

Programming Assignment 5

Due Wednesday, December 5

The Travelling Salesperson Problem

For programming assignment 5 you should implement a Pthreads tree search solution to the travelling salesperson problem (TSP). In TSP a salesperson has a list of cities she needs to visit and a cost for travelling between each pair of cities. The problem is to find a “tour” for the salesperson that minimizes the total cost of the trip. A tour is just an ordered list of all the cities that starts and ends with the salesperson’s hometown.

We’ll look at an especially simple solution that uses a simple form of tree search. The idea is that in searching for solutions, we build a *tree*. Leaves of the tree correspond to tours, and the other nodes correspond to “partial” tours — trips that have visited some, but not all, of the cities.

The serial algorithm we’ll base our program on is called *depth-first search*. The algorithm “branches left” in the tree until it can go no further, at which point, it backs up to the nearest ancestor with unvisited children and starts branching down to the left from the first or leftmost unvisited child. Figure 1 shows a four-city TSP. Figure 2 shows a search tree for the four-city TSP. 0 is the salesperson’s hometown, and the nodes of the tree are labelled with the partial tour and the cost of the partial tour.

We can use the costs of the nodes to reduce the size of the tree we actually search. When we find a partial tour or node of the tree that couldn’t possibly lead to a less expensive complete tour, we shouldn’t bother searching the children of that node. In our example, since we’re searching the tree from left to right, we shouldn’t bother checking the the node labelled “0 → 2 → 1, 21” since the cost of the partial tour 0 → 2 → 1 is 21, which is greater than the cost of the cheapest tour so far, which is 20. See the program `tsp_search.c` on the class website for details.

Our goal in program 5 is to parallelize this program using Pthreads. An obvious issue is how to divide the work. A simple solution (and the solution we’ll use) simply divides “short” partial tours among the threads. A potential problem with this approach is that some short tours may lead to a lot of work, while others may lead to very little. So load-balancing may be an issue. We’ll address this by using “charitable” threads. A thread with nodes to search can donate some of its nodes to threads that have run out of work.

In our serial solution, we use recursion. So the nodes of the tree are hidden on the system stack, and in order to implement a dynamic load balancing strategy, we need to “expose” the stack created by the recursive calls. That is, we want to implement an *iterative* solution, keeping track of the stack ourselves. The basic idea is to create our own stack storing the nodes of the tree. We’ve done this in the program `tsp_search_nr.c` on the website.

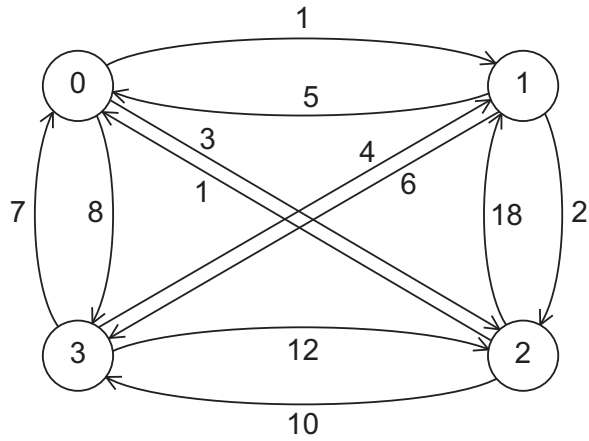


Figure 1: A four-city TSP (from *Introduction to Parallel Programming*, Morgan-Kaufmann, 2011)

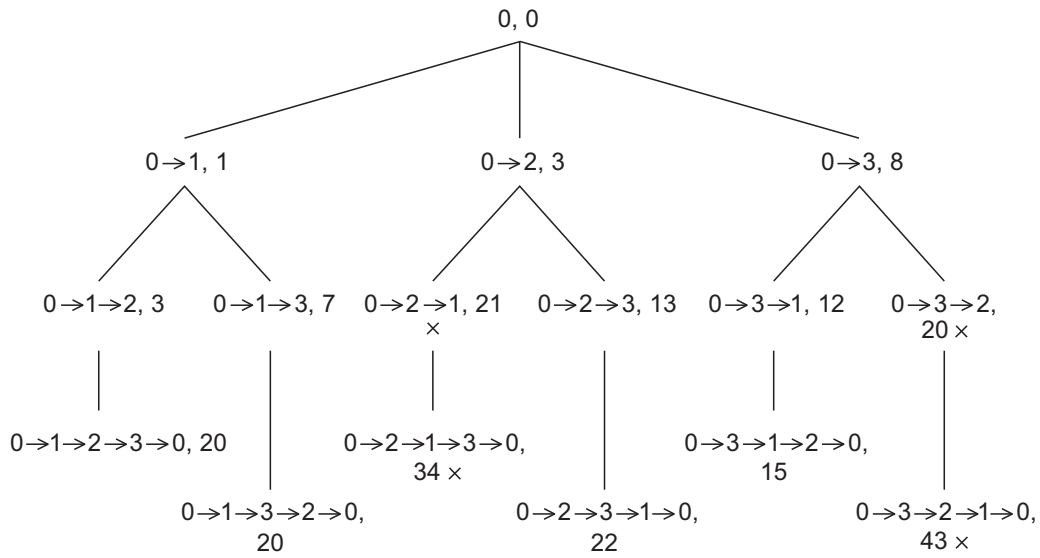


Figure 2: A search tree four-city TSP (from *Introduction to Parallel Programming*, Morgan-Kaufmann, 2011)

At the start of the `Search` function, we create a stack consisting of one node — the partial tour consisting only of 0, the hometown. Then we start a `while` loop that continues until our stack is empty. Each time we enter this loop, we “pop” a new node. This gives a new city, and we push all of its “feasible” children onto our stack.

The program on the website, `tsp_search_nr.c` uses a linked list for a stack. So the nodes of the stack have to be accessed sequentially. In our final parallel solution, it may be useful to be able to access the nodes directly. So it *might* be useful to implement the stack with an array instead.

First Parallel Solution

Now that we have a serial program for solving TSP, how do we get started on parallelizing it? As we mentioned earlier, we’ll partition “short” partial tours. In fact, we’ll partition partial tours with just 0 (the home town) and one other city, two-city partial tours. If our map or graph has n cities, then there will be $n - 1$ two-city partial tours. If $n - 1 \geq \text{thread_count}$, we can divide these partial tours so that every thread gets at least one node close to the root. Of course, if $n - 1 < \text{thread_count}$, this won’t work. But we expect to have considerably more cities than threads. So this isn’t an unreasonable assumption.

Assumption: number of cities $>$ number of threads.

So to start the search, we’ll partition the 2-city partial tours: each thread will get $\approx (n - 1)/\text{thread_count}$ partial tours. This initialization phase can be carried out completely in parallel.

Now how do we divide the 2-city partial tours? Let’s look at an example. Suppose we have 15 cities $(0, 1, \dots, 14)$, and 4 threads. Then there are 14 2-city partial tours:

$$0 \rightarrow 1, 0 \rightarrow 2, \dots, 0 \rightarrow 14$$

Since we have 4 threads, each thread should get roughly $14/4 = 3.5$ partial tours. Of course, we can’t assign half-tours. So some threads should get 4 and others should get 3. We can use modular arithmetic to decide. In C $14/4 = 3$ and $14 \% 4 = 2$. This says that

$$14 = 3 \cdot 4 + 2.$$

So we can assign each thread 3 partial tours, and there will be two left over. The two leftover can be assigned to the first two threads. So threads 0 and 1 each get four partial tours:

```
0: 0->1, 0->2, 0->3, 0->4
1: 0->5, 0->6, 0->7, 0->8
```

and threads 2 and 3 each get three:

```
2: 0->9, 0->10, 0->11
3: 0->12, 0->13, 0->14
```

In general, we can use the following algorithm.

```

quotient = (n-1)/p;
remainder = (n-1) % p;
if (my_rank < remainder) {
    partial_tour_count = quotient+1;
    first_final_city = my_rank*partial_tour_count + 1;
} else {
    partial_tour_count = quotient;
    first_final_city = my_rank*partial_tour_count + remainder + 1
}
last_final_city = first_final_city + partial_tour_count - 1;

```

Now in our first parallel solution, we'll simply have each thread search the tours starting with its assigned 2-city partial tours. The only "communication" between the threads will occur when they need to access the current best tour. So this data structure needs to be protected with a mutex or semaphore. If you examine the serial TSP program, you'll see that only two functions called from `Search` access the current best tour: `Feasible` and `Check_best_tour`. `Feasible` is called for every possible new tree node, but `Check_best_tour` is only called for the leaves. So a strategy for reducing the contention for the best tour is to have each thread store a "local estimate" of the cost of the best tour, use this estimate in `Feasible`, and update the estimate (along with the best tour) only in `Check_best_tour`.

Second Parallel Solution

If the initial distribution of subtrees doesn't do a good job of distributing the work among the threads, the parallelization in the first solution provides no means of redistributing work. The threads with "small" subtrees will finish early, while the threads with large subtrees will continue to work. To address this issue, for the second part of programming assignment 5, you should modify your first version so that work can be redistributed among the threads.

To do this, we can replace the test `!Empty(stack)` controlling execution of the main `while` loop with more complex code. The basic idea is that when a thread runs out of work — i.e., its stack is empty — instead of immediately exiting the `while` loop, the thread waits to see if another thread can provide more work. On the other hand, if a thread that still has work in its stack finds that there is at least one thread without work, and its stack has at least two tours, it can split its stack in half and provide work for one of the threads that has none.

Pthreads condition variables provide a natural way to implement this: when a thread runs out of work it can call `pthread_cond_wait` and go to sleep. When a thread with work finds that there is at least one thread waiting for work, after splitting its stack, it can call `pthread_cond_signal`. When a thread is awakened it can take one of the halves of the split stack and return to work.

This idea can be extended to handle termination. If we maintain a count of the number of threads that are in `pthread_cond_wait`, then when a thread whose stack is empty finds that `thread_count - 1` threads are already waiting, it can call `pthread_cond_broadcast` and as the threads awaken, they'll see that all the threads have run out of work and quit.

Termination

Thus, we can use the pseudo-code shown in Figure 3 for a `Terminated` function that would be used instead of `Empty` for the `while` loop implementing tree search.

There are several details that we should look at more closely. Notice that the code executed by a thread before it splits its stack is fairly complicated: In Lines 1–2

- It checks that it has at least two tours in its stack,
- It checks that there are threads waiting, and
- It checks whether the `new_stack` variable is `NULL`

The reason for the check that the thread has enough work should be clear: if there are fewer than two records on the thread's stack, splitting the stack will either do nothing or result in the active thread's trading places with one of the waiting threads.

It should also be clear that there's no point in splitting the stack if there aren't any threads waiting for work. Finally, if some thread has already split its stack, but a waiting thread hasn't retrieved the new stack, i.e., `new_stack != NULL`, then it could be disastrous to split a stack and overwrite the existing new stack. This makes it essential that after a thread retrieves `new_stack` by (say) copying `new_stack` into its private `my_stack` variable, the thread must set `new_stack` to `NULL`.

If all three of these conditions hold, then we can try splitting our stack. So we can acquire the mutex that protects access to the objects controlling termination (`threads_in_cond_wait`, `new_stack`, and the condition variable). However, the condition

```
threads_in_cond_wait > 0 && new_stack == NULL
```

can change between the time we start waiting for the mutex and the time we actually acquire it. So we need to confirm that this condition is still true after acquiring the mutex (Line 4). Once we've verified that these conditions still hold, we can split the stack, awaken one of the waiting threads, unlock the mutex, and return to work.

If the test in Lines 1–2 is false, we can check to see if we have any work at all — i.e., our stack is nonempty. If it is, we return to work. If it isn't, we'll start the termination sequence by waiting for and acquiring the termination mutex in Line 13. Once we've acquired the mutex, there are two possibilities:

- We're the last thread to enter the termination sequence, i.e.,

```
threads_in_cond_wait == thread_count-1
```

or

- Other threads are still working.

```

1  if (my_stack_size >= 2 && threads_in_cond_wait > 0 &&
2      new_stack == NULL) {
3      lock term_mutex;
4      if (threads_in_cond_wait > 0 && new_stack == NULL) {
5          Split my_stack creating new_stack;
6          pthread_cond_signal(&term_cond_var);
7      }
8      unlock term_mutex;
9      return 0; /* Terminated = False; don't quit */
10 } else if (!Empty(my_stack)) { /* Stack not empty, keep working */
11     return 0; /* Terminated = false; don't quit */
12 } else { /* My stack is empty */
13     lock term_mutex;
14     if (threads_in_cond_wait == thread_count-1) { /* Last thread */
15                                                     /* running */
16         threads_in_cond_wait++;
17         pthread_cond_broadcast(&term_cond_var);
18         unlock term_mutex;
19         return 1; /* Terminated = true; quit */
20     } else { /* Other threads still working, wait for work */
21         threads_in_cond_wait++;
22         while (pthread_cond_wait(&term_cond_var, &term_mutex) != 0);
23         /* We've been awakened */
24         if (threads_in_cond_wait < thread_count) { /* We got work */
25             my_stack = new_stack;
26             new_stack = NULL;
27             threads_in_cond_wait--;
28             unlock term_mutex;
29             return 0; /* Terminated = false */
30         } else { /* All threads done */
31             unlock term_mutex;
32             return 1; /* Terminated = true; quit */
33         }
34     } /* else wait for work */
35 } /* else my_stack is empty */

```

Figure 3: Pseudo-code for Terminated function

In the first case, we know that since all the other threads have run out of work, and we have also run out of work, the tree search should terminate. So we notify all the other threads by calling `pthread_cond_broadcast` and returning `True`. Before executing the broadcast, we increment `threads_in_cond_wait`, even though the broadcast is telling all the threads to return from the condition wait. The reason is that `threads_in_cond_wait` is serving a dual purpose: when it's less than `thread_count` it tells us how many threads are waiting. However, when it's equal to `thread_count`, it tells us that all the threads are out of work, and it's time to quit.

In the second case — other threads are still working — we call `pthread_cond_wait` (Line 22) and wait to be awakened. Recall that it's possible that a thread could be awakened by some event other than a call to `pthread_cond_signal` or `pthread_cond_broadcast`. So, as usual, we put the call to `pthread_cond_wait` in a while loop which will immediately call `pthread_cond_wait` again if some other event (return value not 0) awakens the thread.

Once we've been awakened, there are also two cases to consider:

- `threads_in_cond_wait < thread_count`, and
- `threads_in_cond_wait == thread_count`.

In the first case, we know that some other thread has split its stack and created more work. So we copy the newly created stack into our private stack, set the `new_stack` variable to `NULL`, and decrement `threads_in_cond_wait` (Lines 25–27). Recall that when a thread returns from a condition wait, it obtains the mutex associated with the condition variable. So before returning, we also unlock the mutex (Line 28).

In the second case, there's no work left. So we unlock the mutex and return `True`.

Before discussing the `Split_stack` function, note that it's possible that a thread with work can spend a lot of time waiting for `term_mutex` before being able to split its stack. Other threads may be either trying to split their stacks, or preparing for the condition wait. If we suspect that this is a problem, Pthreads provides a nonblocking alternative to `pthread_mutex_lock` called `pthread_mutex_trylock`:

```
int pthread_mutex_trylock(
    pthread_mutex_t*  mutex_p    /* in/out */);
```

This function attempts to acquire `mutex`. However, if it's locked, instead of waiting, it returns immediately. The return value will be zero if the calling thread has successfully acquired the mutex, and nonzero if it hasn't. So as an alternative to waiting on the mutex before splitting its stack, a thread can call `pthread_mutex_trylock`. If it acquires `term_mutex`, it can proceed as before. If not, it can just return. Presumably on a subsequent call it can successfully acquire the mutex.

Splitting the Stack

Since our goal is to balance the load among the threads, we would like to insure that the amount of work in the new stack is roughly the same as the amount remaining in the original stack. We have

no way of knowing in advance of searching the subtree rooted at a partial tour how much work is actually associated with the partial tour. So we'll never be able to guarantee an equal division of work. However, we can use the same strategy that we used in our original assignment of subtrees to threads: the subtrees rooted at two partial tours with the same number of cities have identical structures. Thus, since on average two partial tours with the same numbers of cities are equally likely to lead to a "good" tour (and hence more work), we can try splitting the stack by assigning the tours on the stack on the basis of their numbers of edges. So the tour with the least number of edges remains on the original stack, the tour with next to the least number of edges goes to the new stack, the tour with the next number of edges remains on the original, etc.

This is fairly simple to implement, since the tours on the stack have an increasing number of edges. That is, as we proceed from the bottom of the stack to the top of the stack, the number of edges in the tours increases. This is because when we push a new partial tour with k edges onto the stack, the tour that's immediately "beneath" it on the stack either has k edges or $k - 1$ edges. So we can implement the split by starting at the bottom of the stack, and alternately leaving partial tours on the old stack and pushing partial tours onto the new stack. So tour 0 will stay on the old stack, tour 1 will go to the new stack, tour 2 will stay on the old stack, etc. If the stack is implemented as an array of tours, this scheme will require that the old stack be "compressed" so that the gaps left by removing alternate tours are eliminated. If the stack is implemented as a linked list of tours, compression won't be necessary.

Details

Your program should accept a command line having the following form:

```
tsp <number of threads> <matrix file>
```

You can assume that the number of threads will be less than the order of the matrix. The main thread should read in the matrix, and store it in a global variable. It will also start the requested threads. Each thread initializes its own stack as described above, and each thread then searches its own stack, keeping track of the *local* best cost. When a thread attempts to update the lowest cost tour, it should also update its local best cost. In the first parallel solution, threads complete after they've searched all the tours starting from their original 2-city partial tours. In the second parallel solution, a thread that runs out of work will wait for additional work or notification that there is no more work. If no thread has enough work, it should terminate. After the threads complete, the main thread prints the lowest cost tour, and the cost of the lowest cost tour.

If there are n cities, the matrix file will contain $n + 1$ lines of text. The first line will contain the number n . Each remaining line will list the costs of travelling from a city to every other city. Costs will be positive ints, except that the cost of travelling from a city to itself will be 0.

You should put your source code for both the first and second parallel solutions in your SVN repository, and you should turn in hardcopy of both solutions.

Submission

As usual, your final electronic copy should be committed to your CS SVN repository by 11 am on Monday, December 5, and you should give me your hardcopy by 4 pm.

Grading

1. Correctness will be 60% of your grade. The first solution will be 40% and the second 20%. For both programs, does your program correctly initialize the stack, conduct the search, and print the correct solution? For the second program, does your program also split stacks correctly?
2. Documentation will be 15% of your grade. Does your header documentation include the author's name, the purpose of the program, and a description of how to use the program? Are the identifiers meaningful? Are any obscure constructs clearly explained? Does the function header documentation explain the purpose of the function, its arguments, its return value, and its use of global variables?
3. Source format will be 10% of your grade. Is the indentation consistent? Have blank lines been used so that the program is easy to read? Is the use of capitalization consistent? Are there any lines of source that are longer than 80 characters (i.e., wrap around the screen)?
4. Quality of solution will be 15% of your grade. Are any of your functions more than 40 lines (not including blank lines, curly braces, or comments)? Are there multipurpose functions? Is your solution too clever – i.e., has the solution been condensed to the point where it's incomprehensible?

Academic Honesty

Remember that you can discuss the program with your classmates, but you cannot look at anyone else's source code. (This includes source code that you might find on the internet.) You also cannot show your source code to anyone else.