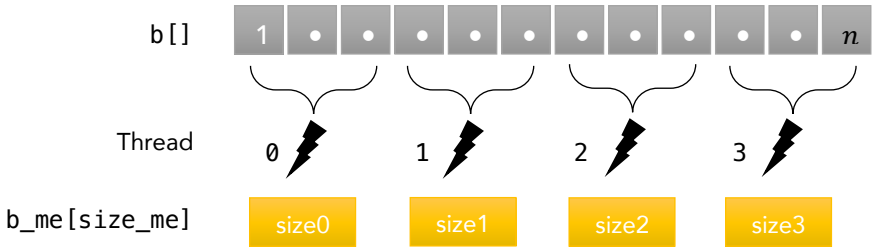
Methods

Header Files

I choose OpenMP to work with, because we'd need to do much more communication among the threads in case of MPI.

Strategy

Sorting problems are difficult to parallelize, because the order matters. Therefore, I couldn't think of any better solution other than splitting the work over the threads, and let each thread sort its array independently.



The sizes of the local arrays aren't equal, because the number of elements that fit into the corresponding interval is unpredictable. For instance, if thread `0` sorts all values that lay between `-990.987` and `-500.784`, we cannot tell beforehand how many elements are going to be there in the array. Therefore, we let each thread iterate over `b[]`, and select the elements that match its criteria.

Because the size of the arrays was unpredictable, I used dynamic allocation of arrays in my implementation.

A more optimized approach can be taken for the sorting process. Prior to the sorting, we can still split the one array into two arrays: one with positive elements, and one with negative ones. This way, we reduce the time need for sorting the negative from the positive values and would slightly improve in the time complexity of the quicksort algorithm by reducing n (see Table 1).

Sorting Algorithm

In that regard I chose an own sorting algorithm, namely the Quicksort, for multiple reasons:
a) I didn't seem to understand how the default sorting algorithm in `sort.c` works. I understood the part that `ind[]` is a *pointer-like* array, i.e. it references the indices of `b[]`, and that the order of `ind[]` is the correct order. However, I wasn't sure why b[0] = 1.
b) I wanted to design the program such that the sorting is achieved by a separate function just like we did in the Water exercise.

Algorithm	Time Complexity		
	Best	Average	Worst
Quicksort	$\Omega(n\log(n))$	$\theta(n\log(n))$	$O(n^2)$

Table 1: Time complexities of the Quicksort algorithm

In my implementation, there is an insignificant elapsed time after finishing reading the file and before sorting. This time is used in preprocessing the data in a desired way before kicking off. However, I used only 4 checkpoints to record the timing:

- Before reading (time0)
- After reading/Before sorting (time1)
- After sorting (time2), and prior to writing
- After writing (time3)

Ideally, we'd use 5 checkpoints of course, but because the preprocessing doesn't take any significant time, I stuck to the default implementation.

Gathering

The optimal solution for this exercise would be if we let each thread print its sorted array immediately to the file `sorted.txt`. However, we'd need to guarantee that the threads print in order (one after each other), just according to the slides in `e11_sorting.pdf`. However, the problem was that the clause ordered didn't work in a directive/parallel region. The loop had to be parallelized in order for the clause to work.

```
#pragma omp parallel
{
  .
  .
  .
  sorting…
  opening by 0
  appending by other threads
  .
  #pragma omp ordered
  for(i=0;i<size;i++) {
    fprintf(fp, "%lf %lf", b_me[i], c_me[i]);
  }
}
```

I didn't manage to get it to work in this fashion, so I used a *2-dimensional* shared array (`sorted_b[]`) to collect the sorted values from local arrays and store them globally. After that, I let thread master do the printing alone.

Submitted Files

- `quick_sort.c`: serial code of quicksort + the executable
- `quick_sorto.c`: parallel code of quicksort using OpenMP+ the executable
- `sorto_100k.xlsx`: Table reporting all wall clocks, when using 1, 2, 4, 8, 12 threads
- `sorted.txt`: file consisting of the sorted values found in 100k.txt
- `check`: an executable file to check whether or not the sorting is correct
- `outputs/`: the folder where the terminal output is saved after every run

Other data such as `1k.txt`, `10k.txt` and `100k.txt` isn't attached, because I assume these are obtainable from the cluster.