A Case Study of Selected SPLASH-2 Applications and the SBT Debugging Tool

Ernesto Novillo and Paul Lu
Department of Computing Science
University of Alberta
Edmonton, Alberta, T6G 2E8
Canada
{ernie|paullu}@cs.ualberta.ca

Abstract

SBT is portable library and tool for on-line debugging and performance monitoring of 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.

To demonstrate SBT's applicability and usefulness, we present a simple, case study performance analysis using three programs from the SPLASH-2 suite. 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.

Keywords: *parallel computing, debugger, performance tuning, SPMD, barrier, SPLASH-2, hardware counters*

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 [4, 5]. 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. 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., [7, 12]). However, they are often based on detailed and voluminous trace generation and post-processing, or they can have high runtime overheads. 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.

We have developed the SBT library [9, 8] to support some simple but useful on-line debugging and performance monitoring of shared-memory, single-program-multiple-data (SPMD) programs. 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.

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 (possibly redirected to a file) 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 [6], should be kept at a minimum.

3. *Ease of use.* Once the instrumentation is inserted, the

```
 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 1. Anonymous, Named, and Loop Barriers in SPMD Programs.

```
Barrier "Start Initialization" reached.
...
-- normal program output --
...
Barrier "End Initialization" reached.
...
-- normal program output --
...
Barrier "End iteration" reached.
...
```

Figure 2. Sample output for the program in Figure 1. Barriers are natural caliper and watchpoints.

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.

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.

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. Failing to see output for a particular barrier after a period of program inactivity, the programmer can detect a deadlock in the current phase. Also, SBT produces the information necessary to identify computationally-intensive phases and the existence of a load imbalance. Then, 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) [2], and Irix's `libperfex` [3].

2. Overview of SBT

We have developed the SBT library in order to provide the user with simple, on-line debugging and performance information. 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.

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 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 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 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. 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 1 is a call to a loop barrier. 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 1 might produce the output depicted in Figure 2. Each barrier, seen as a caliper point, informs the user that a certain phase of the execution has been completed. A more detailed output example will be discussed as part of Figure 4.

After compiling their parallel program to use SBT, users can identify the barrier they wish to watch by setting an environment variable (i.e., `SBT_WATCH`) before execution 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.

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 parameterized by user-controlled environment variables or command line options. The flexibility, configurability, and completeness (e.g., the various performance metrics, automatic warnings) of SBT's functionality is what gives it value above and beyond what many programmers already do with `printf()`s in their code.

3. 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 [11]. To illustrate SBT's usefulness, this section 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 paper 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. These SPLASH-2 applications were selected because, pragmatically, they were the easiest codes to work with. The performance measurements shown in this section are all averages calculated from the output of five 4-process runs. 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. For further details on the applications themselves, we refer the reader elsewhere [11, 8].

3.1 Radix Sort

The parallel version of radix, a sorting algorithm first invented more than a century ago [13], 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.

3.2 LU Decomposition

Another kernel included with the SPLASH-2 suite is parallel LU decomposition. This kernel decomposes a matrix into its LU form. The SPLASH-2 implementation of parallel LU distributes work to processes using a 2D-block scheme, by which the matrix is divided into square blocks along both axes, and blocks are assigned to processes in an interleaved fashion. 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.

3.3 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*. 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.

The program consists of a sequence of *tasks* (or algorithmic phases) and 7 barriers. 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.

4. 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[1] 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. Normally, a user can use environment variable `SBT_WATCH` to turn on detailed performance monitoring at selected barriers (named or anonymous), but setting `SBT_WATCH_ALL` activates detailed monitoring of all barriers.

Total execution times under the described conditions, shown in Table 1, 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 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. 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.

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

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

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

5. Phase Times Analysis

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. Users are quickly presented with a time breakdown of the execution that reflects the different phases of the program at hand (discussed further as part of Figure 4). 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. 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 third block is phase 2, and so on.

5.1 Radix Sort

Phase times for radix sort are in Table 2. According to the normalized phase times shown in Figure 3, 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.

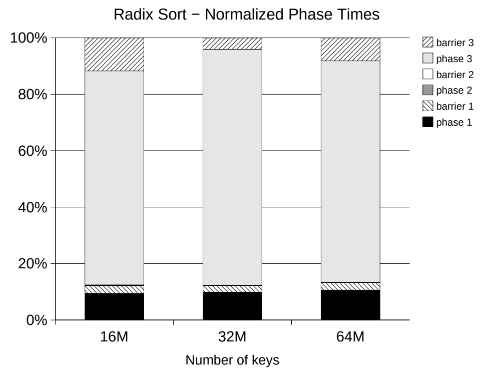

Figure 3. Radix phase time decomposition: 4 processors, radix 1024, Gauss distribution.

The barrier times for barrier 3 present an uneven trend across the three data set sizes. The normalized times depicted in Figure 3 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 2, present

```
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 4. Example Phase-by-Phase Trace: Water n^2 output on 12167 molecules watching all barriers.

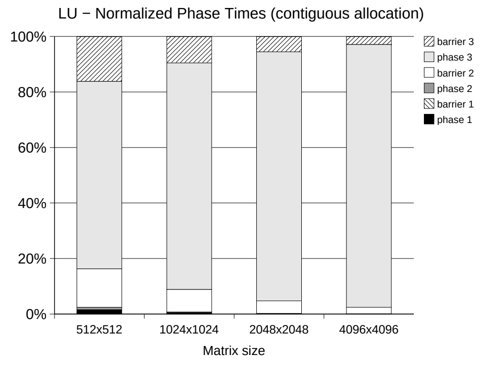

Figure 5. LU phase time decomposition: 4 processors, contiguous block allocation, block size 32×32 elements, matrix sizes ranging from 512×512 to 4096×4096.

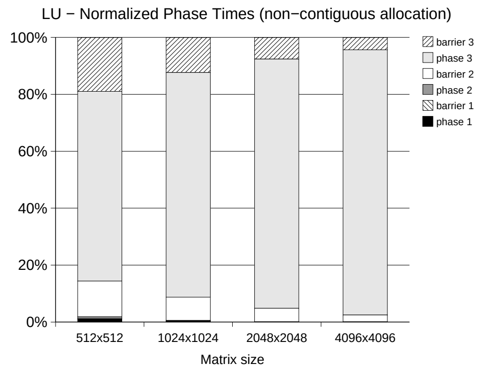

Figure 6. LU phase time decomposition: 4 processors, non-contiguous block allocation, block size 32×32 elements, matrix sizes ranging from 512×512 to 4096×4096.

differences ranging from 310 milliseconds to 1610 milliseconds. These differences are small enough that they fit within measurement error.

Two well-known characteristics of radix sort are reflected in the numbers of Table 2. 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 [10].

5.2 LU Decomposition

LU decomposition is an iterative algorithm that works from the "top left" corner of the matrix towards the "bottom right" corner. Each iteration consists of three phases: *Factor diagonal block* (phase 1), *Update perimeter block* (phase 2), and *Update interior blocks* (phase 3) [8].

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

Although a barrier at the end of phase 3 is not necessary, adding a call to an SBT barrier 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 and phase 3 decreases. The fact that these two phases get smaller as the loop progresses allows us to be certain that the program is executing as expected. As more iterations are performed, processes have fewer blocks to work on. We can use the phase time metric from SBT to verify this fact.

Normalized phase times for LU decomposition show a typical granularity issue: as the data set size increases, the relative amount of time required for synchronization decreases. Figure 5 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.

The relative distribution of time among phases when non-contiguous allocation of blocks is used (Figure 6) 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 3.

5.3 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 4 shows the output for the first two phases (before entering the molecular dynamics loop; lines 1 through 27) and the first iteration of the loop (lines 28 through 65). 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 4, 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 4).

All the data SBT produced for the total execution are aggregated in Figure 7. 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.

6. Concluding Remarks

Debugging and performance-tuning parallel programs are difficult tasks. 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, an SPMD parallel program generates a low-noise trace of its execution at runtime. The cost of the inserted instrumentation is negligible. We observed overheads in the range of 1% to 10% of total execution time. In

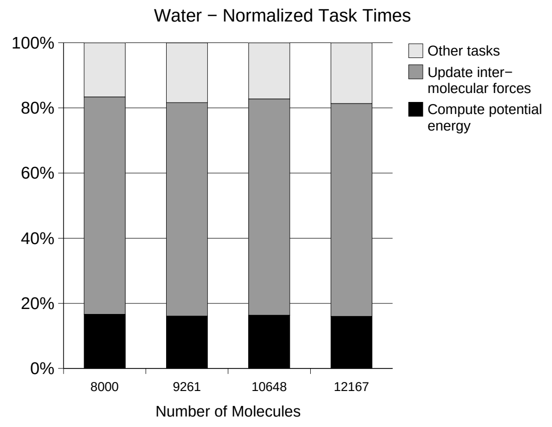

Figure 7. Water n^2 most time-consuming tasks.

addition, a non-instrumented version of the library can be built through conditional compilation, and used for production runs.

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.

Using three SPLASH-2 applications as a case study, we have shown the low overheads of using SBT. Also, we have shown how, in particular, the phase-by-phase analysis (made easier by SBT) can allow a parallel programmer to quickly focus on the portions or phases of a program that consume the most execution time.

The existing barriers in an SPMD program are natural instrumentation and caliper points. It is possible, of course, to do phase-by-phase analysis without using SBT (e.g., via calls to `gettimeofday()` and `printf()`), but SBT makes it much more convenient to insert the instrumentation (via named and anonymous barriers) and to control the amount of debugging output associated with each barrier (via watchpoints and environment variables). Furthermore, it is possible to use low-level hardware performance counters (discussed elsewhere [8]) to further tune the performance of a program.

References

[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] 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.

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

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

[5] 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.

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

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

[8] E. Novillo. On-line performance monitoring of shared memory parallel programs. Master's thesis, University of Alberta, 2002.

[9] 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.

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

[11] 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.

[12] 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.

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