Allyn Vo

Final Project

CS 473, Parallel Computing

Winter 2018

**Technical Report**

1. Problem Description

Write a parallel program in OpenMP to solve this problem:

I will be solving problem 3.15 from the book – Given an array of n records, each containing x and y coordinates of a house. Also given the x and coordinate of a railroad station. Design a parallel algorithm to find the house closest to the railroad station.

2. Design (the four steps of Foster’s methodology)

**Partitioning:** A primitive tasks is in charge of a house with its coordinates and distance from the railroad/train station. Every single task will use the distance formula to calculate its distance from the train station. A task is then contributing its “distance” towards a reduction, finding the minimum within a set. In which, the reduction will find the minimum distance of all the houses with its “house ID”.

**Communication:** The pattern is a binomial tree, in which the reduction would produce in logarithmic time, since the number of tasks will be split in half each time.

**Agglomeration:** The agglomeration approach is to combine tasks to be as evenly distributed as possible to the number of processors. So, each agglomerated task will find the closest house out of its group of houses. Then the reduction will continue with those closest houses. (Assign about n/p houses to each task p)

**Mapping:** There’s a static number of tasks, a structured communication model, and a roughly constant computation time. So, using the “Mapping Decision Tree”, we will agglomerate tasks to minimize communication and create one agglomerated task per processor/thread.

3. Source code

(Attached)

4. Benchmarking, with different values of n and p, where n is the input instance size and p is the number of processors (in case of MPI) or threads (in case of OpenMP and CUDA).

Benchmarking in OpenMP, using visual studios 2015, on Mac book pro 2017 through “Parallels” to run a Windows VM:

**Execution Time (sec) for 100,000 Houses. (10^5)**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Thread(s) | Run in sec. | | | | AVG |
| 1 | 2 | 3 | 4 |
| 1 | 0.028702 | 0.021702 | 0.024056 | 0.021821 | 0.0240703 |
| 2 | 0.015794 | 0.012939 | 0.018311 | 0.009836 | 0.0142200 |
| 4 | 0.010733 | 0.014277 | 0.014290 | 0.010330 | 0.0124075 |
| 8 | 0.022183 | 0.012155 | 0.011500 | 0.136570 | 0.0456020 |

**Execution Time (sec) for 1,000,000 Houses. (10^6)**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Thread(s) | Run in sec. | | | | AVG |
| 1 | 2 | 3 | 4 |
| 1 | 0.217416 | 0.213162 | 0.229735 | 0.235378 | 0.2239228 |
| 2 | 0.182905 | 0.160206 | 0.196680 | 0.132186 | 0.1679943 |
| 4 | 0.161684 | 0.129336 | 0.116278 | 0.116214 | 0.1308780 |
| 8 | 0.139291 | 0.159803 | 0.112092 | 0.113745 | 0.1312328 |

**Execution Time (sec) for 10,000,000 Houses. (10^7)**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Thread(s) | Run in sec. | | | | AVG |
| 1 | 2 | 3 | 4 |
| 1 | 1.959199 | 1.909093 | 1.905372 | 1.884271 | 1.9144838 |
| 2 | 1.070537 | 1.009441 | 1.014733 | 1.016092 | 1.0277008 |
| 4 | 1.095027 | 1.033516 | 1.016695 | 1.020478 | 1.0414290 |
| 8 | 1.080045 | 1.015054 | 1.021603 | 1.003794 | 1.0301240 |

For my benchmarking, I had three general tests for a different number of houses. Each general test ran using 1, 2, 4 and 8 threads. And with each number of thread, there were four trials, which then averaged out.

I chose to test these numbers of houses because any lower would show very small numbers and wouldn’t show a lot of change, for it would just be very fast. Any higher and the results will be almost duplicates of the last, and wouldn’t show much difference for each number of threads.

In the program, all of the coordinates for each houses and railroad were randomly generated. Essentially using randomized input or fixed input showed very similar results in my program.



5. Theoretical efficiency analysis (parallel vs. sequential asymptotic execution time)

When the number of threads is 1, it is running sequentially, while 2, 4 and 8 are running in parallel. From sequential time which is big O of n (linear), to parallel time, should change from linear to logarithmic time (O(log n)). This happens so because n number of houses should be split about evenly across p number of threads.

When program runs sequentially, as the problem size increases (number of houses), the time to execute takes much longer compared any other same program running in parallel. From running with 1 thread to two threads, the execution time nearly halved for each number of houses.

What’s odd is that running 2 or 4 threads seems to be the best choice overall, since any increase in number of threads after seemed to taper off and had little change are fluctuated from even being faster or slower than the lower number of threads. Even though the graph and tables seem as if there was no significant increase at all after using 2 threads, it may be an illusion since the time intervals were so small, and the tests could have gone through more trials to reduce errors or outliers. Another thing to note is that since the number of houses was so large, even when splitting the houses between different threads would be a huge number. And to add on, each thread still computes so fast still in a short amount of time to really “see” the change. Lastly, just spawning up the threads as well will add a fixed amount of time and as well as assigning sections of number of houses.

Overall the efficiency of the program seemed to be best around 2-4 threads according to these tests, but results may show otherwise if able to use a larger number of threads to actually see what will happen. Because in theory, with more processors/threads, the number of houses would be split into smaller chunks significantly. So, it still would be scalable.

However, with increasing threads, it may also be that communication, transition, and communication between the threads could be taking up a fixed amount of time so when the program seems to be approaching it’s limit for speedup and efficiency, this could be the problem.

6. Discussion.

In my own experience from doing this, I think that this problem was too simple to try to create opportunities for speedup of the program, in part of there not being a lot of areas for parallelism other than the loops itself. And even in the loops, there’s not really any variables being restrained. I think something simple can only go so far in terms of trying to complicate something that is simple.

At the start, I would’ve thought that the program can get way more out of problem, but I also think there’s other things that I may be missing out due to software limits and hardware limits of how and what I used to get these results. But this also may be the result of Amdahl’s law where eventually, the speedup will taper off even with more processors.

Another thing that could be implemented directly with the parallel for statements are scalar expression which would only run in parallel depended on an if statement since there are certain thresholds where it is not worth it.

To conclude, I did try testing out the programs in the book as well with the results available from the author, and my reductions did not match the speed up of the books. So, I am not sure if something is wrong or it’s just because of the machine difference. For example, I tested the example from chapter 17.2 in the book about reductions, and the results I got showed a trend like my reductions on houses from above.