HomeWork 1

Task 1

a) In the first subtask of the matrix elements we add to the provided code snippet so that the program can find the max and min value in the matrix as well as printing those locations. We did this by creating a structure that stores the value and position of the extreme value and then we iterate over the entire matrix to find the extremes. The random function does not seem to be particularly random...
b) In this task we would change a) so that we do not use the provided barrier function as well as not using arrays to keep track of the min, max and total. We did this by having global struct for the min, max and total as well as local values. We then took mutex locks over the total, min and max and added to the total and we also did comparisons of the local and global extreme variables and updated the global extreme values if the local were bigger/smaller, then unlocked the mutex lock.
c) In the final task we switched from a divide & conquer strategy in b) to using a bag of tasks. This means that we stopped assigning rows to workers with an algorithm at runtime, and assigned workers to rows that had not been worked with when the worker was free. The bag of tasks was mutex locked when it was changed and read.

In task A) there is no parallelism which shows our baseline, in task B there is parallelism which is faster than matrix A. In matrix C we make sure that no thread is idle and therefore it is slightly faster than matrix B.

Matrix	Workers	Time (s)
Matrix A	10	0,4
Matrix B	10	0,06
Matrix C	10	0,05

Task 2

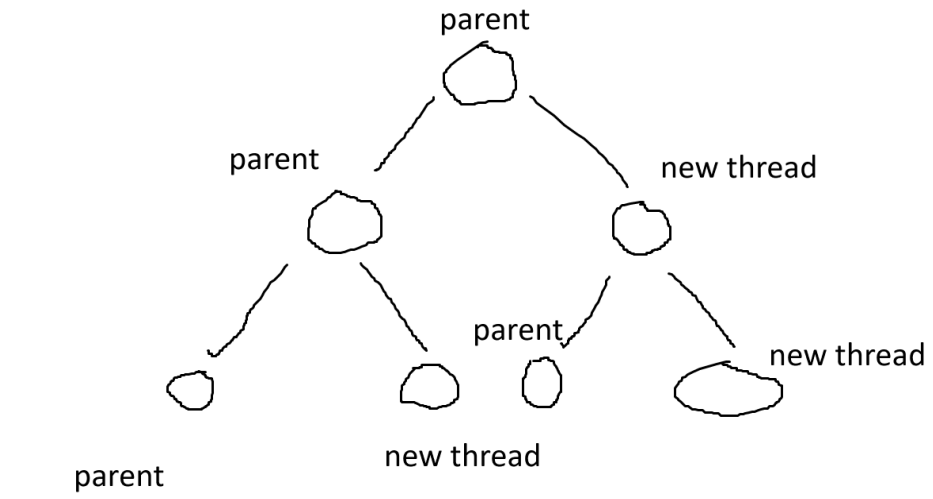
To implement a solution with the quicksort algorithm but using threads when doing the recursive function calls to make the algorithm parallel instead of sequential we first needed to really know how the quicksort algorithm worked.

So we did some research on quicksort and started to work on changing it. The normal quicksort algorithm makes two recursive calls with the higher and lower parts of the array. We needed to rethink this and really understand the data structure behind the algorithm and the parallelism that comes with threads.

It was a lot of trial and error but finally the idea is to only create one new thread with a recursive quicksort call. The current thread makes a new recursive function call on one subarray but not with a new thread. This makes for less overhead execution time, to save time not creating new threads. But still making the process parallel. If there are no free workers (threads) , all threads will just operate under normal quicksort procedures.

These performance results are done with the same code but different number of workers. The one with pthreads is with 10 workers and the normal quicksort is done with 1 worker, so it is sequential. The time that is spent creating threads is what takes time in the runs with a lower number of elements. But the parallelisms' quicker execution time are shown with the larger arrays.

Quicksort with Pthreads (s)	Quicksort normal (s)	Number of Elements
0,013	0,0001	10 000
0,014	0,012	100 000
0,11	0,13	1 000 000
0,24	0,28	2 000 000



This picture illustrates how the thread does quicksort on the left subarray and creates a new thread that does quicksort on the rightsubarray.