Part 1, Theory

Problem 1, Multithreading

a) A thread exists within a process, and can be described as a “light weight” process. A process consists of its own block of memory for the heap and the stack, descriptors of resources, security information and information about the state. Besides its own stack and program counter, a thread share all of this with all the other threads in the respective process. Threads also takes less overhead when creating, suspending and starting up again.

b) The main difference is that in Pthreads you have to define the excatly what the threads are supposed to do, while with OpenMP you can simply state what is supposed to be parallelized and then the compiler/run-time system takes care for the details. Due to this OpenMP is compiler-depended, and will not work for every C compiler.

This is supposed to make parallelizing easier, and allows programmers to incrementally parallelize their programs, something which is hard with Pthreads.

c) Because OpenMP and Pthreads are used to implement parallelism on shared-memory systems. Clusters do share memory, but in addition they have their own private memory, which makes it impossible to fully utilize clusters with threads.

Problem 2, OpenMP Schedules

a) The purpose of the OpenMP schedule clause is to allow the program to decide how the iterations of for-loops should be assigned.

b) Static: Lets you decide how many iterations in a row a thread should get. I.e. if static is 1, you can get this:

Thread 1: 1 3 5 7
Thread 2: 2 4 6 8

But if static is 2, you get this:

Thread 1: 1 2 5 6
Thread 2: 3 4 7 8.

Dynamic**:** The iterations are divided into chunks, and the threads request chunks as soon as their finished with one.

Guided: Same as dynamic, except that every new chunk is smaller than the previous one.

Runtime: An enviornment variable is set at runtime, which decides what kind of scheduling should be used.

Problem 3, Reductions & Races

a) In openMP reduction is when you create a private variable for each respective thread, and preforms the same operation on different operands. You then gather the results from the private variables into a global one, by preforming the same operation on the private variables.

An example of this is my soultion of problem 3, d.

b) There are two cases of race condition:

1. Two or more threads try to write to the same block of memory at the same time.
2. One or more threads try to write to the same block of memory and one or more threads try to read the same block of memory, at the same time.

Thus the result depends on which thread that mangage to complete its task first.

c. The reason for the possibility of incorrectness is that all of the threads tries to increment the same block of memory, the variable sum. Thus some of the incrementations will be overwritten, and lost.

d. Simply change line 11 to:

```
...
11. #pragma omp parallel for \
12.     reduction (+:sum)
13. for (int i = 0; i < 1000; i++) {
…
```

Problem 4, Deadlocks

The problem is the barrier at the end of the while-loop. The problem arises when a thread

does not rebranch to the while-loop, because another thread has found the value, while one or more threads are at the barrier. Since a thread did not re-enter the while-loop, it will never reach the barrier, and the other threads will wait forever for it.

A fix would seem to be to make the following changes:

```
…
11. while (1) {
...
30.if (found == 1) break;
31. #pragma omp barrier
…
```

As it would seem that since the barrier is right after the break, then the threads will still wait at the barrier even though they break out of the while loop. I've tested this, and it seems to work, but could be that I'm just really luck. Could you please verify if this would work?

Part 2, Code

Problem 1, Matrix multiplication
c) time ../gemm 1 1024 1024 1024: 13.846 secs.

time ./gemm_pthread 1 1024 1024 1024: 13.065 secs
time ./gemm_pthread 2 1024 1024 1024: 7.928 secs
time ./gemm_pthread 3 1024 1024 1024: 7.567 secs
time ./gemm_pthread 4 1024 1024 1024: 7.496 secs
time ./gemm_pthread 5 1024 1024 1024: 7.598 secs
time ./gemm_pthread 6 1024 1024 1024: 7.249 secs
time ./gemm_pthread 7 1024 1024 1024: 7.335 secs
time ./gemm_pthread 8 1024 1024 1024:7.807 secs

time ./gemm_openMP 1 1024 1024 1024:
time ./gemm_openMP 2 1024 1024 1024: 8.839 secs
time ./gemm_openMP 3 1024 1024 1024: 8.881 secs
time ./gemm_openMP 4 1024 1024 1024: 8.996 secs
time ./gemm_openMP 5 1024 1024 1024: 8.463 secs
time ./gemm_openMP 6 1024 1024 1024: 8.684 secs
time ./gemm_openMP 7 1024 1024 1024: 8.362 secs
time ./gemm_openMP 8 1024 1024 1024: 8.816 secs

We see a drastic increase from 1 thread to 2 threads in Pthreads, and not much after that. This is due to the fact that my computer has a dual-core processor, and is thus only really able to

utilize two threads running at the same time.

OpenMP decides by itself how many threads it should run, and thus get about the same result for every input.

Problem 2, k-means.

c)

I used the problem size 50 clusters and 10000 points.

time ./kmeans 1 50 10000: 10.360 secs

x = [1, 8]
time ./kmeans_openmp x 50 10000: 5.055 secs.

OpenMP decides by itself how many threads it should use, thus the time does not depend on the input.

time ./kmeans_pthread 1 50 10000: 10.137 secs
Is a bit faster than kmeans since I optimized the code.

x = [2, 3]
time ./kmeans_pthread x 50 10000: ~5.2 secs

x = [4, 8]
time ./kmeans_pthread x 50 10000: ~4.7 secs

I don’t quite understand why it’s faster for 4 or more threads, than for 2-3 threads, since my computer has a dual-core processor.