University of Alberta

Library Release Form

Name of Author: Ernesto Novillo

Title of Thesis: On-line Performance Monitoring of Shared Memory Parallel Programs Using Barriers

Degree: Master of Science

Year this Degree Granted: 2002

Ernesto Novillo
9652 81 Avenue
Edmonton, AB
Canada, T6C 0X7

Date: _____

University of Alberta

On-line Performance Monitoring of Shared Memory Parallel Programs Using Barriers

by

Ernesto Novillo

A thesis submitted to the Faculty of Graduate Studies and Research in partial fulfillment of the requirements for the degree of **Master of Science**.

Department of Computing Science

Edmonton, Alberta
Spring 2002

University of Alberta

Faculty of Graduate Studies and Research

The undersigned certify that they have read, and recommend to the Faculty of Graduate Studies and Research for acceptance, a thesis entitled **On-line Performance Monitoring of Shared Memory Parallel Programs Using Barriers** submitted by Ernesto Novillo in partial fulfillment of the requirements for the degree of **Master of Science**.

Paul Lu
Supervisor

Jonathan Schaeffer

Bruce Cockburn

Date: _____

To Nati,
without a doubt

Abstract

We introduce the SBT library, an on-line debugging and performance monitoring tool for shared-memory parallel programs using the single-program-multiple-data (SPMD) model of parallelism. SPMD programs often use barriers to synchronize threads of execution and to delimit the start and end of different phases of computation. Through its useful barrier constructs, dynamic performance warnings, and integration with hardware event counter libraries, SBT helps programmers localize deadlocks and performance bottlenecks in their parallel programs.

SBT is a portable library that currently supports POSIX threads and SGI Irix `sproc` threads. SBT also supports PAPI and Irix `libperfex` hardware event counter libraries. For production runs, the SBT overheads can be eliminated using conditional compilation.

In order to demonstrate SBT's applicability and usefulness, we present a performance analysis of some of the programs in the SPLASH-2 suite. The information produced by SBT is used to find bottlenecks, identify the most computationally-intensive phases, and generate graphs and tables to facilitate interpretation. In addition, we quantify the overhead incurred by the programs when they are monitored with SBT, and conclude that the cost of the instrumentation is negligible.

Acknowledgements

All through my childhood, and well into my youth, my parents have given me tools. Tools to lose myself in Jules Verne's imagination, tools to understand and appreciate cultural differences, tools to learn, assimilate, and have educated opinions. Now I know that those tools will help me fulfill my dreams. Ma, Pa, thank you. Your tools make me happy. Truly happy. This thesis was developed and written using your tools and the happiness they provide.

My brothers, Diego and Enrique, and my sister Monica, are also to be thanked. We grew up and learned together in Argentina, Spain, and Colombia. Although life keeps us scattered around in three different countries, we are still growing up and learning together. They too have given me tools.

Nati, my wife, makes life perfect. Thank you.

Paul Lu provided many of the ideas and all the guidance that made this thesis possible. Thank you for successfully leading my research through paths that have been satisfying both professionally and personally. Also, thank you for your friendship, for listening, and for understanding me through more surreal experiences.

Baldy and Sparky, the talking manpages, are an infinite source of technical documentation about anything that can happen in the Software Systems lab. My research would not have been possible without their help. Thank you.

The Department of Computing Science at the University of Alberta provided much of the necessary funding to put me through graduate school. I am for ever thankful.

Finally, a big thank you to this great country, one of the few to remain sane through these times of violence and social unrest. Thank you Canada.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Parallel programming is generally accepted to be more difficult than sequential programming. Issues such as synchronization, shared data access, and deadlock must be appropriately addressed to achieve correctness. As well, identifying the performance bottlenecks of a parallel program is more complicated than for a sequential program; it is not just a matter of which function consumes most of the execution time, but also the load balance or imbalance between processors.

Programmers usually rely on parallel debuggers and other tools to obtain information about their programs [7, 8]. Using this information, programmers can correct bugs or optimize performance. However, there are disadvantages to using these debuggers and tools that programmers either learn to live with or try to avoid, sacrificing time or potential optimizations of their code.

Debuggers can give a good snapshot view of an executing program, but they do not always give an adequate trace of the sequence of events leading to an error; they concentrate on showing the current state of the program's data structures. Furthermore, there are few widely-available (e.g., open source) cross-platform parallel debuggers with advanced features. Also, some parallel computers are batch-scheduled (i.e., job queues), so the interactive use of debuggers becomes impractical or impossible. The infamous `printf()` continues to be popular because it is portable, it provides a transcript of the run-up to an error, and it works for both interactive and batch jobs.

There are performance debugging tools as well (e.g., [17, 25]). However, they are often based on detailed and voluminous trace generation and post-processing, or they can have high runtime overheads. Sometimes, even for moderately-sized runs, the size of the traces becomes too unwieldy. For a high-level view of the performance bottlenecks of a problem, off-line performance debuggers can be too inconvenient to use for many programmers. Even worse, a programmer may not suspect that a particular phase of the computation is a bottleneck and therefore will not investigate further. Of course, for detailed performance debugging, after the bottleneck has been localized to a specific phase, these tools continue

to be effective and valuable.

Ideally, the programmer should be able to quickly determine where a program is running into errors and dynamically be informed about potential performance problems. Through the use of a lightweight, on-line performance monitoring tool, the programmer should have access to high-level information about the on-going execution. It is desirable to avoid the extra step of starting up a debugger, starting a trace, or post-processing large amounts of trace data off-line.

1.1 Design Goals and Features

We have developed the SBT library [18] to support some simple but useful on-line debugging and performance monitoring tasks. For programs written in the single-program-multiple-data (SPMD) style, SBT can be used to locate where programs are hanging, where they are encountering performance bottlenecks, and to gain insight as to why there are bottlenecks.

Some of the goals that shape the design of SBT are:

1. *On-line monitoring.* All information produced by the tool should be gathered, processed, and output at runtime. The goal is to keep the programmer informed as the program executes.

2. *Low probe effect.* The cost of the inserted instrumentation, also known as the probe effect [14], should be kept at a minimum. Though it sacrifices detail, a well-implemented, small-footprint instrumentation should be good enough to provide high-level information about the execution. Also, users should have the ability to select different amounts of overhead for the instrumentation, including no overhead (i.e., no instrumentation).

3. *Ease of use.* Once the instrumentation is inserted, the tool should provide information without requiring long sequences of commands or mouse clicks. Also, the instrumentation should be easily removed when it is time to deploy the program for production runs.

4. *Portability.* The tool should not rely on any operating system- or hardware-specific features to extract data and produce information. Additionally, it should support the most popular threading models.

Typical SPMD programs are comprised of phases of execution delimited by barriers. All processes execute the same phase at the same time; once they have all completed the phase, they synchronize at a barrier. No process can proceed beyond a barrier until all processes have reached it. Consequently, barriers are natural places to insert lightweight performance information collection code.

Barriers can be easily converted from synchronization-only points to synchronization-and-instrumentation points. Code that gathers, processes, and outputs performance information can be easily added before the synchronization code. Thus barriers become watchpoints in which information relative to the previous phase is shown to the user before the next phase starts. This information can help programmers to better understand how the different processes behave during each phase at runtime.

Well-known parallel programming systems, including threading libraries (e.g., `sproc` [5]), message-passing libraries (e.g., MPI [21]), and compiler extensions (e.g., OpenMP [6]) include support for barriers.

1.2 Benefits and Features

SBT takes advantage of barriers to produce debugging and performance monitoring information. It provides a barrier construct capable of dynamically gathering performance data that is shown to the user at runtime.

For production runs, SBT code can be removed via conditional compilation. Underneath the information-gathering code lies a call to a plain barrier implementation. Only the underlying barrier code is compiled when the library is built for benchmarking; SBT is thus converted from a debugging and performance monitoring library to a simple invocation of a barrier.

SBT is designed to help answer the following common questions asked by parallel programmers:

1. Where is my program currently executing? If it is deadlocked, where is it deadlocked?
2. What is the most computationally-intensive phase of my program? Where are the bottlenecks?
3. Is there a load imbalance among threads within a phase? Why is there a load imbalance?

By looking at the output from SBT, the programmer is able to determine which phase is currently being executed. After the program passes a barrier, the library outputs information about the phase that has just ended, and about the barrier itself. The simple fact that the programmer can see output for a particular barrier implies the absence of deadlock in the previous phase. On the other hand, upon failing to see output for a particular barrier after a period of program inactivity, the programmer can detect a deadlock in the previous phase. Once a deadlock is associated with a phase, additional barriers can be used to narrow down the part of the code where the program deadlocks.

SBT also produces the information necessary to identify computationally-intensive phases, not only in terms of wall-clock time, but also in terms of hardware events. After locating the most computationally-intensive phase, the programmer can concentrate on the reasons for the bottlenecks in the phase and devise faster alternatives. The observation that 80% of a program's execution occurs in only 20% of its code has little value if we are not able to identify the appropriate 20%.

Parallel programmers constantly search for ways to distribute work loads evenly across processes. Although it is not always possible to achieve perfectly balanced loads, programmers can find room for improvement if they have the right information. SBT uncovers the existence of a load imbalance by computing the amount of time each process spends at a barrier. Also, SBT can be used to find the underlying reasons for a load imbalance.

The current version of the SBT library supports programs written in C/C++ with POSIX threads (Pthreads) and SGI Irix's `sproc` threads. SBT can also use one of three libraries to access hardware performance counters: Performance Counter Library (PCL) [1], Performance API (PAPI) [3], and Irix's `libperfex` [4].

1.3 Overview

This thesis introduces the SBT library as an on-line performance monitoring tool for parallel programs written in the SPMD style. Designed and built to exploit the existence of synchronization barriers in such programs, SBT gives its users quick access to valuable debugging and performance information. SBT's usefulness and ease of use are discussed through the use of examples; and the overhead that SBT introduces due to instrumentation cost is quantified and proven to be negligible.

In the next chapter, we review some parallel debuggers and performance monitoring tools that have features similar to those of SBT. Far from discussing all existing tools, we concentrate on a few of them that are either available to us or commonly used in the development community. We continue with two simple examples that drive an overview of SBT's features and user interface. A chapter that discusses details of the implementation of the library follows; we present the structure of the library and address the issues of portability and hardware performance counters. In order to show SBT's applicability, we dedicate one chapter to discussing the instrumentation with SBT of three commonly known programs from the SPLASH-2 suite [23]. We show that instrumenting the programs is easily done, and we analyze the programs' runtime behavior using performance information produced by SBT. The examples are also used to measure the cost of SBT's instrumentation. Finally, we close the thesis with some concluding remarks and ideas for future improvements and features.

Chapter 2

Related Work

In the Chapter, we review some parallel debuggers and performance monitoring tools. Most of them are more sophisticated than SBT and usually render larger amounts of information. However, SBT and the tools reviewed in this Chapter share a similar goal, which is to help developers improve their parallel programs.

Programmers are presented with a large number of tools that aid writing, debugging, testing, and improving parallel programs. The amount and quality of the information these tools provide ranges from small and useful to cumbersome and difficult to interpret. Depending on the kind of data they generate, and on the stage at which they are helpful, the tools can be categorized as parallel debuggers or parallel performance monitors. However, it is not uncommon to find hybrids able to generate useful data throughout the entire development process [9, 17].

Debuggers concentrate on providing information related to a program's correctness. In a parallel environment, they can give indications of race conditions or potential deadlocks. Performance monitors, in contrast, are more concerned with optimization issues such as pointing out bottlenecks or load imbalances; they are generally used to monitor bug-free programs.

Many debuggers and performance monitoring tools provide metrics on a function body, basic block, or line number basis. However, especially in SPMD programs, barriers are more natural delimiters of the program phases; they stand not only as necessary synchronization points but also as conceptual boundaries between the high-level steps of an algorithm.

2.1 SBT's Niche

SBT is a re-implementation and significant extension of the named barriers in the Aurora Distributed Shared Data system [11, 12]. Named barriers were introduced in Aurora to inform users when certain synchronization points are reached at runtime. A call to the `N_BARRIER(name)` macro under Aurora causes the name of the barrier to be printed to

`stdout`, and a call to be made to the corresponding barrier primitive.

Like other performance monitoring tools, SBT addresses a specific class of programs and a subset of all possible correctness and performance problems. In particular, SBT is targeted at SPMD programs, localizing deadlocks, finding bottleneck phases, and identifying load-balancing problems. Of course, as an application library, SBT may be used in combination with debuggers and other performance analysis tools which can potentially address a larger set of problems.

2.2 Parallel Debuggers

In order to help developers, debuggers produce snapshot views of an executing program. Programmers insert breakpoints in their code during the debugging session and are thus able to view the current state of their program. Another debugging aid is a watchpoint, which allows users to view the information contained in specific instances of a data structure or, in the case of a conditional watchpoint, stop the execution when a certain instance has a specified value. SBT barriers are similar to breakpoints and watchpoints in that they help users to quickly identify points of interest in a program.

One example of a parallel debugger is TotalView [7], a commercial tool from Etnus LLC. TotalView is capable of debugging programs written in a number of parallel paradigms: MPI, OpenMP, HPF, and Pthreads. These paradigms are in turn supported under several different platforms (e.g., IBM/AIX, SGI/Irix, HP/HP-UX, Linux), giving TotalView adequate portability.

In a TotalView parallel debugging session, threads can be grouped together and *barrier breakpoints* can be set for the entire group. TotalView's barrier breakpoints act as regular barriers: whenever a thread or process reaches them, it must wait until all the other members of its group have arrived. This allows users to dynamically introduce barriers in the code, inspect the state of each thread's data structures at that point, and then continue stepping through the execution.

Barrier breakpoints in TotalView are comparable to SBT barriers in the sense that both tools take advantage of a synchronization point to present information to the user. However, there are two important points in which barrier breakpoints and SBT barriers differ. The first and most important difference is the kind of information that users can obtain. TotalView's barrier breakpoints are a mechanism used to obtain debugging information, while SBT barriers produce debugging as well as performance information.

Second, barrier breakpoints are an interactive tool. The user sets them, and when the program reaches them, execution of all threads is suspended. This enables the user to inspect all threads' data structures independently. In contrast, SBT barriers do not interrupt the execution: they synchronize the threads, process and output performance information, and

the execution continues.

2.3 Parallel Performance Monitors

Parallel performance monitors generally rely on the instrumentation of code and the generation of large trace files that require post-processing and analysis. Differences among these tools are found not only in the appropriateness (as determined by the problem domain) and amount of information and analysis they produce, but also in the manner in which the instrumentation is inserted. Moreover, the granularity at which performance monitors gather data is heavily dependent on the method of instrumentation.

Gathering information during execution implies that a certain amount of already limited resources have to be diverted from the subject program. As the information-gathering process becomes more complex, less resources will be available for the program, and actual performance may be hindered. This diversion of resources causes a certain amount of overhead, which is known as *probe effect* or *intrusion* of the instrumentation. For example, instrumented code makes use of cache memory and thus increases the amount of cache misses incurred by the monitored program. Users of performance monitoring tools are well aware of this limitation, and they will hastily discard tools that produce excessively complex instrumentations and have a large probe effect.

In this Section, we categorize the reviewed tools according to the way in which they instrument the programs for monitoring. We distinguish three categories:

1. *Program wrappers:* This is the least intrusive method of instrumentation. It allows data to be gathered only at the process level, and it does not require modifications in the source code, re-compilation, or even re-linking. User programs are launched by the tool, which will gather pertinent data at runtime and show a summary of the execution upon program termination.

2. *Source code instrumentation:* This form of instrumentation requires modification and re-compilation of the source code; re-linking to performance libraries may also be required. Typically, users insert calls to data-gathering functions in the points of interest of their code, and information is output dynamically or in the form of a trace file for later analysis. This method is more flexible in that it allows users to obtain performance information about specific parts of their code, even on a statement basis. However, very detailed instrumentations are prone to have a larger probe effect.

3. *Binary instrumentation:* Under this category, instrumentation is inserted directly into a binary, requiring neither modifications in the source nor re-compilation. Monitored programs are run within the tool, and users are able to select specific aspects of the execution for which performance information should be generated.

2.3.1 Program Wrappers

A typical example of a program wrapper is `perfex` [4], which interfaces with the hardware event counters available on MIPS processors running Irix. Upon execution, `perfex` records the initial state of the specified hardware event counters for each participating processor and forks the given program. Once the program terminates, the differences of the initial and final values of the counters are output. Because the MIPS R10000 and R12000 processors only have two hardware performance counter registers and the Irix kernel is capable of counting up to 32 hardware events, `perfex` takes advantage of the *multiplexing* capabilities of the platform and is thus able to render counts for more events per run than there are actual counter registers available.

Multiplexing hardware counters is a commonly used technique to override the limitation of having fewer performance registers than countable events; the registers are time-division-multiplexed among the number of events to be counted. As a result, the reported event counts are less precise [13]. Nevertheless, they are valuable approximations that are useful for directing attention to the hardware events that bound performance.

2.3.2 Source Code Instrumentation

The `libperfex` library from SGI is an API that allows programmers to start, stop, and print out values of hardware counters from programs written in Fortran and C/C++ under Irix. There are two functional differences between the perfex library and the perfex command: the maximum number of countable events per run, and the granularity. While the command is able to multiplex many events in one run, the library can count only two events per run, one in each hardware performance counter register. Also, the perfex command summarizes hardware event counts for the whole run, whereas using the API gives users the freedom to apply a finer level of granularity to the measurements. For example, users may have little interest in knowing the number of cycles spent during program initialization, so counters can be explicitly started only after all data structures have been initialized.

Another tool that supports source code instrumentation is SpeedShop's `libss` library [9], from SGI. Much like adding an SBT barrier, *caliper points* can be inserted in the code, instructing the library to produce phase-specific information in the trace file (however, caliper points do not synchronize processes). Users can then run their programs through the `ssrun` utility, specifying which of the available *experiments* is to be performed. Experiments provide data about a set of metrics that programmers typically ask about their codes: floating point exceptions, memory utilization, hardware events, i/o system calls, calls to MPI routines, CPU time, real time for each function, etc. Depending on the experiment, granularity ranges from the machine instruction to the phase level (e.g., code executed between two caliper points).

The SpeedShop suite is mainly designed to support binary instrumentation. The use of `libss` and its caliper points is actually an "extra feature" of the suite. For this reason, SpeedShop is discussed in more detail in Section 2.3.3.

2.3.3 Binary Instrumentation

After trace files are generated by SpeedShop's `ssrun` utility, two additional SGI tools, `prof` and `cvperf`, can be used to produce reports based on the gathered data. While `prof` shows information as text, `cvperf` is a visual tool capable of generating graphs and allowing more interaction with the user. In both cases, information is summarized either on a per-processor basis or aggregated throughout all processors. However, SpeedShop only supports parallel processes launched with `sproc()`, `fork()`, or MPI in distributed environments.

A small subset of all the information `prof` is able to produce is shown in Figure 2.1. The output is taken from a parallel implementation of radix sort running on 4 processes. Under the *ideal* experiment —the one shown in the Figure— `prof` generates a list of the functions executed at runtime (see the last 10 lines of the Figure, labeled `[1]` through `[10]`). For each function, it shows estimates of the amount of time required, percentage of total time, cumulative percentage of time, number of cycles, number of instructions, and the number of times the function was invoked.

Users can have access to graphs and a more elaborate interface by using `cvperf`. Figure 2.2 shows a screen shot of `cvperf` using the same trace files that generated the `prof` output shown in Figure 2.1. The lower section of the screen, called the *time line area*, shows the occurrence of events at runtime. Specific time-sections of the execution can be analyzed by dragging the *caliper controls* (triangles with black outline in the time line area). In this case, each *stick* in the time line represents a caliper point explicitly set in the source code. The Figure shows that performance analysis is focused in phase 3 of the execution (see text boxes labeled `Begin` and `End`, in the lower section of the Figure). Caliper points 5 and 6, in which the caliper controls are set, mark the beginning and end of phase 3, respectively. This is an example where SpeedShop is used as a mix of source code and binary instrumentation.

The combination of `ssrun` and `prof` or `cvperf` results in a sophisticated tool capable of producing large amounts of performance data. Additionally, debuggers or memory profilers such as Purify can be attached to programs executed through SpeedShop.

The output in Figure 2.1 and the screen shot in Figure 2.2 do not represent all the information the SpeedShop suite is capable of producing. Depending on the kind of experiment used with `ssrun`, `prof` and `cvperf` can provide users with a large number of different metrics.

Another tool in the category of binary instrumentation is Paradyn [15], a project from the University of Wisconsin. This is a flexible tool for measuring performance of parallel

```
------------------------------------------------------------------------
SpeedShop profile listing generated Sat Sep 22 13:10:12 2001
   prof radix.ideal.m233418 radix.ideal.p233384 radix.ideal.p233420 radix.ideal.p233423
                  radix (n32): Target program
                        ideal: Experiment name
                        it:cu: Marching orders
              R10000 / R10010: CPU / FPU
                           46: Number of CPUs
                          195: Clock frequency (MHz.)
...

------------------------------------------------------------------------
Summary of ideal time data (ideal)--
                    952005994: Total number of instructions executed
                   1738101477: Total computed cycles
                        8.913: Total computed execution time (secs.)
                        1.826: Average cycles / instruction
------------------------------------------------------------------------
Function list, in descending order by exclusive ideal time
------------------------------------------------------------------------
 [index]   excl.secs   excl.%     cum.%        cycles  instructions    calls  function  (dso: file, line)

     [1]       5.087    57.1%     57.1%     992006076     432002646 16000098  product_mod_46 (radix: radix.c, 1168)
     [2]       1.893    21.2%     78.3%     369091250     337112373        4  slave_sort (radix: radix.c, 629)
     [3]       1.477    16.6%     94.9%     288000136     128000212        4  init (radix: radix.c, 1309)
     [4]       0.438     4.9%     99.8%      85440929      51266127       72  _barrier (libc.so.1: barrier.c, 86)
     [5]       0.012     0.1%     99.9%       2286828       2114965      711  _posix_spin_lock (libc.so.1: r4k.s, 50)
     [6]       0.001     0.0%     99.9%        270063        299472     2779  resolve_relocations (rld: rld.c, 2636)
     [7]       0.001     0.0%    100.0%        257612        265026     1202  general_find_symbol (rld: rld.c, 2038)
     [8]       0.001     0.0%    100.0%        112732        141652     1216  elfhash (rld: obj.c, 1184)
     [9]       0.000     0.0%    100.0%         70240         69913        1  fix_all_defineds (rld: rld.c, 3419)
    [10]       0.000     0.0%    100.0%         62608         89901     3913  obj_dynsym_got (rld: objfcn.c, 46)
...
```

Figure 2.1: Sample output from `prof`. Only 30 lines shown, out of 500.

programs running on distributed or shared memory architectures. It is able to gather data at different granularities, starting at the statement level and going up to the procedure and process level. Paradyn's most prominent characteristic is its ability to automatically insert and modify instrumentation dynamically (i.e., during program execution); it also controls the cost of the instrumentation automatically, keeping it within a user-defined threshold.

However, Paradyn's automatic instrumentation is also its biggest caveat, for it requires long execution times to allow its dynamic instrumentation mechanism to come into effect. It was purposely developed to automatically find the most influential performance bottlenecks, making it mandatory that the most resource-consuming pieces of code be either complex —and thus time-bound— or executed several times to detect cumulative effects.

As execution progresses, Paradyn probes the program in search of potential performance problems based on two types of instrumentation: hardware counters and timers. Samples of these two are periodically taken at procedure entry, procedure exit, and individual call statements. Once hardware counter and timer data start flowing, Paradyn tries to associate areas of the code that show poor performance with an adequate member of a hierarchy of *hypotheses*. Typical performance problems that occur in parallel programs, such as costly synchronization or resource contention, are organized in a hierarchy of hypotheses that the tool tests at runtime. After a hypothesis is proven to be true, iterative execution of that piece of code can further refine the reason for the bottleneck by associating the slowdown

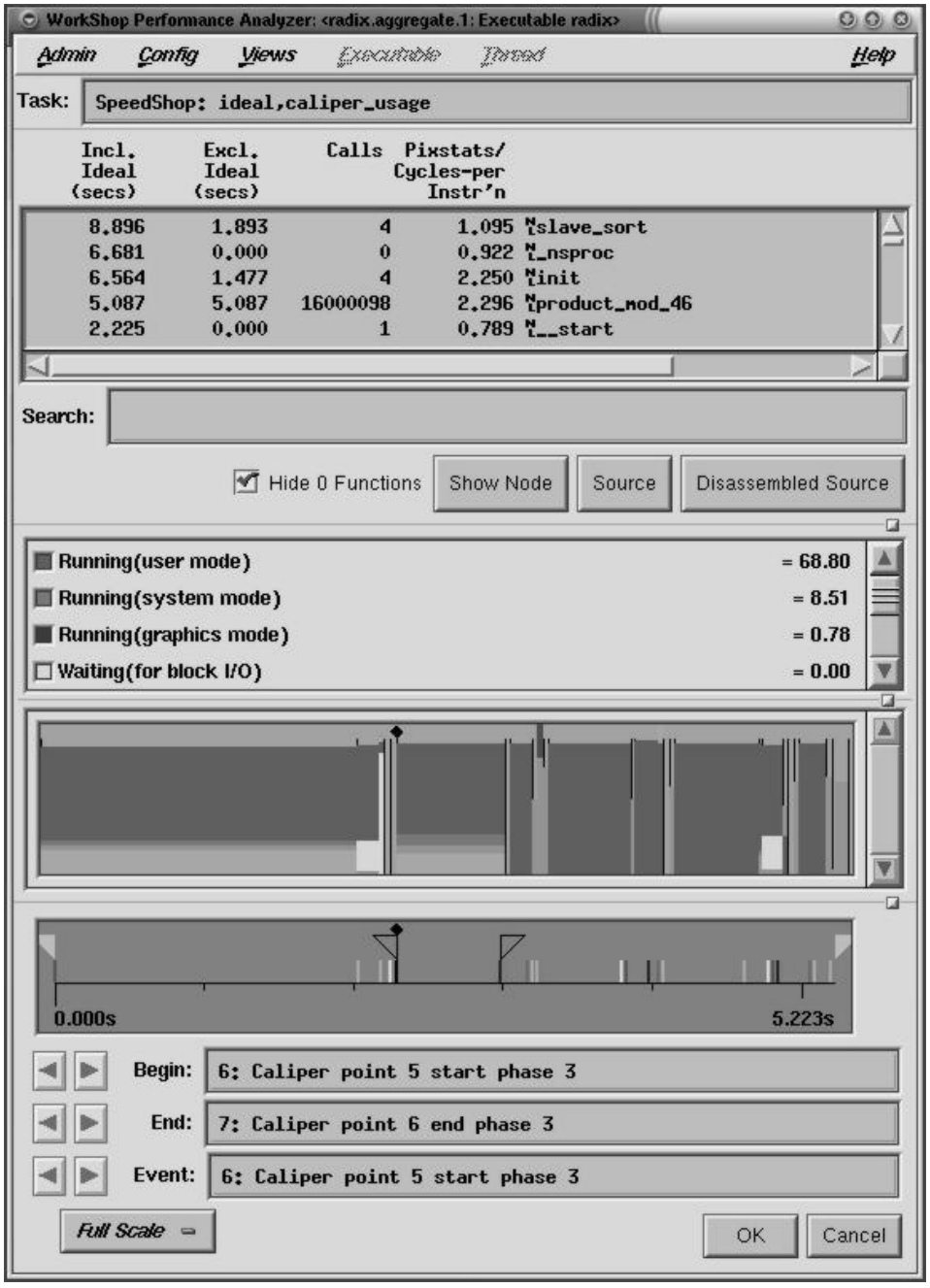


Figure 2.2: `cvperf` screen shot. Caliper points shown in lower section of the screen.

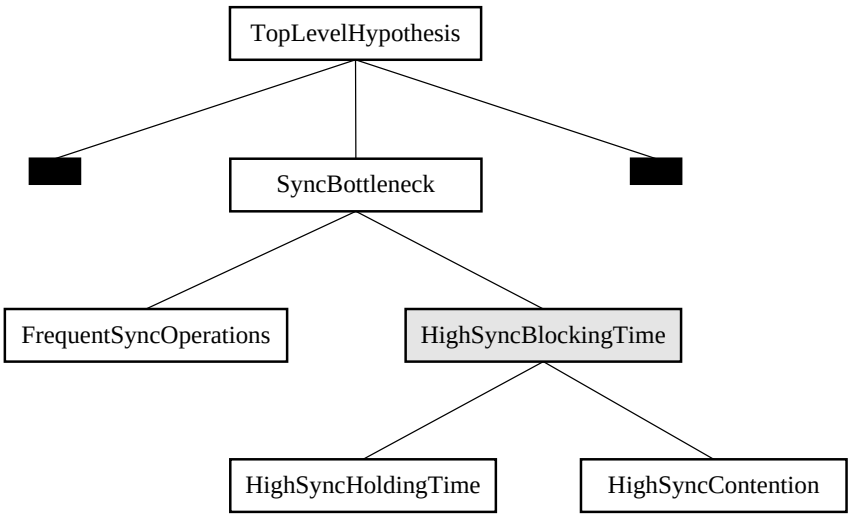


Figure 2.3: Example of Paradyn hypothesis hierarchy [15]. The shaded node represents the hypothesis currently under consideration.

with a hypothesis further down in the hierarchy.

Figure 2.3 shows a subset of the hypothesis hierarchy Paradyn uses to find the reasons for a performance problem. After a program starts, the children of *TopLevelHypothesis* are tested. In this example, hypothesis *SyncBottleneck* tested true, which caused a search for refinement in its children. The shaded node, *HighSyncBlockingTime*, represents the hypothesis currently being tested. If it is found to be true, its children, *HighSyncHoldingTime* and *HighSyncContention*, will be tested. Hypotheses are tested as Boolean evaluations against user-defined or default thresholds.

The process of refining the reason and location in the code of a performance problem takes place automatically at runtime, and allows some interaction with the user through the use of visualization modules. For example, users can specify a set of hypotheses to test, establish rules to prune the hypothesis tree, or direct Paradyn to monitor a specific phase of the execution [22]. Additionally, Paradyn produces time histograms, bar charts, tables, and other sources of information that are useful in the process of optimizing performance.

The last tool we review are the Tuning and Analysis Utilities (TAU) [10], a research project from the University of Oregon. Although it was originally devised as a tool to trace and analyze parallel object-oriented programs written in pC++ [16], it now supports programs written in C, C++, FORTRAN 77/90, HPF, or Java that run on several platforms (SGI, IBM, Intel, Cray, etc.). It also supports many shared and distributed memory threading models that can be accommodated into abstractions of nodes, contexts, and threads, as defined by the High Performance C++ consortium [10].

A modular architectural design adds to TAU's expandability and ability to interface

with other analysis tools. It is implemented in five parts, each with a specific target: program code analysis, performance instrumentation, performance measurement, performance analysis and visualization, and on-line monitoring.

Code instrumented by TAU is able to gather performance data at the application, function, method, and statement levels. After performance data have been gathered, profiling information can be generated and analyzed through the use of various analysis and visualization tools that are part of the TAU environment.

2.4 Concluding Remarks

Although SBT can be used for simple debugging tasks, it is not intended for interactive operation, like most debuggers are. Instead of suspending execution at barriers (as with breakpoints) or monitoring variables (as with watchpoints) SBT tries to be minimally disruptive to the execution and provide useful parallel metrics at runtime. Furthermore, the association of barriers with program phases provides a natural and effective framework for metrics unavailable in debuggers, such as real time spent in a phase and per thread work.

The reviewed performance monitoring tools are able to provide, on one level or another, information similar to what SBT produces; in general, they provide even more information. However, they are not always able to provide quick answers to the questions stated in Section 1.2.

Tools that fall under the category of program wrappers, like `perfex`, cannot inform the user which part of the program is currently executing. They are also unable to produce any kind of information that can lead to the localization of deadlocks. Since they generate information about the whole run, they are unaware of sections of the program, so they do not identify computationally-intensive phases. In certain situations, it is possible for `perfex` to detect load imbalances. Of course, this only applies to load imbalances spanning the whole run.

Source code instrumentation tools are most similar to SBT. The use of SGI's `libperfex` library and some adequately placed calls to `printf()` in the source code can convey phase-specific hardware event counts. Thus, `libperfex` can answer all the questions stated in Section 1.2. However, users are required to add code in order to start and stop the counters, and to output all gathered counts. SBT barriers already include that code, and are able to produce other types of data (e.g., timing information) which `libperfex` does not generate. In a typicial SPMD program, the barriers are required for synchronization. Using SBT barriers does not augment the user's coding effort. Also, `libperfex` can be used only on MIPS processors running SGI's Irix operating system.

The binary instrumentation tools reviewed in this Chapter —SpeedShop, Paradyn, and TAU— are capable of answering the same questions as SBT. They may require the user

to dig through large trace files and menus, but the information can be found. SBT clearly distinguishes the different phases of the program, and it outputs phase-specific data in an easy-to-interpret and direct form. It does not require long executions or a considerable level of expertise in using the tool.

Regardless of the advantages of using SBT, it should be noted that SpeedShop, Paradyn, and TAU are very sophisticated tools, capable of producing more detailed performance data than SBT. After identifying the main bottlenecks of a program with SBT, users can turn to other tools in order to access more detailed information.

Chapter 3

Overview of SBT

We have developed the SBT library in order to provide the user with simple, on-line debugging and performance information. This tool can aid in the development, debugging, and optimization of a parallel program, giving the programmer access to several metrics gathered at runtime. After debugging and performance-tuning the program, SBT's information gathering code can be removed, via conditional compilation, for production runs. Through the use of environment variables or command line options, users can control and focus the monitoring efforts of SBT without recompiling their code.

This Chapter describes the performance information SBT is capable of producing and provides simple examples of use in order to describe the user interface. Appendix B contains a detailed reference of the user interface.

3.1 Features

Parallel programs are often divided into phases that are delimited by barriers. Single-program-multiple-data (SPMD) style programs, in particular, use barriers for synchronization. Figure 3.1 shows a typical SPMD program using SBT barriers for synchronization at the end of each phase. Some parallel programming styles, such as client-server, are not as likely to use barrier synchronization, therefore SBT may not be suitable for those programs.

Normally, barriers are anonymous, like locks and unlocks, but SBT implements the simple concept of a *named barrier* as a way to produce a low-noise trace of the progress of a parallel program. In our experience, programmers often use calls to `printf()` to accomplish the same task. A named barrier (invoked with the `N_BARRIER()` macro) at the beginning of, say, the initialization phase (line 3 in Figure 3.1) produces the output `"Start Initialization"`; a different named barrier at the end of the phase (line 9) produces the output `"End Initialization"`. Therefore, by watching the standard output of the program, the programmer can see where the program is currently executing. If the end-of-phase message —in this case `"End Initialization"`— is not seen, the programmer knows

```
 1    void thread_work()
 2    {
 3        N_BARRIER( "Start Initialization" );      /* Named barrier */
 4
 5        /* Code for initialization phase */
 6
 7        ...
 8
 9        N_BARRIER( "End Initialization" );        /* Named barrier */
10
11        /* Begin computation */
12        ...
13
14        BARRIER;                                  /* Anonymous barrier */
15
16        for( i=0; i<STEPS; i++ )
17        {
18            /* Iterative computation code */
19            ...
20
21            NL_BARRIER( "End iteration" );        /* Loop barrier */
22        }
23    }
```

Figure 3.1: Anonymous, Named, and Loop Barriers in SPMD Programs.

that either the phase of computation is long, or a deadlock has occurred. Also, any user-defined output during the phase is bracketed by the output of the named barriers. Thus, named barriers label and associate output with the corresponding phase of the program.

Anonymous barriers, like the one in line 14 of Figure 3.1, produce output that is easily identified by their source code file name and line number. Named barrier output, on the other hand, is identified by the barrier's name. There are no other differences between named and anonymous barriers; they are all capable of gathering and outputting the same kind of information.

SBT introduces *loop barriers*, which are intended to be used as phase delimiters inside loops. Named and anonymous barriers used in iterative programs that require synchronization inside their loops produce output for every individual iteration. For loop barriers, SBT accumulates information throughout all iterations and outputs the cumulative data at the end of the execution, reducing the amount of noise the user receives from the library. Line 21 of Figure 3.1 is a call to a loop barrier. At the end of the execution, before library resources are freed, SBT outputs cumulative data gathered during the loop and associates it with `"End iteration"`. Loop barriers can be named or anonymous and are invoked with the `NL_BARRIER()` and `L_BARRIER` macros, respectively.

All barriers provide natural caliper and watchpoints for performance monitoring. For example, the program in Figure 3.1 might produce the output depicted in Figure 3.2. Each barrier, seen as a caliper point, informs the user that a certain phase of the execution has

```
Barrier "Start Initialization" reached.
...

-- normal program output --
...

Barrier "End Initialization" reached.
...

-- normal program output --
...

Barrier "End iteration" reached.
...
```

Figure 3.2: Sample output for the program in Figure 3.1. Barriers are natural caliper and watchpoints.

been completed.

After compiling their parallel program to use SBT, users can identify the barrier they wish to watch by setting an environment variable or passing a command line parameter when executing the program. The barrier to watch can be specified by using either its name or its line number. While the program is executing, the following information will be dynamically collected and selectively generated:

1. *Phase time*: The amount of wall-clock time spent between the barrier at the beginning of a phase and the barrier at the end.

   All barriers, either implicitly or explicitly, represent the end of one phase and the start of another phase. In this way, the most computationally-intensive phases are easily identified, since they usually present the longest phase times.

2. *Barrier time*: The amount of time spent by the program at a barrier.

   By definition, barrier time is the time difference between when the first thread arrives at the barrier and when the last thread arrives. Long barrier times suggest that performance is being lost due to idle threads at the barrier. Poor load balancing is a common cause of long barrier times.

3. *Thread inter-arrival time*: The time difference between one thread's arrival and the next thread's arrival.

   The order in which threads arrive at the barrier is also noted. When locating load balancing problems, it can be important to know the order of, and interval between, thread arrivals. A repeated pattern of arrivals in which one thread is always last to arrive provides a hint as to the cause of a load imbalance.

Loop barriers are, by their nature, not capable of producing this metric. Recall that loop barriers accumulate data throughout the iterations of the loop. As a consequence, the order of arrival and the inter-arrival times for each iteration (i.e., each time the loop barrier is reached) are not shown. Instead, they output *thread idle times*, defined as the cumulative amount of time each thread spent waiting at the barrier throughout the loop.

4. *Hardware counter performance metrics*: Depending on the CPU architecture, information about cache misses, graduated instructions, CPU cycles, floating-point operations, etc., can be collected by hardware counters.

   Low-level performance counters can give insight as to what might be the cause of a performance problem. Poor memory locality, poor load balancing, and high synchronization rates (i.e., poor granularity) can be revealed by examining performance counters.

Regardless of whether they are being watched or not, warnings are automatically issued for barriers that are particularly costly (e.g., barrier times longer than a user-selectable threshold). The relevance and frequency of these warnings can also be parametrized by user-controlled environment variables or command line options.

Refer to Appendix A for a complete list of SBT options and their definitions.

3.2 Using SBT

We now consider two illustrative examples of how SBT is used in practice. The matrix multiplication and LU decomposition examples that follow are not tuned for high performance; they are simple examples to demonstrate the features and user interface of SBT.

To illustrate the portability of the library, the examples are run on different platforms and use different threading and hardware counter libraries. Matrix multiplication is built to run on a dual Pentium Linux machine and uses POSIX threads and PAPI for performance counters. The second example, LU decomposition, executes on an SGI Origin 2000 with 48 processors running Irix, and uses `sproc` threads and `libperfex` to access the hardware event counters.

Even though the examples are run on different platforms, SBT provides a similar API for both cases. In fact, there is only one difference in the API for the two platforms: the library initialization function (`sbt_init()`) has different arities.

3.2.1 Matrix Multiplication

The matrix multiplication example uses SBT barriers to synchronize threads at three points in the execution: one before initializing the matrices to be multiplied, one after initializing

them, and the one at the end of the multiplication itself (lines 22, 25, and 28 of Figure 3.3). Work and data are block-distributed between threads. According to this work distribution, each thread is responsible for multiplying a range of contiguous rows of matrix A by matrix B. The result of the multiplication is stored in C.

The implementation for functions `init_block(int, int)` and `matrix_multiply(int, int)`, with headers in lines 14 and 15 of Figure 3.3, is shown in Figure 3.4. In our example, we assume the code in both figures is contained in one single file called `pmm.c`.

We use SBT with Pthreads and the PAPI library for this example. Therefore, the code is linked to 3 libraries: `libpthread`, `libsbt`, and `libpapi`. Figure 3.5 shows the `Makefile` that builds `pmm`, the parallel matrix multiplication program for this example. Line 5 of the figure specifies the three libraries to which `pmm` is linked.

For simplicity, the number of threads to launch and the matrix size are fixed according to the definitions of `THREAD_COUNT` and `SIZE` in lines 8 and 9 of the code, respectively. In general, SBT can support any number of threads and problem sizes.

There are two defines and one include that have to be provided *before* including the SBT header file, `sbt_barrier.h` (lines 1, 2, and 3 of Figure 3.3):

1. `SBT_BARRIER` (line 1): A macro that identifies the specific barrier data structure (of type `sbt_barrier_t`) to be used implicitly for all barrier invocations in the program.

   Structure `sbt_barrier_t` contains a pointer to either an `sproc` barrier or to SBT's implementation of a Pthreads-based barrier, depending on which threading library is specified. Pthreads does not provide a barrier function, so we have implemented one.

   The `sbt_barrier_t` data structure also contains various data fields used for gathering performance statistics.

2. `SBT_THREADID` (line 2): A macro that identifies a function or variable that provides the numerical identity (between 0 and $n-1$ for n threads) of the current thread. The library uses this variable internally, to identify the specific thread for which information is being processed.

   In this example, threads are identified by the value of integer variable `my_id`, defined in line 19 of Figure 3.3.

3. Threading library (line 3): The header file for the appropriate threading library.

   Currently, either `#include <pthread.h>` for Pthreads or `#include <sys/prctl.h>` for `sproc` threads are supported. The `sbt_barrier.h` header file contains definitions that depend on the threading library.

After these preconditions have been satisfied, the library can be initialized with a call to `sbt_init()` (line 38 of Figure 3.3). The initialization function under Pthreads takes as

```
 1  #define SBT_BARRIER     my_barrier
 2  #define SBT_THREADID    my_id
 3  #include <pthread.h>
 4  #include "sbt_barrier.h"
 5  #include <stdlib.h>
 6  #include <stdio.h>
 7
 8  #define THREAD_COUNT    2
 9  #define SIZE            512
10
11  sbt_barrier_t* SBT_BARRIER;
12  int A[SIZE][SIZE], B[SIZE][SIZE], C[SIZE][SIZE];
13
14  void init_block( int, int );
15  void matrix_multiply( int, int );
16
17  void* work( void* id )
18  {
19      int my_id = *(int*)id;
20      int row_count = SIZE/THREAD_COUNT;
21      int first_row = (int)my_id*row_count;
22      BARRIER;
23      /* initialize my block of A and B */
24      init_block( first_row, row_count );
25      N_BARRIER( "End initialization" );
26      /* multiply row_count rows of A by B, starting in first_row */
27      matrix_multiply( first_row, row_count );
28      N_BARRIER( "End multiplication" );
29      return( NULL );
30  }
31
32  int main( int argc, char** argv )
33  {
34      pthread_t threads[THREAD_COUNT];
35      pthread_attr_t attr;
36      int thread_ids[THREAD_COUNT], i;
37      /* initialize SBT library and barrier */
38      SBT_BARRIER = sbt_init( THREAD_COUNT, argc, argv );
39      /* launch threads */
40      pthread_attr_init( &attr );
41      pthread_attr_setdetachstate( &attr, PTHREAD_CREATE_DETACHED );
42      for( i=1; i<THREAD_COUNT; i++ )  {
43          thread_ids[i] = i;
44          if( pthread_create( &(threads[i]), &attr, work,
45                              (void*)&(thread_ids[i]) ) ) {
46              printf( "Error creating thread %d\n", i );
47              return( 1 );
48          }
49      }
50      thread_ids[0] = 0;
51      work( (void*)&(thread_ids[0]) );
52      /* free library resources */
53      sbt_finalize();
54      return( 0 );
55  }
```

Figure 3.3: Matrix multiplication example: SBT barriers for Pthreads.

```
1  void init_block( int first_row, int row_count )
2  {
3      int i, j;
4      srand( SIZE );
5      for( i=0; i<SIZE; i++ )  {
6          for( j=0; j<SIZE; j++ )  {
7              if( i>=first_row && i<first_row+row_count )  {
8                  A[i][j] = (int)(100.0 * rand()/(RAND_MAX + 1.0));
9                  B[i][j] = (int)(100.0 * rand()/(RAND_MAX + 1.0));
10             } else {
11                 100.0 * rand()/(RAND_MAX + 1.0);
12                 100.0 * rand()/(RAND_MAX + 1.0);
13             }
14         }
15     }
16 }
17
18
19 void matrix_multiply( int first_row, int row_count )
20 {
21     int i, j, k, comp=0;
22     for( k=first_row; k<first_row+row_count; k++ )  {
23         for( i=0; i<SIZE; i++ )  {
24             for( j=0; j<SIZE; j++ )  {
25                 comp = comp + A[k][j] * B[j][i];
26             }
27             C[k][i] = comp;
28             comp = 0;
29         }
30     }
31 }
```

Figure 3.4: Matrix multiplication example. Matrix initialization and multiplication functions.

```
1  CC      = cc
2  SBTPATH = /path/to/sbt
3  PAPIPATH= /path/to/papi
4  CFLAGS  = -O2 -I$(SBTPATH) -I$(PAPIPATH) -L$(SBTPATH) -L$(PAPIPATH)
5  LDFLAGS = -lpthread -lsbt -lpapi
6  TARGET  = pmm
7
8  $(TARGET): $(TARGET).c
9          $(CC) $(CFLAGS) $< -o $@ $(LDFLAGS)
```

Figure 3.5: Parallel matrix multiplication example. `Makefile` links the program to `libpthread`, `libsbt`, and `libpapi`.

SBT Option	*Value*
SBT_WATCH	"End multiplication"
SBT_EVENTS	PAPI_TOT_CYC:PAPI_L2_TCM

Table 3.1: SBT options used in matrix multiplication example.

C shell environment variables
$ setenv SBT_WATCH "End multiplication"
$ setenv SBT_EVENTS PAPI_TOT_CYC:PAPI_L2_TCM
$ ./pmm
Command line parameters
$ ./pmm SBT_WATCH "End multiplication" \
SBT_EVENTS PAPI_TOT_CYC:PAPI_L2_TCM

Table 3.2: Alternate methods to set SBT options.

parameters the number of threads that will be synchronized at the barrier, and `argc` and `argv`. SBT allows options to be set either as environment variables before the execution, or as command line options when the program is executed.

As part of library initialization, `sbt_init()` creates a variable of type `sbt_barrier_t` and returns it to the caller. Typically, the return value of `sbt_init()` is assigned to the `SBT_BARRIER` macro.

At the end of the computation, the same thread that initialized the library must call `sbt_finalize()` (line 53 of Figure 3.3). This function not only frees the resources used by the library but also prints any outstanding data (e.g., accumulated hardware counter values and loop barriers data).

In this example, threads use two named barriers: one after initializing their own block of the matrices (i.e., line 25 in Figure 3.3, `"End initialization"`) and another one after their part of the computation is completed (i.e., line 28 in Figure 3.3, `"End multiplication"`). We are interested in the actual multiplication phase of the execution, thus we direct SBT to watch the `"End multiplication"` barrier. Also, we wish to know the total number of cycles and Level 2 cache misses for each thread.

Table 3.1 shows the SBT options that are used, along with their values. To set these options, the user can either pass them as parameters to the command line, or define environment variables before executing the program. The two alternate methods for running the program with the selected SBT options are shown in Table 3.2: the first row indicates the sequence of commands to be executed under the C shell to set the environment variables and then run the program (called `pmm`), and the second row illustrates the invocation of parallel matrix multiplication passing SBT options as command line parameters.

The first of the two options, `SBT_WATCH`, directs SBT to watch the named barrier `"End multiplication"`. Therefore, verbose performance monitoring data is displayed when this

```
 1  SBT options (version 0.9 built for pthreads)
 2          SBT_WATCH       End multiplication
 3          SBT_WATCH_ALL   0
 4          SBT_WARNINGS    1
 5          SBT_WARN_TIME   1000
 6          SBT_PHASE_TIMES 0
 7          SBT_NO_DEBUG    0
 8
 9          PAPI event 0:   PAPI_TOT_CYC
10          PAPI event 1:   PAPI_L2_TCM
11
12  SBT pmm.c:59: "End initialization"
13          (barrier: 0ms, phase 1: 15.651s, from init: 15.651s)
14
15  SBT barrier watch in pmm.c:63 "End multiplication"
16          Barrier time:   2 ms
17          Phase time:     6.078 sec
18          Total time:     21.729 sec
19          Order of arrival:
20                          inter      from               real
21                   id  thread      init               time
22                    0     0ms     21.727s         11:09:44.469
23                    1     2ms     21.729s         11:09:44.471
24
25          SBT: Phase 2 PAPI counter information
26   id    PAPI_TOT_CYC     PAPI_L2_TCM
27    0      4853240157        65759809
28    1      4853799140        65734594
29
30          SBT: Overall accumulated PAPI counter information
31   id    PAPI_TOT_CYC     PAPI_L2_TCM
32    0      6222726397        73965140
33    1      6211535274        73170261
```

Figure 3.6: Example SBT output for matrix multiplication: 512×512 matrices, 2 Pthreads, PAPI, Linux.

barrier is reached. Barrier time, phase time, total time from library initialization (i.e., when `sbt_init()` is called), and thread inter-arrival times are given (lines 15 to 23, Figure 3.6). The `"End initialization"` named barrier produces fewer lines of output (lines 12 and 13) because it is named, but it is not watched. Anonymous barriers, like the one in line 22 of Figure 3.3, do not normally produce any output unless they are watched.

Second, a colon-separated list of PAPI events to be monitored is specified by SBT option `SBT_EVENTS` (Table 3.1). PAPI is a portable interface to hardware counters for events such as Level 2 cache misses and the number of cycles executed by the CPU. The performance counter information is output for every phase that ends at a watched barrier (lines 25 to 28, Figure 3.6). Cumulative totals for all phases are shown at the end of the computation (lines 30 to 33, Figure 3.6) when `sbt_finalize()` is called (line 53, Figure 3.3).

The output in Figure 3.6 corresponds to a multiplication of two integer matrices of size 512×512 with 2 threads under Linux. The first 10 lines of the Figure show the values

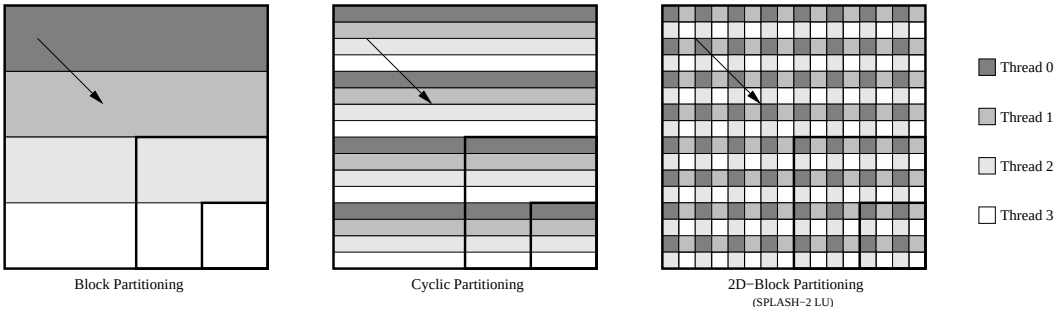


Figure 3.7: LU decomposition work-partitioning strategies.

for SBT options. Note that named barrier `"End initialization"` produced the output in lines 12 and 13, even though it is not watched. Particularly, the information in line 13 tells us that barrier time for `"End initialization"` is 0 milliseconds, phase time for phase 1 is 15.651 seconds, and the time between library initialization (i.e., the call to `sbt_init()`) and the barrier is 15.651 seconds. All named barriers generate this output when they are not watched.

Since named barrier `"End multiplication"` is watched, we have detailed information about it in lines 15 to 28 of Figure 3.6. The hardware event counts depicted in lines 25 to 28 are specific to the phase that starts after barrier `"End initialization"` (line 25, Figure 3.3) and ends at barrier `"End multiplication"` (line 28 of Figure 3.3). Cumulative hardware event counts for the complete execution are in lines 30 to 33 of Figure 3.6. Counts are shown independently for each thread. For example, the thread with identifier 1 incurred 73,170,261 L2 cache misses (line 33 of the Figure).

Hypothetically, if there had been a logical bug in the program and a deadlock occurred within function `matrix_multiply()`, the user would have seen the output `"End initialization"` and then no more output. Since the programmer knows there is a named barrier called `"End multiplication"` that comes after the first named barrier, the immediate suspicion is that the program is hung somewhere between the two named barriers. If necessary, more barriers can be easily added to and removed from a program to further triangulate the problem. Without SBT barriers, the programmer would either have to attach a debugger to the threads or add calls to `printf()` in order to discover where the program is stopped. Therefore, SBT barriers provide a convenient way to localize the problem quickly and easily. In addition, SBT barriers produce a trace of events leading up to the problem. For the remainder of the discussion, we ignore the possibility of deadlock.

3.2.2 LU Decomposition

With the purpose of further illustrating how SBT is used in practice, we now consider two different versions of a naive implementation of LU decomposition. Given a matrix A, the

```
 1  /* block partitioning */
 2  for( k=0; k<mat_size-1; k++ )  {
 3      aux = MAX( k+1, start_row );
 4      for( i=aux; i<=end_row; i++ )  {
 5          a[i][k] /= a[k][k];
 6          for( j=k+1; j<mat_size; j++ )  {
 7              a[i][j] -= a[i][k] * a[k][j];
 8          }
 9      }
10      BARRIER;
11  }
```

Figure 3.8: LU decomposition code with block work-partitioning.

```
 1  /* cyclic partitioning */
 2  for( k=0; k<mat_size-1; k++ )  {
 3
 4      for( i=k+my_id+1; i<mat_size; i+=thread_count )  {
 5          a[i][k] /= a[k][k];
 6          for( j=k+1; j<mat_size; j++ )  {
 7              a[i][j] -= a[i][k] * a[k][j];
 8          }
 9      }
10      BARRIER;
11  }
```

Figure 3.9: LU decomposition code with cyclic work-partitioning.

algorithm decomposes it into two matrices L and U, such that L is lower triangular, U is upper triangular, and $A = LU$. The first version uses block work-partitioning; the second version uses cyclic work-partitioning. We have also successfully applied SBT barriers to the highly-tuned SPLASH-2 implementation of LU decomposition [23] to gain experience in porting existing code to SBT (see Chapter 5).

Figure 3.7 gives an intuitive idea of the different work-partitioning strategies, assuming four threads. With block distribution, each thread is responsible for reducing a contiguous set of rows, while with cyclic distribution each thread reduces rows in round-robin fashion. Finally, 2D-block partitioning, used in the SPLASH-2 implementation, divides the matrix in blocks along both axes and assigns blocks to processes in an interleaved fashion.

In this Chapter, we focus only on block and cyclic work-partitioning schemes applied on simple implementations of LU decomposition. A more sophisticated version of LU decomposition will be discussed in Chapter 5. The core of the block work-partitioning algorithm (block-LU, for short) is shown in Figure 3.8, and the cyclic work-partitioning algorithm (cyclic-LU) is shown in Figure 3.9. Note that there are only two differences between the two fragments of code: block-LU needs to select the appropriate row number to start row reduction (line 3, Figure 3.8), and, more importantly, the declaration of the loop that drives

processes through the rows they will reduce (line 4 in both listings). In cyclic-LU, each process has to have a numeric identifier between 0 and `thread_count`-1 in order to determine the rows for which it is responsible.

An analysis of the differences in performance between the implementations represents a good example of how SBT can produce useful and insightful information about an application's runtime behavior. As with any set of similar parallel algorithms, real time and speedups are the first criteria for comparison. We ran the two programs to decompose a 2048×2048 matrix on 2, 4, and 8 processors. Hence, both programs execute a total of 2,048 iterations. The speedups, shown in Table 3.3, are calculated by comparing the parallel versions against a purely sequential algorithm. Superlinear speedups are due to well-known cache effects. For this discussion, we are interested only in the differences between block and cyclic work-partitioning, not absolute numbers.

The empirical speedups indicate that cyclic-LU achieves higher performance than block-LU — which is a well-known property of LU decomposition. Again, we are using this example merely to demonstrate the capabilities of SBT.

However, suppose the programmer first implemented LU decomposition using block work-partitioning. While running block-LU, output like that shown in Figure 3.10 would be generated. The Figure shows a subset of the entire output and corresponds to the $1{,}071^{st}$ loop iteration of the LU decomposition. The iteration number is purposely selected to be greater than half the total number of iterations, and allows us to clearly show the reason why block-LU has lower speedups. The warning on line 12 is generated because the barrier time exceeded the default threshold of 1000 milliseconds. The same warning can be seen for the first barrier of cyclic-LU in Figure 3.11. Before the first barrier, both programs allocate memory and initialize their data structures.

The output tells the programmer that thread 0 arrived at the barrier first (line 23, Figure 3.10) and thread 7 arrived last (line 30). The four columns of the output in lines 21 to 30 are, from left to right, thread identifier, thread inter-arrival time, seconds since `sbt_init()` was invoked, and real time (as returned by function `gettimeofday()`). Some of the threads arrived at nearly the same time, but there is a thread inter-arrival gap of 39, and 8 milliseconds between some threads, which explains why the barrier time is a relatively large 48 milliseconds (line 17, Figure 3.10). However, it is still unclear why there are large gaps in the thread inter-arrival times.

To provide greater insight, hardware performance counters can be used. In particular, a value of `0` for `libperfex` environment variable `T5_EVENT0` selects thread-by-thread CPU cycle counts for monitoring.[1] CPU cycle counts are a measure of the amount of time that each thread is active during the phase. We can see that the 8 different threads of block-LU

[1] Note that `libperfex` environment variables `T5_EVENT0` and `T5_EVENT1` can also be passed as command line parameters, like any SBT option. For example: `$ ./pmm T5_EVENT0 0`

Number of Processors	*Speedups*	
	Block distribution	*Cyclic distribution*
2	2.62	2.68
4	4.14	5.56
8	8.09	9.32

Table 3.3: LU decomposition speedups: 2048×2048 matrix and different distributions.

```
 1  SBT options (version 0.9 built for sproc)
 2          SBT_WATCH       (null)
 3          SBT_WATCH_ALL   1
 4          SBT_WARNINGS    1
 5          SBT_WARN_TIME   1000
 6          SBT_PHASE_TIMES 1
 7          SBT_NO_DEBUG    0
 8          Perfex events   yes
 9                  T5_EVENT0:      0
10                  T5_EVENT1:      17
11
12  SBT WARNING: barrier in par-lu.c:203
13          (barrier: 98324 ms > 1000 ms, phase 0: 204.628s, from init: 204.628s)
14  ...
15
16  SBT barrier watch in par-lu.c:320
17          Barrier time:   48 ms
18          Phase time:     0.050 sec
19          Total time:     313.048 sec
20          Order of arrival:
21                          inter      from              real
22                    id  thread      init              time
23                     0     0ms     312.999s        11:32:55.221
24                     3     0ms     312.999s        11:32:55.221
25                     1     0ms     312.999s        11:32:55.221
26                     2     0ms     312.999s        11:32:55.221
27                     4    39ms     313.039s        11:32:55.261
28                     6     8ms     313.047s        11:32:55.269
29                     5     0ms     313.047s        11:32:55.269
30                     7     0ms     313.048s        11:32:55.270
31
32          SBT: Phase 1071 perfex counter information
33           id       EVENT0= 0       EVENT1=17
34            0            3529            2573
35            1            3699            2525
36            2            4007            2567
37            3            4031            2561
38            4         7649956        10222319
39            5         9362004        12520577
40            6         9329122        12520602
41            7         9441874        12520601
```

Figure 3.10: Example SBT output for LU decomposition: Block distribution, 8 `sproc` processes.

```
 1  SBT options (version 0.9 built for sproc)
 2          SBT_WATCH       (null)
 3          SBT_WATCH_ALL   1
 4          SBT_WARNINGS    1
 5          SBT_WARN_TIME   1000
 6          SBT_PHASE_TIMES 1
 7          SBT_NO_DEBUG    0
 8          Perfex events   yes
 9                  T5_EVENT0:      0
10                  T5_EVENT1:      17
11
12  SBT WARNING: barrier in par-lu.c:203
13          (barrier: 99087 ms > 1000 ms, phase 0: 195.202s, from init: 195.202s)
14  ...
15
16  SBT barrier watch in par-lu.c:287
17          Barrier time:   3 ms
18          Phase time:     0.043 sec
19          Total time:     305.840 sec
20          Order of arrival:
21                          inter      from              real
22                    id  thread      init              time
23                     1     0ms     305.836s        11:16:21.782
24                     4     1ms     305.838s        11:16:21.784
25                     2     0ms     305.838s        11:16:21.784
26                     0     0ms     305.839s        11:16:21.784
27                     6     0ms     305.839s        11:16:21.785
28                     3     0ms     305.840s        11:16:21.785
29                     7     0ms     305.840s        11:16:21.785
30                     5     0ms     305.840s        11:16:21.785
31
32          SBT: Phase 1071 perfex counter information
33           id       EVENT0= 0       EVENT1=17
34            0         8045902         6017515
35            1         7560788         5968604
36            2         7843514         5968623
37            3         8153874         5968619
38            4         7878369         5968623
39            5         8236333         5968623
40            6         8105004         5968643
41            7         8169154         5968644
```

Figure 3.11: Example SBT output for LU decomposition: Cyclic distribution, 8 `sproc` processes.

SBT Option	*Value*	*Comment*
SBT_WATCH_ALL	1	Monitor all barriers.
T5_EVENT0	0 (CPU cycles)	Measure amount of time.
T5_EVENT1	17 (Graduated instructions)	Measure amount of computation.

Table 3.4: SBT options used with LU decomposition.

execute between 3,529 and 9,441,874 cycles in the LU decomposition loop before hitting the barrier (lines 32 to 41, Figure 3.10). Thread 0 arrives at the barrier first because it does little work in loop iteration 1,071 and it sits idle at the barrier waiting for the other threads after only 3,529 CPU cycles. In contrast, thread 7 executes for 9,441,874 CPU cycles before reaching the barrier. Due to normal non-determinism in the system, the threads may arrive at the barrier in a different order than the CPU cycle counts would indicate (e.g., thread 5 arrives at the same time as thread 6). However, the CPU cycle counts do provide a rough partial order of expected thread arrival orders.

Another hardware event, graduated instructions, is used in this example as a metric of the amount of computation performed by each process. Graduated instruction counts indicate the number of executed instructions that have a "final effect on the visible state of registers and memory" [4]. The third column in lines 33 to 41 of Figure 3.10, with header `EVENT1=17`, shows the number of graduated instructions for each process of the execution of block-LU.

In order to measure the actual amount of program-specific computation, graduated instructions are a more accurate metric than CPU cycles. On one hand, graduated instructions represent the amount of computation that is actually performed to produce the final result of the program (i.e., decompose the matrix). CPU cycles, on the other hand, are a less precise approach. They include the time required to execute instructions toward the completion of the computation, as well as those cycles used for other tasks that the MIPS processors perform (e.g., out-of-order and speculative execution). The difference between the two metrics is better understood when comparing the counts of both events in block-LU with those of cyclic-LU.

Together, the thread inter-arrival times and performance counters indicate that there is a load balancing problem. As can be seen in Figure 3.7, by loop iteration 1,071 of 2,048, half of the threads have no more work to do. Therefore, block-LU achieves relatively low speedups, compared to cyclic distribution, due to a flaw in its work partitioning strategy.

In contrast, cyclic-LU proves to be much better balanced, as can be seen in Figure 3.11. Even after 1,071 iterations, all processes arrive at the barrier within 3 milliseconds of each other (line 17, Figure 3.11). Furthermore, the distribution of work among different processes is roughly equal (lines 32 to 41, Figure 3.11), with all processes doing approximately the

	Graduated Instructions	
Process id	*Block-LU*	*Cyclic-LU*
0	3,209,383,633	17,947,432,340
1	9,084,493,141	17,934,321,387
2	14,120,934,408	17,921,286,970
3	18,318,413,162	17,908,144,635
4	21,676,965,902	17,895,032,607
5	24,196,764,977	17,881,902,442
6	25,877,719,018	17,868,829,960
7	26,719,746,550	17,855,652,277
TOTAL	143,204,420,791	143,212,602,618

Table 3.5: Process-specific cumulative graduated instruction counts for block-LU and cyclic-LU. Both versions perform similar amounts of computation.

same amount of work throughout all of the iterations (both in CPU cycles and graduated instructions).

Although CPU cycles can be useful in determining load imbalances, they can also be misleading if they are interpreted as a measure of the amount of computation performed by the program. As is stated, graduated instructions more accurately represent the amount of computation performed than CPU cycles. The sum of graduated instructions across processes for each phase can be used to compare the amount of computation performed by the two versions of LU in each phase. For example, this sum for phase 1,071 of block-LU totals 47,794,325 graduated instructions (the number is calculated by adding the individual processes' graduated instruction counts shown in the third column of lines 34 to 41 in Figure 3.10). Similarly, the sum for the same phase of cyclic-LU totals 47,797,894 graduated instructions. The difference between the two is a negligible 3,569 instructions, which confirms that the two versions of LU are performing the same amount of computation in iteration 1,071 of the execution. However, while cyclic-LU uses all of the processes to perform the computation, block-LU only uses a subset of them.

Table 3.5 shows the total counts of graduated instructions for each process in block-LU and cyclic-LU. The numbers are extracted from output produced by SBT. Recall that SBT outputs total accumulated event counts for each process upon library finalization. Also shown in Table 3.5 are the sums of those cumulative data across processes for each version of LU. The two programs have very similar numbers of graduated instructions during the execution, which confirms that the amount of computation is similar for the two.

If the programmer had not previously been aware of the benefits of cyclic work-partitioning for LU decomposition, the first warning that there might have been a performance problem would have been a barrier time warning (similar to line 12, Figure 3.10). By watching the indicated barrier and using hardware performance counters, the programmer learns that the cause of the warning is a load imbalance between the threads. Then, the programmer is able

to fix the performance bottleneck by implementing, for example, a cyclic work-partitioning strategy.

3.3 Concluding Remarks

With a few additional lines of code, users can take advantage of the performance information SBT generates. The matrix multiplication example demonstrates that users need add only seven lines of code to the source file in order to use SBT. In addition, SBT offers a simple and flexible user interface. Once programs are built and linked to SBT, users are able to easily select from a variety of performance metrics to monitor (e.g., phase times, barrier times, hardware event counts).

A simple metric, thread inter-arrival time, is shown to the user for all watched barriers. This metric can quickly direct our attention to a bottleneck and a potential load imbalance in block-LU. By correlating thread inter-arrival times with CPU cycle and graduated instruction counts, we confirm the existence of a load balancing problem. Performance information produced by SBT about cyclic-LU —which uses a more balanced work-partitioning technique— leads to the conclusion that the difference in performance between block-LU and cyclic-LU is a direct consequence of the different work-partitioning techniques.

SBT is a portable library; all the performance information that SBT produces is equally available on different hardware and software platforms. In this Chapter, we have illustrated two simple examples running on two common platforms: SGI/Irix and Intel/Linux.

Chapter 4

Implementation of SBT

In this Chapter we discuss the implementation of SBT. The library is implemented in C and supports programs written in C/C++ that use either POSIX threads or Irix's `sproc` as threading libraries. While the Pthreads version of SBT makes it widely available on several platforms, the `sproc` version takes advantage of the more specialized features present in Irix for parallel execution on SGI machines.

Because hardware event counting is an architecture-specific capability, it is not trivial to implement this functionality for a wide array of platforms. Thus, hardware counting capability is provided by linking SBT to one of three libraries: Performance Counter Library (PCL), Performance API (PAPI), and Irix's `libperfex`. The three libraries are supported with little difference in the user interface.

For all versions of SBT, it is possible to remove the overheads of performance monitoring by defining the compile-time label `SBT_OFF` and recompiling (e.g., `-DSBT_OFF` on the compile line). This version of the library contains just the bare calls to the barriers and has no additional overhead compared to calling the barrier synchronization directly from the program. Therefore, production runs of programs using SBT do not incur unnecessary overhead.

4.1 Overview

SBT's functionality is built around two abstract data types: `sbt_env_t` and `sbt_barrier_t`. When the library is initialized, a variable of type `sbt_env_t`, containing a field of type `sbt_barrier_t`, is allocated and initialized. Also, any SBT options set by the user are parsed and stored as data fields of the `sbt_env_t` variable. Performance information gathered at runtime is stored in the `sbt_barrier_t` field.

The `sbt_env_t` and `sbt_barrier_t` data structures, along with their interface functions, are discussed in Sections 4.2 and 4.3, respectively.

4.2 SBT Environment

SBT options, which affect the runtime behavior of the library, are set either through command line arguments or through shell environment variables. The current values of the options, along with the memory space required to store performance data gathered during execution, are part of SBT's *environment*. The environment is implemented using data structure `sbt_env_t`, shown in Figure 4.1. When the library is initialized through a call to `sbt_init()`, variable `env`, of type `sbt_env_t`, is allocated and initialized. The library initialization process performs the following tasks:

1. *General SBT options:* Parse SBT options from shell environment variables and command line arguments.

   Option values are stored in their corresponding fields. Each option has a field to store its value in structure `sbt_env_t`. For example, the value for option `SBT_WATCH_ALL` is stored in field `watch_all` (line 10 of Figure 4.1).

   Options set from the command line override those passed through environment variables.

2. *Event counter options:* If hardware events are to be counted, initialize the hardware event counter library SBT is linked to. Also, initialize the appropriate hardware counter related fields of SBT library variable `env`.

   The library to use for hardware counting (PCL, PAPI, or `libperfex`) is selected through compile-time flags when SBT is built. Storage space is allocated for cumulative and phase-specific hardware event counts. Lines 19 to 53 of Figure 4.1 contain the declarations of the fields for each of the three hardware counter libraries.

3. *Loop barriers:* Allocate and initialize loop barrier data structures.

   For each loop barrier in user code, at least one variable of type `loop_barrier_info_t` has to exist. Since the number of loop barriers in the code is unknown at library initialization time, an array with a default of 20 `loop_barrier_info_t` variables is allocated. Line 18 of Figure 4.1 shows the declaration of the array of loop barrier data structures.

4. *Default barrier data structure:* Allocate and initialize `env.b`, the field that represents the default barrier data structure to use for synchronization. This field is of type `sbt_barrier_t*`, as shown in line 2 of Figure 4.1.

   The `sbt_init()` function returns `env.b` to the user. If desired, users can create other `sbt_barrier_t` variables using `sbt_barrier_create()`. However, normal SPMD programs should not require more than one such structure.

The definition of data structure `sbt_env_t` depends on the flags defined at compile-time. Figure 4.1 shows the declaration of the data structure. Five compilation flags determine the shape of structure `sbt_env_t`. In the simplest case, when `SBT_OFF` is defined, the structure becomes a place holder for a barrier data structure (field `b`, of type `sbt_barrier_t*`, in line 2).

In order to build SBT to support `sproc` threads (i.e., with compile-time flag `SBT_SPROC`), the three fields in lines 4 to 6 of Figure 4.1 are required. These fields are further explained in Section 4.2.1.

Only one of the remaining three flags (`__SBT_PERFEX_H`, `__SBT_PAPI_H`, and `LINK_PCL`) is automatically defined during compilation, depending on the hardware counter specified by the user. All fields of `sbt_env_t` enclosed within these flag definitions (lines 19 to 53) are used at runtime to store and manage hardware counters. Again, only one of the three flags is defined, hence SBT links to only one of the three libraries. For example, if SBT is built to support `libperfex`, only the fields in lines 20 to 29 of Figure 4.1 will be present in the data structure.

4.2.1 `sproc` Arena Initialization

If SBT is built for `sproc` threads, the `SBT_SPROC` flag is automatically defined and the three fields in lines 4 to 6 of Figure 4.1 are included in the structure declaration. Irix's `sproc` threads require the use of an *arena* —a shared address space to which all processes of an application have to attach— to allow synchronization with barriers, locks, or semaphores; of these three synchronization primitives, SBT uses locks and barriers. Although all shared memory can be allocated globally, the same way that it would be done with Pthreads, the arena abstraction provides a more sophisticated set of memory allocation functions to manage shared memory for a set of processes (see Cortesi *et al.* [5], Chapter 3).

The use of `sproc` threads with SBT requires that an arena be initialized and processes added to it, before any instances of `sbt_barrier_t` are used. Users can set up their own arena —in which case a pointer to the arena must be passed to `sbt_init()`— or they can let SBT create and manage its own arena. In both cases, a pointer to the arena is kept in `env.sbt_arena` (line 4 in Figure 4.1).

If `sbt_init()` receives a `NULL` pointer in its arena argument, it creates an arena. In this case, `env.own_arena` is set to 1 to indicate that SBT is using its own arena. As processes arrive at the first barrier, they are explicitly added to the arena; after the barrier is surpassed, `env.all_added` is assigned 1 to keep processes from being added again.

Shared arenas, as well as the locks and semaphores they contain, can be initialized to behave in different ways, depending on a series of attributes set through successive calls to `usconfig()` [5]. Some of these attributes may also affect the behavior of the SBT barriers in

```
typedef struct _sbt_env_t  {
    sbt_barrier_t* b;   // Automatic SBT barrier;
#ifdef SBT_SPROC
    usptr_t* sbt_arena;
    int own_arena;      // Use SBT or user-initialized arena?
    int all_added;      // Have all processes joined the arena?
#endif
#ifndef SBT_OFF
    int options_msg;    // Toggle print SBT options; default is 1
    int watch_all;      // Toggles watch on all barriers; default is 0
    int no_debug;       // Toggles SBT on and off; default is 1
    int warn;           // Toggles warnings; default is 1
    int watch_line;     // Watched barrier's line number
    char* watch_name;   // Watched barrier's name
    int phase_times;    // Phase time output for all phases; default is 0
    int warn_time;      // Barrier time warning threshold
    struct timeval ts_init;
    loop_barrier_info_t* loop_barriers;
#ifdef __SBT_PERFEX_H
    int pfx_count;              // Number of perfex events to count
    int ev0;                    // value of environment variable T5_EVENT0
    int ev1;                    // value of environment variable T5_EVENT1
    int* pfx_gen;               // Thread-specific generation number
    char* ev0_overflow;         // Accumulators overflow flags
    char* ev1_overflow;
    long long *ev0_count;       // Event counts
    long long *ev1_count;
    long long *ev0_accum;       // Event cumulative counts
    long long *ev1_accum;
#endif
#ifdef __SBT_PAPI_H
    int papi_count;             // Number of PAPI events to count
    int* papi_evs;              // PAPI event sets
    char** papi_overflow;       // Accumulators overflow flags
    long long **papi_evs_count; // Event counts
    long long **papi_evs_accum; // Event cumulative counts
    int* active_papi_evs;       // PAPI event codes
#endif
#ifdef LINK_PCL
    int pcl_thread_id;  // thread to use PCL functions; default is 0
    int pcl_events_cnt; // Total number of PCL events to count; default is 0
    int f_pcl_ev_cnt;   // Number of PCL_FP_CNT_TYPE events; dfault is 0
    int l_pcl_ev_cnt;   // Number of PCL_CNT_TYPE events; dfault is 0
    int pcl_event_list[PCL_COUNTER_MAX];        // Events to count
    char l_event_names[PCL_COUNTER_MAX][20];    // PCL event names
    char f_event_names[PCL_COUNTER_MAX][20];
    char** l_overflow_flags;            // Accumulators overflow flags
    char** f_overflow_flags;            // Accumulators overflow flags
    PCL_CNT_TYPE** l_events;            // Integer event counts
    PCL_CNT_TYPE** l_events_accum;      // Integer event cumulative counts
    PCL_FP_CNT_TYPE** f_events;         // Float event counts
    PCL_FP_CNT_TYPE** f_events_accum;   // Float event cumulative counts
#endif
#endif  // SBT_OFF
} sbt_env_t;
```

Figure 4.1: Declaration of data structure `sbt_env_t`.

the application (e.g., maximum number of processes in the arena, lock metering information, permissions to attach to the arena). Users should take all these attributes into account when they provide SBT with an already initialized arena.

In cases where SBT creates its own arena, it will do so with the following attributes:

- *Arena type:* Only processes that are part of a group with a shared file descriptor for the arena will be allowed to attach to it.
- *Maximum number of users:* This will be set to the number of processes that are expected to use the barriers (as per the `int` parameter passed to `sbt_init()`).
- *Arena size:* The file that represents the arena in the file system will be allowed to grow as necessary. As the user initializes more instances of `sbt_barrier_t`, the arena will grow accordingly. Note, however, that this may cause problems if the file system has limited free space.
- *Lock type:* Locks declared within the SBT arena will not generate metering or debugging information.

4.3 SBT Barriers

SBT barriers are wrappers around pre-existing barrier primitives. Our goal with SBT is to build the performance debugging and monitoring functionality around the barrier itself. Although SBT could implement barriers of its own, the decision to use pre-existing barriers gives flexibility to the user. Higher performance implementations of barrier synchronization are possible, and they can be easily inserted to replace the current barrier calls in the SBT code.

When SBT is built to support `sproc` threads, it uses the default `sproc barrier()` function, and when it is built for Pthreads, it uses its own implementation (Pthreads do not have a barrier primitive). Section 4.3.1 discusses SBT's implementation of a barrier function for Pthreads.

Once the appropriate barrier primitive is in place, SBT barriers perform the information-gathering and processing independently of the threading library in use.

The data structure `sbt_barrier_t` wraps the underlying barrier and also provides storage space for gathering performance data. This data structure, shown in Figure 4.2, changes according to compile-time flags `SBT_OFF`, `SBT_SPROC`, and `SBT_PTHREAD`. The definition of `SBT_OFF` results in a data structure with only two fields: the underlying barrier (line 2 of the Figure), and the number of threads that will be synchronized at the barrier (line 3).

On the other hand, when SBT is built for performance monitoring, `sbt_barrier_t` has a number of fields that will be used at runtime to store data (lines 13 to 20). In addition, SBT

```
 1  typedef struct _sbt_barrier_t  {
 2      barrier_t* b;
 3      int thread_count;   // Number of threads expected at the barrier
 4
 5  #ifndef SBT_OFF
 6      int counter;        // Number of threads currently at the barrier
 7  #ifdef SBT_SPROC
 8      ulock_t mutex;      // Mutex variable for critical section
 9  #elif defined(SBT_PTHREAD)
10      pthread_mutex_t mutex;
11  #endif
12
13      int* thread_list;   // List of threads ordered as they hit the barrier
14      int barrier_time;   // Time elapsed from tstamp_first to tstamp_last (ms)
15      double phase_time;  // Seconds elapsed between current and previous barrier
16      double from_init;   // Seconds elapsed from sbt_barrier_init()
17      int phase_number;   // Keep track of phase numbers
18
19      struct timeval * ts_arrival;    // Threads timestamp arrival at barrier
20      struct timeval start_phase;     // Time current phase started
21
22  #endif  // SBT_OFF
23  } sbt_barrier_t;
```

Figure 4.2: Declaration of data structure `sbt_barrier_t`.

barriers use fields `counter` (line 6, Figure 4.2) and `mutex` to allow an orderly gathering of data. The declaration for field `mutex` depends on the threading library to use, as they have different type names for the same functionality. Line 8 of the figure contains the declaration of `mutex` for `sproc` threads, and line 10 for Pthreads.

SBT's performance monitoring barriers include the necessary code to produce performance information. Figure 4.3 shows the pseudocode for such barriers. As threads enter a barrier, they gather and store data. Within a critical section (lines 3 to 5 of the figure), threads increment a counter, record their identifier, and timestamp their arrival. After having stored this information and exited the critical section, threads call the actual underlying barrier (line 6). Following the synchronization point and the information-gathering code, one of the threads is assigned the task of outputting the required information for the current barrier (e.g., barrier time, phase time, hardware counters, warnings). Lines 7 to 9 in Figure 4.3 show that the thread with identifier `MASTER` is responsible for processing the gathered information.

When SBT is built with the compile-time flag `SBT_OFF`, no performance information is gathered. In this case, SBT barriers inform the user only that a barrier has been reached and invoke the underlying barrier. Figure 4.4 shows the pseudocode that SBT barriers execute in such a case.

In the case shown in Figure 4.3, while the gathered performance information is output, a potential race condition could arise if only one instance of `sbt_barrier_t` is used to

```
 1  stop( counters );
 2  save( thread_ID, counters );
 3  acquire_mutex();
 4  save( thread_ID, timestamp );
 5  release_mutex();
 6  underlying_barrier();
 7  if( thread_ID==MASTER )  {
 8      process_information();
 9  }
10  underlying_barrier();
11  start( counters );
```

Figure 4.3: SBT barrier pseudocode when monitoring is enabled.

```
1  if( thd_id==0 )  {
2      printf( "SBT: %s:%d\n", __FILE__, __LINE__ );
3  }
4  underlying_barrier();
```

Figure 4.4: SBT barrier pseudocode when monitoring is disabled (i.e., SBT is built with compile-time flag `SBT_OFF`).

implement two consecutive barriers.[1] Consider the case where a thread reaches the next barrier and overwrites values stored during the previous one. If the information for the first barrier has not yet been printed out, the newly stored values for the faster thread will be output and the information for that thread's previous barrier will be effectively lost. This is why an extra synchronization point is added at the end of the information printing process (line 10 in Figure 4.3. In this way, while one thread is printing information, the others wait for it before continuing on to the next phase of the computation. This also helps produce more accurate performance information, since all threads will start the next phase at the same time, and the thread responsible for printing will not suffer from tardiness. All timestamps, as well as hardware counters' start/stop operations, are done at the beginning and end of the actual computation phases (i.e., at the end and beginning of the SBT barrier code, respectively), so the time spent by threads gathering and printing information is ignored and does not influence the performance information produced for the run. The resulting increase in wall-clock time is negligible and justifiable, considering the useful data that are handed to the user. Again, the synchronization overhead for SBT can be removed by defining `SBT_OFF` and re-compiling for production runs.

After performance information is output, all threads are released and start the next phase of execution.

[1] By default, SBT expects that only one instance of **`sbt_barrier_t`** will exist at runtime. Nevertheless, users are free to create more if desired, and SBT is capable of supporting as many as can fit in memory.

4.3.1 POSIX Threads Barriers

There is no barrier function in Pthreads proper. As a consequence, the first building block of SBT for Pthreads is a simple barrier primitive. Figure 4.5 shows the declaration for data structure `barrier_t` in lines 1 to 7, and the barrier code itself in lines 10 to 26. The initialization function for `barrier_t`, in lines 29 to 44, requires one integer argument that is used to set the value of the `thread_count` field. Also, a function to free a barrier, `free_barrier()`, in lines 47 to 51 of the figure, is implemented.

The `barrier()` function makes use of a mutex and a condition variable defined as fields of the `barrier_t` structure (lines 3 and 4 in Figure 4.5, respectively). As threads arrive at the barrier, they acquire the mutex (line 13) and enter a critical section to increment a counter (line 15) and suspend themselves on the condition variable (line 22), which automatically relinquishes the mutex. The last thread to arrive restarts its siblings by broadcasting a signal for the condition (line 18), and resets the counter to 0 (line 19), leaving it ready for the next barrier call. It is possible for the user to implement and use a higher-performance barrier, if desired.

4.3.2 Loop Barriers

The only difference between loop barriers and anonymous or named barriers, is that while the latter output all information as soon as all threads have reached them, the former aggregate data and output cumulative information when the library is freed (i.e., `sbt_finalize()` is called). Loop barriers are, by nature, ideal to be inserted within loops, in cases where the user wishes to obtain performance information for phases that are contained within the loop (i.e., the barrier that marks the end of a phase is in the loop).

For each loop barrier in the user's code, SBT allocates and initializes one instance of `loop_barrier_info_t`, the data structure shown in Figure 4.6, which is capable of accumulating performance data (i.e., barrier time, phase time, phase start and end time, thread idle times, and hardware counters). As a loop barrier is reached on each iteration, the data that is gathered for the ending phase and the current barrier are accumulated into the loop barrier data structure. At the end of the execution, when the library is freed, cumulative information for all loop barriers is output.

SBT maintains an array of loop barrier data structures as a field in the library's `sbt_env_t` variable (see field `loop_barriers` in line 18 of Figure 4.1).

SBT does not know the number of loop barriers that will be invoked at runtime until the program terminates, and each loop barrier in the code requires its own `loop_barrier_info_t` structure. As a consequence, storage space is allocated for a default of 20 loop barriers during library initialization. Although SBT is capable of allocating more loop barriers dynamically, such an operation can be expensive and can be easily avoided. In order to avoid the potential

```
 1  typedef struct
 2  {
 3      pthread_mutex_t mutex;
 4      pthread_cond_t condition;
 5      int thread_count;
 6      int counter;
 7  } barrier_t;
 8
 9
10  void barrier( barrier_t* b )
11  {
12      /* enter critical section */
13      pthread_mutex_lock( &(b->mutex) );
14      /* increment counter */
15      (b->counter)++;
16      if( b->counter == b->thread_count )  {
17          /* barrier reached => resume threads and reset counter */
18          pthread_cond_broadcast( &(b->condition) );
19          b->counter = 0;
20      } else {
21          /* barrier not reached => suspend */
22          pthread_cond_wait( &(b->condition), &(b->mutex) );
23      }
24      /* leave critical section */
25      pthread_mutex_unlock( &(b->mutex) );
26  }
27
28
29  barrier_t* barrier_init( int count )  {
30      barrier_t* b;
31      b = (barrier_t*)malloc( sizeof( barrier_t ) );
32      if( !b )  {
33          printf( "%s:%d: Could not allocate memory for barrier_t.\n",
34                  __FILE__, __LINE__ );
35          return( NULL );
36      } else {
37          /* initialize mutex and condition with default values */
38          pthread_mutex_init( &(b->mutex), NULL );
39          pthread_cond_init( &(b->condition), NULL );
40          b->thread_count = count;
41          b->counter = 0;
42          return( b );
43      }
44  }
45
46
47  void free_barrier( barrier_t* b )  {
48      pthread_cond_destroy( &(b->condition) );
49      pthread_mutex_destroy( &(b->mutex) );
50      free( (void*)b );
51  }
```

Figure 4.5: SBT's implementation of a barrier primitive for Pthreads.

```
 1  typedef struct _loop_barrier_info_t  {
 2      char* file;                 // file where barrier is invoked
 3      int line;                   // line where barrier is invoked
 4      char* name;                 // barrier name
 5      int flag_used;              // instance being used?
 6      int barrier_time;           // accumulated barrier time in msec
 7      double phase_time;          // accumulated phase time in sec
 8      struct timeval start_time;  // time when the first subphase started
 9      struct timeval end_time;    // time of last thread's arrival to last L_BAR.
10      int subphase_cnt;           // number of times the L_BARRIER is reached
11      int* thd_idle_msec;         // msec each thread spent at the barrier
12
13  #ifdef LINK_PERFEX
14      long long *ev0;
15      long long *ev1;
16  #endif
17
18  #ifdef LINK_PAPI
19      long long **papi_evs;
20  #endif
21
22  } loop_barrier_info_t;
```

Figure 4.6: Declaration of data structure `loop_barrier_info_t`.

extra cost, the default number of loop barriers to allocate during library initialization can be increased before compiling SBT.

At runtime, SBT identifies individual loop barriers using the file name and line number in which the barrier lies (lines 2 and 3 in Figure 4.6). When a loop barrier is reached for the first time, an unused element of the array `env.loop_barriers` (line 18 of Figure 4.1) is found and used to store performance data. The array is re-allocated if all its elements are in use. SBT uses the fields in lines 6 to 11 of Figure 4.6 to store the newly gathered data. When the same loop barrier is reached in later iterations, performance information is accumulated in those same fields.

Loop barriers also have the necessary fields to accumulate hardware counter information (lines 13 to 20, Figure 4.6).

4.4 Hardware Performance Counters

Contemporary CPU architectures support hardware counters for important low-level events that affect performance. Data such as cache miss counts, number of CPU cycles, number of graduated instructions, and floating point operations completed, can be crucial to the process of finding a program's bottlenecks. Since the counters are implemented in hardware, they also incur less overhead than software-instrumented programs.

The main concern regarding the implementation of hardware counting for SBT is portability. The libraries to access the counters, and the specific counters that are available on

different CPU architectures, differ from system to system. The Performance Counter Library (PCL) [1] is a useful tool since it provides a unified API to the hardware counters on a number of different architectures. The main drawback of PCL is that, as of version 1.3 —the one SBT supports— it is not thread-safe. Nevertheless, SBT can be linked to PCL, and although the hardware counter information can only be gathered for one thread, the user can specify and vary the thread to watch. Multiple runs are required to gather performance numbers for all threads.

As a means of getting counting information for individual processors, SBT can be alternately linked to `libperfex` [4], SGI's performance counting library. Additionally, SBT supports the PAPI library [3], which is portable, thread-safe, and has a large platform of users and developers. In the last year, PAPI has become more prevalent in the development community.

For each of the three hardware counter libraries, SBT's `sbt_env_t` structure has fields that are used at runtime to store phase-specific counts and accumulate overall execution counts (see lines 19 to 53 in Figure 4.1). In addition to those fields, there are other fields used for bookkeeping (e.g., number of events to count, event identifiers, overflow flags).

Hardware counters are started at the beginning of each phase and stopped at the end only when the user specifies one or more hardware events to count. This translates into a function call to stop the counters at the beginning of the barrier code (line 1 in Figure 4.3, and another call to restart the counters before returning and entering the next phase (line 11, Figure 4.3). Note that there will be no hardware counters active during phase 0, since the counters are not actually started until the return point of the first barrier.

As the computation progresses, the counter values for each phase are output for every watched non-loop barrier. For loop barriers being watched, event counts are accumulated in fields of the corresponding `loop_barrier_info_t` structure (lines 13 to 20 in Figure 4.6). Overall program event counts are also accumulated at every barrier. Overflow checks are performed on cumulative execution counts and the appropriate overflow flags are set accordingly.

Loop barriers do not support the use of PCL for hardware event counters. SBT's support for PCL was discontinued before loop barriers were added. PAPI provides the same functionality, is equally portable, and is thread-safe.

At the end of the computation, when the library is freed, accumulated event counts for the whole execution and for each loop barrier are shown. At this time, the thread-specific overflow flags are inspected; if any of them evidence an overflow, SBT issues a warning.

4.5 Concluding Remarks

The current implementation of SBT satisfies the design goals of on-line monitoring, low probe effect, ease of use, and portability stated in Section 1.1. As well, SBT provides a common API and user interface for all the platforms it supports.[2]

The interfaces with the three hardware performance libraries are independent of the main implementation file. Any changes in those libraries' interfaces can be quickly applied to SBT without compromising the rest of the implementation.

[2]There is, however, one difference in the API. Function `sbt_init()` takes different arguments depending on the threading library. See Section B.11.

Chapter 5

SPLASH-2 Examples

The Stanford Parallel Applications for Shared Memory (SPLASH-2) suite is a set of applications developed as a tool to compare the performance of different shared memory multiprocessors [23]. It provides a convenient framework to allow such comparisons by making the code freely available, as well as by specifying basic data sets and problem sizes for each application.

As a detailed illustration of SBT's usefulness, this Chapter describes a port of some SPLASH-2 applications to Irix `sproc` threads, and shows performance measurements obtained through the use of SBT. Although the original SPLASH-2 codes leave room for optimizations (dependent on the platform and threading library used), it is beyond the scope of this thesis to actually improve the performance of the applications. The motivation for using these codes is solely to demonstrate the ease and usefulness of instrumenting commonly known programs with SBT.

Three of the eleven applications and kernels that comprise SPLASH-2 are ported: radix, LU decomposition, and water-n^2. SPLASH-2's version of LU decomposition is more sophisticated than the versions discussed in Chapter 3. Even though the suite explicitly defines base problem sizes for each application, larger data sets are used to obtain the results described in this chapter. Moore's Law has taken its toll and problem sizes that looked challenging in 1995 are too small six years later.

The source code for all of the SPLASH-2 programs is sequential. In addition, the codes include a set of hooks in the form of macros that can be used for parallelization. These macros, which are initially null, are mainly concerned with the creation and synchronization of processes. The programs are modified for parallel execution using `sproc` threads and SBT, by defining those macros with calls to the Irix libraries and replacing the original barrier primitives with calls to the SBT barrier macros. No other modifications are introduced in the original SPLASH-2 codes.

The performance measurements shown in this chapter are all averages calculated from the output of five 4-process runs. Again, since the purpose of these experiments is to demon-

strate SBT's capabilities, and not to show optimized performance or scalability, averages over five 4-processor runs suffice. The system used to run the experiments is an SGI Origin 2100 with 4×350 MHz MIPS R12000 processors and 1 GB of shared RAM.

The next Section contains a description of how the SPLASH-2 codes are parallelized and instrumented with SBT barriers, including an example of the actual lines of code that are added or modified for the instrumentation. After this, the parallel algorithms used by the selected SPLASH-2 applications are described in order to introduce the discussion of measurements and program behavior presented in later sections. Instrumentation overhead incurred by SBT is quantified and discussed in Section 5.5.

Finally, some performance measurements extracted from SBT's output are presented, along with discussions of their significance, for each of the ported programs. Three aspects of the execution will be analyzed:

- Phase times.
- Cumulative idle times at loop barriers.
- Hardware counters.

These three metrics help answer questions 2 and 3, as stated in Section 1.2. By looking at phase times and hardware counters, we are able to determine which phase is the most computationally-intensive and where the bottlenecks of the program are. Cumulative idle times at loop barriers can direct our attention to a potential load imbalance; the reason for a poor load balance can then be established by looking at certain hardware event counts.

5.1 Parallelization and Porting

The SPLASH-2 codes are distributed as sequential programs with hooks that can be used for parallelization. Few or no modifications are necessary to compile and run them sequentially on an SGI system. For example, the LU code requires only a definition for a `PAGE_SIZE` macro; after seeing the context in which the macro is used in the code, it is not difficult to infer its correct definition. The radix code, on the other hand, can be compiled and executed sequentially without any modifications to the original source.

Parallelization of the codes is done at two levels. First, a small set of Thread Control Block (TCB) functions, capable of creating either `Pthreads` or `sprocs`, is used. TCB creates an array of `tcb_t` elements — each of which is associated with a thread or process,[1] and contains an internal thread identifier and the total count of threads in the array. The core of TCB's source code is shown in Figure 5.1. Two functions are provided: `tcb_init()` (line 20) to initialize the array of processes and `launch_threads()` (line 38) to launch processes.

[1] Throughout this Chapter, we use the terms *thread* and *process* interchangeably.

```
1  #ifdef TCB_PTHREAD
2  typedef void*(*tcb_function_t)(void*);
3  #elif TCB_SPROC
4  typedef void(*tcb_function_t)(void*);
5  #endif
6
7  typedef struct _tcb_t  {
8      int id;
9      int thread_count;
10 #ifdef TCB_PTHREAD
11     pthread_t self;
12 #elif TCB_SPROC
13     pid_t self;
14 #endif
15 } tcb_t;
16
17 tcb_t* tcb_init( int );
18 int launch_threads( tcb_t*, tcb_function_t );
19
20 tcb_t* tcb_init( int thread_count )  {
21     int i;
22     tcb_t* tcb;
23     // allocate memory for tcb_t*
24     if( !(tcb = (tcb_t*)malloc( thread_count*sizeof(tcb_t) )) )  {
25         printf( "%s:%d: Could not allocate memory for tcb_t*\n", __FILE__, __LINE__ );
26         return( NULL );
27     }
28     // initialize thread id's
29     for( i=0; i<thread_count; i++ )  {
30         tcb[i].id = i;
31     }
32 #ifdef TCB_SPROC
33     tcb[0].self = getpid();
34 #endif
35     return( tcb );
36 }
37
38 int launch_threads( tcb_t* tcb, tcb_function_t foo )  {
39     int i;
40 #ifdef TCB_PTHREAD
41     int error;
42 #elif TCB_SPROC
43     int wait_stat;
44     pid_t pid;
45 #endif
46     for( i=1; i<tcb[0].thread_count; i++ )  {
47 #ifdef TCB_PTHREAD
48         if( error=pthread_create( &(tcb[i].self), NULL, foo, &(tcb[i]) ) ) {
49             printf( "%s:%d: Could not create thread %d. %s\n", __FILE__,
50                     __LINE__, i, strerror( error ) );
51             fflush( stdout );
52             return( 0 );
53         }
54 #elif TCB_SPROC
55         if( (pid=sproc( foo, PR_SALL, &(tcb[i]) ))<0 )  {
56             printf( "%s:%d: Could not create thread %d. %s\n", __FILE__,
57                     __LINE__, i, strerror( errno ) );
58             fflush( stdout );
59             return( 0 );
60         }
61         tcb[i].self = pid;
62 #endif
63     }
64     return( 1 );
65 }
```

Figure 5.1: Thread Control Block (TCB) source code. SPLASH-2 applications are parallelized using TCB functions.

Second, the macros to initialize data structures needed for parallelism (e.g., processes, barriers, locks) are implemented. Recall that the SPLASH-2 codes contain the macro call sites, but the macro bodies have to be implemented. Some of the macros are defined as direct invocations of TCB functions; others are defined as calls to SBT initialization and finalization routines.

We had to make a few other minor modifications — mainly replacement of the barriers and use of `struct timeval` for time measurements, rather than `unsigned int`. Table 5.1 shows all the changes we made to one version of LU in order to use SBT barriers. Each row of the Table represents a contiguous set of lines that differ between the original and the ported source files.

The instrumentation of existing codes with SBT does not require extensive changes to the sources.

5.2 Radix Sort

The parallel version of radix, a sorting algorithm first invented more than a century ago [26], comprises three phases: build local histograms, build a global histogram, and permute keys. Execution of the algorithm proceeds iteratively, analyzing one digit of an integer per iteration to populate the histograms and then permuting the keys accordingly, starting with the least significant digit.

Phase 1: Build local histograms. Each processor is assigned a range of the data set from which it will build a histogram of keys. To avoid lock contention on a global histogram, each processor allocates memory and initializes its own local histogram. There is no need for synchronization or communication among processes during this phase, although they are all required to wait at a barrier, which we call *barrier 1*, before starting work on phase 2.

Phase 2: Build global histogram. A tree-summing algorithm is used to calculate the global histogram, returning in each element the number of digits with a value smaller than its index. Densities and ranks for every digit are accumulated globally, so processes know where to store each key in the next phase. Parallelization of this phase requires synchronization and interprocess communication to keep some processes idle during the tree computation. Although a barrier that divides this phase into two sub-phases is part of the required synchronization, all data relative to phase 2 of radix are calculated as the sum of the numbers pertinent to both sub-phases. The barrier at the end of the second sub-phase, which marks the end of the phase, will be referred to as *barrier 2*.

Phase 3: Permute keys. Using the information stored in the global histogram,

Original Code	*Ported Code*
In function `main()`	

	`/* for getpagesize() */`
	`#include <unistd.h>`
	`#include "tcb.h"`
	`#define SBT_BARRIER  Global->start`
	`#define SBT_THREADID MyNum`
	`#include "sbt_barrier.h"`

`unsigned int starttime;`
`unsigned int rf;`
`unsigned int rs;`
`unsigned int done;`
	`struct timeval starttime;`
	`struct timeval rf;`
	`struct timeval int rs;`
	`struct timeval done;`

`void SlaveSort();`	`/* parameter is thread id */` `void SlaveSort(void*);`

`unsigned int start;`	`struct timeval start;`
	`int PAGE_SIZE = getpagesize();`
	`tcb_t* tcb;`

`MAIN_INITENV(,150000000)`	`/* initialize tcb to use P processors */` `MAIN_INITENV( tcb, P );`

`BARINIT(Global->start);`
`LOCKINIT(Global->idlock);`
`Global->id = 0;`
`for (i=1; i<P; i++) {`
`  CREATE(SlaveStart);`
`}`
	`BARINIT(Global->start, P);`
	`/* TCB handles thread id's; no need to */`
	`/* use Global->idlock                  */`
	
	`CREATE(tcb, SlaveStart);`

`SlaveStart(MyNum);` `WAIT_FOR_END(P-1)`	`SlaveStart((void*)tcb);` `WAIT_FOR_END(tcb);`
`/* timings calculated as difference */` `/* between unsigned int's          */`	`/* timings calculated as difference */` `/* between struct timeval's        */`
In function `SlaveStart()`	

`LOCK(Global->idlock)`
`MyNum = Global->id;`
`Global->id ++;`
`UNLOCK(Global->idlock)`
	`/* TCB handles thread id's; no need to */`
	`/* use Global->idlock                  */`
	`MyNum = ((tcb_t*)id)->id;`

In function `OneSolve()`	

`unsigned int myrs;`
`unsigned int myrf;`
`unsigned int mydone;`
	`struct timeval myrs;`
	`struct timeval myrf;`
	`struct timeval mydone;`

	`/* SBT's BARRIER macros are used   */`
	`/* instead of the original BARRIER */`
	`/* macro                           */`

In function `lu()`	
`unsigned int t1, t2, t3, t4, t11, t22;`	`struct timeval t1, t2, t3, t4, t11, t22;`

	`/* SBT's BARRIER macros are used   */`
	`/* instead of the original BARRIER */`
	`/* macro                           */`

Table 5.1: Lines modified in the original SPLASH-2 LU source code to use SBT. Contiguous blocks version of LU.

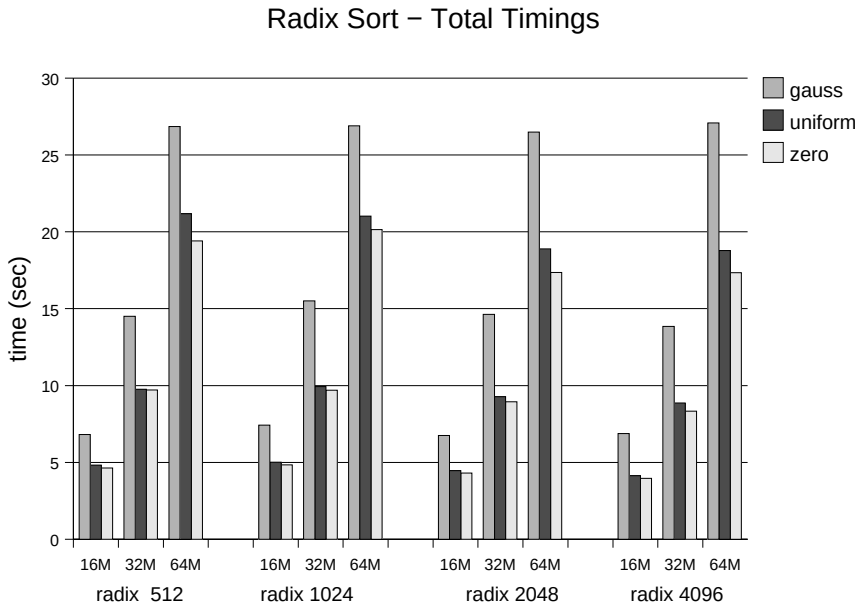


Figure 5.2: Radix sort total timings: 4 processors; different radii, distributions, and data set sizes. Radix binary linked to benchmarking version of SBT (compiled with `SBT_OFF`). Radix 512 corresponds to a radix of 9 bits, radix 1024 corresponds to 10 bits, etc.

processes permute keys to their corresponding positions in the global output array. Although no interprocess communication is necessary in this phase, any given process will store keys in several partitions belonging to other processes (as was determined in phase 1). After all keys have been permuted according to the current digit, all processes meet at a barrier and continue on to the next iteration. We call this barrier *barrier 3*.

Differences in performance are found not only by changing the size and distribution of the data set but also by using radii of different sizes [19]. Figure 5.2 depicts the different total execution times for parallel radix on 4 processors using different data set sizes, key distributions, and radix sizes.

Of the three distributions of keys to be sorted for which results are shown, Gauss is the one used by the original SPLASH-2 radix code. In this distribution, each key is the result of averaging four consecutive pseudo-random numbers recursively generated using the following rule: $x_{k+1} = ax_k \bmod 2^{46}$, where $a = 5^{13}$ and $x_0 = 314159265$. Uniform distribution is a sequence of consecutive, uniformly-distributed pseudo-random numbers in the range $[0, 2^{31})$ generated using `lrand48()`. Finally, the zero distribution is the same as uniform distribution, with the exception that every tenth key is explicitly assigned a value of 0, thus producing a data set with many repetitions of at least one key.

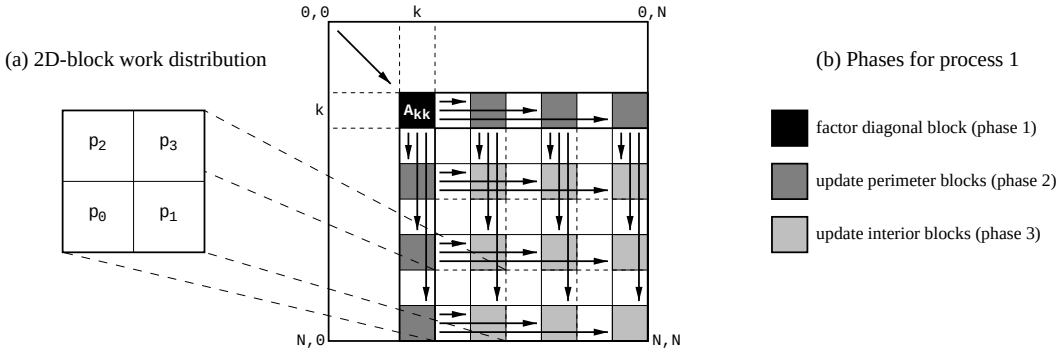


Figure 5.3: Parallel LU work distribution technique (as shown in [24]) on 4 processes, and phase decomposition of the execution for process 1.

5.3 LU Decomposition

Another kernel included with the SPLASH-2 suite is parallel LU decomposition. This kernel decomposes a matrix into its *LU* form, as discussed in Section 3.2.2.

The SPLASH-2 implementation of parallel LU distributes work to processes using a 2D-block scheme, by which matrix A is divided into square blocks along both axes, and blocks are assigned to processes in an interleaved fashion (see Figure 3.7 in Chapter 3). Processes own and are responsible for allocating and working on equal numbers of blocks, thus reducing communication. In this context, the size of the blocks is important, for it determines the runtime behavior of the program in terms of cache misses and load balance [23].

Given a dense matrix $A_{n \times n}$, the algorithm first divides it into an $N \times N$ array of $B \times B$ blocks such that $n = NB$, and then iterates $0 \leq k < N$ times over the following phases [24]:

> **Phase 1: Factor diagonal block A_{kk}.** The process that owns block A_{kk} (as seen in Figure 5.3, part (a)) factors it, while the others wait at the next barrier.
>
> **Phase 2: Update perimeter blocks in column k and row k using factored block A_{kk}.** Once phase 1 has been completed, all processes go on to update the perimeter blocks that they own, using the newly factored block A_{kk}. Perimeter blocks are those on column k, starting at row $k+1$; and row k, starting at column $k+1$.
>
> **Phase 3: Update interior blocks using corresponding perimeter blocks.** With all perimeter blocks modified according to block A_{kk}, each process subtracts from its own interior blocks (all those in rows $k < i \leq N$ and columns $k < j \leq N$) the product of the perimeter blocks $A_{ik} \times A_{kj}$.

Two important parameters that may affect the performance of this algorithm are the size of the matrix and the size of the blocks. Phase 1, which is purely sequential, is mainly

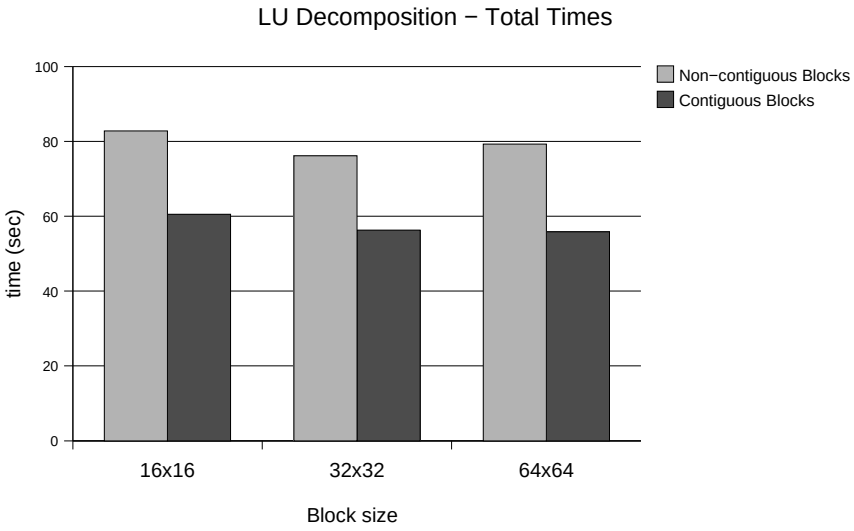


Figure 5.4: LU decomposition computation times on 4 processors using different block sizes and block allocation methods on a 2048×2048 matrix.

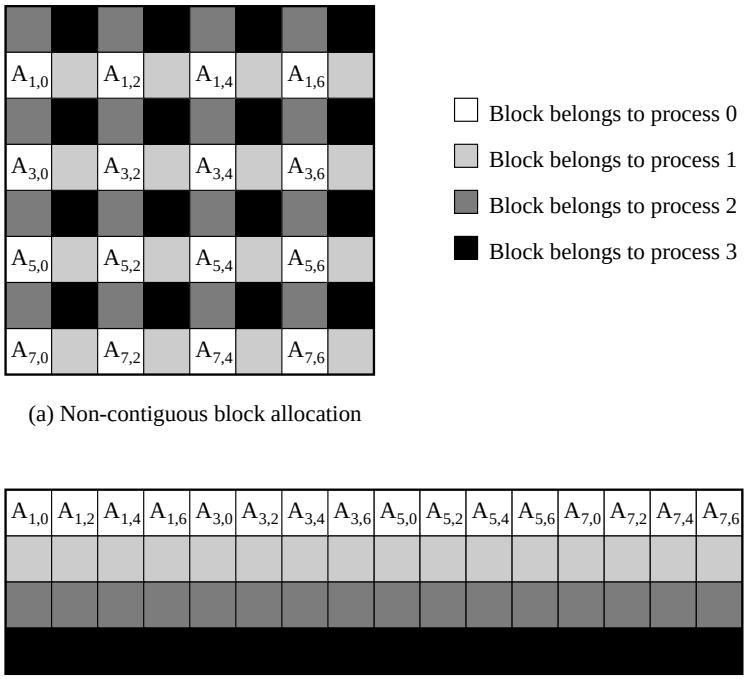
affected by block size; as block size increases, phase time will increase accordingly. Phases 2 and 3 are not only affected by matrix size and block size, but also by iteration number; these three determine the amount of computation required in each phase.

The influence that matrix size has on performance is rather obvious: the larger the matrix, the longer the execution.

The size of the blocks not only determines the time required to decompose diagonal blocks in the first phase of each iteration, it also determines the runtime behavior of the algorithm in terms of cache usage [24], and thus impacts the overall execution time. Quantification of the impact block size has on overall execution time for a 2048×2048 matrix can be seen in Figure 5.4.

There are two versions of LU decomposition within the SPLASH-2 distribution; their difference lies in the manner in which blocks are allocated in each processor's memory. The decomposition algorithm is the same for the two versions. The first and most straightforward version, referred to as non-contiguous block allocation, allocates a 2-dimensional array to store the matrix, assigning to each processor a set of blocks that are not contiguously laid out in memory. Part (a) of Figure 5.5 shows this matrix allocation technique. Blocks belonging to process 0 (labeled blocks in the Figure) are not allocated physically next to each other. Thus, process 0 makes memory accesses spanning all the area where matrix A is allocated.

The second method, called contiguous block allocation, assigns each process a set of blocks that are contiguously allocated in memory. The labeled blocks in part (b) of Figure 5.5 are contiguous in memory, thus process 0 accesses a smaller memory area. In theory, this method is more efficient than the first one, for it enhances data locality. The data shown



(a) Non-contiguous block allocation

(b) Contiguous block allocation

Figure 5.5: Different memory layouts of matrix A in LU decomposition. Blocks belonging to process 0 are labeled. Contiguous block allocation enhances data locality.

in Figure 5.4 confirms that contiguous block allocation renders better execution times than non-contiguous block allocation.

5.4 Water

Water is an N-body molecular dynamics simulation; it evaluates the forces and potentials that exist in a system of water molecules in the liquid state over a user-specified number of iterations or *time-steps* [20]. This loop is called the *molecular dynamics loop*. Gravitational forces and interactions between and within the molecules are calculated at every time-step and for every molecule. Also, the total potential energy of the system can be computed and output every user-specified number of time-steps. Ideally, the number of time-steps to perform should be set large enough to allow the system to reach a steady state. The default number of time-steps in the SPLASH distribution is 3, and the potential energy of the system is computed every three iterations. The available documentation does not specify how the number of time-steps should be adjusted for the larger data sets we use in this Chapter. For this reason, we execute the water program with 3 time-steps, even though it might mean that the system does not reach a steady state.

The SPLASH-2 implementation of water is an enhancement of the version in the first

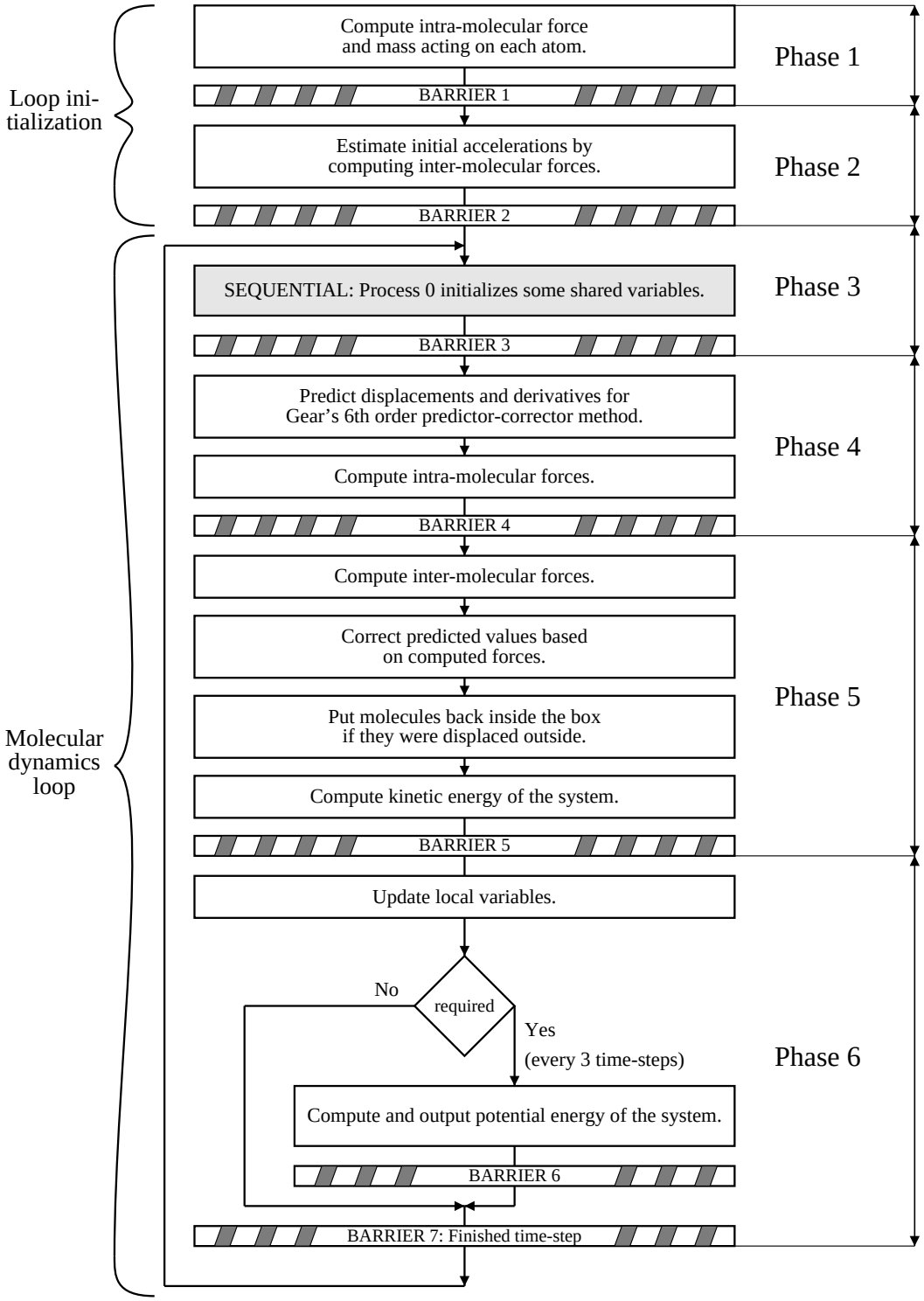


Figure 5.6: Water-n^2 tasks. Each rectangle represents a task. The molecular dynamics loop is executed for every time-step, as specified by the user.

SPLASH distribution. In turn, the SPLASH distribution contains a parallel re-implementation in C of the Fortran sequential version included in the Perfect Club [2] suite of benchmarks.

The enhancements implemented in SPLASH-2 do not change the algorithm; they are mainly concerned with reducing lock contention when updating inter-molecular forces and initial particle accelerations. Processes update local copies of the forces and the global copy is updated once at the end of every time-step, though each process only updates the locations it modified.

A new water algorithm with complexity $O(n)$, dubbed *water spatial*, is introduced in SPLASH-2. This new version results in the original version, which has complexity $O(n^2)$, to be named *water-*n^2. This Chapter focuses on water-n^2.

The input data set to water-n^2, a user-specified number of water molecules, is stored in an array of structures that contain molecule-specific data such as position, velocity, and direction. To preserve the accuracy of the simulation, the molecules are initially positioned in a cubical lattice rather than randomly distributed within the boundaries of an imaginary box. Note, however, that the fact that two molecules occupy neighboring positions in the array does not mean that they are spatially close to each other.

According to the documentation distributed with SPLASH, a random spatial distribution would not "preserve chemical intermolecular distance ranges". Nevertheless, a compile-time flag can be defined to allow the random spatial positioning of molecules. By default, this flag is not defined, thus we do not use it.

Additionally, the three atoms that compose each water molecule (H_2O: two hydrogen and one oxygen) are assigned initial velocities along the x, y, and z axes that are read from a file. Initial atom velocities are expected to have been randomly generated and in the range $(-4.0, 4.0)$ by default.

After all molecules are initialized, the program starts its parallel execution: processes are launched and the actual computation begins. The program consists of a sequence of *tasks*, as depicted in Figure 5.6. Parallelism is available both within and across tasks, meaning that individual tasks can be executed in parallel, and that within a phase it is possible to execute more than one task at the same time. For example, Phase 5 consists of 4 tasks that can be executed in parallel. Because each task is performed on all molecules, scheduling can be done statically. Processes are assigned equal size partitions of the array of molecules.

Before entering the molecular dynamics loop, intra-molecular forces and masses are estimated along with initial molecule accelerations. These two tasks make up Phases 1 and 2 of the parallel execution, respectively. After phases 1 and 2 are completed, the program enters the aforementioned molecular dynamics loop.

At every time-step, the program uses Gear's sixth-order predictor-corrector method to calculate molecule displacements based on the interactions of forces among and within them.

Application	*Data set*	*Total Wall-clock Times (in seconds)*		
		SBT_OFF	SBT_NO_DEBUG	SBT_WATCH_ALL
Radix Sort	16M	10.71	10.76	11.79
	32M	21.97	21.72	22.53
	64M	39.80	40.30	41.13
LU Decomposition	512×512	1.43	1.44	1.46
	1024×1024	8.80	8.83	8.87
	2048×2048	61.76	61.85	62.48
	4096×4096	458.21	458.72	460.52
Water-n^2	8000 molecules	66.04	66.37	66.66
	9261 molecules	86.91	87.09	87.36
	10648 molecules	115.32	115.48	115.59
	12167 molecules	149.41	149.18	148.74

Table 5.2: Total execution times of SPLASH-2 applications with SBT turned off and two different levels of tracing on 4 processors. Averages of 5 runs.

A high-level characterization of phases and the tasks that comprise them is shown in Figure 5.6.

5.5 Overhead of Using SBT

Gathering and outputting performance data inevitably incurs some overhead that programmers have come to accept as a price they pay in exchange for the information. SBT is no different than other performance monitors in this respect and also adds some overhead. This section quantifies the overhead by comparing *total wall-clock time* of production runs (i.e., without using SBT) and monitored runs with different levels of tracing turned on.

For this experiment, all applications are timed through the `time` program[2] to retrieve total wall-clock times. Total execution times taken from `time` include not only the actual computation times, but also those of initialization of SBT and the program's data structures.

Two versions of each program are compiled — one linked to the full version of SBT and the other linked to the faster production version of the library (i.e., SBT compiled with the `SBT_OFF` flag turned on). In addition, total execution times for the binaries linked to the full version of SBT are taken from two kinds of runs: one using `SBT_NO_DEBUG`, which produces no performance information output —although the binary is capable of doing it— and the other using `SBT_WATCH_ALL`, which watches all barriers and thus maximizes the amount of output from SBT. Additionally, the latter also included hardware performance counter information for total CPU cycles and L2 data cache misses.

Total execution times under the described conditions, shown in Table 5.2, prove that the overhead incurred by using SBT is negligible. The numbers shown in the table correspond to executions using the following data sets: radix size 1024, Gauss distribution; LU block

[2] We refer to the `time` program, which is found in `/usr/bin/time`, not the `time` built-in shell command.

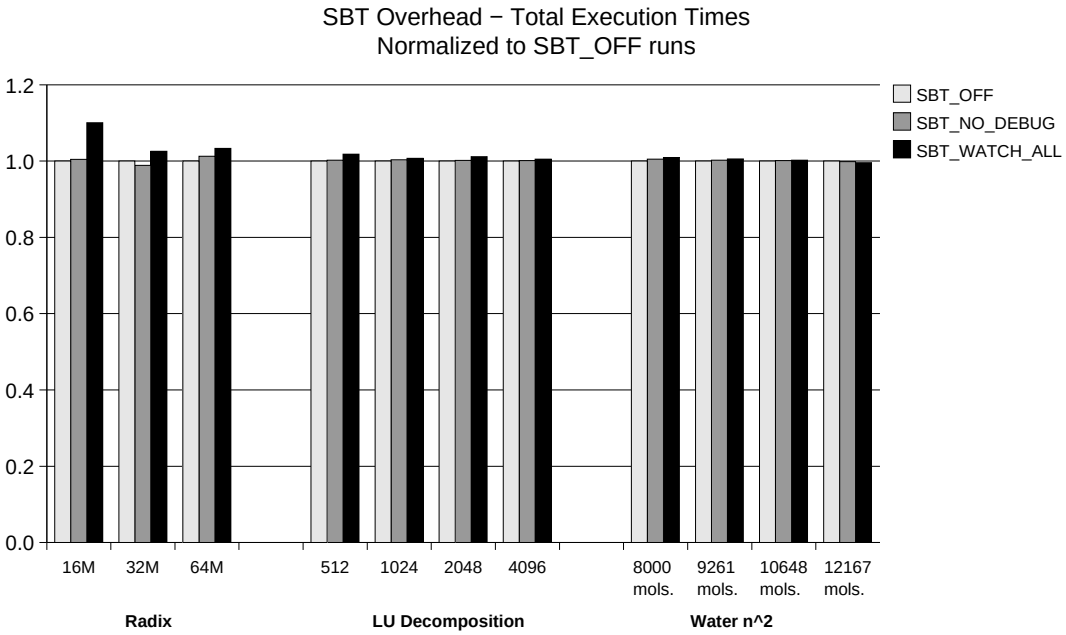


Figure 5.7: SBT overheads on different input data set sizes for radix, LU decomposition, and water-n^2. Normalized to runs using the lean version of the library (i.e., compiled with `SBT_OFF`).

size 32×32, contiguously allocated; water-n^2 as expected by the program. The instance that incurred the greatest absolute cost of all applications is LU decomposition of a 4096×4096 matrix, which increased its execution time by little more than 2 seconds, going from 458.21 for the `SBT_OFF` run to 460.52 for the `SBT_WATCH_ALL` run. However, the 2 seconds account for an increased time of only 0.5% over the `SBT_OFF` run.

The information shown in Table 5.2 is graphed on Figure 5.7, normalized to the production runs' total times. In terms of increased percentage of time, radix sort on 16 million keys suffers the most, with an increase of 10% going from the `SBT_OFF` run to the `SBT_WATCH_ALL` run. However, with a total execution time ranging between 10.7 and 11.8 seconds, a 10% increase is negligible.

In addition, as data set size increases, the overhead in terms of percentage of time is reduced: total time increased an average of only 3% from the production run to the fully instrumented run. For LU decomposition and water n^2 the overhead of using SBT is amortized over longer periods of time, thus reducing the relative differences between the different runs.

Differences between `SBT_NO_DEBUG` and `SBT_WATCH_ALL` are bigger than between `SBT_OFF` and `SBT_NO_DEBUG`. When `SBT_WATCH_ALL` is used, there is more information to process and output. Also, more system calls are involved when hardware counters are used.

In three cases, the instrumented versions reported total execution times slightly smaller

than the production versions. The radix sort of 32 million keys with SBT_NO_DEBUG is 250 milliseconds faster than the SBT_OFF version, and total execution times of water-n^2 with 12167 molecules speeds up as the level of instrumentation grows. These differences, though surprising, are explained by inherent imperfections in the measurement process. The fact that the differences are so small allows us to consider the times to be practically identical.

5.6 Phase Times Analysis

In order to direct the users' attention to the most time-consuming parts of a parallel program, SBT shows phase times, defined as the amount of time between the barrier at the beginning of a phase and the barrier at the end. This metric gives an answer to question 2 of Section 1.2. Users are quickly presented with a time breakdown of the execution that reflects the different phases of the program at hand. Furthermore, bottlenecks become more visible when phase times and thread inter-arrival times are correlated. Suppose that at the end of a phase, one process has a thread inter-arrival time that is as long as the phase time; it is clear that while the last process was working, the others had to wait at the barrier.

All three SPLASH-2 codes are iterative: they are comprised of a number of phases that are executed inside a loop; once the loop exits, the computation is completed. The use of loop barriers to delimit the phases in such algorithms saves a considerable amount of time that would otherwise be spent aggregating iteration-specific data. At the end of the execution, SBT conveys the cumulative data gathered throughout all iterations.

The information shown in this section for radix and LU decomposition is gathered by using loop barriers, and also by watching all barriers at runtime (i.e., SBT option SBT_WATCH_ALL set to 1). Whenever a conceptual phase is divided into sub-phases because it has barriers inside, the data for all sub-phases are aggregated and shown as if they belonged to one phase with only one barrier at the end. For example, to build a global histogram during phase 2 of radix sort, a barrier that divides the phase into two sub-phases is required; in this case, performance data for the two sub-phases and the two barriers (the one inside and the one at the end of the phase) are aggregated by hand and shown here in that form.

In contrast, information shown for water-n^2 is gathered using named barriers rather than loop barriers. This program has phases that are more complex than those of radix and LU, and defaults to executing three iterations. As a result, watching individual phases inside each of the three iterations can provide better insight and does not produce an overwhelming amount of information.

Also, barrier information is always associated with the phase that precedes the barrier. Each phase has a barrier marking its end that is identified with the same number as that of the phase. All phase time decomposition graphs are formed of stacked columns where the bottom block represents phase 1, the second block from bottom to top is barrier 1, the

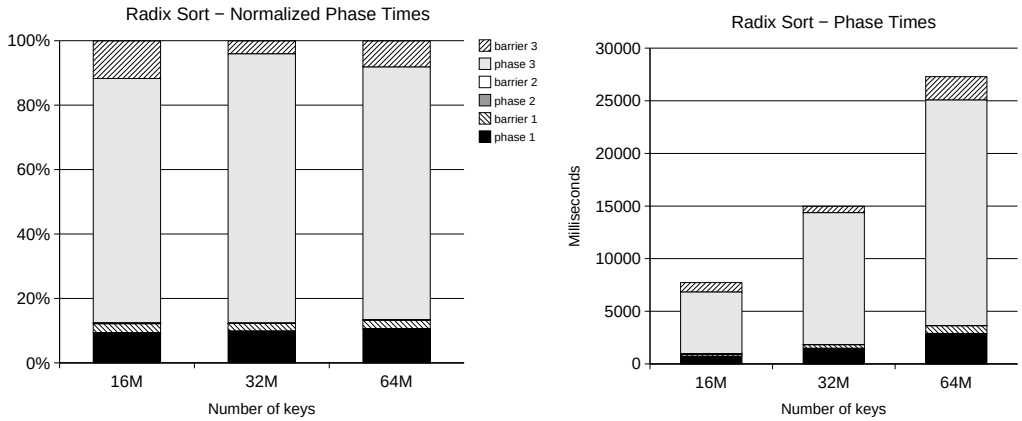


Figure 5.8: Radix phase time decomposition: 4 processors, radix 1024, Gauss distribution.

third block is phase 2, and so on.

5.6.1 Radix Sort

Aggregated phase times for radix sort are depicted on the right side of Figure 5.8 and in Table 5.3. Note that, although hardly visible, data for phase 2 and barrier 2 are shown in both graphs of the figure, between barrier 1 and phase 3. According to the normalized phase times shown on the left side of Figure 5.8, the first and second phases, *build local histograms* and *build global histogram*, respectively —along with their associated barriers— account for approximately an average 12% of the total execution time, whereas the third phase, *permute keys*, proves to be the most time consuming. Using these data points as a starting point, phase 3 might be the first target for optimization.

The barrier times for barrier 3 present an uneven trend across the three data set sizes. The normalized times depicted in Figure 5.8 tell us that barrier 3 takes approximately 15% of total time for 16 million keys, 5% for 32 million keys, and 10% for 64 million keys. However, the absolute barrier 3 times, shown in the last column of Table 5.3, present differences ranging from 310 milliseconds to 1610 milliseconds. These differences are small enough that they fit within measurement error.

There is an uneven pattern of barrier times for barrier 3. Two well-known characteristics of radix sort are reflected in the numbers of Table 5.3. First, shared memory implementations have the advantage of allowing a cheap computation of the global histogram during phase 2, especially when using a small number of processors. Second, permuting keys during phase 3 requires an expensive all-to-all communication [19].

Number of keys	*Time (milliseconds)*					
	Phase 1		*Phase 2*		*Phase 3*	
	Local hist.	*Barrier 1*	*Global hist.*	*Barrier 2*	*Permute*	*Barrier 3*
16M	730	210	10	10	5870	900
32M	1480	360	10	0	12520	610
64M	2890	740	10	10	21420	2220

Table 5.3: Radix phase and barrier times: 4 processors, radix 1024, Gauss distribution.

5.6.2 LU Decomposition

Although a barrier at the end of phase 3 of LU decomposition is not necessary,[3] adding a call to one of the SBT barrier macros at that point is useful to distinguish performance information relative to phases 1 and 3 of the algorithm. This extra barrier adds some overhead, but at the same time allows the extraction of more precise per-phase measurements.

As iterations are performed, the computation time required to complete phase 2 (*Update perimeter blocks*) and phase 3 (*Update interior blocks*) decreases. To illustrate this, a version of LU with named barriers, rather than loop barriers, is used. Figure 5.9 depicts output from SBT for a subset of the 64 iterations performed to decompose a 1024×1024 matrix on 4 processors using a block size of 16×16 elements. Phase numbers shown for each barrier in Figure 5.9 refer to the number of times SBT executed its barrier code, and not to phase numbers 1, 2, and 3 that comprise the algorithm. Phases 2 and 3 of the algorithm end at named barriers `"Done perimeter blocks"` and `"Done interior blocks"`, respectively.

On iteration 7 of the algorithm (lines 3 through 10 in Figure 5.9), phase 2 takes 0.011 seconds (line 7) and phase 3 takes 0.287 seconds (line 10). By the time iteration 35 is reached (lines 13 through 20), phase times for phases 2 and 3 are 0.005 and 0.076 seconds respectively (lines 17 and 20). On the last iteration of the execution, iteration number 64 (lines 23 through 30), timings for phases 2 and 3 shrink to 0 seconds (lines 27 and 30). In addition, note that the timings for phase 1 are constant throughout all iterations. Although in this case phase 1 is reported to have consumed 0 seconds in each iteration (lines 4, 14, and 24), larger block sizes will cause increased phase 1 timings.

The fact that phase times for phases 2 and 3 get smaller as the loop progresses, lets us be certain that the program is executing as expected. Recall that LU decomposition is an iterative algorithm that works from the "top left" corner of the matrix towards the "bottom right" corner (see Figure 3.7 in Section 3.2.2). As more iterations are performed, processes have less blocks to work on. We can use the phase time metric from SBT to verify that the program is executing as we expect.

Normalized phase times for LU decomposition show a typical granularity issue: as the

[3]Note that the same process that updates interior block $A_{(k+1)(k+1)}$ in iteration k will be responsible for factoring it in the next iteration, thus eliminating any potential data dependencies between phase 3 of iteration k and phase 1 of iteration $k+1$.

```
1     ...
2
3     SBT lu.c:690: "Done factor diagonal block"
4             (barrier: 0ms, phase 20: 0.000s, from init: 3.494s)
5
6     SBT lu.c:730: "Done perimeter blocks"
7             (barrier: 10ms, phase 21: 0.011s, from init: 3.505s)
8
9     SBT lu.c:772: "Done interior blocks"
10            (barrier: 37ms, phase 22: 0.287s, from init: 3.792s)
11    ...
12
13    SBT lu.c:690: "Done factor diagonal block"
14            (barrier: 0ms, phase 104: 0.000s, from init: 8.636s)
15
16    SBT lu.c:730: "Done perimeter blocks"
17            (barrier: 5ms, phase 105: 0.005s, from init: 8.642s)
18
19    SBT lu.c:772: "Done interior blocks"
20            (barrier: 11ms, phase 106: 0.076s, from init: 8.718s)
21    ...
22
23    SBT lu.c:690: "Done factor diagonal block"
24            (barrier: 0ms, phase 191: 0.000s, from init: 9.578s)
25
26    SBT lu.c:730: "Done perimeter blocks"
27            (barrier: 0ms, phase 192: 0.000s, from init: 9.579s)
28
29    SBT lu.c:772: "Done interior blocks"
30          (barrier: 0ms, phase 193: 0.000s, from init: 9.579s)
31    ...
```

Figure 5.9: Phases 2 and 3 (*Done perimeter blocks* and *Done interior blocks*, respectively) are executed faster in later iterations. SBT output from LU decomposition: 4 processors, 1024×1024 matrix, 16×16 blocks, iterations 7, 35, and 64 of 64.

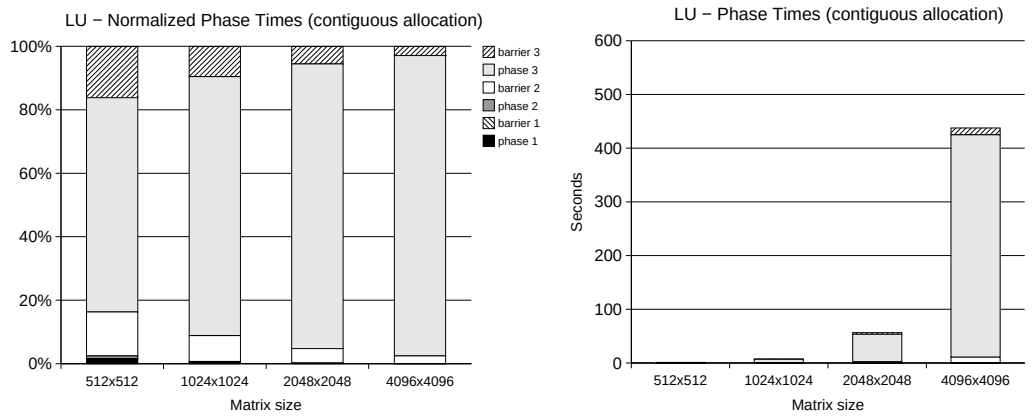


Figure 5.10: LU phase time decomposition: 4 processors, contiguous block allocation, block size 32×32 elements, matrix sizes ranging from 512×512 to 4096×4096.

	Aggregated Phase Times (seconds)		
Matrix Size	*Contiguous allocation*	*Non-contiguous allocation*	*% Time Increase*
512×512	0.70	1.00	43%
1024×1024	6.00	8.00	33%
2048×2048	50.80	66.70	31%
2048×2048	414.50	535.10	29%
		Average % Time Increase:	34%

Table 5.4: LU Decomposition: aggregated phase times and percentage increase of time going from contiguous to non-contiguous block allocation. Blocks of 32×32 elements.

data set size increases, the relative amount of time required for synchronization decreases. Figure 5.10 shows both absolute and normalized phase times decomposing matrices on 4 processes using 32×32 element blocks that are allocated contiguously for each process. The percentage of total execution time spent at the barriers decreases from 30% for a 512×512 matrix to 5% for a 4096×4096 matrix. These percentages are obtained by adding each barrier's corresponding percentage in the normalized times shown in Figure 5.10.

The relative distribution of time among phases when non-contiguous allocation of blocks is used (Figure 5.11) is very similar to that of contiguous allocation. However, the non-contiguous allocation method proves to hurt performance; phase times consistently increase by an average 34% over the contiguous method, as shown in Table 5.4.

5.6.3 Water

Water is a more complex program that Radix and LU decomposition. It spans more phases, three of which have sub-phases. Also, one of the sub-phases to phase 6 is enclosed inside an `if` statement and is only executed once. The potential energy of the whole system is computed in that sub-phase on the last iteration of the molecular dynamics loop.

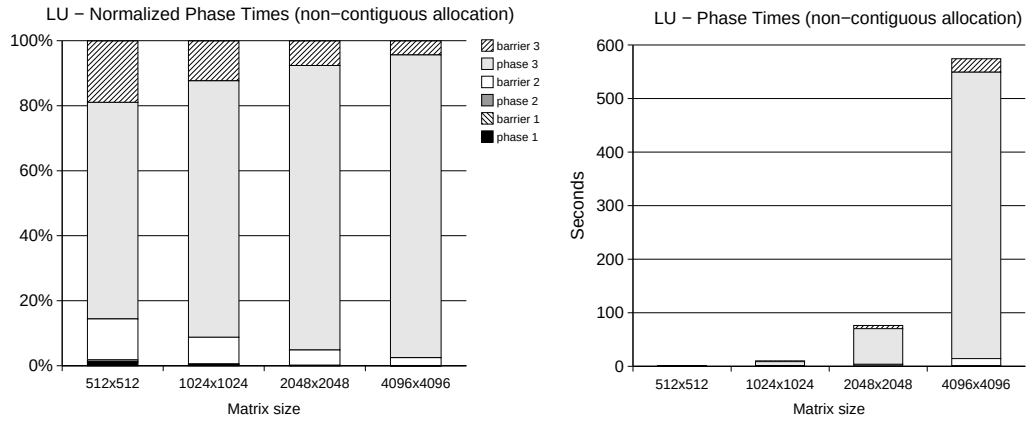


Figure 5.11: LU phase time decomposition: 4 processors, non-contiguous block allocation, block size 32×32 elements, matrix sizes ranging from 512×512 to 4096×4096.

Because the program defaults to performing only three iterations, named barriers are used for the experiments, rather than loop barriers. The benefit of using named barriers in this case is twofold: it allows us to confirm that phases in the molecular dynamics loop consume the same amount of time throughout all iterations, and it facilitates identification of the most time consuming task.

Figure 5.12 shows the output for the first two phases (before entering the molecular dynamics loop; lines 1 through 27 in the Figure), the first iteration of the loop (lines 28 through 65), and the sub-phase that computes the potential energy of the system on the last time-step (lines 67 through 80). The output is taken from a run on 12167 molecules, while watching all barriers. In order to fit the Figure on one page, thread inter-arrival times have been removed from the output, except where SBT issued barrier time warnings. Nevertheless, barrier times for those barriers range from 0 to 8 milliseconds, and thus thread inter-arrival times lack relevance.

There are two barriers that quickly attract attention to themselves because they exceed the default barrier time of 1000 milliseconds: named barriers `"Updated all forces"`, with warnings in lines 19 and 52 of Figure 5.12, and named barrier `"Computed potential energy"`, with warning in line 79. Also, these two barriers mark the end of phases that consume an average of 29 seconds each (see lines 9, 42, and 69 of the Figure). Furthermore, one of these phases is inside the molecular dynamics loop, hence it is executed 3 times. The sum of these average phase times over the complete run (4×29 seconds for `"Updated all forces"` + 29 seconds for `"Computed potential energy"`) results in approximately 145 seconds, where the total execution time is roughly 150 seconds: in line 77, the last process to arrive at the last barrier —process 3— does so 149 seconds after SBT is initialized at the beginning of the program (see column `"from init"` corresponding to the last barrier shown in Figure 5.12).

```
SBT barrier watch in mdmain.c:52 "Done phase 1"
        Barrier time:   13 ms
        Phase time:     0.692 sec
        Total time:     0.692 sec
...

SBT barrier watch in interf.c:196 "Updated all forces"
        Barrier time:   5860 ms
        Phase time:     30.347 sec
        Total time:     31.039 sec
        Order of arrival:
                        inter      from              real
                 id  thread      init              time
                  1     0ms     25.179s         23:35:09.610
                  0   593ms     25.772s         23:35:10.203
                  2   931ms     26.702s         23:35:11.133
                  3  4336ms     31.039s         23:35:15.470

SBT WARNING: barrier in interf.c:196 "Updated all forces"
        (barrier: 5860 ms > 1000 ms, phase 1: 30.347s, from init: 31.039s)

SBT barrier watch in mdmain.c:61 "Done phase 2"
        Barrier time:   8 ms
        Phase time:     0.009 sec
        Total time:     31.049 sec
...

SBT barrier watch in mdmain.c:107 "Done phase 3"
        Barrier time:   0 ms
        Phase time:     0.000 sec
        Total time:     31.050 sec
...

SBT barrier watch in mdmain.c:114 "Done phase 4"
        Barrier time:   1 ms
        Phase time:     0.028 sec
        Total time:     31.078 sec
...

SBT barrier watch in interf.c:196 "Updated all forces"
        Barrier time:   5401 ms
        Phase time:     29.765 sec
        Total time:     60.843 sec
        Order of arrival:
                        inter      from              real
                 id  thread      init              time
                  0     0ms     55.443s         23:35:39.873
                  1   127ms     55.570s         23:35:40.000
                  2  1510ms     57.080s         23:35:41.511
                  3  3764ms     60.843s         23:35:45.274

SBT WARNING: barrier in interf.c:196 "Updated all forces"
        (barrier: 5401 ms > 1000 ms, phase 5: 29.765s, from init: 60.843s)

SBT barrier watch in mdmain.c:176 "Done phase 5"
        Barrier time:   8 ms
        Phase time:     0.025 sec
        Total time:     60.869 sec
...

SBT barrier watch in mdmain.c:240 "Done phase 6; next time-step"
        Barrier time:   0 ms
        Phase time:     0.000 sec
        Total time:     60.869 sec
...

SBT barrier watch in mdmain.c:207 "Computed potential energy"
        Barrier time:   4975 ms
        Phase time:     28.976 sec
        Total time:     149.062 sec
        Order of arrival:
                        inter      from              real
                 id  thread      init              time
                  1     0ms     144.088s        23:37:08.518
                  0   149ms     144.237s        23:37:08.667
                  2  1379ms     145.615s        23:37:10.045
                  3  3447ms     149.062s        23:37:13.492

SBT WARNING: barrier in mdmain.c:207 "Computed potential energy"
        (barrier: 4975 ms > 1000 ms, phase 18: 28.976s, from init: 149.062s)
```

Figure 5.12: Water n^2 output on 12167 molecules watching all barriers.

The computation of inter-molecular forces takes place first during phase 2 and then, inside the molecular dynamics loop, during phase 5. A named barrier is required during such computation: named barrier `"Updated all forces"` is called inside a function implemented in file `interf.c`, according to SBT output in lines 7 and 40. This computation is performed by one function, `INTERF()`, which is always invoked immediately after barrier `"Done phase 1"` or `"Done phase 4"`, and contains the call to barrier `"Updated all forces"` before returning.

On the last time-step of the loop, the potential energy of the system is calculated as a task in phase 6. This calculation takes place in a way similar to that of inter-molecular forces; it is performed at the beginning of the phase and it has a barrier that marks its return.

Clearly, these two tasks suffer from a load imbalance problem. Thread inter-arrival times for barrier `"Updated all forces"`, shown in lines 12 through 17 and 45 through 50, are a good indication of the problem. During phase 1 of the program, process 3 accounts for 4 of the almost 6 seconds of the barrier time: line 8 of Figure 5.12 states that barrier time is 5860 milliseconds, and line 17 says that process 3 arrives 4336 milliseconds after process 2.

Named barrier `"Computed potential energy"` has similar results in lines 67 through 77: barrier time is 4975 milliseconds and process 3 arrives 3447 milliseconds after process 2.

All the data SBT produced for these two barriers and for the total execution are aggregated in Figure 5.13 to show the incidence of the two discussed tasks over total execution time. Note that for all data set sizes, the two tasks associated with barriers `"Updated all forces"` and `"Computed potential energy"`, take more than 80% of the total execution time.

There are a total of 2364 lines of code in this implementation of water. The function that calculates inter-molecular forces is 150 lines long, and function `POTENG()` —which calculates the potential energy of the system— has 138 lines of code. SBT allows us to quickly identify the 288 lines out of 2364 where we should initially focus any attempts to optimize this code.

5.7 Cumulative Idle Times via Loop Barriers

SBT's loop barriers report the cumulative idle time that each thread spent at the barrier during runtime. The range of thread idle times, measured from the minimum to the maximum of all the threads' idle times, can be used as a metric to diagnose poor load balancing. A wide range indicates that while one or more processes accumulate a large amount of idle time, others —which show lesser amounts of idle time— arrived consistently late.

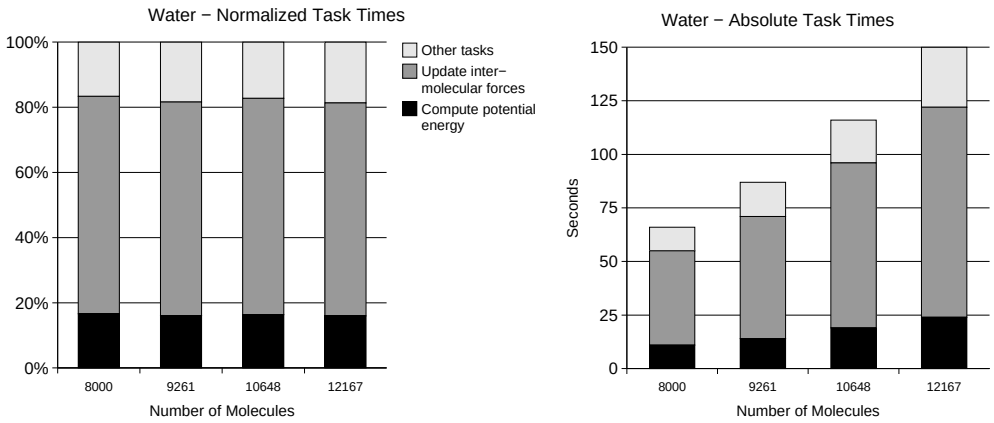


Figure 5.13: Water n^2 most time-consuming tasks.

5.7.1 Radix Sort

Not only is phase 3 of radix sort the most time-consuming, but it is also responsible for a notoriously wide range of thread idle times. In the case of the barrier at the end of phase 3, when sorting 64 million keys in Gauss distribution on 4 processors, cumulative thread idle times range from 165 to 1120 milliseconds, as depicted in the right-most column of Figure 5.14. This range is rather wide, compared with the negligible idle times at barrier 2, and a range of $[9, 120]$ milliseconds for the barrier at the end of phase 1.

Figure 5.14 is generated by running the program five times and calculating the average thread idle times at the loop barriers. Minimum and maximum idle times are extracted, and the mean and standard deviation (represented with Greek letter σ in the graph) are calculated. Three columns, representing the three loop barriers at the end of each phase, are shown for each data set size. Each column is as high as the mean of the thread idle times for the corresponding barrier.

5.7.2 LU Decomposition

Output from SBT for a 4-process run of LU decomposition shows uneven idle times at barrier 1 (`"Done factor diagonal block"`): those of processes 0 and 3 are considerably smaller than the ones for processes 1 and 2. Figure 5.15 shows partial SBT output for barrier 1 from a 4-process LU run on a 2048×2048 matrix using 64×64 element blocks. Since the output corresponds to a loop barrier, idle times for all processes are cumulative values throughout all iterations and are shown, ordered by process id, on the line entitled `"Thread idle times"`. SBT reports that processes 0 and 3 are idle at barrier 1 for a total of 96 and 97 milliseconds respectively, whereas processes 1 and 2 are idle for a total of 193 milliseconds each.

A repetition of the experiment with a version of LU that uses a named barrier, rather

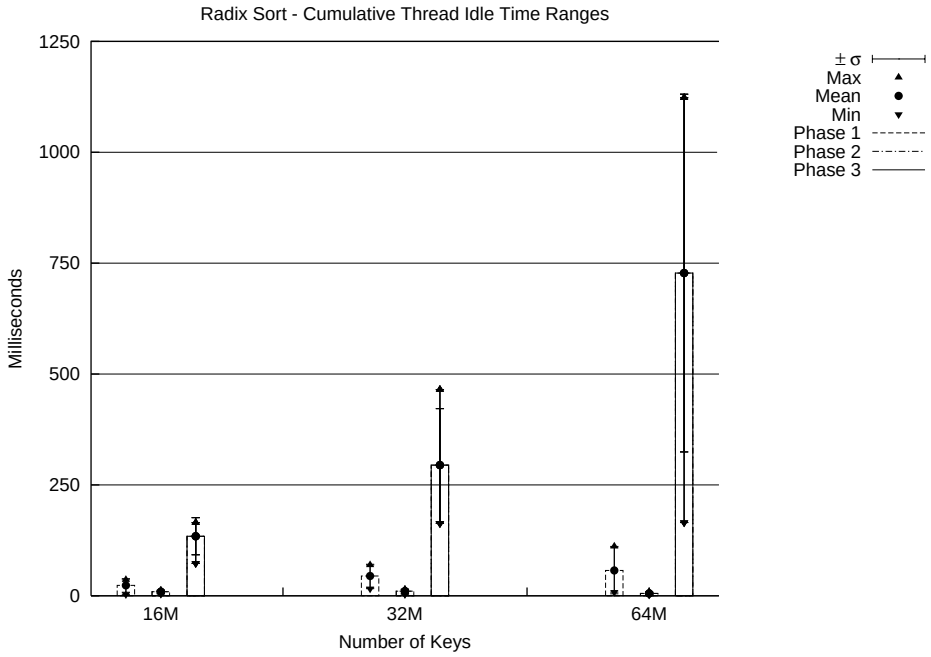


Figure 5.14: Radix loop barrier idle time ranges: minimum, mean, maximum, and standard deviation for all phases using different data set sizes. Radix size 4096, Gauss distribution, 4 processors.

```
Blocked Dense LU Factorization
    2048 by 2048 Matrix
    4 Processors
    64 by 64 Element Blocks

SBT options (version 0.9 built for sproc)
    SBT_WATCH       (null)
    SBT_WATCH_ALL   1
    SBT_WARNINGS    1
    SBT_WARN_TIME   1000
    SBT_PHASE_TIMES 0
    SBT_NO_DEBUG    0
...

SBT Loop Barrier lu.c:690 "Done factor diagonal block"
    Barrier time:   193 msec
    Phase time:     0.214 sec
    Subphase count: 32
    Thread idle times: [ 96 193 193 97 ] msec
...
```

Figure 5.15: Uneven thread idle times at barrier 1 (`"Done factor diagonal block"`). SBT output from LU decomposition using a loop barrier at the end of phase 1: 4 processors, 2048×2048 matrix, 64×64 blocks.

```
Blocked Dense LU Factorization
    2048 by 2048 Matrix
    4 Processors
    64 by 64 Element Blocks

SBT options (version 0.9 built for sproc)
    SBT_WATCH       (null)
    SBT_WATCH_ALL   1
    SBT_WARNINGS    1
    SBT_WARN_TIME   1000
    SBT_PHASE_TIMES 0
    SBT_NO_DEBUG    0
...

SBT barrier watch in lu.c:690 "Done factor diagonal block"
    Barrier time:   6 ms
    Phase time:     0.007 sec
    Total time:     6.138 sec
    Order of arrival:
             inter      from              real
        id  thread      init              time
        2     0ms      6.132s          18:10:25.539
        1     0ms      6.132s          18:10:25.539
        3     0ms      6.132s          18:10:25.539
        0     6ms      6.138s          18:10:25.545
...

SBT barrier watch in lu.c:690 "Done factor diagonal block"
    Barrier time: 6 ms
    Phase time: 0.007 sec
    Total time: 53.039 sec
    Order of arrival:
             inter     from              real
        id  thread     init              time
        0     0ms     53.032s         18:13:18.359
        2     0ms     53.032s         18:13:18.359
        1     0ms     53.032s         18:13:18.359
        3     6ms     53.039s         18:13:18.365
...

SBT barrier watch in lu.c:690 "Done factor diagonal block"
    Barrier time: 6 ms
    Phase time: 0.007 sec
    Total time: 61.481 sec
    Order of arrival:
             inter     from              real
        id  thread     init              time
        0     0ms     61.475s         18:15:31.824
        2     0ms     61.475s         18:15:31.824
        1     0ms     61.475s         18:15:31.824
        3     6ms     61.481s         18:15:31.830
...
```

Figure 5.16: Phase 1 of LU decomposition is sequential; threads 0 and 3 work alternately. SBT output from LU decomposition using a named barrier at the end of phase 1: 4 processors, 2048×2048 matrix, 64×64 blocks, iterations 1, 16, and 32 of 32.

than a loop barrier, to mark the end of phase 1, confirms that only processes 0 or 3 take more than 0 milliseconds to complete phase 1. Using a named barrier produces phase-specific output on *every* iteration, which in this case is useful; it illustrates the fact that in a phase that lasts approximately 7 milliseconds, there is one process (either 0 or 3) that consistently arrives at barrier 1 an average of 6 milliseconds later than the others. This kind of output, which can be seen in the thread inter-arrival times depicted in Figure 5.16, along with the idle times shown in Figure 5.15, is not surprising.

There are two causes for the uneven idle times at barrier 1. First, this implementation of LU decomposition uses a 2D-block work distribution technique, as described in Section 5.3. Running on 4 processes, all diagonal blocks of the matrix belong to either process 0 or process 3 (e.g., process 0 owns block $A_{0,0}$, process 3 owns block $A_{1,1}$. See Figure 5.3). Thus, only processes 0 and 3 are responsible for performing computations on the diagonal blocks of the matrix.

The second cause for the uneven idle times is the nature of phase 1, during which a single block is factored by one process. It is inherently sequential and takes constant time, dependent on block size. The conclusion to be drawn after seeing the output from SBT and linking it to the nature of phase 1 is that it takes each processor approximately 6 milliseconds to factor a matrix of 16×16 elements.

5.8 Hardware Counters

The ability to associate hardware event counter information with the specific phases and processes of a parallel program can help developers identify and understand some of the reasons for the bottlenecks that may occur. For example, a poorly balanced phase will probably generate uneven CPU cycle counts for the different processes of the program. Also, false sharing can be diagnosed from large data cache miss counts; small modifications such as field padding of the application's data structures could enhance cache usage.

SBT can output hardware event counts for every phase and process of a parallel program. The specific events that can be counted depend on two factors: the platform and the hardware counter library SBT is linked to. All hardware counter information shown in the following sections comes from a version of SBT linked to the PAPI library [3].

Since the experiments are executed on an SGI machine, SBT could have been linked to `libperfex` [4]. The reason for not using `libperfex` is that it allows a maximum of 2 events to be counted at a time. The MIPS processors have two event counter registers, and `libperfex` does not multiplex them (even though `perfex`, the program, does). This limits both the number and the possible combinations of events that can be counted together on a single run. Most `perfex` events can be counted only on one of the two hardware event counter registers.

In contrast, PAPI uses techniques originally designed for MPX [13], a library that implements software multiplexing of counter registers. Multiplexing is a technique that allows a fixed number of event counter registers to count any number of hardware events. The registers are time shared between the different events, so multiple events can be monitored on a single register. Although multiplexing the registers introduces some error in the obtained counts, the error is reported to be "within a few percent of counts recorded without multiplexing".

Using PAPI for hardware performance counters gives us the freedom to count more events and to use any combination of them.

5.8.1 Radix Sort

Phase times analysis of radix indicates that the program spends most of its time executing phase 3, and it can be concluded that any efforts to optimize performance should be directed toward that phase. The reasons that this phase is so expensive can be found by understanding the algorithm, as well as by looking at hardware event counts. However, because performance information extracted from SBT also reflects other interesting characteristics of radix, our focus is not set exclusively on phase 3.

A useful metric to measure the amount of work that is invested in a phase is the number of CPU cycles. For example, the tree-summing algorithm used in phase 2 to calculate the global histogram is reflected by the CPU cycle counts that the individual processes have. Only two processes in a 4-process execution are responsible for accumulating ranks and densities. The data graphed in Figure 5.17, which corresponds to a data set of 64 million keys in Gauss distribution, shows that processes 1 and 3 complete phase 2 with more than twice the number of CPU cycles required by the other two processes. Also, the number of CPU cycles executed in phase 2 increases with radix size. On each iteration, densities and ranks are computed for all digits smaller than the radix.

Not surprisingly, phase 3 accounts for an average 83% of the total number of CPU cycles when the data set has the Gauss distribution and includes 64 million keys (Figure 5.18). In contrast, a data set of equal size but with the zero distribution, which has a greater number of duplicate keys and thus an easier permutation phase, averages 72% of the total CPU cycles in phase 3.

A more revealing metric in this case is the number of TLB misses: more than 99% of the total count occurs in phase 3, during which processes perform an all-to-all communication as they write keys to each other's memory areas [23]. Scattered writes such as these are usually a cause for high numbers of TLB misses, which are in turn responsible for increased memory latencies, due to their high cost of recovery. Although the amount of TLB misses is dependent on radix size, the proportion of misses occurring in phase 3 remains constant.

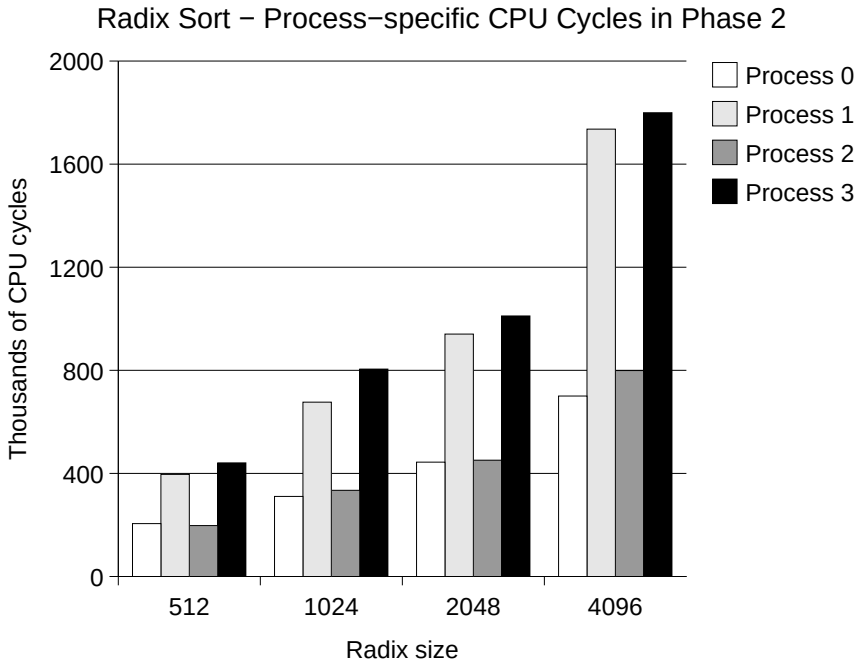


Figure 5.17: Tree-summing algorithm used in phase 2 of radix causes an uneven load balance. Total per-process CPU cycle count on phase 2 Radix Sort: 64 Million keys, Gauss distribution.

Figure 5.19 shows the average TLB misses across 4 processors for phase 3 and the entire execution, using different radix sizes. In general, SBT allows us to collect hardware event counts for individual threads and, within each thread, for individual phases.

5.8.2 LU Decomposition

Similarly to what happens in radix sort with TLB misses, the last phase of LU decomposition is responsible for most of the L2 data cache misses. Using contiguous block allocation on a 2048×2048 matrix with 32×32 blocks, phase 3 accounts for 96% of the total misses; the non-contiguous block allocation version of LU decomposing the same matrix has a phase 3 with 97% of the L2 data cache misses.

However, the two versions are quite different in terms of the absolute numbers of misses. According to the data illustrated in Figure 5.20, the non-contiguous version averages a total count that is approximately four and a half times larger than that of the contiguous allocation version.

Going a step further in profiling the contiguous allocation version of LU, we present total execution times and associate them with average L1 and L2 cache misses across 4 processes, using different block sizes. Table 5.5 shows the absolute numbers and Figure 5.21 presents the same data normalized to the fastest case: 32×32 elements per block.

The differences in execution time, with respect to the fastest case, can be understood by

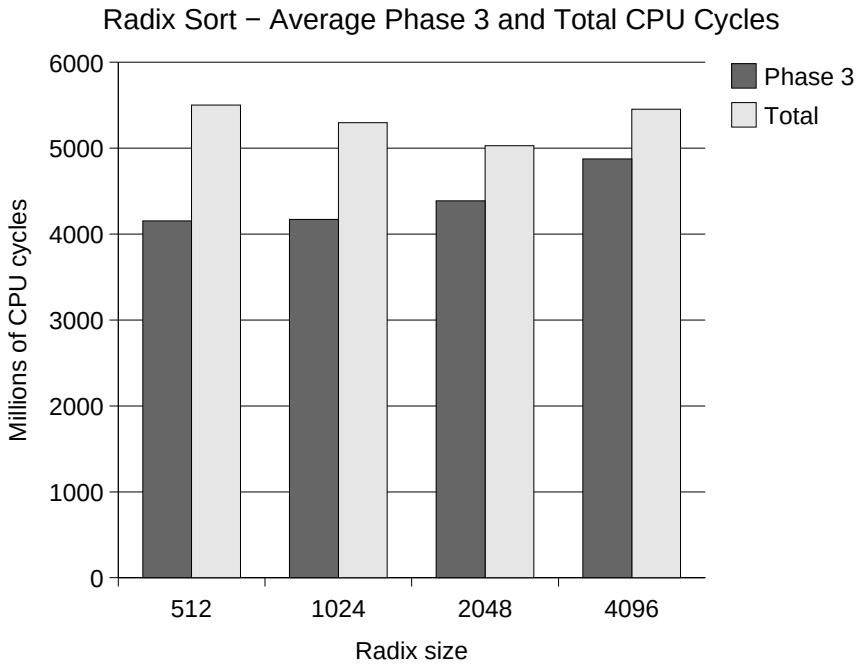


Figure 5.18: Phase 3 and total average CPU cycles across 4 processes for radix sort: 64 Million keys, Gauss distribution.

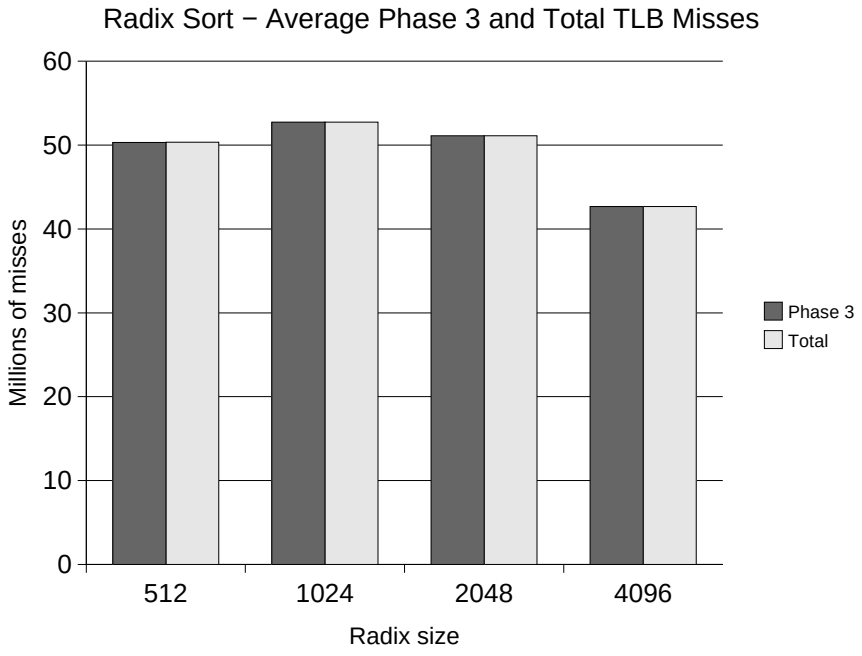


Figure 5.19: Phase 3 and total average TLB misses across 4 processes for radix sort: 64M keys, Gauss distribution.

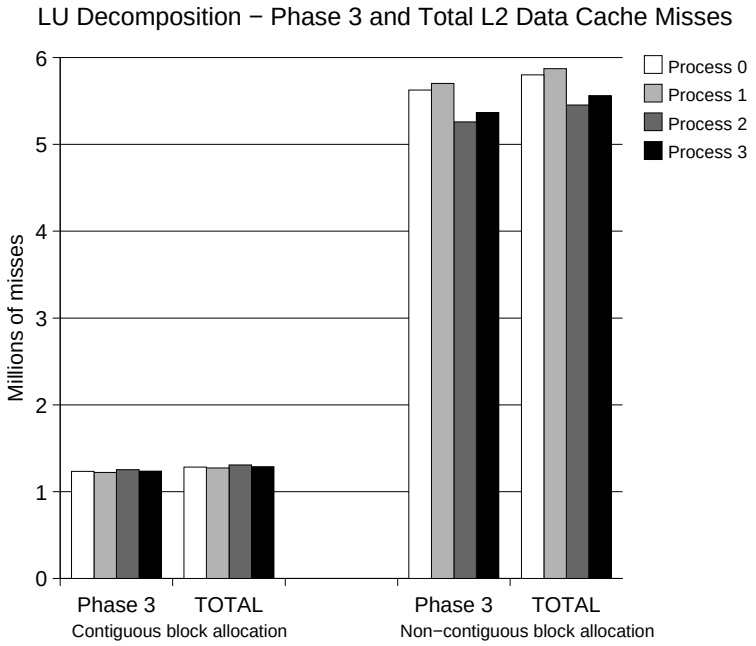


Figure 5.20: Most L2 data cache misses in LU occur during phase 3. Phase 3 and total L2 data cache misses for LU decomposition: 2048×2048 matrix, 32×32 element blocks.

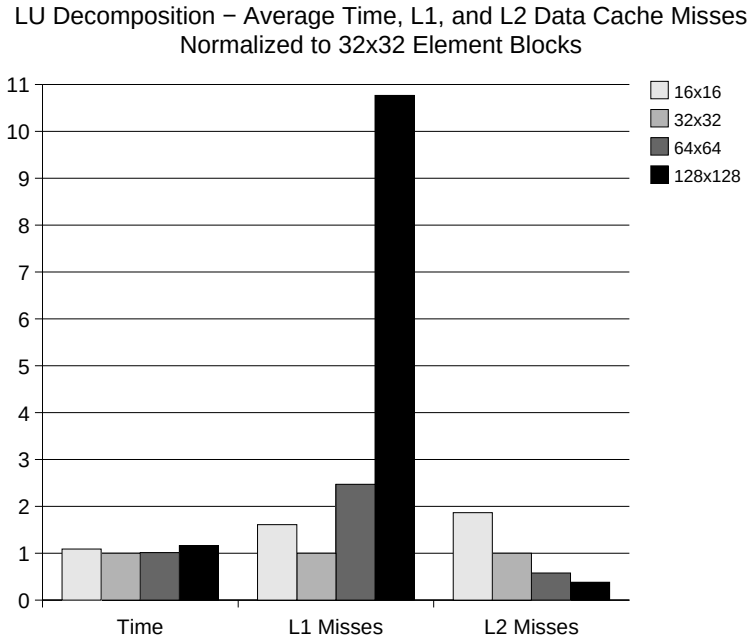


Figure 5.21: LU decomposition on a 2048×2048 matrix. Time, L1 data cache misses, and L2 data cache misses. Averages across 4 processors, normalized to the fastest case: 32×32 blocks.

Block Size	*Total Time*	*L1 Misses*	*L2 Misses*
16×16	60.78 *sec*	27902098	2386933
32×32	55.72 *sec*	17321724	1278722
64×64	56.41 *sec*	42798289	737524
128×128	64.72 *sec*	186519978	487020

Table 5.5: LU total execution times associated with L1 and L2 misses.

looking at the heights of the columns in Figure 5.21. First, using blocks of 16×16 elements, the program incurs the largest number of L2 misses. The cost of recovering from the L2 misses and a rather large number of L1 misses justify the longer time required by the 16×16 version when compared to the faster 32×32 version. The absolute execution time difference between the two versions, depicted in column *Total Time* of Table 5.5, is approximately 5 seconds.

Second, although the number of L2 misses in the run using 64×64 element blocks is roughly 40% less than that of the 32 element case, the increased number of L1 misses neutralizes any significant gains. Nevertheless, this case is slower by less than a second, which is negligible.

Finally, the program flounders when it uses 128×128 element blocks, even though the number of L2 misses in this case is the smallest. The MIPS R12000 processors have an internal L1 cache of 32 KB, and one block of 128×128 elements requires 128 KB of storage; each element, of type double, occupies 8 bytes — thus blocks require $128 \times 128 \times 8$ bytes. Since the computation revolves around blocks, not being able to fit one in the L1 cache results in an excessive number of misses.

Woo *et al* state in [23] that block size should be set to a value "large enough to keep the cache miss rate low, and small enough to maintain good load balance". Taking advantage of the information produced by SBT, we can quickly establish not only the most appropriate value, but also the underlying reasons why that value results in better performance — in this case, L1 and L2 cache misses.

5.8.3 Water

The discussion of water-n^2 phase times in Section 5.6.3 points at two tasks as the most noticeable targets for analysis and potential performance enhancements: *Compute intermolecular forces* and *Compute potential energy of the system.* In this Section, we focus exclusively on hardware event counts for these two tasks. In particular, we present the hardware event counts that can be associated with the load imbalances present in the tasks.

Hardware counter information from SBT for the two tasks is presented in Figure 5.22. The output is from one 4-process run on 12167 molecules and shows task-specific counts of total cycles (`PAPI_TOT_CYC`) and branch mispredictions (`PAPI_BR_MSP`). Named barrier

```
...

SBT barrier watch in interf.c:196 "Updated all forces"
...

        SBT: Phase 1 PAPI counter information
 id    PAPI_TOT_CYC     PAPI_BR_MSP
  0      8483592760       165792738
  1      8541582150       168836968
  2      9060876457       195927860
  3     10248921691       249633949
...


SBT barrier watch in interf.c:196 "Updated all forces"
...

        SBT: Phase 5 PAPI counter information
 id    PAPI_TOT_CYC     PAPI_BR_MSP
  0      8470485996       165715822
  1      8519781937       168788155
  2      9056615061       196167317
  3     10249533693       250318951

...


SBT barrier watch in interf.c:196 "Updated all forces"
...

        SBT: Phase 10 PAPI counter information
 id    PAPI_TOT_CYC     PAPI_BR_MSP
  0      8470654856       165731966
  1      8515238523       168798834
  2      9057146231       196159718
  3     10249941820       250336705
...


SBT barrier watch in interf.c:196 "Updated all forces"
...

        SBT: Phase 15 PAPI counter information
 id    PAPI_TOT_CYC     PAPI_BR_MSP
  0      8470165781       165677203
  1      8515669157       168800974
  2      9057188895       196171695
  3     10249360649       250314394

...


SBT barrier watch in mdmain.c:207 "Computed potential energy"
...

        SBT: Phase 18 PAPI counter information
 id    PAPI_TOT_CYC     PAPI_BR_MSP
  0      8312655964       164681922
  1      8348982466       168223616
  2      8891872343       196012543
  3     10096806482       250926401
...

        SBT: Overall accumulated PAPI counter information
 id    PAPI_TOT_CYC     PAPI_BR_MSP
  0     42258975693       828495010
  1     42492662381       844351859
  2     45165322033       981333326
  3     51135018831      1252427850
```

Figure 5.22: Hardware counter information for barriers `"Updated all forces"` and `"Computed potential energy"`. Water n^2 output on 12167 molecules.

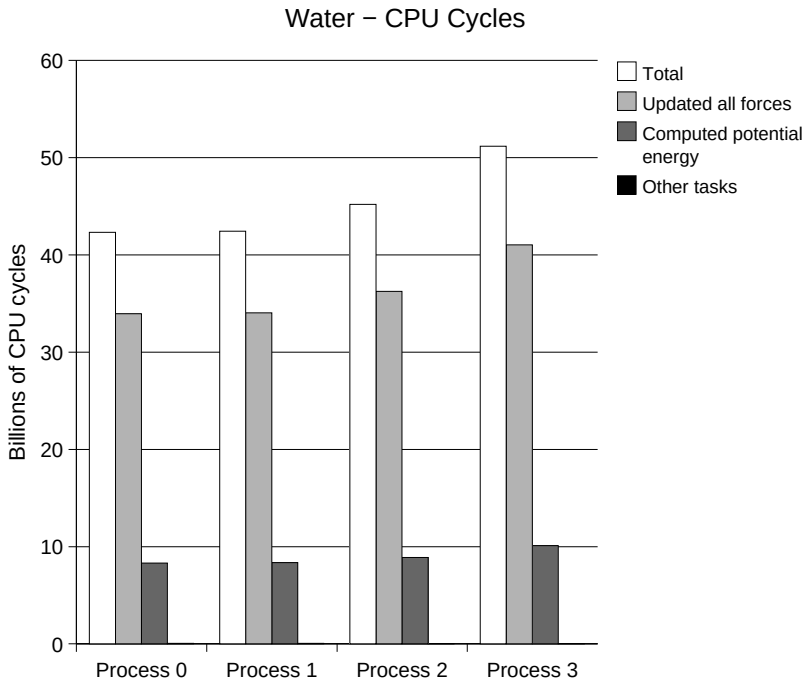


Figure 5.23: CPU cycles for water-n^2 on 12167 molecules. The most computationally-intensive tasks are the calculations of inter-molecular forces and of potential energy.

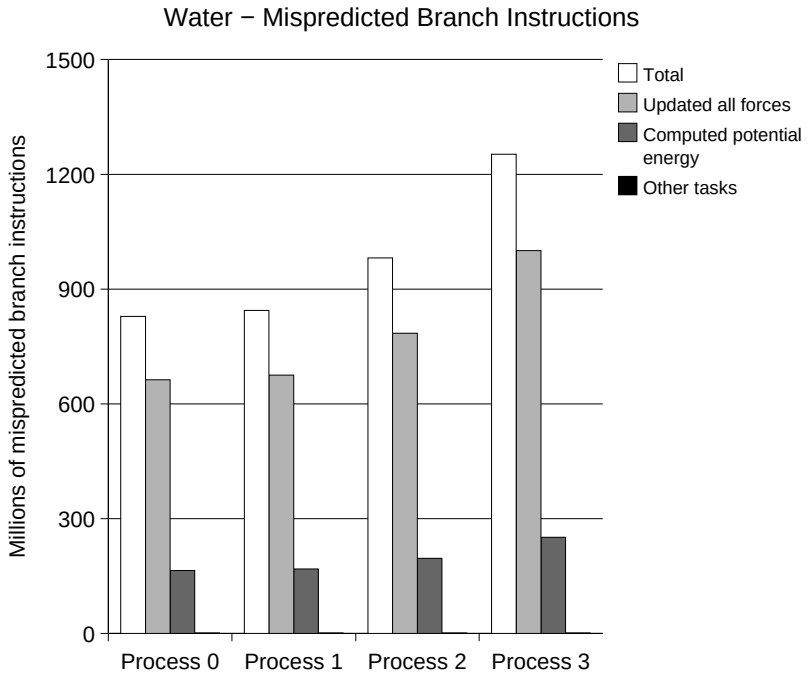


Figure 5.24: Mispredicted branches for water-n^2 on 12167 molecules. The most computationally-intensive tasks are the calculations of inter-molecular forces and of potential energy.

`"Updated all forces"` is invoked once in phase 2 (lines 3 through 11) and once per iteration in phase 5 (lines 15 through 48); barrier `"Computed potential energy"` is called only once, at the end of the last iteration of the molecular dynamics loop (lines 53 through 61). Also shown in the Figure are the total number of occurrences of each of the two events for the complete execution (lines 64 through 69). Although barrier times, phase times, thread inter-arrival times, and barrier time warnings have been removed from this Figure, they can be seen in Figure 5.12.

The output in Figure 5.22 can be used to produce graphs such as those in Figures 5.23 and 5.24.

The fact that different processes execute different numbers of CPU cycles indicates a probability that the processes do not execute the same code. Furthermore, the numbers of mispredicted branch instructions on different processes present similar differences: processes 0 and 1 require fewer cycles and incur fewer mispredictions than processes 2 and 3. This is true for both tasks.

In fact, functions `INTERF()` and `POTENG()` have several nested loops guarded with conditions that are always true for processes 2 and 3, and false for processes 0 and 1. Moreover, the differences in the numbers of events counted for processes 2 and 3 are also explained by the span of the loops; process 3 is always responsible for larger numbers of iterations than process 2.

5.9 Concluding Remarks

Parallel programs in SPMD style can be easily modified to use SBT barriers. It is just a matter of including the SBT header file, defining two macros, and using one of the SBT barrier macros when barrier synchronization is necessary. The SPLASH-2 suite, a well-known collection of SPMD programs for shared memory, provides good examples to demonstrate SBT's usage and capabilities.

Using SBT with some of the SPLASH-2 programs, we are able to identify the most computationally-intensive phases, localize bottlenecks, and analyze load balancing issues. SBT produces the information necessary to answer the questions stated in Section 1.2. Additionally, the information generated by SBT can be summarized in graphs and tables to facilitate its interpretation.

Another important point demonstrated in this Chapter is SBT's low probe effect. The library produces valuable performance information incurring overheads of less than 10%.

Chapter 6

Conclusion

Debugging and performance-tuning parallel programs are difficult tasks. Developers need to be aware of the inherent added complexities of parallel programming: synchronization, shared data dependencies, deadlock, etc. When several processes interact with each other, some bugs may be difficult to reproduce, and they often have to be tracked across all interacting processes. Once a parallel program executes correctly, it is desirable that it will do so in the least amount of time possible. Parallel debuggers and performance profiling tools are valuable and helpful during the process, but they suffer from a series of disadvantages that may render them inconvenient to use.

First, debuggers may require interactive access to the program, which may not be convenient, or even possible. In many cases, multi-processor systems are batch-scheduled; users submit their jobs to a queue and have little chance to attach a debugger to the executing program. Performance profilers, on the other hand, do not necessarily require user interaction while measuring performance; a trace file can be generated as the program executes, and can be processed later.

Second, trace-based performance profiling tools insert some kind of instrumentation into the original code. The cost of the instrumentation (i.e., its probe effect) may be excessive, and result in skewed measurements. Furthermore, the amount of trace data generated may be more than necessary, leaving the user with a large amount of information that takes time to filter and process.

Finally, it is not always possible to have access to the right tools. Most specialized debuggers and profilers are not known for their portability, for they generally produce information on specific platforms. Also, they are often unappealing to the open-source community with their price tags and inaccessible source code.

In addition to debuggers and profilers, it is desirable to provide the programmer with a tool that —with little probe effect— generates dynamic and on-line performance data, is easy to use, and is portable. SBT is a library possessing such characteristics, that can be used to debug and profile shared memory parallel programs in SPMD style.

6.1 SBT Summary

By taking advantage of barrier calls that are already present in the code, SBT inserts instrumentation to produce debugging and profiling information. Using SBT barriers, a SPMD parallel program generates a low-noise trace of its execution at runtime. The tool can be used either interactively, to confirm that the program advances through the phases or to quickly identify the most computationally-intensive phase; or off-line, to analyze the produced information in more detail.

The cost of the inserted instrumentation is negligible. We observed overheads in the range of 1% to 10% of total execution time. In addition, a non-instrumented version of the library can be built through conditional compilation, and used for production runs. In this case, SBT barriers are implemented as direct calls to an underlying barrier primitive.

A wide selection of user-configurable options allows the programmer to focus the debugging and monitoring on a specific part of the program. While all barriers can be watched during execution, programmers can also direct SBT to produce information for a specific barrier. To provide additional flexibility, SBT options can be set as environment variables or command-line options. Furthermore, SBT produces neatly formatted output that is easy to identify and interpret.

SBT can monitor parallel SPMD programs that use POSIX threads or Irix `sprocs`, which makes it portable to several platforms. The fact that it is capable of interfacing with three different hardware counter libraries adds to its flexibility and portability. SBT has been successfully used under two platforms: Irix using Pthreads and `sprocs`, and Linux using Pthreads.

Traces generated by SBT help programmers locate where their shared memory programs are spending their time (or are hung due to a deadlock), and provide insight as to why there may be performance problems. The beginning and end of each computational phase is conveniently labelled and automatically measured. When deadlocks occur, it is clear in which phase they are located. When bottlenecks occur, the programmer can watch the relevant barrier to gather more performance information.

Important metrics, such as phase time, barrier time, and thread inter-arrival time at a barrier are automatically gathered by SBT. If the metrics are outside of an expected range, warnings are generated and the programmer is made aware of a possible problem. Moreover, users can gain more insight into issues such as load balancing and data locality by directing SBT to generate hardware performance counter information.

In summary, SBT is a portable, lightweight, and easy to use library that can be utilized for on-line debugging and profiling SPMD programs. It gathers data to inform the programmer about the runtime behavior of a program: location of deadlocks, most computationally-intensive phase, bottlenecks, and load imbalance.

6.2 Future Work

As a lightweight debugging and monitoring tool, SBT should not grow in complexity in such a way that it degenerates into a costly instrumentation and produces an unnecessarily large amount of data. The initial design goal is to develop a tool that quickly provides its users with useful information. Users should be able to access this information without large modifications to their code. Nevertheless, some features can be added without betraying the original goal.

In the future, SBT will be extended to support distributed memory environments through the use of standard message-passing libraries, such as the Message-Passing Interface (MPI) [21]. There is a large community of parallel programmers writing code that uses MPI to communicate between processes. Also, clusters of workstations are becoming a commonplace platform for the development of parallel applications.

In situations in which a conceptual phase of an algorithm is divided into sub-phases, it would be beneficial to be able to tell SBT where the phase starts and where it ends. With user-defined start and end barriers for a phase, SBT could start accumulating data for that phase at the start barrier. Until the end barrier is reached, SBT would accumulate all gathered information, regardless of the presence of other barriers inside the phase. When the program reaches the end barrier, aggregated data for the whole conceptual phase would be output.

A sequential version of SBT, capable of producing phase-specific information (e.g., phase time, hardware event counts), will be developed. Programmers frequently set caliper points in their code using `printf()` or some other more sophisticated technique. By doing this, they hope to keep track of what their programs are doing, or whether certain points of the execution have been reached.

6.3 Availability

The current version of SBT, along with documentation and a short tutorial, can be found at: `http://www.cs.ualberta.ca/~paullu/SBT/`

Bibliography

[1] R. Berrendorf and H. Ziegler. PCL - The Performance Counter Library: A Common Interface to Access Hardware Performance Counters on Microprocessors (Version 1.3). Technical report, Central Institute for Applied Mathematics, Research Centre Juelich GmbH, 1999.

[2] M. Berry, D. Chen, P. Koss, D. Kuck, S. Lo, Y. Pang, L. Pointer, R. Roloff, A. Sameh, E. Clementi, S. Chin, D. Scheider, G. Fox, P. Messina, D. Walker, C. Hsiung, J. Schwarzmeier, K. Lue, S. Orszag, F. Seidl, O. Johnson, R. Goodrum, and J. Martin. The PERFECT Club Benchmarks: Effective Performance Evaluation of Supercomputers. *The International Journal of Supercomputer Applications*, 3(3):5–40, 1989.

[3] S. Browne, J. Dongarra, N. Garner, G. Ho, and P. Mucci. A Portable Programming Interface for Performance Evaluation on Modern Processors. *The International Journal of High Performance Computing Applications*, 14(3):189–204, Fall 2000.

[4] D. Cortesi et al. *Origin2000 (TM) and Onyx2 (TM) Performance Tuning and Optimization Guide.* Silicon Graphics, Inc., 1998.

[5] D. Cortesi et al. *Topics in IRIX Programming.* Silicon Graphics, Inc., 2000.

[6] L. Dagum and R. Menon. OpenMP: An Industry-Standard API for Shared-Memory Programming. *IEEE Computational Science & Engineering*, 5(1):46–55, January/ March 1998.

[7] Etnus LLC. TotalView Multiprocess Debugger/Analyzer. `http://www.etnus.com/Products/TotalView/`.

[8] M. Gerndt, B. Mohr, and B. Miller. Performance Analysis and Tuning of Parallel Programs: Resources and Tools. In *Tutorial at Supercomputing 2000*, Dallas, Texas, USA, November 2000.

[9] Silicon Graphics Inc. *SpeedShop User's Guide.* Silicon Graphics, Inc., 2000.

[10] Los Alamos National Laboratory: Advanced Computing Laboratory. TAU: Tuning and Analysis Utilities, November 1999. Supercomputing 1999 flyer.

[11] P. Lu. Aurora: Scoped Behaviour for Per-Context Optimized Distributed Data Sharing. In *Proceedings of the 11th International Parallel Processing Symposium*, pages 467–473, Geneva, Switzerland, April 1997. Available at `http://www.cs.ualberta.ca/~paullu/`.

[12] P. Lu. *Scoped Behaviour for Optimized Distributed Data Sharing.* PhD thesis, University of Toronto, Toronto, Ontario, Canada, January 2000.

[13] J. May. MPX: Software for Multiplexing Hardware Performance Counters in Multithreaded Programs. In *Proceedings for the 15th International Parallel and Distributed Processing Symposium (IPDPS)*, San Francisco, California, USA, April 2001.

[14] C. E. McDowell and D. P. Helmbold. Debugging Concurrent Programs. *ACM Computing Surveys (CSUR)*, 21(4):593–622, December 1989.

[15] B. P. Miller, J. M. Cargille, R. B. Irvin, K. Kunchithapadam, M. D. Callaghan, J. K. Hollingsworth, K. L. Karavanic, and T. Newhall. The Paradyn Parallel Performance Measurement Tools. *IEEE Computer*, 28:37–46, November 1995. Special issue on performance evaluation tools for parallel and distributed computer systems.

[16] B. Mohr, D. Brown, and A. Malony. TAU: A Portable Parallel Program Analysis Environment for pC++. In *Proceedings of CONPAR 94 – VAPP VI*, pages 29–40, University of Linz, Austria, September 1994.

[17] B. Mohr, A. D. Malony, and J. E. Cuny. TAU. In Gregory V. Wilson and Paul Lu, editors, *Parallel Programming Using C++*. MIT Press, 1996.

[18] E. Novillo and P. Lu. On-Line Debugging and Performance Monitoring Using Barriers. In *Proceedings for the 15th International Parallel and Distributed Processing Symposium (IPDPS)*, San Francisco, California, USA, April 2001. Available at http://www.cs.ualberta.ca/~paullu/SBT.

[19] H. Shan and J. P. Singh. Parallel Sorting on Cache-coherent DSM Multiprocessors. In *Proceedings Supercomputing '99*, Portland, Oregon, USA, November 1999.

[20] J. P. Singh, W. Weber, and A. Gupta. SPLASH: Stanford Parallel Applications for Shared-Memory. *Computer Architecture News*, 20(1):5–44, March 1992.

[21] M. Snir, S.W. Otto, S. Huss-Lederman, D.W. Walker, and J. Dongarra. *MPI: The Complete Reference*. MIT Press, Cambridge, Massachusetts, USA, 1996.

[22] Paradyn Project Team. *Paradyn Parallel Performance Tools - User's Guide*. Computer Science Department - University of Wisconsin, Madison, Wisconsin, USA, 2001.

[23] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The SPLASH-2 Programs: Characterization and Methodological Considerations. In *Proceedings of the 22nd International Symposium on Computer Architecture*, pages 24–36, Santa Margherita Ligure, Italy, June 1995.

[24] S. C. Woo, J. P. Singh, and J. L. Hennessy. The Performance Advantages of Integrating Block Data Transfer in Cache-Coherent Multiprocessors. In *Proceedings of the Sixth International Conference on Architerctural Support for Programming Languages and Operating Systems (ASPLOS-VI)*, pages 219–229, San Jose, California, USA, October 1994.

[25] Z. Xu, J. R. Larus, and B. P. Miller. Shared-Memory Performance Profiling. In *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPOPP-97)*, volume 32, 7 of *ACM SIGPLAN Notices*, pages 240–251, New York, New York, USA, June 18–21 1997. ACM Press.

[26] M. Zagha and G. E. Blelloch. Radix sort for vector multiprocessors. In *Proceedings Supercomputing '91*, pages 712–721, November 1991.

Appendix A

SBT Options

The following options are read by SBT upon initialization and will dictate the behavior of the library during runtime. SBT options can be set as shell environment variables —before the program is executed— or as command-line arguments — when the program is executed.

`SBT_WATCH`: When set to an integer i, SBT will watch and display information for the barrier that is called in line i of the source file. If, in turn, it is set to a string, SBT will watch the barrier named with that string.

`SBT_WATCH_ALL`: When set to 1, SBT will output information on every barrier in the code. Default value is 0.

`SBT_WARN_TIME`: This is an integer that indicates the maximum amount of time (in milliseconds) a barrier can take before a warning is displayed. Default value is 1000.

`SBT_WARNINGS`: When set to 0, no barrier time warnings will be displayed, otherwise (i.e. any value different than 0) a warning will be issued for all barriers that take more than `SBT_WARN_TIME` milliseconds. Default value is 1.

`SBT_PHASE_TIMES`: Phase times are displayed after every barrier if this variable is set to 1. Phases are numbered starting from 0, and Phase 0 always refers to everything that happens between the call to `sbt_init()` and the first barrier. In this way, Phase 0 is considered to be the initialization phase. Default value is 0.

`SBT_NO_DEBUG`: When this variable is set to 1, all SBT processing and output is suppressed. The only overhead incurred by SBT when this option is set is caused by checking the current value of the option. It is useful after all development and debugging are done as a means of reassuring that the program is behaving as expected. Default value is 0.

`SBT_OPTIONS`: This is a flag to indicate SBT to print all its options and their values upon library initialization; it will also print the threading model for which the library was

built. Additionally, if SBT is linked to a hardware event counter library, it will print out all the events that are set to be counted. Default value is 1.

`SBT_EVENTS`: Colon separated list of PCL or PAPI events that will be counted. SBT will print a table containing phase counter values at every watched barrier, and a table with total accumulated values throughout all phases on library finalization (i.e., when `sbt_finalize()` is called).

`SBT_VERBOSE`: If set to 1, this variable will make SBT print out a short description of all its environment variables, and, if linked to PAPI or PCL, the list of events that are available for the current architecture. Default value is 0.

Appendix B

SBT Library Reference

SBT provides the following macros and functions as interface between the user and the library. They are presented in the same format as standard Unix man pages.

B.1 BARRIER

Anonymous barrier macro.

Synopsis

```
#include <sbt_barrier.h>
BARRIER;
```

Description

`BARRIER` indicates a barrier capable of gathering performance information. The output from this kind of barrier is identified by the source file name and line number where the macro is invoked. `BARRIER` is substituted with the following call to `sbt_barrier()`:

```
sbt_barrier(SBT_BARRIER, SBT_THREADID, __FILE__, __LINE__, NULL, 0)
```

B.2 L_BARRIER

Loop barrier macro.

Synopsis

```
#include <sbt_barrier.h>
L_BARRIER;
```

Description

`L_BARRIER` indicates a barrier called within a loop. Information for loop barriers is gathered and accumulated throughout the iterations, and is only output upon library finalization (e.g., when `sbt_finalize()` is called). This macro is substituted with the following call to `sbt_barrier()`:

```
sbt_barrier(SBT_BARRIER, SBT_THREADID, __FILE__, __LINE__, name, 1)
```

B.3 N_BARRIER()

Named barrier macro.

Synopsis

```
#include <sbt_barrier.h>
N_BARRIER(char* name);
```

Description

`N_BARRIER(name)` indicates a barrier which output will be preceded with the file name, line number, and barrier name (as per the `name` parameter). It is substituted with the following call to `sbt_barrier()`:

```
sbt_barrier(SBT_BARRIER, SBT_THREADID, __FILE__, __LINE__, name, 0)
```

B.4 SBT_BARRIER

Identifies the specific `sbt_barrier_t*` data structure to be used.

Synopsis

```
#define SBT_BARRIER my_barrier
#include <sbt_barrier.h>
sbt_barrier_t* my_barrier;
```

Description

`SBT_BARRIER` must identify the variable of type `sbt_barrier_t*` that will be used implicitly by `sbt_barrier()`. Structure `sbt_barrier_t` wraps either an sproc barrier or an implementation of barriers for Pthreads. Additionally, it is formed of fields where information is gathered and accumulated. This macro has to be defined before `sbt_barrier.h` is included.

B.5 SBT_MSEC

Get difference, in milliseconds, between two time structures.

Synopsis

```
#include <sbt_barrier.h>
SBT_MSEC(struct timeval t1, struct timeval t2);
```

Description

SBT_MSEC returns an int containing the amount of milliseconds between t1 and t2. This macro is substituted with:

```
((t2.tv_sec-t1.tv_sec)*1000+(t2.tv_usec-t1.tv_usec)/1000)
```

B.6 SBT_SEC

Get difference, in seconds, between two time structures.

Synopsis

```
#include <sbt_barrier.h>
SBT_SEC(struct timeval t1, struct timeval t2);
```

Description

SBT_SEC returns a double containing the amount of seconds between t1 and t2. This macro is substituted with:

```
((double)(t2.tv_sec-t1.tv_sec)+(double)(t2.tv_usec-t1.tv_usec)/(double)1e6)
```

B.7 SBT_THREADID

Specifies the function or variable that provides the current thread's id.

Synopsis

```
#define SBT_THREADID my_id
#include <sbt_barrier.h>
int my_id;
```

Description

SBT_THREADID is used by sbt_barrier() to gather and output thread-specific information. Each thread must have a numerical identity between 0 and $n-1$, where n is the total number of threads. This macro has to be defined before sbt_barrier.h is included.

B.8 SBT_TIME

Print time to stdout.

Synopsis

```
#include <sbt_barrier.h>
SBT_TIME(struct timeval t);
```

Description

SBT_TIME prints time t, followed by a newline character, to stdout. This macro is substituted with:

```
printf( "%.2d:%.2d:%.2d.%3.3d\n",
        localtime( &t.tv_sec )->tm_hour,
        localtime( &t.tv_sec )->tm_min,
        localtime( &t.tv_sec )->tm_sec,
        t.tv_usec/1000 )
```

B.9 sbt_barrier()

Synchronize threads before continuing.

Synopsis

```
#include <sbt_barrier.h>
void sbt_barrier( sbt_barrier_t* b,
                  int thdid,
                  char* file,
                  int line,
                  char* name,
                  int loop );
```

b	Data structure to hold barrier information.
thd_id	Numerical identity of current thread.
file	Name of the file in which the call is made.
line	Line number where the call is made.
name	Barrier name.
loop	Boolean value to indicate a loop barrier.

Description

sbt_barrier() is implicitly invoked by BARRIER, N_BARRIER(), and L_BARRIER to synchronize all threads at a certain point of the execution. Upon entering it, each thread gathers and accumulates information to be output as indicated by the user through SBT options. When sbt_barrier() is invoked by the mentioned macros, its parameters take the following default values:

b	SBT_BARRIER
thd_id	SBT_THREADID
file	__FILE__
line	__LINE__
name	name parameter passed to N_BARRIER(), otherwise NULL.
loop	With L_BARRIER, 1, otherwise 0.

Users are encouraged to avoid using this function directly and instead make use of the macros BARRIER, N_BARRIER(), and L_BARRIER. If SBT is compiled with SBT_OFF, this function calls the underlying barrier construct and no information is gathered or output.

B.10 sbt_finalize()

Free resources and output overall accumulated information.

Synopsis

```
#include <sbt_barrier.h>
void sbt_finalize(void);
```

Description

`sbt_finalize()` should be called at the end of the execution to free resources and output any accumulated information. This information consists of overall accumulated hardware counting data (only if SBT is linked to one of the supported hardware counting libraries), and loop barriers cumulative data.

B.11 sbt_init()

Initialize SBT.

Synopsis

```
#include <sbt_barrier.h>
```

When using Pthreads:

```
sbt_barrier_t* sbt_init(int count, int argc, char** argv);
```

When using sprocs:

```
sbt_barrier_t* sbt_init(int count, usptr_t* arena, int argc, char** argv);
```

Description

`sbt_init()` allocates and initializes all necessary memory to gather and accumulate information at runtime (including initialization of whatever hardware counter library is used). This function allocates and initializes an `sbt_barrier_t` data structure that will be used to hold the underlying barrier structure, as well as performance information. If the library is compiled with `SBT_OFF`, no memory is allocated other than that required by the underlying barrier. As well, `sbt_init()` reads the environment variables and command line arguments to configure the library's behavior according to user requirements.

In both versions of the function the `count` parameter refers to the number of threads that will be synchronized. The other two common parameters —`argc` and `argv`— are used to process SBT options passed as command line arguments. Options set through the command line override those set through environment variables.

The `sproc` implementation requires a fourth parameter —`arena`. This parameter can be either a valid arena or `NULL`. In the latter case, SBT will create its own arena and sprocs will be attached to it as they reach the first barrier. If a valid arena is provided, SBT assumes that all sprocs are already attached to it.

If no errors are encountered during library initialization, `sbt_init()` returns an initialized `sbt_barrier_t*`, else `NULL` is returned. The returned value should be assigned to `SBT_BARRIER`.