Appears in Concurrency: Practice and Experience, Vol 9(10),927-945,(October 1997)

# A Performance Debugging Tool for High Performance Fortran Programs

**Takashi Suzuoka**[1] **Jaspal Subhlok**[2] **Thomas Gross**[2,3]

[1] Toshiba Corporation, 1, Komukai, Toshiba-Cho, Saiwai-Ku, Kawasaki 210, Japan
[2] School of Computer Science, Carnegie Mellon University, Pittsburgh, PA 15213
[3] Institut fuer Computersysteme, ETH Zuerich, 8092 Zuerich, Switzerland

## Summary

**Parallel languages allow the programmer to express parallelism at a high level. The management of parallelism and the generation of inter–processor communication is left to the compiler and the runtime system. This approach to parallel programming is particularly attractive if a suitable widely accepted parallel language is available. High Performance Fortran (HPF) has emerged as the first popular machine independent parallel language and remarkable progress has been made towards compiling HPF efficiently. However, the performance of HPF programs is often poor and unpredictable, and obtaining adequate performance is a major stumbling block that must be overcome if HPF is to gain widespread acceptance. The programmer is often in the dark about how to improve the performance of an HPF program since poor performance can be attributed to a variety of reasons, including poor choice of algorithm, limited use of parallelism, or an inefficient data mapping.**

**This paper presents a profiling tool that allows the programmer to identify the regions of the program that execute inefficiently, and to focus on the potential causes of poor performance. The central idea is to distinguish the code that is executing efficiently from the code that is executing poorly. Efficient code uses all processors of a parallel system to make progress, while inefficient code causes processors to wait, execute replicated code, idle, communicate, or perform compiler bookkeeping. We designate the latter code as *non-scalable*, since adding more processors generally does not lead to improved performance for such code. By analogy, the former code is called *scalable*. The tool presented here separates a program into scalable and non-scalable components and identifies the causes of non-scalability of different components. We show that compiler information is the key to dividing the execution times into logical categories that are meaningful to the programmer. We present the design and implementation of a profiler that is integrated with *Fx*, a compiler for a variant of HPF. 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.**

## 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, sometimes even to the algorithm. Thus, a performance debugging 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) [9], the programmer does not write an explicit parallel program, but influences the performance of the program only indirectly by the use of parallel constructs and compiler directives. The compiler generates code for the processors of the parallel system from the source program. So it is important that the feedback about the execution be in terms of the source program and at a level that can be understood by the programmer.

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

Information that is very broad in scope, or not directly related to the program text, is of limited use for performance debugging. For example, a breakdown of the total time into computation and communication time alone may not be enough to provide guidance to the user on how to reduce the communication cost.

To speed up the execution of a program, one should 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 of a program into the time spent executing *scalable* code and the time spent executing *non-scalable* code. The scalable code corresponds to execution without significant parallelization overhead, so additional processors can contribute usefully to this component of execution time. The non-scalable code corresponds to computations that do not efficiently contribute to the progress of the program. For example, when a processor is idle, communicating, performing replicated computations or compiler bookkeeping, it is executing overhead operations – and not program computations. 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 used as a measure of how well the available processors are being used by the program. The broad objective of performance debugging, then, is to maximize the time a program spends executing scalable code and to 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 the relevant information for performance debugging with minimal overhead and without any external runtime support. The compiler is also used to guide the visualization of profiler data. In particular, for visualizing heterogeneous execution (e.g., MIMD mode when the processors 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 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 [5, 16]. The targets for the compiler include networks of workstations using the PVM library, the iWarp system using customized deposit based communication [15], the Cray T3D with SHMEM based communication [2], and the Intel Paragon with the PVM and NX libraries.

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 lists the classes of non-scalable code. Section 4 explains how the compiler knowledge is used by the profiler. Section 5 presents the visualization of profiler data, and Section 6 sketches the implementation. Section 7 describes how the profiler is used for performance debugging for two specific examples. Section 8 contains conclusions.

## 2 Related work

Profilers typically measure the time a processor spends in different program activities and present the time in terms of the source program. For example, a profiler may report the amount of time a processor is computing, communicating, waiting on synchronization, servicing cache misses, etc. Compiler information is often used to relate the runtime measurements to the source code. A number of profilers are discussed in the literature [4, 6, 8, 12, 13, 19].

A parallel language like HPF poses additional challenges for profiling. The relationship between the source code and the runtime measurements is difficult to establish, particularly in the presence of optimizing transformations. In general, it is not sufficient to relate physical activities like communication or idling to lines of the source code, since each line can potentially cause many different activities, and some activities can be meaningfully related only to blocks of code. For example, the communication generated to start a parallel loop can be meaningfully related only to the entire loop, not to individual statements. Also, reporting the communication overhead only for an entire loop hides the distinction between overheads incurred at loop startup and termination versus overheads incurred for each loop iteration. Hence, it is important to relate the profiling measurements to the logical program activities, not just physical parts of the program source.

Our system [18] as well as some other systems discussed in the recent literature [1, 11] attempt to present the measurements of processor states in terms of logical program activities; [1] addresses this problem with emphasis on how performance debugging is done in the presence of aggressive compiler restructuring of a data parallel program, while [11] emphasizes the use of dynamic performance measurement and automated search for performance problems. We assume a relatively simple compilation scheme where the focus of optimization is mainly on generation of efficient communication (in our experience this is the state of the art in compiling HPF), but we deal with programs that combine task and data parallelism, and we specifically focus on the kind of performance problems HPF programmers are likely to encounter on large-scale parallel systems. Profilers designed for shared memory parallel machines focus on memory and cache behavior [10], since that is a major determinant of performance. Our profiler is primarily designed for distributed memory parallel machines with explicit inter–processor communication. In this environment, the primary performance bottlenecks are synchronization and communication overheads.

Popular commercial profiling systems include Visualizing Tool (VT) for the IBM SP2, Code Vision for the Silicon Graphics Challenge, PRISM and Node PRISM for the Thinking Machines CM-5, and Apprentice for the Cray T3D. VT and Code Vision [7] are designed for explicit parallel programs using a message passing or a shared memory model. Apprentice for the Cray T3D [20] is targeted towards Cray's CRAFT programming model, which is a multithreaded programming model that supports distributed data structures. Node PRISM [14] is designed to provide uniform performance debugging support for data parallel and message passing programs.

Other forms of performance analysis include static analysis, animation and simulation. Static analysis [3] is an effective tool for performance estimation, but many performance problems come to surface only during actual execution. An animation provides a detailed view of execution and may help to understand a complicated situation. However, presenting the important aspects of an execution in terms of the source program is generally much faster than replaying an animation and is sufficient to identify most performance bugs. Also, in compiler parallelized programs, it becomes more difficult to relate an animation to the source program. Simulation of parallel execution is 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. Finally, we refer to [7] for a survey of performance analysis tools.

# 3 A criterion for efficient execution

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. This heuristic has obvious merit and is the method of choice for sequential programs. However, the situation is more complex for parallel programs. Compilation of a parallel program involves mapping of computations and data to the nodes of the parallel system and generation of code for each processor in the system. This research only addresses performance issues related to parallelization. In this context, the broad criterion for efficient execution is whether parallelization is effective in improving the overall program performance.

Parallel computing works best when all processors are busy and directly contributing to the progress of the user's program. Redundant or overhead computations, communication, and waiting do not contribute directly to the original program, and are generally not helped proportionately (or at all) as more processors are made available to the program. For this reason, we refer to these activities as *non-scalable*. The performance tuning problem is to get the best performance from a fixed number of processors, but in practice, users of parallel systems are also interested in the scalability of their program. These issues, however, are closely related; in general if the program is not using the available processors efficiently, it is not likely to scale well. Our goal is to identify the regions of non-scalable code and focus on them for performance debugging.

By analogy, we label the part of a processor's work that directly contributes to the progress of a program as *scalable*, implying that it is using the processors efficiently and its performance is expected to increase meaningfully with additional processors, even though few computations scale linearly. The *non-scalable* parts of a program are responsible for inefficiency in execution and should be targeted for improvement. To be sure, most parallel programs have non-scalable components that cannot be eliminated. However, it is important for the programmer to focus on non-scalable code for performance debugging. The approach is analogous 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:** refers to user computations that do not contribute efficiently to the progress of the program. Any program computation that is replicated on multiple processors uses only one processor effectively, and hence, is not scalable. Similarly, a loop computation with a loop carried data dependence may use only one processor at a time. 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. In any case, it is important that the user be aware of the time spent on I/O activity.

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

**Parallelization overhead:** results 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.

**Communication time:** results from transfer of data between processors. The actual behavior of communication steps depends on the communication pattern, but communication is generally 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 the communication time typically does not decrease linearly with the number of processors and often it increases. Further classification of communication times is discussed in Section 4 and Section 7.

Table 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. The point is that the non-scalable code can be classified into meaningful categories for program profiling.

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

Table 1: Processor states in parallel execution

We believe that it is important that the profiler provide a precise measurement of these categories and not just a summary judgement. In general, a simple verdict (such as poor scalability, load imbalance, or poor mapping of distributed arrays) cannot be made accurately as there can be many possible causes of poor performance. The mapping between the symptoms and the causes of poor performance is many to many and of a complex nature. In our model, the profiler provides symptoms of poor performance and possible relationship between symptoms and causes, but the final identification of causes is left to the user.

# 4 Compiler information

It is important to utilize information from the parallelizing compiler 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. Compiler information may not be available for externally linked routines (e.g. HPF EXTRINSIC routines), which may be in a different language and may use a different communication and synchronization model. In this paper we assume that compiler information is available for all relevant routines although it is possible to provide limited profile information for external routines.

We outline some of 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. Compiler information is essential for distinguishing between these computation categories. For instance, a user cannot find the compiler overhead time by hand instrumentation, or by using a runtime tool.

**Classifying communication:** In the compilation of HPF and other parallel languages, communication is primarily generated by the compiler. A part of the communication is related to specific 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. However, since all communication is generated by the compiler, it is easy to classify the communication into different categories with compiler knowledge. The compiler's cooperation also allows classification of communication operations that cannot be associated with a single statement. For example, the compiler may determine that an array must be realigned at the beginning of a function or a procedure. If there are multiple references to this array in the function body, an optimization is to perform the realignment once upon entry. Such communication is due to more than one statement, and we refer to it as *hidden* communication (if we only look at communication operation that is directly attributed to statements, hidden communication operations do not show up).

**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. For task parallel programs, compiler information is used to distinguish between communication inside modules and communication between modules. Compiler knowledge is also used to relate the runtime measurements to the source code and visualize it in terms of the mapping of tasks to processors.

# 5 Performance visualization

We address some of the issues in the design of a visualization system and describe the decisions made for our profiler. The information is normally presented in graphical form, as in Figure 2, 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. 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 and range 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 the 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 is integrated with the Fx compiler [5, 16] and 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 outline the Fx language and compiler, briefly describe the implementation of these two major components, and discuss the overhead of profiling. We refer the reader to [18] for the details of the implementation.

## 6.1 Fx language and compiler

The Fx language is Fortran 77 with array extensions of Fortran 90 and directives to express data layout and parallelism. The data layout directives are similar to those in HPF although there are some minor differences since HPF and Fx evolved in parallel. Fx also supports task parallelism which allows different subprograms to execute on different subgroups of processors. Task parallelism is not a part of the current HPF standard but is one of the endorsed HPF extensions. The targets for the Fx compiler include networks of workstations using the PVM library, the iWarp system using customized deposit based communication, the Cray T3D with SHMEM based communication, and the Intel Paragon with the PVM and NX libraries. All results presented in this paper are gathered for the iWarp system.

## 6.2 Data collector

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

The profiler has knowledge of the static compiler database. For example, 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 Fortran 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/processors)` | |
| `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 along with total time and has 1 counter for each subroutine or region of interest. Therefore, it requires only 12$\times$(number of subroutines or regions of interest) words of data memory. The timing categories correspond to the labels in the main display of the profiler shown in Figure 1 and are described in Section 7. The collected data is written to a file at the termination of the program.

The use of a single set of counters per region of interest makes the memory overhead negligible, which is also a factor in limiting the execution time overhead of profiling. However, this also implies that certain kinds of information, e.g. the variation in execution time for different calls to a subroutine, is not available for visualization. This is an important tradeoff, and in our view, the additional overhead of maintaining different timers for different execution instances does not justify the potential use of this information.

### 6.3 Performance viewer

The viewer has a window based interface implemented in C and Tcl/Tk. Figure 1 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 *routine/file* space. The user can choose to look at any or all of the 11 timing categories by selecting the appropriate buttons. The display in Figure 1 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 highlighted set of processors. The corresponding performance display is shown at the top in Figure 8 and described in Section 7.

The display in Figure 1 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 in Section 7.

### 6.4 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 slowdown has always been 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. Also, the memory used by the profiler is very small and does not impact the execution of the program.

## 7 Using the profiler to improve parallel Fortran programs

We describe how the visualization of performance with the profiler is used to identify and correct several kinds of performance bugs in parallel Fx Fortran programs, and illustrate the process with examples. The central idea is that even though the final parallel program is generated by the compiler, performance visualization can logically lead to steps the programmer can take to correct performance bugs.

We first address performance debugging of data parallel modules executing in SPMD (Single Program Multiple Data) mode and then address the requirements of task parallel programs that contain multiple modules mapped on different parts of a processor array. The first example is a variant of a simple HPF program, while the second program uses the concept of task parallelism. Task parallelism is a key feature that distinguishes Fx from the current standard HPF, although this feature is now an endorsed extension of HPF. All examples use the Fx compiler and profiler executing on a 64 processor iWarp array.

### 7.1 Performance debugging a data parallel module

The profiler provides the following information about the execution of a program. The **boldface** font is used to print items as they appear in the profiler display.

- The amount of time spent in scalable computations (**sc**) and the amount of time spent in non-scalable (or non-parallel) user computations (**!sc**).

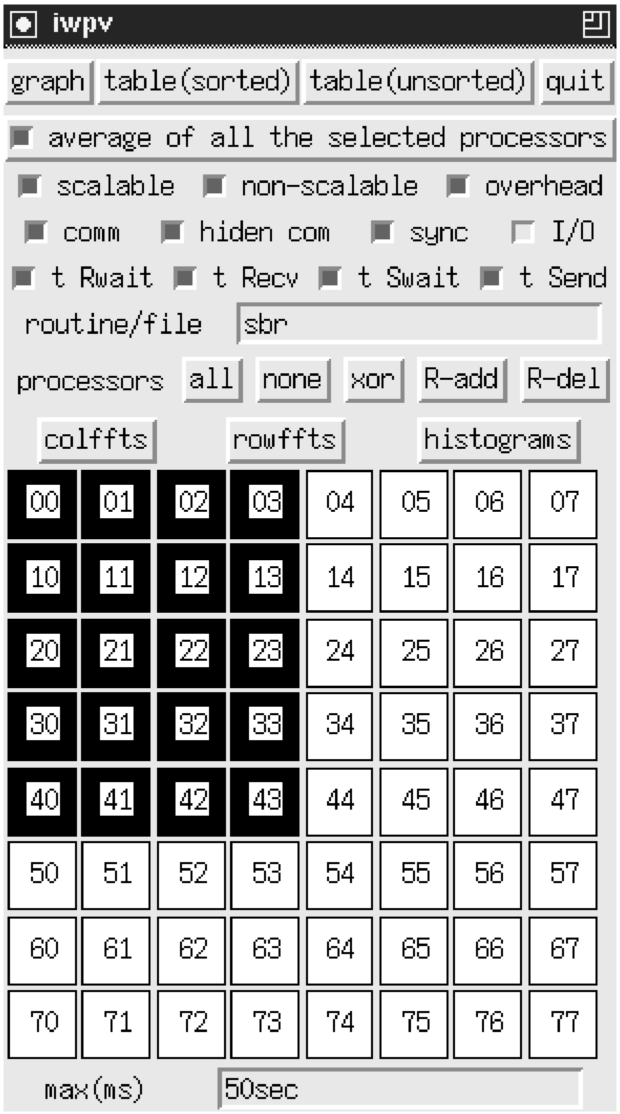


Figure 1: Profiler console

- 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 the 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**

Data parallel languages like HPF 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 can 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$\times$1024 matrix with a block distribution. The profiler display for average execution time for all processors is shown in Figure 2. Note that the top right corner of the figure is a two color legend showing the categories in which the program execution time is divided.

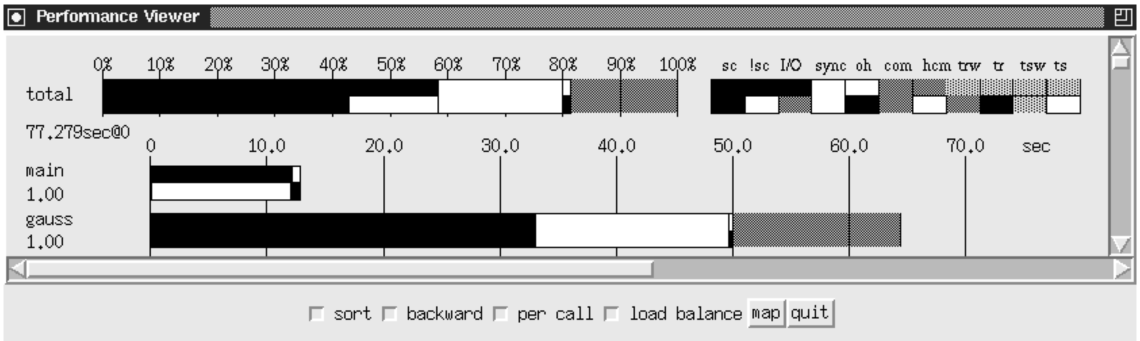


Figure 2: Profiler display for Gaussian elimination

The display shows that the program spends around 65 seconds (or 85% of the total time) in the `gauss` routine, and around 12 seconds (or 15% of the total time) in the `main` routine. Looking at the top bar for the breakdown in categories, we see that this program spends about 42% of the time in scalable computation, around 16% in non-scalable user computation (**!sc**), about 1% in parallelization overhead (**oh**), around 20% of the time in communication (**com**), and around 22% in synchronization (**sync**). In this example, no time is spent on hidden communication and this category is omitted from the legend. The display shows a total execution time around 77.3 seconds.

Our first observation is that `gauss` requires significant synchronization overhead, which is the second-largest category after the scalable computation time. To get a better idea, we display the profile data for individual processors in Figure 3. This figure shows 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 almost 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 the 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 then converted to a parallel loop.

The profiler display for 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 which could not have been directly detected by simply examining the computation and communication related overheads.

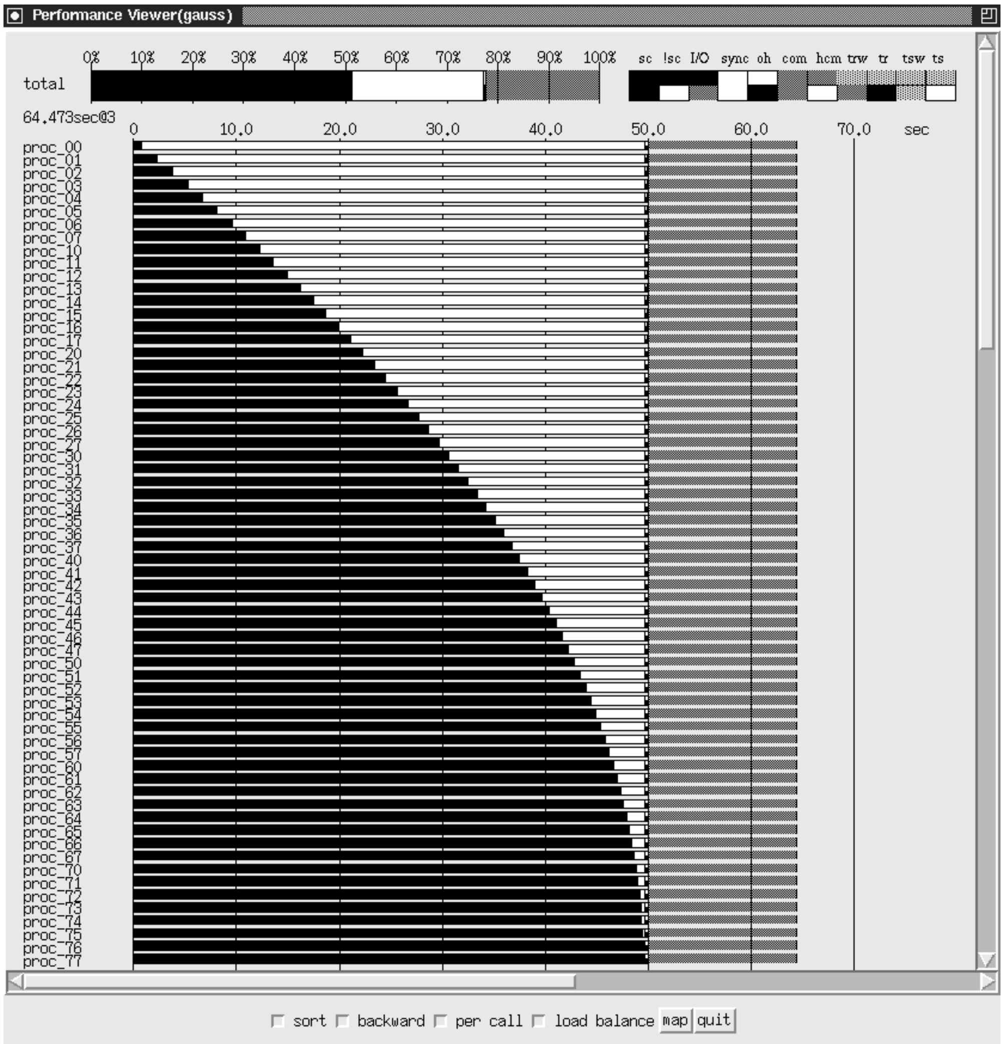


Figure 3: Load balance in Gaussian elimination

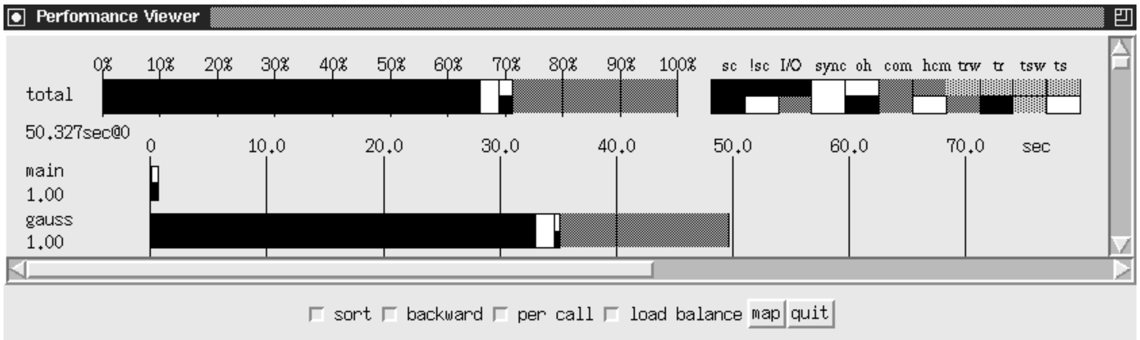


Figure 4: Profiler display for improved Gaussian elimination

### 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. 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. These are special forms of hidden communication and synchronization and profiler provides the following additional information for multiple modules:

- The time that a module is idle waiting to receive input from another module (**twr**), or to send output to another module (**tsw**).
- 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. Techniques for automatic synthesis of good mappings are discussed in [17]. Here we describe some example problems in mapping that can lead to poor performance and show how the profiler is used to correct them.

### 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

Replication is a powerful optimization that can be used with task parallelism. Suppose `M` is a subroutine called in a loop inside a module. Also, assume 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.

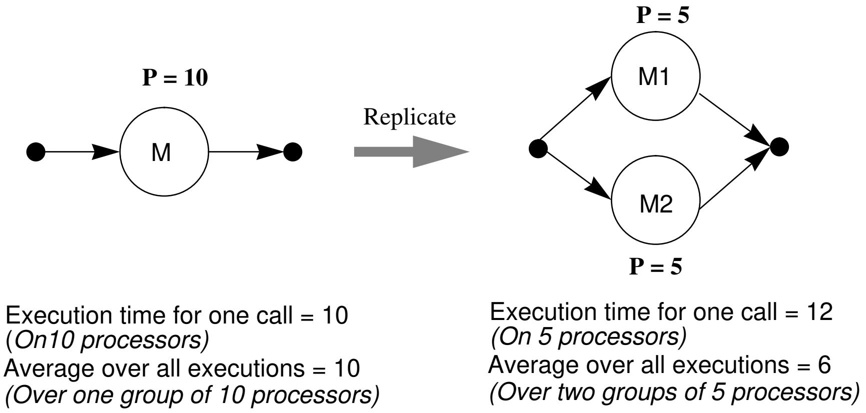


Figure 5: Replication of modules

Legality of replication is dependent on data dependence and memory constraints [16]. However, the profiler can help us identify when replication is potentially useful. If 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.

## 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.

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

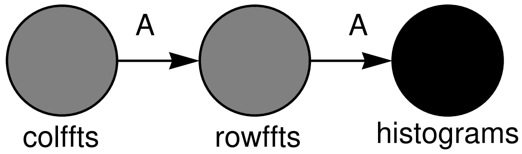


Figure 6: Structure of the FFT-histograms program

The three subroutines correspond to modules *colffts-module, rowffts-module*, and *histograms-module*, which 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 no loop carried dependences in the main program loop. We start by assigning approximately equal 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.

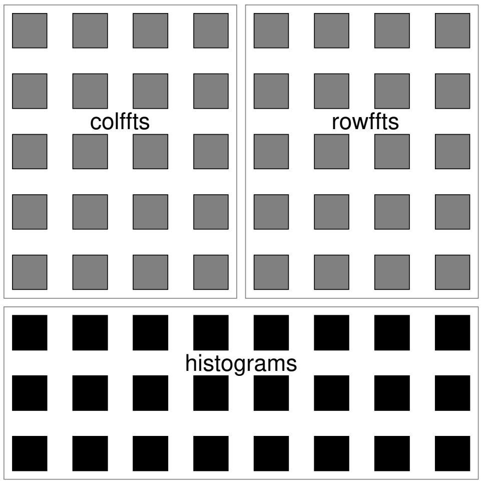


Figure 7: Initial mapping of the FFT-histograms program

The profiler displays for the three distinct groups of processors that execute the three different modules are shown in Figure 8. The top display is for processors executing the *colffts-module*, the middle for the *rowffts-module*, and the bottom for the *histograms-module*. Note that since a particular module executes only one of the three subroutines, the times for the others are zero and omitted from the displays for conciseness. The communication between the 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 fair amount 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 must wait for some time to send output. It is running slower than the *colffts-module*, mainly because it must perform more computation and communication (it sends and receives,

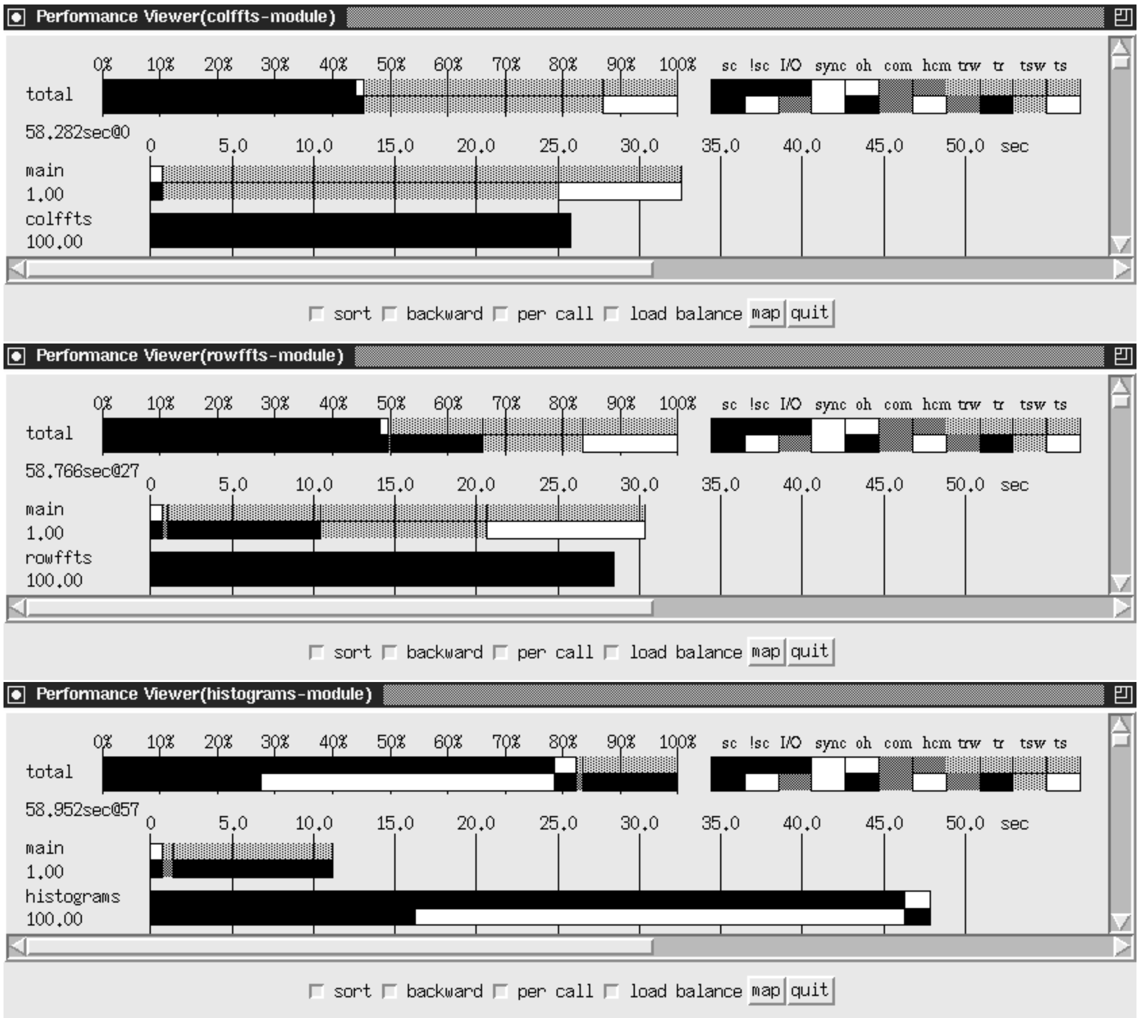


Figure 8: Profiler display for the FFT-histograms program

while the *colffts-module* only sends), 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 compute its results. 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 performance.

Using these observations, we modify the mapping in an attempt to improve performance. 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. The profiler display for execution with the new mapping is shown in Figure 10. We observe that the total execution time decreased by around 22% to around 46 seconds, mainly because the *histograms-module* started executing considerably more efficiently.

# 8 Conclusions

We have described a profiler for programs written in a parallel language like High Performance Fortran. The profiler uses compiler information to improve the collection and presentation of performance information. The compiler assists in minimizing the overhead of profiling, which is typically less than 1% of the execution time in our system, by directing the placement of the profiling code. The compiler also facilitates finer classification of overheads in terms of

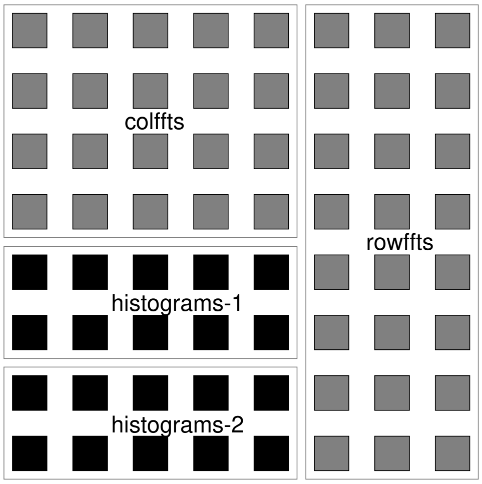


Figure 9: Final mapping of the FFT-histograms program

program activities and allows a meaningful display of the profiler data in terms of the source program.

To bridge the gap between the source program and parallel object code, this tool classifies the execution time of a program into scalable and non-scalable categories, reflecting the "efficient and useful work" and "overhead" components of parallel execution. The non-scalable code is further classified into several categories based on the reason why it causes inefficiency in parallel execution. In our experience, a 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.

This paper focuses on performance problems likely to be encountered by HPF programmers for current target systems. In summary, the paper shows that collection and display of relevant runtime information can be done efficiently in collaboration with an HPF compiler. The success of the compiler's involvement in profiling provides yet another reason why a good compiler is a crucial tool for the development of efficient programs for high-performance parallel computers.

## Acknowledgements

We appreciate the insightful comments of the referees which helped in improving the presentation of this paper. We also thank Peter Dinda, Susan Hinrichs, David O'Hallaron, James Stichnoth, Thomas Stricker, and Bwolen Yang for their contributions to the Fx compiler developed at Carnegie Mellon University.

## References

[1] ADVE, V., MELLOR-CRUMMEY, J., ANDERSON, M., KENNEDY, K., WANG, J., AND REED, D. Integrating compilation and performance analysis for data-parallel programs. In *Supercomputing '95* (San Diego, CA, Dec 1995).

[2] BARRIUSO, R., AND KNIES, A. SHMEM User's Guide for C. Tech. rep., Cray Research Inc., June 1994. Revision 2.1.

[3] FAHRINGER, T., BASKO, R., AND ZIMA, H. Automatic performance prediction to support parallelization of Fortran programs for massively parallel systems. In *Proceedings of the 1992 International Conference on Supercomputing* (Washington, DC, July 1992), pp. 347–56.

[4] GAUR, Y., GUARNA, V., AND JABLONOWSKI, D. An environment for performance experimentation on multiprocessors. In *Supercomputing'89* (1989), pp. 589–596.

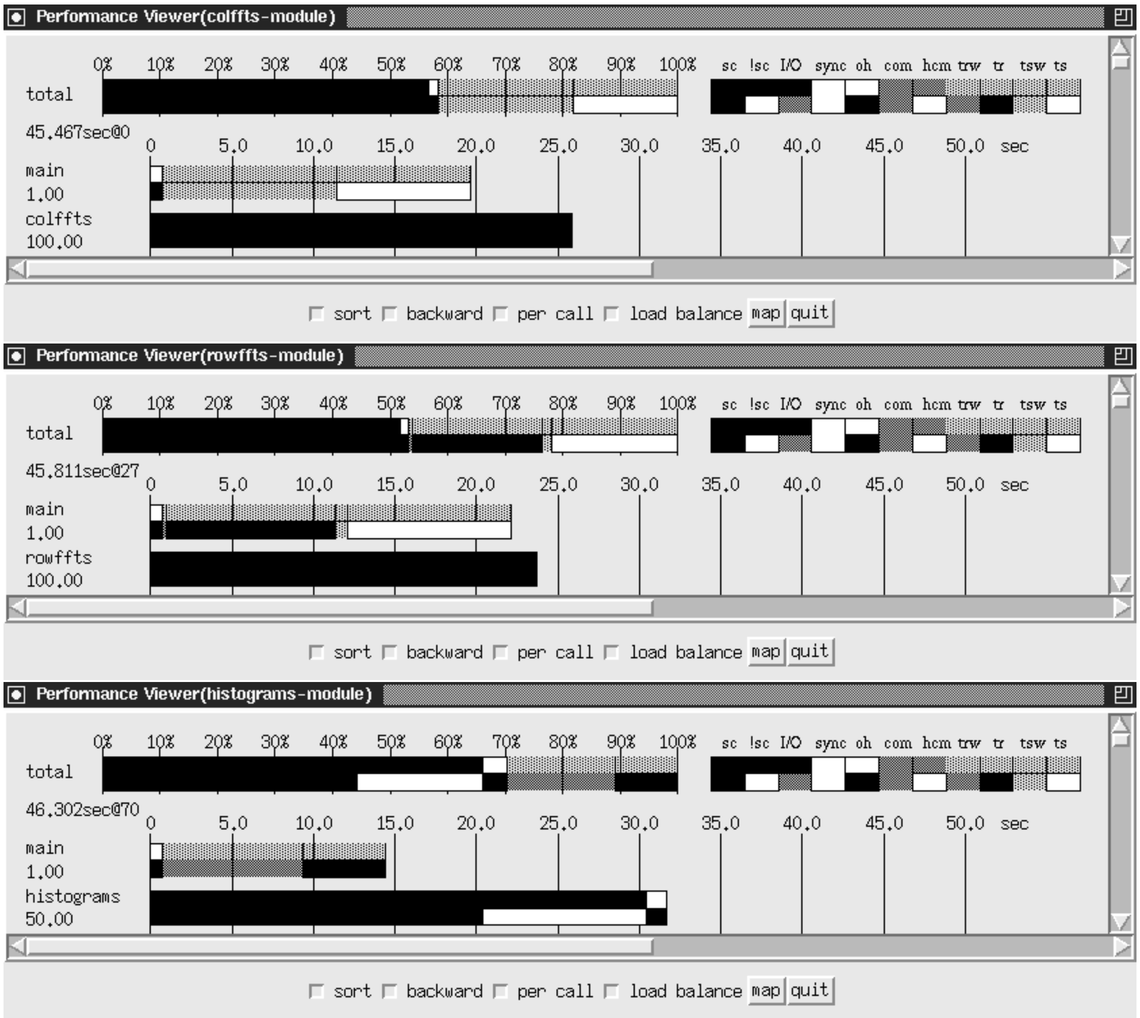


Figure 10: Profiler display for the improved FFT-histograms program


[5] GROSS, T., O'HALLARON, D., AND SUBHLOK, J. Task parallelism in a High Performance Fortran framework. *IEEE Parallel & Distributed Technology*, 3 (1994), 16–26.

[6] HEATH, M., AND ETHERIDGE, J. Visualizing the performance of parallel programs. *IEEE Software 8*, 5 (Sept. 1991), 29–39.

[7] HONDROUDAKIS, A. Performance analysis tools for parallel programs. Tech. rep., Edinburgh Parallel Computing Centre, July 1995. Available at: http://www.epcc.ed.ac.uk/epcc-tec/documents.html.

[8] KIMELMAN, D., AND NGO, T. The RP3 program visualization environment. *IBM Journal of R&D 35*, 5-6 (Sep-Nov 1991), 635–651.

[9] KOELBEL, C., LOVEMAN, D., STEELE, G., AND ZOSEL, M. *The High Performance Fortran Handbook.* The MIT Press, Cambridge, MA, 1994.

[10] MARTONOSI, M., GUPTA, A., AND ANDERSON, T. MemSpy: Analyzing memory systems bottlenecks in programs. In *Proceedings of ACM SIGMETRICS Conference on Measurements and Modeling of Computer Systems* (June 1992), pp. 1–12.

[11] MILLER, B., CALLAGHAN, M., HOLLINGSWORTH, J., IRVIN, R., KARAVANIC, K., KUNCHITHAPADAM, K., AND NEWHALL, T. The Paradyn parallel performance measurement tool. *Computer* (November 1995), 37–46.

[12] ROVER, D. A performance visualization paradigm for data parallel computing. In *Proceedings of the 25th Hawaii International Conference on Systems Sciences* (1992), vol. 2, pp. 149–160.

[13] SARUKKAI, S., KIMELMAN, D., AND RUDOLPH, L. A methodology for visualizing performance of loosely synchronous programs. In *Proceedings of Scalable High Performance Computing Conference* (Apr 1992), pp. 424–432.

[14] SISTARE, S., ALLEN, D., BOWKER, R., JOURDENAIS, K., SIMONS, J., AND TITLE, R. A scalable debugger for masively parallel message-passing programs. *IEEE Parallel & Distributed Technology: Systems & Applications 2*, 2 (1994), 50–56.

[15] STRICKER, T., STICHNOTH, J., O'HALLARON, D., HINRICHS, S., AND GROSS, T. Decoupling synchronization and data transfer in message passing systems of parallel computers. In *Proceedings of the 1995 International Conference on Supercomputing* (Barcelona, July 1995), ACM, pp. 1–10.

[16] SUBHLOK, J., O'HALLARON, D., GROSS, T., DINDA, P., AND WEBB, J. Communication and memory requirements as the basis for mapping task and data parallel programs. In *Supercomputing '94* (Washington, DC, November 1994), pp. 330–339.

[17] SUBHLOK, J., AND VONDRAN, G. Optimal latency–throughput tradeoffs for data parallel pipelines. In *Eighth Annual ACM Symposium on Parallel Algorithms and Architectures* (Padua, Italy, June 1996).

[18] SUZUOKA, T., SUBHLOK, J., AND GROSS, T. Performance debugging based on scalability analysis. In *Proceeding of the Fifth Symposium on the Frontiers of Massively Parallel Computation* (McLean, VA, February 1995), pp. 406–413. Expanded version available as Carnegie Mellon Computer Science Dept. Technical Report CMU-CS-94-133.

[19] SZELÉNYI, F., AND ZECCA, V. Visualizing parallel execution of Fortran programs. *IBM Journal of R&D 35*, 1-2 (Jan-Mar 1991), 70–282.

[20] WILLIAMS, W., HOEL, T., AND PASE, D. The MPP Apprentice performance tool: Delivering the performance of the Cray T3D. In *Programming Environments for Massively Parallel Distributed Systems*, K. M. D. et al., Ed. Birkhauser Verlag, 1994, pp. 333–345.