ORACLE
SOLARIS

An Oracle White Paper
November 2010

Taking Advantage of OpenMP 3.0 Tasking with Oracle® Solaris Studio

ORACLE

Introduction

OpenMP is an emerging standard model for parallel programming in a shared memory environment. Providing more than simple parallelization, OpenMP lets programmers specify shared memory parallelism in Fortran, C, and C++ programs. Several flexible and powerful constructs are available to parallelize non-loop code in an application. Perhaps the most important feature added to OpenMP 3.0, the *tasking model* allows arbitrary blocks of code to be executed in parallel. Developers use directives to define pieces of work, called *tasks*, that can execute concurrently.

Oracle Solaris Studio 12.2 software provides all the compilers and tools developers need to generate scalable, secure, and reliable enterprise applications. The software fully supports the OpenMP 3.0 specification, including the tasking feature. This technical white paper shows how to use Oracle Solaris Studio 12.2 to implement, profile, and debug an example quick sort (`qsort`) OpenMP program.

Start with a Serial Version of the Program

Consider the following `qsort_seq` application.

SOURCE CODE FOR THE SEQUENTIAL VERSION OF QUICK SORT (qsort_seq.c)

```
#include <stdlib.h>
#include <stdio.h>
#include <math.h>
#include <sys/time.h>

int
floatcompare (const void *p1,
              const void *p2)
{
float i = *((float *)p1);
float j = *((float *)p2);

if (i > j)
  return (1);
if (i < j)
  return (-1);
return (0);
}

int
partition (int p,
           int r,
           float *data)
{
float x = data[p];
int k = p;
int l = r + 1;
float t;

while (1) {
  do
    k++;
  while ((data[k] <= x) && (k < r));
  do
```

```
    l--;

  while (data[l] > x);
  while (k < l) {
    t = data[k];
    data[k] = data[l];
    data[l] = t;
    do
       k++;
    while (data[k] <= x);
    do
       l--;
    while (data[l] > x);
  }
  t = data[p];
  data[p] = data[l];
  data[l] = t;
  return l;
}
}

void
seq_quick_sort (int p,
               int r,
               float *data)
{
if (p < r) {
  int q = partition (p, r, data);
  seq_quick_sort (p, q - 1, data);
  seq_quick_sort (q + 1, r, data);
}
}

void
quick_sort (int p,
           int r,
           float *data,
           int low_limit)
{
```

```
if (p < r) {
  if ((r - p) < low_limit) {
    seq_quick_sort (p, r, data);
  }
  else {
    int q = partition (p, r, data);
    quick_sort (p, q - 1, data, low_limit);
    quick_sort (q + 1, r, data, low_limit);
  }
}
}

void
validate_sort (int n,
               float *data)
{
int i;
for (i = 0; i < n - 1; i++) {
  if (data[i] > data[i+1]) {
    printf ("Validate failed\n");
  }
}
printf ("Validate passed\n");
}

int
main (int argc, char *argv[])
{
int i, n, low_limit;
float *data;
hrtime_t start, end;
if (argc != 3) {
  printf ("a.out num_elems low_limit\n");
  return 1;
}
n = atoi(argv[1]);
low_limit = atoi(argv[2]);

/*
```

```
 * Generate the array.
 */
data = (float *)malloc (sizeof (float) * n);
for ( i=0; i<n; i++ ) {
  data[i] = 1.1 * rand() * 5000 / RAND_MAX;
}

printf ("\nSorting %d numbers seqentially...\n\n", n);

start = gethrtime();
quick_sort (0, n - 1, &data[0], low_limit);
end = gethrtime();

printf ("Time: %lld ms\n", (end - start)/1000000);
printf ("Done\n");

validate_sort (n, &data[0]);
free (data);
return 0;
}
```

The first step is to compile and run this serial version of the `qsort` program. The program takes two integer parameters: `num_elems` and `low_limit`. The `num_elems` parameter specifies the size of the array to be sorted. The values of the array elements are generated randomly by the program. The `low_limit` parameter has no impact on the performance of the serial version and is ignored. It was added to be consistent with the OpenMP version.

1. Copy the `qsort_seq` application code and save it in a file named `qsort_seq.c`.
2. Compile `qsort_seq.c` with optimization level `-xO3`. Add the `-g` to option to create an object file that contains debug information, such as line numbers.

```
% CC -xO3 -g -o qsort_seq qsort_seq.c
```

3. Run `qsort_seq`, specifying a `num_elems` value of 5000000 and a `low_limit` value of 100. Note that the sequential version of `qsort` (`qsort_seq`) takes 947 milliseconds to run.

```
% qsort_seq 5000000 100
Sorting 5000000 numbers seqentially...
Time: 947 ms
Done
Validate passed
```

First Attempt at Parallelization: Use OpenMP Sections Directives

Begin the parallelization effort by using OpenMP directives. The main routine in the `qsort_seq.c` source code calls the `quick_sort` function. In the `quick_sort` function, the original array is partitioned into two parts. Each part is handled by the `quick_sort` function recursively. Since the two parts of the array are manipulated independently, work can execute concurrently by using OpenMP parallel sections. For example, consider a system containing four processor cores. Since the `quick_sort` function only has two branches, nested parallel sections can be used to exploit more parallelism.

Source code for the OpenMP program using parallel sections (`qsort_section.c`) is shown below.

SOURCE CODE FOR THE PARALLEL SECTIONS VERSION OF QUICK SORT (qsort_section.c)

```
#include <stdlib.h>
#include <stdio.h>
#include <math.h>
#include <sys/time.h>

int
floatcompare (const void *p1,
             const void *p2)
{
float i = *((float *)p1);
float j = *((float *)p2);

if (i > j)
  return (1);
if (i < j)
  return (-1);
return (0);
}

int
partition (int p,
          int r,
          float *data)
{
float x = data[p];
int k = p;
int l = r + 1;
float t;
while (1) {
  do
    k++;
  while ((data[k] <= x) && (k < r));
  do
    l--;

  while (data[l] > x);
  while (k < l) {
    t = data[k];
    data[k] = data[l];
    data[l] = t;
    do
        k++;
    while (data[k] <= x);
    do
        l--;
    while (data[l] > x);
  }
  t = data[p];
  data[p] = data[l];
  data[l] = t;
  return l;
}
}

void
seq_quick_sort (int p,
               int r,
               float *data)
{
if (p < r) {
  int q = partition (p, r, data);
  seq_quick_sort (p, q - 1, data);
  seq_quick_sort (q + 1, r, data);
}
}
```

```
void
quick_sort (int p,
            int r,
            float *data,
            int low_limit)
{
if (p < r) {
  if ((r - p) < low_limit) {
    seq_quick_sort (p, r, data);
  }
  else {
    int q = partition (p, r, data);

    #pragma omp parallel sections firstprivate(data, p, q, r)
    {
      #pragma omp section
      quick_sort (p, q - 1, data, low_limit);

      #pragma omp section
      quick_sort (q + 1, r, data, low_limit);
    }
  }
}
}

void
validate_sort (int n,
               float *data)
{
int i;
for (i = 0; i < n - 1; i++) {
  if (data[i] > data[i+1]) {
    printf ("Validate failed\n");
  }
}
printf ("Validate passed\n");
}
```

```
int
main (int argc, char *argv[])
{
int i, n, low_limit;
float *data;
hrtime_t start, end;
if (argc != 3) {
  printf ("a.out num_elems low_limit\n");
  return 1;
}
n = atoi(argv[1]);
low_limit = atoi(argv[2]);

/*
 * Generate the array.
 */
data = (float *)malloc (sizeof (float) * n);
for ( i=0; i<n; i++ ) {
  data[i] = 1.1 * rand() * 5000 / RAND_MAX;
}

printf ("\nSorting %d numbers using OpenMP sections...\n\n", n);

start = gethrtime();
quick_sort (0, n - 1, &data[0], low_limit);
end = gethrtime();

printf ("Time: %lld ms\n", (end - start)/1000000);
printf ("Done\n");

validate_sort (n, &data[0]);
free (data);
return 0;
}
```

1. Copy the code for the qsort_section application listed above and save it in a file named `qsort_section.c`.
2. Compile `qsort_section.c` with the `–xopenmp` option to enable OpenMP parallelization.

```
% CC -xO3 -g -xopenmp -o qsort_section qsort_section.c
```

3. Set the OpenMP environment variables as shown below. Refer to the OpenMP specification for a description of these environment variables.

```
% setenv OMP_NUM_THREADS 2
% setenv OMP_THREAD_LIMIT 16
% setenv OMP_NESTED TRUE
% setenv OMP_WAIT_POLICY ACTIVE
% setenv OMP_DYNAMIC FALSE
```

4. Run the `qsort_section` program and specify `a num_elems` value of 5000000 and a `low_limit` value of 100. Example output from a run on an eight-core machine is shown below.

```
% qsort_section 5000000 100
Sorting 5000000 numbers using OpenMP sections...
Time: 866  ms
Done
Validate passed
```

Since `OMP_THREAD_LIMIT` is set to 16, only 16 threads are used to run the `qsort_section` program. On an eight-core machine, the runtime is 866 milliseconds. While this is an improvement over the time taken by the sequential version, the performance gain is less than expected.

Analyze the Parallelized Implementation

Oracle Solaris Studio includes the Oracle Solaris Studio Performance Analyzer, a tool developers can use to determine where time is being spent in applications. Metrics such as timing, hardware counters, and more are presented in an easy to understand source code view, helping developers to identify potential performance problems and locate the affected areas in the source code. These tools can be used to understand why the parallelized implementation of the `qsort_parreg` program does not scale as expected.

To analyze the performance of the `qsort_section` program, perform the following steps.

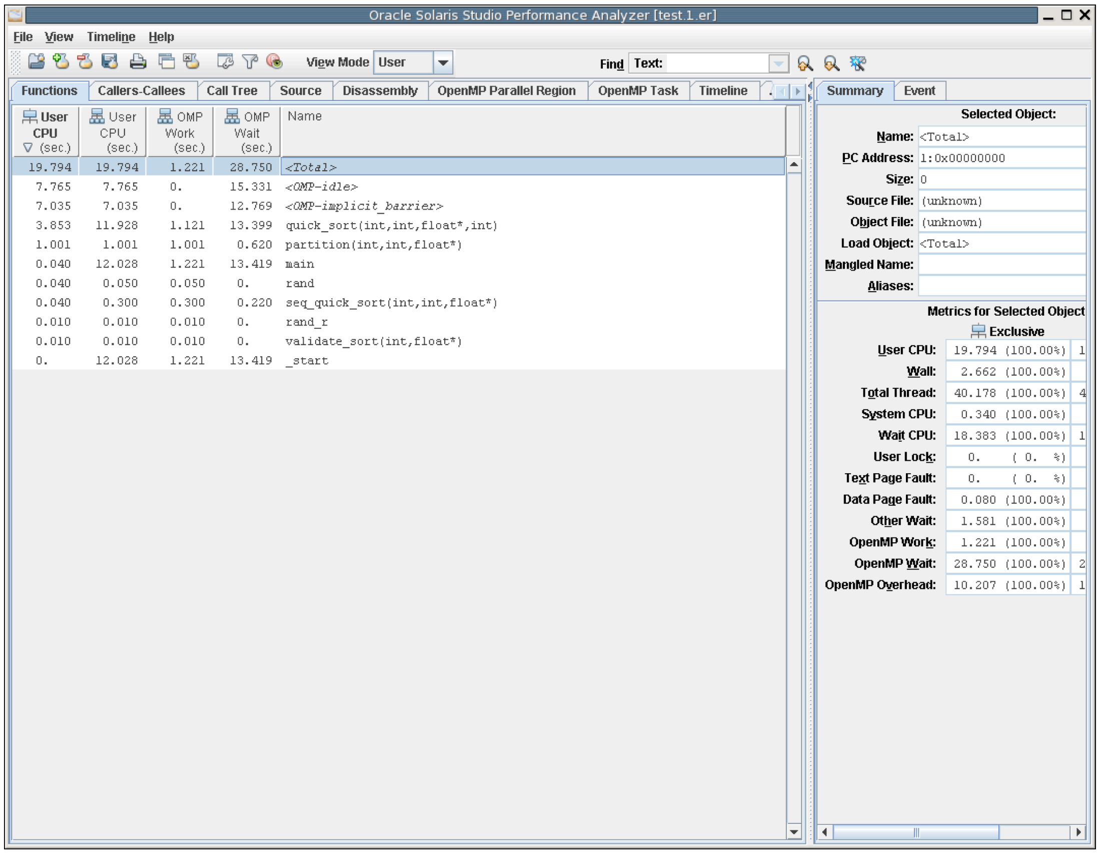
1. Be sure the environment variables are set as noted above. Run the `qsort_section` program under collect. This creates an Oracle Solaris Studio Performance Analyzer experiment called `test.1.er`.

```
% collect qsort_section 5000000 100
Creating experiment database test.1.er ...
Sorting 5000000 numbers using OpenMP sections...
Time: 2528 ms
Done
Validate passed
```

2. Examine the experiment result by invoking the Oracle Solaris Studio Performance Analyzer GUI with `test.1.er` as an argument.

```
% analyzer test.1.er
```

3. Click on the Functions tab and look at the Total statistics for the `OMP-wait` and `OMP-work` metrics (Figure 1). Time is accumulated in the `OMP-work` counter whenever a thread is executing user code. Time is accumulated in the `OMP-wait` counter whenever a thread is waiting for something before it can proceed, regardless of whether the thread is spinning or sleeping while waiting.



User CPU ▽ (sec.)	User CPU (sec.)	OMP Work (sec.)	OMP Wait (sec.)	Name
19.794	19.794	1.221	28.750	*<Total>*
7.765	7.765	0.	15.331	*<OMP-idle>*
7.035	7.035	0.	12.769	*<OMP-implicit_barrier>*
3.853	11.928	1.121	13.399	quick_sort(int,int,float*,int)
1.001	1.001	1.001	0.620	partition(int,int,float*)
0.040	12.028	1.221	13.419	main
0.040	0.050	0.050	0.	rand
0.040	0.300	0.300	0.220	seq_quick_sort(int,int,float*)
0.010	0.010	0.010	0.	rand_r
0.010	0.010	0.010	0.	validate_sort(int,float*)
0.	12.028	1.221	13.419	_start

Selected Object:

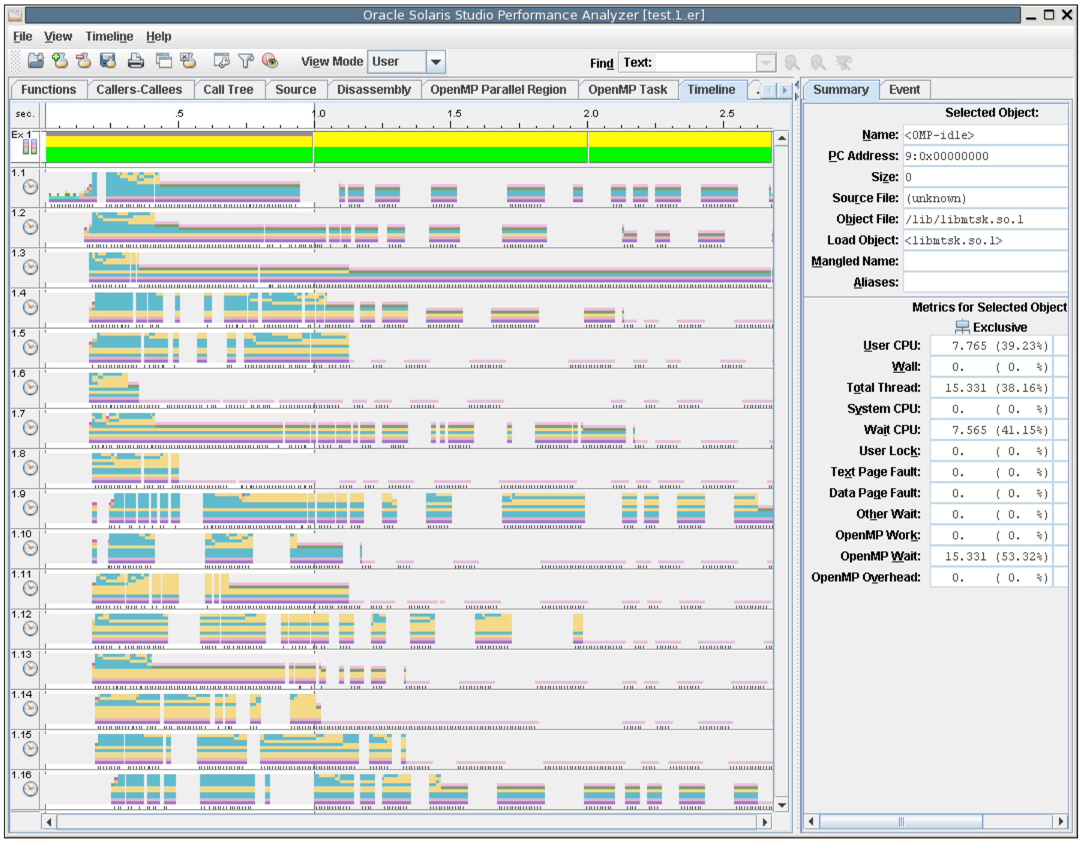
Name:	<Total>
PC Address:	1:0x00000000
Size:	0
Source File:	(unknown)
Object File:	(unknown)
Load Object:	<Total>
Mangled Name:	
Aliases:	

Metrics for Selected Object

	Exclusive
User CPU:	19.794 (100.00%)
Wall:	2.662 (100.00%)
Total Thread:	40.178 (100.00%)
System CPU:	0.340 (100.00%)
Wait CPU:	18.383 (100.00%)
User Lock:	0. (0. %)
Text Page Fault:	0. (0. %)
Data Page Fault:	0.080 (100.00%)
Other Wait:	1.581 (100.00%)
OpenMP Work:	1.221 (100.00%)
OpenMP Wait:	28.750 (100.00%)
OpenMP Overhead:	10.207 (100.00%)

Figure 1. The Oracle Solaris Studio Performance Analyzer graphical user interface

4. Notice that the Total `OMP-wait` time is much larger than the `OMP-work` time. This means that threads spent significantly more time waiting than executing user code. Take a closer look at the `OMP-wait` value for each function. Doing so reveals that a large portion of the `OMP-wait` time comes from the `OMP-implicit_barrier` function.
5. Click on the Timeline tab. A picture of the threads timeline status is shown (Figure 2). There are 16 timelines, one for each of the 16 threads. A timeline shows what the thread stack looked like during program execution.



Selected Object:

Name:	<OMP-idle>
PC Address:	9:0x00000000
Size:	0
Source File:	(unknown)
Object File:	/lib/libmtsk.so.1
Load Object:	<libmtsk.so.1>
Mangled Name:	
Aliases:	

Metrics for Selected Object

	Exclusive
User CPU:	7.765 (39.23%)
Wall:	0. (0. %)
Total Thread:	15.331 (38.16%)
System CPU:	0. (0. %)
Wait CPU:	7.565 (41.15%)
User Lock:	0. (0. %)
Text Page Fault:	0. (0. %)
Data Page Fault:	0. (0. %)
Other Wait:	0. (0. %)
OpenMP Work:	0. (0. %)
OpenMP Wait:	15.331 (53.32%)
OpenMP Overhead:	0. (0. %)

Figure 2. The Oracle Solaris Studio Performance Analyzer shows what the thread stack looked like during program execution.

Clicking on any point in the timeline displays in the right-hand panel what the thread stack looked like at that point in time. The function at the top of the stack is the function the thread was executing at that time. Looking at the timelines verifies that some threads spend a lot of time in the `OMP-implicit_barrier` function, while other threads are working on `quick_qsort()`. This means the load is imbalanced. Some threads do not have enough useful work to do and wait at the barrier at the end of a parallel region.

The load imbalance stems from the inflexibility of nested parallel regions. In the `qsort` program, partitioning splits the array into two parts that likely are not the same size. In the parallel sections model, a thread that finishes its job must wait at the barrier for other threads to finish. While waiting, a thread does not perform useful work and wastes resources.

Improve Scaling with OpenMP Tasking Directives

This section shows how OpenMP 3.0 tasking can improve the scaling the parallelized version of `qsort`. The source code for the tasking version of the application, `qsort_task.c`, is shown below.

SOURCE CODE FOR THE TASKING VERSION OF QUICK SORT (qsort_task.c)

```
#include <stdlib.h>
#include <stdio.h>
#include <math.h>
#include <sys/time.h>

int
floatcompare (const void *p1,
              const void *p2)
{
float i = *((float *)p1);
float j = *((float *)p2);

if (i > j)
  return (1);
if (i < j)
  return (-1);
return (0);
}

int
partition (int p,
           int r,
           float *data)
{
float x = data[p];
int k = p;
int l = r + 1;
float t;

while (1) {
  do
    k++;
  while ((data[k] <= x) && (k < r));
  do
    l--;
  while (data[l] > x);
  while (k < l) {
    t = data[k];
    data[k] = data[l];
    data[l] = t;
    do
       k++;
    while (data[k] <= x);
    do
       l--;
    while (data[l] > x);
  }
  t = data[p];
  data[p] = data[l];
  data[l] = t;
  return l;
}
}

void
seq_quick_sort (int p,
               int r,
               float *data)
{
if (p < r) {
  int q = partition (p, r, data);
  seq_quick_sort (p, q - 1, data);
  seq_quick_sort (q + 1, r, data);
}
}

void
quick_sort (int p,
           int r,
           float *data,
           int low_limit)
{
if (p < r) {
  if ((r - p) < low_limit) {
    seq_quick_sort (p, r, data);
  }
  else {
    int q = partition (p, r, data);

    #pragma omp task firstprivate(data, low_limit, r, q)
    quick_sort (p, q - 1, data, low_limit);
    #pragma omp task firstprivate(data, low_limit, r, q)
    quick_sort (q + 1, r, data, low_limit);
  }
}
}

void
par_quick_sort (int n,
               float *data,
               int low_limit)
{
#pragma omp parallel
{
  #pragma omp single nowait
  quick_sort (0, n, data, low_limit);
}
```

```
}

void
validate_sort (int n,
              float *data)
{
int i;
for (i = 0; i < n - 1; i++) {
  if (data[i] > data[i+1]) {
    printf ("Validate failed\n");
  }
}
printf ("Validate passed\n");
}

int
main (int argc, char *argv[])
{
int i, n, low_limit;
float *data;
hrtime_t start, end;
if (argc != 3) {
  printf ("a.out num_elems low_limit\n");
  return 1;
}
n = atoi(argv[1]);
low_limit = atoi(argv[2]);

/*
 * Generate the array.
 */
data = (float *)malloc (sizeof (float) * n);
for ( i=0; i<n; i++ ) {
  data[i] = 1.1 * rand() * 5000 / RAND_MAX;
}
printf ("\nSorting %d numbers using OpenMP tasks...\n\n", n);
start = gethrtime();
par_quick_sort (n - 1, &data[0], low_limit);
end = gethrtime();
printf ("Time: %lld ms\n", (end - start)/1000000);
printf ("Done\n");

validate_sort (n, &data[0]);
free (data);
return 0;
}
```

The `par_quick_sort` function has a parallel construct that contains a single construct. In the single construct, there is a call to the `quick_sort` function. Two tasks are generated in the `quick_sort` function. The `quick_sort` function is called recursively, causing many tasks to be generated until the low limit threshold is reached.

The execution model of the `qsort_task` program can be described as a single producer, multiple consumer model. The thread executing the single region generates tasks; the threads in the team execute these tasks. All the tasks generated are guaranteed to complete by the time the threads exit the single region. When a thread finishes executing a task, it grabs a new task to execute. In this way, all threads can execute available tasks without barrier synchronization, thereby improving load balancing.

1. Copy the `qsort_task` application code listed above and save it in a file named `qsort_task.c`.
2. Compile `qsort_task.c` with the `–xopenmp` option to enable OpenMP parallelization.

```
% CC -xO3 -g -xopenmp -o qsort_task qsort_task.c
```

3. Set the OpenMP environment variables as shown below. Refer to the OpenMP specification for a description of these environment variables. There is only one parallel region in the qsort_task program. Four threads are used for the parallel region and for executing all tasks. Since there are no nested parallel regions, there is no need to limit the number of threads by setting the `OMP_THREAD_LIMIT` variable.

```
% setenv OMP_NUM_THREADS 4
% setenv OMP_WAIT_POLICY ACTIVE
% setenv OMP_DYNAMIC FALSE
```

4. Run the `qsort_task` program, specifying a `num_elems` value of 5000000 and a `low_limit` value of 100. Example output from a run on an eight-core system is shown below. The runtime is 525 milliseconds. This a significant improvement over the time taken by the parallel sections version, even though only four threads were used instead of 16 threads.

```
% qsort_task 5000000 100
Sorting 5000000 numbers using OpenMP tasks...
Time: 525 ms
Done
Validate passed
```

When `OMP_NUM_THREADS` is set to 2 for the `qsort_section` program, each parallel region tries to get two threads. Since there are nested parallel regions, each inner parallel region tries to get two threads. Eventually all 16 threads are created and used—yet the runtime is longer than that of the `qsort_task` program. The recursive sort tree is unbalanced. Some threads perform more work than other threads. A thread may finish its work early and wait at the barrier even though it could help other threads.

Note that setting `OMP_NUM_THREADS` to 2 for the `qsort_section` program is the most efficient way to use the threads available, since the degree of parallelism in each parallel region is two. If `OMP_NUM_THREADS` is set to a value higher than 2, threads in excess of 2 are not used—they simply wait at the barrier of the parallel region for the other two threads to finish. The tasking version does not exhibit this problem. Any thread can work on any task, and threads can be used efficiently. Because there is only one parallel region, developers can control how many threads are used by setting the `OMP_NUM_THREADS` environment variable.

Perform the following steps to analyze the performance of the tasking version.

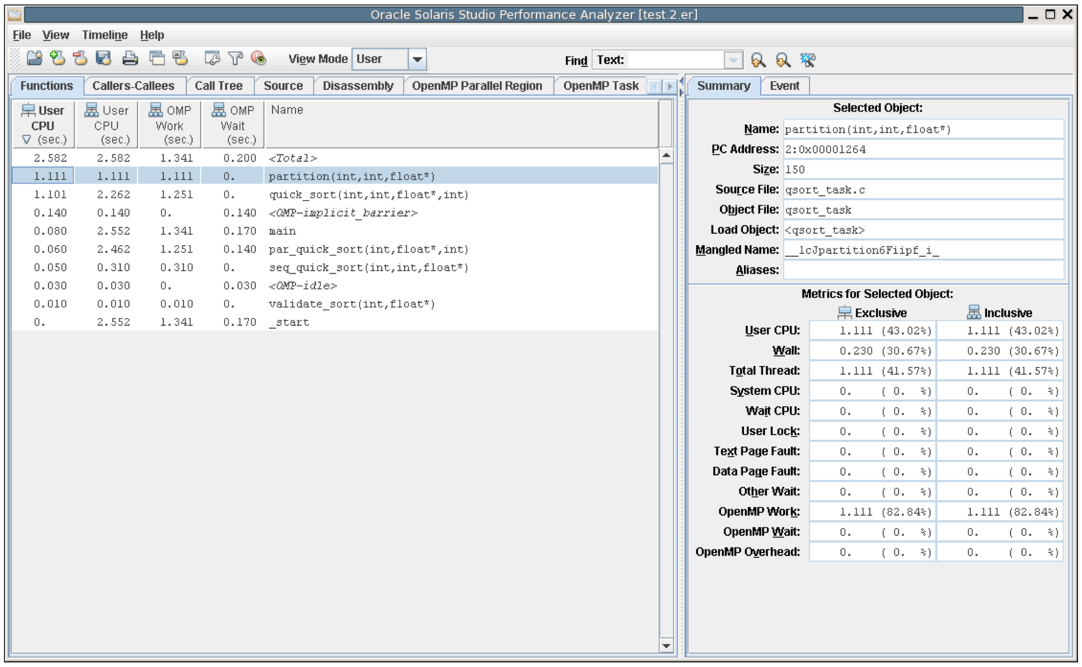
1. Run the `qsort_section` under collect with the environment variables set as noted above. This creates an Oracle Solaris Studio Performance Analyzer experiment named `test.2.er`.

```
% collect qsort_task 5000000 100
Creating experiment database test.2.er ...
Sorting 5000000 numbers using OpenMP tasks...
Time: 630 ms
Done
Validate passed
```

2. Examine the experiment by invoking the analyzer GUI with test.2.er as an argument.

```
% analyzer test.2.er
```

3. Click on the Functions tab and look at the Total statistics for the `OMP-wait` and `OMP-work` metrics (Figure 3). Notice that the Total `OMP-work` time is larger than the `OMP-wait` time. This means that threads spent more time executing user code. The `OMP-implicit-barrier` overhead is almost eliminated, as threads are not wasting time on unnecessary synchronization and the load is balanced among the threads.



User CPU (sec.)	User CPU (sec.)	OMP Work (sec.)	OMP Wait (sec.)	Name
2.582	2.582	1.341	0.200	*<Total>*
1.111	1.111	1.111	0.	partition(int,int,float*)
1.101	2.262	1.251	0.	quick_sort(int,int,float*,int)
0.140	0.140	0.	0.140	*<OMP-implicit_barrier>*
0.080	2.552	1.341	0.170	main
0.060	2.462	1.251	0.140	par_quick_sort(int,float*,int)
0.050	0.310	0.310	0.	seq_quick_sort(int,int,float*)
0.030	0.030	0.	0.030	*<OMP-idle>*
0.010	0.010	0.010	0.	validate_sort(int,float*)
0.	2.552	1.341	0.170	_start

Selected Object:

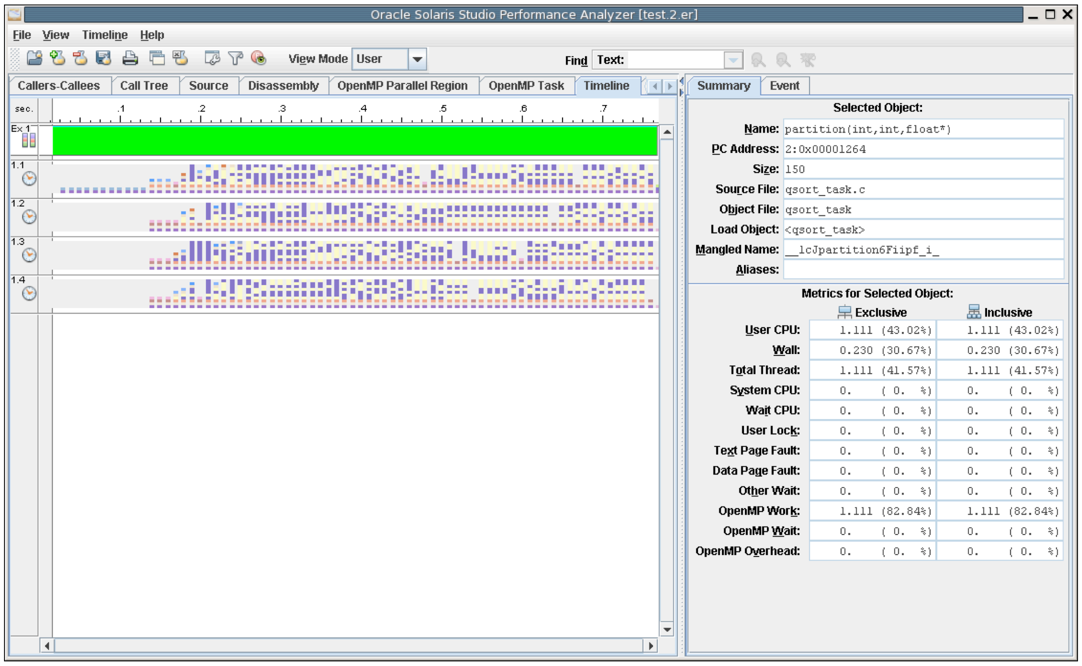
Name:	partition(int,int,float*)
PC Address:	2:0x00001264
Size:	150
Source File:	qsort_task.c
Object File:	qsort_task
Load Object:	<qsort_task>
Mangled Name:	__1cJpartition6Fiipf_i_
Aliases:	

Metrics for Selected Object:

	Exclusive	Inclusive
User CPU:	1.111 (43.02%)	1.111 (43.02%)
Wall:	0.230 (30.67%)	0.230 (30.67%)
Total Thread:	1.111 (41.57%)	1.111 (41.57%)
System CPU:	0. (0. %)	0. (0. %)
Wait CPU:	0. (0. %)	0. (0. %)
User Lock:	0. (0. %)	0. (0. %)
Text Page Fault:	0. (0. %)	0. (0. %)
Data Page Fault:	0. (0. %)	0. (0. %)
Other Wait:	0. (0. %)	0. (0. %)
OpenMP Work:	1.111 (82.84%)	1.111 (82.84%)
OpenMP Wait:	0. (0. %)	0. (0. %)
OpenMP Overhead:	0. (0. %)	0. (0. %)

Figure 3. The Functions tab shows statistics for metrics

4. Click on the `Timeline` tab. A picture of the threads timelines is shown (Figure 4). There are four timelines, one for each of the four threads. Looking at the timelines confirms that the load in the tasking version is balanced among the threads.



Selected Object:

Name:	partition(int,int,float*)
PC Address:	2:0x00001264
Size:	150
Source File:	qsort_task.c
Object File:	qsort_task
Load Object:	<qsort_task>
Mangled Name:	__1cJpartition6Fiipf_i_
Aliases:	

Metrics for Selected Object:

	Exclusive	Inclusive
User CPU:	1.111 (43.02%)	1.111 (43.02%)
Wall:	0.230 (30.67%)	0.230 (30.67%)
Total Thread:	1.111 (41.57%)	1.111 (41.57%)
System CPU:	0. (0. %)	0. (0. %)
Wait CPU:	0. (0. %)	0. (0. %)
User Lock:	0. (0. %)	0. (0. %)
Text Page Fault:	0. (0. %)	0. (0. %)
Data Page Fault:	0. (0. %)	0. (0. %)
Other Wait:	0. (0. %)	0. (0. %)
OpenMP Work:	1.111 (82.84%)	1.111 (82.84%)
OpenMP Wait:	0. (0. %)	0. (0. %)
OpenMP Overhead:	0. (0. %)	0. (0. %)

Figure 4. The Timeline tab shows the execution timelines for threads

View the Call Stack

Because the compiler has the ability to parallelize sections of code, developers are often at a disadvantage when debugging applications. The compiler inserts instructions behind the scenes, and the executed code often is very different than what was written. To help this effort, Oracle Solaris Studio Performance Analyzer includes the ability to show the logical stack for parallel regions and tasks. Take a look at the call stack for the `qsort_task` program.

1. Select Machine from the View Mode drop-down menu at the top of the screen.
2. Look at the Timeline View in Machine mode.
3. Click on a position in the timeline bar for a thread. An event can be seen on the right hand side, along with its call stack and runtime library frames. Note that the stack for slave threads starts at the `_lwp_start` function. These implementation details are difficult to decipher.
4. Switch to User mode to reconstruct the call stack as if the OpenMP runtime library routines were not called. The resulting call stack exhibits the exact logical call tree of the program—without any extra implementation details. Even when OpenMP task execution is deferred until the implicit barrier of the single region, the call stack appears as if the task executed immediately. In addition, slave thread call stacks start from the `main` routine, just like the main application thread.

Debug OpenMP Tasks

Oracle Solaris Studio includes a `dbxtool` debugger that can help developers to view code errors and facilitate the correction of problematic multithreaded code. The following steps outline how to take advantage of debugging support to examine variables of a task region.

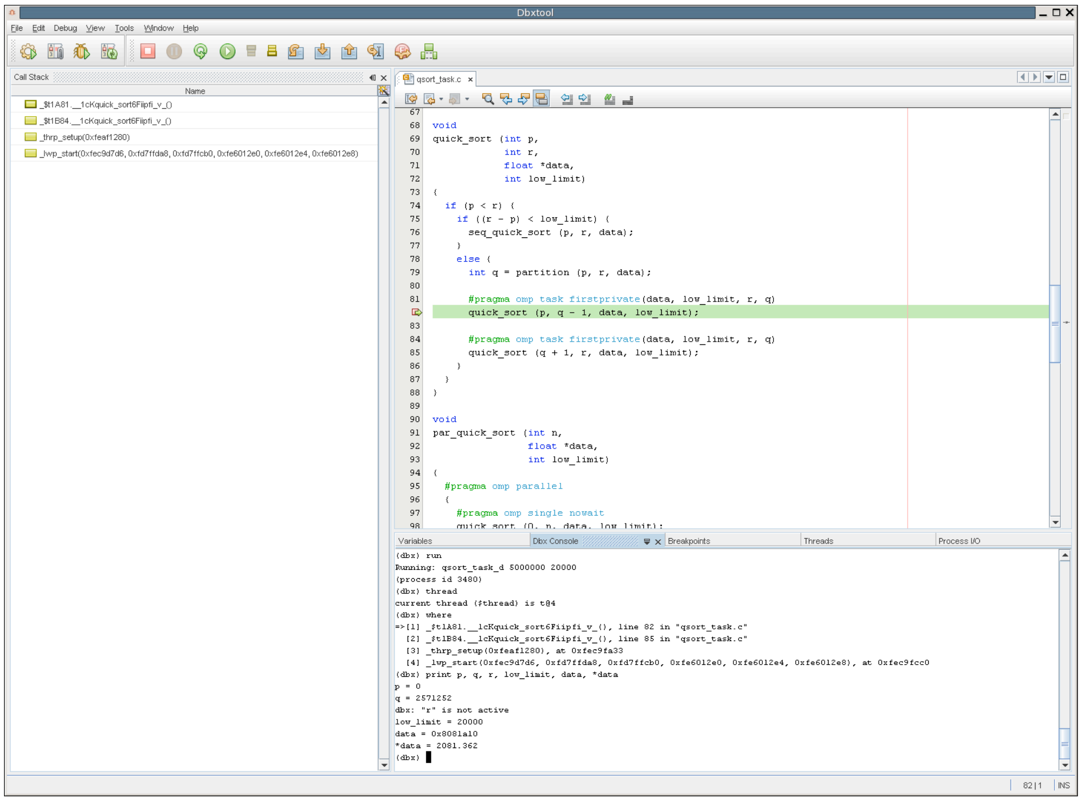
1. Compile the qsort_task.c file with the `–xopenmp=noopt` and `–g` options to prepare the application for debugging.

   ```
   % CC -xO3 -g –xopenmp=noopt -o qsort_task qsort_task.c
   ```

2. Start the `dbxtool` debugger.

   ```
   % dbxtool
   ```

3. Select Debug Executable from the Debug drop-down menu. Specify the Debug Target (`qsort_task`), Arguments (5000000 100), and Environment (`OMP_NUM_THREADS=4`, `OMP_WAIT_POLICY=ACTIVE`) in the pop-up window. Click the Debug button to close the window and start the debugging session.
4. In the source window for `qsort_task.c`, set a breakpoint in the task construct on the line after the first task directive (`quick_sort (p, q-1, data, low_limit`). To set the breakpoint, move the cursor to the line. Use `Ctrl-F8` or `Debug-click` on the line number to set the breakpoint.
5. Debugger commands can be typed in the Dbx Console located at the bottom of the screen. The buttons at the top of the screen can be used. To run the program, type `run` in the Dbx Console or press the Run button at the top of the screen.
6. The program stops at the breakpoint just set, in a task region. The `omp_pr` and `omp_tr` commands can be used to print parallel region and task region information, respectively. Based on the data sharing attribute clause, `p`, `q`, `low_limit`, and `data` are all *firstprivate* variables of this task region. The `print` command can be used to check their values (Figure 5).



```
_$t1A81.__1cKquick_sort6Fiipfi_v_()
_$t1B84.__1cKquick_sort6Fiipfi_v_()
_thrp_setup(0xfeaf1280)
_lwp_start(0xfec9d7d6, 0xfd7ffda8, 0xfd7ffcb0, 0xfe6012e0, 0xfe6012e4, 0xfe6012e8)
```

```

void
quick_sort (int p,
            int r,
            float *data,
            int low_limit)
{
  if (p < r) {
    if ((r - p) < low_limit) {
      seq_quick_sort (p, r, data);
    }
    else {
      int q = partition (p, r, data);

      #pragma omp task firstprivate(data, low_limit, r, q)
      quick_sort (p, q - 1, data, low_limit);

      #pragma omp task firstprivate(data, low_limit, r, q)
      quick_sort (q + 1, r, data, low_limit);
    }
  }
}

void
par_quick_sort (int n,
                float *data,
                int low_limit)
{
  #pragma omp parallel
  {
    #pragma omp single nowait
    quick_sort (0, n, data, low_limit);
```

```
(dbx) run
Running: qsort_task_d 5000000 20000
(process id 3480)
(dbx) thread
current thread ($thread) is t@4
(dbx) where
=>[1] _$t1A81.__1cKquick_sort6Fiipfi_v_(), line 82 in "qsort_task.c"
  [2] _$t1B84.__1cKquick_sort6Fiipfi_v_(), line 85 in "qsort_task.c"
  [3] _thrp_setup(0xfeaf1280), at 0xfec9fa33
  [4] _lwp_start(0xfec9d7d6, 0xfd7ffda8, 0xfd7ffcb0, 0xfe6012e0, 0xfe6012e4, 0xfe6012e8), at 0xfec9fcc0
(dbx) print p, q, r, low_limit, data, *data
p = 0
q = 2571252
dbx: "r" is not active
low_limit = 20000
data = 0x8081a10
*data = 2081.362
(dbx)
```

Figure 5. The dbxtool debugger

For More Information

This technical white paper used the `qsort` application to demonstrate how to write a parallel program using mechanisms in earlier OpenMP versions—and how to rewrite the application to take advantage of OpenMP 3.0 tasking to improve performance. By using the Oracle Solaris Studio Performance Analyzer and debugging tools, readers gain insight into how to profile an OpenMP tasking program, and find and fix overhead hot spots. To learn more about Oracle Solaris Studio and the tasking features of OpenMP, see the references listed in Table 1.

TABLE 1. RESOURCES

Oracle Solaris	http://www.oracle.com/us/products/servers-storage/solaris/index.html
Oracle Solaris Studio	http://www.oracle.com/technetwork/server-storage/solarisstudio/overview/index.html
Oracle Solaris Studio OpenMP API User's Guide	http://docs.sun.com/app/docs/doc/819-5270
What's New in the Oracle Solaris Studio 12.2 Release	http://docs.sun.com/app/docs/doc/821-2414/gkexb?l=en&a=view
OpenMP Specification	http://openmp.org/wp/openmp-specifications/

ORACLE®

Oracle is committed to developing practices and products that help protect the environment

Taking Advantage of OpenMP 3.0 Tasking with
Oracle Solaris Studio
November 2010
Author: Nawal Copty

Oracle Corporation
World Headquarters
500 Oracle Parkway
Redwood Shores, CA 94065
U.S.A.

Worldwide Inquiries:
Phone: +1.650.506.7000
Fax: +1.650.506.7200
oracle.com

Copyright © 2010, Oracle and/or its affiliates. All rights reserved.
This document is provided for information purposes only and the contents hereof are subject to change without notice. This document is not warranted to be error-free, nor subject to any other warranties or conditions, whether expressed orally or implied in law, including implied warranties and conditions of merchantability or fitness for a particular purpose. We specifically disclaim any liability with respect to this document and no contractual obligations are formed either directly or indirectly by this document. This document may not be reproduced or transmitted in any form or by any means, electronic or mechanical, for any purpose, without our prior written permission.

Oracle and Java are registered trademarks of Oracle and/or its affiliates. Other names may be trademarks of their respective owners.

AMD, Opteron, the AMD logo, and the AMD Opteron logo are trademarks or registered trademarks of Advanced Micro Devices. Intel and Intel Xeon are trademarks or registered trademarks of Intel Corporation. All SPARC trademarks are used under license and are trademarks or registered trademarks of SPARC International, Inc. UNIX is a registered trademark licensed through X/Open Company, Ltd. 0310

SOFTWARE. HARDWARE. COMPLETE.