UNIT V

PARALLEL PROGRAM DEVELOPMENT

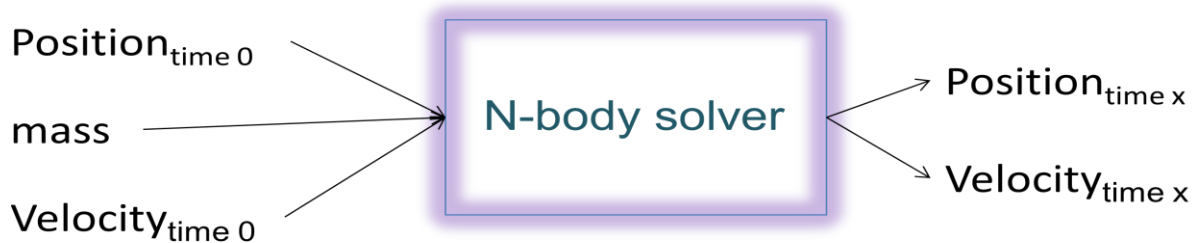
CASE STUDIES: N-BODY SOLVER, TREE SEARCH (OPENMP, MPI)

1. N-BODY SOLVER

- 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.

Definition:

- 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.



Simulating motion of planets:

- Determine the positions and velocities:
  - Newton's second law of motion.

  Newton's law of universal gravitation.

$$\mathbf{f}_{qk}(t) = -\frac{G m_q m_k}{\left|\mathbf{s}_q(t) - \mathbf{s}_k(t)\right|^3}\left[\mathbf{s}_q(t) - \mathbf{s}_k(t)\right]$$

* To find the total force on any particle by adding the forces due to all the particles. If are n particles are numbered 0,1,2,…n-1,then total force on particle q is given by

$$\mathbf{F}_q(t) = \sum_{\substack{k=0 \\ k\neq q}}^{n-1} \mathbf{f}_{qk} = -G m_q \sum_{\substack{k=0 \\ k\neq q}}^{n-1} \frac{m_k}{\left|\mathbf{s}_q(t) - \mathbf{s}_k(t)\right|^3}\left[\mathbf{s}_q(t) - \mathbf{s}_k(t)\right]$$

- Newton's second law of motion states that the force on an object is given by its mass multiplied by its acceleration so if it the acceleration of particle q is

$$\mathbf{s}_q''(t) = -G \sum_{\substack{j=0 \\ j\neq q}}^{n-1} \frac{m_j}{\left|\mathbf{s}_q(t) - \mathbf{s}_j(t)\right|^3}\left[\mathbf{s}_q(t) - \mathbf{s}_j(t)\right]$$

- To find the positions and velocities at the times

$$t = 0, \Delta t, 2\Delta t, \ldots, T\Delta t$$

->Reduced algorithm for 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];
   }
}
```

Euler's Method:

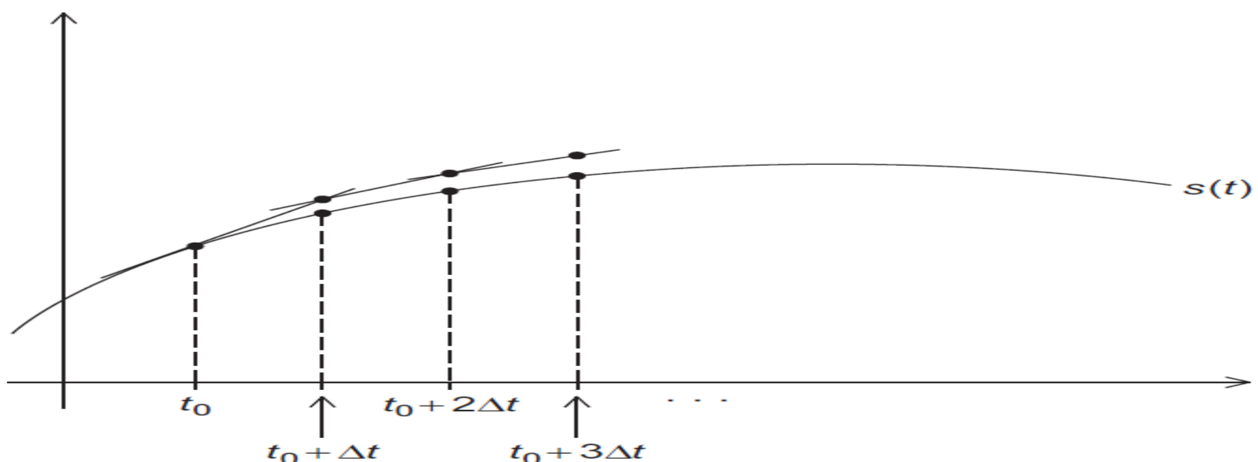
For each particle ,we need to know the values of

Its mass,

Its Position,

Its Velocity

Its acceleration and total force acting on it**.**



2.Parallelizing the basic solver using openmp:

Pseudocode for the serial program :

For each timestep

{

If (timestep output)

print positions and velocities of particles For each particle q

Compute total force on q

Compute position and velocity of q

}

The two inner loops are both iterating over particles. Parallelizing the two inner loops will map task/particles to cores,

For each timestep

{

If (timestep output)

print positions and velocities of particles

#pragma omp parallel for

For each particle q compute total force on q

Compute position and velocity of q}

->In the basic version of first loop

#pragma omp parallel for

For each particle q

This will have the desired effect on the two for each particle loops the same team of threads will be used in both loops and for every iteration of the outer loop. Every thread will print all the positions and velocities and want only one threads to do the I/O. Adding the single directives gives the following pseudo code

```
# pragma omp parallel

For each timestep

{

If(timestep output)

{

 #pragma omp single

Print position 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;

}
```

Then, possible race conditions introduced in the transition from one statement to another. Thread () completes the first for each particle loop before thread1, and it then starts updating the positions and velocities of its assigned particles in the second for each particle loop.Thread1 to use an updated position in the first **for** each particle loop.

- If thread finishes first inner loop before thread1, it will block until thread1 finish the first inner loop and it won't start the second inner loop until all the threads have finished the first. This will also prevent the possibility that a thread might rush ahead and print position and velocities before they've all been updated by second loop.

There is also an implicit barrier after the single directive although in this program the barrier isn't necessary. Since the output statement won't update any memory locations. If its ok for some threads to go ahead and start executing the next iteration before output has been completed. Furthermore the first inner for loop in the next iteration only updates the forces array so it can't cause a thread executing the output statement to print incorrect values and because of the barrier at the end of the first inner loop, no thread can race ahead and start updating positions and velocities in the second inner loop before the output has been completed. Thus could modify the single directive with a no wait clause.

- If the OpenMP implementation support it this simply eliminates the implied barrier that will prevent any one thread from getting more than a few statement ahead of any other.

Finally,

May want to add a schedule clause to each of the for directives in order to insure that the iterations have a block partition

```
#pragma omp for schedule (static,n/thread_count)
```

3. Reduced solver using Openmp:

The reduced solver has an additional inner loop the initialization of the forces array to 0.If try to use the same parallelization for the reduced solver, should also parallelize this loop with a for directive.

Parallelize the reduced solver with the following pseudocode

```
#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

}

Parallelization of the initialization of the forces should be fine as there's no dependence among the iteration. The updating of the positions and velocities is the same in both the basic and reduced solvers.

How does parallelization affect the correctness of the loop for computing the forces?The loop has the following form: there are three ways of solutions

Solution 1:

```
#pragma omp for
For each particle{
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-iff+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 variables *pos, masses, and forces* and variables are only used in a single iteration and hence can be private a thread may update elements of the *forces* array rather than those corresponding to its assigned particles. For example, suppose have two threads and four particles and using a block partition of the particles.

- Then the total force on particle 3 is given by

$F3= -f_{03}-f_{13}-f_{23}$

Thread 0 will compute f_{03} and f_{13} while 1 will compute f_{23} .Thus updates to forces[3] do create a race condition

An oblivious solution to this problem is to use a critical directive to limit access to the elements of the *forces* array. The simplest is put a critical directive before all the updates to *forces.*

#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];
```

Here, only one *force* can be updated at a time and contension for access to the critical section is actually likely to seriously degrade the performance of the program.

***Master thread:**

It will create a shared array of locks ,one for each particle and when we update an element of the forces arrays, we first set the lock corresponding to that particle

```
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]);
```

4. 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.**
  - **Used with MPI collective communication**

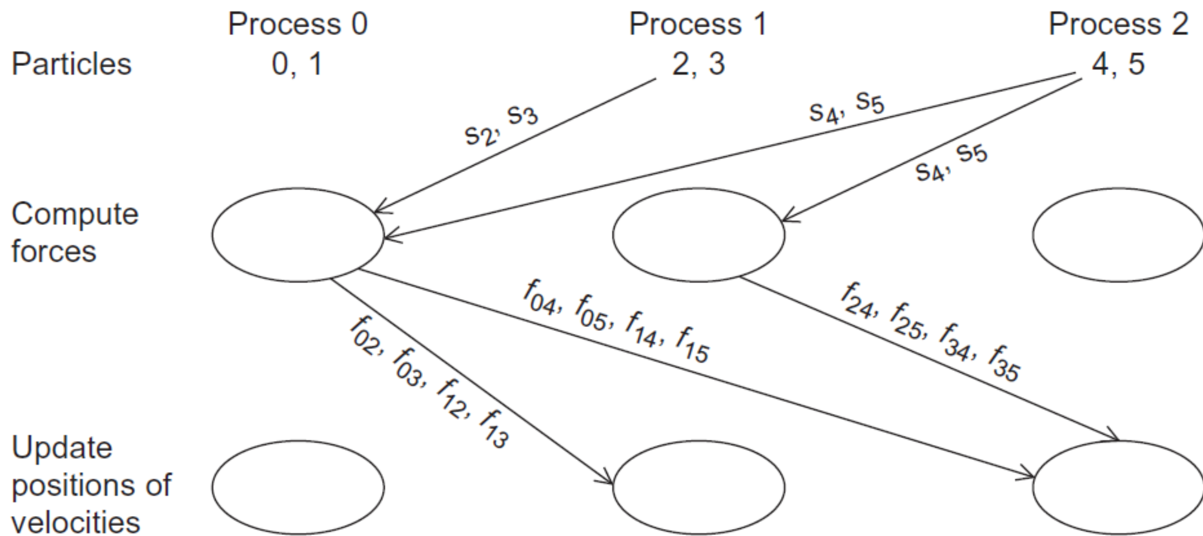
MPI_Allgather (loc_pos,loc_n,Vect_mpi_t,pos,loc_n,vect_mpi_t,comm);

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;
```

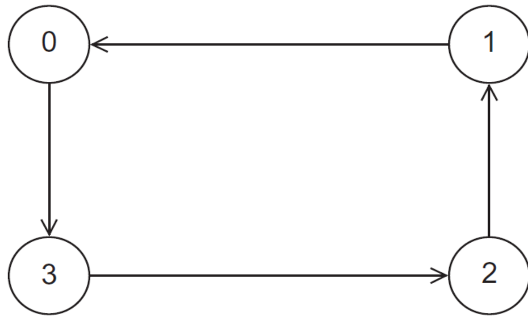
4. Parallelizing the Reduced solver using MPI:

Communication In A Possible MPI Implementation of the N-Body Solver (for a reduced solver):



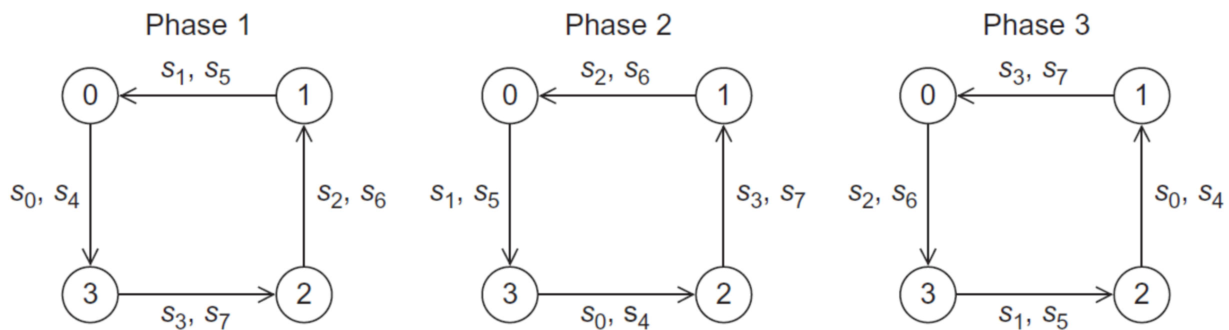
***Before computing the forces each process will need to gather a submit of the positions and after the computation of the forces each process will need to scatter some of the individual forces it has computed and add the forces it receives.**

A Ring of Processes:



Ring Pass of Positions:

The communication in a ring pass takes place in phases and during each phases each process sends the data to its "lower ranked" neighbor and receives data from its higher-ranked neighbor.



***During the Next phase each process will forward the positions it received in the first phase. This process continuous through comm _sz-1 phase until each process has received has positions of all the particles.**

***For example, if we have six particles ,then the reduced algorithm will compute the force on particle 3 as**

F3= $-f_{03}-f_{13}-f_{23}+f_{34}+f_{35}$

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;
```

First Phase Computations for Reduced Algorithm with Block Partition:

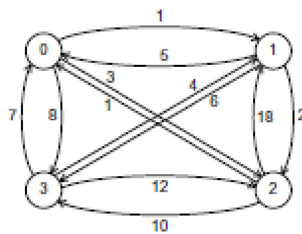
		Thread		
Thread	Particle	0	1	2
0	0	$\mathbf{f}_{01}+\mathbf{f}_{02}+\mathbf{f}_{03}+\mathbf{f}_{04}+\mathbf{f}_{05}$	0	0
	1	$-\mathbf{f}_{01}+\mathbf{f}_{12}+\mathbf{f}_{13}+\mathbf{f}_{14}+\mathbf{f}_{15}$	0	0
1	2	$-\mathbf{f}_{02}-\mathbf{f}_{12}$	$\mathbf{f}_{23}+\mathbf{f}_{24}+\mathbf{f}_{25}$	0
	3	$-\mathbf{f}_{03}-\mathbf{f}_{13}$	$-\mathbf{f}_{23}+\mathbf{f}_{34}+\mathbf{f}_{35}$	0
2	4	$-\mathbf{f}_{04}-\mathbf{f}_{14}$	$-\mathbf{f}_{24}-\mathbf{f}_{34}$	$\mathbf{f}_{45}$
	5	$-\mathbf{f}_{05}-\mathbf{f}_{15}$	$-\mathbf{f}_{25}-\mathbf{f}_{35}$	$-\mathbf{f}_{45}$

5.Serial n-body solver:

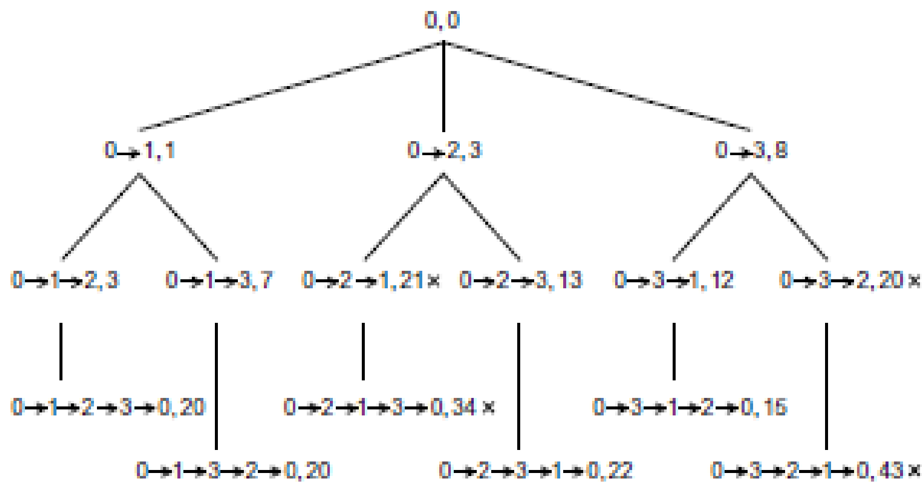
```
1     Get input data;
2     for each timestep {
3        if (timestep output) Print positions and velocities of
            particles;
4        for each particle q
5           Compute total force on q;
6        for each particle q
7           Compute position and velocity of q;
8     }
9     Print positions and velocities of particles;
```

```
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;
   }
}
```

2.TREE SEARCH



- Search tree:



6.RECURSIVE DEPTH FIRST SEARCH WITH EXAMPLE:

Using depth first search can systematically visit each node of the tree that could possibly lead to a least cost solution. The simplest form uses recursion.

The algorithm makes use of several global variables

N: total number of cities in the program

Diagraph: a data structure representing the input diagraph

Hometown: a data structure representing vertex or city 0 the salesperson' hometown.

Best tour: a data structure representing the best tour so far

```
Void depth_first_search(tour-t tour)
{
 City_t City;
if(city_count(tour)==n)
{
if(best_tour(tour);
}
else
 {
for each neighboring city
if(feasible(tour,city))
{
add_city(tour,city)
Depth_first_search(tour);
Remove_last_city(tour,city);
}
}
} /*Depth-first- search*/
```

The function city_count examines the partial tour *tour* to see if there are n cities on the partial tour. If there are know that simply need to return to the hometown to complete the tour and can check to see if the complete tour has a lower cost than the current best tour by calling Best-tour. If it does can replace the current best tour with this tour by calling the function update_best_Tour.

Nonrecursive depth first search with example.

Problem with recursion: Function calls are expensive, recursion can be slow. At any given instant of time only the current tree node is accessible. This is a problem while parallelizing tree search by dividing tree nodes among the threads or processes.

Basic Idea: Push necessary data on our own stack before branching deeper into the tree, and when we need to go back up the tree—either because we've reached a leaf or because we've found a node that can't lead to a better solution—we can pop the stack

Non Recursive depth

```
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 /
```

first solution to TSP: (Version 1)

- Stack is used to avoid recursion.
- The main control structure is while loop and the loop will be terminated when stack is empty.
- As long as the search needs to continue, make sure the stack is nonempty, and, in the first two lines, each of the non-hometown cities are added
- NO_CITY: This constant is used so that we can tell when we've visited all of the children of a tree node;
- Cities are numbered 0, 1, . . . , n – 1
- A tour contains number of cities, the cities in the tour, and the cost of it
- number of cities is city_count (tour)
- Initially, tour contains the first city 0 and cost 0
- Best_tour(tour) checks if this is the best tour so far
- Update_best_tour(tour) updates the best tour
- feasible(tour, city) checks if city has been visited, and if not, if it can be added to tour so that cost upto city < cost(best tour)
- Add_city (tour, city) adds city to tour; city must be feasible
- remove_last_city(tour, city) removes last city from tour
- push(stack,city) pushes city onto stack
- pop(stack) pops city from stack

7.parallelizing tree search using OpenMP.

- 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
```

- This will insure that thread 0 executes the following structured block of code. However, the master directive doesn't put an implicit barrier at the end of the block, so it may be necessary to also add a barrier directive after a structured block that has been modified by a master directive.

- 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.

- OpenMP provides a lock object omp_lock_t and the following functions for

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

  acquiring and relinquishing the lock, respectively:

- It also provides the function

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

- To emulate the functionality of the Pthreads function calls

```
pthread_cond_signal(&term_cond_var);
pthread_cond_broadcast(&term_cond_var);
pthread_cond_wait(&term_cond_var, &term_mutex);
```

```
pthread_cond_wait(&term_cond_var, &term_mutex);
```

- A thread that has entered the condition wait by calling
- Another thread has split its stack and created work for the waiting thread.
- All of the threads have run out of work.
- The simplest solution to emulating a condition wait in OpenMP is to use busy-waiting.

```
/* 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.

```
/* 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);
```

- Complete emulated condition wait should look something like this:
- 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);
}
```

- The awakened thread needs to reset awakened thread to -1 before it

returns from its call to the Terminated function.

8.parallelizing tree search using MPI Static Partitioning:

Implementation of Tree search using MPI with static portioning:

- **The principal differences lie in**
  - **partitioning the tree,**
  - **checking and updating the best tour, and**
  - **after the search has terminated, making sure that process 0 has a copy of the best tour for output.**

Syntax of MPI Scatter

```
int MPI_Scatter(
      void          sendbuf    /* in  */,
      int           sendcount  /* in  */,
      MPI_Datatype  sendtype   /* in  */,
      void*         recvbuf    /* out */,
      int           recvcount  /* in  */,
      MPI_Datatype  recvtype   /* in  */,
      int           root       /* in  */,
      MPI_Comm      comm       /* in  */);
```

- Process root sends sendcount objects of type sendtype from sendbuf to each process in comm. Each process in comm receives recvcount objects of type recvtype into recvbuf. Most of the time, sendtype and recvtype are the same and sendcount and recvcount are also the same. In any case, it’s clear that the root process must send the same number of objects to each process.
- MPI_Gatherv generalizes MPI Gather

```
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  */);
```

- MPI_Send to send it to all the other processes:

```
for (dest = 0; dest < comm_sz; dest++)
   if (dest != my_rank)
      MPI_Send(&new_best_cost, 1, MPI_INT, dest, NEW_COST_TAG,
          comm);
```

- Destination processes can periodically check for the arrival of new best tour costs. We can't use MPI_Recv to check for messages since it's blocking; if a process calls

```
MPI_Recv(&received_cost, 1, MPI_INT, MPI_ANY_SOURCE, NEW_COST_TAG,
      comm, &status);
```

the process will block until a matching message arrives.

- The process that finds a new best cost use MPI Send to send it to all the other processes:
  for (dest = 0; dest < comm sz;
  dest++) if (dest != my rank)
  MPI_Send(&new best cost, 1, MPI_INT, dest, NEW_COST_TAG,comm);

- The destination processes can periodically check for the arrival of new best tour
- costs. We can't use MPI Recv to check for messages since it's blocking; if a process calls
  MPI_Recv(&received cost, 1, MPI_INT, MPI_ANY_SOURCE, NEW_COST _TAG, comm, &status);
- the process will block until a matching message arrives
- MPI provides a function that only checks to see if a message is available; it doesn't actually try to receive a message. It's called MPI_Iprobe, and its syntax is

```
int MPI_Iprobe(
          int          source       /* in  */,
          int          tag          /* in  */,
          MPI_Comm     comm         /* in  */,
          int*         msg_avail_p  /* out */,
          MPI_Status*  status_p     /* out */);
```

- To check for a message with a new cost from any process, we can call MPI_Iprobe(MPI_ANY_SOURCE,NEW_COST_TAG,comm.,&msg_avail, comm,&msg_avail, &status);
- If msg avail is true, then we can receive the new cost with a call to MPI_Recv:

```
MPI_Iprobe(MPI_ANY_SOURCE, NEW_COST_TAG, comm, &msg_avail,
      &status);
while (msg_avail) {
   MPI_Recv(&received_cost, 1, MPI_INT, status.MPI_SOURCE,
         NEW_COST_TAG, comm, MPI_STATUS_IGNORE);
   if (received_cost < best_tour_cost)
      best_tour_cost = received_cost;
   MPI_Iprobe(MPI_ANY_SOURCE, NEW_COST_TAG, comm, &msg_avail,
     &status);
}  /* while */
```

MPI_Recv(&received_cost, 1, MPI_INT, status.MPI_SOURCE, NEW_COST_ TAG, comm, MPI_STATUS_IGNORE);

- Modes and Buffered Sends
  - MPI provides four modes for sends: standard, synchronous, ready, and buffered
- Each mode has a different function: MPI_Send, MPI_Ssend, MPI_Rsend, and MPI_Bsend, respectively, but the argument lists are

```
int MPI_Xsend(
      void*          message        /* in */,
      int            message_size   /* in */,
      MPI_Datatype   message_type   /* in */,
      int            dest           /* in */,
      int            tag            /* in */,
      MPI_Comm       comm           /* in */);
```

identical to the argument lists for MPI_Send:

- The buffer that's used by MPI_Bsend must be turned over to the MPI implementation with a call to MPI_Buffer_attach:

```
int MPI_Buffer_attach(
      void*  buffer       /* in */,
      int    buffer_size  /* in */);
```

- Printing the best tour:
  - After all the processes have completed their searches,they can all call MPI_ Allreduce and the process with the global best tour can then send it to process 0 for output.

```
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;
```

  - When the call to MPI_Allreduce returns, we have two alternatives: . If process 0 already has the best tour, we simply return. . Otherwise, the process owning the best tour sends it to process 0.

9.parallelizing tree search using MPI Dynamic Partitioning:

Implementation of Tree search using MPI with Dynamic Partioning:

- When a thread ran out of work—that is, its stack was empty—it went into a condition wait (Pthreads) or a busy-wait (OpenMP) until it either received additional work or it was notified that there was no more work. In the first case, it returned to searching for a best tour. In the second case, it quit. A thread that had at least two records on its stack would give half of its stack to one of the waiting threads.
- When a process enters the Terminated function, it can check to see if there's a request for work from some other process. If there is, and the process that has just entered Terminated has work, it can send part of its stack to the requesting process. If there is a request, and the process has no work available, it can send a rejection.

- Before entering the apparently infinite **while** loop (Line 13), we set the variable work_request_sent to false (Line 12). As its name suggests, this variable tells us whether we've sent a request for work to another process; if we have, we know that we should wait for work or a message saying "no work available" from that process before sending out a request to another process. The **while**(1) loop is the distributed-memory version of the OpenMP busy-wait loop.We are essentially waiting until we either receive work from another process or we receive word that the search has been completed.
- When we enter the **while**(1) loop, we deal with any outstanding messages in Line 14
- After clearing out outstanding messages, we iterate through the possibilities:
- . The search has been completed, in which case we quit (Lines 15–16).
- . We don't have an outstanding request for work, so we choose a process and send it a request (Lines 17–19). We'll take a closer look at the problem of which process should be sent a request shortly.
- . We do have an outstanding request for work (Lines 21–25). So we check whether the request has been fulfilled or rejected. If it has been fulfilled, we receive the new work and return to searching. If we received a rejection, we set work_request _sent to false and continue in the loop. If the request was neither fulfilled nor rejected, we also continue in the **while**(1) loop.

- **Pseudocode for Terminated function**

```
1  if (My_avail_tour_count(my_stack) >= 2) {
2     Fulfill_request(my_stack);
3     return false;  /* Still more work */
4  } else { /* At most 1 available tour */
5     Send_rejects();  /* Tell everyone who's requested */
6                      /* work that I have none         */
7     if (!Empty_stack(my_stack)) {
8        return false;  /* Still more work */
9     } else {  /* Empty stack */
10       if (comm_sz == 1) return true;
11       Out_of_work();
12       work_request_sent = false;
13       while (1) {
14          Clear_msgs();  /* Msgs unrelated to work, termination */
15          if (No_work_left()) {
16             return true;  /* No work left.  Quit */
17          } else if (!work_request_sent) {
18             Send_work_request();  /* Request work from someone */
19             work_request_sent = true;
20          } else {
21             Check_for_work(&work_request_sent, &work_avail);
22             if (work_avail) {
23                Receive_work(my_stack);
24                return false;
25             }
26          }
27       }  /* while */
28    } /* Empty stack */
29 } /* At most 1 available tour */
```

- **My_avail_tour _count** : The function My_avail_tour_count can simply return the size of the process' stack.It can also make use of a "cutoff length." When a partial tour has already visited most of the cities, there will be very little work associated with the subtree rooted at the partial tour. Since sending a partial tour is likely to be a relatively expensive operation, it may make sense to only send partial tours with fewer than some cutoff number of edges.
- **Fulfill_ request:** If a process has enough work so that it can usefully split its stack, it calls Fulfill_request (Line 2). Fulfill_request uses MPI Iprobe to check for a request for work from another process. If there is a request, it receives it, splits its stack, and sends work to the requesting process. If

there isn't a request for work, the process just returns.

- **Splitting the stack:**
- A Split_ stack function is called by Fulfill_request. It uses the same basic algorithm as the Pthreads and OpenMP functions, that is, alternate partial tours with fewer than split_cutoff cities are collected for sending to the process that has requested work. However, in the shared-memory programs, we simply copy the tours (which are pointers) from the original stack to a new stack.
- MPI provides a function, MPI_Pack, for packing data into a buffer of contiguous memory. It also provides a function, MPI_Unpack, for unpacking data from a buffer of contiguous memory
- Syntax of MPI_pack:
-

```
int MPI_Pack(
      void*          data_to_be_packed    /* in     */,
      int            to_be_packed_count   /* in     */,
      MPI_Datatype   datatype             /* in     */,
      void*          contig_buf           /* out    */,
      int            contig_buf_size      /* in     */,
      int*           position_p           /* in/out */,
      MPI_Comm       comm                 /* in     */);
```

- Syntax of MPI_Unpack

```
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     */);
```

- MPI_Pack takes the data in data_to_be_packed and packs it into contig_buf
- MPI_Unpack reverses the process.

10.Explain about the static and dynamic parallelization of tree search using pthreads

Pseudo-code for a Pthreads implementation of a statically parallelized solution to TSP:

```
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.

```
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 */
```