CIS*3090 – Assignment 4

Impressions Working With Java Threading

My impression of working with Java threads was positive. Since I opted to take the more classical pthread style of threading, I didn't find working with Java threading to be a harsh change from working with pthreads in C. Personally I like the object oriented paradigm more than the procedural paradigm, so I really liked how each worker/thread sat in its own dedicated class, and was able to hold all its subtasks encapsulated as methods.

However, I disliked being tasked with writing fast performing code in Java. I disliked being tasked to write a fast performing Java program because of a combination of two reasons. The first reason being high performance Java code would look similar to c code, but perform worse. The second reason being that writing Java code that looks like c code, nullifies a huge number of the reasons to use Java in the first place. I think c is a much better language to write Data-Oriented design programs, or otherwise high performance programs.

I suppose if your goal is absolute portability, or to make the program scalable, without needing it to be absolutely as fast as possible, Java threading probably wouldn't be a bad choice. In fact, in the case of not needing it to be absolutely as fast as possible, Java threading would probably be a great choice. Java threading would likely offer a program that is easier to maintain than either pthreads or OMP, while stilling offering multithreaded scalability that would allow the program to tap into Moore's law for years to come. I'm thinking this would align with the goals of large business applications, much more so than things like scientific or other high performance computing needs.

While my experience with Java threads didn't vary harshly from pthreads and OMP in c, I did find programming the "scaffolding code" that the red/blue simulation required much easier with Java. Building the scaffolding code using Java was easier than using c, simply because Java does more for you, especially in regards to memory management. By scaffolding code I mean things like parsing and storing the command line arguments.

A self-inflicted issue that ended up impacting this assignment rather largely for me, was that I was unsure how to verify my program produced correct output for anything above a small board. I rather foolishly sat idly, waiting for a third person to agree or disagree with a medium board result that was posted to the forum. While despite being unsure of my programs output, I could have continued experimenting and improving my programs design, provided I kept the implementation simple. Which leads to my second self-inflicted issue: attempted premature optimization. Because I prematurely optimized, my program felt much less nimble to move in the direction of promising performance increase, and yet received seemingly no actual performance benefit from the premature optimization. Like A2 my goal should have been: simple now, fast later if possible.

My Program's Parallel Architecture and Work Distribution Scheme

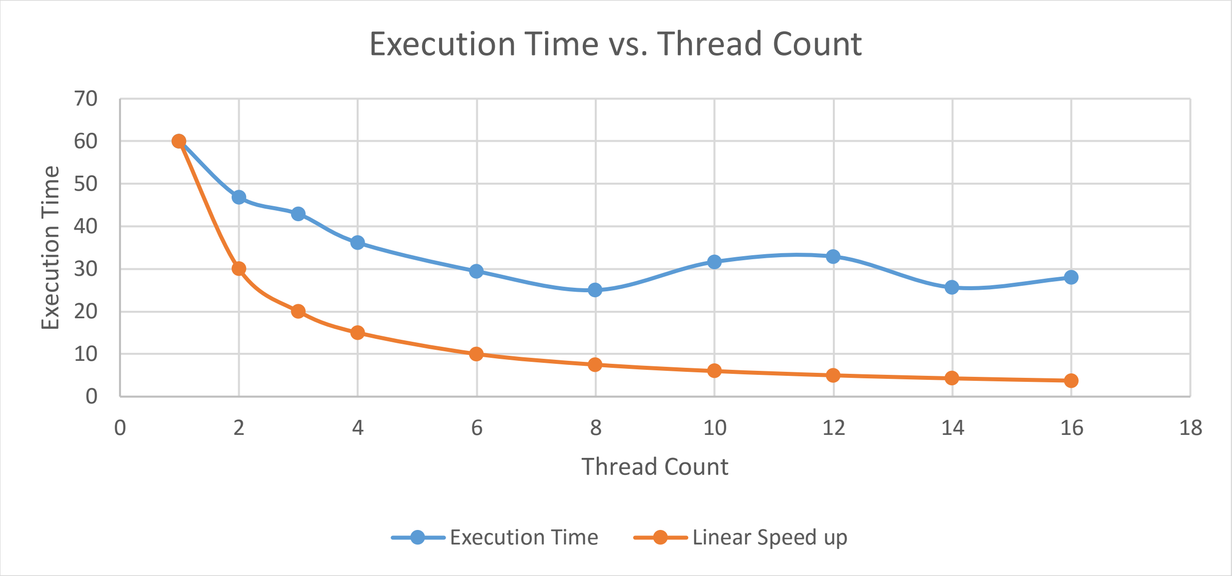
My programs data distribution of cells was a simple cyclic distribution, applied in a needlessly complex way. Each worker in my program fills two lists of cells that it is responsible to move the contents of. My program uses two similar, but different cyclic distribution methods to fill these lists. The first distribution method, is used by a worker to grab all the cells the worker is responsible for on the red stage of a step. The second cyclic distribution method is used to grab all the cells a worker is responsible on the blue stage of a step. The two cyclic distributions were used to give the first workers the excess red cells, and the last workers the excess blue cells. Excess cells here meaning the remainder after dividing the number of cells by the number of workers.

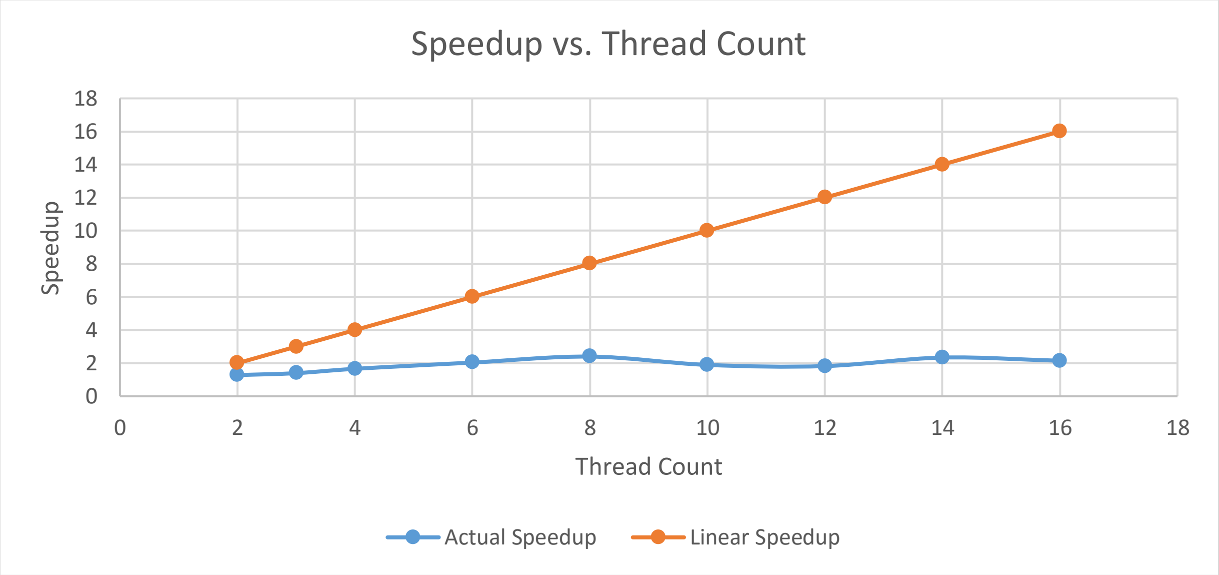
My original plan was to have the red and blue stages interleaved, so that workers that had less red cells, would begin cycling through their blue cells moving the ones that were deemed safe, in an effort to improve load balancing. This turned out to be a terrible idea, for a number of reasons. One the load imbalance between workers becomes tiny as the board grows, since any worker can only have one cell more than any other worker. A worker having 10001 cells to check isn't really a bottle neck when all the other workers have 10000 cells to check. On top of this there is overhead that occurs when trying to make a worker start the blue stage early, which makes this run slower than throwing a barrier in between the stages, and living with the slight load imbalance. A third reason interleaving the red and blue stage was a terrible idea, was the increased difficulty in debugging and modification of the program.

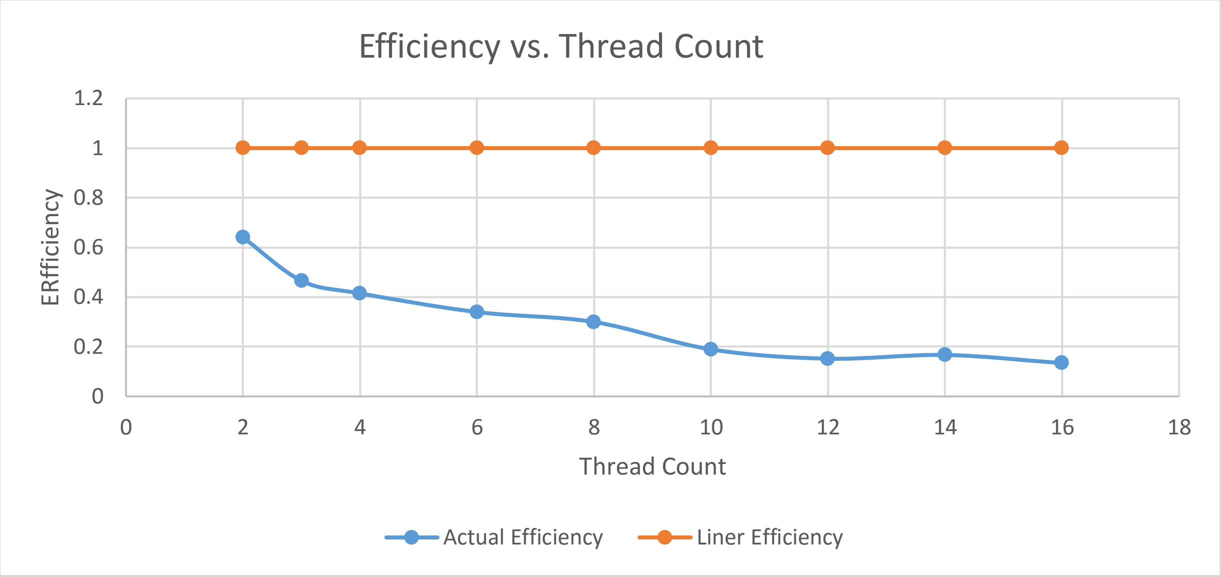
Trying to interleave the red and blue stage was a result of my premature optimization attempt, and I feel I would not have overlooked the details that show this idea to be poor, had I waited until after I had a simple working version in place.

As for the actual algorithm, I think it is about as simple as it could be. In the first part of a red/blue stage, the worker looks at the contents of each cell it is responsible for moving. If the contents of the current cell match the current red/blue stage, then the worker checks the appropriate neighbour to see if the contents of the current cell can be moved to the neighbour. If a cell's contents can be moved to its neighbour, the worker sets a flag to have the current cell emptied, and sets a flag for its neighbour to receive the same colour contents as the current cell. Then once every worker is done going through the cells they own, they go through them again, to remove cell contents marked for removal, and add contents to cells marked for addition, in effect creating the movement of the cell contents from one cell to its neighbour.

Graphs


Execution Time vs. Thread Count
Execution Time
70
60
50
40
30
20
10
0
0
2
4
6
8
10
12
14
16
18
Thread Count
Execution Time
Linear Speed up


Speedup vs. Thread Count
Speedup
18
16
14
12
10
8
6
4
2
0
0
2
4
6
8
10
12
14
16
18
Thread Count
Actual Speedup
Linear Speedup



Observations about Graphs

Something interesting I noticed about my graphs, is that the program benefits from additional threads up until 8 threads. Exceeding 8 threads is detrimental to my program, until it receives more than 11 threads to use, at which point it begins to benefit from more threads again. I'm thinking this has to do with my poor data distribution, as the number of threads increases, the distance between each cell owned by a worker increases. This is due to my cyclic work distribution. So what I'm thinking may be happening at 8 workers, is that the number of cells a worker can pull into a cache line quickly beings to approach a single cell per cache line. Once hitting 11 workers however, cache lines are already holding a single cell, so adding more workers begins to be beneficial again, since spreading out the cells a worker owns can't make the number of cells pulled into a cache line worse than one.

How Scalable was my program?

My program was not very scalable at all. It does gain significant but lackluster performance gains up to 8 cores, at which point it appears to better to stick to 8 cores max. So while it could be worth to run my program with 8 cores, I would hardly consider an 8 core hard limit scalable.

What I Would Change

With more time, the first thing I would do to improve my programs performance would be to change the cyclic distribution to block cyclic distribution. This should alleviate the issues where after 8 cores, my program not only stops gaining benefits from more cores, but actually experiences performance detriment. Like I said above, I believe this is to do with, the number of cells a worker can bring in, in a cache line. This would hopefully push the "scalability wall" of my program much closer to 16 cores. This wouldn't have been a huge overhaul of my program, however, I feel nervous making changes this close to the deadline.

If I were to do this assignment again, I would focus on simplicity like I did for A2, rather than trying to be clever before I had even fleshed out the scope of the problem I was trying to solve. It seems much easier to go from a simple program to a fast program. Than from a program with misguided optimizations, to a fast program. While I know it’s not a good excuse I would like to point out that I felt incredibly lost without a way to test the results of my program on a medium/large board. I was hoping more people would post their results on the forum to compare with.

I am curious to see how much of a benefit my program will receive from cyclic block data distribution, and I intend to implement this data distribution when I have more time and less stress. I am also still curious to the exact reason why my original setup of doing all the cell movements in a single step lead to synchronization issues, as opposed to the modified double buffer approach I use now. Especially when the single step cell movement worked fine with small boards, and mostly but not quite right for large boards.

Raw Data

Threads	Execution Time	Linear Speed up
1	60.04	60.04
2	46.88	30.02
3	42.91	20.013
4	36.16	15.01
6	29.45	10.007
8	25.02	7.505
10	31.67	6.004
12	32.9	5.003
14	25.65	4.2885
16	27.94	3.7525