Programming Parallel Computers

Aalto 2023

SO: sorting

Please read the general instructions on this page first, and then check the individual tasks for more details. For each task you can download a zip file that contains the code templates you can use for development.

Task	Attempts	Expected	Points	Max	Rating	Rec.	Deadline for full points
SO4: merge sort Implement an efficient parallel sorting algorithm for the CPU, using the basic idea of **merge sort**.	0	–	–	5 + 2	★+	R	2023-05-21 at 23:59:59
SO5: quicksort Implement an efficient parallel sorting algorithm for the CPU, using the basic idea of **quicksort**.	0	–	–	5 + 2	★+	R	2023-05-28 at 23:59:59
SO6: fast GPU solution Implement an efficient parallel sorting algorithm for the **GPU**. Any sorting algorithm is fine, but **radix sort** is perhaps the simplest choice.	0	–	–	5 + 2	★★★	R	2023-06-04 at 23:59:59

General instructions for this exercise

In this task, you will implement parallel sorting algorithms that outperform the single-threaded `std::sort`.

Interface

We have already defined the following data type that represents 64-bit unsigned integers:

```
typedef unsigned long long data_t;
```

You need to implement the following function:

```
void psort(int n, data_t* data)
```

Here `n` is the size of the input array `data`. All input elements are of type `data_t`, i.e., 64-bit unsigned integers.

Correct output

Your function should have exactly the same behavior as:

```
std::sort(data, data+n);
```

That is, after calling your function, the array should be in a sorted order. You are free to allocate some additional memory to keep temporary data if needed.

Rules

In your implementation, you are permitted and encouraged to use **all single-threaded features of the C++ standard library**. In particular, you can freely use the single-threaded `std::sort` as a subroutine in your code, as well as all other features of the **algorithms library**.

For multi-threading, you can use the basic primitives of OpenMP, as usual.

Please **do not hard-code the number of threads**; please let OpenMP choose the number of threads, and use e.g. **functions omp_get_max_threads and omp_get_num_threads** if you need to know the number of threads. This way your code should automatically respect e.g. the `OMP_NUM_THREADS` environment variable. Your code should work **reasonably efficiently** for any number of threads between 1 and 20, and it should work **correctly** also for a larger number of threads. For example, your program must not crash if we change `OMP_NUM_THREADS` from 20 to 21, and your program should not immediately fall back to a single-threaded implementation if we change `OMP_NUM_THREADS` from 20 to 19.

For the GPU exercise, you have to stick to the basic CUDA API. In particular, you are **not allowed to use Thrust**.

In the merge sort task and quicksort task, it is perfectly fine to resort to the single-threaded `std::sort` in the base case (regardless of what algorithm the standard library happens to use internally); the key requirement is that the multi-threaded parts are based on the idea of merge sort and quicksort, respectively. For merging and partitioning, you are free to use any strategy, and you are encouraged to use also e.g. vector instructions whenever possible.