Appears in the Proceedings of The Fifth Symposium on the Frontiers of Massively Paralllel Computation, Feb 1995, McLean, VA

Performance Debugging based on Scalability Analysis

Takashi Suzuoka Jaspal Subhlok Thomas Gross

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

{tsuzuoka,jass,trg}@cs.cmu.edu

Abstract

This paper presents scalability *as a basis for profiling and performance debugging of parallel programs, as only the purely scalable code runs efficiently in parallel. The approach is based on separating scalable and various kinds of non-scalable parts of a program, identifying the reasons for non-scalability, and focusing the programmer's attention on why and where non-scalable execution is occurring. We specifically address parallel programs that are generated by a parallelizing compiler, and use compiler information to divide the execution times into logical categories that are meaningful to the programmer. We have designed and implemented a profiler that is integrated with a compiler for a variant of High Performance Fortran. The profiler classifies the execution cycles into eleven categories that are easily understood by the user in terms of the source program. The paper includes two examples that demonstrate how the data reported by the profiler is used to identify and resolve performance bugs in parallel programs.*

Keywords: *parallel processing, performance evaluation, performance debugging, performance tuning, scalability analysis, parallelizing compilers*

1 Introduction

The primary reason for using parallel computers is to obtain fast execution of applications. While a high performance computer, a good compiler, and efficient libraries are necessary to achieve fast execution, they alone do not guarantee satisfactory performance on a given program. The current state of the art still involves the programmer or the application specialist. The development of a parallel application is an iterative process in which the programmer repeatedly analyzes performance data to determine execution bottlenecks and makes changes to the program and/or the algorithm. Thus, a profiler (with a visualization tool) plays a critical role in the development of parallel programs.

Use of parallelizing compilers is becoming the primary mode of programming parallel machines, and this research focuses on performance debugging of compiler generated parallel programs. When writing a program in a language like High Performance Fortran (HPF) [5], the programmer does not write an explicit parallel program, but influences the performance of the program by the use of parallel constructs, compiler directives, and compiler flags. The compiler generates code for the processors of the parallel system from the source program. So all feedback from the profiler must be in terms of the source program, and at a level that can be understood by the programmer. Information that is very broad in scope, or at a low level, is of limited use for performance debugging. For example, a breakdown of the total time spent into computation and communication time alone may not be enough to provide guidance to the user on how to reduce communication cost.

To allow a user to speed up the execution of a program, one must focus on the parts of the program that can be potentially improved, rather than parts that simply take the longest time. Our profiler divides the execution time for the program into *scalable* and *non-scalable* parts. The basic idea is that scalable code makes efficient progress in the execution of a program. Non-scalable code captures those parts of the program that do not efficiently contribute to the progress of the program. When a processor is idle, communicating, performing replicated computations, or doing compiler bookkeeping, it is executing non-scalable code. It is important to note that non-scalable code includes parts of the user program that do not execute efficiently in parallel, in addition to the overheads of parallel execution. Scalability is not only a measure of how well the computation can use more processors, but also a measure of how well the available processors are being used. The objective of performance debugging then is to maximize the time a program spends executing scalable code, and minimize the time it spends executing non-scalable code.

It is not possible to simply probe the state of a processor to determine if it is executing scalable code or not. For instance, a processor may be executing a replicated computation and thus appear to be doing useful work while it is actually not making a contribution to the progress of the program. Similarly, it is not possible to distinguish whether the code that is being executed is user code, or compiler overhead. The key to the solution of this problem is to get the parallelizing compiler involved in the process of profiling.

A parallelizing compiler is a source of critical program information that can be used for profiling, and this observation is central to our design. We use compiler knowledge to instrument the program such that all the interesting execution modes of the program can be distinguished with minimal overhead. It is a straightforward use of compiler information to distinguish between distributed and replicated computations, or between user code and compiler overheads. The result is that we obtain all the relevant information for performance debugging with minimal overhead and without any external

This research was sponsored in part by Toshiba Corporation, and in part by the Advanced Research Projects Agency/CSTO monitored by SPAWAR under contract N00039-93-C-0152.

runtime support (except for access to a timer).

The compiler is also used to guide the visualization of profiler data. In particular, for generating meaningful visualization for heterogeneous execution (e.g. when the nodes of the system execute different programs) the compiler's knowledge is needed to display the information meaningfully in terms of the source code.

In this paper, we present the design and implementation of a program profiling tool based on the notion of scalability, and illustrate its use in performance debugging. The profiler is integrated with the Fx parallelizing compiler for a variant of High Performance Fortran that supports task and data parallelism [10]. The targets for the compiler include networks of workstations using the PVM library, the iWarp system, the Cray T3D, and the Intel Paragon.

The paper is organized as follows. Section 2 relates our work to the evolution of profilers for parallel machines. Section 3 describes the use of scalability as a criterion for profiling, and describes the classes of non-scalable code. Section 4 explains how the compiler knowledge is used by the profiler. Section 5 describes the visualization of profiler data, and Section 6 describes the implementation. Section 7 describes how the profiler is used for performance debugging, and Section 8 contains the conclusions.

2 Related work

Most profilers measure the time a processor spends in different states, and in different parts of a program. For example, a profiler may provide the amount of time the system is computing, communicating, waiting on synchronization, servicing cache misses, etc. Examples include IPS-2 [7], ParaGraph [4], and several others [1, 9, 6, 8]. These profilers generally work by probing the processor state, and they can be independent of the compiler. However, with a parallel language like High Performance Fortran, the compiler generates the final code and the mapping between the machine instructions and the user's program can be difficult to understand.

Several profilers address this problem by using compiler knowledge to present the processor state information in terms of the source program. The examples of such systems include Thinking Machines Prism, Parasoft Express, Mtool [3], and several others, including [2, 12]. These systems present the *physical* state of the processor, not the *logical* state, which is the programmer's view of what is happening. The relationship between physical and logical processor state may be obvious in some cases, but can be difficult to establish in other situations. For example, various kinds of computation and communication overheads are incurred in starting and executing a compiler parallelized loop, and simply reporting the total amount of computation and communication times for the loop hides several important distinctions.

Some profilers present an animated view of the program execution, while others show a summary of execution in terms of the source program. While an animation provides a detailed view of execution, we believe that presenting the important aspects of an execution in terms of the source program is quick and sufficient to identify most performance bugs. Also in a compiler parallelized programs, it becomes more difficult to relate an animation to the source program.

Profiling systems based on simulating parallel execution are particularly valuable when access to a parallel machine is limited or expensive. However, in our experience, many performance problems do not become apparent until the program is executed on the actual target, and hence simulation alone is generally not sufficient.

3 Scalability as a criterion

Performance debugging means improving the parts of a program that are not executing efficiently. But to achieve this, one must first find the inefficient parts. A common heuristic is to treat the most time consuming parts of the program as potential sites of performance bugs. While this heuristic has obvious merit, and is the method of choice for sequential programs, there is potential for a more sophisticated way of finding locations of possible performance bugs in parallel programs[1].

Parallel computing works best when all processors are busy and directly contributing to the progress of the user's program, rather than performing redundant or overhead computations, or being in a communication or waiting state. This part of a processor's work is *scalable*, and the performance of the corresponding parts of the program is expected to increase linearly with the number of processors. However, few parallel programs are completely scalable. The *non-scalable* parts of the program are responsible for inefficiency in execution, and should be targeted for improvement. The approach is similar to vectorization where the programmer focuses on parts of the program that are labeled as "not vectorizable" by the compiler.

The non-scalable execution time can be broadly classified into the following categories:

Non-scalable user time: Any program computation that is replicated on multiple nodes uses only one node effectively and hence is not scalable. Similarly, a loop computation with a loop carried data dependence may use only one node at a time and is not scalable. A special category of user code is I/O, which may or may not be scalable depending on the I/O pattern and the I/O devices. However, irrespective of the scalability behavior, it is important that the user be aware of the time spent on this activity.

Synchronization time: This refers to the time that a node is idle due to synchronization requirements. A high synchronization time is often caused by load imbalance.

Parallelization overhead: Parallelization overhead stems from a variety of tasks that have to be done to enable and support parallel execution. These tasks include setup times for communication resources, message assembly times, compiler generated bookkeeping and data management (e.g. changing distributions at procedure boundaries when needed), and several others.

Communication time: The actual behavior of communication steps depends on the communication pattern, but communication is typically not scalable. For example, in an all-to-all communication pattern generated by a matrix transpose, the volume of data sent and received by a processor decreases linearly as the number of processors increases, but each processor has to send a larger number of messages, and the corresponding overhead increases. The net result is that communication time typically does not decrease linearly with processors, and

[1] This paper deals with the topics unique to parallel systems. Clearly the code for each node is also subject to performance debugging. However, this problem can be addressed using conventional techniques, and in our experience, potential improvements are dwarfed by the speedups that can be realized by more effective use of the parallel system.

may actually increase. Further classification of communication times is discussed in Section 4 and Section 7.

Figure 1 summarizes the breakdown of execution time. Some of the above categories can be further divided depending on the objective of profiling. A fine grained profiling tool may want to include categories like cache misses and page fault overheads. In our implementation, we provide a finer division of communication overhead, which is described later. The point is that the non-scalable code can be classified into meaningful categories for program profiling.

scalable code	user code (parallel loops)
non-scalable code	user code (sequential, I/O)
	synchronization
	parallelization overhead
	communication

Figure 1: Processor states in parallel execution

It is also important that the profiler provides a precise measurement of these categories, and not just a summary judgement. A simple decision (such as poor scalability, load imbalance, or poor mapping of distributed arrays) can be incorrect and misleading since there are many possible causes of poor performance, and mapping between symptoms and causes of poor performance is many to many and complex. The final analysis is best left to the user, once the relationship between symptoms and causes is understood.

4 Compiler Information

It is important to utilize parallelizing compiler information to classify the time spent in different categories because only the compiler knows how the source code is translated to the executable code. It is possible to measure the total time spent on computation, communication and synchronization only using runtime instrumentation, but categorization into finer categories requires the involvement of the compiler. We outline the ways in which the compiler information is used for driving the profiler:

Classifying computation: The computation time is divided into three categories; scalable user code, non-scalable user code and parallelization overhead. Only the compiler is aware of this classification since it generates different kinds of code. For instance, a user cannot find the compiler overhead time by hand instrumentation, or by using a runtime tool.

Classifying communication: A part of the communication inside a data parallel module is directly related to executable statements, but the rest is not. For example, when a subroutine is called with an array argument, communication is generated if the distribution of the formal argument is different from the actual argument. Also, the compiler may generate communication before the execution of a parallel loop to ensure that the loop execution itself is free of communication. Compiler knowledge is necessary for identifying these types of *hidden* communication. Distinction between communication inside a module and various kinds of communication steps between modules (discussed in later sections) is also made with the help of the compiler.

Task parallel execution: Our compiler supports combined task and data parallel execution where different groups of processors are dedicated to executing different parts of the program. The compiler knows the mapping of tasks to processors, and this is necessary to relate the runtime measurements to the source code.

5 Performance visualization

We address the main issues facing the designer of a visualization system, and describe the decisions made for our profiler. The information is normally presented in graphical form, as shown, for example, in Figure 3, but can also be obtained in tabular form.

5.1 Granularity

We have chosen to display the profiler information in terms of the source code rather than a graphical program animation. Within the source code, the information can be shown for each statement, or for a coarser program unit like a basic block or a subroutine. Several profilers show the results for each statement [2]. When using a parallelizing compiler, we believe that most of the interesting information like communication and synchronization overheads relates to computations that can span a sequence of basic blocks, or across a whole subprogram unit. We have chosen to display information at the subroutine (or function) level by default, but allow the user to select smaller regions as units for profiling information.

5.2 Grouped view

The profiler must provide an abstract view of the whole program's execution. For a small number of processors, one can simply display the activities on the individual processors, but this approach is not feasible for massively parallel machines. However, for such machines, typically all the processors are performing similar computations, and the average of information over all processors is most relevant. Therefore, our profiler displays summaries over all processors by default. The information can also be displayed for any individual processor, or averaged over any subset of processors, under user control.

When the programmer uses task parallelism in addition to data parallelism, the processors are divided into groups corresponding to tasks. All processors inside a group perform similar computations, but processors in different groups may perform completely different computations. The profiler identifies the processor groups and allows the programmer to visualize information over these groups.

When the execution times and other performance data vary widely among processors, it is useful to have an abstract idea of overall load imbalance without having to look at processors individually. To this end, the profiler computes a load balancing figure and displays it on demand. The load balance value for a computation is based on the ratio between the maximum time among all processors for a computation, and the average time for that computation. The magnitude of this value represents the degree of load imbalance, with zero representing a perfectly load balanced computation.

6 Implementation

The profiler consists of a data collector and a performance viewer. It is machine independent, and only needs access to a timer on the target machine. We describe the implementation of these two major components, and discuss the overhead of profiling.

6.1 Data collector

The data collector is integrated with the Fx compiler [10] that translates a source program written in Fx Fortran (similar to HPF), into a Fortran 77 program for each processor of the parallel system, with calls to a communication library. The Fx compiler has knowledge of where the profiling information should be collected and embeds that knowledge in its output (Fortran 77) file using naming conventions for compiler generated variables and functions, and auxiliary comments. For example, all variables used for compiler bookkeeping are given a special prefix.

The profiler has knowledge of the static compiler database, e.g. it knows the function of all the communication and synchronization routines, and how to associate their execution time with the program. It uses this knowledge to insert calls to the runtime profiler library in the program. The program is then compiled by the native single-cell compiler as usual. Profiling is completely transparent to the user.

We show the different classes of parallel code generated by the translation of a simple array assignment statement:

Parallel Fortran:

```
integer i
real a(65536)
distribute a(block)
a = 0
i = 10
```

Parallel code generated by the compiler:

Fortran 77(node program)	Categories
`integer i`	
`real a(65536/nodes)`	
`ifxfirst = ...`	parallelization overhead
`ifxlast = ...`	parallelization overhead
`do ifxindex=ifxfirst,ifxlast`	
`   a(ifxindex) = 0`	scalable user code
`enddo`	
`i = 10`	non-scalable user code

At runtime, calls to the profiling library routines read the system timer(s) and accumulate the relevant values in an internal table. The overhead of profiling is low; in addition to a (possibly inlined) procedure call, there is only the cost of reading the timer and adding it to an accumulator. Most high-performance systems provide a hardware timer register in some form, and timer access requires only a few clock cycles. The profiler collects 11 kinds of timer data and has 1 counter for each subroutine or region of interest, and therefore requires only 12×(number of subroutines or regions of interest) words of data memory. The collected data is written to a file at the termination of the program.

6.2 Performance viewer

The viewer has a window based interface that is implemented in C and Tcl/Tk. Figure 2 shows the profiler console, with a button for each processor (in octal), and other controls. The user can select any group of processors by directly clicking on them, or by using the menu on the row starting with *all*. The user can select any group of subroutines for viewing by naming one directly, or by naming a file that contains them, in the `routines/file` space. The user can choose to look at any or all of the 11 categories listed in the rows of squares starting with `scalable`, `comm` and `t Rwait` squares.

.

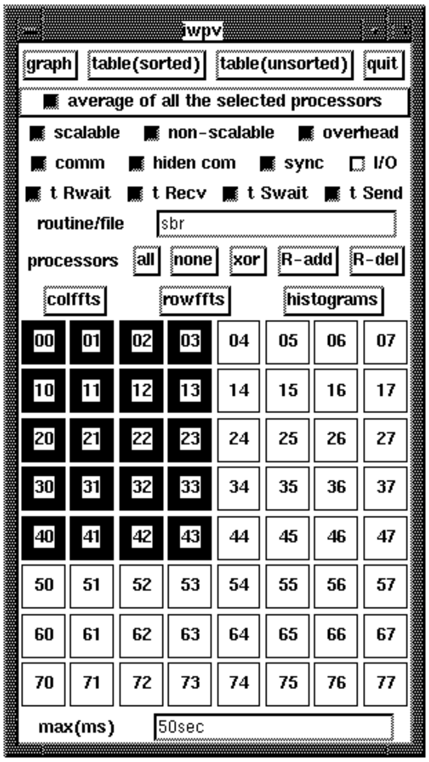

Figure 2: Profiler console

The display in Figure 2 is set to view all categories except I/O, for a set of subroutines listed in file *sbr*, on a scale up to 50 seconds, and for the set of processors that are highlighted. The corresponding display of performance is shown at the top in Figure 8, and described later in Section 7.

6.3 Perturbation by the profiler

The profiler has a very low overhead, and barely perturbs the execution of the user programs. For example, the programs discussed later in this paper run only 0.12%(Gaussian Elimination) and 0.29%(FFT-Histograms) slower with the profiler. We have used the profiler with a range of applications, and the perturbation is always less than 1%. As discussed above, there are two reasons for this low cost; each call to a profiling routine adds only a few instructions, and the placement of profiler calls is optimized with the help of the compiler.

7 Using the profiler to improve programs

We describe how the visualization of performance with the profiler is used to identify and correct several kinds of performance bugs, and illustrate the process with examples. In all of this section, **boldface** words are used to denote items as they appear in the profiler display. We first address performance debugging of data parallel modules executing in SPMD (Single Program Multiple Data) mode, and then programs that contain multiple modules that are mapped on different parts of a processor array.

7.1 Performance debugging a data parallel module

The profiler provides the following information about the execution of a program:

- The amount of time spent in scalable computations (**sc**), and the amount of time spent in non-scalable (or non-parallel) user computations (**!sc**).
- The amount of time spent in I/O (**I/O**).
- The amount of time spent in synchronization (**sync**).
- The amount of time spent in parallelization overhead (**oh**).
- The amount of time spent in explicit communication (**com**), and the amount of time spent in hidden communication (**hcm**).
- The longest total execution time among selected processors (on the top left corner of display).

In a typical performance debugging session, the programmer first looks at the whole program view, and then focuses on the performance of particular subroutines or parts of the program, on all or some processors. To help identify the performance bugs, the programmer is provided with a standard list of potential performance bugs along with lists of "symptoms" in terms of the profiler data that may indicate a particular bug. We omit the full mapping of bugs to symptoms for brevity, but will describe some example performance bugs in compiler generated parallel programs, and discuss how the profiler is used to identify and correct them.

Insufficient parallelization

The amount of time a processor spends on executing parts of the program that are not parallelized is presented as non-scalable computation time, so a high value for it implies potential unexploited parallelism. Often the problem can be resolved with additional compiler directives, or by using a parallel construct.

Poor mapping and alignment of arrays

Many data parallel languages, including Fx Fortran, allow the programmer to control the relative alignment of arrays, and the distribution of arrays onto processors. A poor data distribution can be a significant performance problem, and lead to excessive communication and/or load imbalance. A profiler display showing unexpectedly high communication or synchronization time is a hint that the mapping of data may not be optimal.

Example: Gaussian elimination

Our first example is an Fx parallel program to perform Gaussian elimination on a 1024×1024 matrix with a blocked distribution, running on a 64 processor iWarp array. The profiler display for average execution time for all processors is shown in Figure 3.

The display shows that the program spends around 65 seconds (or 85% of the total time) in `gauss` routine, and around 12 seconds (or 15% of the total time) in the `main` routine. It also spends around 20% of the time in communication (**com**), around 22% in synchronization (**sync**), around 16% in non-scalable user computation (**!sc**), and remaining 42% in scalable user computation. The display shows a maximum total execution time of 77.3 seconds.

Our first observation is that there is a significant synchronization overhead in `gauss` routine. On the display for individual nodes, we see that there is a wide variation in synchronization overhead among processors, suggesting a load imbalance. The second observation is that the `main` routine is taking a surprisingly long time relative to expectations, and all of it is non-scalable user time (**!sc**). We had not paid any attention to this routine since it has a relatively small amount of computation.

One possible reason for load imbalance is that a block data distribution is not the most appropriate distribution for this problem, so we changed the distribution to cyclic. The non-scalable time in `main` was traced to a sequential initialization loop, which was converted to a parallel loop.

The profiler display from the improved program is shown in Figure 4. The synchronization overhead in `gauss` is significantly lower, and the total time spent in `main` has become negligible. The maximum total execution time is reduced by about 35% to 50.3 seconds. Notice that part of the improvement was obtained by identifying an inefficient computation with the help of the compiler, and could not have been detected by simply examining the computation and communication related overheads.

7.2 Performance debugging multiple data parallel modules

An application can be composed with multiple data parallel modules, with each module (or task) assigned to a different set of processors. Support for this form of task and data parallelism is provided by the Fx compiler [10]. The communication in the program now contains communication between different modules, as well as communication inside a module. Further, one module can be waiting to send data to, or receive data from, another module, which introduces a new kind of synchronization overhead that does not exist for programs with a single data parallel module. The profiler provides the following additional information for multiple modules (this is also a form of hidden communication) :

- The time that a module is idle waiting to receive input from another module (**tsw**), or to send output to another module (**trw**).
- The time that a module is actually sending output to another module (**ts**), or receiving input from another module (**tr**).

The presence of multiple data parallel modules introduces the possibility of performance bugs due to a poor mapping of the modules onto processors. We describe some example problems in mapping that can lead to poor performance, and describe how the profiler is used to correct them.

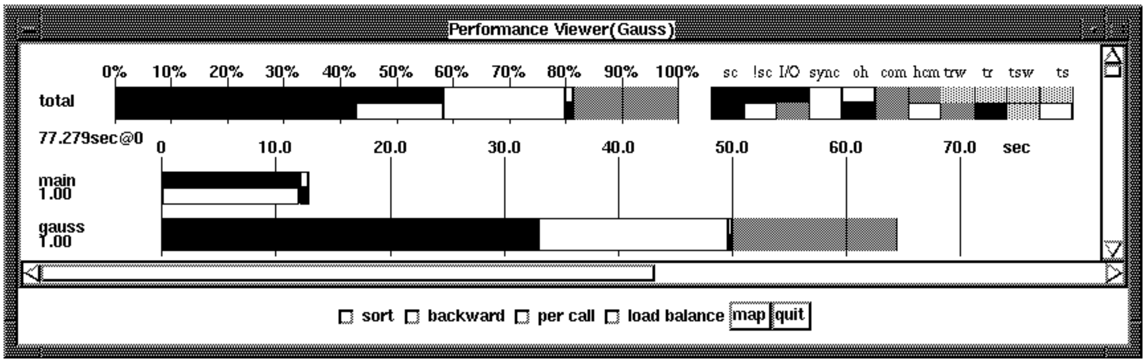

Figure 3: Profiler display for Gaussian elimination

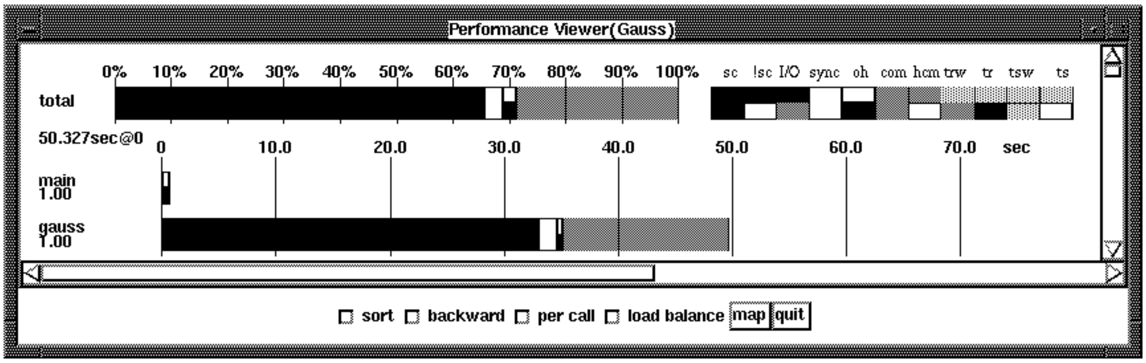

Figure 4: Profiler display for improved Gaussian elimination

Unbalanced allocation of processors to modules

The numbers of processors allocated to modules should be in proportion to the computation in the modules. Otherwise some modules will spend a lot of time waiting to send output, or to receive input, from other modules that are running slower. This is easily detected by high send-wait (**tsw**) and receive-wait (**trw**) times.

Need to replicate modules

We first introduce replication, a powerful optimization that can be used with task parallelism. Suppose `M` is a subroutine that constitutes a module, and is invoked a large number of times in a loop during execution. Also, suppose that each invocation of `M` takes 10 seconds to complete on 10 processors, and 12 seconds on 5 processors, since M is only partially scalable. If a total of 10 processors were available, it is more useful to divide them into two groups of 5 processors with each group performing half the iterations, rather than one group of 10 processors that performs all iterations, as shown in Figure 5. Thus, replication can improve the performance of a module.

Legality of replication is dependent on data dependence and memory constraints, and a discussion is beyond the scope of this paper. However, the profiler can help us identify when replication is potentially useful. When a module is spending a large amount of time executing non-scalable code (that is non-scalable computation, communication, synchronization, and overheads), it is difficult to speed up the module by adding more processors, and replication may lead to more efficient execution.

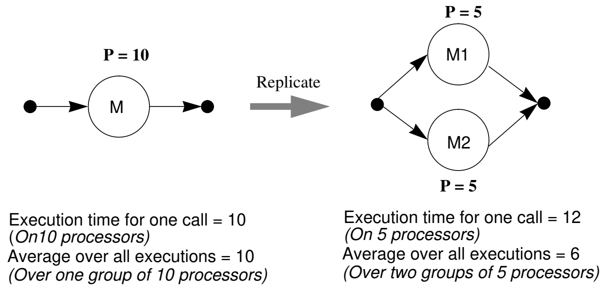

Figure 5: Replication of modules

Example: FFT-Histograms

This example program consists of a 512 point 2DFFT, implemented as a sequence of 1DFFTs with a transpose between them, followed by a statistical analysis routine that builds histograms. The three subroutines corresponding to the computation stages are `colffts`, `rowffts` and `histograms` (for column FFTs, row FFTs and Histograms, respectively). Figure 6 shows the structure of the computation and communication, and a subroutine call level code for the program.

The three subroutines correspond to modules *colffts-module, rowffts-module*, and *histograms-module*, that are mapped on different sets of processors. The computation is pipelined, that is, different modules are working on successive iterations (or successive data sets) at any given time. This is possible since there are

```
do i = 1,m
   call colffts(A)
   call rowffts(A)
   call histograms(A)
enddo
```

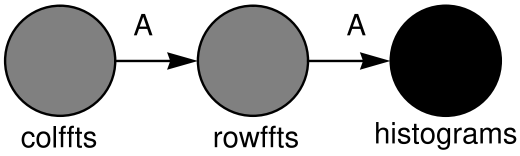

Figure 6: Structure of the FFT-histograms program

no loop carried dependences. We initially assign approximately the same number of processors to the three modules, 20 processors each to *colffts-module* and *rowffts-module*, and 24 to *histograms-module*. The mapping of the modules is shown in Figure 7.

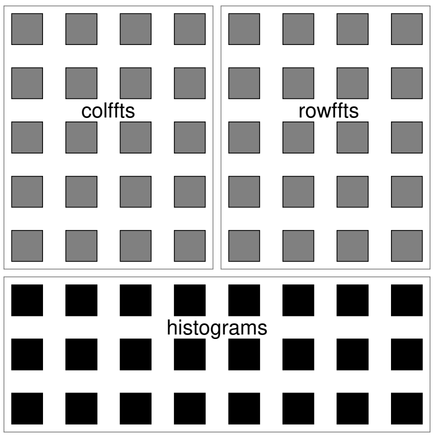

Figure 7: Initial mapping of the FFT-histograms program

The profiler displays for the three distinct groups of processors that execute the three different modules are shown in Figure 8. The first display is for processors executing the *colffts-module*, second for *rowffts-module*, and the last for *histograms-module*. Note that since a particular module executes only one of the three subroutines, the times for others are zero and omitted from the displays for conciseness. The communication between modules is done in the `main` routines of the modules, and the display shows that the main routines incur all the communication costs and overheads.

The maximum total execution time for the program is around 59 seconds. We make the following observations for the three modules:
colffts-module: This module spends a lot of time waiting to send (to rowffts module), as it completes its own computation faster than others. All the computation is scalable, and overheads are minimal.
rowffts-module: This module does not wait for input, but has to wait for some time to send output. It is running slower than the *colffts-module* mainly because it has to do more computation and communication (it does a send and a receive, while the *colffts-module* only does a send), but is faster than the *histograms-module*. All the computation is scalable with minimal overhead, suggesting that increasing the number of processors will speed it up.
histograms-module: This module is the bottleneck as it takes the longest time to execute. Further, most of the time is spent in non scalable user computations, suggesting that adding more processors may not be useful, but replication may improve the performance.

Using these observations, we modify the mapping in an attempt to improve performance [11]. We increase the number of processors assigned to *rowffts-module* from 20 to 24. The *histograms* module is replicated by two, and each instance is assigned 10 processors (minimum possible due to memory limitations). The *colffts-module* is left unchanged. The resulting mapping is shown in Figure 9. We omit the profiler display for the new mapping due to lack of space, and simply report that the total execution time decreased by around 22% to around 46 seconds, mainly because the *histograms-module* started executing considerably more efficiently.

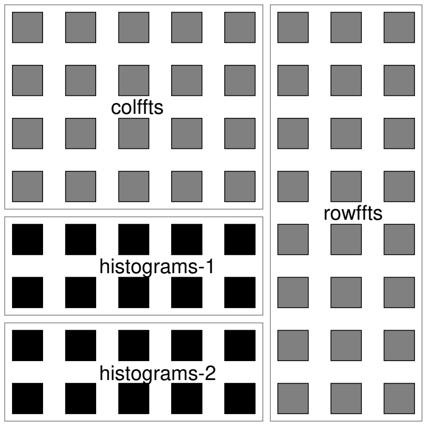

Figure 9: Final mapping of the FFT-histograms program

8 Conclusions

We have described a profiler for compiler parallelized programs. The profiler distinguishes eleven logical categories that characterize the execution of user programs, and presents that information in terms of the user's view of execution of the source code.

These categories provide a detailed breakdown of the overheads in a parallel program, including the time spent on overheads created by the compiler. For example, changing the distribution of an array in a compiled program is an activity that should be reported as overhead, although all processors may be busy. Support for such a detailed user's view of execution requires cooperation between the profiler and the compiler. Our profiler is integrated with the compiler, and uses the information generated by the compiler to obtain separate timing information for each routine and category. Such coupling not only helps in providing meaningful source level information to a user, but also helps reduce the overhead of profiling, which is less than 1% for our system.

We classify the execution time of a program into scalable and non-scalable categories, and the non-scalable code is classified further into several categories based on the reason why it is not scalable. In our experience, the clear classification of the non-scalable code significantly helps programmers understand the performance problems of their programs, and provides a rational basis for tuning parallel programs.

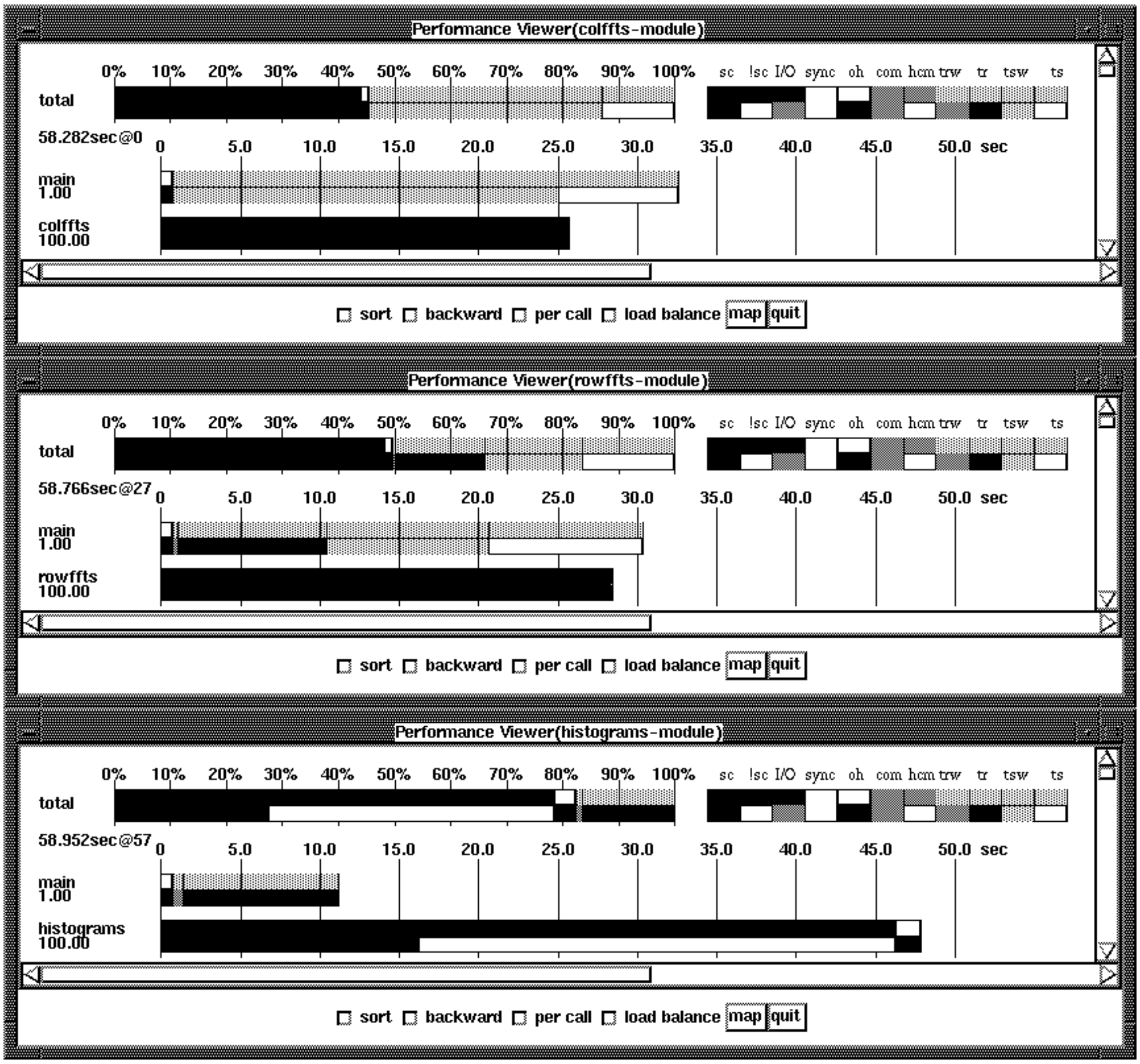

Figure 8: Profiler display for the FFT-histograms program

References

[1] Bernstein, D., Bolmarcich, A., and So, K. Performance visualization of parallel programs on a shared memory multiprocessor system. *ICPP Vol. 2* (Aug. 1989), pp.1–10.

[2] Gaur, Y., Jr., V. A. G., and Jablonowski, D. An environment for performance experimentation on multiprocessors. *Supercomputing'89* (1989), 589–596.

[3] Goldberg, A. J., and Hennessy, J. L. Mtool: An integrated system for performance debugging shared memory multiprocessor applications. *IEEE trans. on Parallel Distributed syst. Vol. 4*, 1 (Jan. 1993), pp.28–40.

[4] Heath, M. T., and Etheridge, J. A. Visualizing the performance of parallel programs. *IEEE Software Vol. 8*, 5 (Sept. 1991), pp.29–39.

[5] High Performance Fortran Forum. *High Performance Fortran Language Specification, Version 1.0*, May 1993.

[6] Kimelman, D. N., and Ngo, T. A. The RP3 program visualization environment. *IBM Journal of R&D Vol. 35*, 5–6 (Sept–Nov 1991), pp.635–651.

[7] Miller, B. P., Clark, M., Hollingsworth, J., Kierstead, S., Lim, S.-S., and Torzewski, T. IPS-2: The second generation of a parallel program measurement system. *IEEE trans. on Parallel Distributed syst. Vol. 1*, 2 (Apr. 1990), pp.206–217.

[8] Rover, D. T. A performance visualization paradigm for data parallel computing. *Proc. 25th Hawaii Intl conf on syst sci Vol. 2* (1992), pp.149–160.

[9] Sarukkai, S. R., Kimelman, D., and Rudolph, L. A methodology for visualizing performance of loosely synchronous programs. *SHPCC* (Apr. 1992), 424–432.

[10] Subhlok, J., Stichnoth, J., O'Hallaron, D., and Gross, T. Exploiting task and data parallelism on a multicomputer. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (May 1993), pp. 13–22.

[11] Suzuoka, T., Subhlok, J., and Gross, T. *Performance debugging based on scalability analysis*, vol. Technical Report CMU-CS94-133. Department of Computer Science, Carnegie-Mellon University, 1994.

[12] Szelényi, F., and Zecca, V. Visualizing parallel execution of FORTRAN programs. *IBM Journal of R&D Vol. 35*, 1–2 (Jan–Mar 1991), pp.70–282.