HPC project 4

Shadow Pritchard

April 2020

1 4.3

Question: Write a Pthreads program that implements the trapezoidal rule. Use a shared variable for the sum of all the threads' computations. Use busy-waiting, mutexes, and semaphores to enforce mutual exclusion in the critical section. What advantages and disadvantages do you see with each approach?

I used the code we used for the other homework as a reference on how to design the program. Pthreads require some different things compare to just using omp. Each thread must be manually forked and joined. It is basically just a more hands on approach. I could have simply passed in the thread number, but I also wanted to figure out how to pass multiple things. I created a struct to hold a, b, and the thread number. I then used a type (void*) argument for the trapezoid function and passed the struct to it in the thread init create. I used time.h to get the time and inserted the start time at the same location in the code shortly after taking input for a, b and n. n is required to be evenly divisible by the number of threads.

busy waiting is trivial to implement. Simply include a flag that starts at 0 and a while loop that loops while flag is not equal to the thread number. After adding the local result simply increment the flag so the next thread can access it.

Mutex and semaphore work by using built in methods to the pthread class and semaphore.h respectively. Each requires a global variable and an initialization in main with a lock and unlock in the trapezoid method surrounding the adding of local result to global sum.

the results in table 1 are from a 12 thread system using time.h.

threads	a	b	n	busy	Mutex	sema
2	0	10	1000000000	3851289	3921065	3932347
5	0	10	1000000000	3970376	3970781	3907797
6	0	10	1000000008	3977435	3962469	3909707
12	0	10	1000000008	7052066	6877767	6884122
16	0	10	1000000000	8859780	6747976	6755824
32	0	10	1000000000	9217983	6863151	6842220

Table 1: number of clock cycles with changing critical methods. This does not work because of the way time.h calculates time.

The results do not match the actual wall time at all. Thus, we can conclude there is something wrong with using the multiple threads with clock() and getting clock ticks.

To try to resolve the problems I imported omp and used the omp_get_wtime() method. This fixed the issues and I could get accurate wall time as opposed to clock ticks which don't mean much without context.

threads	a	b	n	busy	Mutex	sema
2	0	10	1000000000	1.978078	1.994109	2.005122
5	0	10	1000000000	0.786100	0.794980	0.782874
6	0	10	1000000008	0.657685	0.660813	0.683125
12	0	10	1000000008	0.584541	0.583021	0.579366
16	0	10	1000000000	0.751718	0.637965	0.637011
32	0	10	1000000000	0.731973	0.590651	0.583645

Table 2: time in seconds for the trapezoid program to run using different critical section methods.

from the results we can see that if the number of threads doesn't exceed the number of hardware threads, the performance stays very close between the methods. When the number of threads exceeds it though, busy waiting gets worse but the other two methods stay about the same. This is not really surprising, as busy waiting makes every single thread with a lower number finish first before the current thread can add. Thus, there is some wasted time where the OS is told that the thread still needs to run, but the program is just wasting computational time waiting on another thread. Mutex and Semaphore put the thread to sleep so the hardware thread is freed to go work on another thread while it is waiting.

2 5.6

Use OpenMP to implement a producer-consumer program in which some of the threads are producers and others are consumers. The producers read text from a collection of files, one per producer. They insert lines of text into a single shared queue. The consumers take the lines of text and tokenize them. Tokens are "words" separated by white space. When a consumer finds a token, it writes it to stdout.

I created a text generator that creates n files where every word is unique to help with tracing output. There are two lines in each file with a random number from 10-20 words in each line. The second line also ends with an extra word for formatting ease. The words are numbered according to the file number using the format "word*a*" where the first * is the file number and the second is the word number. On the second line "2" proceeds the file number. It isn't perfectly unique but for low numbers of files where debugging is taking place it works. Generate at least half as many files as total threads so each consumer has something.

As I understand the question, each producer reads in text from their own file and then sends that text to a queue that is globally accessible and used by all threads. The consumers then each take a line and split it into "tokens" and prints each one as it reads. So a consumer will fetch a line from the queue and read a word then print it. Then it will read the next word and print until the line is empty and then it checks the queue again. Producers end when they finish reading their file and adding it to the queue. Consumers use the tokenize function as outlined in the textbook.

Performance is a bit awkward to evaluate since as more threads are added more work is added in an equal amount. So, our best hope is that as we add more threads and more files, the time to compute stays about the same. The difference will come with standard out being limited to one thread at a time. Thus, we shouldn't get the same speed every time, but it should scale rather well.

for the below table since the number of threads are split between producers and consumers, the number of files is half the number of total threads. In cases where the number of threads is odd, the remaining thread is added to the consumers. Thus, with 3 threads total 1 is a producer and 2 are consumers.

Threads/files	time
1	0.000049
2	0.000062
3	0.000144
4	0.000268
5	0.000280
6	0.000283
8	0.000418
12	0.000615
16	0.000750
32	0.001516
64	0.002796

Table 3: Times for reading, tokenizing, and printing files

As we can see this does not scale as well as I would have liked, but that is likely due to the small line and file sizes so the threads just trip over each other.