# An Introduction to Parallel Programming Solutions, Chapter 6

Lemuel Mullett and Peter Pacheco

January 17, 2012

1. Recollect that our algorithm computes

```
for each timestep t {
   for each particle i
      compute f(i), the force on i
   for each particle i
      update position and velocity of i using F = ma
   if (output step) Output new positions and velocities
}
```

   The first inner loop uses the current values in positions to find the forces at the current time. The second inner loop updates the positions and velocities using the updated forces. If we put all three computations in a single inner loop, we'd be computing forces using some original positions and some updated positions. So it is necessary to separate the two loops.

2. After compiling the program with optimization and `NO_OUTPUT`, we ran it with 1000 time steps, a stepsize of 0.01, and program-generated initial conditions. With particle counts ranging from 500 to 8000 we got the following run-times:

| Particles | 500 | 1000 | 2000 | 4000 | 8000 |
|---|---|---|---|---|---|
| Run-time (secs) | 6.74 | 27.9 | 108 | 431 | 1720 |

   We see that doubling the number of particles increases the run-time by roughly a factor of 4. If we continue to double the number of particles and multiply the run-times by four, we see that somewhere between 32,000 and 64,000 particles will result in a run-time of 24 hours (or 86,400 seconds).

   If we infer that the run-times are quadratic in the number of particles and use least-squares to estimate the coefficients, we get that the run-time is approximately

```
2.68e-5*n*n + 4.36e-4*n + 1.18e-1
```

Setting this equal to 86,400 and solving gives approximately 56,770. That is, running the program with 56,770 particles will result in a run-time of approximately 24 hours. Of course, this will depend on the particular system on which the program is run.

3. We used OpenMP since it's relatively easy to modify the serial version. Both inner `for` loops were parallelized with the default schedule. The serial and the parallel programs were compiled with full optimization and `NO_OUTPUT` defined. When we ran the programs with 500 particles for 100 timesteps with a stepsize of 0.01 using the program-generated initial conditions we got the following run-times (in seconds):

| Serial | 2 threads | 4 threads |
|---|---|---|
| 0.343 | 2.32 | 2.06 |

Interestingly, with cyclic schedules for both loops, the 2-thread run-time was much better (1.50 seconds), but the 4-thread run-time was much worse (2.64 seconds).

See the source code `ex6.3_omp_nbody_red`.

Every time a thread updates the `forces` array, it must obtain access to the critical section, while the updates to the particles' positions and velocities are embarrassingly parallel. So it appears that the contention for access to the critical section is so bad that it is causing the overall run-time to get worse by more than a factor of 4.

4. This program was compiled and run with the same settings and input as the program in the preceding problem.

| Serial | 2 threads | 4 threads |
|---|---|---|
| 0.343 | 0.702 | 0.616 |

In this case using cyclic schedules performed worse, regardless of the number of number of threads: 0.935 seconds with 2 threads and 1.08 seconds with 4 threads.

It appears that the added cost of a thread's having to acquire and release a lock each time a particle's force is updated is much more than the reduction in cost obtained by having a thread work with a subset of the particles.

5. The change is OK because in the first phase of the calculation of the forces (i.e., when `Compute_force` is called), a thread will only compute forces for particles assigned to threads whose rank is greater than or equal to theirs. For example, if there are $n = 6$ particles and `thread_count` $= 3$ threads, then the following table shows entries in the `loc_forces` array that are zero and nonzero (X):

| Thread | Particle | | | | | |
|---|---|---|---|---|---|---|
| | 0 | 1 | 2 | 3 | 4 | 5 |
| 0 | X | X | X | X | X | X |
| 1 | 0 | 0 | X | X | X | X |
| 2 | 0 | 0 | 0 | 0 | X | X |

The second phase of the forces calculation sums the "columns" of the `loc_forces` array. Thread 0 sums the entries in the first two columns, thread 1 sums the entries in the next two, and thread 2 sums those in the last two. So we see that each thread can exit the sum loop after adding the entries in its columns with "row number" less than or equal to its rank.

Note that if, during the first phase, the particles have a cyclic partition, then this reduction in the second phase isn't possible. In our example, if we use a cyclic partition, then the following table shows the zero and nonzero entries in `loc_forces` after the first phase.

| Thread | Particle | | | | | |
|---|---|---|---|---|---|---|
| | 0 | 1 | 2 | 3 | 4 | 5 |
| 0 | X | X | X | X | X | X |
| 1 | 0 | X | X | X | X | X |
| 2 | 0 | 0 | X | X | X | X |

When we compile the program with full optimization and `NO_OUTPUT`, and then run it with 1000 particles for 1000 timesteps, with a stepsize of 0.01 and program generated initial conditions, we get the following run-times (in seconds):

| Threads | Cyclic | Default | To `my_rank` |
|---|---|---|---|
| Serial | 24.2 | 24.2 | 24.2 |
| 2 | 12.2 | 18.2 | 18.2 |
| 4 | 6.2 | 10.6 | 10.6 |
| 8 | 3.1 | 5.8 | 5.7 |

The column labelled "Cyclic" contains run-times for the version in which the first loop in the computation of the forces has a cyclic schedule. The remaining loops have the default schedule. The column labelled "Default" contains run-times for the version in which all of the loops have the default schedule, and the inner loop runs to `thread_count`. The third column contains run-times for a version which is the same as the second except that the inner loop runs to `my_rank`.

For these data on this system, there is virtually no difference between the times in the second and third columns, and it would appear that the second pair of nested loops makes very little contribution to the overall run-time.

6. Modifying the two "for each particle" loops with `nowait` clauses *could* cause problems. Without the implied barrier at the end of the second loop

```
#     pragma omp for
      for (part = 0; part < n; part++)
         Update_part(part, forces, curr, n, delta_t);
```

   a thread could call `Output_state` and print positions and velocities for particles that hadn't yet had their positions and velocities updated. For example, if there are two threads and four particles, thread 1 could call `Output_state` before thread 0 had updated the position and velocity of particle 1.

   However, if there is an implied barrier after the second loop and `nowaits` modifying the first loop and the `single` directive, then no thread can use or print invalid data.

7. We compiled the program with full optimization, and ran it with 1024 particles for 1000 timesteps. We used a stepsize of 0.01 and program generated initial conditions. We tried `schedule(static,n)` for $n = 1, 2, 4, 8, \ldots 256$. Each time the program was run with four threads. Here are the run-times (in seconds):

| Chunksize | 1 | 2 | 4 | 8 | 16 | 32 | 64 | 128 | 256 |
|---|---|---|---|---|---|---|---|---|---|
| Run-time | 3.70 | 3.70 | 3.72 | 3.76 | 3.84 | 4.02 | 4.36 | 5.04 | 6.40 |

   So it appears that for this problem on this system, when the program is run with 4 threads, the chunksize of 1 is optimal.

8. See the source file `ex6.8_omp_daxpy.c`. The following table shows run-times (in milliseconds) on one of our systems after the program was compiled with full optimization.

| | Vector Order | | | | | |
|---|---|---|---|---|---|---|
| | $10^6$ | | $10^7$ | | $10^8$ | |
| Threads | Cyclic | Block | Cyclic | Block | Cyclic | Block |
| 2 | 34.0 | 5.9 | 340 | 57 | 3300 | 540 |
| 4 | 26.0 | 5.7 | 220 | 56 | 2300 | 320 |

   Clearly the block partitioning is far superior to the cyclic. This is probably due to cache. With block partitioning the threads can work independently of each other: if

thread A accesses cache line L, it's very unlikely that any other thread will also access cache line L. On the other hand, with the cyclic partition each thread will need to access every line of both `x` and `y`. This will result in a tremendous amount of false-sharing for `y`.

9. On the system we used for timings the block distribution consistently outperformed the cyclic distribution:

| | Local Array Size | | | |
|---|---|---|---|---|
| | 100 | | 1000 | |
| Processes | Block | Cyclic | Block | Cyclic |
| 2 | 1.6e-5 | 2.7e-5 | 3.4e-5 | 1.2e-4 |
| 4 | 3.1e-5 | 5.8e-5 | 1.1e-4 | 4.0e-4 |
| 8 | 8.0e-5 | 1.9e-4 | 3.7e-4 | 1.5e-3 |
| 16 | 2.2e-4 | 5.8e-4 | 1.6e-3 | 5.7e-3 |

See the file `ex6.9_mpi_allgather.c`. The program was compiled with full optimization. Times are in seconds and are averages over 100 iterations.

The results aren't surprising: we expect the cyclic distribution to involve some additional work. For example, a process might receive the data from another process as a contiguous block, and the MPI implementation would then have to copy the contiguous block into noncontiguous locations in memory.

10. We'll also assume that

    - No lines from `x` or `y` are in cache when the code begins execution.
    - The only misses that do occur are the result of accesses to `x` or `y`. In other words, except for `x` and `y`, all needed data and instructions are in cache when the code begins executing.

    Thread 0 will execute the body of the first `for` loop when $i = 0, 1, \ldots 31$. So this will result in four write misses for `x` and four write misses for `y`. Thread 1 will execute the body of the first `for` loop when $i = 32, 33, \ldots 63$. So it will also have four misses for `x` and four misses for `y`.

    The execution of the second loop depends on `chunksize`:

    (a) If `chunksize = n/thread_count = 32,` then the thread 0 will execute the body of the loop for $i = 0, 1, \ldots 31$. So the elements of both `x` and `y` that it needs should already be cached. Similar reasoning applies to thread 1. So when `chunksize` is the same as it was for the first loop, the second loop should result in no further misses.

(b) If `chunksize = 8`, then the threads will execute the body of the second `for` loop as follows:

| Thread | Iterations (values of $i$) | | | |
|---|---|---|---|---|
| 0 | $0, \ldots, 7$ | $16, \ldots, 23$ | $32, \ldots, 39$ | $48, \ldots, 55$ |
| 1 | $8, \ldots, 15$ | $24, \ldots, 31$ | $40, \ldots 47$ | $56, \ldots, 63$ |

The elements of `x` and `y` that are accessed in the first two chunks are already cached on thread 0. However, the elements that are accessed in the last two chunks are not. So there will be two additional misses for `x` and two additional misses for `y`. Similar reasoning applies to thread 1. already cached

11. On the system we used for timings `MPI_Allgather` with `MPI_IN_PLACE` consistently outperformed `MPI_Allgather` using separate send and receive buffers:

| | Local Array Size | | | |
|---|---|---|---|---|
| | 100 | | 1000 | |
| Processes | Two bufs | In place | Two bufs | In place |
| 1 | 2.0e-5 | 6.0e-6 | 4.5e-5 | 6.0e-6 |
| 2 | 6.3e-4 | 6.1e-4 | 1.4e-3 | 1.3e-3 |
| 4 | 2.5e-3 | 1.5e-3 | 4.4e-3 | 3.4e-3 |
| 8 | 3.7e-3 | 2.3e-3 | 9.5e-3 | 8.3e-3 |
| 16 | 6.9e-3 | 5.1e-3 | 2.0e-2 | 1.8e-2 |

See the file `ex6.11_mpi_in_place.c`. The program was compiled with full optimization. Times are in seconds and are total times to execute 100 calls to `MPI_Allgather`.

The results with one process aren't surprising: when there's only one process and `MPI_IN_PLACE` is used, the call to `MPI_Allgather` can simply return: the data to be sent is already in the receive buffer. On the other hand, the implementation that uses separate buffers will need to copy from the send buffer to the receive buffer.

The other results can be explained — at least in part — using similar reasoning: with `MPI_IN_PLACE` each process is spared the trouble of copying the contents of its send buffer into its receive buffer.

12. (a) We compiled both versions with `NO_OUTPUT` and full optimization. Both versions were run with 500 particles for 500 timesteps with a stepsize of 0.01 and program-generated initial conditions. The following table shows run-times in seconds.

| Processes | Original | Separate `loc_pos` |
|---|---|---|
| 1 | 3.37 | 3.37 |
| 2 | 1.70 | 1.70 |
| 4 | 0.86 | 0.86 |
| 8 | 0.44 | 0.44 |
| 16 | 0.23 | 0.23 |

There is virtually no difference between the run-times. Since the memory requirements of the original version are somewhat smaller, it is probably preferred.

(b) With a separate array for the masses of the local particles, it's necessary to execute an allgather before computing the forces, since the total force on each particle depends on the masses of all the other particles. Hence in a system with constant masses (e.g., a Newtonian system) the use of separate local storage for the masses of the local particles serves no purpose.

Once again we compiled both versions with `NO_OUTPUT` and full optimization. Both versions were run with 500 particles for 500 timesteps with a stepsize of 0.01 and program-generated initial conditions. The following table shows run-times in seconds.

| Processes | Original | With `loc_masses` |
|---|---|---|
| 1 | 3.37 | 3.37 |
| 2 | 1.70 | 1.70 |
| 4 | 0.86 | 0.87 |
| 8 | 0.44 | 0.45 |
| 16 | 0.23 | 0.26 |

For small numbers of processes the run-times are virtually identical. However, for larger numbers of processes the cost of the extra communication begins to degrade the overall run-time.

13. See Figure 1.

14. See `ex6.14_mpi_nbody_red.c`

15. The following table shows the correspondence between global indexes and local indexes if $p =$ `comm_sz` $= 4$, $n = 12$, and the program is using a block distribution.

| Process | Global Indexes | Local Indexes |
|---|---|---|
| 0 | $0, 1, 2$ | $0, 1, 2$ |
| 1 | $3, 4, 5$ | $0, 1, 2$ |
| 2 | $6, 7, 8$ | $0, 1, 2$ |
| 3 | $9, 10, 11$ | $0, 1, 2$ |

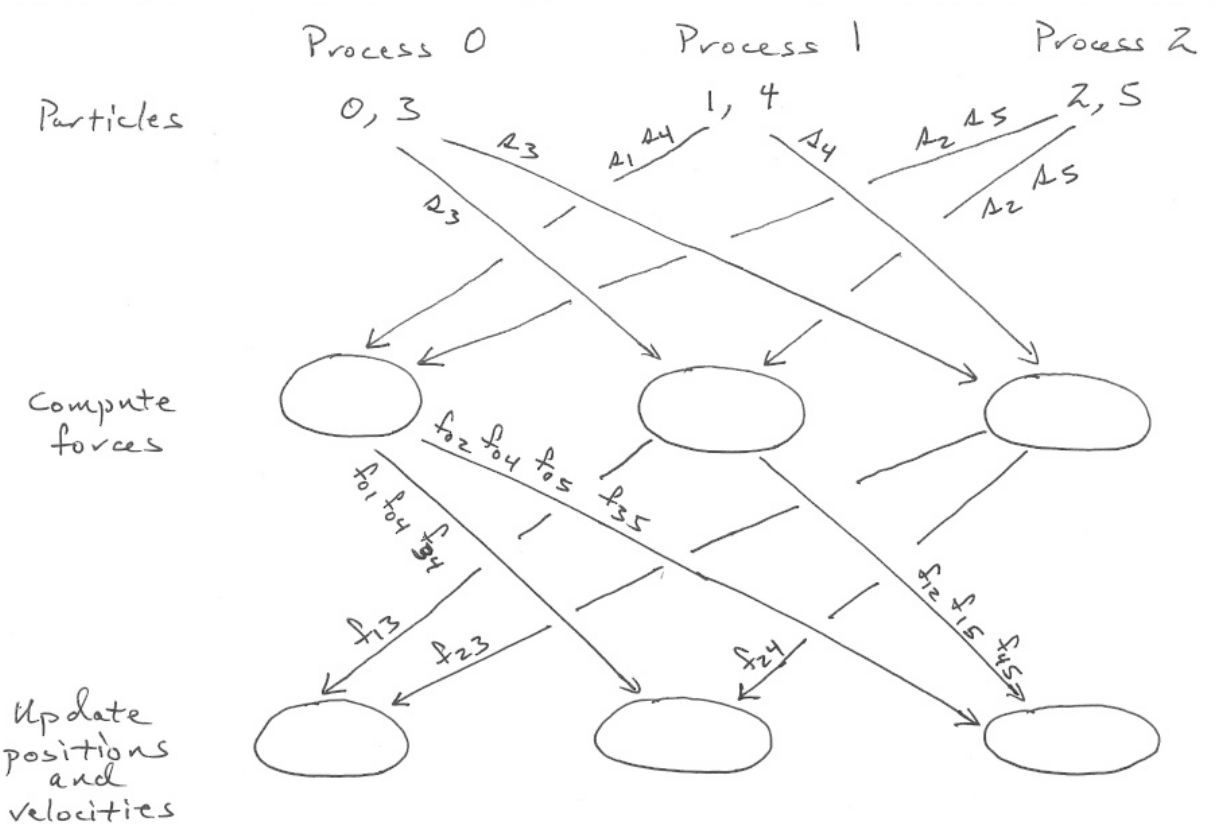


Figure 1: Communications in an "obvious" MPI implementation of the reduced $n$-body solver

(a) So we see that if `local_n = n/p`, then

```
global_i = local_i + my_rank*local_n
```

(b) We also see that

```
local_i = global_i % local_n
```

The following table shows the correspondence when the program is using a cyclic distribution.

| Process | Global Indexes | Local Indexes |
|---|---|---|
| 0 | $0, 4, 8$ | $0, 1, 2$ |
| 1 | $1, 5, 9$ | $0, 1, 2$ |
| 2 | $2, 6, 10$ | $0, 1, 2$ |
| 3 | $3, 7, 11$ | $0, 1, 2$ |

(c) So if a program is using a cyclic distribution, then

```
global_i = my_rank + local_i*p
```

(d) We also have that

```
local_i = global_i/p
```

16. The following table shows the global indexes if $p =$ `comm_sz` $= 4$, $n = 12$, and the program is using a cyclic distribution.

| Process | Global Indexes | | |
|---|---|---|---|
| 0 | 0 | 4 | 8 |
| 1 | 1 | 5 | 9 |
| 2 | 2 | 6 | 10 |
| 3 | 3 | 7 | 11 |

So, for example,

```
First_index(5, 1, 3, 4) = 7
First_index(1, 1, 0, 4) = 4
First_index(7, 3, 1, 4) = 9
```

If the rank of the first process `rk1` is less than the rank of the second `rk2`, then the next index can be computed by simply moving down the appropriate column in the table. So if the global index of the first particle is `gbl1`, `First_index` should return

```
gbl1 + (rk2 - rk1)
```

in this case.

If the rank of the first process is greater than the rank of the second, then the next index is in the column following the column containing `gbl1`. Since there are $p$ entries in each column, we can get to the next column by adding $p$ to `gbl1`. For example, when `gbl1` is 7, $7 + 4 = 11$ is in the next column, but since `rk1` $\geq$ `rk2`, we need to go *up* the new column `rk1 - rk2` slots. So in this case the function should return

```
gbl1 + (rk2 - rk1) + p
```

So the function could be coded as follows.

```
int First_index(int gbl1, int rk1, int rk2, int p) {
   if (rk1 < rk2)
      return gbl1 + (rk2 - rk1);
   else
      return gbl1 + (rk2 - rk1) + p;
```

17. (a) Note that Figure 6.10 shows an edge joining the partial tour $0 \to 1 \to 2$ to the complete tour $0 \to 1 \to 2 \to 3 \to 0$. However, if we examine the code, we'll see that there is an intermediate partial tour, $0 \to 1 \to 2 \to 3$, and it's only after this partial tour is popped off the stack that we stop pushing partial tours, and call the `Best_tour` function. So the stack will contain the maximum number of records when the following partial tours have been pushed:

$$0 \to 1 \to 2 \to 3,\ 0 \to 1 \to 3,\ 0 \to 2,\ 0 \to 3$$

So in this case we see that there is a maximum of 4 records on the stack.

(b) In an attempt to make the diagram more readable, we've omitted the last level of the tree and the arrows joining successive vertices in the partial tours. We've also rotated the tree counter-clockwise by 90 degrees. See Figure 2.

(c) In the five city problem, if we add in the partial tour $0 \to 1 \to 2 \to 3 \to 4$, we see that the stack will contain the maximum number of records when the following partial tours have been pushed:

$$0 \to 1 \to 2 \to 3 \to 4,\ 0 \to 1 \to 2 \to 4,\ 0 \to 1 \to 3,\ 0 \to 1 \to 4,\ 0 \to 2,\ 0 \to 3,\ 0 \to 4$$

So in this case there is a maximum of 7 records on the stack.

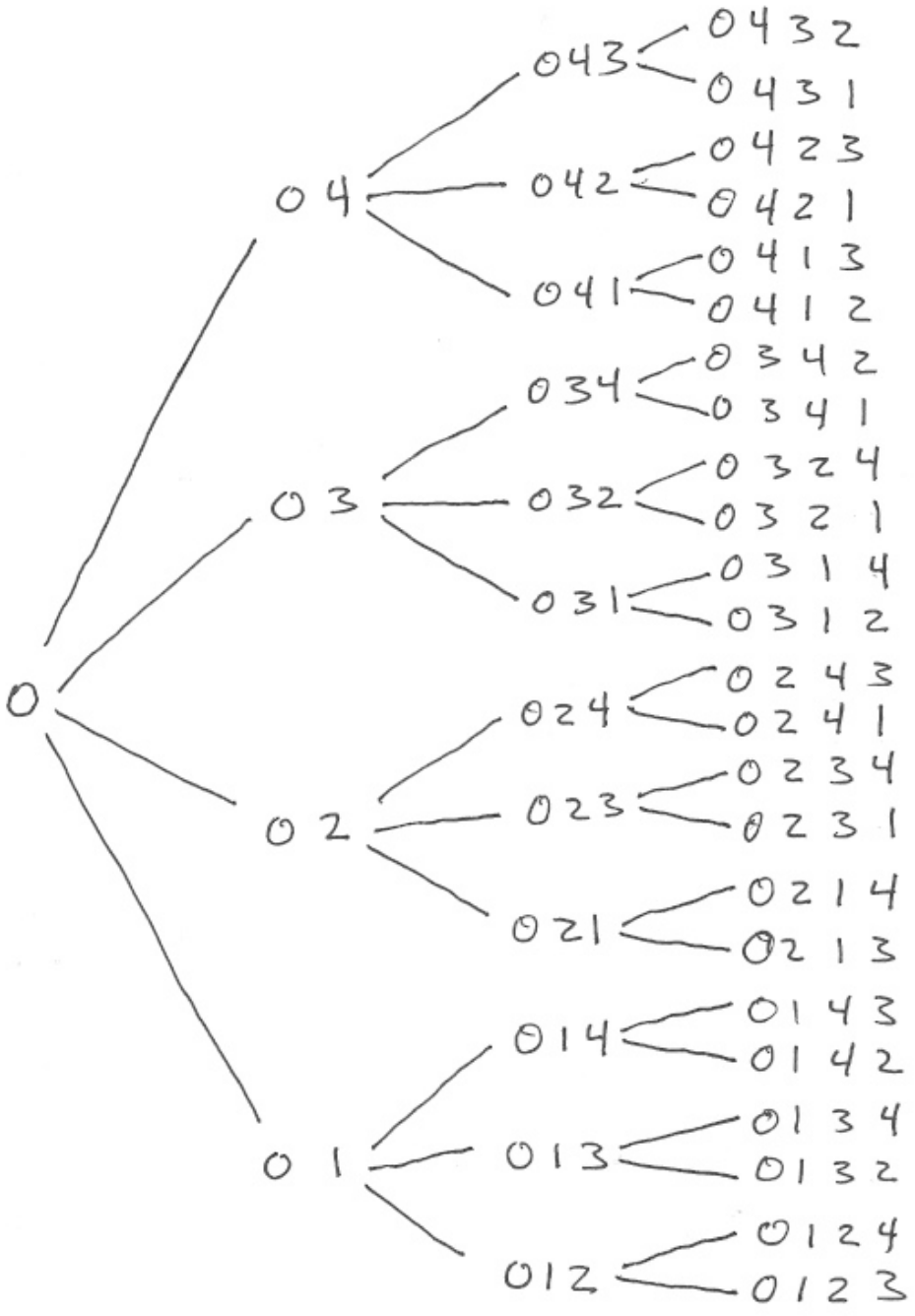


Figure 2: A partial search tree for a five-city TSP

(d) In general, then, we see that there will be a maximum of

$$1 + 1 + 2 + \cdots + n - 2 = 1 + \sum_{k=1}^{n-2} k = 1 + \frac{(n-1)(n-2)}{2}$$

records on the stack.

18. The queue will need to simultaneously store all of the partial tours at a given level in the tree, and the number of tours at a given level will grow very rapidly. If level one contains the 1-city tour, level two contains the 2-city tours, etc., we see that for an $n$-city problem, level $k$, $1 < k < n$, will contain

$$(n - 2 + 1)(n - 3 + 1) \cdots (n - k + 1)$$

$k$-city partial tours, and, this number can be huge. For example, if $n = 15$, and $k = 14$, the product will be $14! = 87,178,291,200$ or about 90 billion.

(a) We can write a function `queue_sz(queue)` which returns the number of partial tours in the queue, and instead of continuing to iterate until the queue is empty, we can iterate until the queue has at least `thread_count` tours:

```
. . .
while (queue_sz(queue) < thread_count) {
   tour = Dequeue(queue);
   for each neighboring city
      if (Feasible(tour, city)) {
         Add_city(tour, city);
         Enqueue(tour);
         Remove_last_city(tour);
      }
   Free_tour(tour);
}
```

(b) As we move from the head to the tail of the queue, the tours will have increasing (really, nondecreasing) length. So it probably makes sense to use a cyclic decomposition of the queue:

```
for (q_elt = my_rank; q_elt < queue_sz(queue); q_elt += thread_count) {
   tour = Link_to(queue, q_elt);
   Push(stack, tour);
}
```

```
Barrier();
if (my_rank == 0) Free_queue(queue);
```

The details will depend on the implementation of the queue and the stack. For example, if `Enqueue` enqueues a copy of its argument, then `Link_to` can simply return a pointer to a tour, and `Push` can push this tour onto the stack — without copying. The `Barrier()` is needed to make sure that the queue's supporting structures aren't freed before every thread has finished acquiring its initial tours.

19. See `ex6.19_pth_tsp_stat.c`. We ran the program with two 15-city problems. The following table show elapsed times in seconds for the original Pthreads version and the version that uses read-write locks:

| Threads | First Problem | | Second Problem | |
|---|---|---|---|---|
| | Orig | RWL | Orig | RWL |
| 1 | 36 | 55 | 212 | 312 |
| 2 | 30 | 390 | 99 | 689 |
| 4 | 28 | 655 | 37 | 1002 |
| 8 | 25 | 901 | 32 | 114 |

In a word, the performance of the version that uses read-write locks is disastrous. From the one-thread times, we can see that the version that uses read-write locks adds about 50% to the run-times just because of calls to the lock and unlock functions. Except for the 8-thread run of the second problem, the performance of the version that uses read-write locks only gets worse as more threads are added. Presumably, this is due to contention for the lock, in spite of the fact that no run acquired the write lock more than 17 times total.

20. (a) When a new partial tour is pushed onto the stack, it either has the same length as the partial tour that was previously on the top of the stack or it has one more city. So the $k/2$ partial tours on the top of the stack will probably be longer than the $k/2$ tours on the bottom of the stack. Since a shorter tour is the root of a larger subtree than a longer tour, we expect that the $k/2$ tours on the bottom of the stack will require more work than the $k/2$ on the top of the stack, and this strategy will probably do a poor job of load-balancing.

    (b) This strategy seems better than the strategy in the preceding part. However, this approach doesn't directly address the problem discussed in the preceding part: short partial tours are the roots of larger subtrees. For example, if a stack contains an $m$-city partial tour with cost $c + 1$ and a $2m$-city partial tour with cost $c$, we wouldn't be surprised if the $m$-city tour led to considerably more work.

(c) This strategy seems to address the problem in the preceding part. When two partial tours have roughly the same total cost, the shorter tour will have a higher average cost per edge, and we might expect it to require less work. On the other hand, when two partial tours have roughly the same number of edges, then the partial tour with lower overall cost and hence lower average cost per edge is more likely to require more work.

See `ex6.20_pth_tsp_dyn.c`. Table 1 shows the results of using the different splitting strategies in the Pthreads implementation when it's run with two different fifteen city problems. The strategies are:

(0) Alternate tours are assigned to the donating thread and the receiving thread — the strategy outlined in the text.

(1) Strategy (a) above.

(2) Strategy (b) above.

(3) Strategy (c) above.

The run-times are in seconds. The numbers in the "Splits" column are the total number of times the stack was split by all threads. It should be noted that because of nondeterminism in the `Terminated` function, there is considerable variability in the number of splits, and, for the 8 and 16 threads, there can be considerable variability in the overall run-time.

Not surprisingly strategy (a) invariably requires considerably more splits than the other strategies. Presumably the thread that gets the top half will, in most cases, soon run out of work again. The remaining strategies result in virtually identical run-times when there are fewer than 8 threads. When there are 8 or 16 threads, the average cost strategy tends to require fewer splits, but the original, alternating tour, strategy generally gives the best run-times.

21. (a) See `ex6.21_mpi_tsp_stat.c`. This implementation will be much slower than the original static implementation if one process quickly finds the best tour and its other tours can be quickly eliminated, while another process has to search a large subtree of relatively expensive tours. For example, suppose there are two processes. Also suppose that process 0 finds the best tour by always branching left and the other tours assigned to process 0 have an expensive edge going from 0 to the first nonzero city. Also suppose process 1 has a subtree in which all the tours are more expensive than the best tour, but all the edges have roughly the same cost. Then process 0 will finish almost immediately, but in the modified program process 1 will have to search a large subtree. Furthermore, if the same

| Threads | Strat | First Problem | | Second Problem | |
|---|---|---|---|---|---|
| | | Run-time | Splits | Run-time | Splits |
| 1 | 0 | 41.0 | 0 | 240.0 | 0 |
| | 1 | 41.0 | 0 | 240.0 | 0 |
| | 2 | 41.0 | 0 | 240.0 | 0 |
| | 3 | 41.0 | 0 | 240.0 | 0 |
| 2 | 0 | 34.0 | 10 | 110.0 | 8 |
| | 1 | 34.0 | 793 | 110.0 | 775 |
| | 2 | 34.0 | 7 | 110.0 | 8 |
| | 3 | 34.0 | 10 | 110.0 | 9 |
| 4 | 0 | 30.0 | 32 | 41.0 | 40 |
| | 1 | 30.0 | 4385 | 41.0 | 2774 |
| | 2 | 30.0 | 36 | 41.0 | 48 |
| | 3 | 30.0 | 44 | 41.0 | 52 |
| 8 | 0 | 27.0 | 171 | 3.3 | 141 |
| | 1 | 36.0 | 10,053 | 3.6 | 7616 |
| | 2 | 35.0 | 231 | 3.3 | 120 |
| | 3 | 28.0 | 141 | 3.3 | 185 |
| 16 | 0 | 5.1 | 484 | 2.0 | 375 |
| | 1 | 6.2 | 22,225 | 2.3 | 13,626 |
| | 2 | 4.2 | 548 | 2.0 | 399 |
| | 3 | 7.6 | 329 | 2.0 | 321 |

Table 1: Run-times (in seconds) and total numbers of stack splits using different stack-splitting strategies

problem is solved by a single process, it will find the solution just as quickly as process 0.

If, on the other hand, the best tour is found by always branching right, and there are many updates to the best tour, then this implementation may be better than the original static implementation, since it won't incur the cost of all the broadcasts.

(b) The performance of the original static implementation in this setting depends on the amount of work done by process 0. If it quickly finds the best tour, and subsequent tours can be eliminated when only a small part of the tour is examined, the original static implementation should be much faster than the modified implementation. On the other hand, if process 0 has to search extensively in its subtree, then the performance of the two implementations should be relatively close.

The following table shows the performance of the two static implementations and the dynamic implementation for two different input problems. Both problems have 17 cities. The cost of the edges $0 \to x$ is 99 for $x = 5, 6, \ldots, 16$. The cost of all other edges is 3, except both problems have a tour with cost 17 in process 0's subtree. For the first problem, the tour is $0 \to 1 \to 2 \to \cdots \to 16 \to 0$. For the second problem the tour is $0 \to 4 \to 1 \to 2 \to 3 \to 5 \to 6 \to 7 \to \cdots \to 16 \to 0$. So in the first problem process 0 will find the best tour very quickly, while for the second problem it will need to search for some time before it finds the best tour:

| Problem | Original Static | Modified Static | Dynamic |
|---|---|---|---|
| First | 4.9 | 46 | $5.7 \times 10^{-3}$ |
| Second | 35 | 53 | 5.6 |

Times are in seconds. The run-times in the last column were taken with minimum stack size for a split of two and maximum tour size for reassignment of seventeen.