Chapter 6

Parallel Program Development
Roadmap

- Solving non-trivial problems.
- The n-body problem.
- The traveling salesman problem.
- Applying Foster’s methodology.
- Starting from scratch on algorithms that have no serial analog.
TWO N-BODY SOLVERS
The n-body problem

- Find the positions and velocities of a collection of interacting particles over a period of time.

- An n-body solver is a program that finds the solution to an n-body problem by simulating the behavior of the particles.
N-body solver

Position_{time 0} → mass → Velocity_{time 0} → N-body solver → Position_{time x} → Velocity_{time x}
Simulating motion of planets

- Determine the positions and velocities:
  - Newton’s second law of motion.
  - Newton’s law of universal gravitation.
(6.1)

\[ f_{qk}(t) = -\frac{Gm_qm_k}{\left| s_q(t) - s_k(t) \right|^3} \left[ s_q(t) - s_k(t) \right] \]

(6.2)

\[ F_q(t) = \sum_{k=0}^{n-1} f_{qk} = -Gm_q \sum_{k=0}^{n-1} \frac{m_k}{\left| s_q(t) - s_k(t) \right|^3} \left[ s_q(t) - s_k(t) \right] \]
(6.3)

\[ s''_q(t) = -G \sum_{j=0}^{n-1} \frac{m_j}{\left| s_q(t) - s_j(t) \right|^3} \left[ s_q(t) - s_j(t) \right] \]

\[ t = 0, \Delta t, 2\Delta t, \ldots, T\Delta t \]
Serial pseudo-code

Get input data;
for each timestep {
    if (timestep output) Print positions and velocities of particles;
    for each particle q
        Compute total force on q;
    for each particle q
        Compute position and velocity of q;
}
Print positions and velocities of particles;
Computation of the forces

```c
for each particle q {
    for each particle k != q {
        x_diff = pos[q][X] - pos[k][X];
        y_diff = pos[q][Y] - pos[k][Y];
        dist = sqrt(x_diff*x_diff + y_diff*y_diff);
        dist_cubed = dist*dist*dist;
        forces[q][X] -= G*masses[q]*masses[k]/dist_cubed * x_diff;
        forces[q][Y] -= G*masses[q]*masses[k]/dist_cubed * y_diff;
    }
}
```
A Reduced Algorithm for Computing N-Body Forces

for each particle q
    forces[q] = 0;
for each particle q {
    for each particle k > q {
        x_diff = pos[q][X] - pos[k][X];
        y_diff = pos[q][Y] - pos[k][Y];
        dist = sqrt(x_diff*x_diff + y_diff*y_diff);
        dist_cubed = dist*dist*dist;
        force_qk[X] = G*masses[q]*masses[k]/dist_cubed * x_diff;
        force_qk[Y] = G*masses[q]*masses[k]/dist_cubed * y_diff
        forces[q][X] += force_qk[X];
        forces[q][Y] += force_qk[Y];
        forces[k][X] -= force_qk[X];
        forces[k][Y] -= force_qk[Y];
    }
}
The individual forces

\[
\begin{bmatrix}
0 & f_{01} & f_{02} & \cdots & f_{0,n-1} \\
-f_{01} & 0 & f_{12} & \cdots & f_{1,n-1} \\
-f_{02} & -f_{12} & 0 & \cdots & f_{2,n-1} \\
\vdots & \vdots & \vdots & \ddots & \vdots \\
-f_{0,n-1} & -f_{1,n-1} & -f_{2,n-1} & \cdots & 0
\end{bmatrix}
\]
Using the Tangent Line to Approximate a Function

\[ y = g(t_0) + g'(t_0) (t - t_0) \]

\[ y = g(t) \]
Euler’s Method

\[ s(t) \]

- \( t_0 \)
- \( t_0 + \Delta t \)
- \( t_0 + 2\Delta t \)
- \( t_0 + 3\Delta t \)
- \( \cdots \)

Copyright © 2010, Elsevier Inc. All rights Reserved
Parallelizing the N-Body Solvers

- Apply Foster’s methodology.
- Initially, we want a lot of tasks.
- Start by making our tasks the computations of the positions, the velocities, and the total forces at each timestep.
Communications Among Tasks in the Basic N-Body Solver
Communications Among Agglomerated Tasks in the Basic N-Body Solver

\[ s_q, v_q, F_q \]

\[ s_r, v_r, F_r \]

\[ t \]

\[ t + \Delta t \]

\[ s_q \]

\[ s_r \]
Communications Among Agglomerated Tasks in the Reduced N-Body Solver

$q < r$

$t$

$q$

$s_r(t)$

$r$

$t + \Delta t$

$s_r(t + \Delta t)$

$q$

$v_q$

$F_q$

$r$

$v_r$

$F_r$

$s_q$

$f_{qr}(t)$

$f_{qr}(t + \Delta t)$
Computing the total force on particle q in the reduced algorithm

```
for each particle k > q {
    x_diff = pos[q][X] - pos[k][X];
    y_diff = pos[q][Y] - pos[k][Y];
    dist = sqrt(x_diff*x_diff + y_diff*y_diff);
    dist_cubed = dist*dist*dist;
    force_qk[X] = G*masses[q]*masses[k]/dist_cubed * x_diff;
    force_qk[Y] = G*masses[q]*masses[k]/dist_cubed * y_diff;

    forces[q][X] += force_qk[X];
    forces[q][Y] += force_qk[Y];
    forces[k][X] -= force_qk[X];
    forces[k][Y] -= force_qk[Y];
}
```
Serial pseudo-code

```plaintext
for each timestep {
    if (timestep output) Print positions and velocities of particles;
    for each particle q
        Compute total force on q;
    for each particle q
        Compute position and velocity of q;
}
```

In principle, parallelizing the two inner for loops will map tasks/particles to cores.
Let's check for race conditions caused by loop-carried dependences.
First loop

```c
#pragma omp parallel for
for each particle q {
    forces[q][X] = forces[q][Y] = 0;
    for each particle k != q {
        x_diff = pos[q][X] - pos[k][X];
        y_diff = pos[q][Y] - pos[k][Y];
        dist = sqrt(x_diff*x_diff + y_diff*y_diff);
        dist_cubed = dist*dist*dist;
        forces[q][X] -= G*masses[q]*masses[k]/dist_cubed * x_diff;
        forces[q][Y] -= G*masses[q]*masses[k]/dist_cubed * y_diff;
    }
}
```
Second loop

```c
#pragma omp parallel for
for each particle q {
    pos[q][X] += delta_t*vel[q][X];
    pos[q][Y] += delta_t*vel[q][Y];
    vel[q][X] += delta_t/masses[q]*forces[q][X];
    vel[q][Y] += delta_t/masses[q]*forces[q][Y];
}
```
Repeated forking and joining of threads

The same team of threads will be used in both loops and for every iteration of the outer loop.

```c
#pragma omp parallel
for each timestep {
    if (timestep output) Print positions and velocities of particles;
#pragma omp for
    for each particle q
        Compute total force on q;
#pragma omp for
    for each particle q
        Compute position and velocity of q;
}
```

But every thread will print all the positions and velocities.
Adding the *single* directive

```c
#pragma omp parallel
for each timestep {
    if (timestep output) {
        #pragma omp single
        Print positions and velocities of particles;
    }

#pragma omp for
for each particle q
    Compute total force on q;
#pragma omp for
for each particle q
    Compute position and velocity of q;
```
Parallelizing the Reduced Solver Using OpenMP

```c
#pragma omp parallel
for each timestep {
    if (timestep output) {
        #pragma omp single
        Print positions and velocities of particles;
    }
    #pragma omp for
    for each particle q
    forces[q] = 0.0;
    #pragma omp for
    for each particle q
    Compute total force on q;
    #pragma omp for
    for each particle q
    Compute position and velocity of q;
```
Problems

\[ F_3 = -f_{03} - f_{13} - f_{23} \]

Updates to forces[3] create a race condition.

In fact, this is the case in general.

Updates to the elements of the forces array introduce race conditions into the code.
First solution attempt

# pragma omp critical
{
    forces[q][X] += force_qk[X];
    forces[q][Y] += force_qk[Y];
    forces[k][X] -= force_qk[X];
    forces[k][Y] -= force_qk[Y];
}

Access to the forces array will be effectively serialized!!!
Second solution attempt

```
omp_set_lock(locks[q]);
forces[q][X] += force_qk[X];
forces[q][Y] += force_qk[Y];
omp_unset_lock(locks[q]);

omp_set_lock(locks[k]);
forces[k][X] -= force_qk[X];
forces[k][Y] -= force_qk[Y];
omp_unset_lock(locks[k]);
```

Use one lock for each particle.
### First Phase Computations for Reduced Algorithm with Block Partition

<table>
<thead>
<tr>
<th>Thread</th>
<th>Particle</th>
<th>0</th>
<th>1</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>$f_{01} + f_{02} + f_{03} + f_{04} + f_{05}$</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>$-f_{01} + f_{12} + f_{13} + f_{14} + f_{15}$</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>$-f_{02} - f_{12}$</td>
<td>$f_{23} + f_{24} + f_{25}$</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>$-f_{03} - f_{13}$</td>
<td>$-f_{23} + f_{34} + f_{35}$</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>$-f_{04} - f_{14}$</td>
<td>$-f_{24} - f_{34}$</td>
<td>$f_{45}$</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>$-f_{05} - f_{15}$</td>
<td>$-f_{25} - f_{35}$</td>
<td>$-f_{45}$</td>
</tr>
</tbody>
</table>
First Phase Computations for Reduced Algorithm with Cyclic Partition

<table>
<thead>
<tr>
<th>Thread</th>
<th>Particle</th>
<th>0</th>
<th>1</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>$f_{01} + f_{02} + f_{03} + f_{04} + f_{05}$</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>$-f_{01}$</td>
<td>$f_{12} + f_{13} + f_{14} + f_{15}$</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>$-f_{02}$</td>
<td>$-f_{12}$</td>
<td>$f_{23} + f_{24} + f_{25}$</td>
</tr>
<tr>
<td>0</td>
<td>3</td>
<td>$-f_{03} + f_{34} + f_{35}$</td>
<td>$-f_{13}$</td>
<td>$-f_{23}$</td>
</tr>
<tr>
<td>i</td>
<td>4</td>
<td>$-f_{04} - f_{34}$</td>
<td>$-f_{14} + f_{45}$</td>
<td>$-f_{24}$</td>
</tr>
<tr>
<td>2</td>
<td>5</td>
<td>$-f_{05} - f_{35}$</td>
<td>$-f_{15} - f_{45}$</td>
<td>$-f_{25}$</td>
</tr>
</tbody>
</table>
Revised algorithm – phase I

```c
#pragma omp for
for each particle q {
    force_qk[X] = force_qk[Y] = 0;
    for each particle k > q {
        x_diff = pos[q][X] - pos[k][X];
        y_diff = pos[q][Y] - pos[k][Y];
        dist = sqrt(x_diff*x_diff + y_diff*y_diff);
        dist_cubed = dist*dist*dist;
        force_qk[X] = G*masses[q]*masses[k]/dist_cubed * x_diff;
        force_qk[Y] = G*masses[q]*masses[k]/dist_cubed * y_diff;
        loc_forces[my_rank][q][X] += force_qk[X];
        loc_forces[my_rank][q][Y] += force_qk[Y];
        loc_forces[my_rank][k][X] -= force_qk[X];
        loc_forces[my_rank][k][Y] -= force_qk[Y];
    }
}
```
Revised algorithm – phase II

```c
#pragma omp for
for (q = 0; q < n; q++) {
    forces[q][X] = forces[q][Y] = 0;
    for (thread = 0; thread < thread_count; thread++) {
        forces[q][X] += loc_forces[thread][q][X];
        forces[q][Y] += loc_forces[thread][q][Y];
    }
}
```
Parallelizing the Solvers Using Pthreads

- By default local variables in Pthreads are private. So all shared variables are global in the Pthreads version.
- The principle data structures in the Pthreads version are identical to those in the OpenMP version: vectors are two-dimensional arrays of doubles, and the mass, position, and velocity of a single particle are stored in a struct.
- The forces are stored in an array of vectors.
Parallelizing the Solvers Using Pthreads

- Startup for Pthreads is basically the same as the startup for OpenMP: the main thread gets the command line arguments, and allocates and initializes the principle data structures.
- The main difference between the Pthreads and the OpenMP implementations is in the details of parallelizing the inner loops.
- Since Pthreads has nothing analogous to a parallel for directive, we must explicitly determine which values of the loop variables correspond to each thread’s calculations.
Parallelizing the Solvers Using Pthreads

- Another difference between the Pthreads and the OpenMP versions has to do with barriers.
- At the end of a parallel for OpenMP has an implied barrier.
- We need to add explicit barriers after the inner loops when a race condition can arise.
- The Pthreads standard includes a barrier.
- However, some systems don’t implement it.
- If a barrier isn't defined we must define a function that uses a Pthreads condition variable to implement a barrier.
Parallelizing the Basic Solver Using MPI

- Choices with respect to the data structures:
  - Each process stores the entire global array of particle masses.
  - Each process only uses a single n-element array for the positions.
  - Each process uses a pointer loc_pos that refers to the start of its block of pos.
  - So on process 0 local_pos = pos; on process 1 local_pos = pos + loc_n; etc.
Pseudo-code for the MPI version of the basic n-body solver

Get input data;
for each timestep {
    if (timestep output)
        Print positions and velocities of particles;
    for each local particle loc_q
        Compute total force on loc_q;
    for each local particle loc_q
        Compute position and velocity of loc_q;
        Allgather local positions into global pos array;
}
Print positions and velocities of particles;
Gather velocities onto process 0;
if (my_rank == 0) {
    Print timestep;
    for each particle
        Print pos[particle] and vel[particle]
}
Communication In A Possible MPI Implementation of the N-Body Solver (for a reduced solver)
A Ring of Processes
Ring Pass of Positions

Phase 1

Phase 2

Phase 3

$S_0, S_4$

$S_2, S_6$

$S_3, S_7$

$S_1, S_5$

$S_3, S_7$

$S_2, S_6$

$S_0, S_4$

$S_3, S_7$

$S_2, S_6$

$S_0, S_4$
# Computation of Forces in Ring Pass (1)

<table>
<thead>
<tr>
<th>Time</th>
<th>Variable</th>
<th>Process 0</th>
<th>Process 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Start</td>
<td>loc_pos</td>
<td>(s_0, s_2)</td>
<td>(s_1, s_3)</td>
</tr>
<tr>
<td></td>
<td>loc_forces</td>
<td>0, 0</td>
<td>0, 0</td>
</tr>
<tr>
<td></td>
<td>tmp_pos</td>
<td>(s_0, s_2)</td>
<td>(s_1, s_3)</td>
</tr>
<tr>
<td></td>
<td>tmp_forces</td>
<td>0, 0</td>
<td>0, 0</td>
</tr>
<tr>
<td>After Comp of Forces</td>
<td>loc_pos</td>
<td>(s_0, s_2)</td>
<td>(s_1, s_3)</td>
</tr>
<tr>
<td></td>
<td>loc_forces</td>
<td>(f_{02}, 0)</td>
<td>(f_{13}, 0)</td>
</tr>
<tr>
<td></td>
<td>tmp_pos</td>
<td>(s_0, s_2)</td>
<td>(s_1, s_3)</td>
</tr>
<tr>
<td></td>
<td>tmp_forces</td>
<td>0, (-f_{02})</td>
<td>0, (-f_{13})</td>
</tr>
<tr>
<td>After First Comm</td>
<td>loc_pos</td>
<td>(s_0, s_2)</td>
<td>(s_1, s_3)</td>
</tr>
<tr>
<td></td>
<td>loc_forces</td>
<td>(f_{02}, 0)</td>
<td>(f_{13}, 0)</td>
</tr>
<tr>
<td></td>
<td>tmp_pos</td>
<td>(s_1, s_3)</td>
<td>(s_0, s_2)</td>
</tr>
<tr>
<td></td>
<td>tmp_forces</td>
<td>0, (-f_{13})</td>
<td>0, (-f_{02})</td>
</tr>
<tr>
<td>After Comp of Forces</td>
<td>loc_pos</td>
<td>(s_0, s_2)</td>
<td>(s_1, s_3)</td>
</tr>
<tr>
<td></td>
<td>loc_forces</td>
<td>(f_{01} + f_{02} + f_{03}, f_{23})</td>
<td>(f_{12} + f_{13}, 0)</td>
</tr>
<tr>
<td></td>
<td>tmp_pos</td>
<td>(s_1, s_3)</td>
<td>(s_0, s_2)</td>
</tr>
<tr>
<td></td>
<td>tmp_forces</td>
<td>(-f_{01}, -f_{03} - f_{13} - f_{23})</td>
<td>0, (-f_{02} - f_{12})</td>
</tr>
</tbody>
</table>
### Computation of Forces in Ring Pass (2)

<table>
<thead>
<tr>
<th>Time</th>
<th>Variable</th>
<th>Process 0</th>
<th>Process 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>After Second Comm</td>
<td>loc_pos</td>
<td>$s_0, s_2$</td>
<td>$s_1, s_3$</td>
</tr>
<tr>
<td></td>
<td>loc_forces</td>
<td>$f_{01} + f_{02} + f_{03}, f_{23}$</td>
<td>$f_{12} + f_{13}, 0$</td>
</tr>
<tr>
<td></td>
<td>tmp_pos</td>
<td>$s_0, s_2$</td>
<td>$s_1, s_3$</td>
</tr>
<tr>
<td></td>
<td>tmp_forces</td>
<td>$0, -f_{02} - f_{12}$</td>
<td>$-f_{01}, -f_{03} - f_{13} - f_{23}$</td>
</tr>
<tr>
<td>After Comp of Forces</td>
<td>loc_pos</td>
<td>$s_0, s_2$</td>
<td>$s_1, s_3$</td>
</tr>
<tr>
<td></td>
<td>loc_forces</td>
<td>$f_{01} + f_{02} + f_{03}, -f_{02} - f_{12} + f_{23}$</td>
<td>$-f_{01} + f_{12} + f_{13}, -f_{03} - f_{13} - f_{23}$</td>
</tr>
<tr>
<td></td>
<td>tmp_pos</td>
<td>$s_0, s_2$</td>
<td>$s_1, s_3$</td>
</tr>
<tr>
<td></td>
<td>tmp_forces</td>
<td>$0, -f_{02} - f_{12}$</td>
<td>$-f_{01}, -f_{03} - f_{13} - f_{23}$</td>
</tr>
</tbody>
</table>
Pseudo-code for the MPI implementation of the reduced n-body solver

source = (my_rank + 1) % comm_sz;
dest = (my_rank - 1 + comm_sz) % comm_sz;
Copy loc_pos into tmp_pos;
loc_forces = tmp_forces = 0;

Compute forces due to interactions among local particles;
for (phase = 1; phase < comm_sz; phase++) {
    Send current tmp_pos and tmp_forces to dest;
    Receive new tmp_pos and tmp_forces from source;
    /* Owner of the positions and forces we're receiving */
    owner = (my_rank + phase) % comm_sz;
    Compute forces due to interactions among my particles
    and owner's particles;
}
Send current tmp_pos and tmp_forces to dest;
Receive new tmp_pos and tmp_forces from source;
Loops iterating through global particle indexes

```c
for (loc_part1 = 0, glb_part1 = my_rank;
    loc_part1 < loc_n - 1;
    loc_part1++, glb_part1 += comm_sz)
for (glb_part2 = First_index(glb_part1, my_rank, owner, comm_sz),
    loc_part2 = Global_to_local(glb_part2, owner, loc_n);
    loc_part2 < loc_n;
    loc_part2++, glb_part2 += comm_sz)
Compute_force(loc_pos[loc_part1], masses[glb_part1],
              tmp_pos[loc_part2], masses[glb_part2],
              loc_forces[loc_part1], tmp_forces[loc_part2]);
```
## Performance of the MPI n-body solvers

<table>
<thead>
<tr>
<th>Processes</th>
<th>Basic</th>
<th>Reduced</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>17.30</td>
<td>8.68</td>
</tr>
<tr>
<td>2</td>
<td>8.65</td>
<td>4.45</td>
</tr>
<tr>
<td>4</td>
<td>4.35</td>
<td>2.30</td>
</tr>
<tr>
<td>8</td>
<td>2.20</td>
<td>1.26</td>
</tr>
<tr>
<td>16</td>
<td>1.13</td>
<td>0.78</td>
</tr>
</tbody>
</table>

(in seconds)
## Run-Times for OpenMP and MPI N-Body Solvers

<table>
<thead>
<tr>
<th>Processes/Threads</th>
<th>OpenMP</th>
<th>MPI</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Basic</td>
<td>Reduced</td>
</tr>
<tr>
<td>1</td>
<td>15.13</td>
<td>8.77</td>
</tr>
<tr>
<td>2</td>
<td>7.62</td>
<td>4.42</td>
</tr>
<tr>
<td>4</td>
<td>3.85</td>
<td>2.26</td>
</tr>
</tbody>
</table>

(in seconds)
Tree search problem (TSP)

- An NP-complete problem.
- No known solution to TSP that is better in all cases than exhaustive search.
- Ex., the travelling salesperson problem, finding a minimum cost tour.
A Four-City TSP
Search Tree for Four-City TSP

0, 0

0 → 1, 1
0 → 1 → 2, 3
0 → 1 → 3, 7
0 → 1 → 2 → 3 → 0, 20

0 → 2, 3
0 → 2 → 1, 21
0 → 2 → 3, 13
0 → 2 → 1 → 3 → 0, 34 ×
0 → 2 → 3 → 1 → 0, 22

0 → 3, 8
0 → 3 → 1, 12
0 → 3 → 2, 20 ×
0 → 3 → 1 → 2 → 0, 15
0 → 3 → 2 → 1 → 0, 43 ×
Pseudo-code for a recursive solution to TSP using depth-first search

```c
void Depth_first_search(tour_t tour) {
    city_t city;

    if (City_count(tour) == n) {
        if (Best_tour(tour))
            Update_best_tour(tour);
    } else {
        for each neighboring city
            if (Feasible(tour, city)) {
                Add_city(tour, city);
                Depth_first_search(tour);
                Remove_last_city(tour);
            }
    }
} /* Depth_first_search */
```
Pseudo-code for an implementation of a depth-first solution to TSP without recursion

```plaintext
for (city = n−1; city >= 1; city--)  
    Push(stack, city);
while (!Empty(stack)) {
    city = Pop(stack);
    if (city == NO_CITY) // End of child list, back up  
        Remove_last_city(curr_tour);
    else {
        Add_city(curr_tour, city);
        if (City_count(curr_tour) == n) {
            if (Best_tour(curr_tour))
                Update_best_tour(curr_tour);
            Remove_last_city(curr_tour);
        } else {
            Push(stack, NO_CITY);
            for (nbr = n−1; nbr >= 1; nbr--)
                if (Feasible(curr_tour, nbr))
                    Push(stack, nbr);
        }
    } /* if Feasible */
} /* while !Empty */
```
Pseudo-code for a second solution to TSP that doesn’t use recursion

Push_copy(stack, tour); // Tour that visits only the hometown
while (!Empty(stack)) {
    curr_tour = Pop(stack);
    if (City_count(curr_tour) == n) {
        if (Best_tour(curr_tour))
            Update_best_tour(curr_tour);
    } else {
        for (nbr = n-1; nbr >= 1; nbr--)
            if (Feasible(curr_tour, nbr)) {
                Add_city(curr_tour, nbr);
                Push_copy(stack, curr_tour);
                Free_tour(curr_tour);
            }
        Remove_last_city(curr_tour);
    }
}
/* Find the ith city on the partial tour */
int Tour_city(tour_t tour, int i) {
    return tour->cities[i];
} /* Tour_city */

/* Find the ith city on the partial tour */
#define Tour_city(tour, i) (tour->cities[i])
The digraph contains 15 cities. All three versions visited approximately 95,000,000 tree nodes.
Making sure we have the “best tour” (1)

- When a process finishes a tour, it needs to check if it has a better solution than recorded so far.
- The global Best_tour function only reads the global best cost, so we don’t need to tie it up by locking it. There’s no contention with other readers.
- If the process does not have a better solution, then it does not attempt an update.
Making sure we have the “best tour” (2)

- If another thread is updating while we read, we may see the old value or the new value.

- The new value is preferable, but to ensure this would be more costly than it is worth.
Making sure we have the “best tour” (3)

- In the case where a thread tests and decides it has a better global solution, we need to ensure two things:
  1) That the process locks the value with a mutex, preventing a race condition.
  2) In the possible event that the first check was against an old value while another process was updating, we do not put a worse value than the new one that was being written.

- We handle this by locking, then testing again.
First scenario

**process x**

- local tour value
  - 22

**process y**

- local tour value
  - 27

Global tour value
- 30

Steps:
- 1. test
- 2. lock
- 3. test
- 4. update
- 5. unlock
- 6. lock
- 7. test again
- 8. update
- 9. unlock
Second scenario

process x

local tour value

29

3. test
6. lock
7. test again
8. unlock

process y

local tour value

27

1. test
2. lock
4. update
5. unlock

global tour value

30
Partition_tree(my_rank, my_stack);

while (!Empty(my_stack)) {
    curr_tour = Pop(my_stack);
    if (City_count(curr_tour) == n) {
        if (Best_tour(curr_tour)) Update_best_tour(curr_tour);
    } else {
        for (city = n-1; city >= 1; city--)
            if (Feasible(curr_tour, city)) {
                Add_city(curr_tour, city);
                Push_copy(my_stack, curr_tour);
                Remove_last_city(curr_tour);
            }
    }
    Free_tour(curr_tour);
}
Dynamic Parallelization of Tree Search Using Pthreads

- Termination issues.
- Code executed by a thread before it splits:
  - It checks that it has at least two tours in its stack.
  - It checks that there are threads waiting.
  - It checks whether the new_stack variable is NULL.
Pseudo-Code for Pthreads Terminated Function (1)

```c
if (my_stack_size >= 2 && threads_in_cond_wait > 0 &&
    new_stack == NULL) {
    lock term_mutex;
    if (threads_in_cond_wait > 0 && new_stack == NULL) {
        Split my_stack creating new_stack;
        pthread_cond_signal(&term_cond_var);
    }
    unlock term_mutex;
    return 0;  /* Terminated = False; don’t quit */
} else if (!Empty(my_stack)) { /* Stack not empty, keep working */
    return 0;  /* Terminated = false; don’t quit */
} else { /* My stack is empty */
    lock term_mutex;
    if (threads_in_cond_wait == thread_count-1) { /* Last thread */
        /* running */
        threads_in_cond_wait++;
        pthread_cond_broadcast(&term_cond_var);
        unlock term_mutex;
    return 1;  /* Terminated = true; quit */
```
else { /* Other threads still working, wait for work */
    threads_in_cond_wait++;
    while (pthread_cond_wait(&term_cond_var, &term_mutex) != 0);
    /* We've been awakened */
    if (threads_in_cond_wait < thread_count) { /* We got work */
        my_stack = new_stack;
        new_stack = NULL;
        threads_in_cond_wait--;
        unlock term_mutex;
        return 0; /* Terminated = false */
    } else { /* All threads done */
        unlock term_mutex;
        return 1; /* Terminated = true; quit */
    }
} /* else wait for work */
/* else my_stack is empty */
typedef struct {
    my_stack_t new_stack;
    int threads_in_cond_wait;
    pthread_cond_t term_cond_var;
    pthread_mutex_t term_mutex;
} term_struct;

typedef term_struct* term_t;

term_t term;  // global variable
Run-times of Pthreads tree search programs

15-city problems

<table>
<thead>
<tr>
<th>Threads</th>
<th>First Problem</th>
<th></th>
<th></th>
<th>Second Problem</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Serial</td>
<td>Static</td>
<td>Dynamic</td>
<td>Serial</td>
<td>Static</td>
<td>Dynamic</td>
</tr>
<tr>
<td>1</td>
<td>32.9</td>
<td>32.7</td>
<td>34.7 (0)</td>
<td>26.0</td>
<td>25.8</td>
<td>27.5 (0)</td>
</tr>
<tr>
<td>2</td>
<td>27.9</td>
<td>28.9</td>
<td>(7)</td>
<td>25.8</td>
<td>19.2</td>
<td>(6)</td>
</tr>
<tr>
<td>4</td>
<td>25.7</td>
<td>25.9</td>
<td>(47)</td>
<td>25.8</td>
<td>9.3</td>
<td>(49)</td>
</tr>
<tr>
<td>8</td>
<td>23.8</td>
<td>22.4</td>
<td>(180)</td>
<td>24.0</td>
<td>5.7</td>
<td>(256)</td>
</tr>
</tbody>
</table>

(in seconds)

(numbers of times stacks were split)
Parallelizing the Tree Search Programs Using OpenMP

- Same basic issues implementing the static and dynamic parallel tree search programs as Pthreads.
- A few small changes can be noted.

```c
if (my_rank == whatever)

# pragma omp single
```
OpenMP emulated condition wait

/* 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);
Performance of OpenMP and Pthreads implementations of tree search

<table>
<thead>
<tr>
<th>Th</th>
<th>First Problem</th>
<th></th>
<th></th>
<th></th>
<th>Second Problem</th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Static</td>
<td>Dynamic</td>
<td></td>
<td></td>
<td>Static</td>
<td>Dynamic</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>OMP</td>
<td>Pth</td>
<td>OMP</td>
<td>Pth</td>
<td>OMP</td>
<td>Pth</td>
<td>OMP</td>
<td>Pth</td>
</tr>
<tr>
<td>1</td>
<td>32.5</td>
<td>32.7</td>
<td>33.7</td>
<td>(0)</td>
<td>34.7</td>
<td>(0)</td>
<td>25.6</td>
<td>25.8</td>
</tr>
<tr>
<td>2</td>
<td>27.7</td>
<td>27.9</td>
<td>28.0</td>
<td>(6)</td>
<td>28.9</td>
<td>(7)</td>
<td>25.6</td>
<td>25.8</td>
</tr>
<tr>
<td>4</td>
<td>25.4</td>
<td>25.7</td>
<td>33.1</td>
<td>(75)</td>
<td>25.9</td>
<td>(47)</td>
<td>25.6</td>
<td>25.8</td>
</tr>
<tr>
<td>8</td>
<td>28.0</td>
<td>23.8</td>
<td>19.2</td>
<td>(134)</td>
<td>22.4</td>
<td>(180)</td>
<td>23.8</td>
<td>24.0</td>
</tr>
</tbody>
</table>

(in seconds)
IMPLEMENTATION OF TREE SEARCH USING MPI AND STATIC PARTITIONING
Sending a different number of objects to each process in the communicator

```c
int MPI_Scatterv(
    void* sendbuf /* in */,
    int* sendcounts /* in */,
    int* displacements /* in */,
    MPI_Datatype sendtype /* in */,
    void* recvbuf /* out */,
    int recvcount /* in */,
    MPI_Datatype recvtype /* in */,
    int root /* in */,
    MPI_Comm comm /* in */
)
```
Gathering a different number of objects from each process in the communicator

```c
int MPI_Gatherv(
    void* sendbuf,    /* in */,
    int sendcount,    /* in */,
    MPI_Datatype sendtype, /* in */,
    void* recvbuf,    /* out */,
    int* recvcounts,  /* in */,
    int* displacements, /* in */,
    MPI_Datatype recvtype, /* in */,
    int root,         /* in */,
    MPI_Comm comm,    /* in */
);
```
Checking to see if a message is available

```
int MPI_Iprobe(
    int source /* in */,
    int tag /* in */,
    MPI_Comm comm /* in */,
    int* msg_avail_p /* out */,
    MPI_Status* status_p /* out */);
```
if (My_avail_tour_count(my_stack) >= 2) {
    Fulfill_request(my_stack);
    return false; /* Still more work */
} else { /* At most 1 available tour */
Send_rejects(); /* Tell everyone who's requested */
    /* work that I have none */
if (!Empty_stack(my_stack)) {
    return false; /* Still more work */
} else { /* Empty stack */
    if (comm_sz == 1) return true;
Out_of_work();
work_request_sent = false;
while (1) {
    Clear_msgs(); /* Messages unrelated to work, termination */
    if (No_work_left()) {
        return true; /* No work left. Quit */
    } else if (!work_request_sent) {
        Send_work_request(); /* Request work from someone */
        work_request_sent = true;
    } else {
        Check_for_work(&work_request_sent, &work_avail);
        if (work_avail) {
            Receive_work(my_stack);
            return false;
        }
    }
} /* while */
} /* Empty stack */
} /* At most 1 available tour */

Terminated Function for a Dynamically Partitioned TSP solver that Uses MPI.
Modes and Buffered Sends

- MPI provides four modes for sends.
  - Standard
  - Synchronous
  - Ready
  - Buffered
Printing the best tour

```c
struct {
    int cost;
    int rank;
} loc_data, global_data;

loc_data.cost = Tour_cost(loc_best_tour);
lc_data.rank = my_rank;

MPI_Allreduce(&loc_data, &global_data, 1, MPI_2INT, MPI_MINLOC, comm);
if (global_data.rank == 0) return; /* 0 already has the best tour */
if (my_rank == 0)
    Receive best tour from process global_data.rank;
else if (my_rank == global_data.rank)
    Send best tour to process 0;
```
Terminated Function for a Dynamically Partitioned TSP solver with MPI (1)

```c
if (My_avail_tour_count(my_stack) >= 2) {
    Fulfill_request(my_stack);
    return false; /* Still more work */
} else { /* At most I available tour */
    Send_rejects(); /* Tell everyone who's requested */
    /* work that I have none */
    if (!Empty_stack(my_stack)) {
        return false; /* Still more work */
    } else { /* Empty stack */
        if (comm_sz == 1) return true;
        Out_of_work();
        work_request_sent = false;
        while (1) {
            Clear_msgs(); /* Messages unrelated to work, termination */
            if (No_work_left()) {
                return true; /* No work left. Quit */
            }
        }
    }
}
```
Terminated Function for a Dynamically Partitioned TSP solver with MPI (2)

```c
} else if (!work_request_sent) {
    Send_work_request(); /* Request work from someone */
    work_request_sent = true;
} else {
    Check_for_work(&work_request_sent, &work_avail);
    if (work_avail) {
        Receive_work(my_stack);
        return false;
    }
}
} /* while */
} /* Empty stack */
} /* At most 1 available tour */
```
Packing data into a buffer of contiguous memory

```c
int MPI_Pack(
    void* data_to_be_packed /* in */,
    int to_be_packed_count /* in */,
    MPI_Datatype datatype /* in */,
    void* contiguous_buf /* out */,
    int contiguous_buf_size /* in */,
    int* position_p /* in/out */,
    MPI_Comm comm /* in */
)
```
Unpacking data from a buffer of contiguous memory

```c
int MPI_Unpack(
    void* contig_buf, /* in */
    int contig_buf_size, /* in */
    int* position_p, /* in/out */
    void* unpacked_data, /* out */
    int unpack_count, /* in */
    MPI_Datatype datatype, /* in */
    MPI_Comm comm, /* in */
)
```
<table>
<thead>
<tr>
<th>Time</th>
<th>Process 0</th>
<th>Process 1</th>
<th>Process 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Out of Work</td>
<td>Out of Work</td>
<td>Working</td>
</tr>
<tr>
<td></td>
<td>Notify 1, 2</td>
<td>Notify 0, 2</td>
<td>oow = 0</td>
</tr>
<tr>
<td></td>
<td>oow = 1</td>
<td>oow = 1</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>Send request to 1</td>
<td>Send Request to 2</td>
<td>Recv notify fr 1</td>
</tr>
<tr>
<td></td>
<td>oow = 1</td>
<td>oow = 1</td>
<td>oow = 1</td>
</tr>
<tr>
<td>2</td>
<td>oow = 1</td>
<td>Recv notify fr 0</td>
<td>Recv request fr 1</td>
</tr>
<tr>
<td></td>
<td>oow = 2</td>
<td></td>
<td>oow = 1</td>
</tr>
<tr>
<td>3</td>
<td>oow = 1</td>
<td>oow = 2</td>
<td>Send work to 1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>oow = 0</td>
</tr>
<tr>
<td>4</td>
<td>oow = 1</td>
<td>Recv work fr 2</td>
<td>Recv notify fr 0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>oow = 1</td>
<td>oow = 1</td>
</tr>
<tr>
<td>5</td>
<td>oow = 1</td>
<td>Notify 0</td>
<td>Working</td>
</tr>
<tr>
<td></td>
<td>oow = 1</td>
<td></td>
<td>oow = 1</td>
</tr>
<tr>
<td>6</td>
<td>oow = 1</td>
<td>Recv request fr 0</td>
<td>Out of work</td>
</tr>
<tr>
<td></td>
<td></td>
<td>oow = 1</td>
<td>Notify 0, 1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>oow = 2</td>
</tr>
<tr>
<td>7</td>
<td>Recv notify fr 2</td>
<td>Send work to 0</td>
<td>Send request to 1</td>
</tr>
<tr>
<td></td>
<td>oow = 2</td>
<td>oow = 0</td>
<td>oow = 2</td>
</tr>
<tr>
<td>8</td>
<td>Recv 1st notify fr 1</td>
<td>Recv notify fr 2</td>
<td>oow = 2</td>
</tr>
<tr>
<td></td>
<td>oow = 3</td>
<td>oow = 1</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>Quit</td>
<td>Recv request fr 2</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>oow = 1</td>
<td></td>
</tr>
</tbody>
</table>
## Performance of MPI and Pthreads implementations of tree search

<table>
<thead>
<tr>
<th>Th/Pr</th>
<th>First Problem</th>
<th></th>
<th></th>
<th></th>
<th>Second Problem</th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Static Pth</td>
<td>MPI</td>
<td>Dynamic Pth</td>
<td>MPI</td>
<td>Static Pth</td>
<td>MPI</td>
<td>Dynamic Pth</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td>35.8</td>
<td>40.9</td>
<td>41.9 (0)</td>
<td>56.5 (0)</td>
<td>27.4</td>
<td>31.5</td>
<td>32.3 (0)</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>29.9</td>
<td>34.9</td>
<td>34.3 (9)</td>
<td>55.6 (5)</td>
<td>27.4</td>
<td>31.5</td>
<td>22.0 (8)</td>
</tr>
<tr>
<td>4</td>
<td></td>
<td>27.2</td>
<td>31.7</td>
<td>30.2 (55)</td>
<td>52.6 (85)</td>
<td>27.4</td>
<td>31.5</td>
<td>10.7 (44)</td>
</tr>
<tr>
<td>8</td>
<td></td>
<td>35.7</td>
<td></td>
<td>45.5 (165)</td>
<td></td>
<td>35.7</td>
<td></td>
<td>16.5 (161)</td>
</tr>
<tr>
<td>16</td>
<td></td>
<td>20.1</td>
<td></td>
<td>10.5 (441)</td>
<td></td>
<td>17.8</td>
<td></td>
<td>0.1 (173)</td>
</tr>
</tbody>
</table>

(in seconds)
Concluding Remarks (1)

- In developing the reduced MPI solution to the n-body problem, the “ring pass” algorithm proved to be much easier to implement and is probably more scalable.

- In a distributed memory environment in which processes send each other work, determining when to terminate is a nontrivial problem.
Concluding Remarks (2)

- When deciding which API to use, we should consider whether to use shared- or distributed-memory.

- We should look at the memory requirements of the application and the amount of communication among the processes/threads.
Concluding Remarks (3)

- If the memory requirements are great or the distributed memory version can work mainly with cache, then a distributed memory program is likely to be much faster.

- On the other hand if there is considerable communication, a shared memory program will probably be faster.
In choosing between OpenMP and Pthreads, if there’s an existing serial program and it can be parallelized by the insertion of OpenMP directives, then OpenMP is probably the clear choice.

However, if complex thread synchronization is needed then Pthreads will be easier to use.