資工系100062236林修安

Parallel Programming HW2

Roller Coaster & N-Body Problem

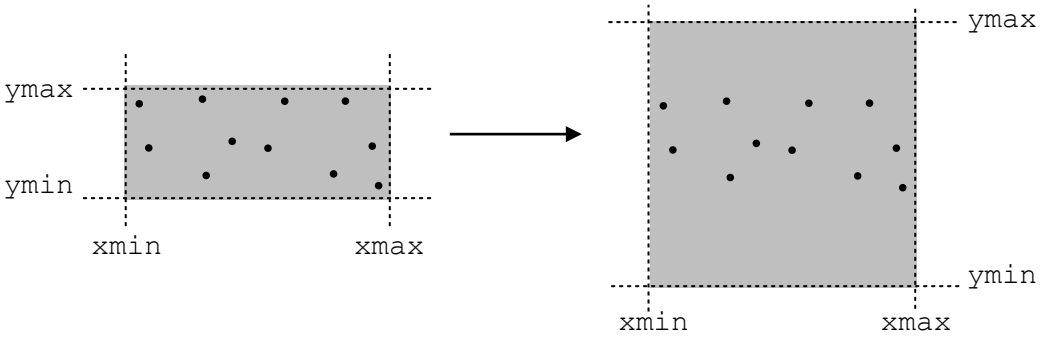
Implementation

1. Single Roller Coaster Car Problem Implementation

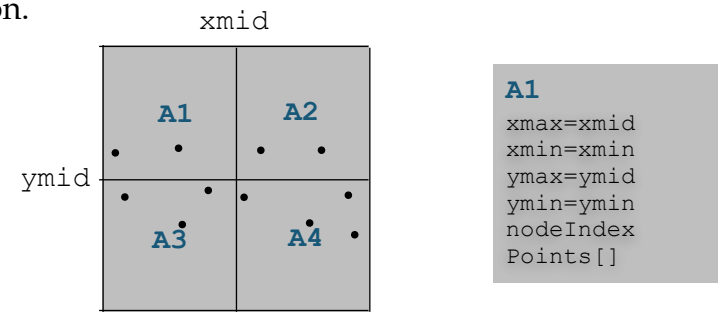
- Define a passenger struct with its `id` and a `inQ` to determine whether it's in queue. Thread with `id=0` is the roller coaster car.
- Passenger thread:
  (1) Every thread sleeps for random time, and is enqueued afterward. The enqueue action is locked with mutex.
  (2) Set the enqueued thread `inQ=1` if the simulation step not yet reach `N`.
  (3) Wait until `inQ` is reset to 0 by `while(inQ);`
- Car thread:
  (1) Wait until the number of passengers in queue reaches `C` by `while(Q.size()<C);`
  (2) Pop `C` passengers from the queue and put them into an array. The dequeue action is locked with mutex.
  (3) Sleeps for `T` ms.
  (4) After `T` ms, set `inQ` of the passengers to 0 to let them wander.
  (5) Increments the step count.

2. Barnes-Hut Algorithm Implementation

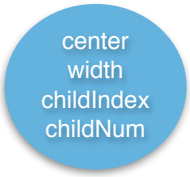
- Tree Construction:
  - Main thread:
  (1) The root node is built by main thread. Find the boundaries `xmax`, `xmin`, `ymax`, `ymin` of the input, and adjust them to equal length.



(2) Divide the input into four sections according to the boundaries. Assign node index corresponding to the section, save points within area and the new boundaries for each section.

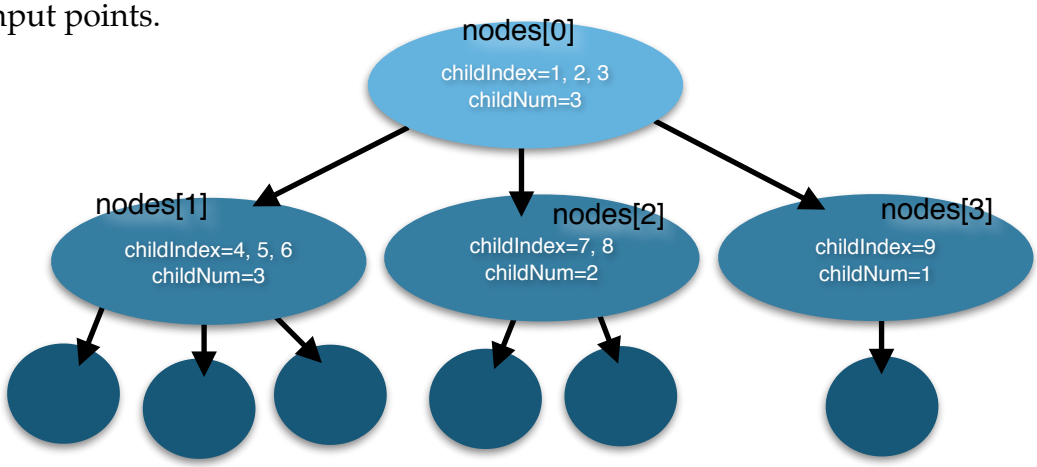


(3) Enqueue the sections if the number of `Points[]` is not 0.
(4) Root node holds the attributes:
the position of the mass center, the width of the section,
the node indexes of its children and the number of its children.



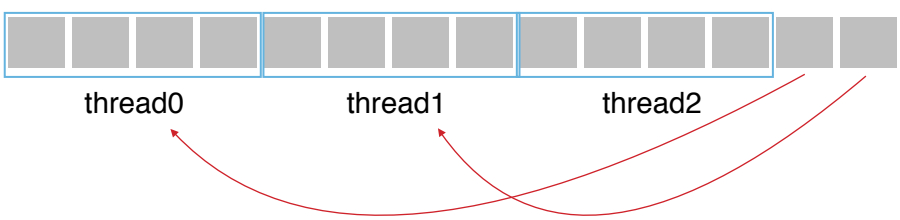
- Pthreads:

(1) Get the sections from queue if the queue is not empty. The dequeue action is locked with mutex.
(2) Create a new node of such section.
(3) Divide the section into another 4 smaller sections, and enqueue them if number of `Points[]` is not 0. If the section has only one point, do not enqueue the section. The enqueue action is locked with mutex.
(4) Repeat the above steps until the number of nodes is equal to the number of total input points.



• Force Computation:

(1) Divide the input points to each threads. If it's not dividable, add one additional point to each thread of the first `dataN%threadN` threads, and save the indexes of the first and last points one thread should hold.

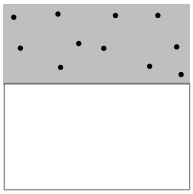


(2) Traverse the tree recursively. If `r/d<θ` or the node has no children, then calculate the force and return the value, else search the children of the current node.

- Synchronization:
  (1) Force computation is performed only when the tree is built. `treeBar` is the barrier that waits for all threads including main thread to finish building tree so further action can be started.
  (2) The tree must be cleared and rebuilt after every iteration. `compBar` is the barrier that waits for all threads including main thread to finish force computation, points update, and several initialization to start next iteration.
- Technique to balance loading:
  When partitioning the input points into 4 sections, we want the points to be evenly distributed between sections so the loading of threads is more equally shared.

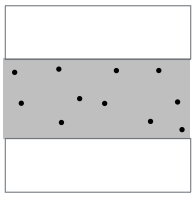
```
diffx = xmax - xmin;
diffy = ymax - ymin;

if(diffx > diffy)
     ymax += diffx - diffy;
else if(diffy > diffx)
     xmax += diffy - diffx;
```

If we expand the input section only by the difference of sides as above, it results in empty space with no points, and this causes threads to process empty sections, which is redundant.



```
if(diffx > diffy){
     ymax += (diffx - diffy)/2;
     ymin -= (diffx - diffy)/2;
}
else if(diffy > diffx){
     xmax += (diffy - diffx)/2;
     xmin -= (diffy - diffx)/2;
}
```

In order to avoid too much empty space, increment the max value of the shorter side by half of the difference, and decrement the min value of the shorter side by the same amount. Using the algorithm above can more likely place the points toward center, and enable more balanced divisions.



Experiment & Analysis

1. Single Roller Coaster Car

• Average Waiting Time v.s. Round Trip Time (T) v.s Car Capacity (C)

(1) Input setting:

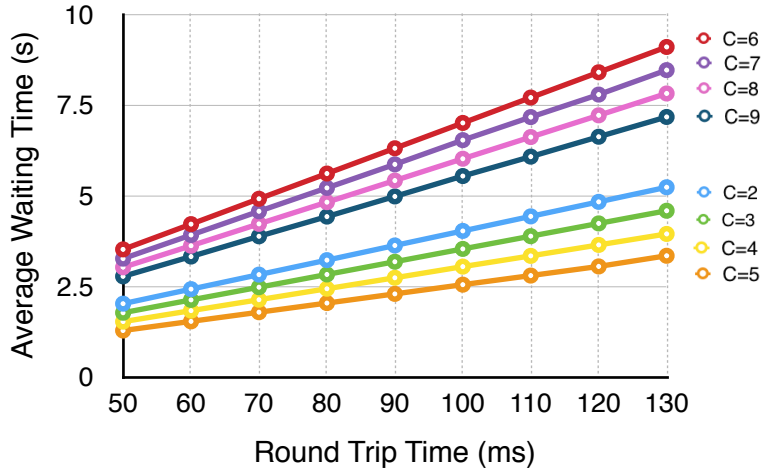
# of passengers (n) : 10
# of iteration (N) : 100
Wandering Time : T/2 (for observation convenience)

(2) Numeric Result

Wandering Time = T/2

n=10, N=100	C=2	C=3	C=4	C=5	C=6	C=7	C=8	C=9
50	2.009291	1.759077	1.513620	1.268600	3.505635	3.250802	3.011128	2.761482
60	2.410187	2.112847	1.816620	1.521444	4.201577	3.901697	3.605964	3.308547
70	2.809847	2.462433	2.118429	1.773565	4.900841	4.556148	4.209360	3.866402
80	3.209179	2.814377	2.420761	2.025589	5.596141	5.200454	4.806740	4.409479
90	3.613252	3.165258	2.723184	2.278779	6.292829	5.849141	5.403455	4.965100
100	4.014503	3.518655	3.030486	2.533191	6.991103	6.517343	6.004200	5.527260
110	4.415553	3.869625	3.326800	2.783787	7.690877	7.152457	6.601805	6.062278
120	4.815902	4.221381	3.628581	3.035443	8.387461	7.772893	7.200854	6.610659
130	5.218025	4.570517	3.931462	3.329747	9.085220	8.447179	7.803287	7.155937

(3) Figurative Result



(4) Observation:

(a) **Round Trip Time ↑, Average Waiting Time ↑**
As the car runs for a longer time per round, passengers waiting in queue must wait longer for the round to end.

(b) **Capacity ↑, Average Waiting Time ↓ (for C=1~5 & C=6~10)**
As the capacity increases, the car can hold more passengers per round, so the passengers in queue are more likely to be able to board sooner, thus the waiting time drops.

• Average Waiting Time v.s. Car Capacity (C) v.s. Wandering Time (t)

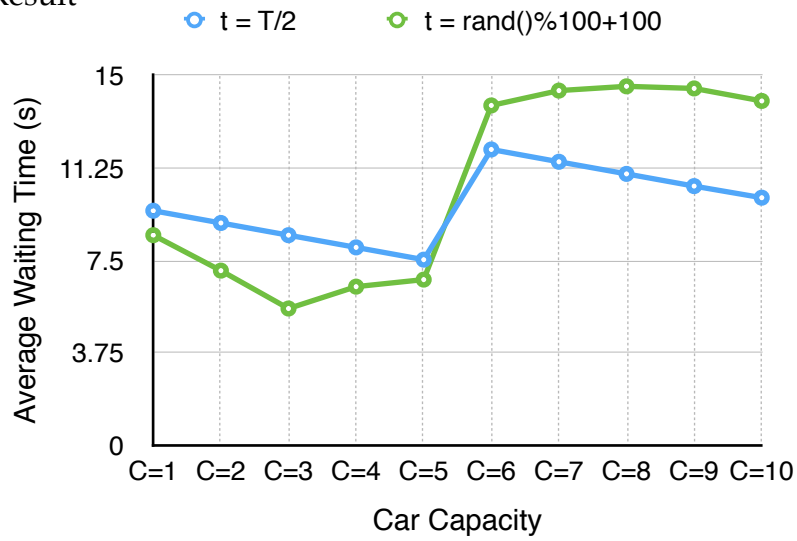
(1) Input setting:

# of passengers : 10
# of iteration : 100
Round Trip Time (T) : 100

(2) Numeric Result

n=10, N=100, T=100	t = T/2	t = rand()%100+100
C=1	9.510327	8.517519
C=2	9.012849	7.078074
C=3	8.517347	5.540475
C=4	8.021593	6.428391
C=5	7.523698	6.714647
C=6	11.997197	13.784453
C=7	11.497643	14.387043
C=8	11.006065	14.560887
C=9	10.509780	14.474916
C=10	10.039519	13.967546

(3) Figurative Result



(4) Observation:

Average waiting time decreases as C grows from 1 to 5, but suddenly rises as C is 6. Then as C grows from 6 to 10, waiting time drops again, but greater than that from C=1~5.

For C=1, after the first trip, all other 9 passengers are waiting in queue, so the last passenger needs to wait for 8 trips. For C=3, the last passenger wait for 3 trips, and for C=5, the last passenger wait for 1 trip, and this the same reason that waiting time decreases when C=6~10.

The gap from C=5~6 occurs because when C<=5, the number of passengers in queue always exceeds C, so the car starts the next trip right after the last one ends. For example, the car takes 5 passengers out for trip, there are still 5 passengers in queue, so the car can start another trip immediately. Since wandering time is less than round trip time, it is ensured that the queue is always filled with C or more passengers.

But when C>5, the car has to wait for passenger to return to queue to fulfill the capacity.
For example, the car first takes 6 passengers out for trip, only 4 passengers are in queue, which is not enough for the car to go for another round immediately, so the car must wait for the wandering passengers.

When the wandering time is selected randomly, the average waiting time still catch the similar trend.
Since the random time is determined by rand()%100+100, it may exceeds the round trip time, so when C=4, 5, it might not be able to move on right away, so the trend rises; when C=6, 7, the car may have to wait a little longer, so the trend rises a little bit, but it then drops as expected.

2. N-Body Problem

• Strong Scalability

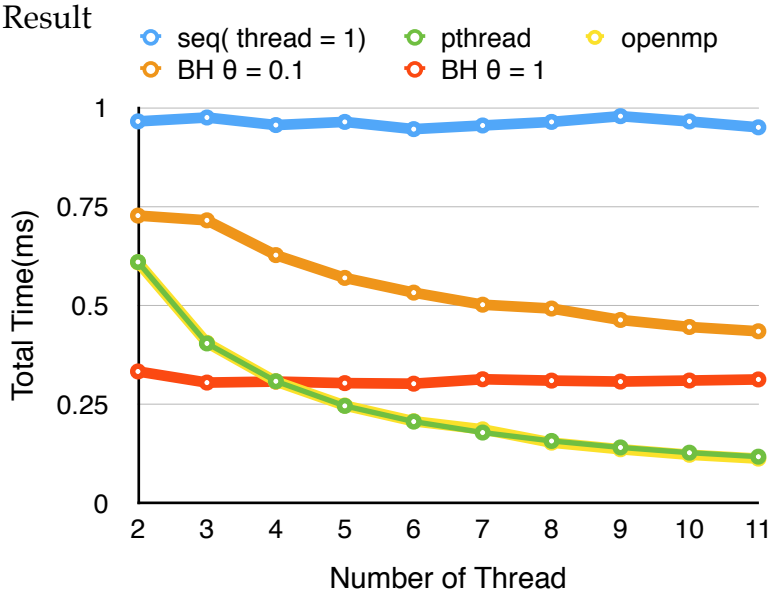
(1) Input setting:

Mass (m) : 1
# of iteration : 100
Δt : 1
# of bodies : 500

(2) Numeric Result

m=1, T=100, t=1, #bodies=500	seq (thread = 1)	pthread	openmp	BH θ = 0.1	BH θ = 1
2	0.964215	0.609024	0.608192	0.726311	0.332627
3	0.974015	0.403418	0.404957	0.714326	0.304531
4	0.954938	0.307889	0.307265	0.626496	0.307090
5	0.962911	0.245667	0.246569	0.568901	0.303150
6	0.944685	0.206029	0.206857	0.531721	0.301722
7	0.953873	0.178322	0.186392	0.501232	0.312935
8	0.962882	0.157095	0.152843	0.491495	0.309492
9	0.977393	0.140715	0.136125	0.462724	0.307064
10	0.964039	0.127453	0.122206	0.444928	0.309680
11	0.949209	0.117383	0.112497	0.434123	0.312458

(3) Figurative Result



(4) Observation:

The sequential version tends to remain the same, and pthread and openmp versions take the same amount of decreasing execution time. Since the number of bodies are small, BHalgo version does not exhibit its advantage.

• Strong Scalability

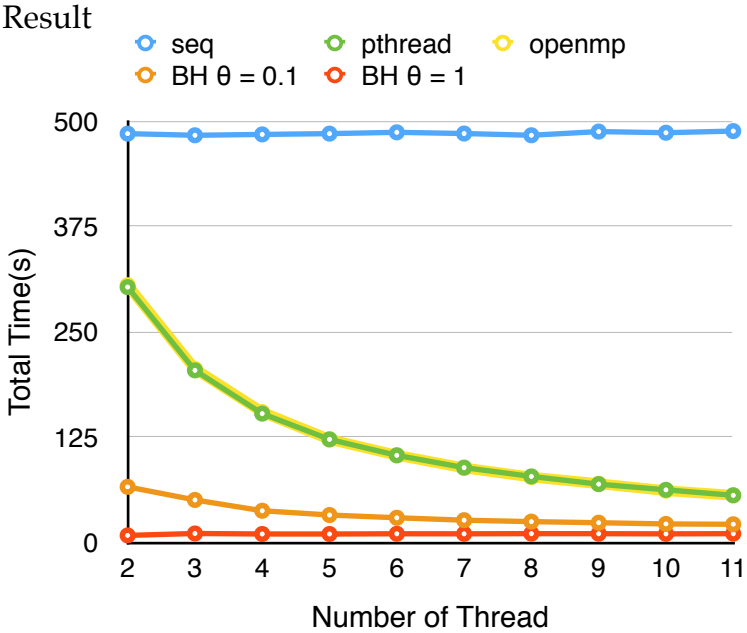
(1) Input setting:

Mass (m) : 1
# of iteration : 200
Δt : 1
# of bodies : 8000

(2) Numeric Result

m=1, T=200, t=1, #bodies=8000	seq (thread=1)	pthread	openmp	BH θ = 0.1	BH θ = 1
2	485.225451	302.771072	305.138930	65.460303	7.757423
3	483.267493	203.914775	205.583826	49.901266	10.028812
4	484.303937	152.479824	154.323134	37.068279	9.453575
5	485.229947	121.897643	121.867317	32.003779	9.420744
6	486.778262	102.9077350	103.054852	28.728640	9.703688
7	485.357461	88.325492	87.945641	25.827842	9.661048
8	483.289347	77.854348	77.140568	24.143909	9.850565
9	487.665479	68.671322	69.215336	22.834379	9.791125
10	486.298441	61.932402	60.935734	21.371015	9.596203
11	488.272491	55.403420	55.462418	20.997079	9.825789

(3) Figurative Result



(4) Observation:

The sequential version always spends constant execution time since only one a main thread is used.

The pthread and openmp versions take same amount of execution time, and as more threads are used, the total execution time is reduced.

Under this input setting, BHalgo version shows its outperformance when dealing with big amount of input and number of iteration by using centers of mass for approximate calculation. BHalgo with θ = 1 spends less time because when

computing the force, it is more likely to choose the center of mass to compute the result, so the traversal does not need to run as deep as θ = 0.1.

• Weak Scalability - Total Time v.s. Number of Iteration

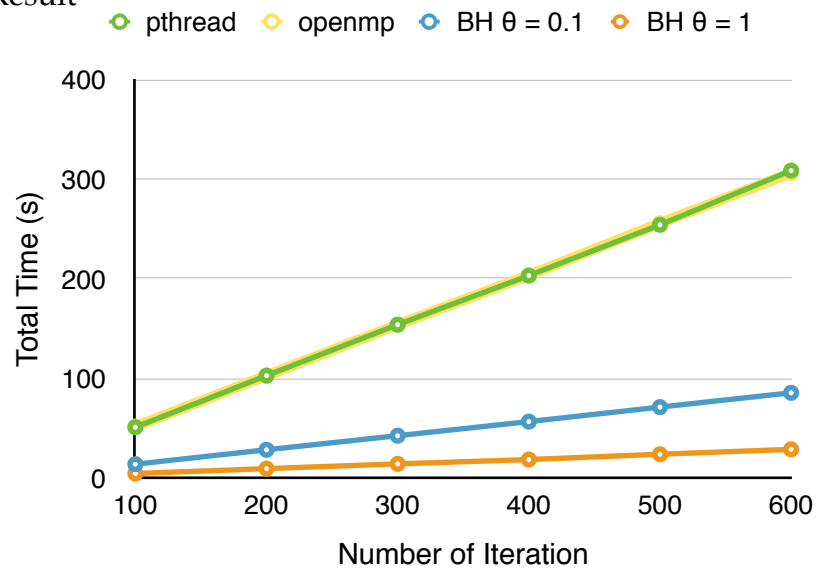
(1) Input setting:

# of thread : 6
Mass (m) : 1
Δt : 1
# of bodies : 1000

(2) Numeric Result

m=1, t=1, #bodies=1000, #thread=6	pthread	openmp	BH θ = 0.1	BH θ = 1
100	51.244948	51.866612	14.213625	4.815913
200	103.023883	102.931688	28.635711	9.761057
300	154.092519	153.911083	42.758936	14.506621
400	203.314276	203.545912	56.889353	18.837094
500	254.191997	255.803017	71.448811	24.191343
600	308.580800	306.209480	85.765144	29.089770

(3) Figurative Result



(4) Observation:

As the number of iteration increases, the execution time of all versions also increase. Moreover, all of them have linear growth. The execution time is proportional to the number of iteration since the number of threads used and the number of bodies remain the same in each iteration. So total execution time is approximately the multiple of time spent in one iteration.

• Phase Time for Different θ

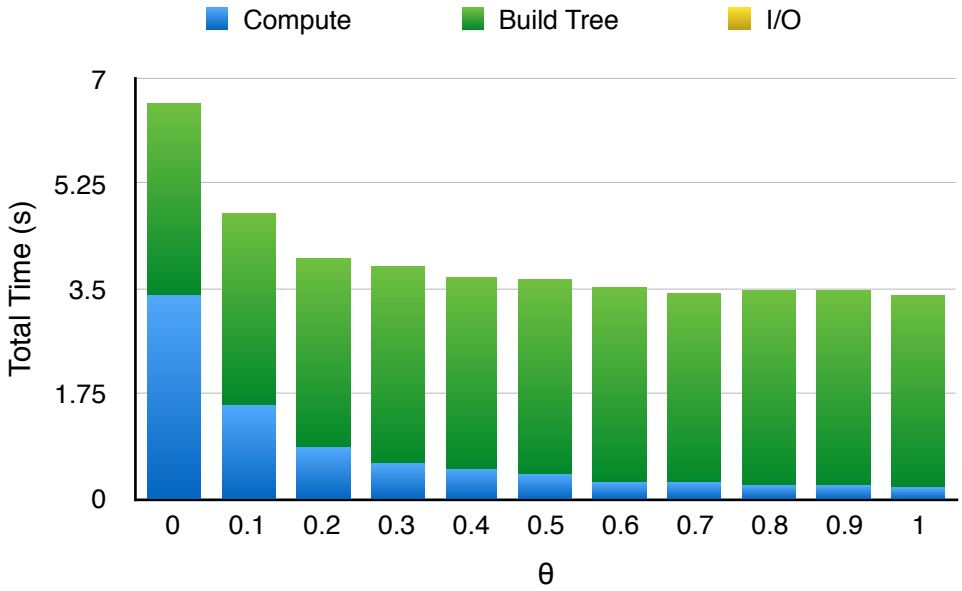
(1) Input setting:

# of threads : 10
Mass (m) : 1
# of iteration : 50
Δt : 1
# of bodies : 500

(2) Numeric Result

m=1, t=1, T=50, #thread=10, #bodies=500	Compute	Build Tree	I/O
θ=0	1.674437	1.590612	0.000663
θ=0.1	0.717017	1.567437	0.000680
θ=0.2	0.369871	1.578119	0.000721
θ=0.3	0.241705	1.576585	0.000668
θ=0.4	0.187798	1.568771	0.000805
θ=0.5	0.152175	1.603382	0.000826
θ=0.6	0.130371	1.552357	0.000677
θ=0.7	0.116086	1.549271	0.000681
θ=0.8	0.104505	1.566512	0.000801
θ=0.9	0.091073	1.549882	0.000831
θ=1	0.068778	1.556741	0.001409

(3) Figurative Result



(4) Observation:

(a) Computing time decreases as θ gets larger. When θ increases, the threshold of choosing center of mass to compute force becomes less tight, so more calculation uses center of mass when computing force, and thus the execution time decreases.

(b) Time of building tree remains almost the same since it's not affected by θ. In my BH algorithm, I slice the input into 4 sections and put them into a queue. When a thread needs to get a section from the queue, the action must be locked. While popping and pushing must be locked, the tree building phase is not purely parallel, thus the tree building phase becomes the bottleneck of my program.

• Phase Time v.s. Number of Thread

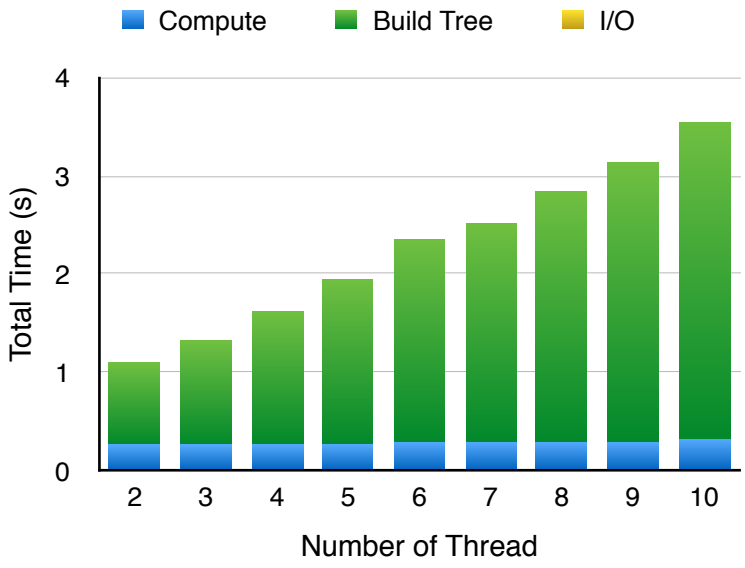
(1) Input setting:

Mass (m) : 1
# of iteration : 100
Δt : 1
# of bodies : 500
θ : 0.5

(2) Numeric Result

m=1, t=1, T=100, θ=0.5, 500 bodies	Compute	Build Tree	I/O
2	0.276918	0.819478	0.000687
3	0.263525	1.070544	0.000828
4	0.263927	1.352499	0.000690
5	0.267722	1.684505	0.00820
6	0.299991	2.062719	0.000680
7	0.281844	2.219444	0.000736
8	0.287750	2.553143	0.000692
9	0.302129	2.827530	0.000704
10	0.310109	3.227209	0.000685

(3) Figurative Result



(4) Observation:

Since number of bodies and iteration, and θ are controlled, the computing time and I/O time stay almost the same, and tree building time becomes the bottleneck. As mentioned above, the queuing operation can't be performed in parallel since popping and pushing should be locked. As a result, it's reasonable that more threads mean more locking, causing more time consumption for building the tree as more threads are used.

Experience

From the single roller coaster car problem, I become more familiar with pthread implementation and the locking mechanism than when learning the same thing in OS course.

From the N-body problem, besides knowing more about pthread and openmp, how to transform a sequential algorithm to a parallel one is the most essential technique I learned form the assignment. In order to finish the work, I have thought about several kinds of algorithm, but very few of them are feasible.

The hardest part is how to build the tree. I have thought of other approaches but they are too complicated to program.

Another difficulty is that I use an array to maintain the tree, but the memory size should be allocated is unknown. At first, a section is divided into 4 parts, and each part is divided into 4 smaller parts afterward. But thinking in this way, it turned out that N^2 size should be allocated, which is not appropriate. So by thinking another way, I obtain the size required. Two leaves are separated from one node, so by bottom-up strategy, 2N is the actual memory size needed.