Performance Debugging
Based on Scalability Analysis

Takashi Suzuoka, Jaspal Subhlok and Thomas Gross

June 21, 1994

CMU-CS-94-133

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

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 ten 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.

This research was supported in part by Toshiba Corporation, Research and Development Center, 1 Komukai Toshiba-Cho, Saiwai-ku, Kawasaki 210, Japan, and was performed while Takashi Suzuoka was a visiting scientist at Carnegie Mellon.

Keywords: parallel processing, performance evaluation, performance debugging, performance monitor, scalability analysis

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) [Hig93], the programmer does not write an explicit parallel program, but influences the performance of the program by the use 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, at a level that can be understood by the programmer. Information at a lower level is of limited use for performance debugging. For example, some profilers show a breakdown of computation and communication time. However, this information alone is not enough to provide guidance to the user on how to reduce communication. And not all computations contribute to the progress of the program, an example computations that are replicated elsewhere.

To allow a user to speed up the execution of a program, a profiler must focus on 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 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 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 processors could 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 not only get all the relevant information for performance debugging, but the overhead of profiling is also extremely small. 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 [SSOG93]. The targets for the compiler include networks of workstations using the PVM library and the iWarp system, and will include the Cray T3D in the future.

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 describes the visualization of profiler data, and Section 5 describes the implementation. Section 6 describes how the profiler is used for performance debugging, and Section 7 has conclusions.

2 Related work

Profiling of parallel programs has been an active area of research for several years. In this section, we relate our profiler to some of the other profiling systems that are described in the literature.

Traditional 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 of such profilers include IPS-2 [MCH$^+$90], ParaGraph [HE91], and several others [BBS89, SKR92, KN91, Rov92]. Such profilers generally work by probing the processor state, and they can be independent of the compiler. However, they are not adequate if the compiler performs code transformations, since the mapping between machine instructions and the user's program is difficult to understand.

Several profilers address this problem by using compiler knowledge, and they present the processor state information in terms of the source program. The examples of such systems include Prism [Thi91], Express [Par90], Mtool [GH93], and several others, including [GJJ89, SZ91]. The main difficulty with these systems is that they 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 hides several important details.

Our objective is to present all information in terms of a user's view of the program, i.e. the information refers to the source program, and timings are presented in terms of the logical operations performed by a processor.

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 typically 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 five categories:

User time: Any program computation that is replicated on all 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.

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.

Communication time: The actual behavior of communication steps completely 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 certainly does not decrease linearly with processors, and may actually increase. Further classification of communication times is discussed in Section 6.

[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.

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.

Input/Output time: The final category is I/O, which may or may not be scalable depending on the I/O pattern and the I/O devices. But it is important that the user be aware of the time spent on this activity.

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 sections)
	synchronization
	communication
	parallelization overhead
	I/O

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 Performance visualization

Programmers want to observe the behavior of their program in a way that can be used for performance debugging. We address the main issues facing the designer of a visualization system, and describe the decisions made for our profiler.

4.1 Granularity

We have chosen to display the profiler information in terms of the source code rather than a graphical program animation, since the objective of visualization is to identify performance bugs in the source code.

Animated program execution provides some benefits but leaves it to the user to relate the observed behavior to the source program.

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 [Par90, Thi91, GJJ89]. When using a parallelizing compiler, the relationship between statements in the source program, and the executing program for which the profiler data is collected, becomes blurred. In particular, communication is generated uniformly for constructs like loops that span multiple basic blocks and cannot be associated with a particular statement. Similarly, the synchronization overhead is typically caused by the nature of computations that can span across a whole subprogram unit. Parallelizing transformations make the situation only worse.

We have chosen to display information at the subroutine (or function) level by default. These subprogram units usually cleanly break the major computations in the program, and in our experience, analyzing the inefficient behavior is most effective at the level of specific subroutines. This strategy may not work well when subroutines are large, and contain several major computation steps. To handle such situations, the profiler allows the programmer to explicitly divide a function or subroutine into smaller regions of interest, each of which then becomes a unit for the display of profiler information.

4.2 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.

A special situation emerges when the programmer uses task parallelism. With task parallelism, the processors are divided into groups. All processors inside a group perform similar computations, but processors in different groups typically perform completely different computations. The profiler allows the programmer to visualize information over these groups to allow meaningful browsing of profiler data.

In some atypical cases, the execution times and other performance data vary widely among processors. In such cases, it is useful to have an abstract idea of overall load imbalance without having to look at each processor 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 a zero value for a perfectly load balanced computation.

4.3 Graphs and tables

The profiler present its information in the form of graphical bar charts, as shown in Figure 3. However, the information can also be provided in the form of a table if needed. This is useful if other programs have to operate on the profiler's output.

5 Implementation

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

5.1 Data collector

The data collector is integrated with the Fx compiler [SSOG93] 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 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.

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 procedure call, there is only the cost of reading the timer and adding it to an accumulator. Most high-performance systems provide some form of hardware timer register, and timer access requires only a few clocks. The profiler collects 11 kinds of timer data for each subroutine or region of interest, and therefore requires only $11\times$(number of subroutines) words of data memory. The collected data is written to a file at the termination of the user's program.

5.2 Performance viewer

The viewer has a window based interface that is implemented in C and Tcl/Tk[Ous94]. 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 for performance viewing, by directly clicking on them, or using the menu on the row starting with *all*. The user can select any group of subroutines for viewing by naming one directly, or naming a file that contains them in the `routines/file` space. The user can choose to look at any or all of the 10 categories listed in the rows of squares starting with *scale* and *comm* squares. The user can also select the scale of the map using the *max(ms)* space.

.

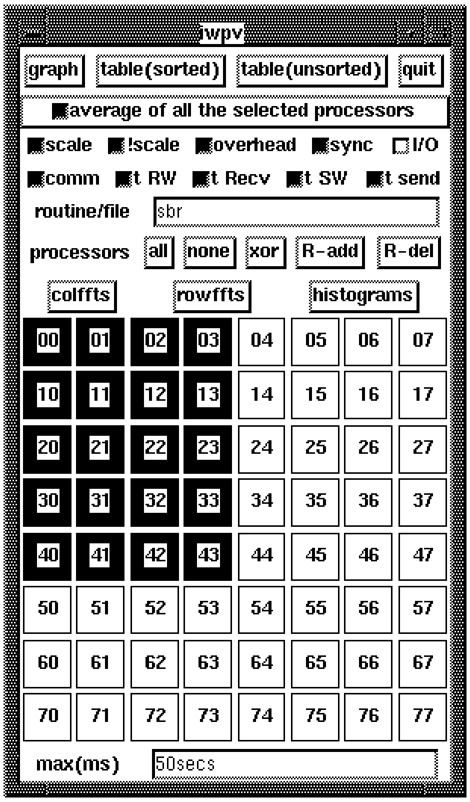

Figure 2: Profiler console

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 9, and described in more detail in Section 6

5.3 Perturbation by the profiler

This profiler has a very low overhead, and barely perturbs the execution of the user programs. For example, the programs discussed in Section 6 run only 0.12%(Gaussian Elimination) and 0.29%(FFT-Histograms) slower with the profiler than without the profiler. We have used the profiler with a range of other 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.

6 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.

6.1 Performance debugging a data parallel module

The profiler provides the following information for the program, as outlined in Section 3.

- 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 communication inside a module (**com**).
- The amount of time spent in synchronization (**sync**).
- The amount of time spent in I/O (**I/O**).
- The longest total execution time among selected processors (on the top left corner of display).

The profiler gives the option of visualizing the timing information in several different forms, as discussed in Section 4. In a typical performance debugging session, the programmer first looks at the whole program view, and then focuses on the performance of particular subroutines, on all or some processors. Examination of profiler output can lead to detection of a variety of performance bugs. We 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.

Suitability of algorithms for other architectures

An examination of scalability, and computation to communication ratio helps us decide how the program will perform in a different computation environment and for different numbers of processors. For example, a scalable program with low communication can be implemented efficiently on a network of loosely coupled workstations, while a high performance parallel machine with a good communication system is necessary for programs that have large non-scalable computations, or a high communication overhead. While this is not a performance problem, it is often an important step in application development.

Example: Gaussian elimination

Our first example is an Fx parallel program to perform Gaussian elimination on a 1024$\times$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.

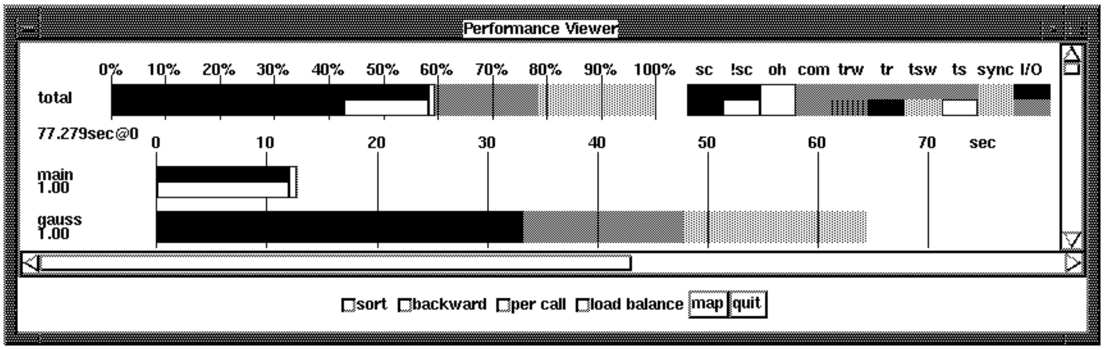

Figure 3: Profiler display for Gaussian elimination

The display shows that the program spends around 85% of the total time in `gauss` routine, and the remaining 15% 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(Figure 4) , 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 beneficial 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 5. The synchronization overhead in `gauss` is significantly lower, and the total time spent in `main` has become negligible. The maximum total

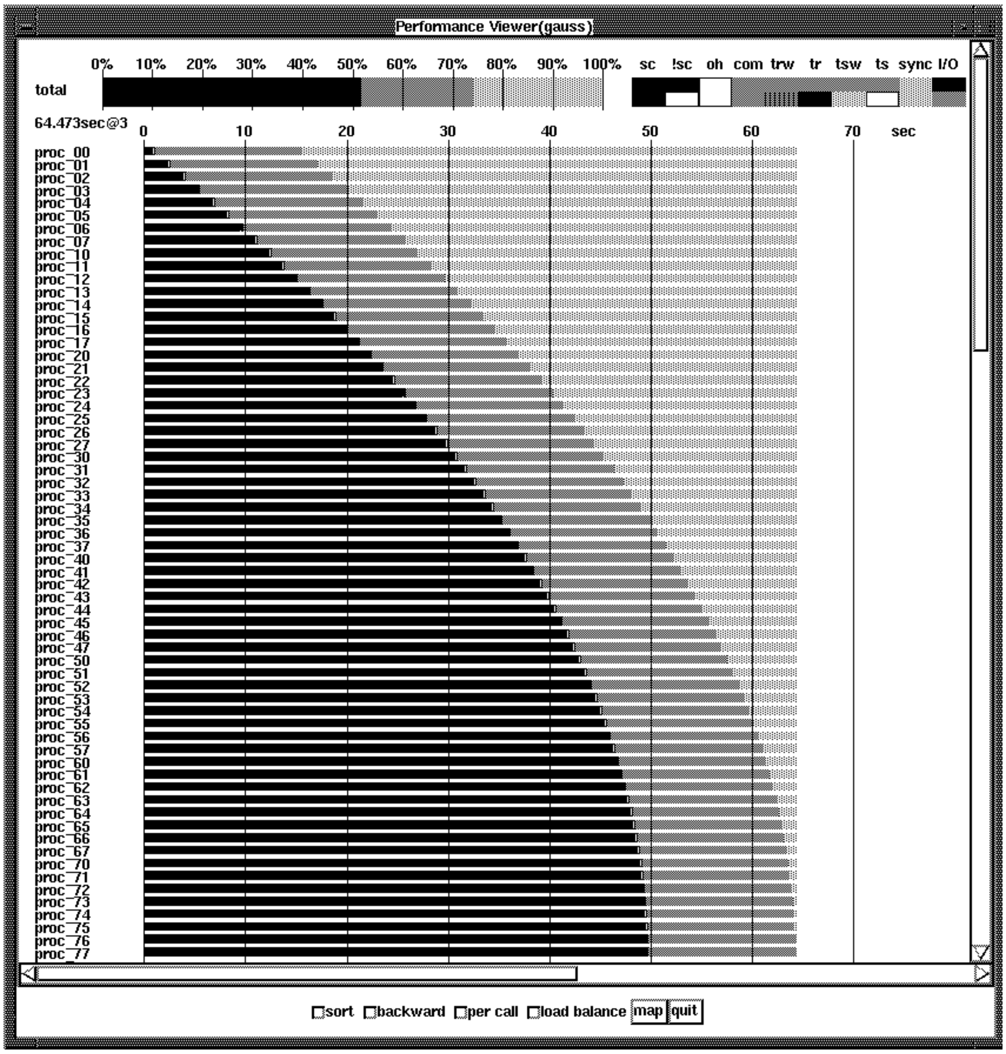

Figure 4: Load balance in Gaussian elimination

execution time is reduced by about 35% to 50.3 seconds.

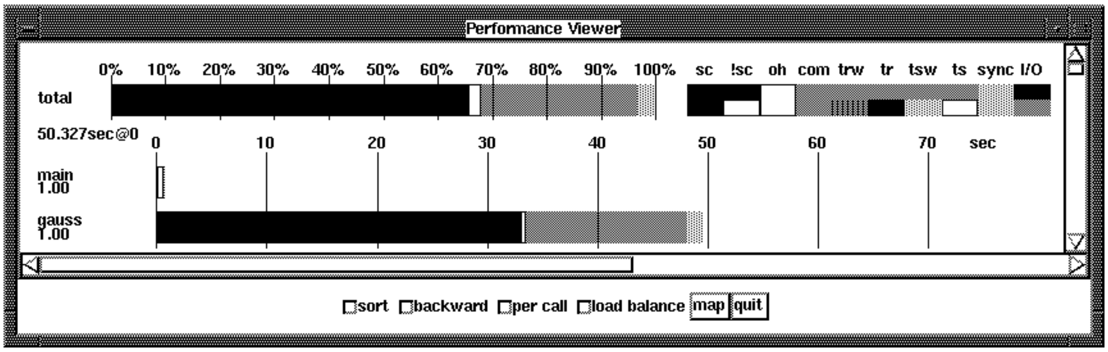

Figure 5: Profiler display for improved Gaussian elimination

6.2 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 [SSOG93]. 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:

- 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 (**tw**).

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 can be used to correct them.

Unbalanced allocation of processors to modules

The numbers of processors allocated to modules should be in proportion to the amount of work in the modules. When this is not the case, 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.

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.

Need to replicate modules

Legality of replication is dependent on data dependence and memory constraints, and its discussion is beyond the scope of this paper. However, the profiler can help us identify when replication is potentially useful. Need for replication is indicated when a module is spending a large amount of time executing non-scalable

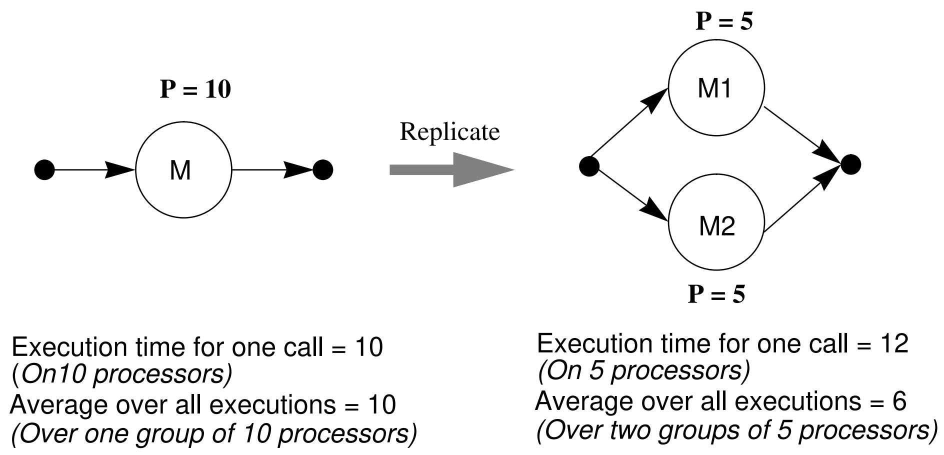

Figure 6: Replication of modules

code, that is non-scalable computation, communication, synchronization, and overheads. In such a scenario, it is difficult to speed up the module by adding more processors, and a greater speedup may be possible by replication.

Other remapping considerations

There are several other changes that can be made to improve the mapping of a task parallel program. These include combining modules to reduce communication cost, and changing the placement of modules onto processors. Also, the choice of communication primitives can be changed by compiler options. We omit the details of the considerations for these changes, but the profiler provides vital information that is useful in making these decisions.

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 7 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 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 8.

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

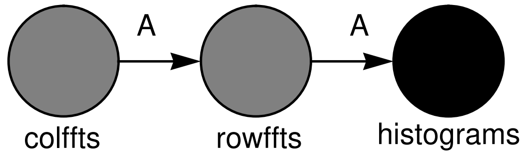

Figure 7: Structure of the FFT-histograms program

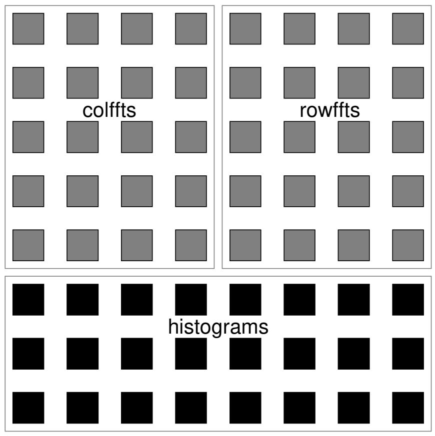

Figure 8: Initial mapping of the FFT-histograms program

The profiler displays for the three distinct groups of processors that execute the three different modules are shown in Figure 9. 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, which is the reason the main routines are seen to have all the communication costs and overheads in the display.

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 it's own computation faster relative to 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 *colffts-module* only does a send, but is running faster then 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

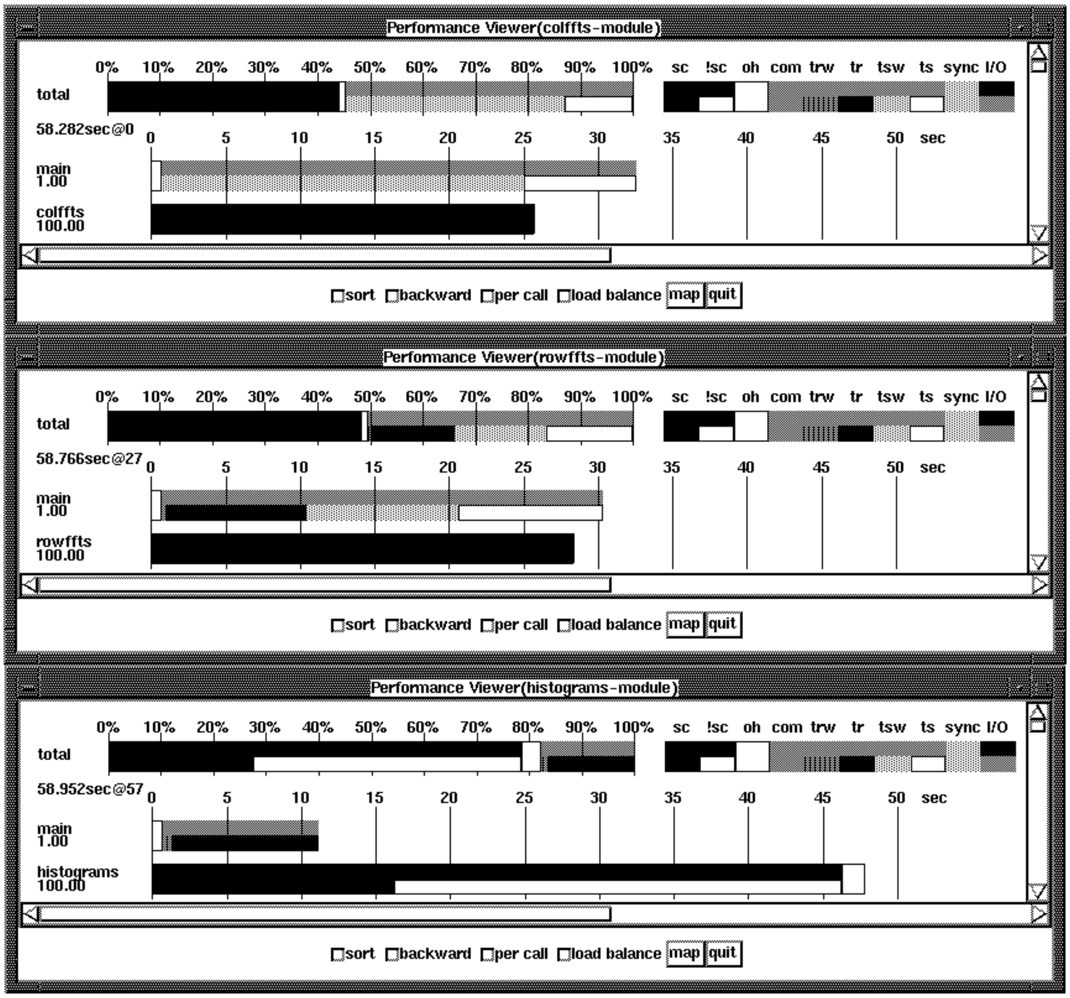

Figure 9: Profiler display for the FFT-histograms program

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. 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 10, and the corresponding profiler display is shown in Figure 11.

The most dramatic change is in the performance of the `histograms` subroutine which speeded up by over 30% due to replication and *histograms-module* is not the bottleneck any more. Moreover, around 2/3rds of the computation in `histograms` subroutine is now scalable, which implies that parallelism is being used more effectively. The `rowffts` subroutine has speeded up somewhat with extra processors (up to 24 from 20), although the *rowffts-module* is relatively the slowest module now. Both *colffts-module* and *histograms-module* have to wait to send and receive to *rowffts-module* (total about 11 seconds and 8 seconds

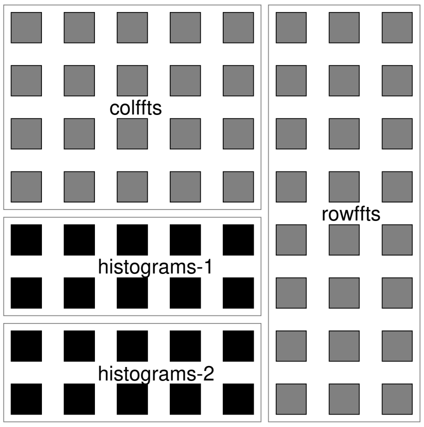

Figure 10: Final mapping of the FFT-histograms program

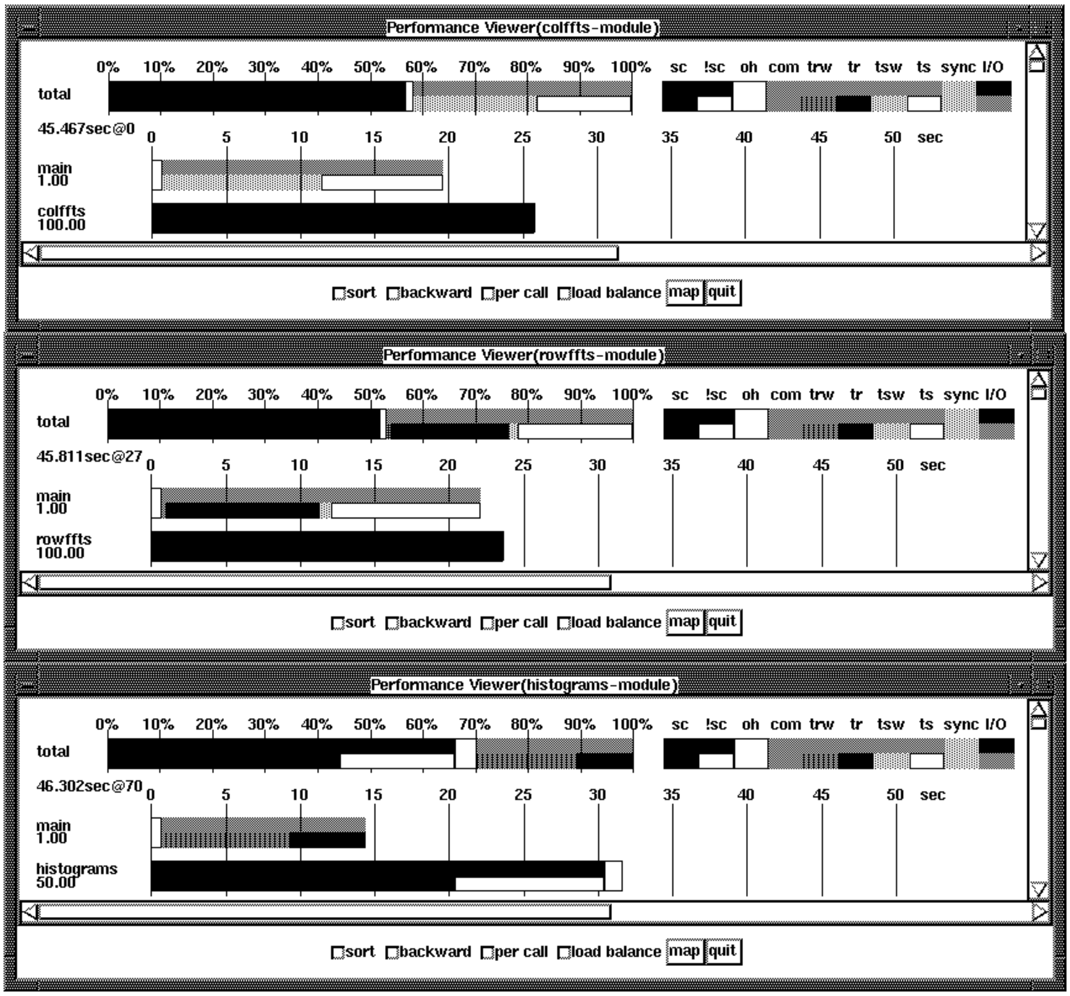

Figure 11: Profiler display for the improved FFT-histograms program

respectively), but the longest waiting time is down sharply (from about 24 to 11 seconds in *colffts-module*), and the total execution time of the program has decreased by around 22% to around 46 seconds. (Further improvement may be possible by more tuning, but we are restricted by the fact that tasks must be assigned a contiguous rectangular subarrays of the processor array) It is important to note that the breakdown of the communication and synchronization overheads into meaningful categories was critical in understanding the performance bugs in this example.

7 Conclusions

We have described a profiler for compiler parallelized programs. The profiler distinguishes ten 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 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. From our experience, the clear classification of the non-scalable code significantly helps programmers understand the bottlenecks and performance limitations of their programs, and provides a rational basis for tuning parallel programs.

References

[BBS89] David Bernstein, Anthony Bolmarcich, and Kimming So. Performance visualization of parallel programs on a shared memory multiprocessor system. *ICPP*, Vol. 2:pp.1–10, August 1989.

[GH93] Aaron J. Goldberg and John L. Hennessy. Mtool: An integrated system for performance debugging shared memory multiprocessor applications. *IEEE trans. on Parallel Distributed syst.*, Vol. 4(1):pp.28–40, January 1993.

[GJJ89] Yogesh Gaur, Vincent A. Guarna Jr., and David Jablonowski. An environment for performance experimentation on multiprocessors. *Supercomputing'89*, pages 589–596, 1989.

[HE91] Michael T. Heath and Jennifer A. Etheridge. Visualizing the performance of parallel programs. *IEEE Software*, Vol. 8(5):pp.29–39, September 1991.

[Hig93] High Performance Fortran Forum. *High Performance Fortran Language Specification*, January 1993. Version 1.0 DRAFT.

[KN91] D. N. Kimelman and T. A. Ngo. The RP3 program visualization environment. *IBM Journal of R&D*, Vol. 35(5–6):pp.635–651, Sept–Nov 1991.

[MCH+90] Barton P. Miller, Morgan Clark, Jeff Hollingsworth, Steven Kierstead, Sek-See Lim, and Timothy Torzewski. IPS-2: The second generation of a parallel program measurement system. *IEEE trans. on Parallel Distributed syst.*, Vol. 1(2):pp.206–217, April 1990.

[Ous94] John K. Ousterhout. *Tcl and the Tk Toolkit*. Addison-Wesley Publishing Company, 1994.

[Par90] ParaSoft Corp. *Express User's Guide*, 1990.

[Rei90] Matthew Reilly. Presentation tools for performance visualization: The M31 instrumentation experience. *proc. 23rd Hawaii intl conf on syst sci*, Vol. 1:pp.307–313, 1990.

[Rov92] Diane T. Rover. A performance visualization paradigm for data parallel computing. *Proc. 25th Hawaii Intl conf on syst sci*, Vol. 2:pp.149–160, 1992.

[SKR92] Sekhar R. Sarukkai, Doug Kimelman, and Larry Rudolph. A methodology for visualizing performance of loosely synchronous programs. *SHPCC*, pages 424–432, April 1992.

[SSOG93] J. Subhlok, J. Stichnoth, D. O'Hallaron, and T. Gross. Exploiting task and data parallelism on a multicomputer. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, May 1993.

[SZ91] F. Szelényi and V. Zecca. Visualizing parallel execution of FORTRAN programs. *IBM Journal of R&D*, Vol. 35(1–2):pp.70–282, Jan–Mar 1991.

[Thi91] Thinking Machines Corp. *Prism User's Guide ver.1.0*, 1991.

Appendix: Profiler Manual

This appendix includes the profiler manual for the iWarp system. Section A gives a brief overview on how to use the profiler. Section B explains the options controlled via the main window of the profiler. The display of the information gathered by the profiler is controlled by this window. Then in Section C we describe the performance viewer window, which is started from the main window; this window presents you results in bar charts. In Section D, we discuss further techniques to exploit this tool. Finally in Section E, we point out some notices for the profiler. These may help users to better understand the profiler's operation.

A Usage summary

You can use the profiler by following these three steps (we assume a basic understanding of invoking the Fx compiler and running programs on iWarp):

1. Run `/iwarp/member/fx/bin/iwprofile` *<F77-files...>*
   The `iwprofile` reads *<F77-files>* generated by Fx-translator (`fx_omega` translator) and embeds profiling calls in them. Therefore you must run `iwprofile` after the `fx_omega` translator but before the F77 compiler. `iwprofile` requires all the F77 source file names as its arguments. If the source file is `foo.fx`, you can obtain the relocatable file as follows.
   ```
   /iwarp/member/fx/bin/fx_omega -dp -dmsg foo.fx
   /iwarp/member/fx/bin/iwprofile .foo.f
   iwf77 -w -c .foo.f
   ```
   If you use the *fx shell script*(`/iwarp/member/fx/bin/fx`), you do not need to worry about invoking `iwprofile`. In this case, just add the `-p` option to your `makefile`:
   ```
   FXFLAGS = -dmsg -p
   ```
   Do not pass the `-p` option to either `fx.omega` or `iwf77`.

2. Run the program as usual
   At the end of the execution of your program, the profiler saves the profile data in the current (working) directory. Saving the data may take about 10 seconds. Please be patient. You can see how the results are saved by watching `iwSpy`. Before saving the results, the red (leftmost) LED is turned on, and after saving the results, it is turned off. The profile data are saved in files with names `.iwprof0000`, `.iwprof0001`,..., `.iwprof0063`. The suffix numbers correspond to the physical processor identifier.

3. Run /iwarp/member/fx/bin/iwpv [<*prof_file*>]
   The iwpv is a performance viewer corresponding to the prof command in Unix. After you invoke iwpv, a new window appears. The features and control of this window is explained in the next section. <*prof_file*> is an optional argument (see below).

   If invoked without arguments, iwpv uses the defaults data files generated in the previous step, i.e. the files .iwprof0000, .iwprof0001,..., .iwprof0063 in the current working directory. At the first run of iwpv, it merges the 64 data files and one extra system file into one file named .iwprof.all. In future runs (but before the next execution of the module on iWarp), .iwprof.all is the default file. You can rename this file to <*prof_file*> and invoke iwpv with the renamed file name. This is useful if you want to compare the results of various versions of your programs.

B Main window

The profiler provides three ways to show the results. In every case, you must select categories(B.2), routines (B.3) and processors (B.4) and click on graph button in the main window (Figure 2). The graph shows the relevant results only for selected processors, routines and categories [2].

- Average of all the selected processors
  This shows average time of all the selected processors for all the selected routines [3]. This is the most frequent usage of this tool(Figure 3).
  1. Select average-of-all-the-selected-processors button
  2. Choose categories
  3. Choose routine(s)
  4. Choose processors
  5. Click on graph button

  Notice: You can get the results in tabular form, if you click on table button instead of graph button.

[2]The profiler shows a graph based on the collected data $T(p, r, c)$ and $N(p, r)$.

- $T(p, r, c)$ is the time spent in category c of routine r on processor p. The time means directly elapsed time in a routine. For example, if main calls foo, the time spent in foo is not included in the time of main.
- $N(p, r)$ is the number of times routine r is called in processor p.

[3]This shows $\mid R \mid$ bars. Each bar represents the average time of all the selected processors. The c-th element of the r-th bar is $\frac{1}{|P|}\sum_{p\in P} T(p, r, c)$, where P, R and C are the selected set of processors, routines and categories, respectively.

- Each processor

  This shows time spent on each of the selected processors. It depicts the aggregate time for each processor, when you choose more than one routine [4]. This graph clearly depicts the difference among processors(Figure 4).

  1. Deselect `average-of-all-the-selected-processors` button
  2. Choose categories
  3. Choose routine(s)
  4. Choose processors
  5. Click on `graph` button

  Notice: You can get the results in tabular form, if you click on `table` button instead of `graph` button.

- Load balance

  This shows load balance among the selected processors. The load balance is better, when the bars are shorter [5].

  1. Choose categories
  2. Choose routine(s)
  3. Choose processors
  4. Click on `graph` button
  5. Click on `load-balance` button in the viewer window

  Notice: Selection of `average-of-all-the-selected-processors` button does not matter.

B.1 Display and quit

The simplest way to use this tool is to just press `graph` button (use the leftmost button of your mouse), and a bar chart summary of the execution profile of the program is shown. Here we discuss the buttons for displaying and quitting.

graph: Show the results graphically. A new window pops out, when you click on this button. The new window is explained in the next section.

table(sort): Show the results in a table directed to the standard output. The results are sorted by the total execution time. Usually a sorted table is easier to look at than an unsorted table.

table(unsort): Show the results in a table directed to the standard output. The results are not sorted by the time. An unsorted format is useful in the following occasion: If you want to see the results of all the

[4] This shows $\mid P \mid$ bars. Each bar represents the time spent on the corresponding selected processor. The c-th element of the p-th bar is $\sum_{r\in R} T(p,r,c)$.

[5] This shows $\mid R \mid$ bars. Each bar shows the load balance of a routine. The load balance is expressed by the value L, which is defined as $L = \frac{\max_{p}(\tau(p))}{\text{ave}_{p}(\tau(p))} - 1.0$, where $\tau(p) = \sum_{r\in R}\sum_{c\in C} T(p,r,c)$ The value L is 0 if the program is completely load balanced. The value increases as the load balancing becomes poorer.

processors on a specific routine (either subroutine or function), you want to get the results shown in the order of processor number rather than in the order of length of spent time by each processor.

quit: Quit this tool.

max(ms): Specify the max time to be shown on the display. When you click on `graph`, the profiler by default shows the results so that the longest bar just fits into the window. This means that the profiler displays various pictures scaled according to the results. This is neat for one picture, but is not convenient when you want to compare several graphs at one glance because they are usually scaled differently. In such a case, type the maximum time in milli-second in this slot (the time is considered to be in second if the numbers are followed by an `s`, ex. `50s`), then the profiler uses this value as the maximum time. It ignores the user's specification when the real maximum time exceeds the user specified time. This option is meaningless for a tabular format.

B.2 Categories

You can choose the categories that you want to see, though the profiler shows all the categories by default. You can select or deselect categories by clicking on the 10 toggle switches corresponding to the categories discussed below. The times for deselected categories are not further considered.

`scale`	User's scalable code execution time
`!scale`	User's non-scalable code execution time
`overhead`	Parallelization overhead
`comm`	Communication time
`t RW`	Receive-waiting time for task parallelism
`t Recv`	Receiving time for task parallelism
`t SW`	Send-waiting time for task parallelism
`t Send`	Sending time for task parallelism
`sync`	Synchronization time
`I/O`	I/O operation time

B.3 Routines

You can specify the routines that you want to use in the `routine/file` slot employing one of the following ways.

- Specify nothing
  This means that you choose all the routines. This is the default selection.

- Specify one routine name
  You can write the routine name directly in the slot when there is only one routine of interest.

- Specify the file which contains the routine names

  You must specify routines by including the names in a file if you want to choose more than one routine. The file contains the names of the routines in the order in which they should be displayed. You can pick any file name, type a list of routines in the file, and type the name in the slot. The file must contain at least one routine name like this.

  ```
  %cat sbr
  foo
  bar
  main
  ```

B.4 Processors

By default, you see the results of all the 64 processors. You can also select a subset of processors, and then the profile information is shown only for those processors. The selected processors are indicated by black squares around the processor number in the master display, and other processors are displayed as white squares. We explain how to select/deselect processors below.

all: Select all the processors.

none: Deselect all the processors.

xor: Reverse the states of selection of all the processors.

R-add: Select processors in a rectangular region. You have to click on two diagonal corner processors right after clicking on this button.

R-del: Deselect processors in a rectangular region. You have to click on two diagonal corner processors right after clicking on this button.

processor group: Select one of processor groups predefined in a file. You can select a particular processor group by pressing the corresponding named button in the processor group list. To use this, create a file named `cgroup` or `.cgroup` in the current working directory. In this file each line contains a group name and a *proc-list*, which is specifid according to the following syntax:

proc-list ::	*<nothing>* | *items*
items ::	*item* | *item items*
item ::	*num* | *num-num*
num ::	*<octal processor number>*

Only a space (not comma, TAB or semicolon) can be a delimiter. Here are some examples.

legal :	3	
legal :	03	
illegal :	08	$\leftarrow$ must be an octal number
illegal :	177	$\leftarrow$ no such processor (00-77)
legal :	03-67	$\leftarrow$ 03,04,...06,13,14,...17,...60,61,...67
legal :	67-03	$\leftarrow$ same as 03-67
legal :	63-07	$\leftarrow$ same as 03-67
illegal :	63 - 07	$\leftarrow$ no redundant spaces
legal :	5 19 4 35 2	
illegal :	5,19,4,35,2	$\leftarrow$ comma is not allowed
legal :	19 66-77 2-45	

For example, if you want to define processor groups as follows:

group dog : 0, 1, 2, 3, and 55

group cat : processor inside of rectangular (11-44) and 77

the `cgroup` file should be as follows:

```
%cat cgroup
dog 0 1 2 3 55
cat 11-44 77
```

Note: When a program contains task parallelism, the Fx Fortran compiler generates a `.cgroup` file that specifies processor groups used in your program. The name of the processor group is shown in the window name of the performance viewer.

processor number (00-77): Directly click on the number and reverse the state of selection of the processor.

C Performance viewer

A window pops up when you press the `graph` button in the main window. It shows a graph as you specified in the main window. You can open multiple windows – as many as you want and compare the statistics. This window supports scroll bars for both the horizontal and vertical directions. You can resize the window as you like.

The top left bar shows the percentage of the time spent by each category for the average of all the selected processors and categories (Figure 3). The top right charts shows the legend of 10 categories. The value in the form of *time@number* on top left shows the pair of the longest time and its processor number(in octal format) [6]. The graph shows several bars. It shows a routine name and the average time that the routine was called on the left of a bar, if you select `average-of-all-the-selected-processors`. The width

[6] This pair is exactly $\tau(p)@p$, while $\tau(p) = \max_{q \in P} \tau(q)$ and τ is defined in the footnote of section B.

of the bars is self adjustable. This means that these bars become narrower when the figure has to include more bars (Notice the difference in width between Figure 3 and 4).

This window has several buttons on the bottom. Selecting them, a user can get a more appropriate graph.

sort: Sort the bars according to the total amount of consumed time.

backward: Specify the direction of the sort; forward or backward. This does not make sense unless you select `sort`.

per call: Show the average time of each invocation of a routine, though the graph shows the aggregate time of multiple calling by default [7].

load balance: Show clearly whether the load balance is good or poor. If the program is completely load balanced, the value is 0. The value becomes greater as the load balancing becomes poorer (see the footnote in section B for details).

map: Show the selected processors of the main window for the current performance viewer window. This is helpful when you are not sure which processors are selected on the current window when you open multiple performance viewers.

quit: Quit this window. The main window is *not* closed by this operation.

D Advanced usage

D.1 Dividing a routine

The profiling tool shows the results for each routine by default. Sometimes, you may want to obtain detailed information about a routine, e.g., when the routine is long or complicated. You can measure the time of sections inside of a routine, using by a `fx_prof('`*label*`')` call, which specifies the beginning of the section named *label*.

In the following example *statement 0, 1* and *2* are grouped in the section labeled `sample`, `foo` and `bar` respectively. Note that the time spent for *statment 1* and *2* is not included in the routine `sample` any more. The labeled section is handled in the same way as a routine.

```
subroutine sample()
   statement_0
   call fx_prof('foo')
   statement_1
   call fx_prof('bar')
```

[7]When you choose `per call`, the figure shows the time as $T'(p,s,c)$ instead of $T(p,s,c)$, where $T'(p,s,c)$ is $\frac{T(p,s,c)}{N(p,s)}$ if $N(p,s) \neq 0$, otherwise 0.

statement_2

```
end
```

You can use an already defined label as a routine name. For example, you can write `call fx_prof('foo')` as many times as you wish and replace `call fx_prof('bar')` with `call fx_prof('sample')` in the above example.

The call of `fx_prof` is executed at run time. Therefore, in the following example, only the statement `a(2)=2` is measured for `foo`, instead of all the statement `a(i)=i`. The time spent for the statement `a(i)=i`($i \geq 3$) is included in `bar`.

```
call fx_prof('foo')
do i = 2, 5
    a(i) = i
    call fx_prof('bar')
    b(i) = i
enddo
```

The profiler regards `fx_prof('`*label*`')` as a call to routine *label*. For the above example, the performance viewer shows that `bar` is called 4 times.

If you do not want the time to be accumulated for each `fx_prof('`*label*`')` call, you should use `fx_prof('`*label*`', 0)`. The profiler suppresses the accumulation if the `fx_prof` is called following with argument `0`. This is useful if you want to accumulate the cost of executing a routine (which is called from more than one place) at the caller site.

D.2 Including time spent in callee's routine

By default, the profiler does not include time spent in the callee. You may want to know the total time spent in a section including the time spend in callees. For example, a routine is called from two different places on the code and you want to know which of the two calls takes a longer time to execute. If you call `fx_prof` with a label preceding an asterisk, i.e. like

$$\texttt{fx\_prof}('{*label}')$$

then from that point on all the time is accumulated for the labeled section. This mechanism is called *inheritance* because the callee routines *inherit* the label (name of the routine) of the caller. In the following example, the time spent in both subroutines `routine_a` and `routine_b` is included in the time for `foo`.

```
call fx_prof('*foo')
call routine_a()
call fx_prof('bar')
  . . .
```

```
subroutine routine_a()
call routine_b()
  . . .
end
```

You can use `fx_prof('*label', 0)` when you do not want to increment the time of calling.

If you want to obtain the time including all the callees' time, you can use

```
subroutine foo()
call fx_prof('*')
any statements
end
```

The second line of `call fx_prof('*')` serves as `call fx_prof('*foo', 0)`. They are not the same when subroutine `foo` is called under *inheritance*. In this case, the second line of the above example is just ignored.

Let us explain the effect of using `fx_prof` call when the *inheritance* mechanism is already used. The new `fx_prof` call temporally overtakes the *inheritance*. In the following example, the time spent in *statement A* and *B* are included the section labeled `foo` and `bar`, respectively.

```
call fx_prof('*foo')
call sbrt0()
statement A
. . .
subroutine sbrt0()
call fx_prof('bar')
call sbrt1()
end
subroutine sbrt1()
statement B
end
```

The sixth line of `call fx_prof('bar')` is the same as `call fx_prof('*bar')`. As we explained before, the asterisk is ignored because you cannot use *inheritance* mechanism in a nest.

E Notices

E.1 Scalable code

The profiler identifies two kinds of code as scalable ones. The first kind includes element-wise operations on whole arrays and on sections of arrays. Fx Fortran has the same syntax as Fortran 90 to specify these operations. For example, the following code segments depict scalable code if the variables `a`, `b` and `c` are distributed arrays.

```
a = b + c
a(:,j) = b(i,:)
a(:, 2:n) = b(:, 1:n-1)
a(:, 1:n:2) = b(:, 2:n:2)
```

The other kind includes parallel loops explicitly defined by a `PDO` statement. The user's code inside of a `PDO` statement is considered as scalable code. The following is an example of such a `PDO` statement.

```
pdo i=1,m
  pin a(i,:),c(i,:)
  pout c(i,:)
  do j = (m/chunks)*jp+1, (m/chunks)*(jp+1)
      do k=1,n
          c(i,j) = c(i,j)+a(i,k)*v(k,j-(m/chunks)*jp)
      enddo
  enddo
endpdo
```

A routine can be called from both inside and outside of a `PDO` statement. In the following example, function `foo(0)` is called outside of a `PDO` statement and function `foo(i)` is called inside of a `PDO` statement.

```
m = foo(0)
pdo i=1,m
  pout a(i)
  a(i) = foo(i)
endpdo
```

In this case, the profiler recognizes the first call of `foo(0)` as non-scalable and the second call of `foo(i)` as scalable.

E.2 Fx-prefix rule

Do not use as names for variables and routines strings that begin with `fx`. Statements that include them are classified as parallelization overhead instead of user code because the profiler assumes that only and all compiler internals begin with `fx`.

It is better not to use names starting with `fx` apart from the profiler because user defined names that begin with `fx` may collide with names used by the fx compiler for compiler internals.

E.3 I/O operations

It is wise to insert `fx_sync()` after I/O operations to avoid nondeterministic execution. For example, suppose that the main routine calls subroutines `foo` and `bar` successively. Each processor finishes subroutine `foo` at a completely different time because subroutine `foo` manipulates I/O operations. Suppose that subroutine `bar` performs a barrier synchronization before communication. The time for synchronization varies wildly among processors, and the results of a performance profile for `foo` and `bar` may be different for every run. Therefore it is recommended that the synchronization should be performed after I/O operations.

```
subroutine foo()
   Each processor reads data from a file on the host machine
   call fx_sync()

subroutine bar()
   synchornization inserted by Fx-compiler for the deposit model
   inter-processor communication
```

Inserting a synchronization step after I/O operations isolates the next routine from the time variations of input and output.

E.4 Deposit model communication

It is highly recommeded that you use the deposit model communication because it is usually faster than the usual message passing. There is one more reason motivated by the profiler. The usual message passing does not perform synchronization before it sends messages, but the deposit model communication does so. Therefore the usual message passing introduces variability into the measurements in a manner similar to I/O operations.

E.5 tstatflag

When you want to distinguish between the waiting time for send/receive and the actual transmission/extraction time for a task-parallel program, you must compile your source code with `tstatflag` (this means to pass `-tstats` to `fx.omega`). Otherwise the profiler merges the waiting time and the actual send/receive time into one sum, which is displayed as `t_RW`/`t_SW`.

E.6 Tool chain

The following is the exact tool chain sequence for this profiler:

user's source code
    ↓   **fx.omega** (Fx-source code to F77-source code translator)
f77 code
    ↓   **iwprofile** (profiling code embedder)
f77 code
    ↓   **iwf77** (Fortran 77 compiler)
executable code
    ↓   *execution*
profile files ← one file for each processor saved on the host computer
    ↓   **iwpv** (performance profile viewer)
display

E.7 Color terminal

The viewer supports color terminals also. It determines if the terminal is color one or not at the run time and shows the results in color if possible.

E.8 Error messages

This robust performance viewer prints out error messages and warnings on the window from which you invoked `iwpv`, e.g. when it encounters an unexpected situation. It is recommended that you look at the original window when the profiler seems to take much longer time than expected.