Chapter 6
HPC Profiling with the Sun Studio™ Performance Tools

Marty Itzkowitz and Yukon Maruyama

Abstract In this paper, we describe how to use the Sun Studio Performance Tools to understand the nature and causes of application performance problems. We first explore CPU and memory performance problems for single-threaded applications, giving some simple examples. Then, we discuss multi-threaded performance issues, such as locking and false-sharing of cache lines, in each case showing how the tools can help. We go on to describe OpenMP applications and the support for them in the performance tools. Then we discuss MPI applications, and the techniques used to profile them. Finally, we present our conclusions.

6.1 Introduction

High-performance computing (HPC) is all about performance. This paper describes the various techniques implemented in the Sun Studio Performance Tools to profile HPC applications. We first describe how users can recognize the symptoms of performance problems. We then discuss problems common to single-threaded programs, and then go on to describe additional issues that manifest in multi-threaded programs. We then describe the characteristics of two of the main HPC programming models, OpenMP and MPI, and review the specific performance issues with each, and show how the tools can help.

6.1.1 The Sun Studio Performance Tools

The Sun Studio Performance Tools are designed to collect performance data on fully optimized and parallelized applications written in C, C++, Fortran, or Java,

Marty Itzkowitz, Yukon Maruyama
Sun Microsystems, 16 Network Circle, Menlo Park, CA 94025, USA
e-mail: {marty.itzkowitz, yukon.maruyama}@oracle.com

M.S. Müller et al. (eds.), *Tools for High Performance Computing 2009*,
DOI 10.1007/978-3-642-11261-4_6,

and any combination of these languages. Data is presented in the context of the user's programming model. With appropriate settings, measurements can be done at production-scale, in terms of the numbers of threads and processes, the size of the address-space, and length of run.

The tools support code compiled with the Sun Studio or GNU compilers. They also work on code generated by other compilers, as long as those compilers produce compatible standard ELF and DWARF symbolic information.

The tools run on the Solaris™ or Linux operating systems, on either SPARC® or x86/x64 processors. The current version, Sun Studio 12 update 1, is available for free download [1], and was used to record the screen-shots in this paper.

The objective in designing the tools was to minimize the number of mouse clicks that it takes to reach the point at which the performance problem is shown.

6.1.2 The Sun Studio Performance Tools Usage Model

The usage model for the performance tools consists of three steps. First, the user compiles the target code. No special compilation is needed, and full optimization and parallelization can be used. It is recommended that the **-g** flag be used to get symbolic and line-number information into the executable. (With the Sun Studio compilers, the **-g** flag does not appreciably change the generated code.)

The second step is to collect the data. The simplest way to do so is to prepend the **collect** command with its options to the command to run the application. The result of running **collect** is an experiment which contains the measured performance data. With appropriate options, the data collection process has minimum dilation and distortion, typically about 5%, but significantly larger for tracing runs.

The third step in the user model is to examine the data. Both a command-line program, **er_print,** and a GUI interface, **analyzer**, can be used to examine the data. (The "**er**" refers to "experiment record," the original name for what is now called an experiment.) Much of the complexity introduced into the execution model of the code comes from optimizations and transformations performed by the compiler. The Sun compilers insert significant compiler commentary into the compiled code. The performance tools show the commentary, allowing users to understand exactly what transformations were done.

6.1.3 The Sun Studio Performance Tools Features

The Sun Studio performance tools support data collection by statistical sampling of callstacks, based on either clock-ticks or hardware-counter overflow events. They also support data collection based on tracing synchronization API calls, memory allocation and deallocation API calls, and MPI API calls.

They can show a list of functions, annotated by metrics, both exclusive (in the function itself) or inclusive (in the function, and any that it calls). They also support a caller-callee view, and annotated source or disassembly listings. They can show a list of source lines or instructions, annotated with metrics, and a graphical timeline showing profiling events as a function of time. Other views are specific to various programming models or data collected and will be described below.

All of the data can be filtered in various ways to drill down into specific performance problems in function, or source-lines, or calling context, as well as by thread or CPU.

6.1.4 Diagnosing Performance Problems

The first step in understanding performance problems is determining whether or not a problem exists. That understanding is best achieved with a repeatable example (benchmark) that uses input data and problem size of a scale comparable to the production runs for which the program is being tuned. From such runs, many techniques can be used to determine if there is a problem.

Many problems have an intrinsic scale factor, N, and typical high-performance computing codes are intended to run at the largest practical scale. Performing measurements for different values of N can show whether the performance of the application scales with N, or $ln\,N$, or N^2 or even a higher power of N. If simple tests show that there are scaling problems with the application, more detailed data, including clock- and hardware-counter statistical profiling data and various kinds of tracing data, can be collected to isolate and fix the problem.

The most important question to ask is "what can I change to improve the performance of the application"? To answer that question, data is presented in the context of the user model, showing what resources are being used by the application, and where in the application they are being used. The data can also show how the execution got to that point in the program.

The next section of this paper describes various single-threaded application performance issues; the following section explores additional issues presented by multi-threaded applications. The fourth section discusses issues relating to the OpenMP programming model, and the fifth section describes issues relating to MPI programs. In the last section, we present our conclusions.

6.2 Single-Threaded Application Performance Issues

The main issues for single-threaded applications are CPU algorithmic inefficiency and memory subsystem performance.

Two techniques are used in the tools to explore these issues: clock-based profiling, used on various UNIX systems for at least 25 years [2]; and hardware-counter profiling first described in 1996 [3].

6.2.1 Algorithmic Inefficiency

The signature of algorithmic inefficiency is high resource consumption for parts of the program that do not represent the actual computational core of the algorithm. We will use two examples of low-hanging fruit to show how the tools can be used to find such signatures.

The most straightforward way to look for algorithmic inefficiency is to use clock-based statistical callstack sampling, the default experiment recorded with the Sun Studio Performance tools. On the Solaris operating environment, clock profiling collects metrics of User CPU Time, System CPU Time, Wait CPU Time, User Lock Time, Data Page Fault Time, Text Page Fault Time, and Other Wait Time. On Linux, only User CPU Time can be monitored. Such measurements show where the resources are being consumed during execution, but do not tell you whether that is bad or good. The user must decide whether the resource-consumption is necessary for the computation, or if it can be optimized.

The example program used is very simple, with two operations, each coded in an efficient way, and in an inefficient way. The two pairs of functions are named **good_init/bad_init**, and **good_insert/bad_insert**. The first of these is an apparently trivial botch which consists of doing a static initialization many times within a loop, instead of once before the loop. Figure 6.1 shows the function list from a clock-profiling experiment on the program.

The two functions highlighted are the two versions of initialization. Although the Exclusive CPU Time (column 1) is about the same, the Inclusive CPU Time (column 2) is very different, reflecting the time spent in **static_init_routine**. Figure 6.2 show the source for both of the initialization functions, clearly showing the high inclusive CPU usage in the **bad_init** version.

While this may seem too obvious for anyone ever to program, similar problems may arise more subtly, especially if the programmer is using APIs written by others.

This example is based on a performance problem in a commercial parallelization tool product developed by one of us (MI). The user interface of the product presented a list of loops in a program, with an icon representing the parallelization state of the loop. The application was written to a library that provided two APIs to add icons to a table. One added an icon to a specific row, and was easy to code; the second added a vector of icons to the table. When used on targets with relatively few loops, no problem was seen, but when used on a target with more than 900 loops, the interface took more than 20 minutes to show up. The root cause was that using the simpler algorithm caused a recomputation of the table geometry with each icon added, while the vector API only recomputed the geometry once. It took only 23 seconds to come up, a 60X improvement.

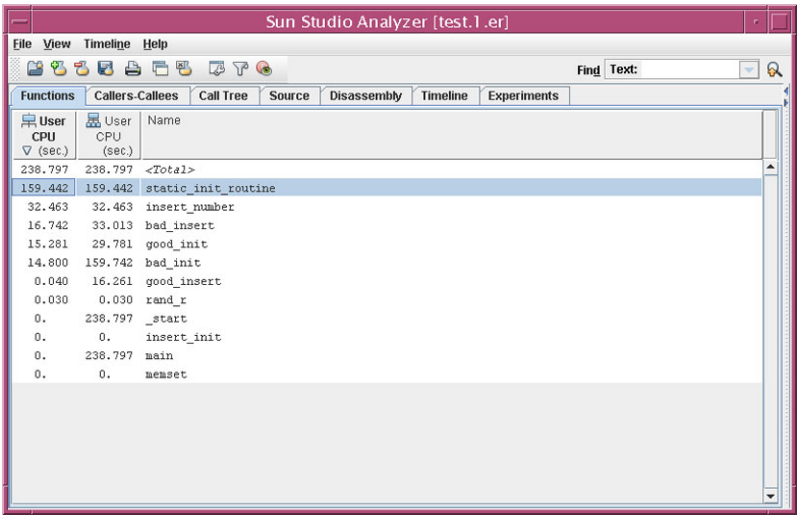


Fig. 6.1: Function List from the **lowfruit.c** program

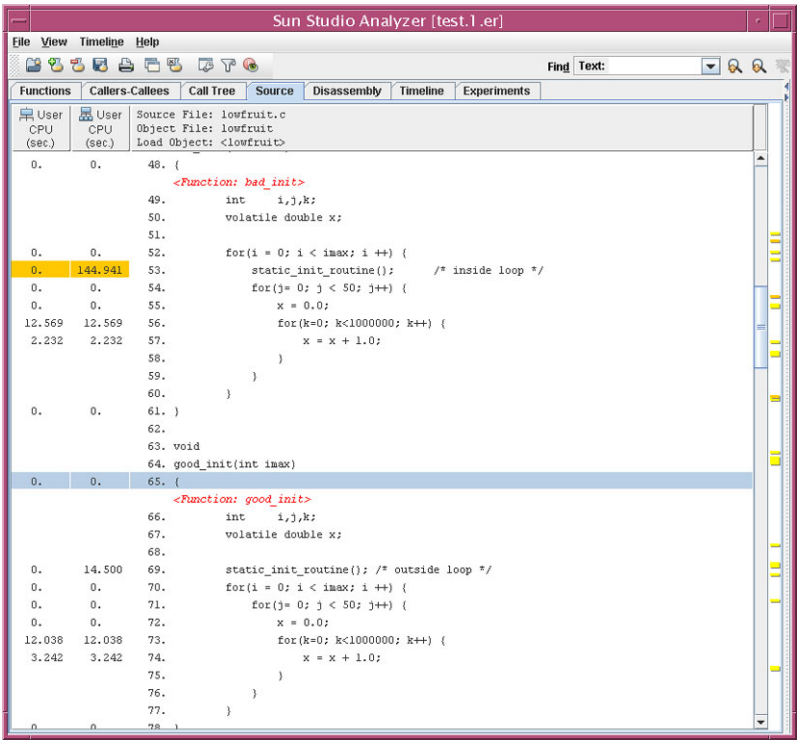


Fig. 6.2: Source Display for **good_init** and **bad_init**

The second example is a typical one where at small scale, no problem is manifest, but at large scale performance drops dramatically. The example shows two different ways to insert an element in an ordered list. Figure 6.3 shows the source of the inefficient version, **bad_insert**. It does a linear search to determine where to insert the next element.

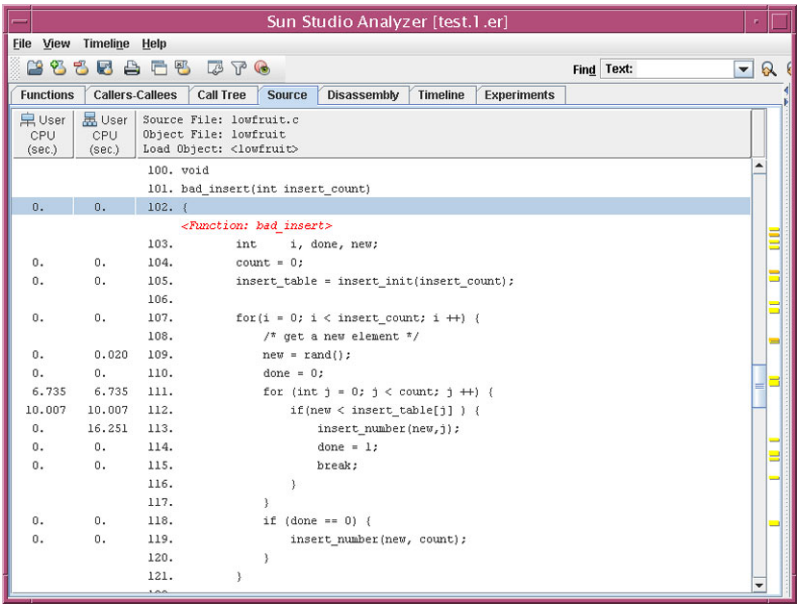


Fig. 6.3: Source Display for **bad_insert**

Note the approximately 17 seconds spent on lines 111 and 112, which represents the time spent determining where to insert the next element.

By contrast, Figure 6.4 shows the efficient version of the same problem, which does a binary search to find the insertion point.

Note that less than 0.1 seconds is spent determining where to insert the next element. The time to perform the actual insertion is approximately 16 seconds and is the same in both versions.

The above examples show performance issues in the user source code. For some problems, the issues arise in the compiler's code-generation. In those cases, the disassembly of the code, with per-instruction performance data, can be used to understand the behavior.

Clock profiling provides a good way to identify where the users should focus their tuning efforts. Sometimes, the problem areas uncovered by clock profiling can not be fully explained by algorithmic inefficiency; often, the user needs to also de-

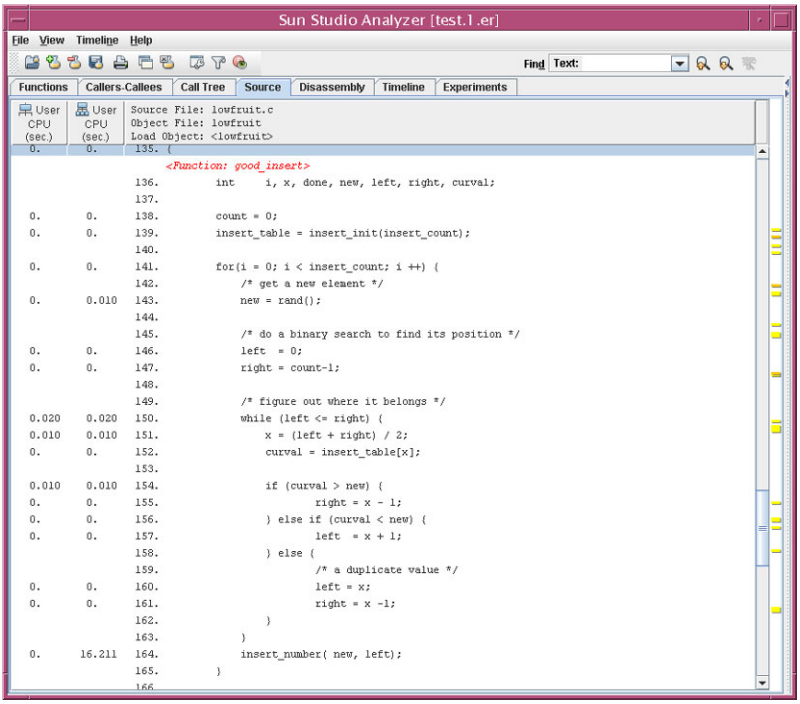


Fig. 6.4: Source Display for **good_insert**

termine how the program is using the underlying hardware, especially the memory subsystem.

6.2.2 Memory Subsystem Performance Issues

In modern computer systems, access to memory is mediated by various components in the memory subsystem. It contains hardware that maps virtual addresses to physical memory pages (a translation-lookaside-buffer, or TLB) and one or more levels of cache, designed to minimize the latency of memory fetches. These elements of the memory subsystem are not directly in the user-model of the computation. The efforts of the hardware designers to minimize latency make it difficult to relate the performance issues directly to the code in which they occur.

To help understand how a program interacts with the hardware, most modern chips have counters to measure the performance of the CPU and other subsystems in the machine. The counters can be used either by reading them directly, or by statistical profiling based on counter overflows. While the support for counters varies

from chip to chip, most chips have counters that measure memory-related activity, including TLB and cache misses. Some CPUs have counters that count not only cache-miss events, but also measure the cost of accessing memory, for example, the number of cycles stalled waiting for a cache-miss to be satisfied. Hardware counters can be used to present a detailed breakdown of CPU time waiting for various components of the memory subsystem.

Operating-system support for hardware-counter profiling is included in Solaris in **libcpc.so**. Linux systems require kernel support and a user API, such as **perfctr** [4], **perfmon2** [5], or **PCL** [6], to enable hardware-counter profiling. The Sun Studio Performance Tools currently support only the **perfctr** API on Linux systems. Users can collect hardware-counter overflow profiles based on whatever counters are available on the particular chip being utilized. Invoking the **collect** command with no arguments will print a list of all the counters available on that system.

As an example of the use of hardware-counter profiles, we constructed a simple program, **cachetest**, that does an identical matrix-vector-multiply computation in eight different ways. This code was originally intended to demonstrate the effects of optimization, so the source code consists of four copies of the same source file (with different function names), compiled with different optimization levels: no optimization, **-O** optimization, **-fast** optimization, and auto-parallelization. One version of the computation in each file is in row-column order and other is in column-row order. The eight functions are named **dgemv_***, with a suffix indicating the optimization level (**g**, **opt**, **hi**, or **p**) and the loop-order (either **1** or **2**).

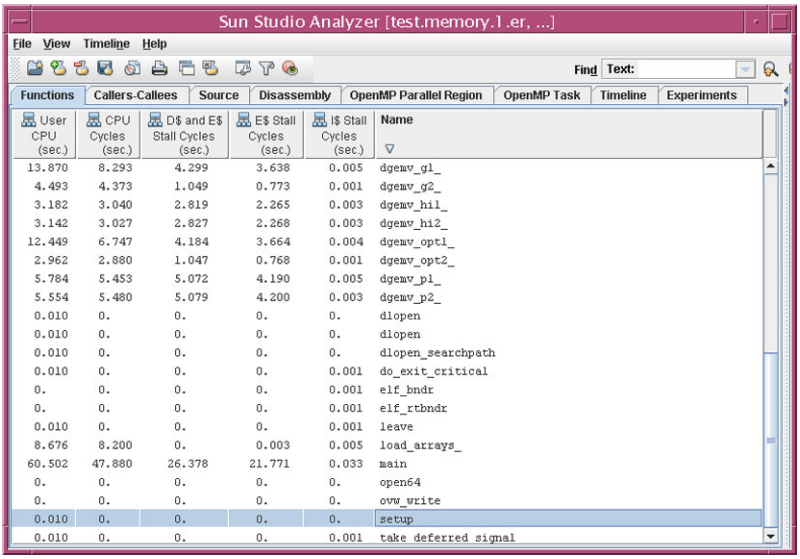
The data were recorded on a Solaris 10 system with an UltraSPARC® III-Cu processor. That chip has a TLB and two levels of cache, a first-level cache called the D-cache, and a second-level cache called the E-cache.

Two experiments were recorded, one collecting clock profiles and hardware-counter profiles for cycles and combined D-cache & E-cache stall cycles, and the second collecting E-cache stall cycles and I-cache stall cycles. The two experiments are merged, and Figure 6.5 shows the function list, sorted alphabetically.

There are a number of points of interest in this data.

First note that no appreciable time is spent dealing with instruction-cache stalls (column 5). The program is too small for I-cache performance to be an issue.

Next, note that for some functions, User CPU Time (column 1, based on clock-profiling) and CPU Cycle Time (column 2, based on hardware counter profiling for cycles) are different. Although one might naively expect them to be the same, they are not. User CPU Time represents the time the operating system thinks the process was running in user mode. CPU Cycle Time represents the time the chip thinks the process was running in user mode. They differ in an important way: in processing a TLB miss, the operating system does not change its notion of whether the process is running in user-mode – to do so would significantly increase the overhead of processing the miss. Thus the difference between these two metrics represents the time lost due to TLB misses. (There are other events that can contribute to the difference, but they are not relevant to this program.)



User CPU (sec.)	CPU Cycles (sec.)	D$ and E$ Stall Cycles (sec.)	E$ Stall Cycles (sec.)	I$ Stall Cycles (sec.)	Name
13.870	8.293	4.299	3.638	0.005	dgemv_g1_
4.493	4.373	1.049	0.773	0.001	dgemv_g2_
3.182	3.040	2.819	2.265	0.003	dgemv_hi1_
3.142	3.027	2.827	2.268	0.003	dgemv_hi2_
12.449	6.747	4.184	3.664	0.004	dgemv_opt1_
2.962	2.880	1.047	0.768	0.001	dgemv_opt2_
5.784	5.453	5.072	4.190	0.005	dgemv_p1_
5.554	5.480	5.079	4.200	0.003	dgemv_p2_
0.010	0.	0.	0.	0.	dlopen
0.010	0.	0.	0.	0.	dlopen
0.010	0.	0.	0.	0.	dlopen_searchpath
0.010	0.	0.	0.	0.001	do_exit_critical
0.	0.	0.	0.	0.001	elf_bndr
0.	0.	0.	0.	0.001	elf_rtbndr
0.010	0.	0.	0.	0.001	leave
8.676	8.200	0.	0.003	0.005	load_arrays_
60.502	47.880	26.378	21.771	0.033	main
0.	0.	0.	0.	0.	open64
0.	0.	0.	0.	0.	ovw_write
0.010	0.	0.	0.	0.	setup
0.010	0.	0.	0.	0.001	take_deferred_signal

Fig. 6.5: Function List from **cachetest**

Optimization affects both the memory performance and the efficiency of the actual generated code. From the numbers shown, the breakdown of time spent is straightforward. For the slowest version, **dgemv_g1**, a total CPU time of ~13.9 seconds is broken down into ~5.6 seconds lost due to TLB misses, ~3.5 seconds lost due to E-cache misses, ~0.7 seconds lost due to D-cache misses, and ~4.0 seconds of real computation. The fastest version, **dgemv_opt2**, shows little time lost due to TLB misses, ~0.7 seconds lost due to E-cache misses, ~0.3 seconds lost due to D-cache misses, and ~2.0 seconds of real computation.

Also note that at the lowest optimization levels, **g** and **opt,** the **1** versions are significantly slower than **2** versions. The loop-orders stride through memory differently: one is relatively efficient in cache utilization, while the other one is not. This difference disappears in the **hi** and **p** versions, because the compiler understands the stride-order implications for cache performance, and at high optimization, it interchanges the order of the two loops, so that **1** versions use the same efficient order as the **2** versions. The Sun Studio compilers insert commentary explaining the interchange into the object code, and the commentary is displayed with the source code in the Analyzer.

Hardware counter profiling can also be used to understand other performance aspects of machine behavior (branch misprediction, microcode assists, *etc.*). Tuning at this level requires an intimate understanding of the CPU architecture, and we will not discuss these issues further.

6.2.2.1 Dataspace Profiling

An extension to hardware counter profiling called dataspace profiling has been implemented [7] to better understand the data that is responsible for the cache misses. The data collector attempts to backtrack from the interrupt PC to find the actual instruction causing the cache event. From that instruction and the registers at the time of the interrupt, the collector can usually construct the virtual address being referenced. It then asks the operating system for the corresponding physical address. The Sun Studio compiler outputs information associating each load and store instruction to the symbol table entries of the data being referenced. With these pieces of information, the tools can show cache misses *vs*. the data structures and elements in the program.

The Sun Studio Performance Tools can perform dataspace profiling on SPARC®/Solaris systems, but not on other chips or operating systems. The reason is that on SPARC® processors, we can backtrack in address space to find the causal instruction, but on x86/x64, that backtracking can not be done. With the advent of new hardware and operating system mechanisms for instruction-sampling, precise instruction and virtual and physical data addresses may be directly captured on both SPARC® and x86/x64 systems.

The work described in [7] was done on one of the SPEC CPU2000 benchmarks, **`mcf`**. By using dataspace profiling, and presenting a display of time-lost due to cache misses against the data structures in the program, and the fields within them, sufficient insight was obtained to yield a 20% decrease in run time for this real application.

6.3 Multi-threading Performance Issues

Multi-threaded applications have the same potential performance issues as single-threaded applications with some additional issues relating the thread interactions and competition for machine resources.

Two of the most important issues are lock contention and false-sharing of cache lines. They are discussed in the next two subsections.

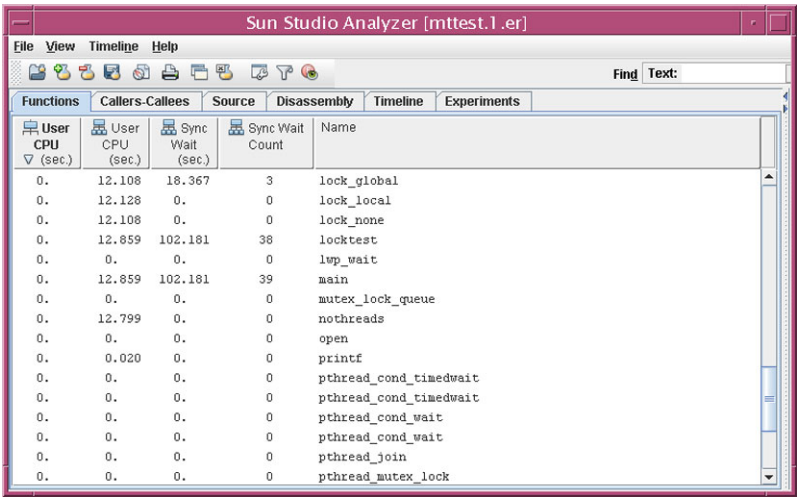
6.3.1 Lock Contention

To ensure consistency of shared data structures in an application, updates to the structures must be done atomically. Atomicity is often implemented using locks—the application must ensure that no thread alters a shared data structure without acquiring a lock governing that structure. If the application has many threads trying to update a structure, contention for the lock can lead to significant application slowdown.

One of the most important factors in the choice of locking strategies is the scope of a lock. Program safety is most easily guaranteed by using locks covering large amounts of data; program efficiency, specifically the minimization of lock contention, is most easily guaranteed by using locks covering relatively small amounts of data.

The Sun Studio Performance tools support measurement of lock contention with a technique called synchronization tracing. During data collection, the measurement library interposes on the standard functions for managing mutex-locks, reader-writer-locks, *etc*. By wrapping these calls, the collector can determine how much time was spent waiting to acquire a lock, and report that as a metric. To minimize the volume of data collected, only those synchronization events that take longer than some threshold are recorded. The threshold can be specified, or will default to approximately five times the calibrated time to acquire an uncontended lock.

Figure 6.6 shows the function list from a test program, **mttest**. The program queues up a number of work blocks, and then spawns a number of threads. Each thread fetches a work block from the queue, synchronizes with the other threads based on a parameter set in the work block, and then does a computation. Two functions, **lock_global** and **lock_local**, represent the same work done with two different synchronization methods. As the names would imply, **lock_global** uses a global mutex so that the threads can only run one at a time, while **lock_local** uses a mutex that is local to the work block, and therefore does not have any contention. In this example, the code was run with four work blocks and four threads.



User CPU ▽ (sec.)	User CPU (sec.)	Sync Wait (sec.)	Sync Wait Count	Name
0.	12.108	18.367	3	lock_global
0.	12.128	0.	0	lock_local
0.	12.108	0.	0	lock_none
0.	12.859	102.181	38	locktest
0.	0.	0.	0	lwp_wait
0.	12.859	102.181	39	main
0.	0.	0.	0	mutex_lock_queue
0.	12.799	0.	0	nothreads
0.	0.	0.	0	open
0.	0.020	0.	0	printf
0.	0.	0.	0	pthread_cond_timedwait
0.	0.	0.	0	pthread_cond_timedwait
0.	0.	0.	0	pthread_cond_wait
0.	0.	0.	0	pthread_cond_wait
0.	0.	0.	0	pthread_join
0.	0.	0.	0	pthread_mutex_lock

Fig. 6.6: Function List from **mttest**

The two functions, **lock_global** and **lock_local** each consume ~12 seconds of CPU time, reflecting the fact that the processing of the work block is independent of synchronization. However, **lock_global** shows ~18 seconds spent waiting for the lock, while l**ock_local** spends no time.

With the global lock, one thread immediately acquires the lock and does 3 seconds of computation while the other three threads wait. After the first thread completes, a second thread acquires the lock and does 3 seconds of computation while the two other threads wait. When the second completes, the third acquires the lock and computes while the last thread waits. Finally, the last thread acquires the lock and does its computation.

The total wait time is 3 X 3, plus 3 X 2, plus 3 X 1, adding up to the 18 seconds shown as synchronization wait time. Each of the three threads that wait also contributes one synchronization wait event to the count.

Figure 6.7 shows the source of **lock_global**, with the synchronization wait on the source line that calls the lock function.

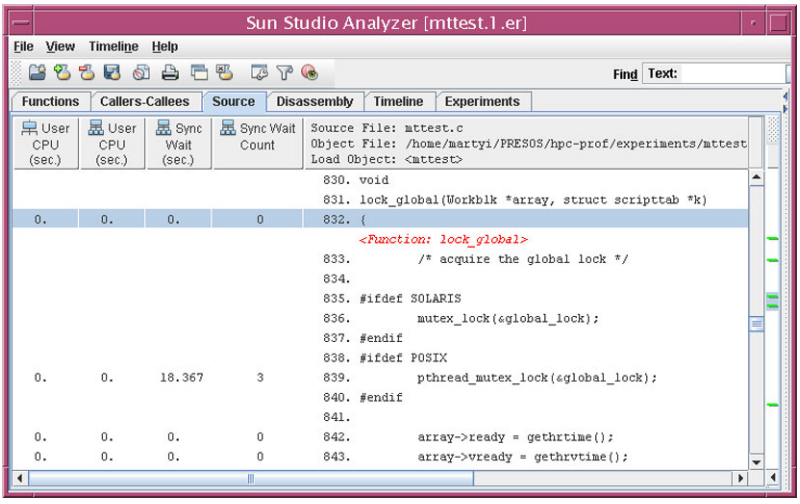


Fig. 6.7: Source of **lock_global**

In this example, it is clear that there is no need to have a global lock; in typical programs, the appropriate scope of a lock is harder to determine. In more complex cases, synchronization tracing is a valuable technique for determining which locks should be targeted for optimization efforts.

6.3.2 False Sharing of Cache Lines

Thread interactions around memory caches are another important performance issue in multi-threaded applications. Multiprocessors have hardware mechanisms to ensure that memory modification by one CPU will invalidate copies of the corresponding cache-line on other CPUs. When one thread updates the data, the cache lines in the CPUs running the other threads will be evicted, and the next reference will force a cache miss and memory access.

If the two threads are referring to the same data, the performance costs of the repeated memory accesses are unavoidable. However, if the two threads are referring to different data, but the data is on the same cache line, the repeated evictions and memory fetches are not really needed, but the hardware can't tell. That circumstance is called "false sharing."

With dataspace profiling, each memory-counter profiling event contains the virtual and physical addresses being referenced. With knowledge of the cache mapping algorithm, an event can be mapped to a cache line.

The performance tools have very powerful filtering mechanisms, and false sharing can be detected using those mechanisms. In the **mttest** example, one of the compute functions exhibits false sharing. Dataspace profile data shows that there is only one hot cache line. By filtering to show only data referring to that cache line, we can determine that it is referred to by four threads, each of which is using a different address within that line. The techniques have been described in detail [8].

6.4 OpenMP Performance Issues

A key challenge for a user-friendly profiling tool is relating information gathered from low-level machine instructions to the user's source code which is written in a high-level programming model like OpenMP. OpenMP programs have a simple fork-join model that is governed by directives in the source code. In the user model, when a parallel region is entered, additional worker threads are created, and, when the computations inside the region are completed, the worker threads disappear. OpenMP 3.0 introduces an additional model, tasking, whereby threads queue up tasks to be performed and worker threads pick up and perform those tasks.

The underlying execution model for OpenMP is significantly more complex than the user model. One of the challenges of any profiling tool is to figure out how to represent the execution-model data back in the user model. Figure 6.8 shows the user-model and execution-model callstacks when the program is executing within a parallel region.

All four threads in this example are executing in the same parallel region, although the leaf PC is shown on different lines in different threads. In the execution model, the function **foo** calls into the OpenMP runtime which dispatches work to the three slave threads as well as back to the master thread. The function called to

Master	Slave1	Slave2	Slave3
foo, line nn main _start	foo, line mm main _start	foo,line pp main _start	foo, line rr main _start

(a) User-model Callstack

Master	Slave1	Slave2	Slave3
foo-OMP, line nn libmtsk foo main _ start	foo-OMP, line mm libmtsk lwp_start	foo-OMP, line pp libmtsk lwp_start	foo-OMP, line rr libmtsk lwp_start

(b) Execution-model Callstack

Fig. 6.8: User-model and Execution-model Callstacks

do the work is shown as **foo-OMP**; it is a so-called outline function constructed by the compiler, but not part of the user model. The line numbers refer to the original source file.

The user-model callstacks are constructed by stripping the frames below the OpenMP runtime in the execution-model in the master, stripping the frames above the OpenMP runtime in the slaves and the master, and stitching the two pieces together to yield the user-model callstacks [9].

When profiling OpenMP applications, the collector records data representing the OpenMP runtime's notion of what the application is doing with every profiling tick. In the current tools, two metrics are computed: OMP Work Time and OMP Wait Time. OMP Work Time includes both serial and parallel CPU time. OMP Wait Time includes overhead as measured in the runtime, and implicit or explicit wait. On the Solaris operating system, OMP Wait Time accumulates whether specified as a busy-wait, consuming CPU time, or as a sleep-wait, not consuming CPU time. On Linux, it accumulates only with busy-wait.

There are four major issues in understanding the performance of OpenMP programs. They are excess parallel overhead, insufficient parallelism, lock contention and synchronization, and load imbalance. They will be discussed in the next four subsections.

6.4.1 Excess Parallel Overhead

Excess parallel overhead arises from applications which are parallelized, but where the work to set up the parallelism is a significant fraction of the total work done in parallel. It is difficult to directly measure: while the OpenMP Performance Measurement API does report time spent in an overhead state, additional work is done in

the application to prepare for parallel execution, and that overhead is not detected. (Future work is directed to a more accurate and explicit measurement.)

In the current version of the tools, a pseudo-function, <**OMP-overhead**>, gets metrics attributed to it whenever the program has its leaf function inside the OpenMP runtime. That function is shown as called from the particular parallel regions or tasks responsible for the overhead, allowing the user to see how the program got to that point.

6.4.2 Insufficient Parallelism

Too little parallelism is manifested by high CPU time spent in non-parallel code. According to Amdahl's law, the amount of serial work limits scalability of the application, and thus should be minimized to extract the maximum performance and scalability out of the machine.

In the OpenMP model, all programs start in what is called the "Implicit OpenMP Parallel Region." All serial code is executed in that region, despite its being called a parallel region, while parallel code is executed in other parallel regions. By filtering the performance data to show only data relating to the Implicit OpenMP Parallel Region, direct measurement of serial execution is shown.

Figure 6.9 shows the Parallel Region Tab of an application, with the Implicit OpenMP Parallel Region selected and used to set a filter.

Figure 6.10 shows the function list with the filter applied; it thus shows only the serial portions of the computation.

The function named **serial_** contains an expensive loop that is being executed in serial mode, and represents an opportunity for improving parallelism.

The above example represents the simplest case of not having sufficient parallelism in the code. There are many other cases where this might be a problem. For example, a code may be parallelized using OpenMP sections, where the user has specified 32 threads, but only coded 4 sections. In that case, 28 threads will be doing nothing, despite the code being parallelized. The discrepancy will show up as a load balance issue (see section 6.4.4). Another cause of insufficient parallelism may be in queueing and processing tasks, a topic for future discussion.

6.4.3 Lock Contention

Lock contention causes one or more threads to wait for execution on a lock. It is easily detected as high OMP Wait Time that shows up on the statement representing the lock. It can occur either as an lock call, or as an OpenMP "critical" or "atomic" directive or pragma.

Figure 6.11 shows a picture of source from a program that has a performance hit representing contention for a critical region.

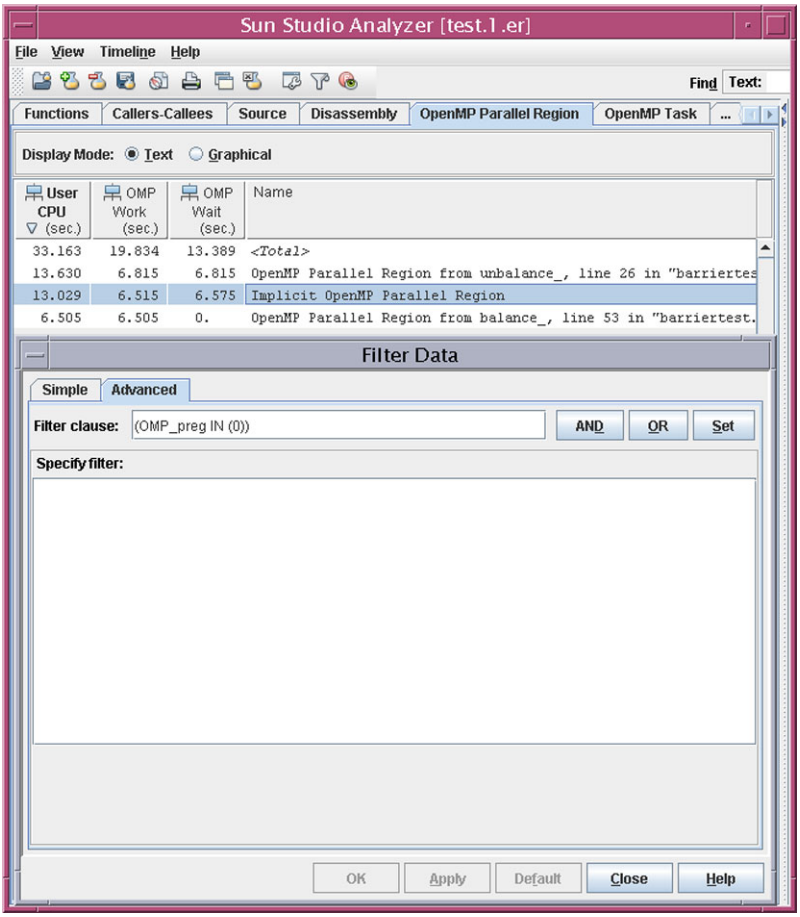


Fig. 6.9: Parallel Region Tab and Filter

The lines with high metric values are highlighted, pointing out the contention. Other scenarios will also show high OMP Wait Time in other pragmas.

6.4.4 Load Imbalance

Load imbalance also shows up as high OMP Wait Time. At the end of a parallel region, synchronization creates an implicit barrier, and the time spent at the barrier represents load imbalance: some threads are done, but none can proceed until all threads reach the barrier. Time spent in that synchronization is attributed to the artificial function <**OMP-implicit_barrier**>.

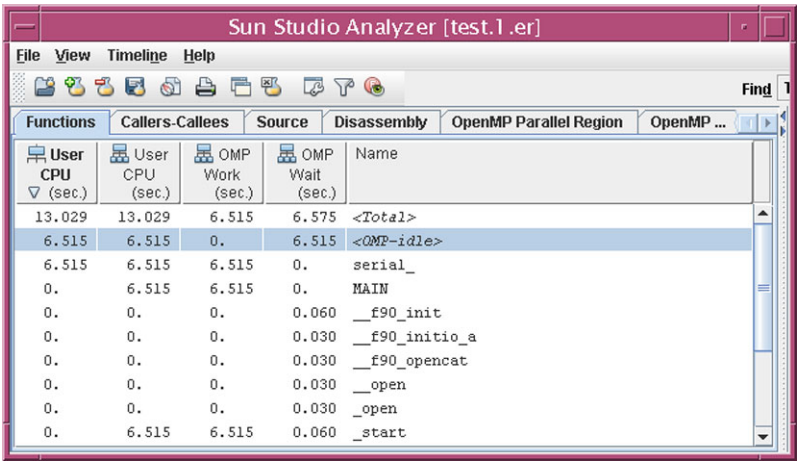


Fig. 6.10: Function list, Filtered to Show only Serial Portions of the Code

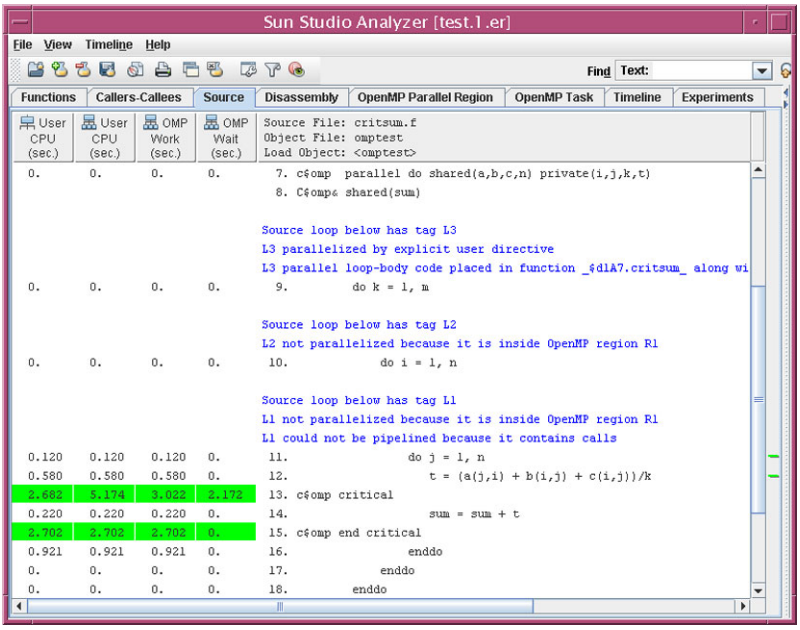


Fig. 6.11: High OMP Wait Time in a Critical Section

OpenMP programs can be run either with a sleep-wait or a spin-wait. OpenMP Wait Time accumulates on Solaris in either form of wait, while CPU time accumulates only for a spin-wait. (On Linux, where profiling is only for CPU time, OpenMP Wait Time is also only accumulated for spin-waits.)

6.5 MPI Performance Issues

MPI programs run as a number of distinct processes, on the same or different nodes of a cluster. Each process does part of the computation, and the processes communicate with each other by sending messages.

The challenge in parallelizing a job with MPI is to decide how the work will be partitioned among the processes, and how much communication between the processes is needed to coordinate the solution. To address these aspects of MPI performance, data is needed on the overall application performance, as well as on specific MPI calls.

Communication issues in MPI programs are explicitly addressed by tracing the application's calls to the MPI runtime API. The data is collected using the VampirTrace [10] hooks, augmented with callstacks associated with each call. Callstacks are directly captured, obviating the need for tracing all function entries and exits, and resulting in lower data volume.

MPI tracing collects information about the messages that are being transmitted and also generates metrics reflecting the MPI API usage: MPI Time, MPI Sends, MPI Receives, MPI Bytes Sent and MPI Bytes Received. Those metrics are attributed to the functions in the callstack of each event.

Unlike many other MPI performance tools, the Sun Studio Performance Tools can collect statistical profiling data and MPI trace data simultaneously on all the processes that comprise the MPI job. In addition, during clock-profiling on MPI programs, state information about the MPI runtime is collected indicating whether the MPI runtime is working or waiting. State data is translated into metrics for MPI Work Time and MPI Wait Time. State data is available only with the Sun HPC ClusterTools™ 8.1 (or later) version of MPI, but trace and profile data can be captured from other versions of MPI.

6.5.1 Computation Issues in MPI Programs

The computation portion of an MPI application may be single-threaded or multi-threaded, either explicitly or using OpenMP. The Sun Studio Performance Tools can analyze data from the MPI processes using any of the techniques described in the previous sections for single- and multi-threaded profiles. The data is shown aggregated over all processes, although filtering can be used to show any subset of the processes. Computation costs are shown as User CPU Time (with clock-

profiling); computation costs directly attributable to the MPI communication are shown as MPI Work time, a subset of User CPU Time. Time spent in MPI is shown as MPI Time, which represents the wall-clock time, as opposed to the CPU Time, spent within each MPI call.

The data is shown in the function list, Figure 6.12.

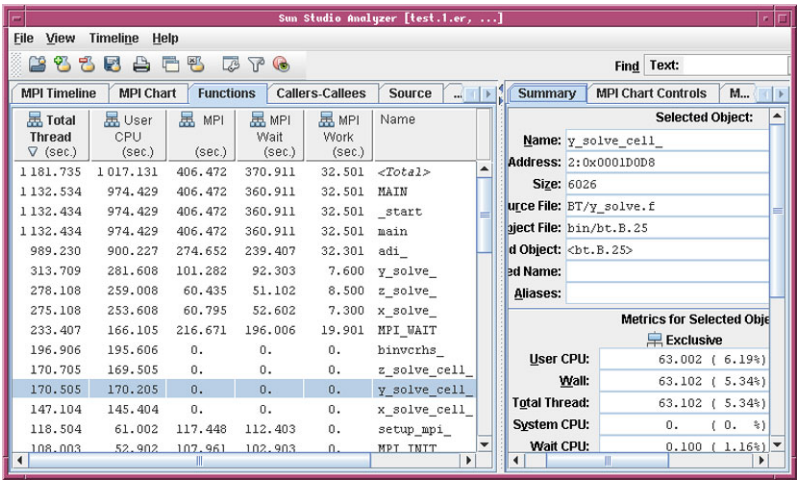


Fig. 6.12: MPI Function List

Functions, such as **y_solve_cell_**, that have high User CPU Time but little or no MPI Work Time or MPI Wait Time represent the actual computations that are done. All of the techniques discussed earlier are relevant to understanding the performance of the computational part of the application, and the tuning that would be done for them is exactly analogous to what would be done for a non-MPI program.

6.5.2 Parallelization Issues in MPI Programs

While the performance issues in computation can be recognized using the techniques described above, problems in partitioning and MPI communication can be recognized by excessive time spent in MPI Functions. The causes of too much time in MPI functions may include: load imbalance; excessive synchronization; computation granularity that is too fine; late posting of MPI requests; and limitations of the MPI implementation and communication hardware.

Many MPI programs are iterative in nature, either iterating on a solution until numerical stability is reached, or iterating over time steps in a simulation. Typically,

each iteration in the computation consists of a data receive phase, a computation phase, and a data send phase reporting the results of the computation.

6.5.2.1 Using the MPI Timeline to Visualize Job Behavior

The MPI Timeline gives a broad view of the application behavior, and can be used to identify patterns of behavior and to isolate a region of interest.

The MPI Timeline, shown in Figure 6.13, initially covers the entire run of the application, including initialization (**MPI_Init**) and finalization (**MPI_Finalize**).

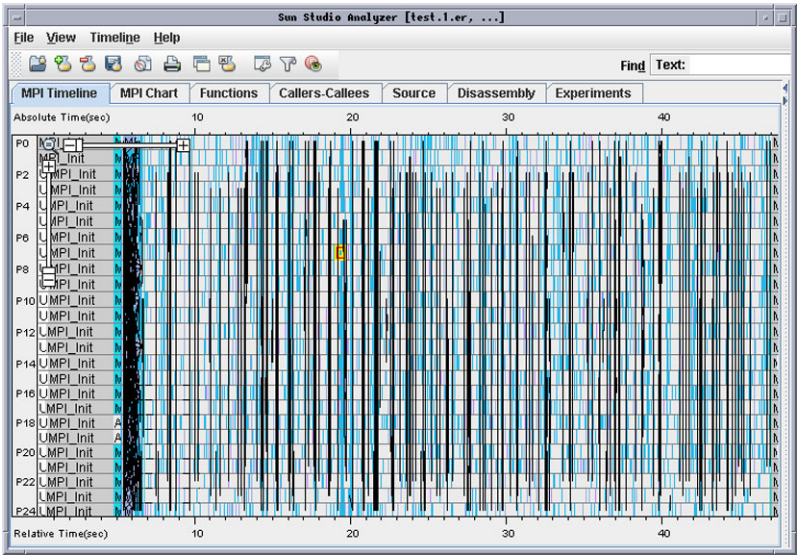


Fig. 6.13: MPI Timeline, Full Scale

The Timeline shows the MPI processes vertically, with time displayed horizontally. For each process, blocks indicating MPI calls and application code are shown. Lines indicating messages are drawn between the sending call and the receiving call for each message. In typical applications, the message volume is quite high, which can lead to a picture that is obscured by the message lines. The user can adjust the display to set a percentage of messages to be shown. Priority of display is given to the most costly messages, that is, the messages that represent the largest amount of time spent in sending and receiving them.

The user can zoom in to help recognize the pattern of execution. The same experiment shown above, when zoomed in, shows a clear pattern (Figure 6.14). In this case, three iterations are shown.

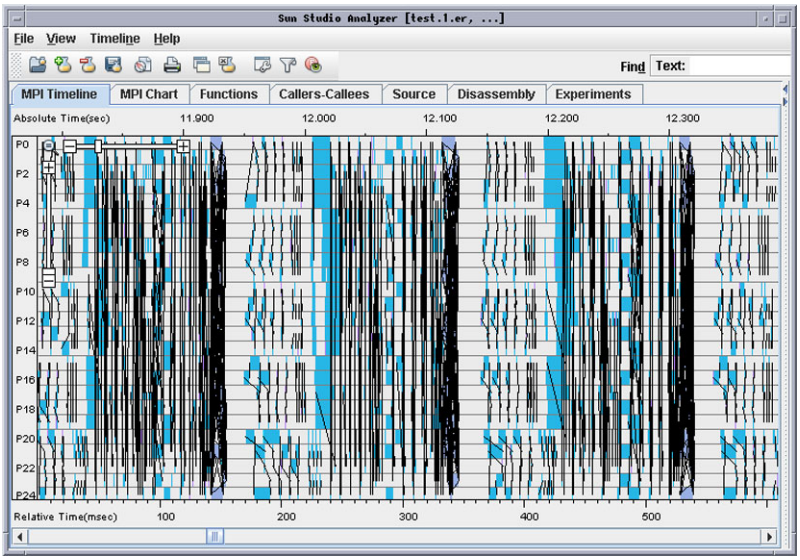


Fig. 6.14: MPI Timeline, zoomed in

The user can zoom in further, and will then see the names of the MPI functions inside their blocks. At any point, the user can select specific MPI events to determine the callstack of the process, and the duration of the call. The user can also select a message to see the message size and the sending and receiving processes and their callstacks.

A filter can be set based on any zoomed-in view of the data, allowing the user to isolate patterns of communication. Typically, the user will set a filter from the MPI Timeline to focus analysis on the steady-state heart of the computations.

6.5.2.2 Using MPI Charts to Understand Where Time Was Spent

The Analyzer's initial MPI Chart shows in which MPI function the time is spent. Figure 6.15 shows the time spent in each of the MPI calls, and in the Application (which is the time spent between MPI calls).

In this example, we can see that approximately 75% of the total time is spent in the application's computation. The remaining 25% of time is spent in MPI calls, with almost all of it spent in the function **MPI_Wait**.

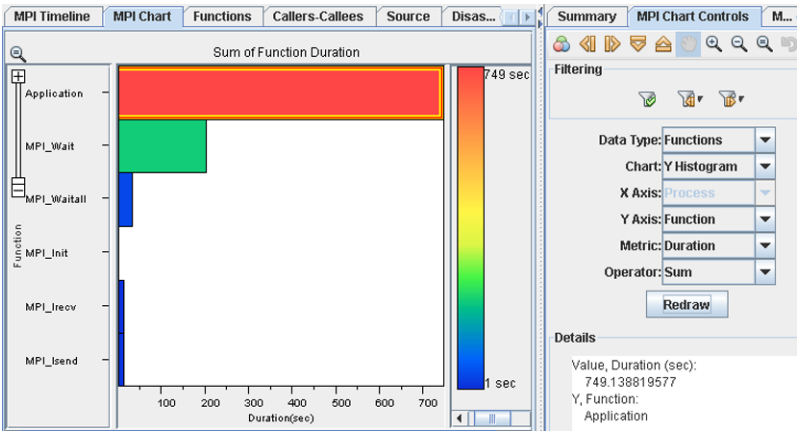


Fig. 6.15: MPI Chart: Application Time vs. MPI Time

6.5.2.3 Using MPI Charts to Understand Message Traffic

The MPI Charts can be used to understand the patterns of communication between processes. In Figure 6.16, we show a 2-dimensional plot, showing data volume in bytes as a color in a grid of sending and receiving processes.

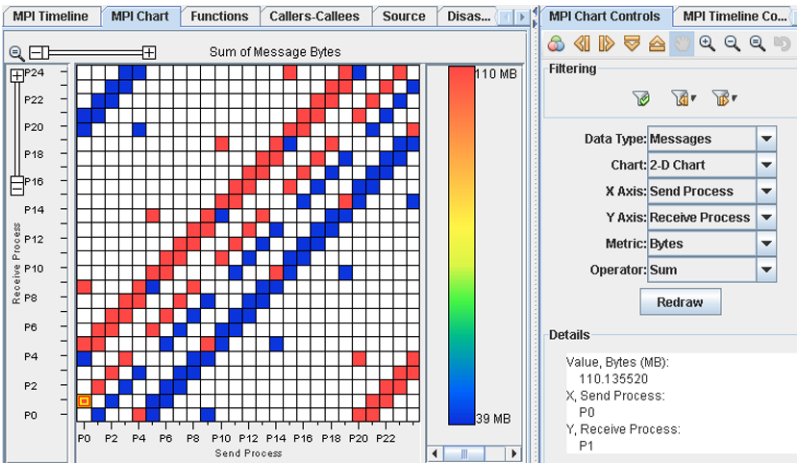


Fig. 6.16: MPI Chart: Bytes-Transmitted among Processes

This chart shows how much data is being passed between the processes. Charts can also be used to explore other aspects of message traffic, including delivery times, and send and receive functions.

6.5.2.4 Using MPI Charts to Understand Work Distribution

The previous techniques have been directed towards understanding the average behavior of the application. They do not indicate if, for example, some processes are running slower than others, or if the behavior is consistent over time. The MPI Charts provide a powerful way to explore these types of issues.

To investigate work distribution, the user can first set a filter to isolate the time in Application, the pseudo-function that represents work done between MPI calls. Then the user can display the amount of time spent in the Application state for each process, as shown in Figure 6.17.

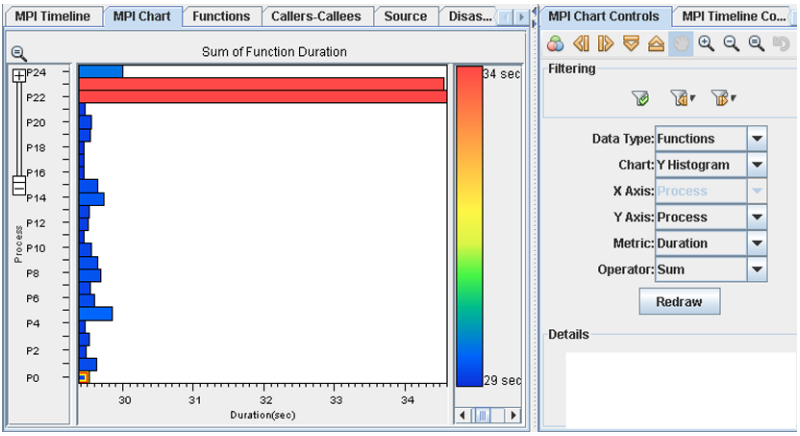


Fig. 6.17: MPI Chart: Time in Application vs. Process

In this example. processes 22 and 23 spend more time in computation than the other processes. Fixing this imbalance may improve the overall performance of the application.

With the filter still set, the user can look at the work distribution over time. Figure 6.18 is a 2-dimensional chart showing process number vertically, wall-clock entry time horizontally, and coloring to represent the relative amount of Application work being done.

The excess time spent in processes 22 and 23, which was visible in Figure 6.17, is now seen to be consistent over the whole run. At the wall-clock times of approximately 19 and 40 seconds into the run, there appears to be hiccups where all the processes are getting less work done.

To investigate time-based anomalies like those shown in Figure 6.18, the user can look at the distribution of Application work periods as a function of wall-clock time, as shown in Figure 6.19.

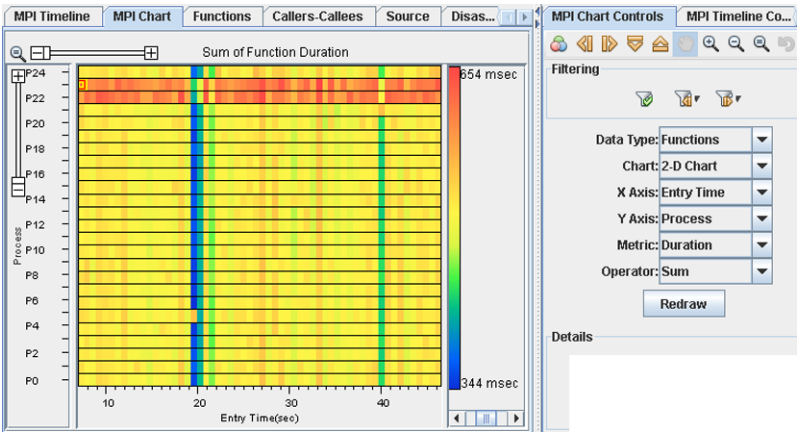


Fig. 6.18: MPI Chart: Time in Application Per-process as a Function of Wall-clock Time

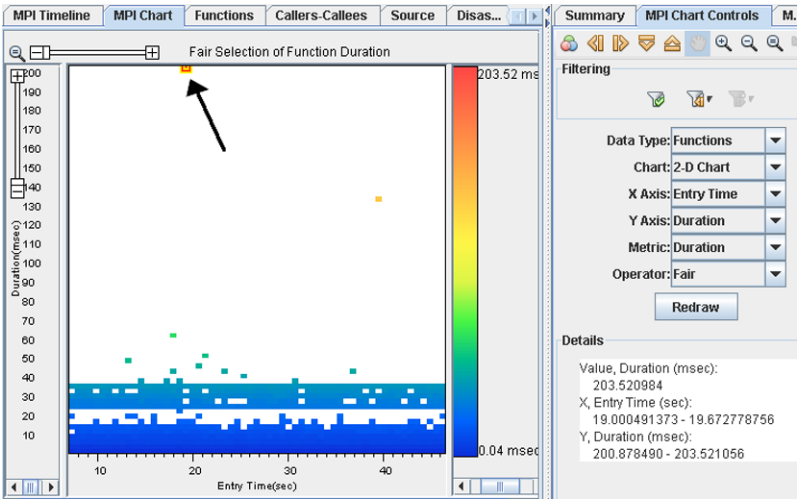


Fig. 6.19: MPI Chart: Time in Application vs. Wall-clock Time

Most work periods (time in Application) are less than 40 milliseconds in duration, but at ~19 seconds into the run, there is a data point showing a work period of 203 milliseconds. There is a second outlier representing the stutter at approximately 40 seconds into the run.

6.5.2.5 Using Filters to Isolate Behaviors of Interest

The MPI filters can be used to pick out behaviors of interest and determine which events are responsible. For example, to focus on that anomalous data point in Figure 6.19, the user can zoom in and apply a filter to isolate the data of interest. Then, the user can switch to the Timeline, zoom in on that event, and remove the filter to show the context of the other processes around that event. Figure 6.20 shows the anomalous event and context on the Timeline.

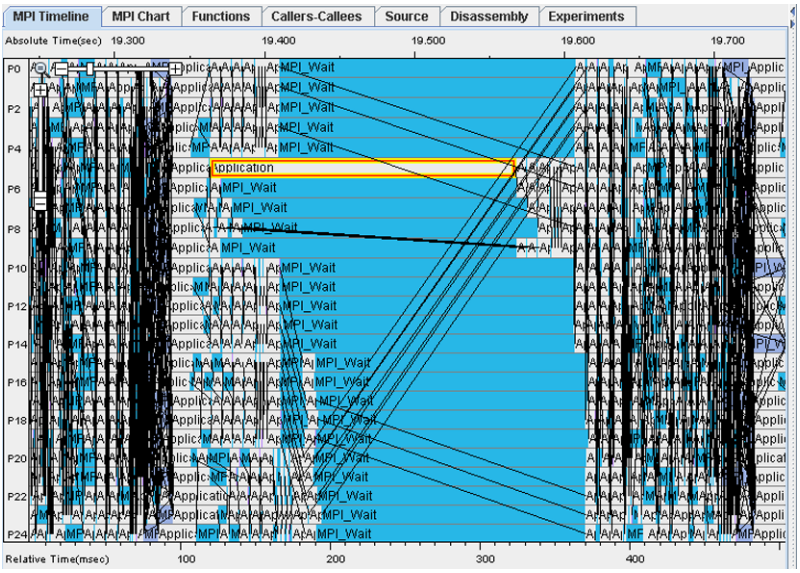


Fig. 6.20: MPI Timeline: Outlier Event Shown

In Figure 6.20, we can see that the long-duration Application event in process 5 impacted all the other processes: while process 5 is computing, all the other processes are waiting. Further drilling down using the Timeline would allow the user to see the source contexts of the anomaly and the surrounding events.

6.6 Conclusions

We have described the Sun Studio Performance Tools and the user model they support. We then discussed single-threaded applications, and the importance of both algorithmic efficiency and memory subsystem behavior to the overall performance

of the application. We described the techniques in the performance tools to measure both of these.

We then explored the issues introduced by multi-threading, and gave examples of locking issues, and memory- and cache-contention issues among threads.

We described support in the tools for the OpenMP programming model, and the performance issues concerning OpenMP, including detection of too-little-parallelism, excess-overhead, lock-contention, and load-balance. In each case we showed how the tools can highlight the problems.

Finally, we explored the MPI programming model and the ways in which the tools can measure MPI performance. We described some of the typical characteristics of MPI jobs, and showed how the patterns of communication and computation can be explored. We then showed how the tools can be used to isolate behaviors of interest and to understand their causes.

Acknowledgements We would like to thank Oleg Mazurov, Eugene Loh, Brad Lewis, Nik Molchanov, and Vladimir Mezentsev who did much of the work developing the tools. We would also like to thank some of our users, in particular, Darryl Gove, Ruud van der Pas, Karsten Guthridge, Miriam Blatt, and others too numerous to name.

References

1. Sun Studio Downloads, http://developers.sun.com/sunstudio/downloads/index.jsp.
2. S.L.Graham, P.B. Kessler, and M.K.McKusick, An Execution Profiler for Modular Programs, Software Practice and Experience, 13, 671-685, August, 1983.
3. Marco Zagha, Brond Larson, Steve Turner, and Marty Itzkowitz, Performance Analysis using the MIPS R10000 Performance Counters, Proceedings of SuperComputing '96, Pittsburgh, PA, November, 1996.
4. Mikael Pettersson, Linux Performance-Monitoring Counters Driver, http://user.it.uu.se/~mikpe/linux/perfctr/ Computing Science Division, Uppsala University, Sweden.
5. Stéphane Eranian, "Perfmon2: a standard performance monitoring interface for Linux", http://perfmon2.sourceforge.net/perfmon2-20080124.pdf, January 2008.
6. Ingo Molnar, Thomas Gleixner, "[ANNOUNCEMENT] Performance Counters for Linux", http://lkml.org/lkml/2008/12/4/401, December 2008.
7. Marty Itzkowitz, Brian J. N. Wiley, Christopher Aoki, and Nicolai Kosche, Memory Profiling using Hardware Counter, Proceedings of SuperComputing '03, Phoenix, AZ, November, 2003.
8. Marty Itzkowitz, Memory Subsystem Profiling with the Sun Studio Performance Analyzer, http://cscads.rice.edu/workshops/summer09/slides/performance-tools/DProfile.cscads.pdf.
9. Yuan Lin and Oleg Mazurov. Providing Observability for OpenMP 3.0 Applications, Proceedings of the 5th International Workshop on OpenMP. Dresden (2009).

10. The VampirTrace Project, http://www.tu-dresden.de/zih/vampirtrace, Technische Universität Dresden, Center for Information Services and High Performance Computing (ZIH), Dresden, Germany.