Parallel Programming Test Program 1

Ziad A. Arafat

New Mexico State University

Abstract

This is a demonstration of how parallel threads perform and behave when accessing memory in various ways. First we will test threads having unconstrained access to a single integer in memory. Then we will test the same but with a mutex from the OS. After which we will instead test a scenario where each thread has its own independent variable in an array in memory that is then added together with the others when complete to get the final value. Finally we will do the same array but will add padding such that each segment of the array for the threads is in a separate cache line.

We hypothesize that the first method will have incorrect outputs due to a race condition, the second method will be slow due to the waiting on the mutex, the third method will be more performant in general but constrained by coherency issues compared to the appropriately padded version which will align to the cache lines.

Parallel Programming Test Program 1

Method

System Specifications

The results are obtained on an core processor AMD Ryzen 7 4700U running at 2 GHz max and 1400 GHz min. It has three caches. 2 instances of 8Mb L3 cache with a 64B cache line, 8 instances of 4Mb L2 cache with a 64B cache line, and 8 instances of L1 cache (data and instruction) with a 64B cache line. The operating system is Arch Linux 64 bit with linux kernel 5.13.

Assessments and Measures

For each of the setups we will run a test for thread count, array size, and padding (for the last one). Each time we record the values we will run it at least 5 times to ensure we have representative data without any interference from other processes. We will ensure each test is done on the same machine and uses the same algorithm to measure the time.

The first setup will have N threads counting the number of threes by incrementing a single variable ‘COUNT’. There will be no mutex used and we expect we will get bad outputs so we probably wont run it more than once.

The second setup will be identical to the first except we rill have a mutex lock protecting the increment of COUNT. We predict that this will fix the bad outputs and so we will have data. This test is also meant to test the performance impact of asking the operating system for a mutex.

The third setup will not have a mutex but instead of using a single variable we will use a dynamically allocated array so that each thread will have its own value in memory to increment. When all the threads are finished running we will add all the values together to get the total of threes counted. We predict that this method will have better performance than the second setup.

The fourth setup will be identical to the third but will use memory padding to attempt to put each section of the array in a different cache line. This is meant to bypass cache coherency performance issues. We will achieve this by putting each count in a struct which has a count variable as well as a character string which is set to be the size of the cache lines (in this case 64B).

Results

The results largely reflected how we expected the programs to behave. When threads access memory they perform better when they can work on memory segments independently. Using operating system features to protect memory is also cumbersome and immediately seems to cause significant overhead with threads. While using threads to break up a long process can be performant many factors can cause race conditions and overhead that might make it less performant than the serial version.

Setup 1

In this setup we ended up getting inconsistent results between the serial code and the parallel code. Because of this we did not test further and simply recorded the output which demonstrated this fact.

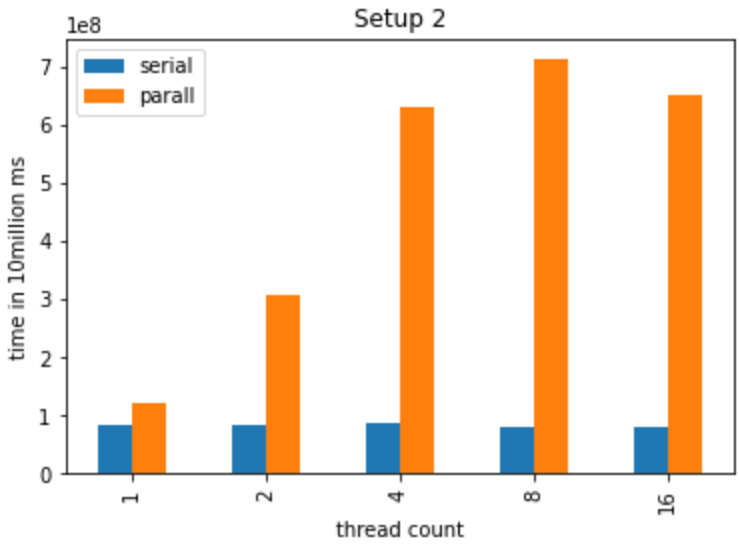
2 Threads

- Since the output was not the same for parallel likely due to the race condition caused by line 85 we will stop testing this code

```
doge@godemperordoge1998 ~/D/p/figure1_7 (main)> ./figure1_7 10000000 2
Time parallel: 62501107
Time serial:   39678580
parallel count is 2186343
serial count is   2499866
doge@godemperordoge1998 ~/D/p/figure1_7 (main)> ./figure1_7 10000000 2
Time parallel: 61941468
Time serial:   40740028
parallel count is 2185462
serial count is   2498949
```

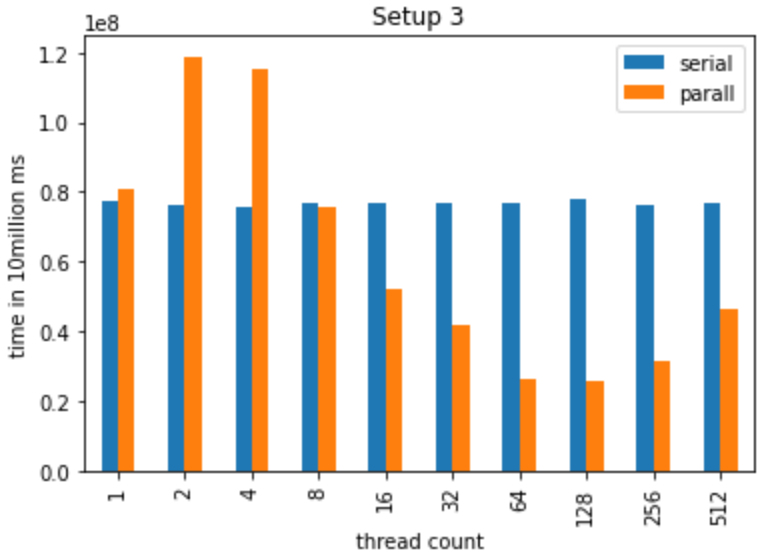
Setup 2

When we added the mutex to the code the results were fixed. However even with 1 thread there was significant overhead. This is likely because it takes time to ask the OS for a mutex at all and adds overhead when locking and unlocking the mutex to increment. There was no point where an increase in threads yielded better performance.



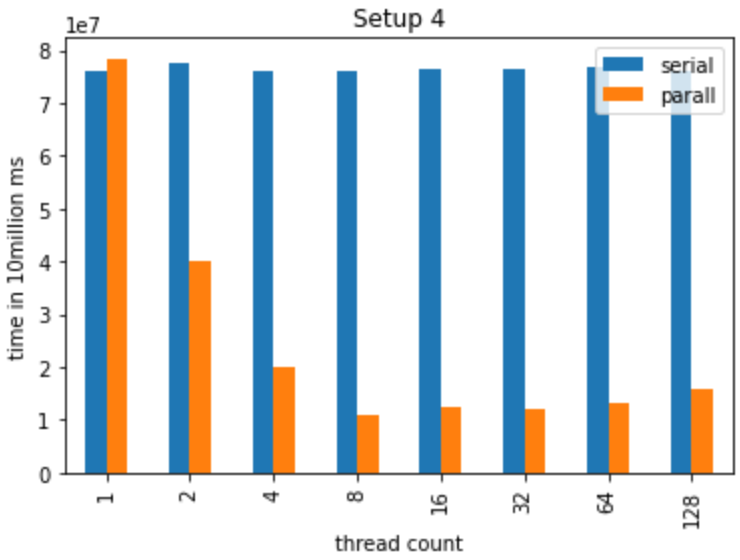
Setup 3

In the third setup we got good outputs and while the performance at first was similar to the serial we see gradual improvement as thread count increases. The increase is likely not fast because of the false sharing issue when all the data is accessed on the same cache line. The performance peaked at 128 threads and it seems that the thread count causes overhead after.



Setup 4

In the final setup we saw the best performance such that the time halves from the serial as soon as we have 2 threads. The plan to add padding to avoid false sharing seems to be very effective here. The results seem to go flat after 8 and we start to see some overhead after 128 threads.



Discussion

This demonstration shows that in order for parallel programming to be both correct and efficient we need to carefully set up memory access and processing so that each thread can work independently without overhead. While a mutex is good for doing processes where performance doesnt matter as much as simply protecting a variable from race conditions it is not a good solution when trying to distribute a problem to threads. The ideal situation is when threads can all work independently without needing to communicate or wait on each other in both hardware and software.

Raw Data and Code

The raw test data and experiments are all compiled in a file called ‘Raw Outputs.pdf’ as well as the individual code and makefiles in their folder structure. The code can also be found my Gitthub repo https://github.com/RustyRaptor/parallel-programming.