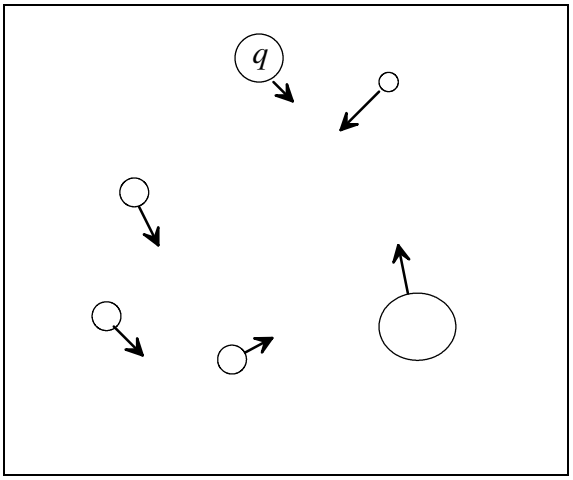
MPI versions of Chapter 6 "real-world" examples: n-body problem and TSP (traveling-salesperson problem).



Serial code:

```
Get input data;
for each timestep {
   if (timestep output) Print positions and velocities of par
   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;
```

The for each particle q:
Computer total force on q"
code needs to perform individual force calculations:
Nodes:

- row 0 are the forces on particle 0 by other particles, etc.
- matrix is "symmetric", except opposite forces are negated (two versions: *basic* and *reduced* utilitizing symmetry)

$$\begin{bmatrix} 0 & \mathbf{f}_{01} & \mathbf{f}_{02} & \cdots & \mathbf{f}_{0,n-1} \\ -\mathbf{f}_{01} & 0 & \mathbf{f}_{12} & \cdots & \mathbf{f}_{1,n-1} \\ -\mathbf{f}_{02} & -\mathbf{f}_{12} & 0 & \cdots & \mathbf{f}_{2,n-1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ -\mathbf{f}_{0,n-1} & -\mathbf{f}_{1,n-1} & -\mathbf{f}_{2,n-1} & \cdots & 0 \end{bmatrix}$$

1. In the MPI version we could assign an equal number of particles to each process(or) with:
   - 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.

MPI algorithm for basic n-body problem:

```
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;
```

```
int MPI_Allgather(
      void*          send_buf_p  /* in  */,
      int            send_count  /* in  */,
      MPI_Datatype   send_type   /* in  */,
      void*          recv_buf_p  /* out */,
      int            recv_count  /* in  */,
      MPI_Datatype   recv_type   /* in  */,
      MPI_Comm       comm        /* in  */);
```

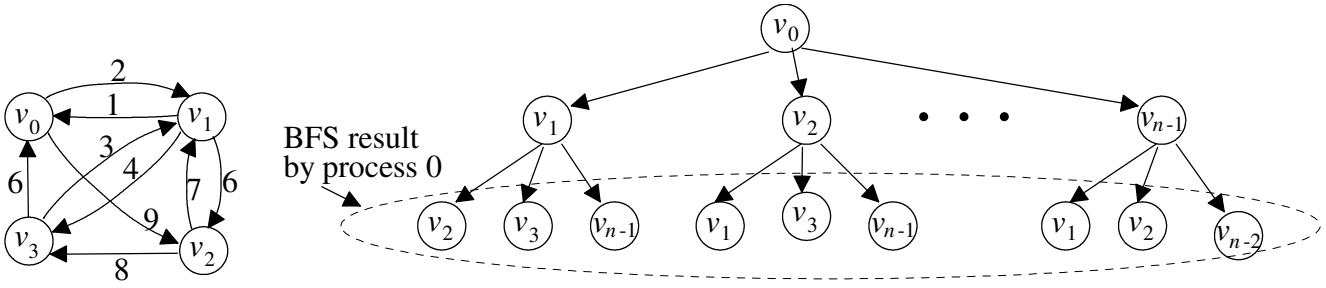
NOTES on MPI_Allgather:

- Concatenates the contents of each process' send_buf_p and stores this in each process' recv_buf_p.
- recv_count is the amount of data being received from each process.

a) Why is MPI_Allgather only used to distribute the positions of all particles, but not their velocities?

b) How are processes synchronized for each timestep?

2. As with pthreads we can statically allocate the work to each MPI process by having process 0 do a breadth-first search (BFS) of "*state-space tree*" until it has enough tree nodes for all the processes.



.

a) What MPI routine could we use to distribute the tree nodes to all processes?

b) Would each process get the same number of tree nodes?

c) Each process could run independently to completion using their local best tour for pruning, but why is maintaining a "global" best tour better?

d) What is wrong with each of the following ways to try to maintain a global best tour?

- Using MPI_Bcast to broadcast new best tour found by a process

- Looping to MPI_Send a "new best tour message" to all other processes individually with each process periodically performing a MPI_Recv

e) A process can use the non-blocking MPI_Iprobe routine to check to see if a message is available, but which might be the `source` process?

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

f) How might we use the `tag` parameter?

4. If a process runs out of work (completed searching its assigned subtree(s)), what should it do?

5. If process 0 is out of work and received a "completion message" from everybody, how can it determine the global best tour?

6. As with pthreads, the MPI processes can dynamically allocate the tree search by allowing an MPI process which runs out of work to obtain work from another process. Outline the procedure for processes to dynamically request and receive work from another process.

7. How can we detect that all processes have runs out of work?

```
/*--------------------------------------------------------------------
 * Function:  Terminated
 * Purpose:   Determine whether the program has terminated.  If it
 *            hasn't try to fulfill requests for work if process
 *            has any.  Otherwise look for work until some is
 *            received or the program terminates.
 * In/out args:  stack, avail
 */
int Terminated(my_stack_t stack, my_stack_t avail) {
   int work_request_sent;
   int work_avail;
   int tour_count = Short_tour_count(stack);

   if (tour_count > min_split_sz) {
      Fulfill_request(stack, tour_count, avail);
      return FALSE;
   } else { /* Not enough work to fulfill a request */
      Send_rejects();
      if (!Empty_stack(stack)) {
         return FALSE;
      } else {  /* Empty stack */
         Send_energy();
         if (comm_sz == 1) return TRUE;
         work_request_sent = FALSE;
         while (1) {
            Send_rejects();
            Look_for_best_tours();  // Get them out of the msg queue
            if (Term_msg()) {
                 return TRUE;
              } else if (!work_request_sent) {
                 Send_work_request();
                 work_request_sent = TRUE;
              } else {
                 Check_for_work(&work_request_sent,
                                &work_avail);
                 if (work_avail) {
                    Receive_work(stack, avail);
                    return FALSE;
                 }
              }
          }  /* while */
       } /* Empty stack */
   }  /* Not enough work */
} /* Terminated */
```

```
void Check_for_work(int* work_request_sent_p, int*
                    work_avail_p) {
   int msg_recd, buf = 0;
   MPI_Status status;

   MPI_Iprobe(work_req_dest, FULFILL_REQ_TAG, comm,
              &msg_recd, &status);
   if (msg_recd) { /* We got work! */
      *work_avail_p = TRUE;
   } else { /* We didn't get work */
      MPI_Iprobe(work_req_dest, REJECT_REQ_TAG, comm,
                 &msg_recd, &status);
      if (msg_recd) {  /* We got a reject */
         MPI_Recv(&buf, 0, MPI_INT, work_req_dest,
               REJECT_REQ_TAG, comm,MPI_STATUS_IGNORE);
         *work_request_sent_p = FALSE;
         *work_avail_p = FALSE;
      } else { /* We didn't get anything from
                  work_req_dest */
         *work_request_sent_p = TRUE;  // Not necessary
         *work_avail_p = FALSE;
      }
   }  /* Didn't get work */
}  /* Check_for_work */
```

```
/*---------------------------------------------------
 * Function:    Par_tree_search
 * Purpose:     Use multiple threads to search a tree
 * In arg:
 *    rank:     thread rank
 * Globals in:
 *    n:        total number of cities in the problem
 * Notes:
 * 1. The Update_best_tour function will modify the
 *    global vars loc_best_tour and best_tour_cost
 */
void Par_tree_search(void) {
   city_t nbr;
   my_stack_t stack;  // Stack for searching
   my_stack_t avail;  // Stack for unused tours
   tour_t curr_tour;

   avail = Init_stack();
   stack = Init_stack();
   Partition_tree(stack);

   while (!Terminated(stack, avail)) {
      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, avail);
               Remove_last_city(curr_tour);
            }
      }
      Free_tour(curr_tour, avail);
   }
   Free_stack(stack);
   Free_stack(avail);
   MPI_Barrier(comm);
   Get_global_best_tour();

   Cleanup_msg_queue();

}  /* Par_tree_search */
```

```
/*---------------------------------------------------
 * Function:  Send_rejects
 * Purpose:   Send a reject message to each process
 *            that's requested work
 */
void Send_rejects(void) {
   int req_recd, buf = 0;
   MPI_Status status;

   MPI_Iprobe(MPI_ANY_SOURCE, WORK_REQ_TAG, comm,
              &req_recd, &status);
   while (req_recd) {
      MPI_Recv(&buf, 0, MPI_INT, status.MPI_SOURCE,
               WORK_REQ_TAG, comm, MPI_STATUS_IGNORE);
      MPI_Send(&buf, 0, MPI_INT, status.MPI_SOURCE,
               REJECT_REQ_TAG, comm);
      MPI_Iprobe(MPI_ANY_SOURCE, WORK_REQ_TAG, comm,
                 &req_recd, &status);
   }
}  /* Send_rejects */
```

```
/*---------------------------------------------------------
 * Function:  Get_global_best_tour
 * Purpose:   Get global best tour to process 0
 */
void Get_global_best_tour(void) {
   struct {
      int cost;
      int rank;
   } loc_data, global_data;
   loc_data.cost = Tour_cost(loc_best_tour);
   loc_data.rank = my_rank;

   /* Both 0 and the owner of the best tour need global_data */
   MPI_Allreduce(&loc_data, &global_data, 1, MPI_2INT,
                 MPI_MINLOC, comm);
   if (global_data.rank == 0) return;
   if (my_rank == 0) {
      MPI_Recv(loc_best_tour->cities, n+1, MPI_INT,
               global_data.rank, 0, comm, MPI_STATUS_IGNORE);
      loc_best_tour->cost = global_data.cost;
      loc_best_tour->count = n+1;
   } else if (my_rank == global_data.rank) {
      MPI_Send(loc_best_tour->cities, n+1, MPI_INT, 0, 0, comm);
   }
}  /* Get_global_best_tour */
```

```
/*---------------------------------------------------------
 * Function:   Bcast_tour_cost
 * Purpose:    Asynchronous broadcast of tour cost
 *
 * Note:
 *  MPI_Bcast is a point of synchronization for the processes.
 *  So it can't be used.
 */
void Bcast_tour_cost(int tour_cost) {
   int offset = Get_cost_msg(tour_cost);
   int dest;

   for (dest = 0; dest < comm_sz; dest++)
      if (dest != my_rank)
         MPI_Isend(&Cost_msg(cost_msgs,offset), 1, MPI_INT,
                   dest, TOUR_TAG, comm, &Cost_req(cost_msgs,
                   offset, dest));
#  ifdef STATS
   best_costs_bcast++;
#  endif
}  /* Bcast_tour_cost */
```