L19: Putting it together: 
N-body (Ch. 6) 

November 22, 2011

---

Dense and sparse linear algebra

Dense matvec from L15:

```c
for (i=0; i<n; i++) {
    for (j = ptr[i]; j<ptr[i+1]-1; j++)
        t[i] += data[j] * b[indices[j]];
}
```

Equivalent CSR matvec:

```c
for (i=0; i<n; i++) {
    for (j = ptr[i]; j<ptr[i+1]-1; j++)
        t[i] += data[j] * b[indices[j]];
}
```

---

Recall Dense CUDA code (Automatically Generated by our Research Compiler)

```c
__global__ mv_GPU(float* a, float* b, float** c) {
    int bx = blockIdx.x; int tx = threadIdx.x;
    __shared__ float bcpy[32];
    double acpy = a[tx + 32 * bx];
    for (k = 0; k < 32; k++) {
        bcpy[tx] = b[32 * k + tx];
    }
    __syncthreads();
    //this loop is actually fully unrolled
    for (j = 32 * k; j<32 * k + 32; j++) {
        acpy = acpy + c[j][32 * bx + tx] * bcpy[j];
    }
    __syncthreads();
    a[tx + 32 * bx] = acpy;
}
```

---

**Administrative**

- Project sign off due today, about a third of you are done (will accept it tomorrow, otherwise 5% loss on project grade)
- Next homework, CUDA, MPI (Ch. 3) and Apps (Ch. 6)
  - Assignment by Monday, goal is to prepare you for final
  - We'll discuss it in class on Thursday
  - Solutions due on Friday, Dec. 1 (should be straightforward if you are in class)
- Poster dry run on Dec. 6, final presentations on Dec. 8
- Optional final report (4-6 pages) due on Dec. 14 can be used to improve your project grade if you need that
Outline

• Go over programming assignment

• 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?

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., $v_p = \sum g_{mp} v_p / \sum g_{mp}$.
4. Conservation of momentum solved on mesh giving updated mesh velocity and (in principal) position.
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)

$$
F_{pq}(t) = \frac{G m_q m_p}{|s_q(t) - s_p(t)|^2} [s_q(t) - s_p(t)]
$$

**Total force on a particle** (6.2)

$$
F_q(t) = \sum_{k=0}^{n-1} \left\{ \sum_{k \neq q} m_k \frac{m_q}{|s_q(t) - s_k(t)|^3} [s_q(t) - s_k(t)] \right\}
$$

Calculate acceleration

(6.3)

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

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

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

Get input data:
for each timestep {
  if (time step input) 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

```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^2 + y_diff^2);
        dist_cubed = dist^3;
        force[q][X] = -G*maassess[q]*maassess[k]/dist_cubed * x_diff;
        force[q][Y] = -G*maassess[q]*maassess[k]/dist_cubed * y_diff;
    }
}
```

Computation of the forces: Reduced Algorithm

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

Compute half the pairs

```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^2 + y_diff^2);
        dist_cubed = dist^3;
        force[k][X] = G*maassess[q]*maassess[k]/dist_cubed * x_diff;
        force[k][Y] = G*maassess[q]*maassess[k]/dist_cubed * y_diff;
    }
}
```

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 != 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] = G*maassess[q]*maassess[k]/dist_cubed * x_diff;
        force[q][Y] = G*maassess[q]*maassess[k]/dist_cubed * y_diff;
    }
}
```
Computation of the forces: Reduced Algorithm

Reduce calculation of Total Force, capitalizing on $f_{ek} = -f_{ke}$

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($x_{diff}^2 + y_{diff}^2$);
       dist_cubed = dist * dist * dist;
       force_qk[X] = $G$ * mass[k] / dist_cubed * $x_{diff}$;
       force_qk[Y] = $G$ * mass[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 positions.
  - So on process 0 local_pos = pos; on process 1 local_pos = pos + loc_n; etc.
**MPI Parallelization: Basic version**

Get input data:

```c
for each timestep {  
    if (timestep output)  
        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()**

```c
MPI_Scatter();  
int MPI_Scatter(void *sendbuf, int sendcount, MPI_Datatype sendtype,  
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)
- **root**: rank of root process (integer)
- **comm**: communicator (handle)

Output Parameter:
- **recvbuf**: address of receive buffer (choice, significant only at root)

**Distribute Data from input using a scatter operation**

```c
Int length_per_process = length/sizeof(int);  
mypart = (int *) malloc(length * sizeof(int));  
for (i = 0; i < length; i++)  
{  
    if (i%num_procs == root)  
        printf("%d", mypart[i]);  
        fscanf(fp, "%d", &length);  
    else  
        fscanf(fp, "%d", &mypart[i]);  
}  
```

**MPI_Gather is analogous**

```c
int MPI_Gather(void *sendbuf, int sendcount, MPI_Datatype sendtype,  
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)
- **root**: rank of root process (integer)
- **comm**: communicator (handle)

Output Parameter:
- **recvbuf**: data type of recv buffer elements (significant only at root) (handle)
**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)
- `recvbuf`: address of receive buffer (choice)
- `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:
```c
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;
}
```

Copyright © 2010, Elsevier Inc. All rights Reserved
**Communication In A Possible MPI Implementation of the N-Body Solver**
(for a reduced solver)

- Process 0
  - 0, 1
  - Particles
  - Compute forces
  - Update positions of velocities

- Process 1
  - 2, 3

- Process 2
  - 4, 5

---

**Ring Pass of Positions**

Phase 1

1. 0, 3
2. 3, 1
3. 1, 2
4. 2, 0

Phase 2

1. 0, 3
2. 3, 1
3. 1, 2
4. 2, 0

Phase 3

1. 0, 3
2. 3, 1
3. 1, 2
4. 2, 0

---

**Pseudo-code for the MPI implementation of the reduced n-bodysolver**

```plaintext
source = (my_rank + 1) % comm_sz;
dest = ((my_rank + 1 + comm_sz) % comm_sz;
loc_pos = 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;
    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;
```