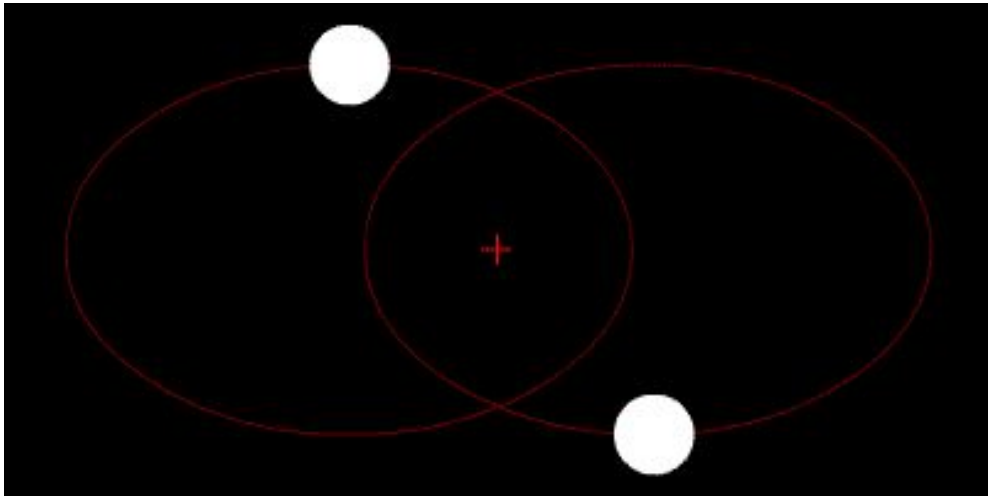
The N-Body Problem Optimized with MPI

John Sabo

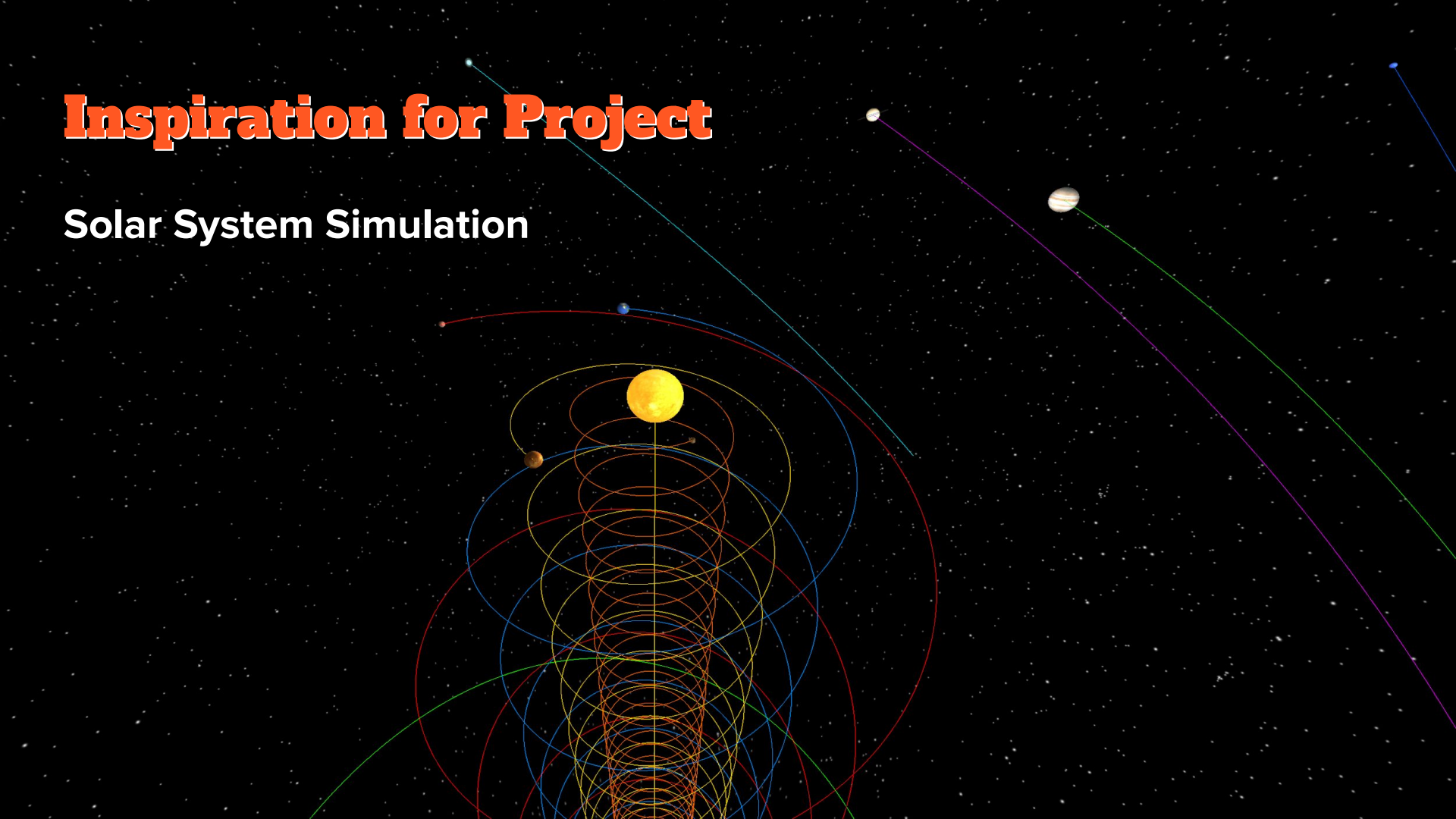
What is the N-Body problem?

- “In physics, the ***n*-body problem** is the problem of predicting the individual motions of a group of celestial objects interacting with each other gravitationally.” [1]



Inspiration for Project

Solar System Simulation



Objective

- Implement main calculation in C
- Optimize program with OpenMPI
- Verify identical output
- Test performance

Serial Algorithm (Javascript)

- Runtime : O(n^2)
- Each body takes into account every other body's position and velocity in order to determine its next position and velocity.

```
for(var planetA of planets)
    for(var planetB of planets)
        if(planetA != planetB)
            var denominator = Math.pow(
                        Math.pow(planetA.x - planetB.x, 2)
                      + Math.pow(planetA.y - planetB.y, 2)
                      + Math.pow(planetA.z - planetB.z, 2), 3/2.0)

            nextVX += (G * planetB.mass * (planetA.x - planetB.x)
                      * parameters.timeStep) / denominator
            nextVY += (G * planetB.mass * (planetA.y - planetB.y)
                      * parameters.timeStep) / denominator
            nextVZ += (G * planetB.mass * (planetA.z - planetB.z)
                      * parameters.timeStep) / denominator
```

$$x'_j = vx_j$$
$$y'_j = vy_j$$
$$z'_j = vz_j$$
$$vx'_j = \frac{G\sum_{j\neq k} m_k(x_k - x_j)}{[(x_j - x_k)^2 + (y_j - y_k)^2 + (z_j - z_k)^2]^{\frac{3}{2}}}$$
$$vy'_j = \frac{G\sum_{j\neq k} m_k(y_k - y_j)}{[(x_j - x_k)^2 + (y_j - y_k)^2 + (z_j - z_k)^2]^{\frac{3}{2}}}$$
$$vz'_j = \frac{G\sum_{j\neq k} m_k(z_k - z_j)}{[(x_j - x_k)^2 + (y_j - y_k)^2 + (z_j - z_k)^2]^{\frac{3}{2}}}$$

Image Source

MPI vs Sequential Data Handling

Sequential:

```
all_Bodies = [ Body1: { x, y, z, vx, vy, vz, m },
               Body2: { x, y, z, vx, vy, vz, m },
               Body3: { x, y, z, vx, vy, vz, m },
               Body4: { x, y, z, vx, vy, vz, m },
               Body5: { x, y, z, vx, vy, vz, m },
               Body6: { x, y, z, vx, vy, vz, m },
               Body7: { x, y, z, vx, vy, vz, m },
                ...
                ...
                ...
               BodyN: { x, y, z, vx, vy, vz, m } ]
```

MPI:

0

```
all_Bodies, local_B = all_Bodies[0:A]
```

1

```
all_Bodies, local_B = all_Bodies[A+1:C]
```

...

N

```
all_Bodies, local_B = all_Bodies[C+1:N]
```

Serial int main()

1. Allocate Memory for **bodies**
2. Initialize each **body** with random values
3. Run the calculation for **K** steps
4. Print time taken
5. Free memory

MPI int main()

1. Allocate memory for **all_bodies** and **local_bodies**
2. Based on **N** establish **local_body** count and **displacement** in **all_bodies** for each process. (This allows data from each process to be sent to the correct index in **all_bodies** in every other process)
3. Initialize **all_bodies** within process of **rank zero**.
4. Broadcast **all_bodies** from **rank zero** to **all** other processes.
5. ScatterV **all_bodies** to every process evenly into **local_bodies**.
6. Run the calculation for **K** steps on **local_bodies** in each process.
7. After each calculation, use All-GatherV to update **all_bodies** with each processes' updated **local_bodies**. Utilize **displacement** to ensure **local_bodies** are entered into correct index within **all_bodies**.
8. Print time taken from **rank zero** and free memory.

1) Allocate memory for *all_bodies* and *local_bodies*

```
Body *localBodies = malloc(local_n * sizeof(Body));
Body *allBodies = malloc(numBodies * sizeof(Body));
```

2) Establish displacement indexes for each process

```
int *recvcounts = malloc(world_size * sizeof(int));
int *displs = malloc(world_size * sizeof(int));
int sum = 0;
for (int i = 0; i < world_size; i++) {
    // data amount for process i
    recvcounts[i] = (numBodies / world_size) * sizeof(Body);
    // starting index in allBodies for process i
    displs[i] = sum;
    sum += recvcounts[i];
}
```

3) Initialize All_Bodies

```
if (world_rank == 0) {
        initializeBodies(allBodies, numBodies);
    }
```

4) Broadcast All_Bodies to all processes

```
MPI_Bcast(allBodies, numBodies * sizeof(Body), MPI_BYTE, 0, MPI_COMM_WORLD);
```

```
allBodies                -> receiving/sending buffer
numBodies * sizeof(Body) -> data size
MPI_BYTE                 -> data type
0                        -> root process
MPI_COMM_WORLD           -> communicator
```

5) ScatterV All_Bodies to each processes' Local_Bodies

```
MPI_Scatterv(allBodies, recvcounts, displs, MPI_BYTE, localBodies, local_n * sizeof(Body), MPI_BYTE, 0, MPI_COMM_WORLD);
```

```
allBodies              -> sending buffer
recvcounts             -> sent data size
displs                 -> send from index
MPI_BYTE               -> sent data type
localBodies            -> receiving buffer
local_n * sizeof(Body) -> received data size
MPI_BYTE               -> received data type
0                      -> root process
MPI_COMM_WORLD         -> communicator
```

6/7) Run the simulation and All-GatherV

```
for (int step = 0; step < numSteps; step++) {
    // calculation same as sequential except outer loop uses localBodies
    computeForcesAndMove(localBodies, allBodies, local_n, numBodies);
    MPI_Allgatherv(localBodies, local_n * sizeof(Body), MPI_BYTE, allBodies,
recvcounts, displs, MPI_BYTE, MPI_COMM_WORLD);
}
```

```
localBodies             -> sending buffer
local_n * sizeof(Body)  -> sent data size
MPI_BYTE                -> sent data type
allBodies               -> receiving buffer
recvcounts              -> received data size
displs                  -> receive into index
MPI_BYTE                -> received data type
MPI_COMM_WORLD          -> communicator
```

8) Print time taken and free memory

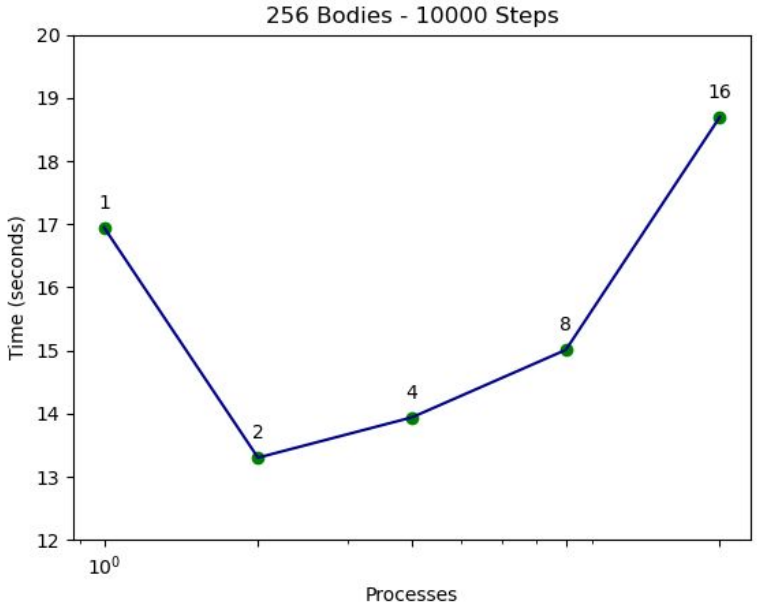
```
if(world_rank == 0)
{
    printf("%lf\n", endtime - starttime);
}

free(localBodies);
free(allBodies);
free(recvcounts);
free(displs);
MPI_Finalize();
```

Verification

- Run sequential and MPI scripts with set random seeds and equal body count / step count.
- Print each body's final position and velocity into a file.
- Utilize diff command to check output files are identical.

```
Outputs are identical for 10 bodies and 10 generations.
Outputs are identical for 100 bodies and 100 generations.
Outputs are identical for 1000 bodies and 1000 generations.
```

Performance

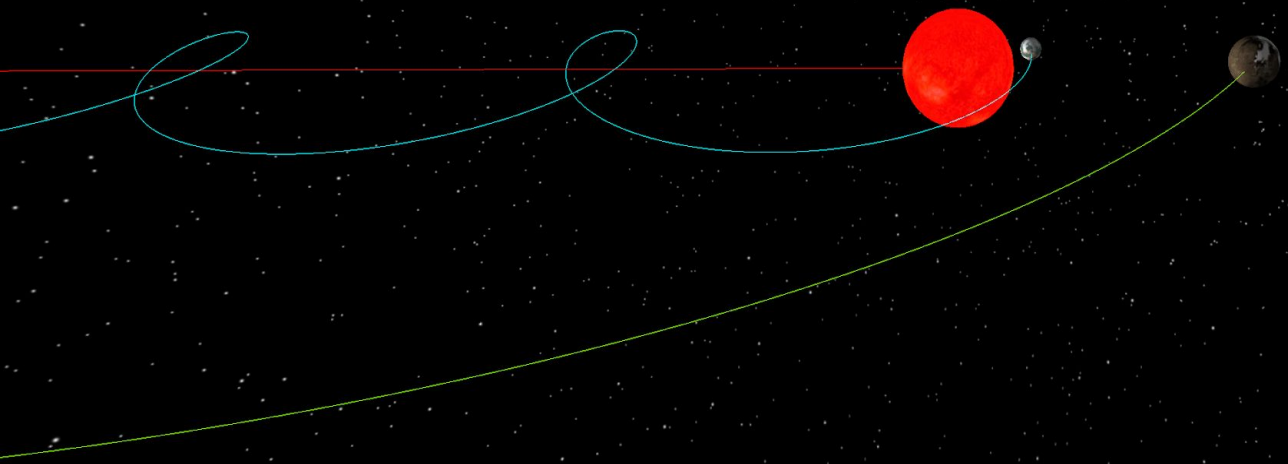


Analysis and Conclusion

- **Speed up decreases** with process count **greater than two** due to **overhead** associated with communicating with every process each step in the simulation.
- Testing on a machine with **more than two cores** may demonstrate **greater speed up**.
- Successful implementation of **OpenMPI**!

What's next?

- Implement visualization of the simulation.
- Optimize with CUDA?
- How can we avoid communicating excess information between processes?



Thanks!

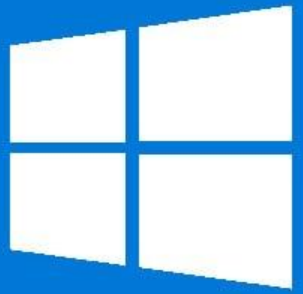
Your PC ran into a problem and needs to restart. We're just collecting some error info, and then we'll restart for you.

20% complete

For more information about this issue and possible fixes, visit https://www.windows.com/stopcode

If you call a support person, give them this info:

Stop code: CRITICAL_PROCESS_DIED



Sources

[1] https://en.wikipedia.org/wiki/N-body_problem

[2] https://www.open-mpi.org/doc/

[3] https://www.mpich.org/static/docs/latest/www3/