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.

2 Lab09_Pthreads_II

2.1 Task 1

Output:

```
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().
Data from thread malloc 1234
```

We see that we can acces the data in the main function. This happens because in the thread we malloc the variable on the stack, thus it becomes accessible by its pointer globally until we free it. By returning the pointer to the main function, we now have acces to to this variable in the main part. Of course, this means we also have to free the variable in the end.

2.2 Task 2

In our inital code, we see that we use 200% CPU. This is because the thread is constantly checking if **DoItNow** is set to 0, which wastes CPU resources. If we don't set **DoItNow=1** in the main, our program will not exit since our thread will not finish because it still waits for **DoItNow=1**. Now we use a condition variable. As expect, we just use 100% CPU now, since our thread is put to sleep. Some further modifications to break the code:

- Removing the **pthread_cond_signal(&cond);** leads to the program not finishing, since our thread is never woken up.

- Not releasing the mutex in the main function leads to the program not finishing, since our thread is now not allowed to acces the **DoItNow** variable.
- Not locking the mutex in the thread still leads to the same behaviour as before. However in a larger scale code this could become dangerous, since we now read **DoItNow** while others can write it.

2.3 Task 3

By setting a thread to detached, we can't join it anymore. Furthermore it will continue to run, even after the main programm finished. It is however necessary to call **pthread_exit** otherwise, the detached thread will be killed after completion of the main and the programm will exit.

```
Main: creating thread 0
Main: creating thread 1
Thread 0 starting...
Main: program completed. Exiting.
Thread 1 starting...
Thread 0 done. Result = -2.183590e+08
Thread 1 done. Result = -2.183590e+08
```

We see that the main thread completes and exits, but the programm still waits for the detached threads to complete, since we called **pthread_exit** in the end.

2.4 Task 4

Output of synch:

```
Hello World! 0
Hello World! 1
Hello World! 2
Bye Bye World! 2
Bye Bye World! 0
Bye Bye World! 1
```

Output when removing barrier:

```
Hello World! 0
Bye Bye World! 0
Hello World! 1
Bye Bye World! 1
Hello World! 2
Bye Bye World! 2
```

We can see that when we have a barrier, we first get all hellos before all byes. This is since the barrier lets the threads wait for the others to start. However when removing the barrier, the threads don't wait for the others to start, thus some finish before others start.

For spinwait, we see that when using optimizer flags, our programm doesn't terminate. This is because of the empty loop in line 19. The optimizer sees that the loop does nothing, and thus removes it, which can be also seen in assembly. When using the keyword volatile, we tell the compiler that this variable might change over time, thus it doesn't optimize that loop out.

2.5 Task 5

We parallelize the code by letting each thread compute the integral over a certain interval and the summing up all the integrals. We see that the parallel version is more accurate. By choosing a datatype, we necessarly get a rounding error. However when using multiple threads, they both get their own rounding error for their interval, which in sum is a smaller rounding error than having just one thread. In other words, by having the sum distributed over more variables, we don't accumulate the rounding error in one.

2.6 Task 6

We implement the new strategy by giving each thread a range of elements to compute the rank of. Each thread then iterates over all its elements in serial. This removes the overhead of creating threads all the time and leads to a time of 0.36s vs 1.53s for 4 threads and array size of 100000.

When studying the complexity, a serial enumeration sort has complexity $O(n^2)$, since we need to iterate over all elements, and for each element need to iterate over all elements again to comute the rank. When parallelizing the code, we see that we can perform the outer loop in parallel, however the cost for the inner loop stays constant. For 4 threads for example, we get $O(\frac{n}{4} \cdot n) = O(n^2)$ (assuming runtime scales linearly with number of threads). This means that compared to merge-sort which has $O(n \log(n))$, enumeration sort is worse.

2.7 Task 7

We see that the computation of each element in the output matrix is independent of the rest. Therefore, we can theoretically get each computed by a different thread. With the results from the previous Task however, we instead opt to give each thread a block to compute. This leads to the following performance, $N = 1000$:

Number of threads	1	4	16	25
Runtime (s)	1.08	0.44	0.35	0.33

We see that the runtime decreases, although not linearly. If our matrix gets to small, the overhead from threading is more than the reward. For exmaple for $N = 10$, we get 0.000087s for 1 thread and 0.000132s for 4.