. Then, when the user peels back layers of abstraction by moving to the RTS or node level, ParaMap uses the context to constrain the set of nouns and verbs that are visible to the user; only the nouns and verbs that map to reduction of array A will be available for constructing sentences. Furthermore, ParaMap also constrains the available performance information to that collected during reductions of array A. In this way, contexts allow users to evaluate the low-level performance impact of high-level nouns and verbs.

3.2. Implementation

A user instruments their application program by compiling with the ParaMap compiler driver. The driver uses the standard CM Fortran compiler, but also automatically inserts probes at subroutine boundaries of the sequential user object code and links the application with an instrumented version of the CM Runtime System (CMRTS). The instrumented CMRTS monitors RTS level sentences on the control processor and node level sentences on the processor nodes.

Performance information for each noun and verb is gathered in three formats: count, time, and time histogram. A time histogram [10] is a discrete representation of a performance metric over time. Because counters, timers, and time histograms are constant size formats, ParaMap can record long running executions in the same amount of space as short executions.

In addition to performance data, ParaMap collects mapping information so that node and RTS level sentences can be explained at the language level and so that CM Fortran nouns and verbs can be mapped to lower levels. Static mappings such as symbol tables and line number maps are collected during compilation, and dynamic mappings for array layouts are collected during execution.

Compiler optimizations often obscure the mappings between CM Fortran statements and PN code blocks. The CM Fortran compiler compiles each statement into several individual code blocks and co-optimizes the resulting code blocks to reduce the total cost of execution. As a result, a single PN code block may correspond to several CM Fortran statements, and performance information gathered for a particular code block must be mapped to the corresponding statements.

One way to map between CM Fortran statements and co-optimized PN code blocks is to divide an individual code block's costs among the corresponding CM Fortran statements [32]. However, this method assumes that an equal portion of a code block's computations are performed on behalf of each statement. This assumption may not hold for all optimization techniques. Instead, ParaMap joins groups of co-optimized statements into inseparable statement lists. A statement list is simply a group of statements whose corresponding PN code blocks have been optimized together. ParaMap users may not select an individual statement as a noun if it has been co-optimized with other statements; only statement lists may be selected.

One unique feature of Paramap is its ability to provide performance information for parallel CM Fortran arrays. ParaMap users may request costs for entire arrays or rectangular subgrids of arrays. The tool maps such requests to the processor nodes on which the selected array subgrid was stored. However, because ParaMap only collects performance information down to the node level of granularity, it provides accurate performance information only for array subgrids that do not partially overlap processor nodes. If the user chooses a subgrid that only partially overlaps a processor node, then ParaMap asks the user to expand or shrink the subgrid so that it fits the array distribution perfectly.

ParaMap records CMRTS array distribution data structures (called *array geometries*) to map subregions of a CM Fortran array to the processor nodes. A geometry specifies the shape and size of an array and determines how an array is laid out in memory. The ParaMap instrumentation records an array's geometry when the array is allocated and associates the geometry with the performance information recorded for the array.

On the processor nodes, each measurement of computation, waiting, or communication cost is tagged with the memory location of the routine that is executed and with the memory location of the CMF arrays being processed. The tags create a vector of performance information, one set of values for each routine executed and for each array processed on each node. ParaMap maps the node routine tags to routine names using the application's symbol table, and maps the array tags to array names with an associative table that is updated each time the control processor allocates an array.

The tags implement mappings that allow ParaMap to satisfy sentence queries that contain one noun and one verb. For example, the user can ask for node level computation that maps to array assignments involving array B. However, tag combinations are not supported because the memory required to keep counters, timers, and time histograms for all tag combinations is too large. In Section 6, we discuss how selective, dynamic instrumentation could be used to support arbitrarily complex sentence mappings.

As an application executes, the instrumentation probes update the counters, timers, and time histograms in memory until the application exits. When the application exits, the instrumented CMRTS collects the performance and mapping data from all of the processors and stores it in a file for post mortem analysis.

4. Experience

In this section we present results from using ParaMap to study the performance of two CM Fortran applications. The first study examines a toy example and illustrates a performance problem that can be found in many CM Fortran applications. The second study examines a real chemical engineering code and demonstrates how performance information attributed to arrays can be more useful than information attributed to code statements. In each case, we compiled the application with maximum compiler optimizations, linked it with ParaMap instrumentation, executed it on a 32 node CM-5, studied performance information at multiple levels of abstraction, and finally changed the source code to reduce runtime. These studies show that even a simple tool based on the NV model can organize performance information from a complex layered language system so that we can identify important performance problems and greatly improve runtime performance.

4.1. Simple Example

In our first example, we use ParaMap to analyze the program shown in Figure 1. This simple example is useful because it exhibits a significant performance problem that is common in programs written by novice CM Fortran programmers. The first row in Figure 3 shows the execution time of the initial version of the program. When compared with the runtime of a serial version of the program (third row of Figure 3) measured on a single processor Sun Sparc 10/30, the initial program appears to be very slow.

Version	Execution Time
Initial Parallel (uninstrumented)	4 min 54.5 sec
Initial Parallel (instrumented)	9 min 12.0 sec
Serial (uninstrumented)	20.3 sec
Final Parallel (uninstrumented)	4.8 sec

Figure 3.

Execution times for small example

To analyze the example program, we used ParaMap to determine the noun costs at the CMF level. The tool told us that array `A` was responsible for a majority of the total CPU time. Figure 4 shows a cost breakdown for array assignments involving array `A`. A cost breakdown maps the noun and verb of a given sentence (in this case, the sentence `MAIN/A:Assignment`, i.e. assignment operations involving array `MAIN/A`) to lower levels and summarizes the lower level costs by the type of mapping and the purpose of the lower level activity. In this case, assignments involving `A` map to small amounts of explicit and implicit computation, and a large amount of implicit communication. The explicit computation costs include the time spent computing values for assignments involving `A`. The implicit computation costs include time spent in the runtime system managing memory for `A`.

```
MAIN/A:Assignment -- Cost Breakdown

COMPONENT                                      COUNT               TIME

Explicit Computation(Compute)                     11          1.297 sec
Implicit Communication(Broadcast)            5000000  45 min  0.412 sec
Implicit Computation(MemoryManagement)             1          0.001 sec
                                                      -----------------
                                         Total Time:  45 min  1.712 sec
```

Figure 4.

Costs of assignments involving A. Costs are summed over all processor nodes that map to the array.

To investigate the implicit communications, we created a context for the sentence MAIN/A:Assignment and moved to the NODE level. At the node level, we found that five million node broadcasts mapped to the sentence MAIN/A:Assignment. Since we knew that exactly five million elements of A were used to compute B, we investigated whether the broadcasts were caused by transfers of A to B. We asked the tool to refine the context to the upper left quadrant of A because assignments to B use only elements from the upper left quadrant of A (line 9 of Figure 1). However, the tool told us that we could only shrink our subregion to the left half of A (subregion [0:1000, 0:511]) because the runtime system had not distributed the row axis of A across processors. Nevertheless, we found that all five million broadcasts mapped to the left half of A. Since all of the broadcasts mapped to the left half of A and since five million elements of A were used to compute B, we concluded that all of the broadcasts of A were used to send elements to the control processor for computation of B. Figure 5 shows a time histogram of broadcasts that mapped to the selected subregion of A. The display shows that broadcasting of subregion [0:1000, 0:511] of array A occurred during nearly the entire execution of the program.

We improved the program by inserting an array alignment pragma. The pragma (CMF$ ALIGN B(I,J) WITH A(I,J)) instructs the compiler to map B alongside A on the processor nodes. The result was a 98% decrease in the program's runtime as shown in the fourth row of Figure 3. The alignment pragma allows the computations on Line 9 to occur in parallel without any transfers of A to the control processor.

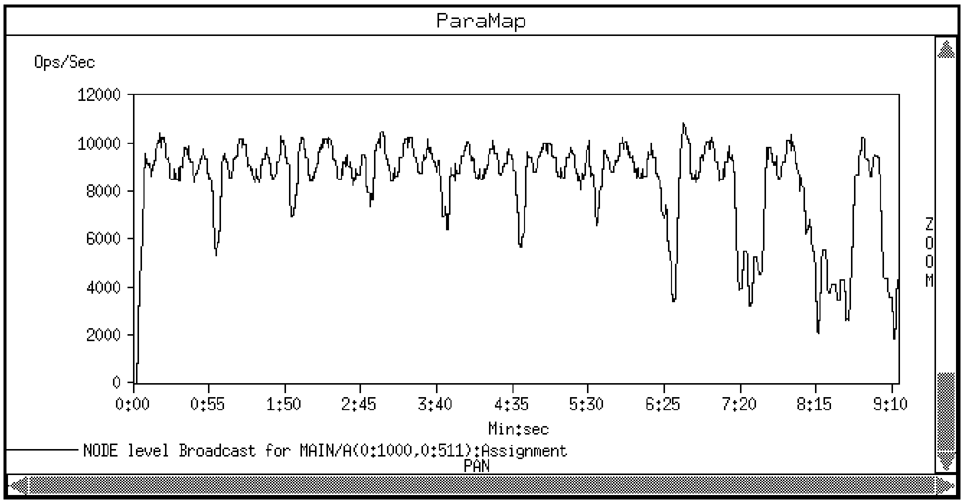

Figure 5.

Time histogram of Broadcasts that map to an array sub-region. Only events from the nodes that map to the selected sub-region are shown.

4.2. Dual Reciprocity Boundary Element Method

Our second application is a parallel implementation of the Dual Reciprocity Boundary Element Method (DRBEM) [28], a non-linear solution technique used for heat transfer, and vibration analysis applications. The DRBEM allows non-linearities to be solved as boundary integral problems. Like other boundary element methods, it relies on Green's theorem to reduce a two-dimensional area problem into a one-dimensional line integral. The line-integral is then solved by discretizing and solving sets of linear equations.

The full DRBEM application is comprised of 2200 lines of code spread over 18 source files. The application reads initial conditions from a file, sets up a system of linear equations, solves the equations for a series of time steps, and finally writes the results to file. We ran the program on a problem involving 1000 boundary elements, 250 interior elements, and 200 time steps. The runtime of the initial parallel version is shown in the first row of Figure 6.

Version	Time	Percent Improvement
Initial	66 min 16 sec	
Initial (instrumented)	71 min 31 sec	
CMSSL Gaussian Elimination (uninstrumented)	48 min 17 sec	27.1 %
Eliminate Unused Arrays (uninstrumented)	65 min 12 sec	1.6 %
CMSSL Solver (uninstrumented)	43 min 35 sec	34.2 %
CMSSL Inverse (uninstrumented)	56 min 52 sec	14.2 %
All Changes Together (uninstrumented)	**17 min 37 sec**	**73.4 %**

Figure 6.

Runtimes for Parallel DRBEM. Rows 3-6 show the results of implementing each improvement separately, while the last row shows the result of applying all improvements at once.

We began our analysis of the DRBEM application by examining CMF level profiles of the verbs Statement_Execution and Assignment (Figures 7 and 8). The verb Statement_Execution maps to the execution of PN code blocks generated by the compiler to implement CM Fortran statements. The verb Assignment maps to both execution of PN code blocks and to implicit runtime system activities such as allocation, shifting, and rotating arrays. Each profile lists all nouns that participated in sentences involving the given verb. The nouns are sorted by the cumulative costs measured for the sentences. The cumulative costs for statement executions include only explicit node-level computations that map to each execution; mapping of implicit costs to statement executions is not yet supported. The cumulative costs for array assignments include the costs of both implicit and explicit lower level activities that map to each assignment.

CMF Profile -- *:Statement_Execution

NOUN	COUNT	TIME
decomp.fcm<92,96>	1249	1 hr 0 min 12.063 sec
inverse.fcm<100,102>	1250	36 min 33.571 sec
decomp.fcm<81,83,82>	1249	29 min 56.119 sec
decomp.fcm<61>	1249	14 min 8.491 sec
inverse.fcm<92,83,93>	1250	12 min 16.351 sec
inverse.fcm<144,145>	1250	10 min 27.330 sec
inverse.fcm<149>	1250	6 min 26.298 sec
inverse.fcm<141,147>	1250	6 min 21.923 sec
hg.fcm<105>	1000	3 min 14.325 sec
pack.fcm<88,88>	200	1 min 42.361 sec

Figure 7.

Statement Execution Profile for DRBEM. Line numbers listed within angle brackets represent groups of statements that were merged during compiler optimization

The execution and assignment profiles illustrate why it is necessary to examine different types of nouns to best localize performance problems. The statement execution profile (see Figure 7) shows that 6 different statements in the file decomp.fcm are responsible for most of the explicit node-level computation in the application. By examining the code in decomp.fcm, we found that the top array shown in the assignment profile (`MAIN/H` in Figure 8) was accessed in every statement listed in the execution profile. Therefore, we decided to concentrate on how array `MAIN/H` was accessed rather than concentrate on the code of any particular line.

The code in decomp.fcm that processed `MAIN/H` was an implementation of Gaussian elimination factorization. A cost breakdown of `MAIN/H` (not shown, but similar to Figure 4) showed us that two-thirds of the node-level time that mapped to the array was spent in point-to-point communication routines. Therefore, we concluded that the Gaussian elimination implementation had caused the point-to-point communications as well as the node-level computations. We decided that since Gaussian elimination is a well understood method that has been implemented in many linear algebra libraries, we could simply replace the entire subroutine with a call to a library routine. Therefore, we replaced the routine with a call to the

CMF Profile -- *:Assignment

NOUN	COUNT	TIME
MAIN/H	9996	2 hr 57 min 11.215 sec
decomp/TMP	2499	1 hr 44 min 19.918 sec
MAIN/F	7501	1 hr 7 min 2.516 sec
inverse/AI	11280	53 min 31.442 sec
MAIN/G	3702	28 min 9.189 sec
pack/CH	5804	15 min 44.178 sec
back2/ROW	10362	14 min 56.545 sec
elim2/TMP	6253	1 min 50.018 sec
decomp/PVT	2898	49.640 sec
MAIN/DFI	400	48.631 sec

Figure 8.

Array Assignment Profile for DRBEM. This table sorts CM Fortran Arrays by their cumulative explicit and implicit costs.

vendor provided CM Scientific Software Library (CMSSL) Gaussian elimination routine. The improvement reduced the runtime of the application by 27.1% as shown in the third row of Figure 6.

Figure 7 also shows large explicit computation costs for 10 lines in the file inverse.fcm. When we examined the listed lines, we found that array `MAIN/F` (listed third in the assignment profile) was accessed on 8 of the 10 lines. Again, it seemed more important to study the processing of `MAIN/F` rather than examine any particular line. We displayed a Cost Breakdown table (see Figure 9) and found that more than 50% of the costs of assignments involving `MAIN/F` were due to point-to-point communication of elements of `MAIN/F` and spreading subsections of `MAIN/F` across processors. We examined the use of `MAIN/F` on the 8 lines and identified the operations that cause point-to-point communications and spreading of array elements. The code lines were part of a routine that computed the inverse of matrix `MAIN/F`. The routine took advantage of symmetries in the matrix, but we found that by employing a CMSSL routine for general matrix inversion, we improved the execution time of the entire application by 14.2% as shown in the sixth row of Figure 6.

Array assignment profiles are also useful for locating unused arrays. Unused arrays are arrays that are specified by the programmer, allocated by the runtime system, but never used by the program. In ParaMap they appear as CMF-level nouns, but they are not recorded in any assignment sentences. Therefore, an array assignment profile lists unused arrays at the bottom showing no cumulative cost. For the

MAIN/F:Assignment -- Cost Breakdown

COMPONENT	COUNT	TIME
Explicit Computation(Compute)	5004	28 min 51.634 sec
Implicit Communication(Broadcast)	2500	2.627 sec
Implicit Communication(Point-to-Point)	1250	23 min 46.354 sec
Implicit Communication(Spread)	3750	14 min 21.902 sec
	Total Time:	1 hr 7 min 2.516 sec

Figure 9.

Cost Breakdown for assignment operations involving `MAIN/F`*. The table shows that most low-level operations involving the array were implicit and caused communication*

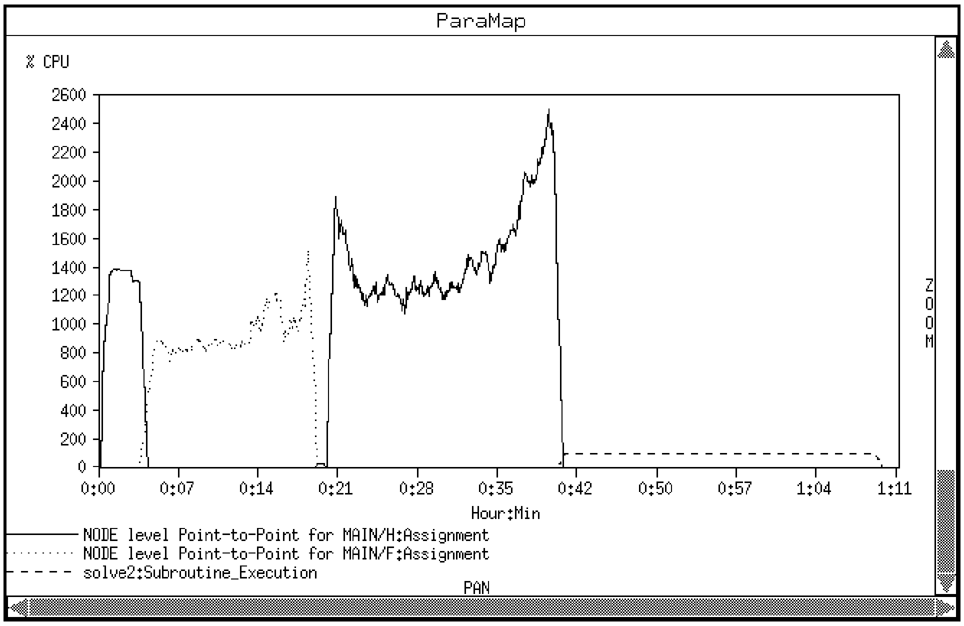

Figure 10.

Time histogram of CPU Time for the entire initial parallel DRBEM application. The display shows three primary runtime contributors: Point-to-Point communications that map to arrays `MAIN/H` *and* `MAIN/F`*, and sequential execution in the routine* `solve2`*.*

DRBEM application we found 16 unused arrays and improved the program by eliminating them from the code. The runtime savings (shown in the fourth row of Figure 6) were minimal, but the memory savings amounted to 32 Megabytes (5% of the total).

A final major performance problem involved sequential subroutine execution time. Subroutine execution time is measured as process time on the control processor and is measured for each subroutine in the application. We displayed a profile of subroutine execution and found that 27 minutes were spent in the subroutine `solve2`. The subroutine implemented the solution phase of the application and had apparently never been parallelized. We decided that the vendor provided CMSSL parallel linear system solver could be used to improve runtime performance, and inserted the CMSSL solver in place of `solve2`. This change reduced the overall runtime of the program by 34.2% as shown in the fourth row of Figure 6.

Figure 10 illustrates key aspects of the behavior of the DRBEM application over time. The figure presents the process time spent performing point-to-point communications of elements of arrays `MAIN/H` and `MAIN/F`,and serial time spent in subroutine `solve2`. The figure shows that the implicit communications and the serial computations degraded the performance of the application by using a significant amount of the parallel processing resources of the machine. In particular, during the 27 minutes in which `solve2` executed on the control processor, all of the processor nodes remained idle. The final line of Figure 6 presents the total savings achieved by incorporating all of the improvements listed above.

5. Related Work

Many performance tools measure least common denominator events such as procedure calls [1, 7, 27], basic blocks [18, 23], runtime library events [27], or synchronizations [19]. Such events are independent of programming model, and therefore the corresponding tools can measure programs written in many programming languages. However, information about low-level events may not be relevant to or understood by a programmer who has written a program in a high-level parallel programming language.

Several performance tools have been designed for specific languages or programming environments [13, 14, 20, 21, 25, 32]. Language specific tools use knowledge of the programming model to reduce measurement costs and give detailed performance feedback about specific language constructs or data types. Language specific tools generally provide only one view of program performance – the language level view. Information about implicit lower level activity that may be necessary to fully understand the performance of a complex application is either hidden or omitted. The user should be allowed to separate the language level from lower levels but should also be allowed to see lower level views of performance when necessary.

A hybrid approach is to collect low-level events and map them to higher levels of abstraction for analysis [2, 15, 31, 33]. The user describes how low-level events are to be combined and identified as high-level events. This approach provides maximum flexibility in handling program performance information, but the general problem addressed by existing tools – mapping all event streams to all types of performance information – is a difficult problem.

Performance measurement of programs written in high-level languages bears some resemblance to the problem of debugging optimized code [3, 6, 8, 36]. In the latter problem, a symbolic debugger must present a view of an optimized program that is consistent with the original source code and must hide the effects of optimizations that have reordered statements, eliminated variables, or otherwise altered the steps of a computation. Performance measurement of parallel programs written in high-level languages is fundamentally simpler than the debugging of optimized code because performance measurement tools need not reconstruct the instantaneous state of a computation. A symbolic debugger must be able to stop the execution of a program at any instruction, identify the corresponding location in the source code, and provide access to variables in the original program. A performance measurement tool, on the other hand, is generally concerned with the cumulative activity of program elements (code constructs and data objects). Performance measurement tools must identify the program elements that are active at a given point but rarely need access to the values of variables.

6. Conclusions

We claim that even a simple tool based on the NV model can have great advantages when studying the performance of programs written in high-level parallel languages. With ParaMap, we answered performance questions that could not easily be answered with other tools. In particular, by displaying performance data for CM Fortran parallel arrays we quickly located primary performance problems, and by examining runtime system and processor activities (while maintaining references to high-level arrays and subsections of arrays) we evaluated the low-level results of array operations. Specific examples of this type of analysis include the localization of broadcasts to a particular portion of an array in Section 4.1, and attribution of point-to-point messages to a few important arrays in Section 4.2. ParaMap finds problems that could be found with simple profilers (such as sequential processing bottlenecks in subroutines), but by employing NV mapping functions, providing access to array information, and providing time plots of performance data, ParaMap allows programmers to go beyond traditional performance measurement analysis techniques and study more difficult problems.

We have found that providing performance information for the fundamental constructs of a language (e.g. parallel arrays in CM Fortran) provides a good first step in understanding any application's performance. However, when we have located an array with high performance costs, we have used ParaMap to drop down to the runtime system or node level to evaluate the object's impact on the system and to determine whether the cumulative costs are due to the programmer's explicit requests for computation or whether the programmer has implicitly and perhaps unknowingly caused extra runtime activity. This type of analysis represents a departure from high-level language tools that provide information exclusively at the language level.

We intend to address important performance questions that ParaMap does not yet answer. For example, we have often wanted to cross reference performance information for arrays by performance information for statements, because arrays are assigned using CM Fortran statements. With our present instrumentation system, such a cross-product of information would be very expensive to gather. Therefore, we will employ a dynamic instrumentation system [12] that will allow us to selectively insert measurement probes to measure arbitrary noun and verb combinations without measuring all combinations at all times.

High-level parallel language systems are often intricate and sorting through all possible performance information for all nouns and verbs at all levels of abstraction can be an arduous task, even for simple applications. Therefore, we will employ automated search techniques [11] to assist in the evaluation of real applications. Automated searching for performance bottlenecks in high-level parallel language applications will require NV style mapping for locating and explaining performance bottlenecks to programmers.

Acknowledgements

We thank Jens Christoph Maetzig for the original serial DRBEM code, and Bruce Davis and Brad Richards for the initial CM Fortran version.

1. Z. Aral and I. Gertner, "Non-Intrusive and Interactive profiling in Parasight", *Proc. ACM/SIGPLAN PPEALS*, 1988, pp. 21-30.
2. P. Bates and J. Wileden, "An Approach to High-Level Debugging of Distributed Systems", *ACM SIGSOFT/SIGPLAN Software Engineering Symposium on High Level Debugging*, March 1983, pp. 107-111.
3. G. Brooks, G. J. Hansen and S. Simmons, "A New Approach to Debugging Optimized Code", *ACM SIGPLAN Proc. on Programming Language Design and Implementation*, 1992, pp. 1-11.
4. N. Carriero and D. Gelernter, "Applications Experience with Linda", *Proceedings of the 1986 International Conference on Parallel Processing.*, July 1988, pp. 173-187.
5. R. Chandra, A. Gupta and J. L. Hennessy, COOL: A Language for Parallel Programming, in *Languages and Compilers for Parallel Computing*, MIT Press, Cambridge MA, 1990, 126-148.
6. D. S. Coutant, S. Meloy and M. Ruscetta, DOC: A Practical Approach to Source-Level Debugging of Globally Optimized Code, 1988.
7. S. L. Graham, P. B. Kessler and M. K. McKusick, "Gprof: A call graph execution profiler", *Proceedings of the ACM SIGPLAN Symposium on Compiler Construction*, June 1982.
8. J. L. Hennessy, "Symbolic Debugging of Optimized Code", *ACM TOPLAS 4*, 3 (July 1982), pp. 323-344.
9. *High Performance Fortran Language Specification – Version 1.0*, High Performance Fortran Forum, January 27, 1993.
10. J. K. Hollingsworth, R. B. Irvin and B. P. Miller, "The Integration of Application and System Based Metrics in A Parallel Program Performance Tool", *1991 ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, April 1991, pp. 189-200.
11. J. K. Hollingsworth and B. P. Miller, "Dynamic Control of Performance Monitoring on Large Scale Parallel Systems", *7th ACM International Conference on Supercomputing*, Tokyo, July 1993, pp. 185-194.

12. J. K. Hollingsworth, B. P. Miller and J. Cargille, Dynamic Program Instrumentation for Scalable Performance Tools, (submitted to SHPCC94).

13. L. V. Kale and A. B. Sinha, ‘‘ Projections: A Scalable Performance Tool ’’, University of Illinois Dept. of Computer Science 92-3.

14. C. Kesselman, Integrating Performance Analysis with Performance Measurement in Parallel Programs, UCLA-CS-TR-91-03, UCLA, Los Angelos, CA, 1991.

15. C. Kilpatrick and K. Schwan, "ChaosMON - Application-Specific Monitoring and Display of Performance Information for Parallel and Distributed Systems", *DEBUG'91*, , pp. 48-59.

16. M. S. Lam and M. C. Rinard, "Coarse-Grain Parallel Programming in Jade", *Proceedings of the 1991 ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, April 1991, pp. 94-105.

17. J. R. Larus, B. Richards and G. Viswanathan, ‘‘ C**: A Large-Grain, Object-Oriented, Data-Parallel Programming Language ’’, UW-Madison Computer Sciences Technical Report #1126, November 1992.

18. J. R. Larus and T. Ball, Rewriting Executable Files to Measure Program Behavior, TR 1083, Computer Sciences Department, University of Wisconsin-Madison, March 1992.

19. T. J. LeBlanc, J. M. Mellor-Crummey and R. J. Fowler, *Analyzing Parallel Program Executions Using Multiple Views*, Journal of Parallel and Distributed Computing, 1990.

20. L. B. Linden, Parallel Program Visualization Using ParVis, in *Performance Instrumentation and Visualization*, ACM Press, New York, 1990, 157-187.

21. K. M. Lord and M. L. Simmons, Improving Performance with CRI UNICOS Tools: A Case Study, 1991.

22. E. Lusk and R. Butler, ‘‘ User's Guide to the p4 Parallel Programming System ’’, Tech Report ANL-92/17, Argonne National Laboratory, Argonne, IL, October 1992.

23. *UMIPS-V Reference Manual* , MIPS Computer Systems, Sunnyvale, CA, 1990.

24. A. Malony, B. Mohr, P. Beckman, D. Gannon, S. Yang, F. Bodin and S. Kesavan, "Implementing a Parallel C++ Runtime System for Scalable Parallel Systems", *Proceedings of Supercomputing '93*, Portland, OR, November, 1993, pp. 588-597.

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

26. J. McGraw, S. Skedzielewski, S. Allan, R. Oldehoeft, J. Glauert, C. Kirkham, B. Noyce and R. Thomas, SISAL: Streams and Iteration in a Single Assignment Language, Languge Reference Manual -- Version 1.2, Lawrence Livermore National Laboratory, University of California, Davis CA, March 1, 1985.

27. B. P. Miller, M. Clark, J. Hollingsworth, S. Kierstead, S. Lim and T. Torzewski, "IPS-2: The Second Generation of a Parallel Program Measurement System", *IEEE Transactions on Parallel and Distributed Systems 1*, 2 (April 1990), pp. 206-217.

28. D. Nardini and C. A. Brebbia, "A New Approach for Free Vibration Analysis Using Boundary Elements", *Boundary Element Methods in Engineering*, 1982.

29. *The PVM User's Guide – Version 3.0*, Oak Ridge National Laboratory, Oak Ridge, TN, February 1, 1993.

30. J. Ousterhout, " "Tcl: An Embedable Command Language"", *Proceedings of the Winter 1990 USNIX Conference*, Washington, DC, January 22-26, 1990.

31. S. Perl, "Performance Assertions", *Proceedings of the 14th Symposium on Operating System Principles*, December 1993.

32. S. Sistare, D. Allen, R. Bowker, K. Jourdenais, J. Simons and R. Title, Data Visualization and Performance Analysis in the Prism Programming Environment, in *Programming Environments for Parallel Computing*, North-Holland, 1992, 37-52.

33. R. Snodgrass, A Relational Approach to Monitoring Complex Systems, Vol. 6, May 1988.

34. *CM Fortran Reference Manual*, Thinking Machines Corporation, Cambridge MA, January 1991.

35. *CMSSL for CM Fortran: CM-5 Edition, Vols I and II*, Thinking Machines Corporation, Cambridge MA, January 1993.

36. P. T. Zellweger, "An Interactive High-Level Debugger for Control-Flow Optimized Programs", *ACM SIGPLAN Notices 18*, 8 (March 1983), pp. 159-172.