Assignment 3 N Body Simulation
CSC4005 Distributed and Parallel Computing

Chen Yuan 117010038
School of Science and Engineering
The Chinese University of Hong Kong, Shenzhen
117010038@link.cuhk.edu.cn

Abstract

This assignment implements a N body problem, using 4 versions – one sequential method and 3 parallel methods, namely the MPI method, pthread method, and openmp method. The report will present the principle, codes, as well as comparasion and analysis of performance of different implementations.

1. Introduction

The N Body problem refers to the problem of predicting the motion of n celestial bodies that interact gravitationally. Numerical methods must be used to simulate such systems. Initially, the bodies are listed in random places in a 2D space. The gravity between N-body is : $F = G\frac{m_1 \times m_2}{r^2}$.

Here are some assumptions about the collision and bouncing:

1. The display size is 200×200, and iteration number is 100 for all cases.

2. When a body goes to the edge of the display, it will change the direction so as to keep within the 200×200 display. For example, when a point goes right to the edge of x axis, it will turn left with the same velocity.

3. When two bodies have a collision, it will not produce small bodies, and we simply assume that nothing else happens.

The following Figure 1 is a screenshot during the simulation with display size 200×200.

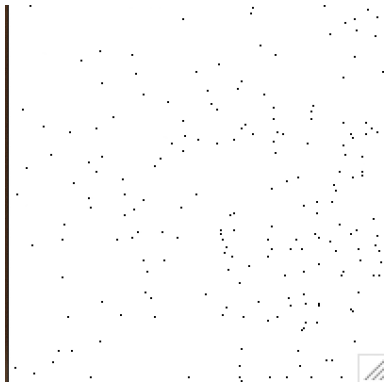

Figure 1. caputre of N body simulation by X11

The implentation details and performance analysis are shown below.

2. Method

For all methods, we use For parallel methods, we use the number of processes or threads ranging from 1 to 16.

2.1. Sequential

At one time stamp, need to calculate the force of each body with all other bodies, then get the acceleration, then get the velocity and the new position by numerical method. (Here, we set the Δt to be 0.01).

Inspired by the lecture, we have the following formulas to calculate the new position:

$F = G\frac{m_1 \times m_2}{r^2}$

$F = ma = m \cdot \frac{v^{t+1} - v^t}{\Delta t}$

then, we can get the new velocity

$v^{t+1} = v^t + \frac{F\Delta t}{m}$

then we can update the new position (in both xx and y axis)

$x^{t+1} = x^t + v\Delta t$

The psudocode is shown in the following:

for $t = 1 \cdots, iteration$ do
  for $i = 0 : num\_bodies - 1$ do
    $F = Force\_routine(i)$;
    $v[i]\_new = v[i] + F \cdot dt/m$;
    $x[i]\_new = x[i] + v[i] \cdot dt$;
  end for
  for $i = 0 : num\_bodies - 1$ do
    update velocity;
    update poosition;
  end for
end for

For the Force_routine part, a typically method is to compute the forces of all other bodies on a particular body and add them up. However, this method may do some exhausting calculations since the force between any two bodies are calculated twice. For a more simplicated version, we can just do the following to compute the all body forces:

Initialize all body forces to be 0.
for $i = 0 : num\_bodies - 1$ do
  for $j = i + 1 : num\_bodies - 1$ do
    $F[i] += Force\_between(body[i], body[j])$;
    $F[j] -= Force\_between(body[i], body[j])$;
  end for
end for

Data Structures: We define three data structures in the code: Body(contain x, y, and mass); Force(contain Fx, Fy), and Velocity(contain Vx, Vy) for simplier representation of 2D graph. All the instances are in DOUBLE format.

2.2. OpenMP

In the sequential method, there are two for loops in each iteration. One is for calculating the force and velocity for each body, and another is for updating the location and velocity. Then we can add "#pragma omp parallel for" before each for loop.

The flow chart is shown in Figure 2.

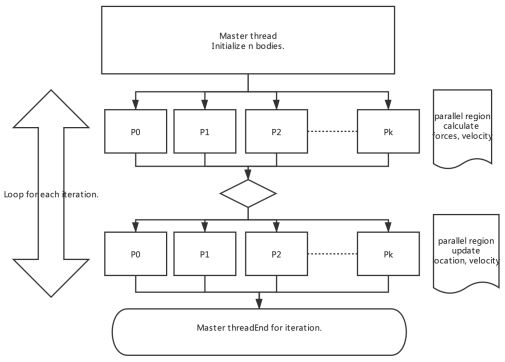

Figure 2. Flow chart for OpenMP implementation

2.3. MPI

In the MPI implemantation, assume there are n nodies and k processes including the MASTER node.

1. First attribute the $\frac{n}{k}$ part to each node, calculate the forces and velocities.
2. Second do the computation for the extra parts $n\%k$ on MASTER.

During each iteration, we need MPI_Bcast to synchronize all bodies' position and mass. Then in each process, do the calculation for force and velocities of $\frac{n}{k}$ bodies. Do the calculation for force and velocities of $n\%k$ bodies on MASTER. Then use MPI_Allreduce to synchronize bodies' position in each slave node, and use MPI_Bcast again to synchronize the $n\%k$ bodies on MASTER to all slaves.

Since the MPI version needs extra communication costs, the method of calculating all body forces without exhausting calculation is hard to implement. We just calculate the force between each body with all other bodies.

The flow chart is shown in Figure 3 .

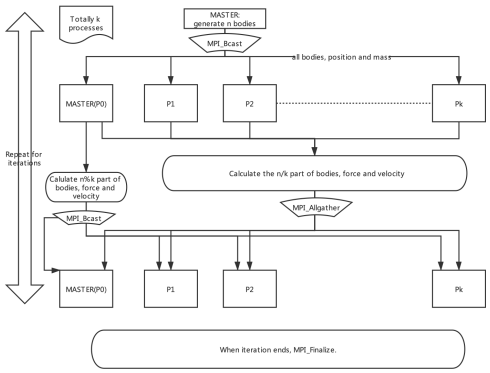

Figure 3. Flow chart for MPI implementation

2.4. Pthread

Since all threads can work in parallel, here we adopt the number of threads ranging from 1 to 16 in the experiment. The program is flexible for different inputs of thread numbers, and bodies. Just like the MPI version, we distribute data to each threads with $\frac{n}{k}$ bodies to compute, but the last thread contains $\frac{n}{k} + n\%k$ bodies. In the implementation, pthread_barrier_init(barrier) is used to maintain each thread's pace, in order to finish one iteration. Global variable bodylist, newblist, vellist, newvlist are used to record and update the position and velocity of bodies after each iteration.

Figure 4 shows the flow chart of Pthread method.

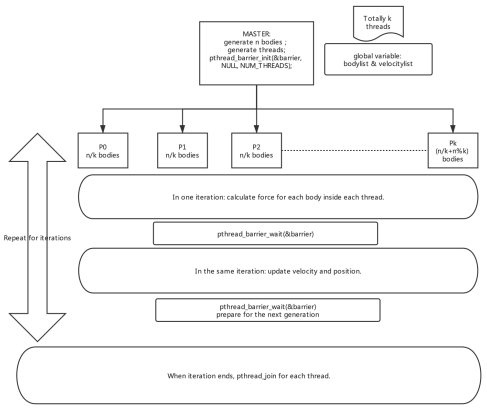

Figure 4. Flow chart for Pthread method

Remark: for drawing the program, when it comes to the second pthread_barrier_wait in one iteration, we can add one barrier wait in the master process. Then we can conduct drawing in master for each iteration.

2.5. Command lines

All the heigh and width in the display are 200 by default. For the MPI program, the first instance of main function is number of bodies. For Openmp programs, the first instance is number of threads(default is 1) and the second instance is number of bodies. For pthread program, the first instance is number of threads(default is 1), and the number of bodies needs to change the variable in the file.

In order to evaluate the performance namely runtime, the drawing procedure is not included in the parallel programs. However, the "seq_draw.cpp" indicates a sequential version with the X11 drawing procedure. And a video recording the simulation is attached in the package.

The sample compile codes are listed below:

```
#Sequential
g++ ./seq.cpp -o ./seq -lX11
./seq 800
#MPI
mpic++ ./mpi.cpp -lX11 -o ./mpi
mpirun -n 8 ./mpi 800
#Pthread
g++ ./pth.cpp -o ./pth -lpthread -lX11
./pth 8
#OpenMP
g++ -o ./openmp ./openmp.cpp -fopenmp
./pth 8 800
```

3. Experiment

The most important metric here is runtime under different circumtances. Since the X11 drawing procedure is slow, here we only consider the computation runtime, including the data transmiting, and without displaying the graph. The total results are attatched in Appendix A. One important figure that consolidates all usful information in performance is shown in Figure 5. The red horizontal line in the figure refers to the sequential implementation time (calculated by the average of 5 times).

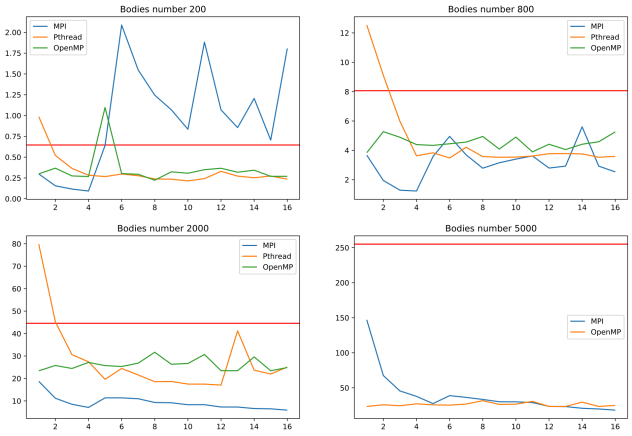

Figure 5. Performance comparasion figure.

Sections below are several analysis based on different evaluation metrics:

3.1. The number of processes or threads used in the program

As shown in Figure 5, generally, as the number of threads or processes grows, the total time for a program decreases. Meaning that as the nodes for parallel process grows, the total excucation time generally shrinks. On the figure shown above we can observe that especially for Pthread implementation, as the number of

the parallel nodes increases, the excucation time first experiences a sudden decreases for the first 1-4 nodes, then may shrink to a rather low area for more nodes. For parallel nodes exceeding 8, the speed up performance is may not be severe and clear.

However, there is an exception in MPI inplementation on small datasets (namely 200 bodies).As the number of processes increases, the total time for overall computation increases, and even increases much more than that for sequential implementation. The reason behind this may be that since the MPI needs send and receive implementation, the overhead for data transmitting between nodes are huge.

3.2. The number of bodies ranging from 200, 800, 2000, to 5000

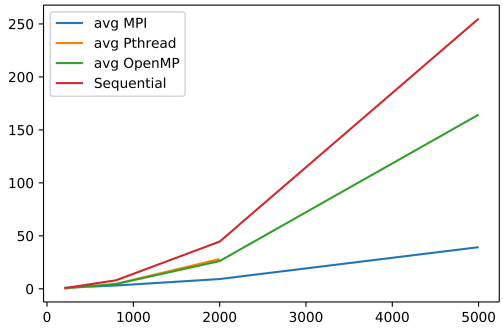

Figure 6. Average running time VS number of bodies.

The number of bodies in the system greatly influences the time for data processing. As shown in Figure 6, as the number of bodies increases, the running time also increases. It shows a convex function trend and the increase rate in time versus body number is the biggest for sequential work, then for Pthread, then OpenMP, at last MPI. MPI runtime increases really small comparing to the other three methods.

3.3. MPI vs Sequential vs Pthread vs OpenMP

With the study with different metrics, we can say that the MPI method performs really well, with nearly 4 nodes to be the best option. All three methods would not have greater speed up when processors become more than 6-8.

Generally, MPI implementation is good for larger size of data. As data size goes up, it would have a better effciency, and it performs more stable for large number of processors.

OpenMP could be the second option. It performs not bad in all cases, and would be stable for different number of nodes. Actually for OpenMP method, we can use the simplified version of force computing as mentioned above, to avoid exhausting calculation. However, in real test, it results in a larger running time than normal method. This reason may be that the simplified version of Force calculation requires another for loop for velocity calculation, which increases the overall time in turn.

Pthread method encounter a big drop from 1 to 4 threads in general, which is more obvious than other methods. Sometimes in lower number of nodes, its running time would be larger than that of sequential running time. However, as the input size increases, it may increase more in running time than other methods.

Overall, the parallel method performs better than sequential method. Here is also an analysis of efficiency for different methods theroically(omit the transmission time):

Sequential: under each iteration: $O(n^2)$; total: $O(m \cdot n^2)$, where m is the number of iterations.

Parallel: under each iteration: $O(\frac{n^2}{k})$;

total: $O(m \cdot \frac{n^2}{k})$

4. Conclusion and Reflection

In this experiment, 3 parallel versions of N body Problem is conducted with pthread, MPI, and OpenMP approach. The results shows that generally as the number of threads or processes increases, the total time for the problem will decrease. After 8 nodes, the performance would not have a big speed up. And the number of bodies determines generally how much time a sequential version could be used. As the size of figure increases, the time needed increases as well. Also, the best version is the MPI parallel version, normally with 4 processes.

Through this project, I understand more about the implementation details of different parallel methods, and trained my X11 implementation as well. Since one task is only limited to 5 min on the server, it would be better if I can find thhe running time for Pthread, 5000 bodies or larger if time permits. OpenMP+MPI methhod also needs more digging.

A. Experiment Results

The total results are listed in Figure 7. Focus on runtime for each case.

1

		num_bodies			
method	process	200	800	2000	5000
Sequential	1	0.645882	8.064199	44.577399	255.027984
MPI	1	0.2981	3.67086	18.72262	147.04071
MPI	2	0.15543	1.94913	11.19838	67.49418
MPI	3	0.11652	1.28626	8.50533	45.56408
MPI	4	0.09284	1.23017	7.09912	37.82939
MPI	5	0.64013	3.58089	11.36644	27.52291
MPI	6	2.08806	4.9552	11.37187	38.86694
MPI	7	1.54778	3.68956	10.97583	36.32744
MPI	8	1.24449	2.78559	9.30494	33.44564
MPI	9	1.06947	3.15581	9.18108	30.12844
MPI	10	0.83531	3.40343	8.30846	30.0228
MPI	11	1.88236	3.62112	8.32321	28.95005
MPI	12	1.06833	2.79426	7.2943	23.53277
MPI	13	0.85489	2.92653	7.27868	23.25591
MPI	14	1.20559	5.60511	6.6145	20.81836
MPI	15	0.70487	2.92596	6.48221	19.84742
MPI	16	1.80682	2.5325	5.89102	18.1489
Pthread	1	0.98586	12.53905	79.83276	
Pthread	2	0.52076	9.10933	45.23536	
Pthread	3	0.36487	5.98104	30.6123	
Pthread	4	0.28334	3.63162	27.44217	
Pthread	5	0.26642	3.83432	19.62874	
Pthread	6	0.29691	3.48015	24.45099	
Pthread	7	0.27698	4.2092	21.56743	
Pthread	8	0.23811	3.57593	18.56438	
Pthread	9	0.2354	3.53097	18.66904	
Pthread	10	0.21427	3.54023	17.5046	
Pthread	11	0.24177	3.61254	17.48104	
Pthread	12	0.3293	3.77262	17.06076	
Pthread	13	0.27209	3.77987	41.19403	
Pthread	14	0.25214	3.75693	23.70812	
Pthread	15	0.27292	3.531	21.97542	
Pthread	16	0.23557	3.5885	25.08728	
OpenMP	1	0.29904	3.83754	23.4094	137.78547
OpenMP	2	0.36739	5.27477	25.78404	172.37507
OpenMP	3	0.27437	4.89286	24.45077	165.40293
OpenMP	4	0.26668	4.39321	27.15419	144.42339
OpenMP	5	1.09712	4.33929	25.73201	167.84307
OpenMP	6	0.30273	4.4495	25.32849	151.07292
OpenMP	7	0.29416	4.5613	26.82351	150.09084
OpenMP	8	0.22294	4.94593	31.68013	140.11702
OpenMP	9	0.32342	4.08713	26.33266	179.85084
OpenMP	10	0.30674	4.90584	26.67821	183.94087
OpenMP	11	0.3491	3.89418	30.68288	147.63798
OpenMP	12	0.36647	4.41612	23.4694	173.90883
OpenMP	13	0.31859	4.05506	23.45593	194.22293
OpenMP	14	0.34322	4.41763	29.61361	198.3523
OpenMP	15	0.26923	4.5843	23.43605	160.15953
OpenMP	16	0.26956	5.25995	24.86235	163.8167

result.xls

Figure 7. Overall result

B. Codes

All codes, video, analysis codes, can be found in the folder attached. On the server, the codes are just in /home/117010038.