Project #3: Barnes-Hut N-body simulation

Introduction

This project implements the parallel version of the Barnes hut nbody simulation using a quadtree in sequential, basic parallel, and work-stealing refinement. The simulation works as follows:

1. The program generates an input setting of the particles (2D: defined by positions, velocities, and mass). This may be completely random, in a circular pattern, or orbital type with multiple particles revolving around a central particle.
2. The program now runs for the given number of timesteps. At each timestep, it builds the quadtree for the current setting. It then calculates the force on each particle to update the velocity of each particle and then updates the positions of the particles. All this is done in parallel for the parallel and work-stealing implementations.
3. Once all the iterations have been completed, the result is stored in the output file particles.dat, which can be used to generate GIFs of the simulation.

The code base consists of the following packages:

- nbody.go: the main program that carries out the simulation and stores the output
- qTree: contains the quadtree implementation used to build the quadtree from the array of particle positions. The quadtree helps calculate the force on a particle efficiently using the center of mass approximations
- barrier: implements a barrier on the go using conditional variables. This is used for each iteration between calculating the force on each particle and updating the positions of all particles
- bdequeue: Implements the lock-free bounded dequeue. The threads use it to take tasks for processing and steal tasks from other threads
- worksteal: Implements the workstealing refinement. Each thread runs separately using its separate queue, until it becomes empty, and then it steals tasks from some other non-empty queue using the critical Balance method

Input Data

The project does not have any pre-defined data input. It generates seeded random input data of particle positions, velocities, and masses. Also, the implementation provides a pseudo-random particle initialization function circular and orbit, which puts particles in a random circle and in a random orbit around an object to get a more natural depiction of gravity on particles. Users can switch between the type of initializations using the command line arguments.

Output Data

The output data is stored on the particles.dat file if file writing is enabled through the command line argument. The file stores the data of all particles at each iteration. The particle output data can be converted into a simulation GIF by running the plot.py file.

Running nBody.go

Command:

go run nbody.go [nParticles] [nIter] [theta] [init_type] [run_type] [ntheads] [is_benchmark] [file_output]

- nParticles: specify the number of particles
- nIter: specify the number of iterations
- theta: specify the approximation parameter (0.0 -> no approximation [slow], >1.0 more approximation [faster]
- init_type: type of initialization:
    - 1: random position velocities and mass
    - 2: in a circle
    - 3: in an orbit around a central big particle
- run_type: specify the system:
    - 0: sequential
    - 1: parallel (without work stealing)
    - 2: work stealing parallel
    - 3: test all the above and check the correctness of the output
- nthreads: specify the number of threads
- is_benchmark: set to 1 to suppress all print statements for automated benchmarking default 0
- file_output: set to 1 to generate an output file. Not available if is_benchmark is 0

Example:
Run:

```
$ go run nbody.go 9 100 0.00 3 2 16 0 1
worksteal : nthreads =  16
iteration: 100
0.217333
```

Give the output in particles.dat which can be plotted using "python plot.py" to give plot "proj3/nbody/result_gifs/work-steal-BH-simulation-.gif"

Running Tests

Make the run_type = 3 to test all the implementations on the same input. For example:

```
$ go run nbody.go 10000 1 0.00 3 3 16 0 1
test

sequential begins
seq
iteration: 1
```

```
sequential took 7.027249 seconds

parallel begins
parallel : nthreads =  16
iteration: 1

parallel took 0.933895 seconds
Parallel - CORRECT

work steal begins
worksteal : nthreads =  16
iteration: 1

workstealing took 1.326536 seconds
Work steal - CORRECT

Total time taken = 9.496225
```

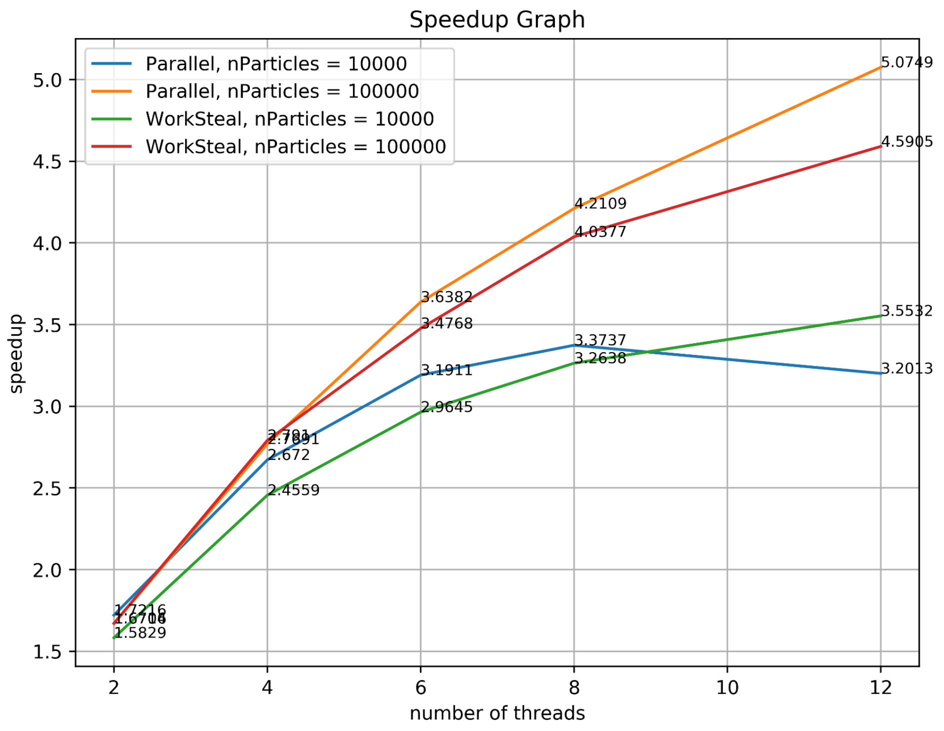
Running Tests

To run the automated test for producing the speedup plots, go to the benchmark directory to run the speedup-slurm.sh file. Run using the command **sbatch speedup-slurm.sh**

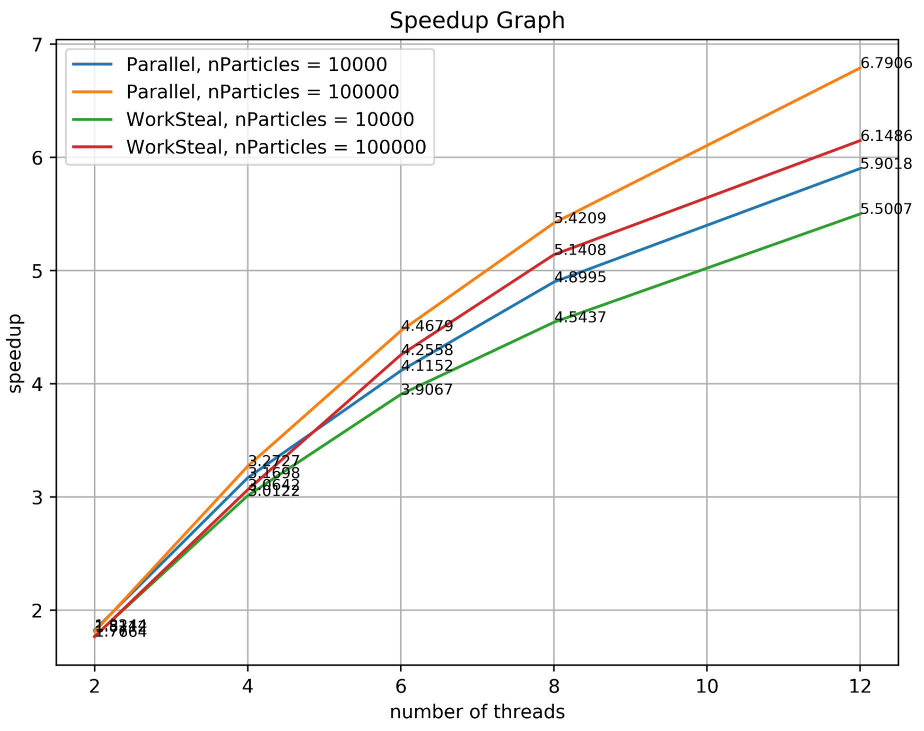
The speedup-slurm.sh works as follows:

1. Runs the nbody.go script to get the running times for the different configurations.
2. First, the script runs the sequential version. All configurations run three times (not five as the times are very consistent), and the average is taken as the overall runtime.
3. Next, the script runs the parallel and work stealing refinement version for 2,4,6,8 and 12 threads. All configurations are again run three times, and the average is taken as the overall runtime.
4. All the above output is stored in a slurm output file in the slurm/out/ directory. Now, we run a python file **bench.py** that parses the output file to get the necessary data to plot the speedup graph. The program then plots the graphs using matplotlib plotting library and saves the output graph in the speedup-graphs directory. All output graphs have the slurm job id prepended to their name.

The outputs of Slurm used to write this report are provided in the repo for reference.



Speedup graph for 1 timestep (iteration)



Speedup graph for 10 timesteps (iterations)

Observation and Plots

We have 4 different speedup plots. We run the nbody.go for 10,000 and 100,000 particles for all systems (sequential, parallel, and work-stealing). We can make the following observations:

- The implementation gives us a significant speedup from 2 to 12 threads. The speedup is almost linear, thereby showcasing good parallelization of the simulation.
- The higher number of particles gives a better speedup than the lower one. This can be because the greater number of particles can take advantage of parallelization better and compensate for the overhead of parallelization better.
- The work-stealing implementation performs better than the simple parallel version for a single iteration. This is expected for the Barnes hut implementation as for some force calculation, the function has only to take the center of mass of a large number of faraway objects, and hence, there is a lot of variation in the running time of force calculation for each particle. Hence, more often than not, some threads finish way earlier than others and then the ability to steal works helps distribute the tasks evenly across threads. Thereby giving better performance.
- Workstealing slows down the speedup for multiple iterations/timesteps. This must be due to the added overhead of creating and population the queues at each timestep.

Questions

- Details of parallelization and why did I choose BSP pattern?

  Each iteration has three steps: build qtree, calculate the force on each particle, and update each particle. The force calculation and particle update is parallelized in each iteration. Load balancing is essential in force calculation because the time to calculate the force on a single particle can vary greatly depending on its position and configuration. Hence, threads finish their current queue of tasks at different times. Thus, for the earliest finishing thread, it can steal work from other threads, improving performance. Also, the update particle step can only take place after the force calculation is done, and thus, a barrier is needed between the two steps.

- Describe the challenges you faced while implementing the system. What aspects of the system might make it challenging to parallelize? In other words, what did you hope to learn by doing this assignment?

  The biggest challenge was implementing a lock-free bounded dequeue, as it is essential to performance improvement but must be carefully implemented to avoid race conditions and ABA problems. The implementation is similar to the implementation provided in the textbook. However, several Java functionalities are not present in go and had to be improvised. The reference-stamp for the top variable is implemented using a single int64, with the first 32 bits being the reference and the last 32 bits being the stamp. In addition, as the simulation consisted of many individual particles and multiple iterations, it was a challenge to implement the parallelization correctly and avoid deadlocks and race conditions.

- Did the usage of a task queue with work stealing improve performance? Why or why not?

  As stated earlier, the task queue did improve the performance. As most force calculations take vastly different amounts of time due to approximating the center of mass in Barnes hut, some queues finish much earlier. Without work stealing, they would just wait for other threads to finish and reach the barrier, which is wasteful. So in work stealing, once they finish their work, they steal others' work, leading to better load balancing and hence better performance.

- What are the hotspots (i.e., places where you can parallelize the algorithm) and bottlenecks (i.e., places where sequential code cannot be parallelized) in your sequential program? Could you parallelize the hotspots and remove the bottlenecks in the parallel version?

  Hotspots are the force calculation and updating the particles after the force calculation. Bottlenecks are the actual building of the qtree at the start of each timestep iteration and the individual timesteps. The hotspots were parallelized in the parallel and work-stealing implementation. However, the bottlenecks remained as it is. This is because qtree building depends on the relative position of particles in the instance, and the timesteps are naturally sequential and cannot be parallelized.

- What limited your speedup? Is it a lack of parallelism? (dependencies) Communication or synchronization overhead?

  The speedup was limited by the bottleneck of sequentially doing the timesteps and qtree building at each timestep. Moreover, the work steals overhead of making the queues and populating them at each timestep harms the program's performance for many particles and timesteps. This can be observed from the graphs above and the following outputs where work steal took much more time:

```
$ go run nbody.go 1000 100 0.5 1 1 16 0 1
parallel : nthreads =  16
iteration: 100
5.395218 s
```

```
$ go run nbody.go 1000 100 0.5 1 2 16 0 1
worksteal : nthreads =  16
iteration: 100
8.952452 s
```

  Moreover, the balancing of queues takes place in a lock, which lowers the performance.

- Compare and contrast the two parallel implementations. Are there differences in their speedups?

  There are apparent differences wrt the speedups for varying numbers of particles and iterations. All this has been discussed in the observations and the above questions.