Assignment #2

Implementing a Parallel Task Library

CMPT 431 Fall 2015

Graeme Smith Alysha van Duynhoven

Contents

Introduction

Our original testing occurred on the N3 machine (see Environment Specifications pg 9). For experimental data in part 6, we decided to run parallel_sort(), parallel_for(), and parallel_reduce() on the Quad Core i7 machine. We chose to normalize all of our graph data to the infinite grain size to showcase our runtime trends and events over the variety of grain sizes. Our grain sizes range from 80,000,000 down to 305, logarithmically declining to this smallest grain size.

We provide 3 test cases for each graph, as follows:

- Sequential, single-threaded with Central Task Queue, unmodified and given in the original PPPv3 folder
- 4 threads with Central Task Queue, also using the unmodified TaskQueue.C file given in the original PPPv3
- 4 threads with our own implementation of the Distributed Task Queue

Our graphs depict the normalized average data, and for each grain size, we ran 5 trials with 40,000,000 particles per test for parallel_sort() and parallel_for(). For parallel_reduce(), we opted to run with 100,000,000 particles for these tests.

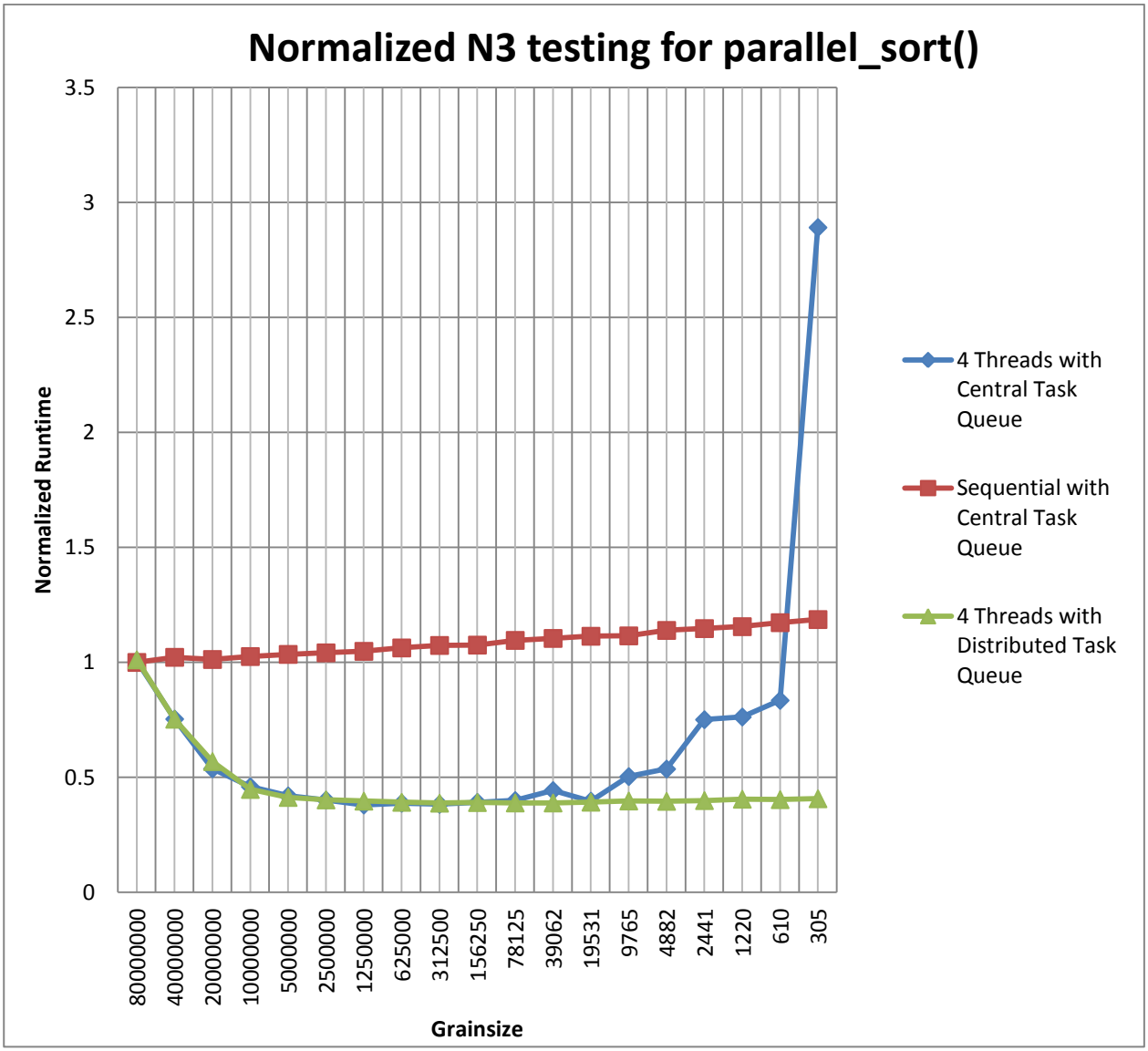

Figure 1: N3 parallel_sort() runtime

Part 1: Characterize Sequential Overhead of parallel_sort

Using the N3 Amoeba Server, the smallest grain size that is still within 5% of the infinite grain size is 1,250,000, for the sequential version with the centralized task queue.

Part 2: Characterize Parallelism in parallel_sort

The maximum speed up on four cores with the centralized task queue is at grain size 1,250,000 also, where we can see a 38% increase over the sequential version.

On the N3 machine, it is apparent that following the grain size of 19,531, the 4 thread with central task queue implementation begins to perform poorly. It even surpasses the sequential implementation after the grain size of 610. This can perhaps be a trait of an increased cache miss rate. Also, due to these independent tasks being popped off the central task queue by each thread in no particular order, there is most likely a downgrade in runtime because of the lessened locality of data accesses.

Part 3: Distributed Task Queue

A distributed queue is capable of improving performance because it can reduce contention on the central queue. However, based on our analysis it is not always faster. For larger grain sizes, the contention is not as much of an issue because the time spent retrieving from queues is small relative to the time spent executing a task, thus the overhead of a distributed queue could be larger than the gains from reducing contention. A distributed queue offers improved performance with smaller grain sizes where there are more tasks.

A decentralized work queue has an impact on which order the tasks execute, however, it is not an issue in this implementation, as all tasks created are independent. At lower grain sizes with the centralized work queue, there exists more contention as threads are all drawing from the same queue. A distributed queue works better at this level because separate threads won't all be contending for the same queue. The run time of the decentralized task queue version may also be enhanced with its locality of data accesses being maintained for longer spans of time than that of the centralized task queue. While the decentralized work queue with stealing implementation may closely follow the trend of the centralized queue graph at first, we can see the stability and less contention allow it to achieve improved runtimes.

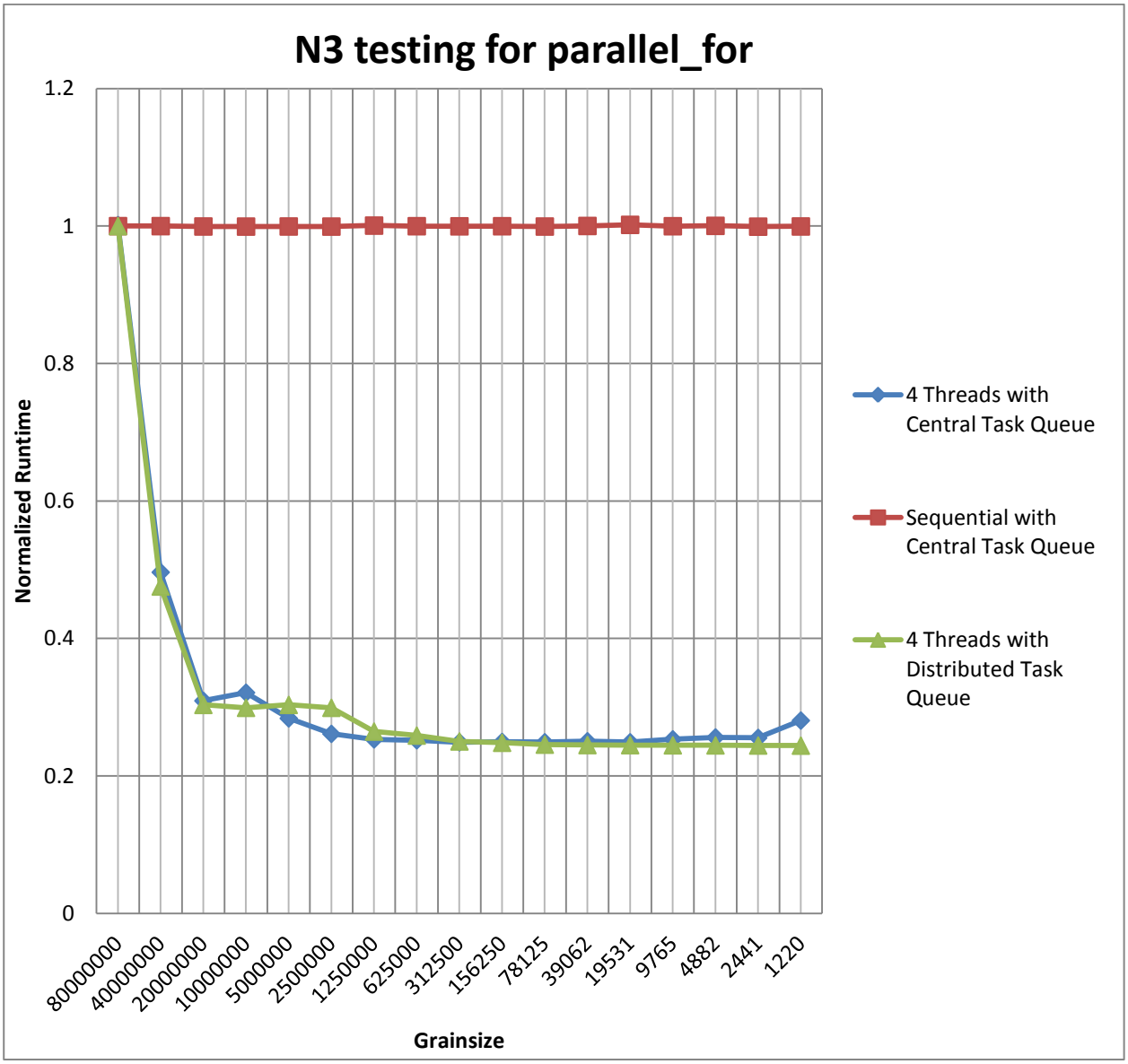

Figure 2: N3 parallel_for() runtime

Part 4: Implement parallel_for

Using the N3 Amoeba Server, the smallest grain size that is still within 5% of the infinite grain size is 1220, for the sequential version with the centralized task queue, which begins to segmentation fault following this grain size.

The maximum speed up on four cores with the centralized task queue is at grain size 312,500 also, where we can see a 24% increase over the sequential version.

The order at which these independent tasks are popped from the centralized queue will not have an impact on runtime. Utilizing the central task queue and the distributed task queues for both 4 thread runtimes did not vary much until the end, where the central task queue implementation began to undergo performance issues, and continued to suffer due to small grain sizes.

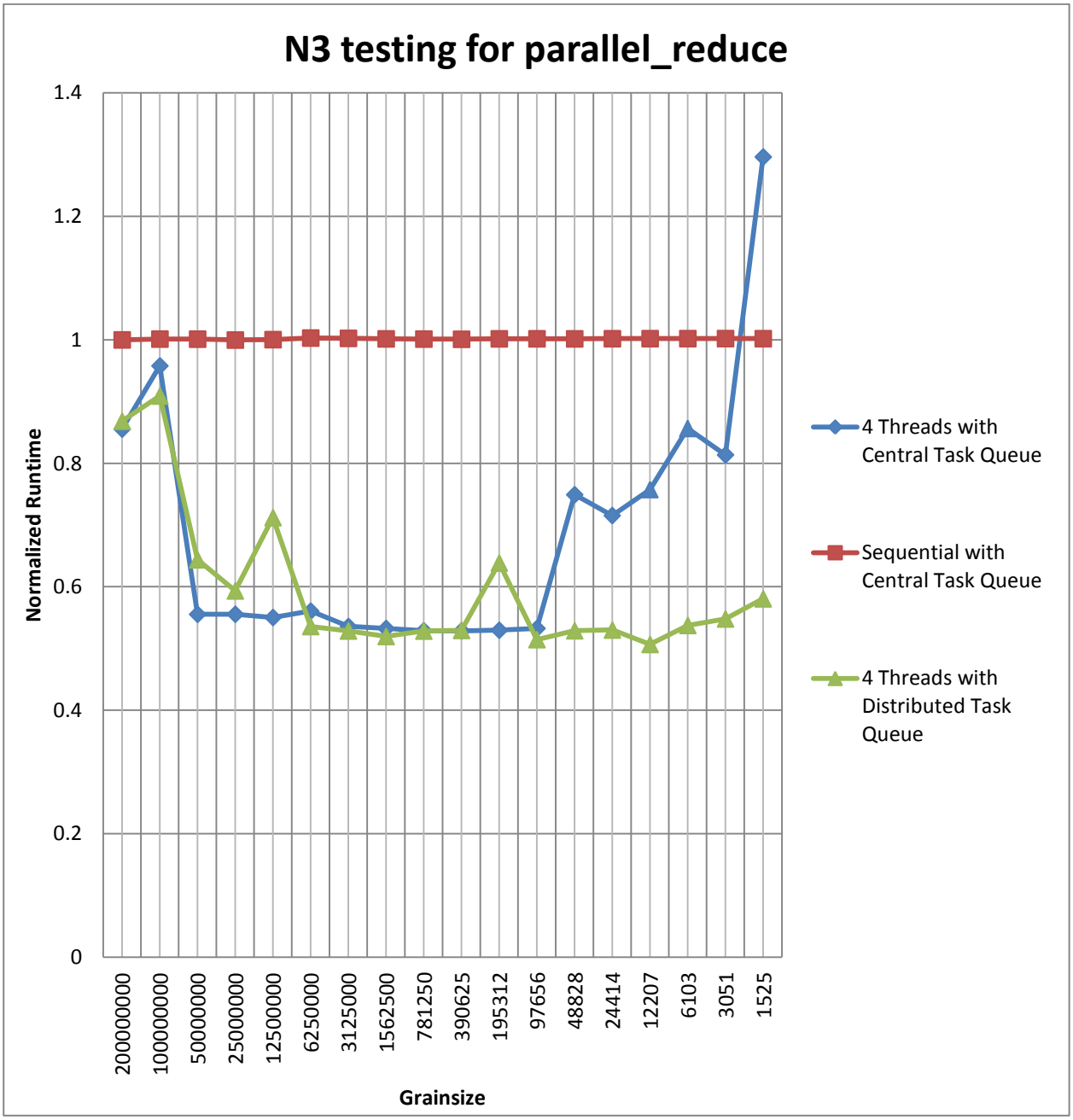

Figure 3: N3 parallel_reduce() runtime

Part 5: Implement parallel_reduce

Using the N3 Amoeba Server, the smallest grain size that is still within 5% of the infinite grain size is 1525, for the sequential version with the centralized task queue, which begins to segmentation fault following this grain size.

The maximum speed up on four cores with the centralized task queue is at grain size 390,625 also, where we can see a 53% increase over the sequential version.

The order at which these independent tasks are popped from the centralized queue will not have an impact on runtime. We can see that contention between the 4 threads making use of the centralized task queue starts to decrease in performance.

Part 6: Experiment on Other Machines

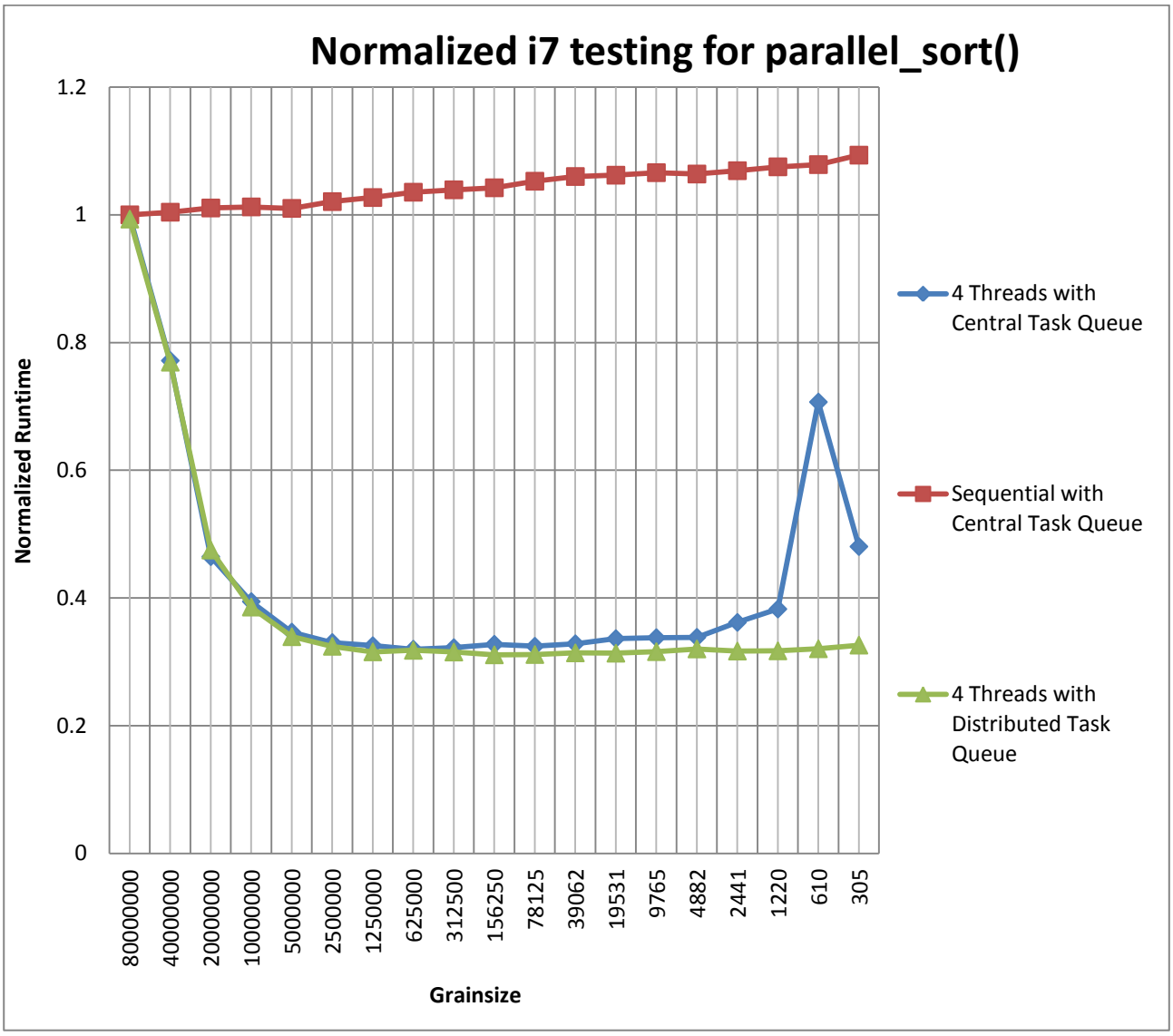

Figure 4: Quad Core i7 parallel_sort() runtime

Exploring the performance of the parallel sort code with varying grain size, we can see from Figure 4 how the single-threaded sequential runtime suffered as grain size decreased. The smallest grain size that is still within 5% of the infinite grain size was at around 156,250.

The maximum speed up on four cores, using the centralized task queue, occurs at the grain size of 625,000. This is 32% of the runtime of the sequential version at the infinite grain size. From Figure 4, we can also observe that as the graph approaches smaller grain sizes (less than 1220), the method runtime increases. As stated in the assignment, this could be due to running out of per-thread stack space, thus assisting us in determining our graph bound of grain size 305.

The order at which these tasks are executed, again, does not matter due to each task being independent. It is visible from Figure 4 that contention becomes a greater issue of the centralized task queue implementation at small grain sizes. This becomes especially apparent at the grain sizes below 1220.

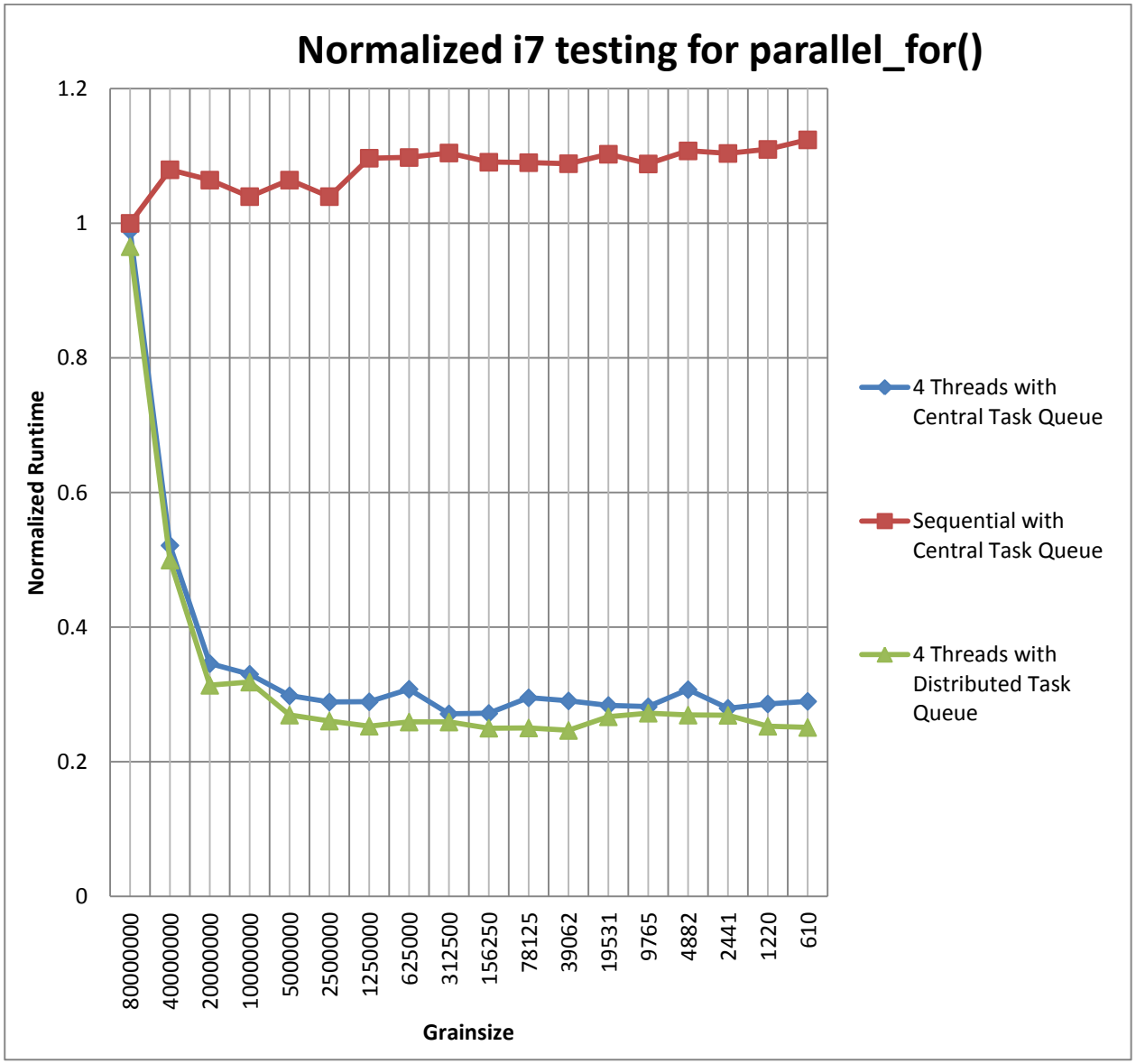

Figure 5: Quad Core i7 parallel_for() runtime

For parallel_for() implementation run on the Quad Core i7 machine, the smallest grain size that is still within 5% of the infinite grain size is 10,000,000, for the sequential implementation with the centralized task queue.

The maximum speedup achieved on four cores, using the centralized task queue and the improved version of parallel_for(), occurs at the grain size 312,5000. This achieves a 27% speedup from the sequential version, normalized at the infinite grain size.

The order of which the tasks executed again, as with parallel_sort(), does not have an impact on the performance, as each task is created to run independently. The decentralized work queue with stealing implementation follows the trend of the centralized work queue until after the infinite grain sizes. As the grain sizes decrease, the distributed task queue maintains a fairly stable performance, in contract to the performance of the slightly-slower central task queue implementation.

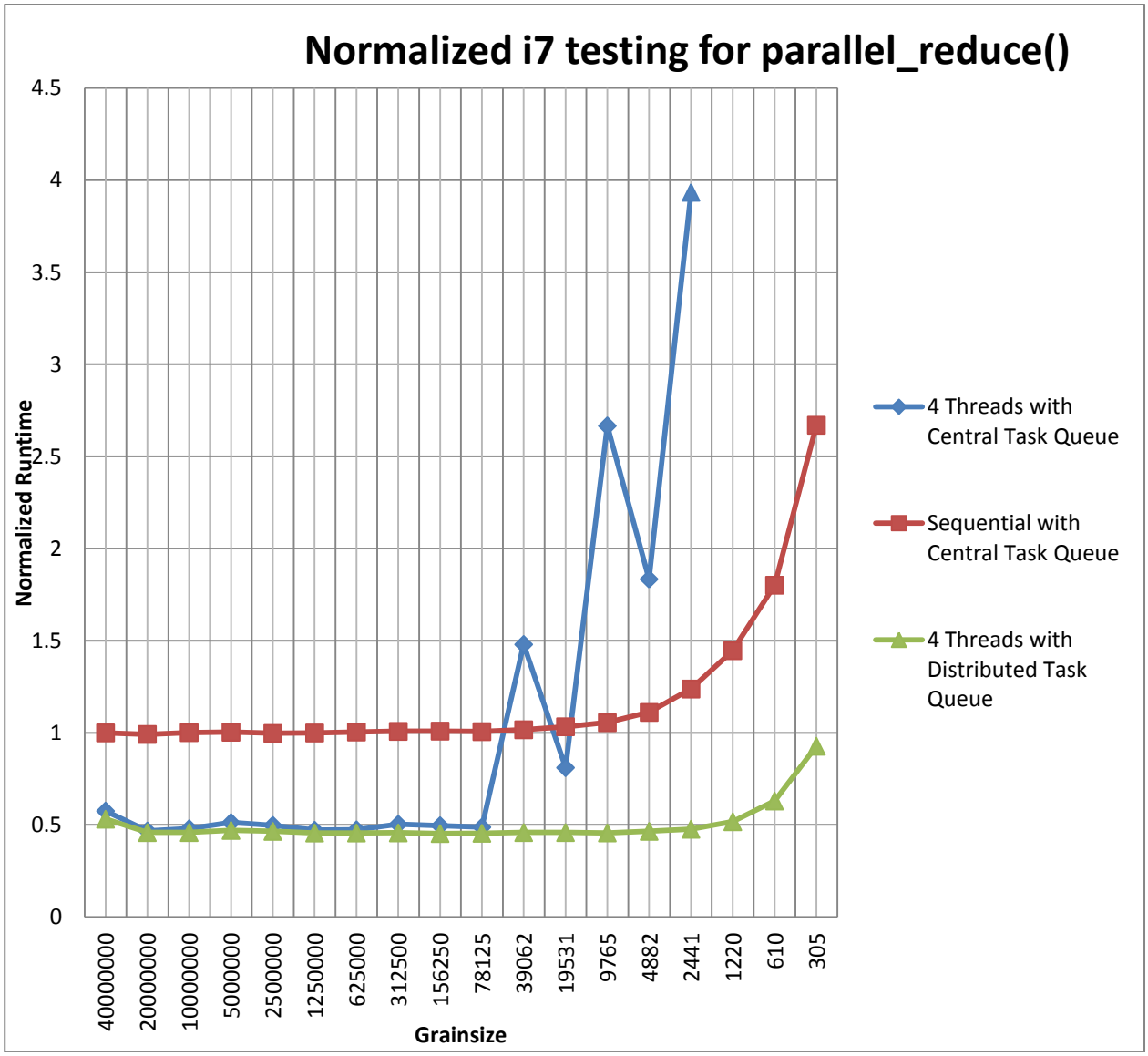

Figure 6: Quad Core i7 parallel_reduce() runtime

For a single thread, the smallest grainsize that we can use for sequential parallel_reduce() is approximately 9,765, which will still be within 5% of the infinite grainsize.

The maximum speed up on four cores occurs at 20,000,000 for the parallel_for() implementation with the centralized task queue. This is about 47% improvement over the sequential version of this method. This is because it makes use of threading, but it will not suffer from as many cache misses. Since parallel_reduce() is summing an array and maintains values in cache as it continuously adds numbers of similar locality.

In terms of the order of tasks, finally, they are able to independently run and are expedited in doing so by the distributed task queue with stealing implementation. This work stealing implementation proves itself to be a low-contention method up until grain sizes became less than approximately 1,000. This can also be attributed to increased cache misses, as smaller subsets of the array are summed, and since tasks are smaller, task stealing may divide the work on sub-arrays of similar locality.

Conclusion

In conclusion, we observed a notable speedup between the sequential and parallelized implementations of parallel_sort(), parallel_for(), and parallel_reduce(). Amongst these parallelized versions, we were able to achieve stable speedups when making use of a distributed task queue with random task stealing. This permitted our runtime to stabilize at low grain sizes (below the grain size of 610), while the centralized queue tended to run longer and eventually produce a segmentation fault, in most cases.

By selecting a random task stealing method to implement the distributed task queue, it reduces tendencies toward certain queues, and was designed in hopes to reduce contention on the first queue, where all tasks originate. We tried implementing a round-robin style task stealing method, but determined its overhead of maintaining counters and essentially all tending to the first queue would have negative effects on the runtime.

In terms of grain sizes selected for the normalized graphs, we selected to showcase the data within these bounds due to their ability to satisfy the "infinite" grain size, as well as achieve tests across all implementations that could go as close to 1 without causing a segmentation fault. With the majority of our graphs, it is seen that they begin to improve, level off, and then in parallel_sort() and parallel_reduce() especially, the 4 thread implementation with the centralized task queue performs worse at small grain sizes. As stated previously, we believe this degradation in runtime at small grain sizes for the central task queue can be attributed to cache misses, as well as the contention for the single task queue.

Comparing the N3 machine normalized results with the results of the Quad Core i7, we observed that the i7 results were overall slightly improved over the results produced by N3 testing. This could be for a variety of reasons. One reason we might assume these results from is the difference in processor architectures between the two testing environments. The N3 machine may be caching data further apart, thus resulting in a higher cache miss rate than that of the i7 processors. The locality of the i7 cache accesses is our hypothesis in regards to how our tests perform better in that environment.

Environment Specifications

Amoeba N3:

```
Architecture:          x86_64
CPU op-mode(s):        32-bit, 64-bit
Byte Order:            Little Endian
CPU(s):                24
On-line CPU(s) list:   0-23
Thread(s) per core:    1
Core(s) per socket:    12
Socket(s):             2
NUMA node(s):          4
Vendor ID:             AuthenticAMD
CPU family:            16
Model:                 9
Stepping:              1
CPU MHz:               2100.071
BogoMIPS:              4200.09
Virtualization:        AMD-V
L1d cache:             64K
L1i cache:             64K
L2 cache:              512K
L3 cache:              5118K
NUMA node0 CPU(s):     0,2,4,6,8,10
NUMA node1 CPU(s):     12,14,16,18,20,22
NUMA node2 CPU(s):     13,15,17,19,21,23
NUMA node3 CPU(s):     1,3,5,7,9,11
```

Quad Core I7:

```
Architecture:          x86_64
CPU op-mode(s):        32-bit, 64-bit
Byte Order:            Little Endian
CPU(s):                8
On-line CPU(s) list:   0-7
Thread(s) per core:    2
Core(s) per socket:    4
Socket(s):             1
NUMA node(s):          1
Vendor ID:             GenuineIntel
CPU family:            6
Model:                 58
Stepping:              9
CPU MHz:               1600.000
BogoMIPS:              6784.59
Virtualization:        VT-x
L1d cache:             32K
L1i cache:             32K
L2 cache:              256K
L3 cache:              8192K
NUMA node0 CPU(s):     0-7
```