Levi Overcast
DSA F21
October 17

Parallel & Distributed Computing
Homework 2

1. Exercise 4.1 from Chapter 4 in Pacheco
   4.1 When we discussed matrix-vector multiplication we assumed that both m and n, the number of rows and the number of columns, respectively, were evenly divisible by t, the number of threads. How do the formulas for the assignments change if this is not the case?

   The formulas as given only assume that m is divisible by t, the number of threads. Each thread loops over every value of n. And this is all to the good since array are stored contiguously in memory, so dividing individual rows by threads would add unnecessary jumps to the code. However, assuming that the number of columns is evenly divisible by the thread count is not a general solution. Suppose there were a non-zero remainder in the number of rows. Then the corresponding entries in the output vector would end up unintentionally as zeros. To fix this, simply assign all as normal except give the last thread its normal portion plus any remainder.

```
my_last_row = (my_id == nthreads - 1) ? my_first_row + stride
                                      : arrlen;
```

2. Exercise 4.2
   4.2 What would we have to do in order to divide A and y among the threads? Dividing y wouldn't be difficult–each thread could allocate a block of memory that could be used for storing its assigned components. Presumably, we could do the same for A–each thread could allocate a block of memory for storing its assigned rows. Modify the matrix-vector multiplication program so that it distributes both of these data structures. Can you "schedule" the input and output so that the threads can read in A and print out y? How does distributing A and y affect the run-time of the matrix-vector multiplication? (Don't include input or output in your run-time.)

Shared	Number of threads	Side length of square matrix	Time
Yes	1	10000	4.19617e-05
Yes	4	10000	7.00951e-05
No	1	10000	5.79357e-05
No	4	10000	8.4877e-05

This table summarizes my results across a range of sizes of matrices and numbers of threads. I observed no speedup for parallelization and I observed a significant slowdown between comparable runs with shared variables and runs with distributed copies of memory.

I attribute the slowdown in performance to the copying overhead. The data has to be copied from the original matrices into the copies when the threads cannot share global variables, which is an $O(n)$ time unavoidable cost. The benefit, of course, is that the distributed memory runs stand no chance of overwriting, and are therefore viable.[1][2]

3. Exercise 4.4

4.4 The performance of the π calculation program that uses mutexes remains roughly constant once we increase the number of threads beyond the number of available CPUs. What does this suggest about how the threads are scheduled on the available processors?

Suppose each CPU scheduled its threads totally equally, without caring about which ones are working and which ones are waiting. In a program with mutex locks, at most one thread among all the scheduled threads is running over a critical section at any given moment. If the CPU gives equal priority to the thread that is locked as to the thread that is working in a critical section, then as we add threads, I would expect a slowdown, because the number of threads that are locked at any given time will increase.

Given that the performance of the program is roughly constant as we increase the number of threads beyond the number of CPUs, I conclude that each CPU schedules its threads proportionally to the work that they are given. Threads which are locked get less CPU time and threads which aren't get more.

4. Exercise 4.5

4.5 Modify the mutex version of the π calculation program so that the critical section is in the for loop. How does the performance of this version compare to the performance of the original busy-wait version? How might we explain this?

The following results were taken with $n = 100,000$.

With Busy Loop

# threads	π estimate error	Time	speedup	efficiency
1	3.18309889e-06	0.00151801	1.0	1.0
2	3.18309887e-06	0.000638962	2.37574378	1.18787189
3	3.18313069e-06	0.000442028	3.43419421	1.144731403
4	3.18309886e-06	0.000329018	4.61375973	1.153439932
5	3.18309886e-06	0.000431061	3.52156655	0.70431331
6	3.18322619e-06	0.000275135	5.51732785	0.919554642
7	3.18325802e-06	0.000281096	5.40032586	0.771475123
8	3.18309887e-06	0.000258923	5.86278546	0.732848182

[1]Main: `https://github.com/lovercast/DS-A/blob/main/dist/pthreads/matvect.c`
[2]Headers:`https://github.com/lovercast/DS-A/blob/main/dist/pthreads/matvect.h`

With mutex inside for loop

# threads	π estimate error	Time	speedup	efficiency
1	3.18309889e-06	0.00288701	1.0	1.0
2	3.18309889e-06	0.00471282	0.61258651	0.306293255
3	3.18313067e-06	0.0159318	0.18121053	0.06040351
4	3.18309889e-06	0.00493002	0.58559803	0.146399507
5	3.18309889e-06	0.00450921	0.64024740	0.12804948
6	3.18322621e-06	0.00446701	0.64629584	0.107715973
7	3.18325800e-06	0.00411415	0.70172696	0.100246709
8	3.18309889e-06	0.00367498	0.78558522	0.098198152

Some significant difference are readily apparent. The fastest time in the second experiment (the first) was twice as slow as the slowest run in the first experiment. The experiment with the mutex inside the `for` loop is characterized by less-than-1 speedup, whereas the busy-waiting model is characterized by above-1 and increasing speedup with added cores. Similarly, the efficiency for the second experiment is about an order of magnitude worse than that for the first experiment. The estimates of π have a similar degree of error, which leads me to believe that neither experiment experiences much memory overwriting, i.e. the busy-waiting and mutex thread control mechanisms work as intended.

This difference is to be expected. In the second case, we have added the locking mechanism inside the loop, adding all the overhead for a mutex lock and unlock to every loop iteration. In addition, this will tend to cause traffic jams as one loop (effectively at random) secures access to the inside of the `for` loop first and stops all the other loops from executing. This causes a complete loss of all the optimizations gained by caching and branch prediction inside of the loop, because as soon as a thread enters, it will more than likely be interrupted by a locked mutex and all the overhead that entails.

Exercise 4.6

4.6 Modify the mutex version of the π calculation program so that it uses a semaphore instead of a mutex. How does the performance of this version compare with the mutex version?

With Mutex Lock

# threads	π estimate error	Time	speedup	efficiency
1	3.18309889e-06	0.000729799	1.0	1.0
2	3.18309887e-06	0.000426054	1.71292606	0.856463031
3	3.18313069e-06	0.000313997	2.32422284	0.774740948
4	3.18309886e-06	0.000277996	2.62521403	0.656303508
5	3.18309886e-06	0.000461102	1.58272790	0.31654558
6	3.18322619e-06	0.000291109	2.50696131	0.417826885
7	3.18325802e-06	0.000247955	2.94327196	0.420467424
8	3.18309887e-06	0.000248909	2.93199121	0.366498901

With Semaphore

# threads	π estimate error	Time	speedup	efficiency
1	3.1830989e-06	0.00118995	1.0	1.0
2	3.1830989e-06	0.000673056	1.767980673	0.883990337
3	3.1831307e-06	0.000417948	2.847124523	0.949041508
4	3.1830989e-06	0.000370026	3.215855102	0.803963776
5	3.1830989e-06	0.000267029	4.45625756	0.891251512
6	3.1832262e-06	0.000246048	4.836251463	0.806041911
7	3.183258e-06	0.000280857	4.23685363	0.605264804
8	3.1830989e-06	0.000239849	4.961246451	0.620155806

The semaphore run performs about 50% worse for a single thread, but the speedup curve is much steeper for the semaphore experiment, and consequently the efficiency holds up much better than for the mutex lock, such that the fastest time in any of the runs is held by the semaphore experiment for 8 cores.

This leads me to conclude that a semaphore-based solution scales much better in this use case. Why that is, I am unsure. What is the relative overhead of a semaphore and a mutex lock? What is the penalty for waiting for a mutex lock and how does it compare to the penalty for sitting at `sem_wait`?

I'd like to note that after having run the experiment with no guards around the critical section at all, the control run with one thread ran about twice as fast as the semaphore run, but with 8 cores, the semaphore run was only 10% slower. That seems like remarkably low overhead for the small benefit of making the data race-free. Why is the semaphore based solution so scalable here?[3]

Exercise 4.8

4.8 If a program uses more than one mutex, and the mutexes can be acquired in different orders, the program can deadlock. That is, threads may block forever waiting to acquire one of the mutexes. As an example, suppose that a program has two shared data structures–for example, two arrays or two linked lists–each of which has an associated mutex. Further suppose that each data structure can be accessed (read or modified) after acquiring the data structure's associated mutex.

a. Suppose the program is run with two threads. Further suppose that the following sequence of events occurs:

Time	Thread 1	Thread 2
0	`pthread mutex lock(&mut0)`	`pthread mutex lock(&mut1)`
1	`pthread mutex lock(&mut1)`	`pthread mutex lock(&mut0)`

What happens?

Each thread locks the mutex that the other needs in order to advance beyond Time 2. Consequently, both threads will block until one of their locks is unlocked by some external process.

[3]Code for these ex.: `https://github.com/lovercast/DS-A/blob/main/dist/pthreads/calcpi.c`

b. Would this be a problem if the program used busy-waiting (with two flag variables) instead of mutexes?

We can imagine a parallel situation:

In Thread 1:

```
while (flag1 != my_rank) { }
while (flag2 != my_rank) { }
```

And in Thread 2:

```
while (flag2 != my_rank) { }
while (flag1 != my_rank) { }
```

In this case it's immediately obvious why there is no danger of deadlock here. Since the flags are set in order, there is no race to grab the flag. The value of the flag is independent from the time that a thread arrives at the `while` loop. Whichever thread has a lower assigned id will pass both loops when it is their turn while the other waits, and when the first thread has finished using the resources, it will increment the flag and the next thread waiting will proceed.

c. Would this be a problem if the program used semaphores instead of mutexes?

Again, we can imagine a parallel situation:

In Thread 1:

```
sem_wait(&sem1);
sem_wait(&sem2);
```

In Thread 2:

```
sem_wait(&sem2);
sem_wait(&sem1);
```

The routine `sem_wait` blocks until the lock is acquired, so in the case that two threads acquire the first lock at the same time, they both will be unable to pass the other without external intervention.