Semiar 2 Report - PThreads

Philipp Noel von Bachmann

March 30, 2022

1 Lab08_Pthreads_I

1.1 Task 1

The code produces the following output:

```
This is the main() function starting.
the main() function now calling pthread_create().
This is the main() function after pthread_create()
main() starting doing some work.
second thread() starting doing some work.
thread() starting doing some work.
Result of work in second thread(): sum = 0.010000
Result of work in main(): sum = 10.000000
the main() function now calling pthread_join().
Result of work in thread(): sum = 99.999998
```

We see that first the threads are created, and then the main function is doing some work. Since the workload in the second thread is the samllest, it finished first. After that the mam function finished its work. That when we call join. Now we wait for all threads to join, which means waiting for thread 1, since its the only one left working.

1.2 Task 2

```
This is the main() function starting.
the main() function now calling pthread_create().
This is the main() function after pthread_create()
the main() function now calling pthread_join().
The values are 5.700000 9.200000 1.600000
The values are 1.000000 2.000000 3.000000
```

We can see that by giving different data to the threads, we get different results, thus showing that we acces the data given to the threads in the right way.

1.3 Task 3

We compare the runtime:

Split	800000000/0	700000000/100000000	400000000/400000000
Runtime (s)	0.39	0.34	0.21

The runtime nearly havens for the most even split compared to the most uneven, which we would expect since the work can be done perfectly in parallel and in the even case both threads get the same amount of work. We also see that if we don't balance the work evenly, we get a small increase in runtime but one thread has to wait for the other one to finish.

1.4 Task 4

As proposed, we take the naive algorithm where we just loop over all smaller numbers and try to divide to determine if a number is prime. In the outer loop, we just loop over all numbers up to the input N to count the primes up to N. Now we create two threads and give each of them a start and end value between which it should count the primes. In addition it gets a variable to write the final count to. Since for large numbers, the workload per number becomes higher, just splitting the number range evenly between threads will likely result in unbalanced workload. Thus, we also try a different split. The results for $N = 100000$ are:

Split	Single thread	$\frac{1}{2}$ and $\frac{1}{2}$	$\frac{2}{3}$ and $\frac{1}{3}$
Runtime (s)	0.3	0.22	0.17

We see that if we divide half-half, we don't get an increase in runtime of two because of imbalanced work, as expect. By giving the first thread (which iterates over the smaller numbers) more total number, we get a more even work and nearly reach a halfen of runtime.

1.5 Task 5

We pass the id to thread as an integer. Since integers are smaller than pointers, we can pass the integer directly instead of needing to pass a pointer to it. The result for $N = 4$ is:

```
Thread 0
Thread 2
Thread 3
Thread 1
```

We see that even though we create the threads in order, the print statement is in a different order, which shows that the creation if threads takes a very small amount of time and thus all threads compete about who can print first.

1.6 Task 6

Our code is nearly identical to 1.4, except that we now have a variable number of threads M. Results for $N = 100000$:

M	1	2	4	8	10	20
Runtime (s)	0.32	0.24	0.15	12	0.1	0.09

We see that our increase is less than linear, which we would expect from 1.4, since dividing the range up evenly leads to an unbalanced workload split. Nevertheless, we see an decrease in runtime the more thread we add, since every new thread cuts the range per thread smaller. However when we go over the number of available cores (8), the speed-up slows

down significatly, since we don't have more cores to compute in parallel. The speed up here mostly comes through other things like instruction level parallelism.

1.7 Task 7

```
Hello from thread
Hello from thread
Hello from SubThread
Hello from SubThread
Hello from SubThread
Hello from SubThread
```

We see that each of the 2 threads gets called, and all of the 2x2 subthreads as well.

1.8 Task 8

If we don't protect the **sum** variable by a mutex, we get a different result every time since all threads try to write to the variable at the same time and the results become random. However when protecting the variable, we get the same result every time, since the mutex just allows one threads to write to the sum variable at a time, and the other threads have to wait for their turn.