09/12/2023

PJ1 – CS 380P – JOHN PETERSON - JLP5729

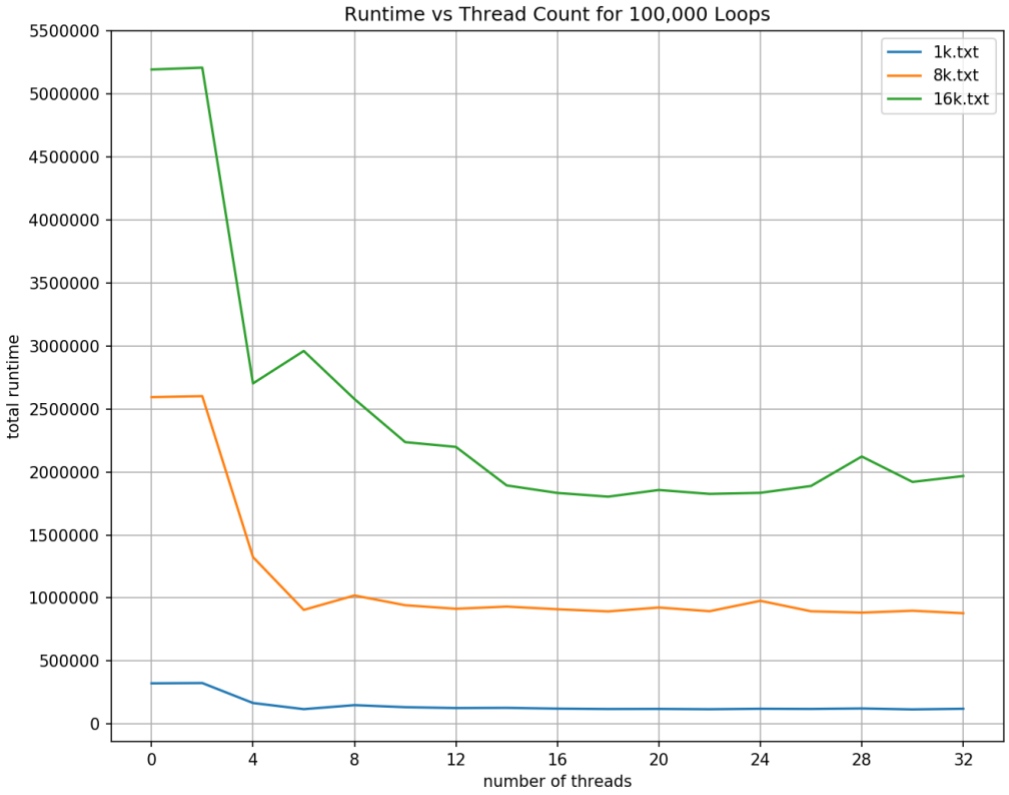
1. STEP 1: PARALLEL IMPLEMENTATION

Recall that prefix scan requires a barrier for synchronization.

In this step, you will write a work-efficient parallel prefix scan using pthread barriers.

For each of the provided input sets, set the number of loops of the operator to 100000 (-l 100000) and graph the execution time of your parallel implementation over a sequential prefix scan implementation as a function of the number of worker threads used. Vary from 2 to 32 threads in increments of 2.

Then, explain the trends in the graph. Why do these occur? From here: "A prefix sum algorithm is work-efficient if it does asymptotically no more work (add operations, in this case) than the sequential version. In other words, the two implementations should have the same work complexity, O(n)."



My algorithm uses a modified version of Blelloch scan as given in "Prefix Sums and Their Applications" by Guy Blelloch, Carnegie Mellon University, 1993. I chose to break up the work sequentially by thread, iterating through addition operations at each step according to the thread ID number and the remaining operations at that step, then use pthread_barrier object to wait for all threads to complete their assigned indices. This required padding the input to the next highest power of two then dropping the remaining indices when printing the output.

As such, there are several opportunities for overhead in the setup and teardown of these parallel structures. While the order O(n) is the same for addition operations, setting up the barrier, waiting on threads to complete at each step if work isn't completely evenly assigned, and teardown of the barrier adds extra processor time.

The program is set up with an artificial inflation of work via a scan_operator that adds a certain number of addition operations to each addition computation. The graph above has 100,000 increment operations for each addition.

As we add more threads to the input, this allows each thread to do less work respectively, and the program finishes earlier. However, we are limited by the architecture of the host machine. Adding more threads at a certain point, in this case past 6-12 adds more switching between threads on the internal processor, as we do not have a supercomputer to run this.

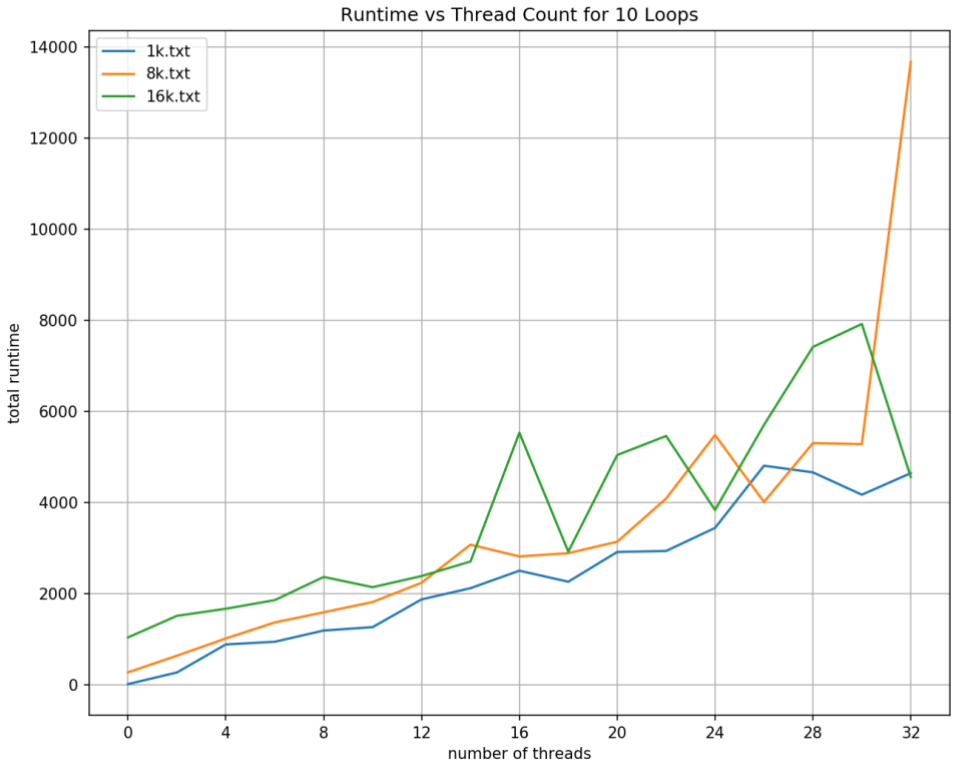
We reach the limit also partially due to Amdahl's law; only a certain segment of the work is parallelizable so throwing more processors at the problem can only speed up the part of the work that is parallelizable.

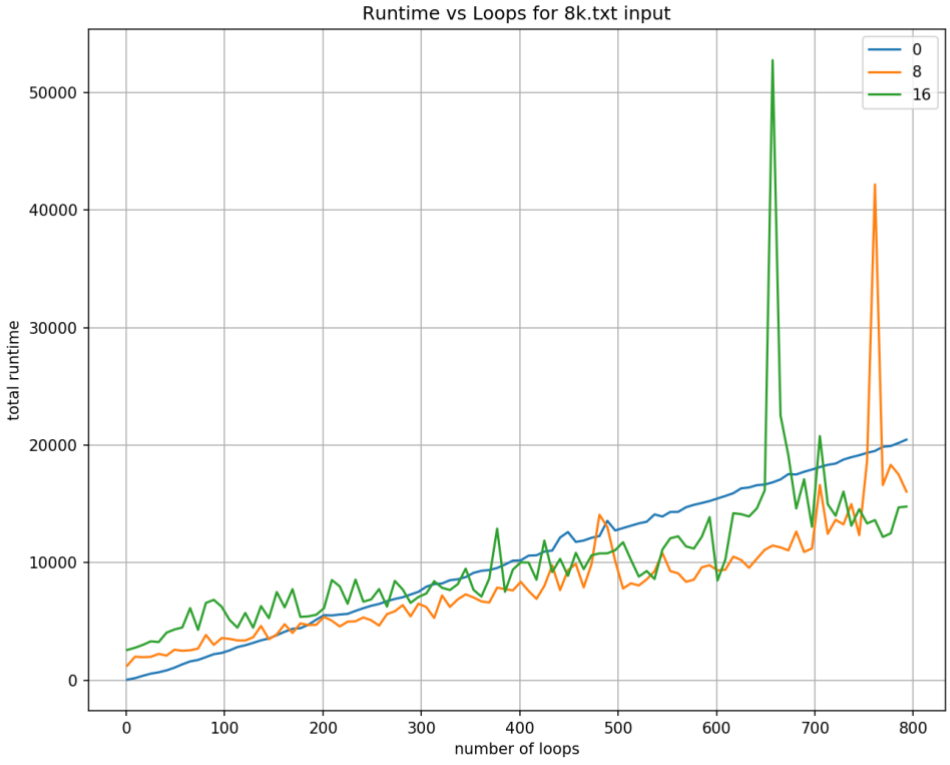
2. PART 2: PLAYING WITH OPERATORS

Now that you have a working and, hopefully, efficient parallel implementation of prefix scan, try changing the amount of loops the operator does to 10 (-l 10) and plot the same graph as before. What happened and why? Vary the -l argument and find the inflexion point, where sequential and parallel elapsed times meet (does not have to be exact, just somewhat similar). Why can changing this number make the sequential version faster than the parallel? What is the most important characteristic of it that makes this happen? Argue this in a general way, as if different -l parameters were different operators, after all, the -l parameter is just a way to quickly create an operator that does something different.

The first graph below shows an increase in runtime as the number of threads increases for my "efficient" parallel implementation. This happens due to overhead, setup, and teardown time. Starting the threads, waiting for synchronization with the barrier, and tearing down the threads and barrier all does more work than the sequential algorithm.

I also mentioned earlier the scan_operator function. The scan_operator at 10 loops does not provide enough computation work to compensate for the overhead of threads being


Runtime vs Thread Count for 10 Loops
1k.txt
8k.txt
16k.txt
total runtime
number of threads
0
2000
4000
6000
8000
10000
12000
14000
0
4
8
12
16
20
24
28
32


Runtime vs Loops for 8k.txt input
0
8
16
total runtime
number of loops
0
10000
20000
30000
40000
50000
0
100
200
300
400
500
600
700
800

generated. As seen in the second graph on the page (now above this sentence), varying the number of loops means that right around 200 loops the parallel implementation becomes on par with the sequential implementation, and continues to get more efficient over time (albeit with some small anomalies/outliers in the higher numbers).

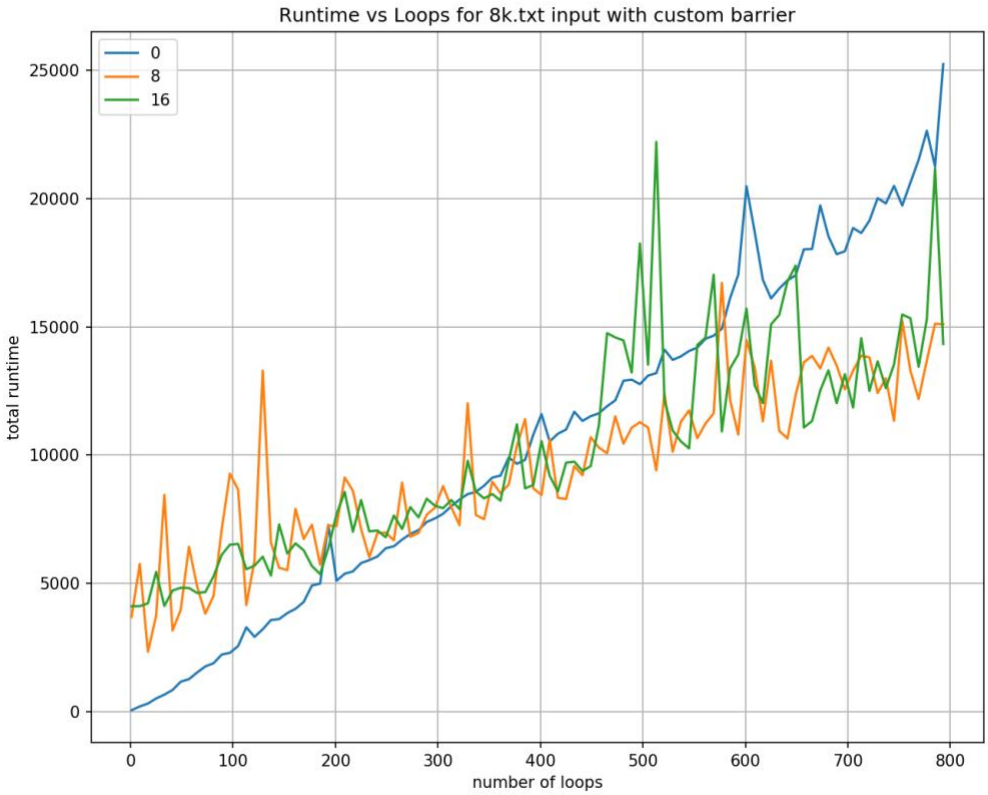
We can generalize this argument to different types of operators. Let's say that we had an operator that calculates a factorial. Higher numbers would obviously take more calculations to calculate a factorial, so each thread completing more work inside their thread apparatus would mean that the parallel implementation could be more efficient, respectively. Any operation that does more work compared to the overhead/initialization and thread management CPU work would make the parallel implementation more efficient than the sequential.

3. STEP 3: BARRIER IMPLEMENTATION

In this step you will build your own re-entrant barrier. Recall from lecture that we considered a number of implementation strategies and techniques. We recommend you base your barrier on pthread's spinlocks, but encourage you to use other techniques we discussed in this course. Regardless of your technique, answer the following questions: how is/isn't your implementation different from pthread barriers? In what scenario(s) would each implementation perform better than the other? What are the pathological cases for each? Use your barrier implementation to implement the same work-efficient parallel prefix scan. Repeat the measurements from part 2, graph them, and explain the trends in the graph. Why do these occur? What overheads cause your implementation to underperform an "ideal" speedup ratio?

How do the results from part 2 and part 3 compare? Are they in line with your expectations? Suggest some workload scenarios which would make each implementation perform worse than the other.

I chose to utilize a tree-structure barrier as defined in lecture. For the "arrive" and "go" arrays I used an array of semaphore items as defined in semaphore.h; this structure aligned perfectly with the await and assignment constructs given. Unfortunately, as shown in the graphs in the next page, my approach is not as efficient as the built-in pthread barrier for the given test cases. Pthread's barrier utilizes a simple counter which has the benefit of less overhead and more quickly updating the structure, with the potential downside of introducing memory contention on the single counter. I could not determine a test case that aggravated this, but one could imagine that if all or many of the threads were trying to update the counter in the pthread_barrier at the same time, that my implementation (or a more efficient version of my implementation) would be faster, as there would be no memory contention on the individual items in the semaphore array.



It is also possible, that as shown in the textbook, that adding padding to the semaphore arrays could reduce cache operations by placing items on separate cache lines. I did not research enough into the Codio platform and underlying virtualized and physical hardware to optimize to this level.

Pathological use case for a tree-based structure like utilized in mine is when we want to reduce memory contention and have a O(logn) complexity for updating all barriers (as opposed to an O(n) complexity for an array-based structure). Use case for a counter is when the threads are expected to complete at varying times and thus you are less worried about contention on updating a single counter item in memory.

One can see in the graph above that the graph above that my implementation starts to be more efficient than the sequential implementation around 300-320 loops in the scan_operator. This is slightly less efficient than pthread, for reasons posited above.

Overall, it was an excellent learning exercise to utilize the standard library implementation, build my own, and compare the two. I spent about 30 hours on this lab overall between understanding the questions asked and designing, implementing, testing, and measuring the results.