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;

MPI code, main loop of 2D SOR computation

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

```c
177 average = (val[i][j] + val[i][j+1]) / 2;
178 diff = val[i][j] - average;
179 MPI_Reduce(&diff, &delta, 1, MPI_FLOAT, MPI_MAX, 0, MPI_COMM_WORLD);
180 if (delta > threshold) {
181   new[i][j] = average;
182 }
183 }
184 /* Find maximum diff */
185 MPI_Reduce(&globaldelta, &delta, 1, MPI_FLOAT, MPI_MAX, 0, MPI_COMM_WORLD);
186 if (delta > threshold) {
187   new[i][j] = globaldelta;
188 }
```

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.

N-body solver

<table>
<thead>
<tr>
<th>Position_{time 0}</th>
<th>N-body solver</th>
<th>Position_{time x}</th>
</tr>
</thead>
<tbody>
<tr>
<td>mass</td>
<td></td>
<td>Velocity_{time 0}</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Velocity_{time x}</td>
</tr>
</tbody>
</table>

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.: $v_i = \sum S_{gp}v_{pj}/\sum S_{gp}$
4. Conservation of momentum solved on mesh giving updated mesh velocity and (in principal) position.
5. Stress at particles computed based on gradient of the mesh velocity.
6. Particle positions/velocities updated from mesh solution.
7. Discard deformed mesh.
8. Define new mesh and repeat

Calculate Force as a function of mass and positions

Force between two particles

$$F_{qk}(t) = -\frac{Gm_qm_k}{S_q(t) - S_k(t)^3} [S_q(t) - S_k(t)]$$

Total force on a particle

$$F_q(t) = \sum_{k=0}^{n-1} F_{qk} = -Gm_q \sum_{k=0, k \neq q}^{n-1} \frac{m_k}{|S_q(t) - S_k(t)|^3} [S_q(t) - S_k(t)]$$

Calculate acceleration

$$s_q''(t) = -\sum_{j=0}^{n-1} \frac{m_j}{|s_q(t) - s_j(t)|^3} [s_q(t) - s_j(t)]$$

$t = 0, \Delta t, 2\Delta t, \ldots, T\Delta t$
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 {
        // Compute force
        x_diff = pos[q](X) - pos[k](X);
        y_diff = pos[q](Y) - pos[k](Y);
        dist_squared = sqrt(x_diff^2 + y_diff^2);
        forces[q](X) = -mass[q]*mass[k]/dist_squared * x_diff;
        forces[q](Y) = -mass[q]*mass[k]/dist_squared * y_diff;
    }
}

Compute half the pairs

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?

```c
for each particle \(q\) {
  for each particle \(k \neq q\) {
    x_diff = pos[q][x] - pos[k][x];
    y_diff = pos[q][y] - pos[k][y];
    dist = sqrt(x_diff^2 + y_diff^2);
    dist_cubed = dist^3;
    force[q][x] = -masses[q]/masses[k]/dist_cubed * x_diff;
    force[q][y] = -masses[q]/masses[k]/dist_cubed * y_diff;
  }
}
```

Copyright © 2010, Elsevier Inc. All rights Reserved

Computation of the forces: Reduced Algorithm

Reduce calculation of Total Force, capitalizing on \(f_{kq} = -f_{qk}\)
Parallelization? Scheduling?

```c
for each particle \(q\) {
  force[q] = 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^2 + y_diff^2);
    dist_cubed = dist^3;
    force[q][x] = -masses[q]/masses[k]/dist_cubed * x_diff;
    force[q][y] = -masses[q]/masses[k]/dist_cubed * y_diff;
    force[k][x] += force[q][x];
    force[k][y] += force[q][y];
    force[k][x] -= force[q][x];
    force[k][y] -= force[q][y];
  }
}
```

Copyright © 2010, Elsevier Inc. All rights Reserved

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
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.
- $5p$ on process 0 $local\_pos = pos$; on process 1 $local\_pos = pos + loc\_n$, etc.

```
Detour: MPI_Scatter()
```

```

Distribute Data from input using a scatter operation
```

```
10 length_per_process = length/size;
11 myarray = (int *) malloc(length_per_process*sizeof(int));
12
13 array[istart + i] = malloc(length*sizeof(int));
14
15 /* Read the data, distribute it among the various processes */
16 if (myid==0) {
17   (*fopen("poses", "r"))->read(length, &myarray); // read input
18   for (i=0; i<length; i++) // read entire input file
19     for (j=0; j<size; j++) // read entire process
20       (*fopen("poses", "w"))->write(&myarray[i]);
21   free(myarray);
22 }
23      
24 MPI_Scatter(myarray, length_per_process, MPI_INT, 
25   myarray, length_part_process, MPI_INT,  
26   MPI_COMM_WORLD);

```
MPI_Gather is analogous

MPI_Gather: Gathers together values from a group of processes.

```c
int MPI_Gather(void *sendbuf, int sendcnt, MPI_Datatype sendtype,
                void *recvbuf, int recvnt, 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.

```c
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**
- `recvbuf`: address of receive buffer (choice)

---

I/O: Ch. 6, p. 291

```c
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:
        Compute position and velocity of loc_q:
        Allgather local positions into global pos array:
    Print positions and velocities of particles:

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;
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 (int_part1 = 0, glob_part1 = my_rank;
    loc_part1 < loc_n - 1;
    loc_part1++, glob_part1 = comm_sz)
for (glob_part2 = first_index(glob_part1, my_rank, owner, comm_sz);
    loc_part2 = Global_to_local(glob_part2, owner, loc_n);
    loc_part2 < loc_n;
    loc_part2++, glob_part2 = comm_sz)
Compute_forces(loc_part1, masses[glob_part1],
               tmp_pos[loc_part1], masses[glob_part2],
               loc_forces[loc_part1], tmp_forces[loc_part2]);