ORACLE
SOLARIS STUDIO

DEBUG MULTITHREADED APPLICATIONS QUICKLY AND EASILY WITH POWERFUL ANALYSIS TOOLS

Today's hardware platforms offer unprecedented performance, particularly when combined with multithreaded applications. Oracle Solaris Studio Thread Analyzer offers developers a powerful tool for analyzing and debugging complex, parallel applications.

KEY FEATURES

- Analyzes the execution of multithreaded programs
- Detects data race conditions
- Detects deadlock conditions

BENEFITS

- Improves application quality and reliability
- Enhances developer productivity

Introduction

While multicore systems offer unprecedented performance, taking advantage of their unique capabilities adds complexity for developers. With the ability to run multiple tasks at the same time, multithreaded applications require less computation time and deliver faster performance and responsiveness—but are much more difficult to debug.

Consider a matrix multiplication operation programmed with four threads and running on a system with two dual-core processors. All threads run concurrently on four processor cores, with each thread computing part of the resulting matrix. While the job completes four times faster, hazards remain for developers.

See Figure 1 for example program code for a matrix multiplication operation. The outermost `for` loop (that is, the `i` loop) is parallelized using the OpenMP for pragma (or directive). The `shared` clause in the pragma specifies which variables are shared among the threads, while the `private` clause specifies which variables are private to each thread. Each thread will have its own private copies of the private variables. The `num_threads` clause in the pragma specifies that four threads should be used. Each of the four threads will execute a subset of the iterations of the `i` loop.

```
#pragma omp parallel for shared(A,B,C, n) private(i,j,k,sum)
num_threads(4)
for(i=0; i<n; i++)
{
 for(j=0; j<n; j++)
 {
  sum=0;
  for(k=0; k<n; k++)
  {
   sum=sum+(A[i][k]*B[k][j]);
   C[i][j]=sum;
  }
 }
}
```

Figure 1: Code snippet showing a matrix multiplication operation programmed for four threads.

When code is parallelized, mathematical operations can be completed out of order, which may or may not affect the accuracy of computations. Threads can overwrite data or utilize stale information.

Optimizing performance and ensuring application accuracy requires developers to implement locking mechanisms properly and quickly determine if, and where, problems exist.

Oracle Solaris Studio Thread Analyzer

Oracle Solaris Studio includes a tool specifically designed to analyze thread execution and ease the task of debugging parallelized applications. The Oracle Solaris Studio Thread Analyzer detects multithreading errors, such as data races and deadlocks, in programs written using the following standards and frameworks:

- POSIX threads API
- Oracle Solaris threads APIs
- OpenMP directives
- A combination of the above

Find Hard to Detect Data Race Conditions

A multithreaded program has two or more threads (in the same process). Communication between threads is simple—threads share data, address space, virtual memory, file descriptors, and more. Because threads share address space, data produced by one thread is immediately available to all other threads in the process.

Data sharing can lead to programming challenges. For example, each thread in a matrix multiply program computes a portion of the resulting matrix. However, developers need to be careful that data sharing among the threads does not lead to data races.

A *data race* occurs when two threads access the same memory location concurrently, one of the accesses is a write operation, and the threads do not use locks to control access to that memory location. Data races often cause incorrect and nondeterministic results, and they are very hard to debug. For example, in the matrix multiply code in Figure 1, loop indices `i`, `j`, and `k` are used as counters when traversing elements in the rows or columns of the matrices. It is important that each thread use its own private counters; otherwise, the traversals and computations may be incorrect due to data races. While some data races do not impact application accuracy, others can—and likely are due to programming errors.

The Oracle Solaris Studio Thread Analyzer makes it possible for developers to find data race conditions through a series of straightforward steps:

- Instrument the program code to be checked by compiling it with the special `-xinstrument=datarace` compiler option or by using the Sun Memory Error Discovery Tool (Discover). The Discover tool is an advanced development tool included in Oracle Solaris Studio for detecting memory access errors.
- Create an experiment to detect data races by running the instrumented executable using the `collect` command with the `-r race` option. Any data races detected are recorded in the experiment.
- Examine experiment results (using the `tha` or `er_print` command) and determine whether the data races revealed by the Oracle Solaris Thread Analyzer are benign or are bugs in the code.
- Fix any bugs causing the reported data race(s).
- Devise additional experiments for retesting. Repeating tests using differing factors, such as input data, number of threads, loop schedules, or servers, can help to locate nondeterministic problems in the code.

For example, suppose the programmer omitted the clause `private(i, j, k, sum)` in the OpenMP pragma shown in Figure 1. That is, the programmer should have written this:

```
#pragma omp parallel for shared(A,B,C, n) private(i,j,k,sum) num_threads(4)
```

But instead, the programmer wrote this:

```
#pragma omp parallel for shared(A,B,C, n) num_threads(4)
```

The Thread Analyzer will detect that there is a data race on `j`, `k`, and `sum`, as shown in Figure 2 and Figure 3.

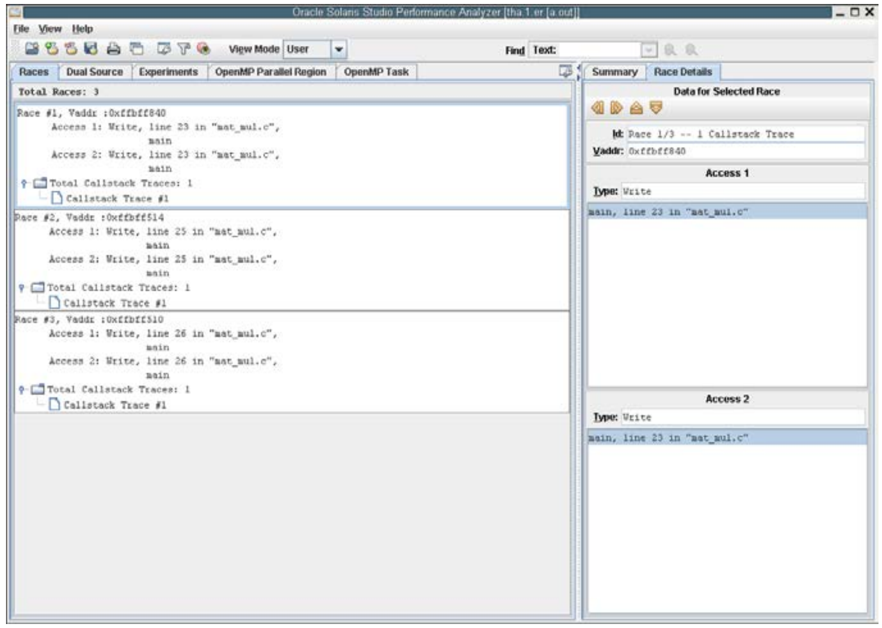

Figure 2: The Races tab in the Thread Analyzer GUI lists detected data races.

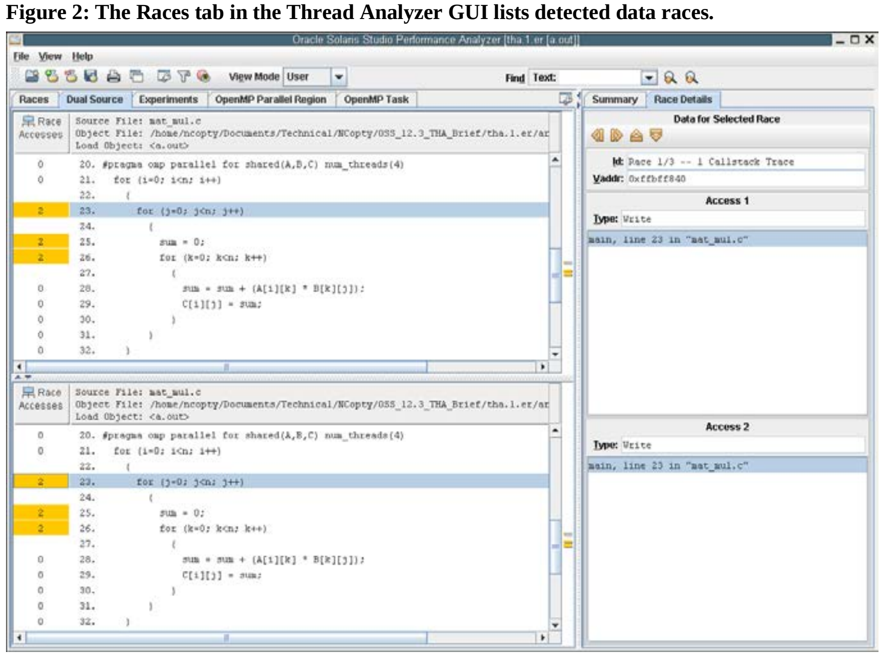

Figure 3: The Dual Source tab in the Thread Analyzer GUI pinpoints where in the source code a data race occurred.

The Oracle Solaris Studio Thread Analyzer supports debugging whether or not the source code is available. When the source code is unavailable, the binary can be instrumented using the Discover tool. Use binary instrumentation when the source code for shared libraries is not available and the libraries cannot be instrumented through recompilation.

Eliminate Deadlocks

A *deadlock* occurs when two or more threads in the program compete to acquire rights to the same global resources and permanently block each other. Consider two threads, where each thread is holding a resource (lock) while requesting the resource held by the other thread. Neither thread can proceed until the other releases the resource it is holding. The threads are blocked indefinitely, making it impossible for the application to complete successfully.

Detecting deadlocks can be difficult. Even if a thread can make progress, it does not mean that a deadlock has not occurred somewhere else in the program. The Oracle Solaris Studio Thread Analyzer detects potential and actual deadlocks in multithreaded applications:

- **Actual deadlock.** An actual deadlock is a deadlock that actually occurred during the execution of the program. Two or more threads were blocked waiting for each other. While the threads involved hang, the entire process may or may not hang.
- **Potential deadlock.** A potential deadlock does not necessarily occur in a given run but can occur in any execution of the program depending on the scheduling of threads and the timing of resource (lock) requests by the threads.

Analyzing programs for deadlocks is accomplished through a series of steps similar to those for detecting data races:

- Create an experiment to detect deadlocks by running the program using the `collect` command with the `-r deadlock` option. Any deadlocks identified are recorded in the experiment.
- Examine experiment results (using the `tha` or `er_print` command) to determine if there are any deadlocks.
- Fix any bugs causing the reported deadlock(s).
- Devise additional experiments that vary factors such as input data, number of threads, loop schedules, or servers. Repeat experiments with different conditions or timing changes—this is the best way to detect deadlocks in multithreaded code.

Note that normal termination does not mean the program is safe from deadlocks. It simply means that the resources (locks) that were held and requested by the threads did not result in a deadlock during a given run. If the timing changes in other runs, an actual deadlock can occur.

Figure 4 shows a snippet of a "dining philosophers" program parallelized using the POSIX threads API. In the program, five philosophers (threads) are seated around a table to eat. There are five chopsticks on the table (one between each pair of philosophers). A philosopher must grab the two chopsticks on his left and right to eat. A chopstick is a shared resource represented by a lock in the program.

```
void
grab_chopstick (int phil,
                int c,
                char *hand)
{
    pthread_mutex_lock (&chopstick[c]);
    printf ("Philosopher %d: got %s chopstick %d\n", phil, hand, c);
}

void
down_chopsticks (int c1,
                 int c2)
{
    pthread_mutex_unlock (&chopstick[c1]);
    pthread_mutex_unlock (&chopstick[c2]);
}
```

Figure 4: Code snippet from the "dining philosophers" program showing calls to POSIX lock and unlock routines.

A version of the dining philosophers program with a potential deadlock is executed. Thread Analyzer detects the deadlock and reports it, as shown in the Figure 5 and Figure 6.

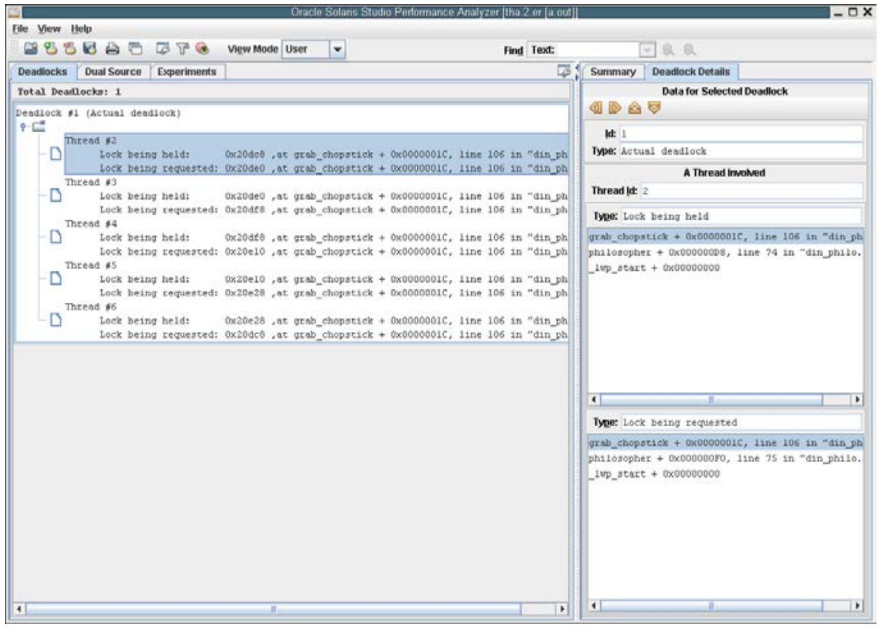

Figure 5: The Deadlocks tab in the Thread Analyzer GUI lists actual and potential deadlocks.

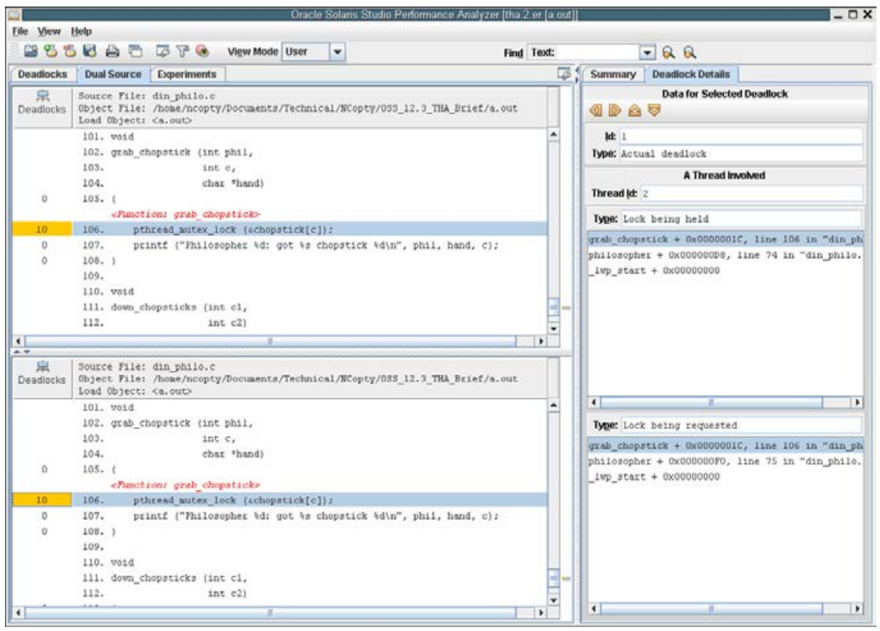

Figure 6: The Dual Source tab in the Thread Analyzer GUI pinpoints which locking operations cause a deadlock.

Find and Fix Issues Faster

The Oracle Solaris Thread Analyzer contains several features that make it easier for organizations to deploy multithreaded applications quickly and obtain maximum value from server investments. Features in the Oracle Solaris Studio Thread Analyzer help developers reduce the amount of time spent debugging multithreaded application code.

Using the Oracle Solaris Studio Thread Analyzer, it is possible to detect both data races and deadlocks at the same time. To do so, users should follow the steps to detect data races, but run the program using the collect command with the `-r race,deadlock` flag (or `-r all` for short). Any data races or deadlocks identified are recorded in the experiment.

Oracle Provides Complete Development Environments

Oracle is the only provider of complete development environments optimized for Oracle hardware and available for both Oracle Solaris and Linux operating systems. With a suite of fully integrated and optimized tools, including compilers, debuggers, performance and thread analysis tools, performance libraries, and more, Oracle provides a next-generation integrated development environment.

Using these tools, developers can bring high-performance, high-quality, standards-based enterprise Oracle Solaris applications to market faster, obtain maximum value from server investments, and increase ROI.

Contact Us

For more information about Oracle Solaris Studio, visit oracle.com/goto/solarisstudio or call +1.800.ORACLE1 to speak to an Oracle representative.