Cmpt431 – A2

Rafiq Dandoo

Part 1: Characterize sequential Overhead of parallel_sort

What is the smallest grainsize that is still within 5% of the infinite grainsize?

From testing the smallest gran size within 5% of the time of an infinite grain size was roughly 625000. The speed of the sequential run with infinite grain size was 4.00245 seconds, and a grainsize of 625000 yielded a time of 4.20665.

Part 2 – Characterize Parallelism in parallel_sort

Now perform the same analysis, but with four threads. Plot the data on the same graph (normalizing the runtimes to the infinite grainsize single-thread runtime).

What is the maximum speedup on four cores? At approximately what grainsize does this occur?

The maximum speed up I was able to achieve was 2.8x from the 4.0 seconds of the infinite grainsize sequential run, to 1.42538 seconds using a 625000 grainsize on the parallel run. This most likely occurred at around this grainsize because it is the smallest grainsize value that is still extremely close to an infinite one on a sequential run, thus it doesn't lose too much time with splitting up the tasks too much on a multithreaded run.

Part 3 – Distributed Task Queue

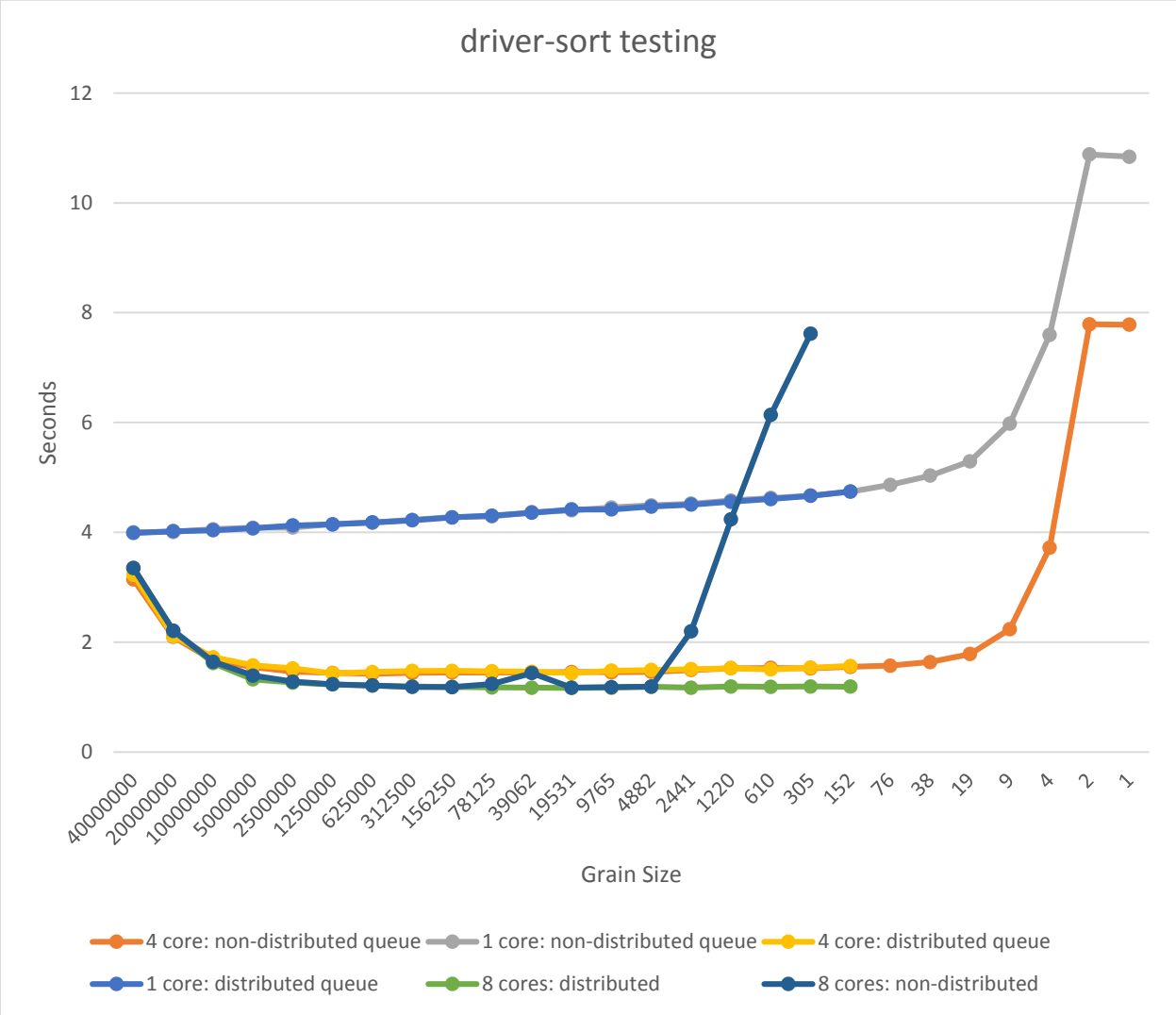
A decentralized work queue has two effects on runtime (1) reduces contention on the central queue and (2) impacts order in which the tasks execute. Is the distributed queue always faster? Why or why not. Comment on the likely relative impact of the two effects. (Basically, look at the data and some something insightful about it).

The way I planned on setting up the steal() function, was that each thread would steal from the queue with the largest amount of items in it. The reason I chose this method is because this way I felt that this way it would reduce the overall amount of steal() calls needed which would reduce further contention between threads locking each others queue. The pros of this method are that the load is more evenly balanced among cores. For example, if you had 4 threads, one with one task, two with 0 tasks, one with 10 tasks, if you were to do a random approach and have the thread with zero tasks steal from the one with 1 task, and then the next one also tries to steal from it, it would cause the first thread to have to randomly steal a task back, and it could target another thread with 0, thus a chain of waiting downtime is created in a worst case scenario.

With this method of stealing from the largest queue you eliminate that risk, but the downside might be that you have two threads going after the same queue which means one would have to wait a bit.

The data that my tests produced is a bit confusing. What it shows it that a distributed task queue essentially had no effect on the run-time. I thought this was due to faulty code on my part and it not stealing tasks from other threads, but I did do debugging and made certain that threads were stealing from each other. My hypothesis on why there was no speed up is because there was very minimal contention between four threads on the main queue, if there were something like 24 threads all competing for access to the queue, or if the threads were doing complex operations on objects in the queue, or anything that would increase the overall time the queue is locked, the distributed work queue would increase speed ups. But I think since all that is happening is queuing and de-queue.

So I tested this theory by increasing the number of threads to create more contention between them, and what I found was that on 8 cores, with a very small grain size the time differences were pretty drastic. A non-distributed work queue at a grain size of 305 on 8 cores took 7.6 seconds, where a distributed work queue with 305 grain size was took 1.18 seconds. This would make sense with a small grain size and many cores meaning lots of contention on a single queue being locked and other threads having to wait.



Part 4 – parallel_for

For this one I'm really not sure if my test data is right or not, I've tested it multiple times on my machine at home and the school's machine and the results were about the same. It also passes the assert() tests and recursively splits up tasks just as required. So I'm basing my hypothesis off what the data says and not what I think should be happening.

- **For a single thread, what is the smallest grainsize that is still within 5% of the infinite grainsize?**

  This problem ran really slow compared to the first one, so a grain size <150 was still of equal time to an infinite grain size.

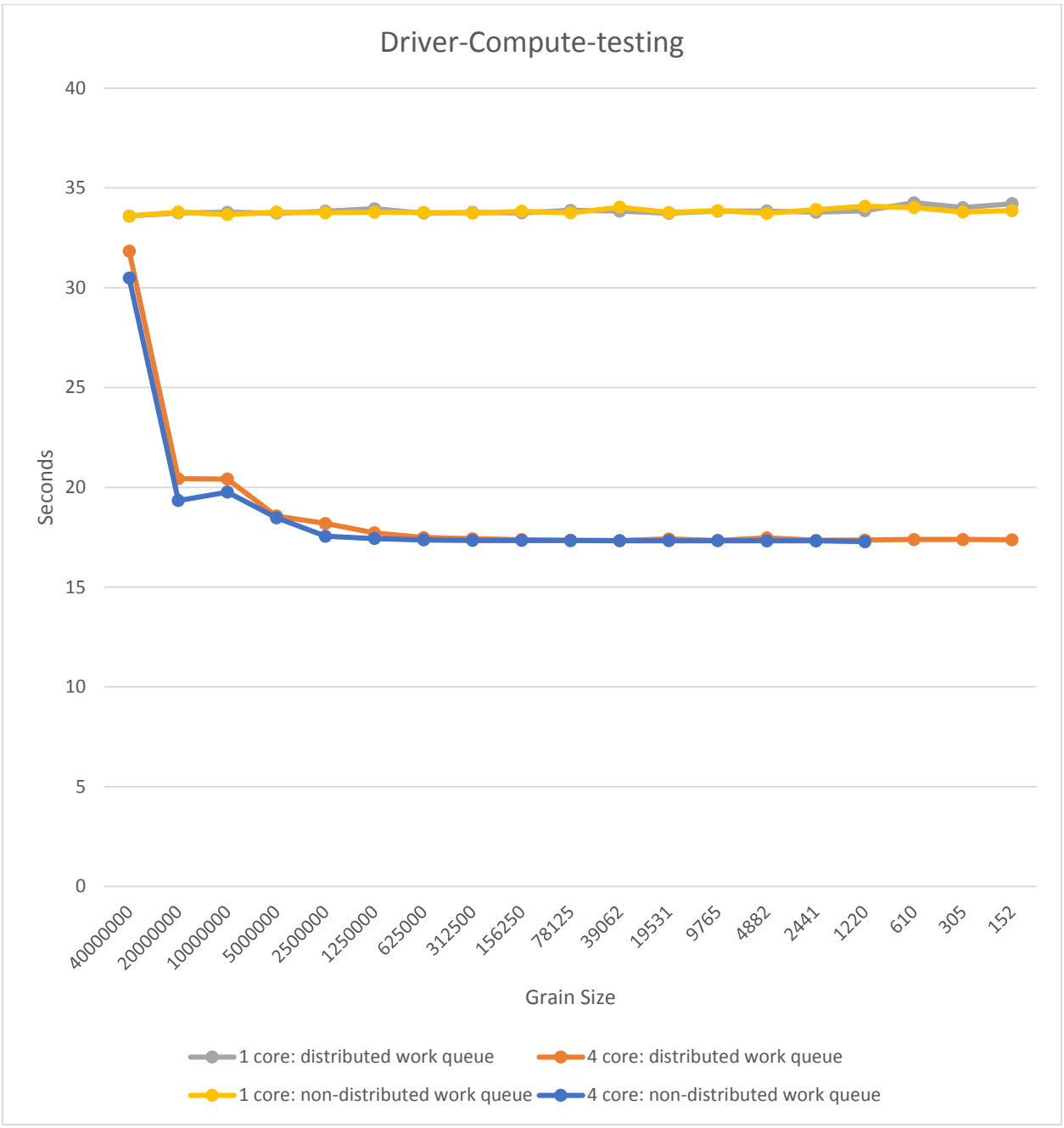
- **What is the maximum speedup on four cores? At approximately what grainsize does this occur?**

  The maximum speedup on four cores that my tests got was 1.9x , with 33.595 seconds being the fastest single core run, and 17.333 seconds on four cores. What this speed up can tell us, from what we learned in the previous assignment, is that since the max speed up is roughly 2x, on four cores, that means that quite a decent portion of the work isn't being done in parallel. This could mean that threads are blocking each other from accessing the queue and hurting the parallelism, so one result might be that distributing the work so that each thread doesn't have to content for a single lock may have big improvements.

- **A decentralized work queue has two effects on runtime (1) reduces contention on the central queue and (2) impacts order in which the tasks execute. Is the distributed queue always faster? Why or why not. Comment on the likely relative impact of the two effects. (Basically, look at the data and some something insightful about it).**

  As that data in the graph below shows, a non-distribution and distribution didn't have much effect on overall runtime. My hypothesis on why this is contradicts my earlier idea on what would happen is possibly because there exists a sequential bottleneck area in the code that doesn't have anything to do with the queue.

  One thing to note is that the speeds didn't slow down with very small grain sizes, this was most likely due to the fact that there is computation required on the items for the threads, which makes it so that the threads aren't finishing their work before their queues are being filled up.


Driver-Compute-testing
40
35
30
25
20
15
10
5
0
Seconds
40000000
20000000
10000000
5000000
2500000
1250000
625000
312500
156250
78125
39062
19531
9765
4882
2441
1220
610
305
152
Grain Size
1 core: distributed work queue
4 core: distributed work queue
1 core: non-distributed work queue
4 core: non-distributed work queue

Part 5 - parallel_reduce

Extra Credit: For the "reduce" operation, I had a version in mind that used neither locking nor per-thread accumulators. Of course, using a lock to protect a global counter for each coarse-grained task might also perform well, but it wasn't what I had in mind.

For this summing operation, the only synchronization required is a variable all the threads need to access to increment to find out the sum. The only way this is an issue is if while one thread reads in the variables value then adds it and stores it back, if between the reading and writing there is a context switch or another thread writes in a new value causing a race condition. So the most simplistic solution seems to just be reading, incrementing and writing back all in one atomic step.
So my solution to this was just to make the variable everyone increments an atomic using the atomic.h class provided and pass a pointer to all of the threads to increment.

- **For a single thread, what is the smallest grainsize that is still within 5% of the infinite grainsize?**

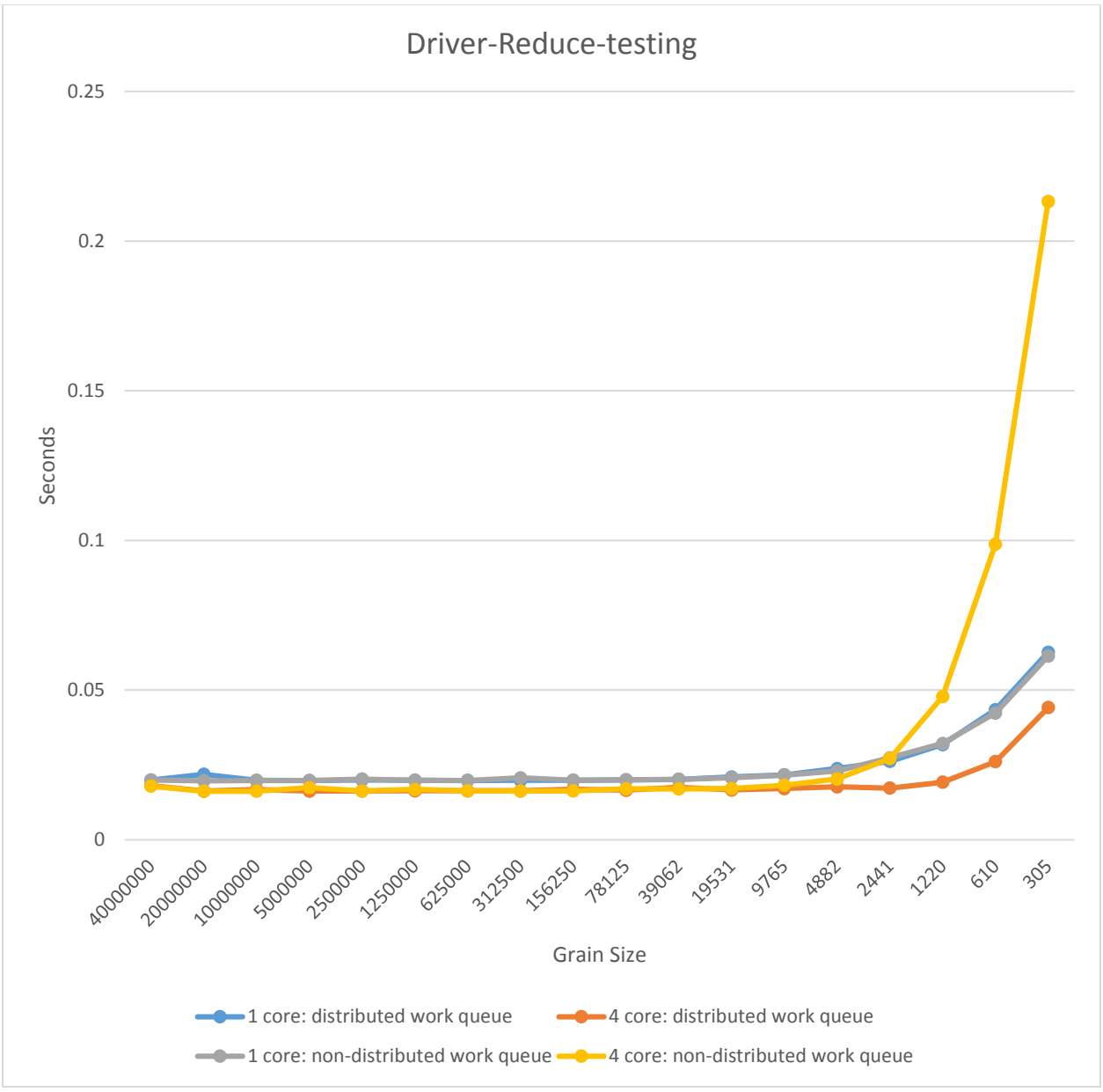
  The maximum speedup achieved on this was only 1.26x. I think this was mostly due to the fact that the task was so fast, the overhead of spawning extra tasks just outweighed the actual benefit of their parallelism.

- **What is the maximum speedup on four cores? At approximately what grainsize does this occur?**

  The maximum speed up was at a grainsize of around 5000000 on four cores. This makes sense due to it being close to being the maximum grain size where each thread will be assigned the maximum amount of work.

- **A decentralized work queue has two effects on runtime (1) reduces contention on the central queue and (2) impacts order in which the tasks execute. Is the distributed queue always faster? Why or why not. Comment on the likely relative impact of the two effects. (Basically, look at the data and some something insightful about it).**

  I think this task shows the benefits of a distributed work queue. The yellow and orange lines are both 4 cores, but the distributed work queue was much faster. My hypothesis about why this is, is because each thread completes their tasks so quickly they are going back to the queue for more jobs extremely fast, and with other threads locking it and blocking it, this results in quite a big slow down.


Driver-Reduce-testing
0.25
0.2
0.15
0.1
0.05
0
Seconds
40000000
20000000
10000000
5000000
2500000
1250000
625000
312500
156250
78125
39062
19531
9765
4882
2441
1220
610
305
Grain Size
1 core: distributed work queue
4 core: distributed work queue
1 core: non-distributed work queue
4 core: non-distributed work queue

Part 6 – Testing on other machines

All the above graphs are from tests run on my home machine, below are tests run on the sfu amoeba machines.

I tested on the SFU amoeba machines using the same scripts and I got the same results with just slower times.