Lab 10 Daniel Hjelm

My computer does not seem to support openMP so I will need to use the school computers (Fredholm).

Task-1:  
If we just compile it without any parallelisation flag we only run one thread and “bonjour” is only printed once.

If we include the optimization flag it will print out five times since we have set threads to 5.

If we change number of threads to 10 it will print out 10 times.

If we do not specify number of threads it uses the default number of threads which is 16 for Fredholm.

Changing the number of threads using export OMP\_NUM\_THREADS=n works perfectly fine.

We can use the “omp\_set\_num\_threads(n)” within the program to set the amount of threads used. We also need to include the header <omp.h>

One good thing to know is that “num\_threads(n)” that one specify besides the “#pragma omp parallel” before the function overrides the “omp\_set\_num\_threads(n)”, which of course overrides the default/changed default threads given by OMP\_NUM\_THREADS.

Task-2:

2 threads: Work took 9.295573 seconds

5 threads: Work took 9.367324 seconds

10 threads: Work took 11.888256 seconds

20 threads: Work took 17.230434 seconds

Task-3:

Fredholm seems to be 16 threads!

<https://stackoverflow.com/questions/59434959/openmp-omp-get-num-threads-v-s-omp-get-max-threads>

Task-4:

Using openMP is much more easier than using pthreads because we don’t have to create a thread, then type cast the arguments to void, then type cast back and later join every thread. With openMP it is just more smooth.

Print the “result” values for arr[0] and arr[1] works.

Task-5:

When split 7,1 we get: Work took 2.161943 seconds

When split 8,0 we get: Work took 2.473520 seconds

When split 4,4 we get: Work took 1.243046 seconds

So when we go from one thread doing all to splitting the work equally between two we get a speed up around 2.

Since the cpu time (user time) is the same for 4,4 and 2,6 the finished thread seems not to take any more cpu time after it is finished. This is the same as for pthreads.

Task-6:

Let us calculate how many primes up to 100.000:

Time using 1 thread: real 0m2,220s

Time using 2 thread: real 0m1,633s

Time using 4 thread: real 0m0,945s

Time using 8 threads: real 0m0,508s

Time using 16 threads: real 0m0,424s

Time using 32 threads: real 0m0,319s

Time using 64 threads: real 0m0,298s

Time using 128 threads: real 0m0,303s

So no significant speed up after 32 threads I guess.

We get a speed up but not perfectly since we have unbalanced work load. This is because it is heavier to calculate primes between 10-20 than between 0-10.

I can’t come up with any good balancing algorithm unfortunately.

Task-7:

Now it works with and without the –fopenmp and it also works on my macbook.

Task-8:

We see that by letting all threads change the global variable at the same time we get weird results. So if we run this multiple times for the same n we get different values.

We had to use lock etc in pthreads but in openmp it’s easier. We can use “#pragma omp critical” before changing the global variable.

If we use that we no longer get weird results. However, it seems to be a little bit slower, but I’m not sure.

Task-9:

If nested loops are turned on then we can use nested loops. If it is turned off the nesting is not happening but the program can still run and we get a result, but not what we anticipated.

Get a weird error for the second part with the merge sort…