Final Project Report: Odd-Even Sort

Federico Finocchio

Contents

1 Introduction

The selected project is the **Odd-Even Sort** implementation.
The problem consists in sorting a vector using consecutive pair-wise comparison.

It is divided in two *phases*:

- **Phase 1**: pair-wise comparison on even-indexed elements;
- **Phase 2**: pair-wise comparison on odd-indexed elements;

At each phase the algorithm proceeds by swapping the pair of elements that are not in order (i.e. a[i] > a[i+1]) and it stops when no element swaps are performed at the end of the second phase.
Parallel implementations are based on the parallelization of the two phases with synchronization between each of them due to the sequential structure of the original algorithm.
Next sections will walk through the project structure, main implementation choices, performance improvements and which are the tests executed to get them.

2 Project Structure

The project was developed with the aim to achieve good results using the main optimization's mechanisms seen during lectures, for example *vectorization*, *thread pinning* and taking care of *false sharing*.

The project is composed by various files, the main ones are:

- **oe-sortseq.cpp**: sequential implementation of the Odd-Even Sort algorithm.
- **oe-sortparnofs.cpp**: parallel implementation using only C++ standard mechanisms.
- **oe-sortmw.cpp**: parallel implementation using FastFlow library.

Additional files provided are:

- **test.sh**: bash script used to run a fixed number of time the same configuration;
- **stats.sh**: bash script used to make the average completion time of the same execution ran through *test.sh*;

All the **.cpp** files can be compiled using the *make* command as described in the **Results** section.

The mandatory parameters change between different version, fixed ones are:

- **seed**: used for random number generation during initialization phase;
- **length**: number of elements to insert in the vector to be sorted;

- **max**: maximum value to be inserted;

The specific parameters for the two parallel versions are:

- **nw**: number of workers;
- **cache-size**: cache line size represented in bytes, used for padding;

Parameters order must be: **seed**, **len**, **nw**, **cache-size**, **max**

2.1 Implementation

All the versions are based on the **std::vector<int16_t>** data structure, it is initialized using random numbers generated using as seed the related parameter provided at execution time. The first choice concerns the usage of **int16_t** data type, changing the default *int* data type to this one lead to a non-negligible performance improvement due to the fact that in this way a greater number of elements can be processed by SIMD operations.

Further optimizations were applied to the sequential code in order to achieve the best sequential execution time, these optimizations are **vectorization** and **thread pinning**.
The former one was applied refactoring the code in such a way the C++ compiler could apply such optimization. For that reason the main loop is composed by two inner loops to better represent the two phases with no *phase-flag* switching.
The latter was initially applied to optimize cache utilization, but no noticeable performance improvement was recorded even if during long computations the running thread was moved to different cores.

Besides these shared optimizations, further implementation choices are specific to different parallel versions in order to achieve better results based on the used mechanisms/framework.

2.1.1 Parallel versions

Parallel versions are based on parallelization of each *phase loop*. Given that all phases have to be executed sequentially, between each of them synchronization is needed to await threads still running the current phase.

To assign regions the **Range** data structure is used. It contains the assigned region of the starting vector on which the threads have to perform sorting, but due to padding applied to the original vector, the real starting position is recomputed during initialization phase.
The resulting vector, after padding, contains for each worker the assigned region of the original one with an extra element that is the copy of the first element in the next assigned region. This choice comes from the necessity of avoiding concurrent updates to shared elements (e.g. elements used in odd-phase by $worker_i$ and in even-phase by $worker_{i+1}$).
Each worker updates its first/last element, depending on the current phase, retrieving it directly from the region of nearby workers. These updates, performed at the start of each

phase, require no lock/synchronization mechanisms because at a given phase no workers need the same element at the same time.

Specific implementation choices are the following:

- **oe-sortparnofs.cpp**: each thread keeps sorting the assigned region until the global exit condition is met. Between each phase workers wait for each other using **Barrier** objects. Two of them are needed to let synchronization work properly and to update/reset the exit condition.
- **oe-sortmw.cpp**: based on the *Master-Worker* pattern, **Master** node schedules task using **Task** objects containing the current phase and local exit condition. It checks for exit condition and let workers synchronize between each phase.
  Workers keep sorting the same region, starting at different positions based on current phase, until an **EOS** message is received.

Differently from sequential version, in these cases thread pinning led to better results in terms of total completion time.

3 Usage and Results

All the tests were run on the remote XeonPHI machine provided, main results obtained are showed in this section, along with instructions to reproduce them and to execute further tests.

The main plots provided are relative to *speedup* and *scalability*, showing how they increase with increasing vector lengths, due to the fact that greater is the vector size, smaller is the percentage of time spent to synchronize threads between phases.

A **Makefile** is provided to compile, run and test all the implementations.
To compile the code you can run:

```
$ make all
```

To run tests you have to specify which kind of version you want to use (using different make targets) and which are the parameters of the execution using **SEED**, **LEN**, **CACHE**(expressed in bytes), **NW** and **MAX**(defaulted by MAX=INT16_MAX), like:

```
$ make test-par SEED="1234" LEN="500000" NW="32" CACHE="64"
```

Statistics used for plotting were retrieved using the **test.sh** bash script, like:

```
$ ./test.sh seq 1234 10000 1000
```

And:

```
$ ./test.sh parnofs 1234 10000 1000 64 64
```

Parameters must be specified in the order **seed**, **len**, **max**, **nw**, **cache-size**

3.1 Tests

All the tests were run on the remote machine with parameters **SEED="1234"**, **MAX="1000"** for all the versions and **CACHE="64"** for both parallel versions.
Other parameters were used to study how the different implementations behaved with increasing vector lengths and number of workers.

3.1.1 Expected results

The expected results were computed starting from the various execution times retrieved running the sequential version.
A rough estimated execution time for both parallel versions can be computed starting from the average time spent *per phase*, multiplied by the total number of phases (i.e. the completion time of the sequential version) and divided by the number of workers. In this way we are assuming that the number of phases is proportional to the length of the vector, but that's not the case, so we can only get an estimate of the total completion time for the parallel versions. Actually the total number of phases changes based also on the vector length, seed used for random number generation and max elements value.

As can be seen in Figure 1 and Figure 2 the expected results are met up to **nw=16**.

```
[seed=1234 len=500000 max=1000]
tot_avg: 363145912
phases: 249420
avg_p1: 342.331
avg_p2: 1111.93
```

Figure 1: Sequential execution time (usecs)

```
[seed=1234 len=500000 nw=128 cache=64 max=1000] -- avg: 22646090.000
[seed=1234 len=500000 nw=64 cache=64 max=1000] -- avg: 16018974.000
[seed=1234 len=500000 nw=32 cache=64 max=1000] -- avg: 14805058.000
[seed=1234 len=500000 nw=16 cache=64 max=1000] -- avg: 22785663.000
[seed=1234 len=500000 nw=8 cache=64 max=1000] -- avg: 42610009.000
[seed=1234 len=500000 nw=4 cache=64 max=1000] -- avg: 84679358.000
[seed=1234 len=500000 nw=2 cache=64 max=1000] -- avg: 166298390.000
[seed=1234 len=500000 nw=1 cache=64 max=1000] -- avg: 359196732.000
```

(a) Thread version

```
[seed=1234 len=500000 nw=128 cache=64 max=1000] -- avg: 59690239.000
[seed=1234 len=500000 nw=64 cache=64 max=1000] -- avg: 27171997.000
[seed=1234 len=500000 nw=32 cache=64 max=1000] -- avg: 16903349.000
[seed=1234 len=500000 nw=16 cache=64 max=1000] -- avg: 25030367.000
[seed=1234 len=500000 nw=8 cache=64 max=1000] -- avg: 45925628.000
[seed=1234 len=500000 nw=4 cache=64 max=1000] -- avg: 89182639.000
[seed=1234 len=500000 nw=2 cache=64 max=1000] -- avg: 177695796.000
[seed=1234 len=500000 nw=1 cache=64 max=1000] -- avg: 371632122.000
```

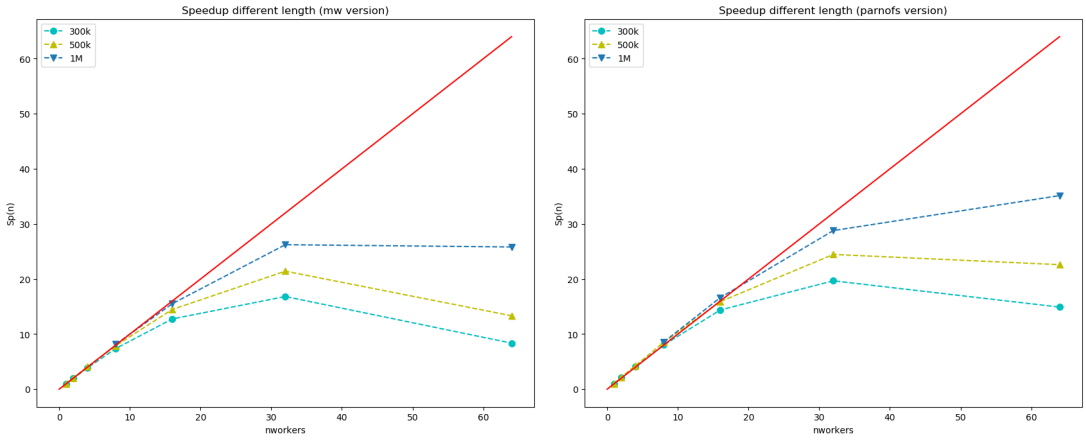
(b) FastFlow version

Figure 2: Parallel execution time (usecs)

Increasing the number of workers leads the expected results to be lower than the actual one, especially due to the synchronization overhead between each phase.
In the next section will be shown that this overhead decreases with longer vectors, in fact the speedup increases along with the vector length.

3.1.2 Speedup and Scalability

The results obtained are shown in this section through speedup and scalability plots. In Figure 3 are shown the speedups for three different configurations of the initial vector, namely 300k, 500k and 1M elements.
For both versions, increasing the vector length for a fixed *nw* led to an increase in the speedup value. In fact, for 1M elements vector, the achieved speedup is grater than the ideal one up to **nw=16**. We can also see that for *FastFlow* version the increase in speedup is lower, due to the communication between workers and master nodes. This condition is visualized in Figure 4, which shows the changes in speedup values for increasing length with fixed **nw=64**.
The increasing trend of speedup is also present for all the other values of **nw**.



(a) Speedup M-W version

(b) Speedup thread version

Figure 3: Speedup comparison

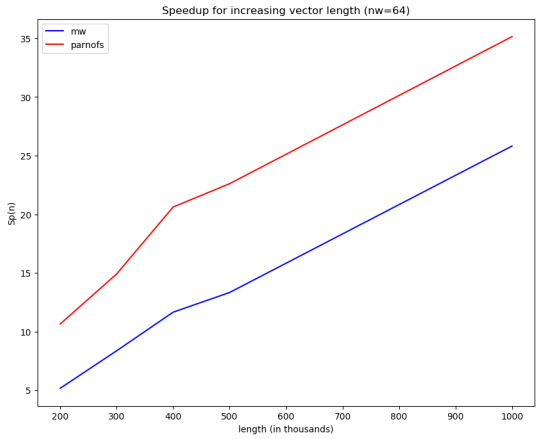
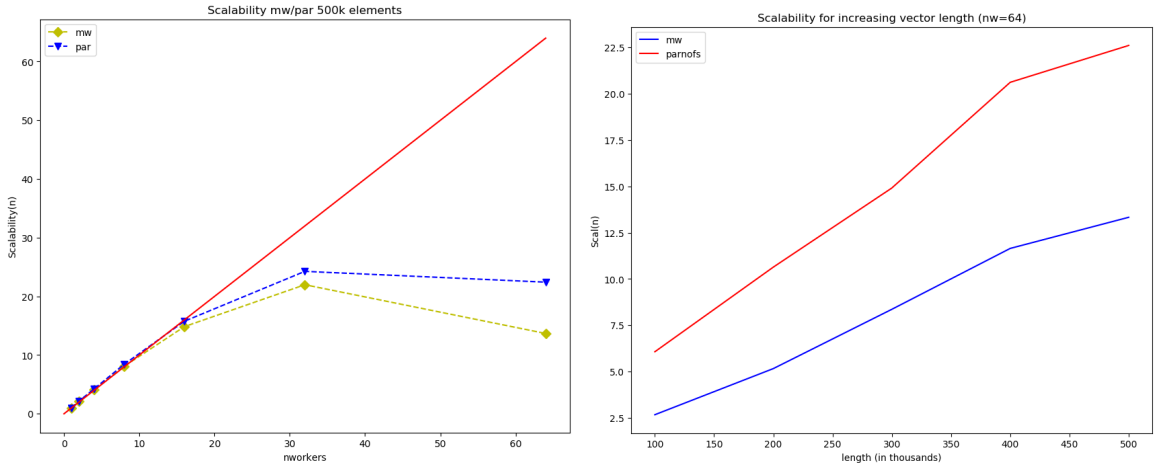


Figure 4: Speedup with increasing vector length

A similar consideration can be done for scalability. In Figure 5.a is shown the comparison between the achieved scalability values for both version with a vector of 500k elements. As done for speedup, in Figure 5.b we can see how scalability changes based on the vector length.



(a) Scalability comparison

(b) Scalability for increasing length

Figure 5: Scalability parallel versions

4 Changelogs

The first version provided as project submission missed the vectorization of the main loop in both parallel versions, even if the compiler shown that it was applied.
Changes were necessary to transform loops formerly starting from a pre-computed variable to loops starting from a constant value, in this case **0** for even phases and **1** for odd phases. Some minor changes were applied to remove junk/debugging code and a memory leak still present in the *FastFlow* version.

Changes are:

- range assignment: ranges are assigned in such a way that all the workers get a region of the original vector starting from an even index. Modifications were relative to the **assignRanges** function;
- local vector: threads work on a local vector generated at the starting point of each thread. *Threaded* version uses a "*logical*" local vector, in fact each thread shifts the vector pointer to region starting point and proceeds sorting. *FastFlow* version's **Worker**s create a new vector copying elements from the assigned region, sort until **EOS**, then copy back sorted local vector to the original one;
- no phase switching: **Worker**s now have two for loops (one per phase) to avoid switching the starting index of phase's *for* loop;
- memory leaks: fixed a memory leak in *FastFlow* version, **Master** node now correctly deletes tasks assigned to workers;

- alignment of sequential code: refactoring of sequential code to align it with other versions, added documentation string to functions and changed unclearly named variables;
- other minor changes: refactoring/cleaning of junk code;