Parallelizing the tree-search programs using OpenMP

- The static and dynamic parallel tree-search programs using OpenMP are the same as the issues involved in implementing the programs using Pthreads.

- There are almost no substantive differences between a static implementation that
  uses OpenMP and one that uses Pthreads. However, a couple of points should be mentioned:
  1. When a single thread executes some code in the Pthreads version, the test
     ```
     if (my_rank == whatever)
     ```
     can be replaced by the OpenMP directive
     ```
     # pragma omp single
     ```

     - This will insure that the following structured block of code will be executed by one thread in the team, and the other threads in the team will wait in an implicit barrier at the end of the block until the executing thread is finished.

     - When **whatever** is 0 (as it is in each test in the Pthreads program), the test can also be replaced by the OpenMP directive

       ```
       # pragma omp master
       ```

  2. The Pthreads mutex that protects the best tour can be replaced by a single

     critical directive placed either inside the **Update_best_tour** function or immediately before the call to **Update_best_tour**. This is the only potential source of a race condition after the distribution of the initial tours, so the simple **critical** directive won't cause a thread to block unnecessarily.

- The dynamically load-balanced Pthreads implementation depends heavily on Pthreads condition variables, and OpenMP doesn't provide a comparable object. The rest of the Pthreads code can be easily converted to OpenMP. In fact, OpenMP even provides a nonblocking version of omp_set_lock. OpenMP provides a lock object omp_lock_t and the following functions for acquiring and relinquishing the lock, respectively:

```
void omp_set_lock(omp_lock_t*    lock_p   /* in/out */);
void omp_unset_lock(omp_lock_t*  lock_p   /* in/out */);
```

  It also provides the function

```
int omp_test_lock(omp_lock_t*    lock_p   /* in/out */);
```

- The simplest solution to emulating a condition wait in OpenMP is to use busy-waiting. Since there are two conditions a waiting thread should test for, we can
  use two different variables in the busy-wait loop:

```
/* Global variables */
int awakened_thread = -1;
int work_remains = 1;  /* true */
. . .
while (awakened_thread != my_rank && work_remains);
```

- If **awakened_thread** has the value of some thread's rank, that thread will exit immediately from the while, but there may be no work available. Similarly, if **work_ remains** is initialized to 0, all the threads will exit the while loop immediately and quit. we should relinquish the lock used in the **Terminated** function before starting the **while** loop.

- when a thread returns from a Pthreads condition wait, it reacquires the mutex associated with the condition variable. This is especially important in this setting since if the awakened thread has received work, it will need to access the shared data structures storing the new stack. Thus, our complete emulated condition
  wait should look something like this:

```
/* Global vars */
int awakened_thread = -1;
work_remains = 1;  /* true */
. . .
omp_unset_lock(&term_lock);
while (awakened_thread != my_rank && work_remains);
omp_set_lock(&term_lock);
```

- Emulating the condition broadcast is straightforward: When a thread determines
  that there's no work left (Line 14 in Program 6.8), then the condition broadcast (Line 17) can be replaced with the assignment

```
Work_remains = 0;   /* Assign false to work remais*/
```

- Emulating the condition signal requires a little more work. The thread that has split its stack needs to choose one of the sleeping threads and set the variable
  awakened thread to the chosen thread's rank. Thus, at a minimum, we need to keep a list of the ranks of the sleeping threads. A simple way to do this is to use a shared queue of thread ranks. When a thread runs out of work, it enqueues its rank before entering the busy-wait loop. When a thread splits its stack, it can choose the thread to awaken by dequeuing the queue of waiting threads:

```
got_lock = omp_test_lock(&term_lock);
if (got_lock != 0) {
   if (waiting_threads > 0 && new_stack == NULL) {
      Split my_stack creating new_stack;
      awakened_thread = Dequeue(term_queue);
   }
   omp_unset_lock(&term_lock);
}
```