PART V - The n-body problem

Objectives

- Problem formulation
- Program development OpenMP
- Program development MPI

Problem Formulation

The n-body problem is one of the most famous problems in mathematical physics, with its first complete mathematical formulation dating back to Newton's *Principia*.

Classically, it refers to the problem of predicting the motion of n celestial bodies that interact gravitationally.

Nowadays, other problems, such as those from molecular dynamics, are also often referred to as n-body problems.

For $n = 2$, the problem was completely solved by Johann Bernoulli.

For $n = 3$, solutions exist in special cases.

In general, numerical methods must be used to simulate such systems.

Problem Formulation

We will specialize the discussion in terms of the gravitational problem, so we treat as inputs the mass, position, and velocity of each particle.

The output is typically the positions and velocities of all the particles at some final time or sequence of times.

We assume Newtonian physics to describe the motion.

Suppose we have n particles with masses m_i and positions $\mathbf{r}_i(t)$.

Then particle i has a (gravitational) force exerted on it by particle j given by

$$\mathbf{f}_{i,j}(t) = -\frac{Gm_i m_j}{||\mathbf{r}_i(t) - \mathbf{r}_j(t)||^3}(\mathbf{r}_i(t) - \mathbf{r}_j(t)),$$

where $G = 6.673 \times 10^{-11}$ m/(kg $\cdot$ s^2).

Problem Formulation

The total force on particle i is thus given by

$$\begin{aligned}\mathbf{F}_i(t) &= \sum_{\substack{j=0 \\ j\neq i}}^{n-1} \mathbf{f}_{i,j}(t) \\ &= -Gm_i \sum_{\substack{j=0 \\ j\neq i}}^{n-1} \frac{m_j}{||\mathbf{r}_i(t) - \mathbf{r}_j(t)||^3}(\mathbf{r}_i(t) - \mathbf{r}_j(t)).\end{aligned}$$

Now using Newton's second law $\mathbf{F} = m\,\mathbf{a}$ translated to our notation, we have

$$\mathbf{F}_i(t) = m_i \ddot{\mathbf{r}}_i(t),$$

for $i = 0, 1, \ldots, n-1$.

Problem Formulation

From this, we obtain a system of second-order ordinary differential equations (ODEs)

$$\ddot{\mathbf{r}}_i(t) = -G \sum_{\substack{j=0 \\ j \neq i}}^{n-1} \frac{m_j}{||\mathbf{r}_i(t) - \mathbf{r}_j(t)||^3} (\mathbf{r}_i(t) - \mathbf{r}_j(t)),$$

for $i = 0, 1, \ldots, n-1$.

To make life slightly simpler, we assume $\mathbf{r}_i(t) \in \mathbb{R}^2$.

We also assume that we will integrate the ODEs with an *explicit* numerical method and *constant stepsize* Δt.

(More details on exactly what this means in a moment.)

The output is to be given at a future time $T = N\Delta t$.

Algorithm Formulation

A serial n-body solver consists of the following steps:

Require: input data

1: **for** each (constant) timestep **do**
2: **for** each particle i **do**
3: Compute $\mathbf{F}_i(t)$.
4: Update $\mathbf{r}_i(t)$ (and $\dot{\mathbf{r}}_i(t) := \mathbf{v}_i(t)$).
5: **end for**
6: **end for**
7: **return** $\mathbf{r}_i(T)$ for each particle i.

Pseudocode to compute the $\mathbf{F}_i(t)$ might look like

1: **for** each particle $j \neq i$ **do**
2: `dx = r[i][x] - r[j][x];`
3: `dy = r[i][y] - r[j][y];`
4: `d = sqrt(dx*dx+dy*dy);`
5: `d3 = d*d*d;`
6: `F[i][x] -= G*m[i]*m[j]/d3*(r[i][x]-r[j][x]);`
7: `F[i][y] -= G*m[i]*m[j]/d3*(r[i][y]-r[j][y]);`
8: **end for**

Algorithm Formulation

Of course, there is a symmetry that we can exploit.

By Newton's third law, $\mathbf{f}_{i,j}(t) = -\mathbf{f}_{j,i}(t)$.

We can reduce the number of force calculations by half if we are careful about the sign of each force.

We can see how to do this from the *force matrix*:

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

Now we can perform a (truncated) nested loop over the upper (lower) triangle of the force matrix and update the contribution of $\mathbf{f}_{i,j}$ to *both* particles i and j.

Algorithm Formulation

1: **for** each particle i **do**
2: $\quad \mathbf{F}_i(t) = \mathbf{0}$;
3: **end for**
4: **for** each particle i **do**
5: $\quad$ **for** each particle $j > i$ **do**
6: $\quad\quad \mathbf{F}_i(t)$ **+=** $\mathbf{f}_{i,j}(t)$;
7: $\quad\quad \mathbf{F}_j(t)$ **-=** $\mathbf{f}_{i,j}(t)$;
8: $\quad$ **end for**
9: **end for**

We must not forget of course that

$$\mathbf{a}_i(t) = \ddot{\mathbf{r}}_i(t) = \mathbf{F}_i(t)/m_i.$$

Ultimately, we integrate the ODEs for the $\mathbf{r}_i(t)$ to simulate their evolution over time.

Algorithm Formulation

We are given the *initial conditions*

$$\mathbf{r}_i(0) = \mathbf{r}_0, \quad \dot{\mathbf{r}}_i(0) = \mathbf{v}_0, \quad i = 0, 1, \ldots, n-1.$$

Thus we have a so-called *initial-value problem*.

Another type of problem, known as a *boundary-value problem*, might have conditions that look like

$$\mathbf{r}_i(0) = \mathbf{r}_0, \quad \mathbf{r}_i(T) = \mathbf{r}_T, \quad i = 0, 1, \ldots, n-1.$$

Boundary-value problems are sufficiently more general than initial-value problems that the numerical algorithms for their solution are distinctly different.

The question we now face is how do we advance the $\mathbf{r}_i(t)$ from $t = 0$ to $t = T$.

There are many different strategies to do so; we focus on a simple one: Euler's method (or *forward Euler*).

Algorithm Formulation

Euler's method is designed to solve IVPs for systems of *first-order* ODEs

$$\dot{\mathbf{y}}(t) = \mathbf{f}(t, \mathbf{y}), \quad \mathbf{y}(0) = \mathbf{y}_0, \quad t > 0.$$

Geometrically, the idea behind the method is to "freeze" the vector field at the point $(t_k, \mathbf{y}_k)$, where the solution is, and have the system follow this (constant) vector field for length Δt.

This leads to the algorithm

$$\mathbf{y}_k = \mathbf{y}_{k-1} + \Delta t\, \mathbf{f}(t_{k-1}, \mathbf{y}_{k-1}), \quad k = 1, 2, \ldots, N,$$

where $\mathbf{y}_k \approx \mathbf{y}(t_k)$.

Alternatively, it can be interpreted as using the zeroth-order Taylor-series approximation of the vector field $\mathbf{f}(t, \mathbf{y})$ at the point $(t_k, \mathbf{y}_k)$.

Algorithm Formulation

Often mathematical models involve second- or higher-order derivatives.

Any model based on Newton's second law $\mathbf{F} = m\mathbf{a}$ corresponds to $\ddot{\mathbf{r}}(t) = \mathbf{F}/m$.

As mentioned, the n-body problem that we are considering is an example of this.

These problems are not of the form $\dot{\mathbf{y}} = \mathbf{f}(t, \mathbf{y})$, so it is not possible to directly apply a standard integrator such as Euler's method for their solution.

Fortunately, they can be converted to first-order form using a standard change of variables.

In such cases, the unknowns are the function and its derivatives up to one order less than the given ODE.

This means that an mth-order ODE is converted to m first-order ODEs.

Algorithm Formulation

For example, consider the second-order differential equation describing a simple harmonic oscillator:

$$\ddot{x}(t) = -x(t).$$

In this case, we create the vector of unknown functions to be

$$y_1(t) = x(t), \qquad y_2(t) = \dot{x}(t).$$

Then the second-order ODE is converted into two first-order ODEs:

$$\dot{\mathbf{y}}(t) = \begin{bmatrix} \dot{y}_1(t) \\ \dot{y}_2(t) \end{bmatrix} = \begin{bmatrix} y_2(t) \\ -y_1(t) \end{bmatrix}.$$

Algorithm Formulation

Similarly, the n-body problem can be written as the system of $2n$ first-order ODEs

$$\begin{aligned}\dot{\mathbf{r}}_i(t) &= \mathbf{v}_i(t),\\ \dot{\mathbf{v}}_i(t) &= \mathbf{F}_i(t)/m_i, \quad i = 0, 1, 2, \ldots, n-1,\end{aligned}$$

subject to the initial conditions

$$\mathbf{r}_i(0) = \mathbf{r}_{i,0}, \quad \mathbf{v}_i(0) = \mathbf{v}_{i,0}, \quad i = 0, 1, 2, \ldots, n-1.$$

Thus the positions (and velocities) may be updated via

```
1: r[i][x] += dt*v[i][x];
2: r[i][y] += dt*v[i][y];
3: v[i][x] += dt*f[i][x]/m[i];
4: v[i][y] += dt*f[i][y]/m[i];
```

Data Structures

Before moving on to parallelization, we review the assumptions about the data structures involved.

We have been using arrays to store vectors.

A good way to define their size is via a macro `DIM`.

Then if we wish to change dimensions, in principle we would only change the definition of `DIM`.

For example,

```
#define DIM 2

typedef double vect_t[DIM];
```

Data Structures

To advance the solution at each time step, we need the values of each particle's

- mass,
- position,
- velocity,
- total force acting on it, and
- acceleration.

To advance the system, it suffices to store the masses and current positions, velocities, and forces.

Note that the predominant nature of numerical methods for initial-value problems is to only store a limited history of the solution.

Data Structures

For the purposes of this example, we group the mass, position, and velocity of each particle into a single struct and store the forces in a separate array.

Because the forces are thus stored in a contiguous block of memory, we can initialize them (to 0) quickly using `memset`.

```
#include <string.h> /* for memset */
...
vect_t* f = malloc(n*sizeof(vect_t));
...
for (step = 1; step <= n_steps; step++) {
   ...
   /* Assign 0 to each element of f */
   f = memset(f,0,n*sizeof(vect_t));
   for (i = 0; i < n-1; i++
      Compute_force(i,f,...);
   ...
}
```

If the total force on each particle were a member of a struct, the force members would not be contiguous in memory, and we would have to use a relatively slow `for` loop to zero the entries.

Applying Foster's methodology

1. *Partitioning.* Compute the $\mathbf{F}_i(t_{k-1})$, $\mathbf{v}_i(t_k)$, and $\mathbf{r}_i(t_k)$ for each particle i and time step k.

2. *Communication.* Computation of $\mathbf{v}_i(t_k)$ requires $\mathbf{v}_i(t_{k-1})$ and $\mathbf{F}_i(t_{k-1})$ (and hence $\mathbf{r}_i(t_{k-1})$ and $\mathbf{r}_j(t_{k-1})$); computation of $\mathbf{r}_i(t_k)$ requires $\mathbf{r}_i(t_{k-1})$ and $\mathbf{v}_i(t_{k-1})$.

3. *Aggregation.* Most communication between tasks can be organized on a per-particle basis, so it makes sense to agglomerate computations associated with a single particle into a composite task.

4. *Mapping.* There are nN tasks with both n, $N \gg p$, the number of processes.

   So, there are two "dimensions" to this mapping.

   However, assigning tasks for one particle at different time steps to different cores will not work well because of the sequential nature of Euler's method.

Applying Foster's methodology

Hence, the map of tasks to cores reduces to mapping particles to cores.

Assuming the work done per step is roughly equal, a block partitioning that assigns roughly n/p particles per core should provide a well-balanced load.

This assumption is valid for the case where symmetry is *not* taken advantage of when computing $\mathbf{f}_{i,j}(t)$.

When symmetry is taken advantage of, the loops for lower i will more expensive than those for larger i.

In this case, a cyclic partition is more effective.

However, in a shared-memory framework, a cyclic partition is almost certain to lead to a higher number of cache misses than a block partition.

In a distributed-memory framework, the communication overhead involved with a cyclic partition will probably be greater than that for a block partition.

Applying Foster's methodology

This leads us to the following summary of how best to map tasks to cores in a basic implementation of the n-body problem.

- A block distribution should give the best performance for the algorithm that does not take advantage of symmetry in the force calculations.

- In principle, a cyclic distribution should give the best performance for the algorithm that does take advantage of symmetry in the force calculations. However, additional cache misses in the shared-memory context and additional communication overhead in the distributed-memory context may render a block distribution superior in practice.

In order to make the final decision as to the optimal mapping of tasks to cores, we will need to appeal to some experimentation.

Parallelization with OpenMP

In order to see how to write a parallel n-body solver using OpenMP, we first recall the basic structure:

1: **for** each (constant) timestep **do**
2: **for** each particle i **do**
3: Compute $\mathbf{F}_i(t)$.
4: **end for**
5: **for** each particle i **do**
6: Update $\mathbf{r}_i(t)$ (and $\dot{\mathbf{r}}_i(t) := \mathbf{v}_i(t)$).
7: **end for**
8: **end for**

The two inner loops are both iterating over particles. So, it might suffice to put in a `parallel for` construct:

1: `# pragma omp parallel for`
2: **for** each particle i **do**
3: Compute $\mathbf{F}_i(t)$.
4: **end for**
5: `# pragma omp parallel for`
6: **for** each particle i **do**
7: Update $\mathbf{r}_i(t)$ (and $\dot{\mathbf{r}}_i(t) := \mathbf{v}_i(t)$).
8: **end for**

Parallelization with OpenMP

We might not like all the forking and joining that such a program might do, but before we worry about that, we should ensure the program actually works.

In particular, we need to check for race conditions resulting from loop-carried dependencies.

Recall the inner workings of the first loop:

1: **for** each particle $j \neq i$ **do**
2: `dx = r[i][x] - r[j][x];`
3: `dy = r[i][y] - r[j][y];`
4: `d = sqrt(dx*dx+dy*dy);`
5: `d3 = d*d*d;`
6: `F[i][x] -= G*m[i]*m[j]/d3*(r[i][x]-r[j][x]);`
7: `F[i][y] -= G*m[i]*m[j]/d3*(r[i][y]-r[j][y]);`
8: **end for**

Parallelization with OpenMP

The `parallel for` construct ensures only one thread will access `F[i]` for any `i`.

It is true that thread `i` will access `m[j]` and `r[j]`, but these values are only *read*.

As long as they are not *written* to, there is no risk of a race condition being established.

All other variables are temporary in nature, so they can be private.

So, this first loop does not introduce race conditions.

Parallelization with OpenMP

Recall the inner workings of the second loop:

```
1: r[i][x] += dt*v[i][x];
2: r[i][y] += dt*v[i][y];
3: v[i][x] += dt*f[i][x]/m[i];
4: v[i][y] += dt*f[i][y]/m[i];
```

In this case, it is straightforward to see that thread `i` only accesses quantities associated with particle `i`.

The only other variable referenced is the scalar `dt` and that is only read.

So this second loop does not introduce race conditions.

Recall that there is an implicit barrier at the end of each `parallel for` construct, so no race conditions are set up when going from one loop to the next.

Finally, we may wish to add a `schedule` clause to the `for` directives to ensure a block partition is used:

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

Parallelization with OpenMP

Things are a little trickier when trying to take advantage of symmetry in the force calculations.

Recall the algorithm:

1: **for** each particle i **do**
2: $\quad \mathbf{F}_i(t) = \mathbf{0}$;
3: **end for**
4: **for** each particle i **do**
5: $\quad$ **for** each particle $j > i$ **do**
6: $\quad\quad \mathbf{F}_i(t)$ += $\mathbf{f}_{i,j}(t)$;
7: $\quad\quad \mathbf{F}_j(t)$ -= $\mathbf{f}_{i,j}(t)$;
8: $\quad$ **end for**
9: **end for**

The additional loop to initialize the forces to 0 can be easily parallelized using a `parallel for` construct.

All iterations are independent, and so there is no risk of introducing any race conditions.

Parallelization with OpenMP

However, a simple insertion of `parallel for` constructs does introduce a race condition for the $\mathbf{f}_{i,j}$ computations.

Suppose we have 4 particles and 2 threads with a block partition of the particles.

The total force on say particle 3 is

$$\mathbf{F}_3 = -\mathbf{f}_{0,3} - \mathbf{f}_{1,3} - \mathbf{f}_{2,3}.$$

Thread 0 computes $\mathbf{f}_{0,3}$ and $\mathbf{f}_{1,3}$ and Thread 1 computes $\mathbf{f}_{2,3}$.

Thus, the threads are racing to update the forces.

Parallelization with OpenMP

The knee-jerk solution to resolve race conditions is via `critical` directives.

For example, a `critical` directive could be put before all the force updates:

1: **for** each particle i **do**
2: **for** each particle $j > i$ **do**
3: `# pragma omp critical` $\{$
4: $\mathbf{F}_i(t)$ += $\mathbf{f}_{i,j}(t)$;
5: $\mathbf{F}_j(t)$ -= $\mathbf{f}_{i,j}(t)$;
6: $\}$
7: **end for**
8: **end for**

Of course, this effectively serializes the code, and the performance of the code is likely to degrade accordingly.

Parallelization with OpenMP

An alternative would be to have one critical section for each particle by using a lock for each particle:

1: **for** each particle i **do**
2: **for** each particle $j > i$ **do**
3: `omp_set_lock(& locks[i])`
4: $\mathbf{F}_i(t)$ += $\mathbf{f}_{i,j}(t)$.
5: `omp_unset_lock(& locks[i])`
6: `omp_set_lock(& locks[j])`
7: $\mathbf{F}_j(t)$ -= $\mathbf{f}_{i,j}(t)$.
8: `omp_unset_lock(& locks[j])`
9: **end for**
10: **end for**

The master thread creates a shared array of locks (one for each particle), and the locks are set according to which particle is going to have its force updated.

This approach does perform much better than having a single critical section, but it is still not competitive with the serial code.

Parallelization with OpenMP

A more effective way to proceed is to separate the computation of the forces into two phases and use local temporary storage.

Specifically, phase one is the previous computation (with the race condition) but using local storage.

The second phase then gathers the local values and puts them into the shared vector for the forces.

Thus, for 2 threads, 4 particles, and block partitioning, Thread 0 computes and locally stores $-\mathbf{f}_{0,3} - \mathbf{f}_{1,3}$, and Thread 1 computes and locally stores $-\mathbf{f}_{2,3}$.

Then Thread 1, which has been assigned particle 3, computes $\mathbf{F}_3$ by adding these two values.

Parallelization with OpenMP

To understand potential impacts of scheduling, we now consider a larger example: 3 threads and 6 particles.

With a block partition, the computations are as follows:

i / p	0	1	2
0	$\mathbf{f}_{0,1} + \mathbf{f}_{0,2} + \mathbf{f}_{0,3} + \mathbf{f}_{0,4} + \mathbf{f}_{0,5}$	—	—
1	$-\mathbf{f}_{0,1} + \mathbf{f}_{1,2} + \mathbf{f}_{1,3} + \mathbf{f}_{1,4} + \mathbf{f}_{1,5}$	—	—
2	$-\mathbf{f}_{0,2} - \mathbf{f}_{1,2}$	$\mathbf{f}_{2,3} + \mathbf{f}_{2,4} + \mathbf{f}_{2,5}$	—
3	$-\mathbf{f}_{0,3} - \mathbf{f}_{1,3}$	$-\mathbf{f}_{2,3} + \mathbf{f}_{2,4} + \mathbf{f}_{2,5}$	—
4	$-\mathbf{f}_{0,4} - \mathbf{f}_{1,4}$	$-\mathbf{f}_{2,4} - \mathbf{f}_{3,4}$	$\mathbf{f}_{4,5}$
5	$-\mathbf{f}_{0,5} - \mathbf{f}_{1,5}$	$-\mathbf{f}_{2,5} - \mathbf{f}_{3,5}$	$-\mathbf{f}_{4,5}$

Then in phase 2, Thread 0 collects the contributions for particles 0 and 1, Thread 1 collects the contributions for particles 1 and 2, etc.

However, we notice the load is not well balanced.

The load is equivalent to the number of contributions of one sign of $\mathbf{f}_{i,j}$.

So Thread 0 has 9 units of work, Thread 1 has 5 units of work, and Thread 2 only has 1 unit of work.

If we use a cyclic partition, the computations are:

i / p	0	1	2
0	$\mathbf{f}_{0,1} + \mathbf{f}_{0,2} + \mathbf{f}_{0,3} + \mathbf{f}_{0,4} + \mathbf{f}_{0,5}$	—	—
1	$-\mathbf{f}_{0,1}$	$\mathbf{f}_{1,2} + \mathbf{f}_{1,3} + \mathbf{f}_{1,4} + \mathbf{f}_{1,5}$	—
2	$-\mathbf{f}_{0,2}$	$-\mathbf{f}_{1,2}$	$\mathbf{f}_{2,3} + \mathbf{f}_{2,4} + \mathbf{f}_{2,5}$
3	$-\mathbf{f}_{0,3} + \mathbf{f}_{3,4} + \mathbf{f}_{3,5}$	$-\mathbf{f}_{1,3}$	$-\mathbf{f}_{2,3}$
4	$-\mathbf{f}_{0,4} - \mathbf{f}_{3,4}$	$-\mathbf{f}_{1,4} + \mathbf{f}_{4,5}$	$-\mathbf{f}_{2,4}$
5	$-\mathbf{f}_{0,5} - \mathbf{f}_{3,5}$	$-\mathbf{f}_{1,5} - \mathbf{f}_{4,5}$	$-\mathbf{f}_{2,5}$

Now, Thread 0 has 7 units of work, Thread 1 has 5 units of work, and Thread 2 only 3 units of work.

Not perfect, but better, and the improvements increase as the problem size and the number of threads increase.

Parallelization with OpenMP

The code for phase 1 would look something like

```
1: # pragma omp for
2: for each particle i do
3:    F_i(t) = 0.
4:    for each particle j > i do
5:       F_loc,i(t) += f_i,j(t);
6:       F_loc,j(t) -= f_i,j(t);
7:    end for
8: end for
```

The code for phase 2 would look something like

```
1: # pragma omp for
2: for each particle i do
3:    for each thread p do
4:       F_i(t) += F_loc,p(t);
5:    end for
6: end for
```

Parallelization with OpenMP

Finally, of course, we should double-check to ensure this new approach does not introduce any race conditions.

In phase 1, each thread writes to its own local array, so no race condition is created there.

Similarly, during phase 2, only the thread that owns particle i writes to $\mathbf{F}_i(t)$, so there is no race condition created there either.

Finally, we check to ensure no race condition is created in the transition between the loops.

Again, the implicit barrier at the end of each `parallel for` construct prevents threads from racing ahead and using variables that have not been properly initialized.

Parallelization with OpenMP

We now quote some results from the text as to the performance of the OpenMP code.

We would like to compare the performance of the codes depending on whether symmetry is taken advantage of.

When symmetry is not taken advantage of, we have seen that any schedule that divides the iterations equally among the threads (assuming one thread per core) should do a good job of load balancing.

We also argued a block partition would result in fewer cache misses than a cyclic partition.

Thus, the block partition is likely to be the best option for this implementation.

Parallelization with OpenMP

When taking advantage of symmetry, we recall that the amount of work done in phase 1 decreases as the `for` loop proceeds.

A cyclic schedule should do the best job of load balancing in this case.

However, in the remainder of the code (initializing $\mathbf{F}_{\text{loc}}(t)$, phase 2, and updating $\mathbf{r}(t)$ and $\mathbf{v}(t)$), the work per iteration is roughly equal.

Unfortunately, the schedule of one loop can affect the performance of another, so it may be that choosing a cyclic partition for one loop and block partitions for the others may degrade the performance.

Parallelization with OpenMP

We consider the performance of the n-body solver when run with no I/O, $n = 400$ particles, and $N = 1000$ time steps; times are given in seconds.

Differences between serial code and codes with $p = 1$ are less than 1%.

p	ns/block	s/block	s/$\mathbf{F}$ cyclic	s/all cyclic
1	7.71	3.90	3.90	3.90
2	3.87	2.94	1.98	2.01
4	1.95	1.73	1.01	1.08
8	0.99	0.95	0.54	0.61

Parallelization with OpenMP

Observations:

- For $p > 1$, cyclic outperforms block partitioning; i.e., cache misses are more than made up for by improved load balancing.

- For $p = 2$, there is little difference in performance between the cyclic partitions, but applying a cyclic partition only to the force calculations outperforms as p increases; i.e., the overhead in changing partitions is less than that from false sharing.

- Not taking advantage of symmetry leads to execution times of about twice those where a cyclic partition is used for the force calculations. However, the latter implementation requires more p times more memory. So *if enough memory is available*, the partially cyclic solver is the method of choice.

Parallelization with MPI

With composite tasks grouped according to particles, it is fairly straightforward to parallelize the version that does not take advantage of symmetry with MPI.

The "only" communication between processes occurs when computing the $\mathbf{f}_{i,j}$.

Unfortunately, each particle needs the position of every other particle (and at every time step).

Fortunately, the `MPI_Allgather` function is designed for exactly this type of communication.

We expect that a block partition will have the best performance, so the mapping of particles to processes will be done this way.

Parallelization with MPI

In the OpenMP version of the program, we used a single struct to collect most of the data for a given particle (m_i, $\mathbf{r}_i$, and $\mathbf{v}_i$ but not $\mathbf{F}_i$).

The equivalent in MPI would require a derived data type, and communication with derived data types is generally slower than with native MPI data types.

So, we will just use arrays for these quantities.

In fact, if there is enough memory, each process can store its own array for the m_i because they are always needed but never changed.

Parallelization with MPI

We make the following design decisions regarding the data structures to be used.

Suppose that the array `pos` stores the positions of all n particles.

Further, suppose that `vect_mpi_t` is an MPI data type that stores two contiguous `doubles`.

Finally, suppose that n is evenly divisible by p and define `loc_n` $= n/p$.

- Each process stores all the m_i.
- Each process uses a single array for the $\mathbf{r}_i(t)$.
- Each process uses a pointer `loc_pos` to the start of its block of $\mathbf{r}$.

  So on process p, `loc_pos = pos +` p`loc_n`.

Parallelization with MPI

With these choices, the MPI code to perform an n-body simulation without using symmetry might look something like

1: Get (and broadcast) input data.
2: **for** each timestep **do**
3: **for** each local particle i_{loc} **do**
4: Compute $\mathbf{F}_{i_{\text{loc}}}(t)$.
5: **end for**
6: **for** each local particle i_{loc} **do**
7: Compute $\mathbf{r}_{i_{\text{loc}}}$ and $\mathbf{v}_{i_{\text{loc}}}$.
8: **end for**
9: `Allgather` $\mathbf{r}_{i_{\text{loc}}}$ to global array `pos`.
10: **end for**

Note that the m_i and `pos` are broadcast because they are needed everywhere to compute the forces.

On the other hand, the velocities are only ever needed locally, so the array `vel` is scattered.

Parallelization with MPI

Trying to take advantage of symmetry in the force calculations looks at first to be extremely complicated.

In order to compute the forces, each process would have to gather a subset of the positions.

Then it would have to scatter some of these forces and receive others in order to update the forces on the particles of which it is in charge.

As usual, complicated strategies can work well but only if great care is exercised.

In such situations, it is one challenge just to get the correct answer.

It is quite another to get the correct answer in an efficient (scalable) manner.

Parallelization with MPI

Fortunately, we can take advantage of symmetry in our problem using a well-known communication construct called the *ring pass*.

As the name implies, in a ring pass, we imagine the processes to be connected in a ring.

Process p communicates only with processes $p \pm 1$.

Communication takes place in phases, with, e.g., process p sending data to process $p-1$ and receiving data from process $p+1$.

In general, process p sends data to process $(p-1+$ `comm_sz`$)$ % `comm_sz` and receives data from process $(p+1)$% `comm_sz`.

After `comm_sz`-1 phases, all the processes have received all the data.

This will be our plan for communicating the positions.

Parallelization with MPI

We still need to figure out how to exploit symmetry in the force calculation, i.e., the fact that $\mathbf{f}_{i,j} = -\mathbf{f}_{j,i}$.

The first thing to notice is that some interparticle forces are added whereas others are subtracted.

In the ring pass computation of the interparticle forces, the forces computed by the owning task/particle are added, whereas those computed by other tasks/particles are subtracted.

For example, with 6 particles,

$$\mathbf{F}_3 = -\mathbf{f}_{0,3} - \mathbf{f}_{1,3} - \mathbf{f}_{2,3} + \mathbf{f}_{3,4} + \mathbf{f}_{3,5}$$

and the first sub-index indicates which task/particle carries out the computation.

Parallelization with MPI

So in our ring pass, we can pass `loc_n` forces along with the `loc_n` positions in each phase.

Thus in each phase, each process can

1. compute the $\mathbf{f}_{i,j}$ between the particles it owns and those whose positions it receives;

2. add the $\mathbf{f}_{i,j}$ to the particles it owns *and* subtract them from the force array it receives.

Let's see what this looks like for $n = 4$, $p = 2$, and a cyclic distribution of particles among the processes.

Table 6.4 Computation of Forces in Ring Pass

Time	Variable	Process 0	Process 1
Start	loc_pos	$\mathbf{s}_0, \mathbf{s}_2$	$\mathbf{s}_1, \mathbf{s}_3$
	loc_forces	0,0	0,0
	tmp_pos	$\mathbf{s}_0, \mathbf{s}_2$	$\mathbf{s}_1, \mathbf{s}_3$
	tmp_forces	0,0	0,0
After Comp of Forces	loc_pos	$\mathbf{s}_0, \mathbf{s}_2$	$\mathbf{s}_1, \mathbf{s}_3$
	loc_forces	$\mathbf{f}_{02}, 0$	$\mathbf{f}_{13}, 0$
	tmp_pos	$\mathbf{s}_0, \mathbf{s}_2$	$\mathbf{s}_1, \mathbf{s}_3$
	tmp_forces	$0, -\mathbf{f}_{02}$	$0, -\mathbf{f}_{13}$
After First Comm	loc_pos	$\mathbf{s}_0, \mathbf{s}_2$	$\mathbf{s}_1, \mathbf{s}_3$
	loc_forces	$\mathbf{f}_{02}, 0$	$\mathbf{f}_{13}, 0$
	tmp_pos	$\mathbf{s}_1, \mathbf{s}_3$	$\mathbf{s}_0, \mathbf{s}_2$
	tmp_forces	$0, -\mathbf{f}_{13}$	$0, -\mathbf{f}_{02}$
After Comp of Forces	loc_pos	$\mathbf{s}_0, \mathbf{s}_2$	$\mathbf{s}_1, \mathbf{s}_3$
	loc_forces	$\mathbf{f}_{01}+\mathbf{f}_{02}+\mathbf{f}_{03}, \mathbf{f}_{23}$	$\mathbf{f}_{12}+\mathbf{f}_{13}, 0$
	tmp_pos	$\mathbf{s}_1, \mathbf{s}_3$	$\mathbf{s}_0, \mathbf{s}_2$
	tmp_forces	$-\mathbf{f}_{01}, -\mathbf{f}_{03}-\mathbf{f}_{13}-\mathbf{f}_{23}$	$0, -\mathbf{f}_{02}-\mathbf{f}_{12}$
After Second Comm	loc_pos	$\mathbf{s}_0, \mathbf{s}_2$	$\mathbf{s}_1, \mathbf{s}_3$
	loc_forces	$\mathbf{f}_{01}+\mathbf{f}_{02}+\mathbf{f}_{03}, \mathbf{f}_{23}$	$\mathbf{f}_{12}+\mathbf{f}_{13}, 0$
	tmp_pos	$\mathbf{s}_0, \mathbf{s}_2$	$\mathbf{s}_1, \mathbf{s}_3$
	tmp_forces	$0, -\mathbf{f}_{02}-\mathbf{f}_{12}$	$-\mathbf{f}_{01}, -\mathbf{f}_{03}-\mathbf{f}_{13}-\mathbf{f}_{23}$
After Comp of Forces	loc_pos	$\mathbf{s}_0, \mathbf{s}_2$	$\mathbf{s}_1, \mathbf{s}_3$
	loc_forces	$\mathbf{f}_{01}+\mathbf{f}_{02}+\mathbf{f}_{03}, -\mathbf{f}_{02}-\mathbf{f}_{12}+\mathbf{f}_{23}$	$-\mathbf{f}_{01}+\mathbf{f}_{12}+\mathbf{f}_{13}, -\mathbf{f}_{03}-\mathbf{f}_{13}-\mathbf{f}_{23}$
	tmp_pos	$\mathbf{s}_0, \mathbf{s}_2$	$\mathbf{s}_1, \mathbf{s}_3$
	tmp_forces	$0, -\mathbf{f}_{02}-\mathbf{f}_{12}$	$-\mathbf{f}_{01}, -\mathbf{f}_{03}-\mathbf{f}_{13}-\mathbf{f}_{23}$

Parallelization with MPI

Before the ring pass begins, the arrays storing the positions are initialized and forces set to 0.

Each process also computes the $\mathbf{f}_{i,j}$ associated with the particles it owns.

These values are added into the appropriate locations in `loc_forces` and subtracted in `tmp_forces`.

Then `tmp_pos` and `tmp_forces` are exchanged.

The $\mathbf{f}_{i,j}$ between the local and received particles are computed; the lower-ranked process does the computation.

As before, the newly computed forces are added into the appropriate locations in `loc_forces` and subtracted in `tmp_forces`.

The algorithm completes with a final exchange of `tmp_forces` and an update

`loc_forces += tmp_forces.`

Parallelization with MPI

A few final notes on implementation:

- Because we can use the same memory for outgoing and incoming data, we can (should) use `MPI_Sendrecv_replace`.

- A single array to store `tmp_pos` and `tmp_forces` would reduce communication overhead.

- The use of *global indices*, as opposed to local ones as might be tempting, makes life much easier. For example, it is clear that the m_i are most easily accessed with global indices. Global indices simplify the determination of whether a given process should compute the $\mathbf{f}_{i,j}$ for a particle whose position it has just received. A function `Global_to_local` can convert global indices to local ones where this is logistically advantageous.

Performance of the MPI solvers

We quote results from the text on the performance of the MPI solvers that may or may not take advantage of symmetry in the force calculation.

The results (in s) are for 800 particles run for 1000 time steps run on a cluster with an Infiniband interconnect, i.e., a tightly coupled system (like `plato`, but not like `socrates`).

Differences between serial code and codes with $p = 1$ are less than 1%.

p	no symm.	symm.
1	17.30	8.68
2	8.65	4.45
4	4.35	2.30
8	2.20	1.26
16	1.13	0.78

Performance of the MPI solvers

Observations:

- Exploiting symmetry improves performance.

- Exploiting symmetry results in lower efficiencies; e.g., for $p = 16$ the efficiency with exploiting symmetry is about 0.70 versus 0.95 without.

  Efficiency is not always a good measure of performance!

- Exploiting symmetry results in much more efficient memory use; more precisely $p/2$ times less storage is required.

  When n and p are large, this can easily make the difference between being able to run the program within a node's main memory and not.

Comparison of the OpenMP and MPI solvers

In terms of execution time, we quote results from the text from running the OpenMP and MPI programs for the n-body problem on a quad-core node.

The results (in s) again are (presumably) for 800 particles run for 1000 time steps.

	OpenMP		MPI	
p	no symm.	symm.	no symm.	symm.
1	15.13	8.77	17.30	8.68
2	7.62	4.42	8.65	4.45
4	3.85	2.26	4.35	2.30

Comparison of the OpenMP and MPI solvers

Observations:

- When symmetry is not exploited, the OpenMP program outperforms the MPI program. This is not surprising because `MPI_Allgather` is expensive.

- The MPI solver that takes advantage of symmetry is competitive with its OpenMP counterpart. This should not be surprising given the commentary that we have made throughout this course that MPI can be made to outperform OpenMP even in shared-memory environments.

- The execution times of MPI solvers are *identical* whether run in a shared-memory environment or a *tightly coupled* distributed-memory environment.

Comparison of the OpenMP and MPI solvers

In terms of memory, each solver allocates the same amount of storage for the $\mathbf{r}_{\text{loc},i}$ and the $\mathbf{v}_{\text{loc},i}$.

The OpenMP solver allocates $2pn + 2n$ `doubles` for the forces and n `doubles` for the m_i, so in addition to the $\mathbf{r}_{\text{loc},i}$ and the $\mathbf{v}_{\text{loc},i}$, it stores

$$3\frac{n}{p} + 2n \quad \texttt{doubles per thread.}$$

The MPI solver allocates n `doubles` per process for the m_i and $4n/p$ `doubles` for the `tmp_pos` and `tmp_forces` arrays.

So in addition to the $\mathbf{r}_{\text{loc},i}$ and the $\mathbf{v}_{\text{loc},i}$, the MPI solver stores

$$n + 4\frac{n}{p} \quad \texttt{doubles per process.}$$

Comparison of the OpenMP and MPI solvers

The difference in storage is thus

$$n - \frac{n}{p} \quad \text{more } \texttt{doubles} \text{ for the OpenMP solver.}$$

In other words, if n is large, the local storage required for the MPI solver is substantially less than for the OpenMP solver.

So for fixed p, we should be able to run much larger simulations with the MPI solver.

We are also likely to be able to make p much larger in an MPI setting than an OpenMP setting.

So, the size of the largest possible MPI simulation may well be *much* larger than with OpenMP.

n-body solvers: final word

Although we have described a fairly intuitive and reasonably efficient method for solving the n-body problem, the method of choice today is easily the *fast multipole method* (FMM).

The FMM was introduced by Greengard and Rokhlin in 1987.

It was named as one of the top 10 algorithms of the 20th century by the Society for Industrial and Applied Mathematics (others on this list include the Monte Carlo method, the QR algorithm, the fast Fourier transform, and the Quicksort algorithm).

The FMM reduces the intuitive $\mathcal{O}(n^2)$ complexity of the standard n-body algorithm to $\mathcal{O}(n)$ in a fully parallelizable way and with rigorous error estimates.

n-body solvers: final word

The basic idea is to use multipole expansions (net mass, dipole moment, quadrupole moment, etc. — think Green's functions) to approximate the effects of distant groups of particles on local groups.

Space is basically divided into a hierarchy of "panels" or clusters of sources.

Expansions are used to capture the effect of far-away particles while the standard $\mathcal{O}(n^2)$ algorithm is used for particles deemed "close".

There are a number of non-trivial concepts that would go into a general-purpose code, but it has been done, and in practice, if you needed to solve a serious n-body problem, the FMM would be the way to go.

Summary

- Problem formulation
- Program development OpenMP
- Program development MPI