Parallel Programming Homework 2

Single Roller Coaster Car Problem and N-Body Problem

CS 101062337 Hung Jin, Lin

1. Design: implementation of Barnes-Hut algorithm

(a) How to parallelize each phase of Barnes-Hut algorithm?

First create a thread pool with given argument, #threads, then make them wait for later coming jobs. The main implementation concept is to build 'workers' by creating new threads, and make the computing into separated jobs. When simulation starts, first push jobs into job queue, and wake workers thread up and make them begin processing, consuming the job by pop job from queue.

In Barnes-Hut algorithm, there are two phases can be parallelized, tree building and force computing. Both the above phases are related to n bodies' independent computations, every computation will not interfere each other, so it can easily be separate into jobs. Next, with jobs queue and triggering workers run to realize the parallel computing.

(b) How to partition the task?

It's more like streams with help of workers and jobs queue model. Not simply distributing jobs into each threads but make jobs flow into workers pool in stream, once finish a job then take in another, which is similar to the concept of dynamic scheduling with partition size of one in OpenMP.

First let the number of bodies be represented as N. There will be twice the number of bodies' tasks, one for N insertion to put bodies into corresponding space in quadrant tree, another one for N force computing process and calculate the target body's new position and properties.

(c) How to prevent from synchronization problem?

It has mentioned above that we have one resource sharing with multithreads: jobs queue, and therefore we have to use mutex to protect it from incoherence. Take a real example in code, we have to access it in each recalculation iteration cycle, below is the snippet

Push job into queue:

```
for (int i = 0; i < iters; ++i) {
    ...
    pthread_mutex_lock(&queuing);
    queuing_jobs = num_body, num_done = 0;
    pthread_mutex_unlock(&queuing);
    ...
}
```

Consuming jobs:

```
void* worker(void* param)
{
    while (true) {
        pthread_mutex_lock(&queuing);
        ...
        int i = --queuing_jobs;
        pthread_mutex_unlock(&queuing);
        ...
        pthread_mutex_lock(&queuing);
        num_done++;
        ...
        pthread_mutex_unlock(&queuing);
    }
}
```

Enclosed by mutex lock operation, inside are resource manipulations, first one is to push all possible jobs into queue by update queuing jobs indices and counter of finished job; the one in later also do the same thing, protect resource's indices by mutex.

Besides mutex, there's another important role: condition variable. When jobs queue is ready, we have to wake **all** waiting workers up by broadcast and make them consuming jobs. Next, the main thread has to wait for **all** jobs has been finished as a complete iteration so it make a conditional wait, and later will be notify and wake up by worker who's done the last job.

```
for (int i = 0; i < iters; ++i) {
    pthread_mutex_lock(&queuing);
    ...
    pthread_cond_broadcast(&processing);
    pthread_cond_wait(&iter_fin, &queuing);
    pthread_mutex_unlock(&queuing);
    Body* t = new_bodies; new_bodies = bodies; bodies = t;
}
```

(d) What techniques are used to reduce execution time and increase scalability?

I choose the thread pool model is due to the consideration of high cost in thread creations if create it whenever we need and destroy it when jobs end. And now it's easy to scalability by just tuning the numbers of workers (threads) from command-line arguments, and they will be control by condition variables: as needed then wake them up, and set a barrier condition to make them synchronized.

Another big optimization is to reduce the iteration in swapping old bodies set and new bodies set, namely remove the loop to copy new data into original data structure array, I just swap the pointer of the two set, this will reduce almost (number of bodies) * (iteration cycles) times memory copy. Some other trivial improvements are:

- Access data and pass parameters to functions by reference
- Make function tagged with compiler *inline* hints to possibly reduce cost in high frequency function calls
- Reduce duplicate calculation by collecting similar computing problems and temporarily saving them in variable

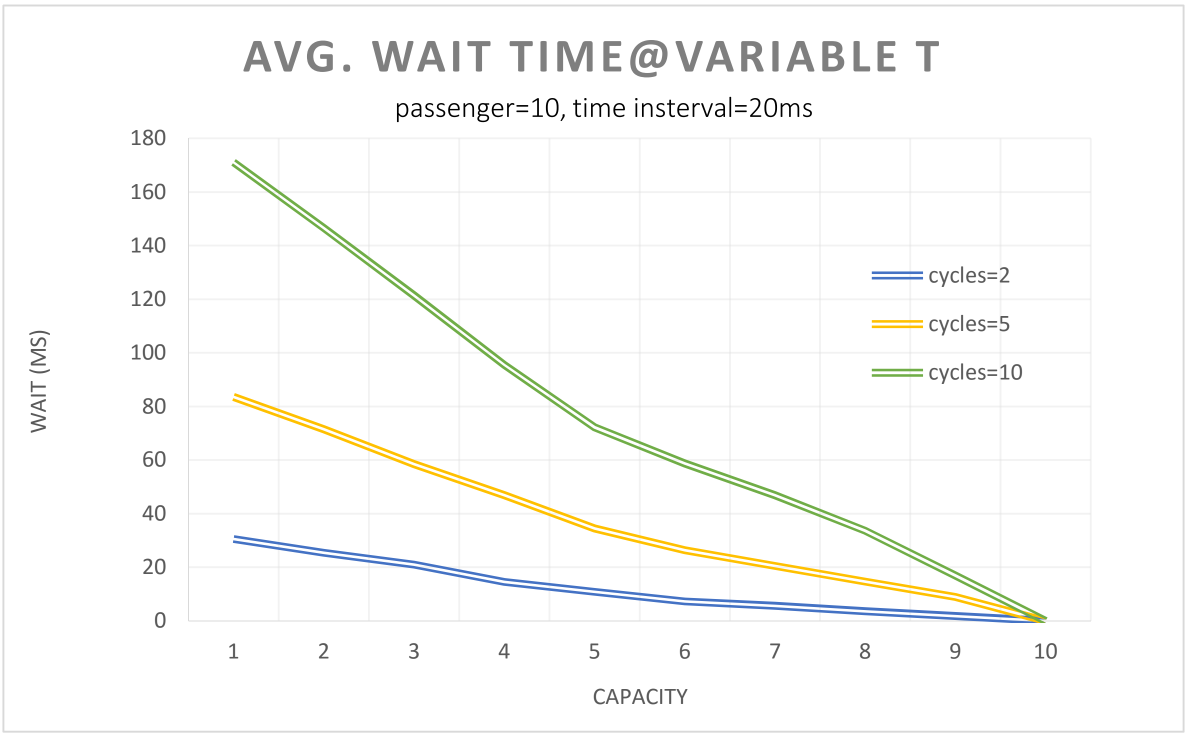
(e) Other efforts made in program

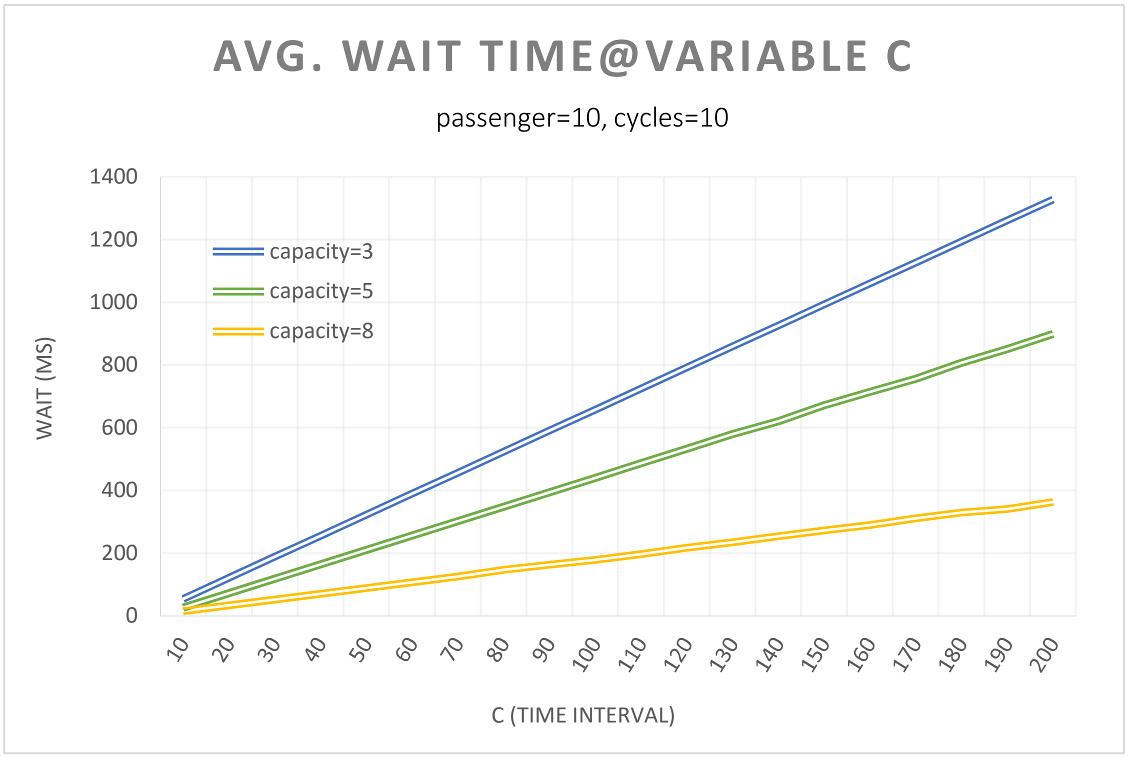
Follow the DRY (Don't Repeat Yourself) concept, I've made the duplicate part in four versions of code, including sequential ver., OpenMP ver., pthread ver., and Barnes-Hut approximation ver., into header file (utils.h) and implementation file (utils.cpp).

In Xlib programming part, I made the window manipulations reduced by preventing doing duplicate works, like draw lines to the same place, and finally get better performance in display without annoying flicker.

2. Performance analysis of Single Roller Coaster Car Problem

Plot the average waiting time vs the input parameter C or T

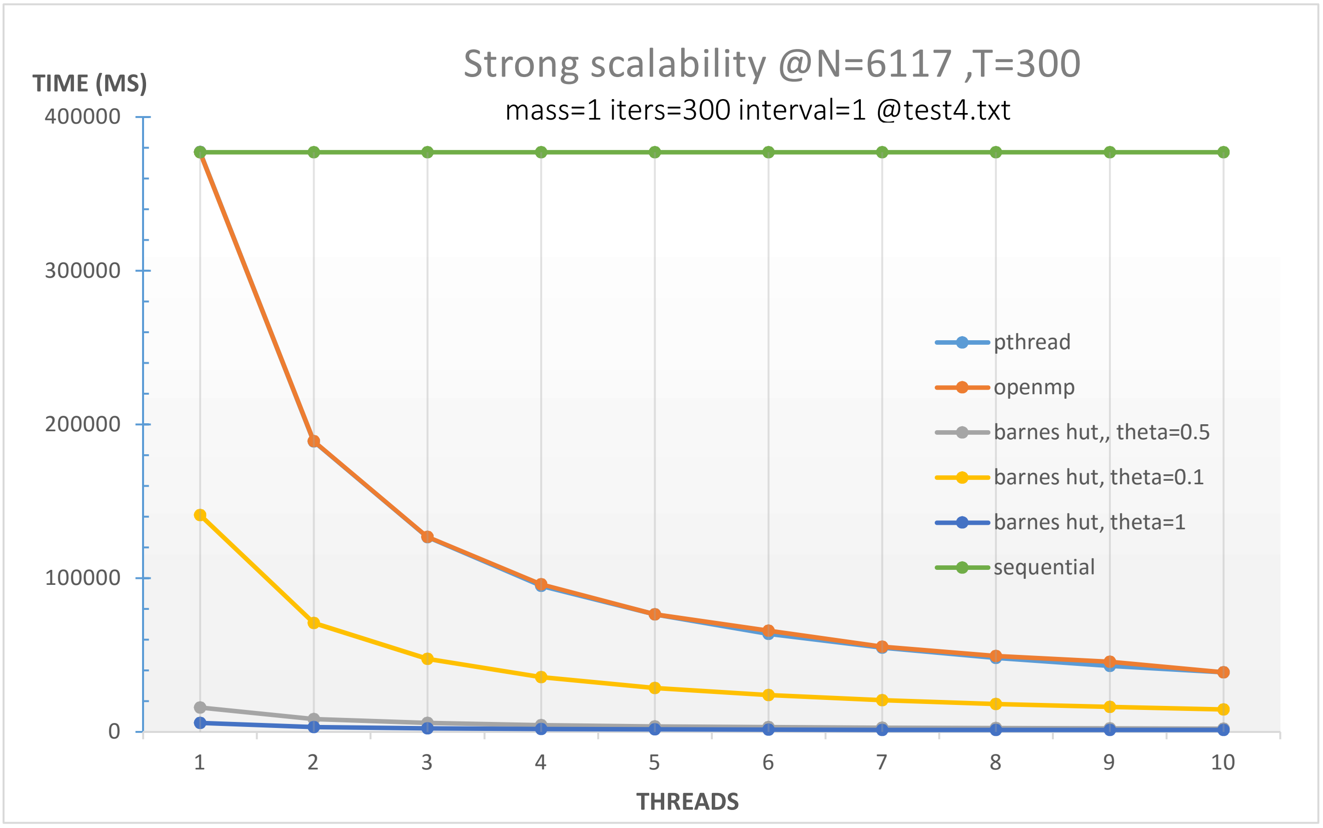




3. Performance analysis of N-Body Problem

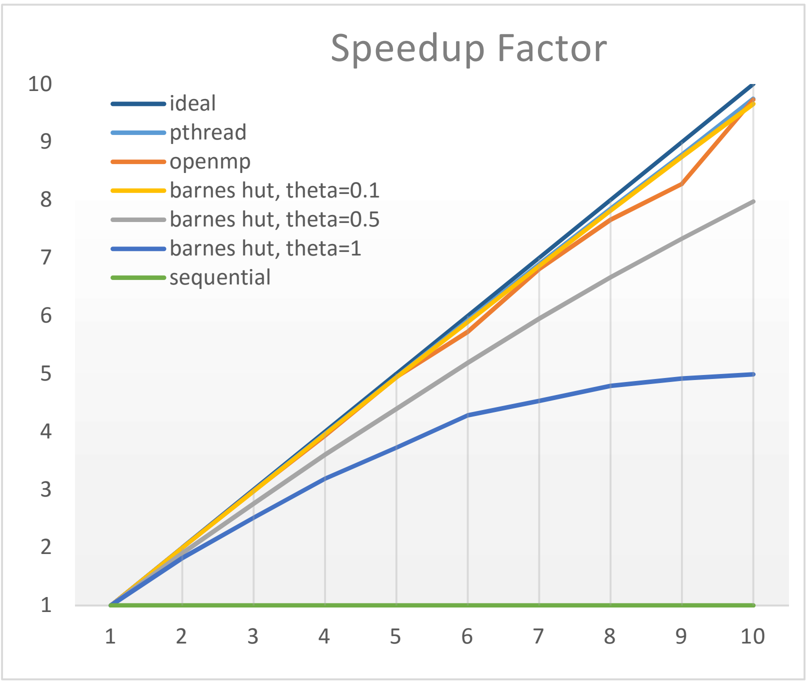
(a) Strong scalability

i. Fix N, T, and manipulates #threads. Compare execution time of each version.



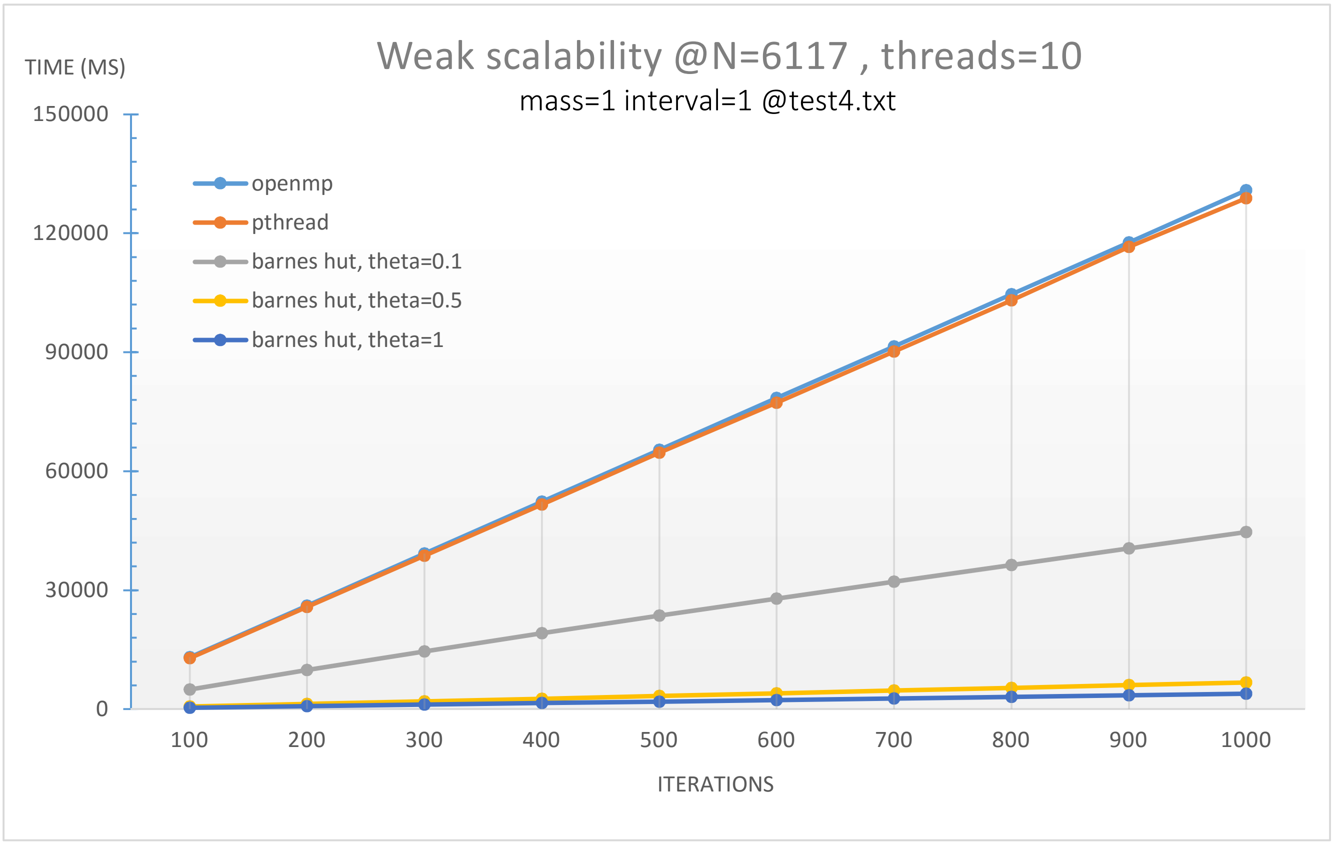
Use the 4th arguments series in TA's *parameter* file, fix the number of bodies (N) as 6117 and iteration cycles (T) as 300. We can observe:

- Brute force method in sequential vs. multithread version, though there's some cost in threads' synchronization, the total computing time are very close to sequential (the time difference is always less than 1ms in experimental result)
  - I use *mutex* lock for threads' synchronization, and the operations between *lock* and *unlock* are easy task, assigning jobs by change the job counter and counter for number of done job, which consuming only little time for that.
- Two implementations of brute force method, OpenMP and pthread versions, get almost the same result.
  - That means the_thread pool in pthread implementation is as efficient as the one in OpenMP which is east in programming by using some tags for thread manipulation. In my opinion, maybe the compiler sugar does the almost same thing comparing to my pthread implementation.
- Barnes-Hut approximation algorithm has a great speedup even with a small θ parameter. As θ =0.1, it wins twice in performance than the brute force multithreading versions.
  - In theory, the approximation version must faster than brute force one. But there may be some obstacles in implementation which make it not efficient as expected. There're two important phase in Barnes-Hut algorithm: building tree and body computing; building tree cost N*log(N) and so does body computing. With so many factors will may make poor r, the approximation on a well-distributed data case finally wins and overcome other bad factors.
- With θ =0.5 and θ =1, the Barnes-Hut method gets the overwhelming win. But the result of theta=1 is not as linear as theta=0.5 performs.

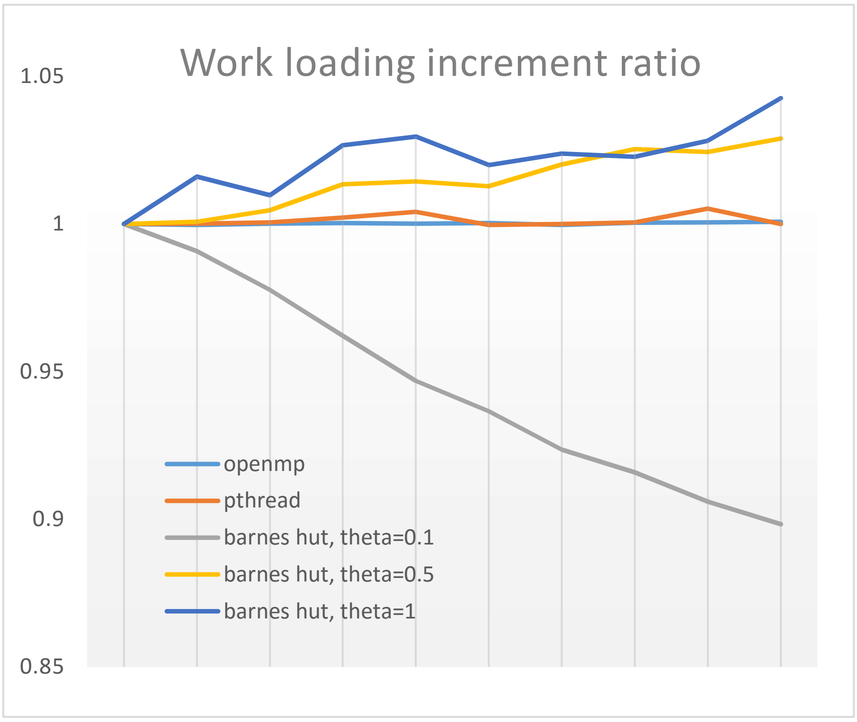


(b) Weak scalability

i. Fix N, #threads, and manipulates T. Compare execution time of each version.



We can observe that the lines are all growing linearly. However, if we look inside and take a minor view, there are some little difference.



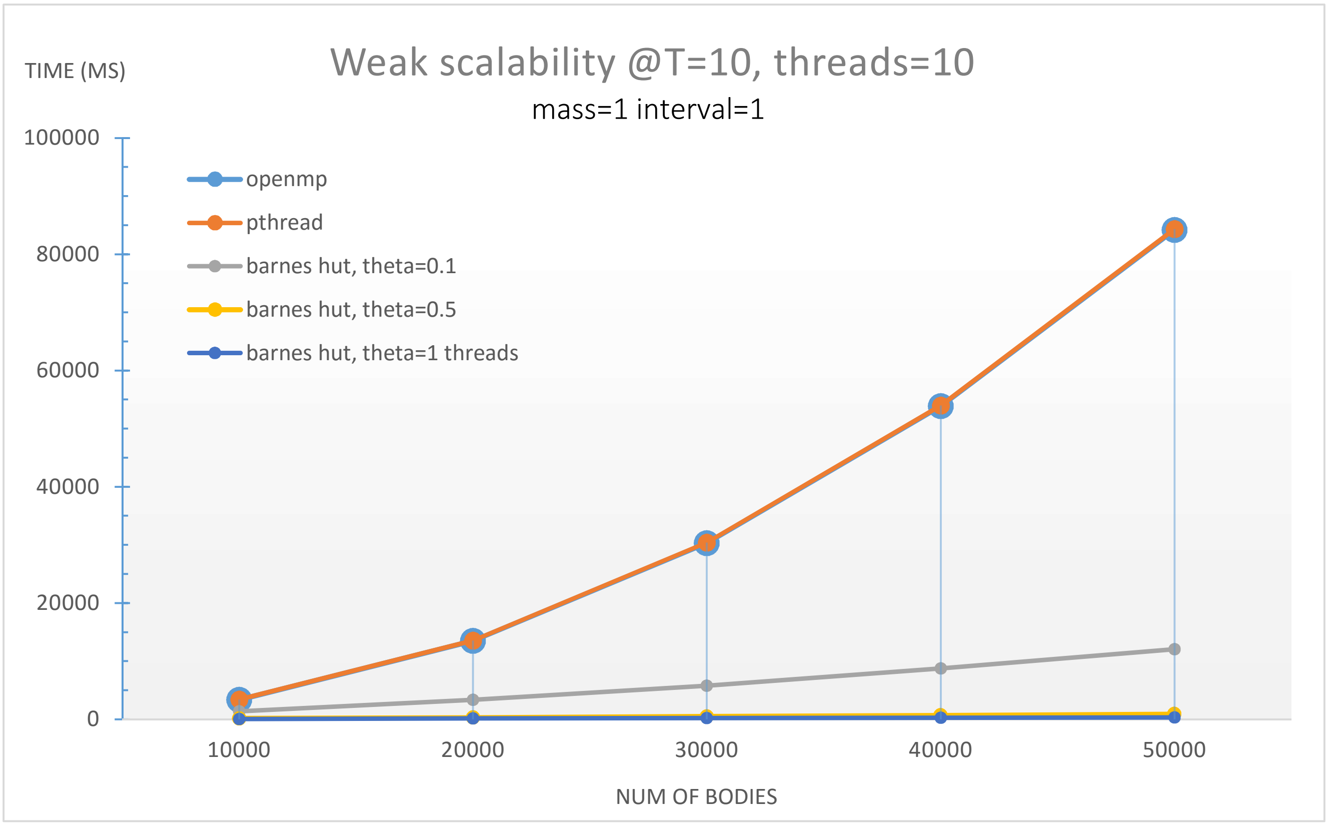
I make a function, $\frac{\mathrm{Time}(N\times100)}{N\times\mathrm{Time}(100)}$ to show how close the work load is than N times of first one.

● The pthread and OpenMP versions get this ratio close to 1.0.

● Barnes-Hut with different theta have dramatically results. With theta=0.1, the early job is heavy so the later ratio will decrese; yet with big theta will reduce large amount of works in early time, but later results won't get great speedup as its early time.

- Two of brute force multithreading versions are almost strict linear increment.

ii. Fix T, #threads, and manipulates N. Compare execution time of each version.



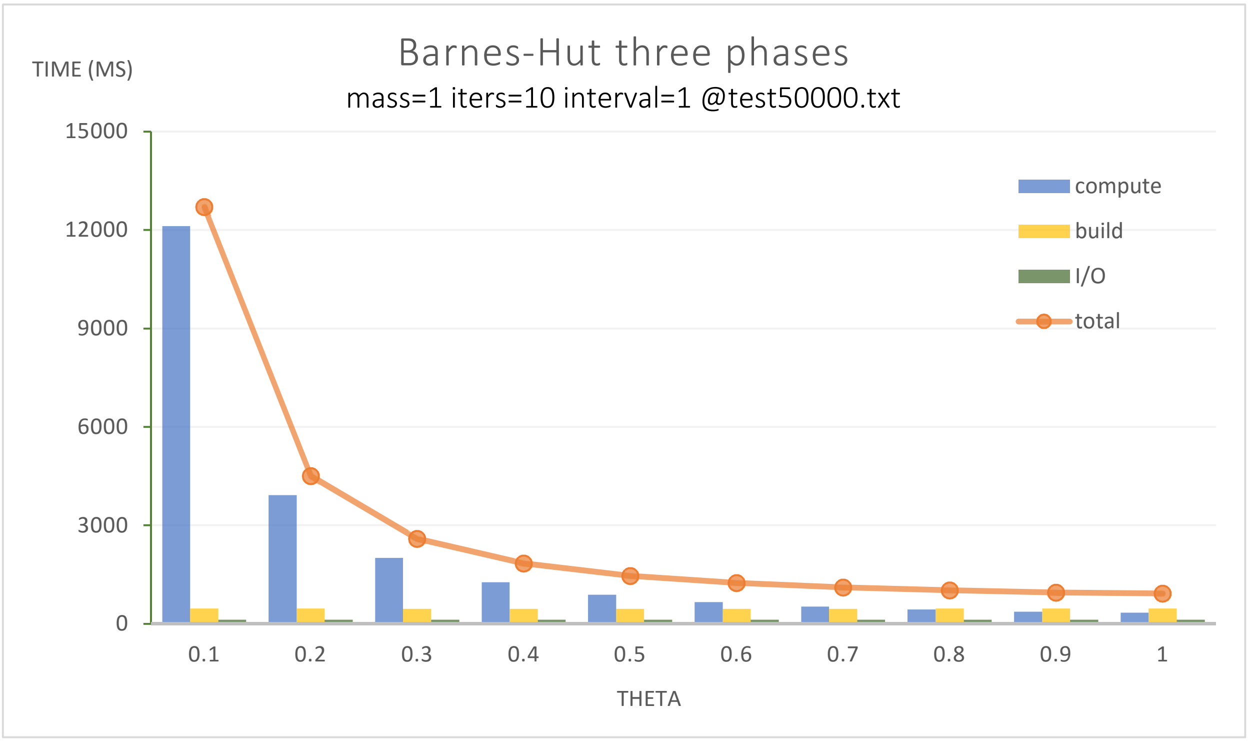
In this experiment, I generates the bodies with some mathematical functions and which is to make the position of bodies in two-dimension surface distributed uniformly. With uniform distributed testing data, we can make the Barnes-Hut algorithm implemented with simple quadrants cutting version runs in a better and close to average case.

With average case, we can see that the cost in OpenMP and pthread of brute force grows over linear increment; while in Barnes-Hut algorithm can ease the growth rate by the approximation with different theta. The larger the theta is,

- the more bodies group will be reduced into single giant body
- and furthermore the point distributed uniformly so the spreading of groups will be good for the algorithm.

All of above make Barnes-Hut approximation algorithm more competitive than brute force implementation. From the graph, we can obviously find that it's a great choice to use this algorithm for a brief view of n-body simulations!

(c) For Barnes-Hut algorithm, Show the time of three phases (I/O, building tree, and computing) based on different θ



In this experiment, I use the 50000 bodies described in previous section in which the bodies is uniform distribution and iterates them in 10 rounds. The phase of computing decreases visibly as the theta get greater. The speedup with theta is strongly related to the distribution of bodies, in this case we assume that the spreading of bodies is perfect, and we can see the decreasing curve is exponential.

(d) Other experiments you have done to compare the performance.

4. Experience

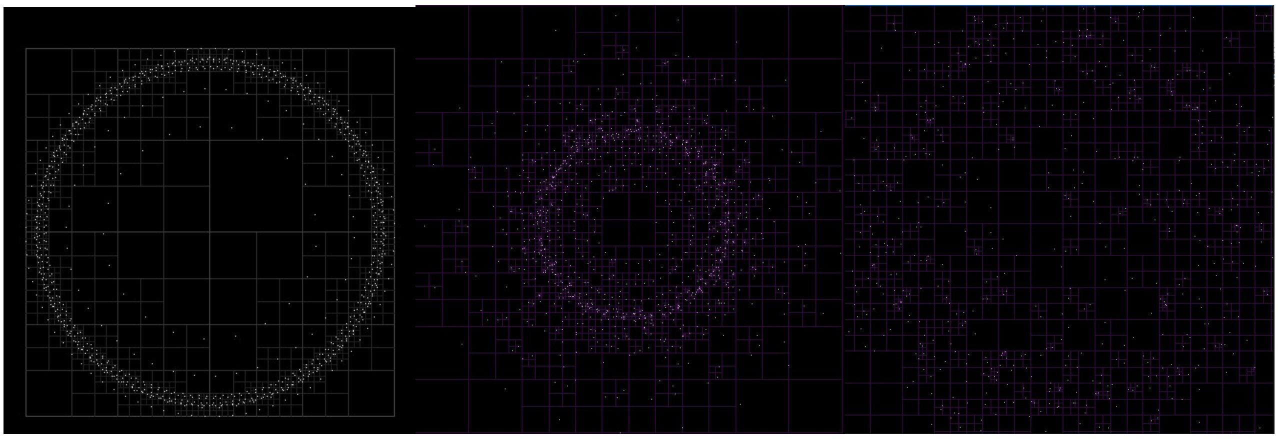
(a) What have you learned from this assignment?

Its' easy to develop multithread versions if you've done the basic sequential version. Also from the experience in previous homework, I know that the reliable basic version is an essential element if don't want to debug in these parts when other code, threading, displaying, etc., has been added onto it and makes the debug procedure more terribly complicated. So I started my basic implementation of sequential brute force version very early and have it through many tests in almost half my developing time. Finally, I find that it's worth! I finish two multithread version, OpenMP and pthread, in half an hour!

Another hard time experienced is in building a tree in Barnes-Hut algorithm. I tried to make it into an object with more object-oriented properties. This tree, namely quadtree, I have made lots of works on how to make insertion of each body correct and how to create quadrants effectively and won't cause memory leaks while the tree nodes are deleted in every iteration round. Now I learned more about how to create a data structure in good manner and learned many optimization skills in programming.

Function *pow()* in C math library is very slow in easy square or cubic computations. I finally use: x*x to get square of x, x*x*x to get cube of x and use *sqrt(x)* instead of *pow(x, 0.5)* to get square root of x, and the result is: it has almost 200% speedup.

(b) What difficulty did you encounter when implementing this assignment?



Above are the snapshots from program execution, first one is test case 2 on TA's sample program; the later are my previous program's running results. We can see the results are hardly saying that they are the same and this made me very disappointing. So I started checking my algorithm in building tree, calculating force on each body and pointer usages in program, however I got nothing to make it run correctly. As I gave up and told the situation to my friend, he said that: you may try to add some small enough number to all the divisions. It works! Then I know that the divisors in this program might always very small, and due to this it let the division result in *double* precision computing fails. So the solution to this is to add a positive small enough number onto division result.

(c) If you have any feedback, please write it here.

Could TA make your implementation code public after deadline? I'm wondering how to make it run effectively and won't make the display flicker.