L15: Putting it together: N-body (Ch. 6)

October 30, 2012

Outline

- Review MPI Communication
  - Blocking
  - Non-Blocking
  - One-Sided
  - Point-to-Point vs. Collective
- Chapter 6 shows two algorithms (N-body and Tree Search) written in the three programming models (OpenMP, Pthreads, MPI)
  - How to approach parallelization of an entire algorithm (Foster's methodology is used in the book)
  - What do you have to worry about that is different for each programming model?

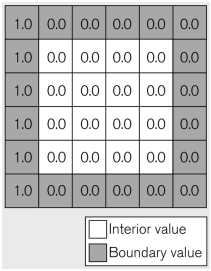
THE UNIVERSITY OF UTAH

Review p2p example: 2D relaxation

Replaces each interior value by the average of its four nearest neighbors.

Sequential code:

```
for (i=1; i<n-1; i++)
  for (j=1; j<n-1; j++)
    b[i,j] = (a[i-1][j]+a[i][j-1]+
         a[i+1][j]+a[i][j+1])/4.0;
```



THE UNIVERSITY OF UTAH

MPI code, main loop of 2D SOR computation

```
 1  #define Top      0
 2  #define Left     0
 3  #define Right    (Cols-1)
 4  #define Bottom   (Rows-1)
 5
 6  #define NorthPE(i)       ((i)-Cols)
 7  #define SouthPE(i)       ((i)+Cols)
 8  #define EastPE(i)        ((i)+1)
 9  #define WestPE(i)        ((i)-1)
...
101  do
102  { /*
103    * Send data to four neighbors
104    */
105    if(row !=Top)                                  /* Send North */
106    {
107      MPI_Send(&val[1][1], Width-2, MPI_FLOAT,
108               NorthPE(myID), tag, MPI_COMM_WORLD);
109    }
110
111    if(col !=Right)                                /* Send East */
112    {
113      for(i=1; i<Height-1; i++)
114      {
115        buffer[i-1]=val[i][Width-2];
116      }
117      MPI_Send(buffer, Height-2, MPI_FLOAT,
118               EastPE(myID), tag, MPI_COMM_WORLD);
119    }
120
121    if(row !=Bottom)                               /* Send South */
122    {
123      MPI_Send(&val[Height-2][1], Width-2, MPI_FLOAT,
124               SouthPE(myID), tag, MPI_COMM_WORLD);
125    }
126
127    if(col !=Left)                                 /* Send West */
128    {
```


THE UNIVERSITY OF UTAH

MPI code, main loop of 2D SOR computation, cont.

```
129       for(i=1; i<Height-1; i++)
130       {
131         buffer[i-1]=val[i][1];
132       }
133       MPI_Send(buffer, Height-2, MPI_FLOAT,
134                WestPE(myID), tag, MPI_COMM_WORLD);
135     }
136
137     /*
138      * Receive messages
139      */
140     if(row !=Top)                          /* Receive from North */
141     {
142       MPI_Recv(&val[0][1], Width-2, MPI_FLOAT,
143                NorthPE(myID), tag, MPI_COMM_WORLD, &status);
144     }
145
146     if(col !=Right)                        /* Receive from East */
147     {
148       MPI_Recv(&buffer, Height-2, MPI_FLOAT,
149                EastPE(myID), tag, MPI_COMM_WORLD, &status);
150       for(i=1; i<Height-1; i++)
151       {
152         val[i][Width-1]=buffer[i-1];
153       }
154     }
155
156     if(row !=Bottom)                       /* Receive from South */
157     {
158       MPI_Recv(&val[0][Height-1], Width-2, MPI_FLOAT,
159                SouthPE(myID), tag, MPI_COMM_WORLD, &status);
160     }
161
162     if(col !=Left)                         /* Receive from West */
163     {
164       MPI_Recv(&buffer, Height-2, MPI_FLOAT,
165                WestPE(myID), tag, MPI_COMM_WORLD, &status);
166       for(i=1; i<Height-1; i++)
167       {
168         val[i][0]=buffer[i-1];
169       }
170     }
171
172     delta=0.0; /* Calculate average, delta for all points */
173     for(i=1; i<Height-1; i++)
174     {
175       for(j=1; j<Width-1; j++)
```


MPI code, main loop of 2D SOR computation, cont.

```
177         average=(val[i-1][j]+val[i][j+1]+
178                  val[i+1][j]+val[i][j-1])/4;
179         delta=Max(delta, Abs(average-val[i][j]));
180         new[i][j]=average;
181       }
182     }
183
184     /* Find maximum diff */
185     MPI_Reduce(&delta, &globalDelta, 1, MPI_FLOAT, MPI_MIN,
186                RootProcess, MPI_COMM_WORLD);
187     Swap(val, new);
188   } while(globalDelta < THRESHOLD);
```


Question: Does this lead to deadlock?

- No! Why not?
- Even though communication is blocking,
  - Send returns when MPI library on receiving end acknowledges that data has been received
  - The receiving process may not yet have received the data
  - So, a sequence of blocking Sends will not block
- What about receives?
  - Receiving process must wait until data arrives before proceeding
  - A sequence of blocking receives must be received in order

Foster's methodology (Chapter 2)

1. Partitioning: divide the computation to be performed and the data operated on by the computation into small tasks.

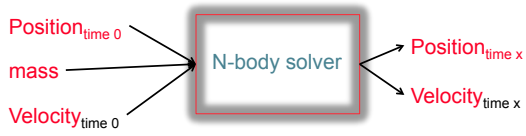
   The focus here should be on identifying tasks that can be executed in parallel.
2. Communication: determine what communication needs to be carried out among the tasks identified in the previous step.
3. Agglomeration or aggregation: combine tasks and communications identified in the first step into larger tasks.

   For example, if task A must be executed before task B can be executed, it may make sense to aggregate them into a single composite task.
4. Mapping: assign the composite tasks identified in the previous step to processes/threads.

   This should be done so that communication is minimized, and each process/thread gets roughly the same amount of work.

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.

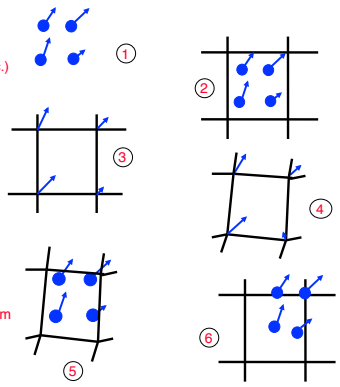


Example N-Body: Material Point Method (MPM)

1. Lagrangian material points carry all state data (position, velocity, stress, etc.)
2. Overlying mesh defined
3. Particle state projected to mesh, e.g.:
$$v_g = \sum_p S_{gp} m_p v_p \Big/ \sum_p S_{gp} m_p$$
4. Conservation of momentum solved on mesh giving updated mesh velocity and (in principal) position.

Stress at particles computed based on gradient of the mesh velocity.

5. Particle positions/velocities updated from mesh solution.
6. Discard deformed mesh. Define new mesh and repeat



Calculate Force as a function of mass and positions

Force between two particles (6.1)

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

Total force on a particle (6.2)

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

Calculate acceleration

(6.3)

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

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

Serial pseudo-code: calculates position of particles at each time step

```
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: Basic Algorithm

For all pairs (except <q,q>) compute Total Force

```
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;
   }
}
```


Computation of the forces: Reduced Algorithm

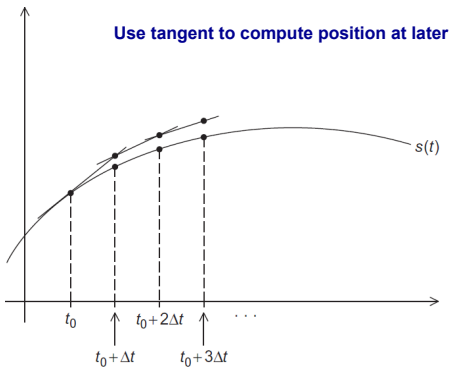
Reduce calculation of Total Force, capitalizing on $f_{qk} = -f_{kq}$
Compute half the pairs

```
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];
   }
}
```


Compute position using Euler's Method

Use tangent to compute position at later time step



OpenMP parallelization: Basic Algorithm

For all pairs (except <q,q>) compute Total Force
Parallelization? Scheduling?

```
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;
   }
}
```


Computation of the forces: Reduced Algorithm

Reduce calculation of Total Force, capitalizing on $f_{qk} = -f_{kq}$
Compute half the pairs
Parallelization? Scheduling?

```
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];
   }
}
```


Other topics on OpenMP parallelization

Data structure choice – group together in structure or separate arrays?

Thread creation

Alternative synchronization (reduced method)

Private data (reduced method)

Nowait

I/O (single)

What's different with Pthreads

- Must write more code to do basic OpenMP things:
  - Create threads, pass parameters, determine number and mapping of iterations and initiate "parallel loop"
  - Explicit barriers between "loops"
  - Explicit synchronization
  - Explicit declaration of global data that is shared, or thread-local declaration of private data

MPI Parallelization: Basic version

- 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.

MPI Parallelization: Basic version

```
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;
```


Detour: MPI_Scatter()

MPI_Scatter()

```
int MPI_Scatter(                    // Scatter routine
  void          *sendbuffer,        // Address of the data to send
  int           sendcount,          // Number of data elements to send
  MPI_Datatype  sendtype,           // Type of data elements to send
  int           destbuffer,         // Address of buffer to receive data
  int           destcount,          // Number of data elements to receive
  MPI_Datatype  desttype,           // Type of data elements to receive
  int           root,               // Rank of the root process
  MPI_Comm      *comm               // An MPI communicator
);
```

Arguments:

- The first three arguments specify the address, size, and type of the data elements to send to each process. These arguments only have meaning for the root process.
- The second three arguments specify the address, size, and type of the data elements for each receiving process. The size and type of the sending data and the receiving data may differ as a means of converting data types.
- The seventh argument specifies the root process that is the source of the data.
- The eighth argument specifies the MPI communicator to use.

Notes:

This routine distributes data from the root process to all other processes, including the root. A more sophisticated version of the routine, `MPI_Scatterv()`, allows the root process to send different amounts of data to the various processes. Details can be found in the MPI standard.

Return value:

An MPI error code.

Distribute Data from input using a scatter operation

```
16  length_per_process=length/size;
17  myArray=(int *) malloc(length_per_process*sizeof(int));
18
19  array=(int *) malloc(length*sizeof(int));
20
21  /* Read the data, distribute it among the various processes */
22  if(myID==RootProcess)
23  {
24    if((fp=fopen(*argv, "r"))==NULL)
25    {
26      printf("fopen failed on %s\n", filename);
27      exit(0);
28    }
29    fscanf(fp,"%d", &length);       /* read input size */
30
31    for(i=0; i<length-1; i++)       /* read entire input file */
32    {
33      fscanf(fp,"%d", myArray+i);
34    }
35  }
36
37  MPI_Scatter(Array, length_per_process, MPI_INT,
38              myArray, length_per_process, MPI_INT,
39              RootProcess, MPI_COMM_WORLD);
```


MPI_Gather is analogous

MPI_Gather: Gathers together values from a group of processes

int MPI_Gather(void *sendbuf, int sendcnt, MPI_Datatype sendtype, void *recvbuf, int recvcnt, MPI_Datatype recvtype, int root, MPI_Comm comm)

Input Parameters

sendbuf starting address of send buffer (choice)

sendcount number of elements in send buffer (integer)

sendtype data type of send buffer elements (handle)

recvcount number of elements for any single receive (integer, significant only at root)

recvtype data type of recv buffer elements (significant only at root) (handle)

root rank of receiving process (integer)

comm communicator (handle)

Output Parameter

recvbuf address of receive buffer (choice, significant only at root)

Detour: MPI_Allgather (a collective)

MPI_Allgather: Gathers data from all tasks and distributes the combined data to all tasks

int MPI_Allgather (void *sendbuf, int sendcount, MPI_Datatype sendtype, void *recvbuf, int recvcount, MPI_Datatype recvtype, MPI_Comm comm)

Input Parameters

sendbuf starting address of send buffer (choice)
sendcount number of elements in send buffer (integer)
sendtype data type of send buffer elements (handle)
recvcount number of elements received from any process (integer)
recvtype data type of receive buffer elements (handle)
comm communicator (handle)

Output Parameter

recvbufaddress of receive buffer (choice)

I/O: Ch. 6, p. 291

```
if (my_rank == 0) {
    for each particle, read masses[particle], pos[particle], vel[particle];
}

MPI_Bcast(masses, n, MPI_DOUBLE, 0, comm);
MPI_Bcast(pos, n, vect_mpi_t, loc_vel, loc_n, vect_mpi_t, 0, comm)
MPI_Scatter(vel, loc_n, vect_mpi_t, 0, comm);
```

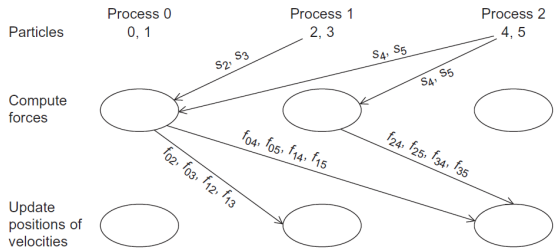
MPI Issues

- Communicating standard data types performs much better than derived data types like structures
  - Separate fields (position, velocity, mass) into arrays that can be communicated independently
- Is it feasible for each processor to have all of the processor mass data locally? The position data?
  - Depends on how many bodies and capacity constraints

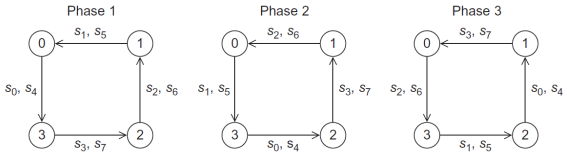
MPI Parallelization: Basic version

```
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;
```


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



Ring Pass of Positions



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

```
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]);
```