ID1217 Concurrent Programming

Programming Project

Parallelize Particle Simulation

Saurav Puri
sauravpuri200@gmail.com

March 27, 2016

1 Objective

The main objective of this project is to develop parallel applications in shared and distributed memory models and practice improving the performance of parallel applications. The task is to parallelize a particle simulator that shows the interaction of numerous forces on a select particle. The particles are only affected by the neighboring particles within a certain range. The density of the particles is set sufficiently low so that given n particles, only $O(n)$ interactions are expected. The algorithm is given; however, the implementation runs in $O(n^2)$. The goal of this project is to create a sequential program that runs in $T = O(n)$ and parallel programs using Pthreads, OpenMP, and MPI that run in T/p when using p processors.

2 Environment

This project was made on Mac iOS using XCode and ran on Scientific Linux 6.7 Carbon using a remote server connection to home university. The runtime values were collected around similar time, to minimize remote machine error.

3 Approach

My approach to this task was to first, understand the provided code and understand the algorithm used to perform the task in $O(n^2)$ time. I began with the serial implementation because serial would be the most basic task to tackle. The serial algorithm checks every particle for interaction with the selected particle. Due to the small density, only the neighboring particles have an impact. So this is where time can be saved.

I used an ADT to index through all the particles to find which particles would have a force on the current particle selected.

3.1 ADT

The ADT class holds references to all the particles. The structure creates a grid and one particle is put into each cell. The ADT class has an insert function that implements this. The grid can now be divided into larger neighborhoods. Within these neighborhoods, the relevant particles would have an effect on each other. Now, it becomes easy to determine which particles are close enough to have a force on the current particle. The indexing through neighboring particles can be done in $O(n)$ time for every particle because:

- Indexing is constant
- The backend vector class *push_back* function is constant
- There are finite interactions on every particle

Therefore, iterating over all the particles in the ADT can be done in $O(n)$ time.

For all of the algorithms, I inserted all the particles into the grid and then distributed the particles to different threads/processes.

3.2 Shared Memory Implementation - Pthreads

The Pthread implementation of the algorithm required a barrier to synchronize the multiple threads. The same barrier could be reused multiple times. By using a barrier, the program is segmented into three portions

- forces are calculated for the particle interactions,
- then, the particles are moved,
- finally, the particles are stored to their new position on the ADT by the main thread, thread 0.

3.3 Shared Memory Implementation - OpenMP

The OpenMP implementation of the algorithm required the main thread to create and allocate work to other threads. Each of the portions mentioned above requires its own for loop and the corresponding parallelization. There is no need to add a manual barrier in OpenMP because after each for loop is parallelized, there is a implicit barrier.

3.4 Distributed Memory Implementation - MPI

The MPI implementation had one key design choice. I chose to implement the parallelization to resemble a shared memory model. By doing this, the design matches very closely to the other 2 parallel implementations. The main process initializes all the particles and broadcasts this on a shared "bus." Each process then takes these particle locations and inserts it onto its own grid. Then the process calculates the forces on its own particles based on the neighbors, as done in the other two implementations. Then the process broadcasts its own new updated location and gathers information about all the other particles from the other processes from the same shared "bus."

4 Analysis with Graphs

The Figures below shows the runtime of the serial, Pthread, OpenMP, MPI implementations. The plots show that the implementation runs in $O(n)$.

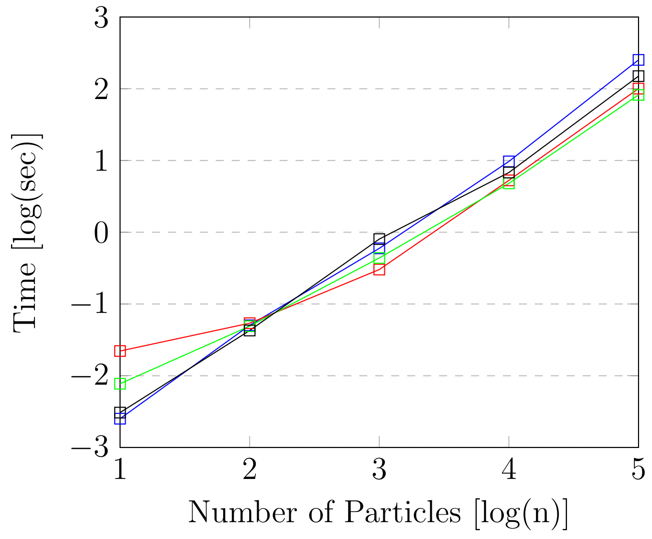
🟦 Serial

🟥 Pthreads

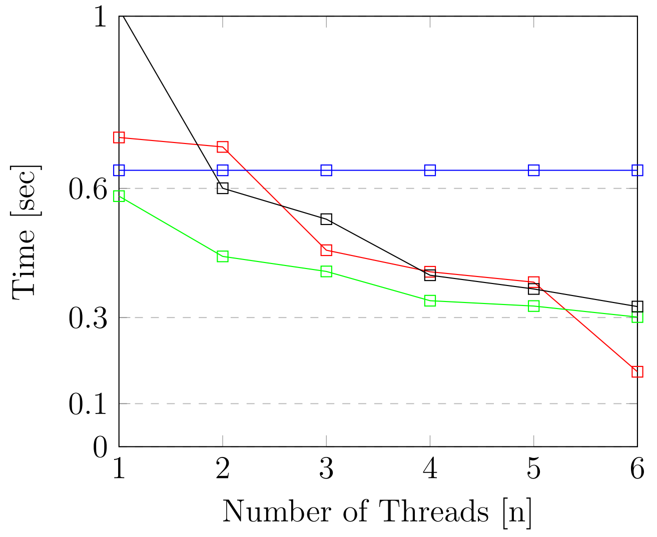
🟩 OpenMP

⬛ MPI

Runtime for Serial, Pthreads, OpenMP, MPI [default 2 threads] vs. Number of Particles



Runtime for Serial, Pthreads, OpenMP, MPI vs. Number of Threads[1000 particles]



By analyzing the graphs, it is clear the algorithms run in $O(n)$ and it is also evident that performance of the parallel algorithms could be better. The initialization overhead for OpenMP and MPI surpass the serial speed, while Pthreads does not have too much overhead but the speedup increases much more slowly.

5 Conclusion

As seen by the graphs in the previous section, the parallelized code does not reach the idealized T/p runtime. The main reason for the lower speedup would be optimizations in the code and additional added overhead when trying to parallelize the algorithm.

In the shared memory model implementation, the insertion of particles onto the ADT is done serially, because, this was the most obvious solution to change the $O(n^2)$ runtime to $O(n)$. However, the insertions could have been done in parallel using concepts similar to matrix multiplication, where each thread takes a column or row. By parallelizing the insertion, the algorithm would require another synchronization point, so that no thread begins processing forces before all the points have been inserted to the grid. But, this synchronization not be as slow as serially entering each particle one by one.

The distributed memory implementation is non-resourceful when implementing the communication background. In my implementation, every particle is sent to every process, which is definitely not necessary. This will definitely increase the runtime, but would still be able to run bounded by $O(n)$. A better method would be to calculate which particles are needed per process; but that would require a significant amount of calculations and error checking, and could also be costly. This is very similar to the matrix multiplication concept in ideals, but in practice very different to implement.

The ADT implementation succeeds in running the algorithms in $O(n)$ time and works fine to reduce the runtime from $O(n^2)$. However, the ADT structure may not be the best idea to achieve ideal parallelism of the algorithm. This is because, creating the ADT structure will always take $O(n)$. Data structures like AVL trees and Binary Search trees can insert values in $O(log(n))$ and search for values in $O(log(n))$.

From this project I learned the importance of viewing algorithms in a whole different manor when implementing it in parallel. I also learned that the best way to achieve maximum parallelism is by choosing the correct data structure through analysis and trial-error.