# SPM Project Report
# Odd-Even Sort

Marco Lepri
m.lepri2@studenti.unipi.it

A.Y. 2019/20

## Contents

# 1 Introduction

## 1.1 The Problem

The Odd-even Sort is a sorting algorithm where each iteration is made of two phases: a "Even" phase and a "Odd" phase.
In the Even phase the elements in the even positions are compared with their subsequent element. If the two elements are out of order then they are swapped, otherwise they are left unchanged. In the Odd phase the process is repeated but this time the elements in the odd positions are compared with their successors.
The algorithm goes on until no swap is performed in either of the two phases. Therefore, each iteration has linear complexity in the number of elements $N$, while the whole algorithm has a complexity that is quadratic in $N$.

## 1.2 Presented Versions

In the following, 4 different versions of the algorithm are presented:

- A sequential version (file `odd-even-seq.cpp`), used to gather general statistics about the problem and as a baseline for the evaluation of the other versions
- Two parallel versions using `c++` pthreads, one with a static division of the work (file `odd-even-par-static.cpp`) and one with dynamic scheduling of the tasks among the workers (file `odd-even-par-dyn.cpp`)
- A parallel version using FastFlow as framework for the creation and management of the parallel activities (file `odd-even-ff.cpp`).

## 1.3 Compiling and Execution

All the tested code have been compiled with
`g++ -g -O3 -std=c++17 -Wall -ftree-vectorize -pthread -I ./Include`.
To repeat the experiments it is suggested to use the same compiling options so that the results match.
By default, all the implementations print some statistics at the end of the computation, like the total completion time, the number of iterations and the average time spent in each iteration. In case you want to see more detailed statistics, including the (averages) time for each phase, the number of swaps and averages for the overheads in the parallel versions, compile adding `-DSTATS`. Moreover, if you want to display the vector content after every phase, you can compile adding `-DPRINT`.
Note that, in these cases, the final total time will consider also the time spent to gather all the statistics or to print the vectors and, hence, it should not be taken as the real completion time.
All the implementations are controlled by a number of command line parameters:

- N : number of elements in the array
- niter : number of iterations (optional)
- seed : seed for the problem generation ($-1 \Rightarrow$ reversed array)

- nw : number of workers (only for parallel versions)
- chunksize: size of a single computation (only for `odd-even-par-dyn.cpp` and `odd-even-ff.cpp`)

# 2 Analysis

## 2.1 Problem analysis

A single iteration of the algorithm can be seen as the sequential execution of two map-reduce, one corresponding to each phase. The map function takes two subsequent elements and swaps them if they are out of order, otherwise it leaves them unchanged. The reduce function is used to obtain a single value stating whether there was at least one swap in the map computation, to be used for the stopping condition of the algorithm. Therefore, the reduce function computes the logical OR of the conditions used to decide whether swapping the elements or not. This allows the computation of the map and of the reduce to happen simultaneously, at least in the sequential case. In the parallel case a further phase to collect the partial reduce results of each worker and compute a final reduce operation is still required.
Therefore, let $T_{seq} = 2 \times T_{map-red}$ be the sequential completion time for a single iteration, with $T_{map-red}$ equal to the time for a single map-reduce computation. Ideally, one could achieve a parallel completion time, with $nw$ workers, equal to $T_{nw} = \frac{2 \times T_{map-red}}{nw} + T_{reduce}$ per iteration, where $T_{reduce}$ is the time spent to gather all the partial reduce results from the workers. This final time can be considered only once as the final reduce can be computed on the partial results coming from both the map computations.
However, each phase can start only when the previous one is completely finished, in order to avoid accessing elements that still need to be swapped. This requires some synchronization mechanism among the workers which adds some more overhead that must be considered. The parallel completion time for a single iteration becomes then

$$T_{nw} = \frac{2 \times T_{map-red}}{nw} + T_{reduce} + 2 \times T_{synch}$$

where $T_{synch}$ is the time spent for synchronizing the two phases. It is considered two times because the synchronization is required also at the end of each iteration before starting the next one.
Of course the above approximation considers the workload as perfectly balanced among the workers. However the map function is not guaranteed to take the same amount of time for all the elements because of the conditional swap. This might create load balancing problems, especially in case of a static division of the work. Although it is reasonable to think that, for a randomly filled vector, the workload is evenly distributed, as the algorithm proceeds also the workload changes since the vector is closer to be sorted and less elements require a swap. Therefore, in the following, both a version using a static division of the work and a version using auto scheduling are presented and analyzed.

## 2.2 Alternatives

An alternative solution to the problem could be to consider a single iteration as the pipeline execution of two maps. In the first stage a portion of the vector is processed as in the Even

phase while in the second stage the portion is processed as in the Odd phase. Problems could arise because the limit elements of each portion might need to be compared with the limit element of a portion that is still in the previous phase. However, this could be easily solved arranging the intervals so that the second stage works always on portions of the vector fully processed by the first stage. This avoids having the synchronization between the two phases, although the synchronization between two iterations is still required.
This would lead to a parallel completion time per iteration equal to

$$T_{pipe} = T_{chunk} + T_{map-red} + T_{reduce} + T_{synch}$$

where $T_{chunk}$ is the time to process a single vector chunk and corresponds to the time to fill up the pipeline stages, while $T_{reduce}$ and $T_{synch}$ are still the overheads for the final reduce and the synchronization between two iterations.
Of course, each stage can be further paralellized, computing the map-reduce with more than one worker, obtaining the parallel completion time (per iteration) of

$$T_{nw} = \frac{T_{chunk}}{nw/2} + \frac{T_{map-red}}{nw/2} + T_{reduce} + T_{synch}$$

where the number of workers have been evenly divided among the two stages.
Compared to the previous solution, we have one synchronization overhead less. However the time to fill up time pipeline might be much higher than this synchronization time, depending on the number of workers and on the chunk size. Indeed, a small chunksize leads to a shorter time for the pipeline to fill up. However, making it too small risks to reduce the single work piece down to the point where the overheads take as long as the parallel computation. On the other side, a big chunksize mitigates the overheads cost, but the pipeline fill up time would then be much higher than the spared synchronization overhead.

## 3 Sequential Analysis

As said before, the sequential completion time for a single iteration is $T_{seq} = 2 \times T_{map-red}$ where $T_{map-red}$ is the time to perform a single map-reduce computation. Of course, this time depends on the number of elements $N$ and from the time to execute the map function once, $T_f$. In particular, in each phase, the map-reduce function is computed $N/2$ times as each execution takes two elements. Therefore, the completion time for a single iteration is $T_{seq} = N \times T_f$.

The map function is implemented with an explicit `if-then` statement that checks whether the two elements are in order. In case they are not the `then` branch is taken and the two elements are swapped. Otherwise the function just returns. Of course, the swap is performed in place and not using a support array. This makes the execution time dependent from the given elements, since an execution on two elements in order takes a different number of instructions compared to an execution on two elements out of order.
In order to have an estimation of $T_f$, the sequential version have been executed on a number of problems, allowing it to gather extended statistics (`odd-even-seq.cpp` compiled with `-DSTATS` flag). Table 1 shows the average map-reduce function execution time and the average iteration execution time.

| Problem | $T_f$ | $T_{seq}$ |
|---|---|---|
| N=10K | $5.5 \sim 5.8$ ns | $\sim 57$ $\mu$s |
| N=50K | $5.5 \sim 5.7$ ns | $\sim 282$ $\mu$s |
| N=100K | $5.5 \sim 5.8$ ns | $\sim 565$ $\mu$s |
| N=1M, niter=100 | $7.5 \sim 7.9$ ns | $\sim 7742$ $\mu$s |
| N=1M, niter=1000 | $7.5 \sim 8.0$ ns | $\sim 7778$ $\mu$s |
| N=1M, niter=1 | $5.8 \sim 5.9$ ns | $\sim 5910$ $\mu$s |

Table 1: Execution time averages

$T_f$ is always above 5 nanoseconds, although it reaches 7-8 nanoseconds in some cases. This depends on the task at hand: in the case a number of iteration is given, the vector is generated such that it could be sorted in, approximately, that number of iterations, while, in the other cases, the vector is generated completely randomly. In the latter case, the final iterations of the algorithm performs less swaps since the vector is closer to be sorted. This makes the average $T_f$ more similar to the average in the case of a single iteration, where the vector is already sorted and no swap is performed.

Such a small $T_f$ makes the number of element in the vector crucial for the parallelization to be worth the effort. A vector of 100K elements takes, on average, 600 microseconds to go through an entire iteration, 300 each phase. Dividing this work among 8 threads means having a single work piece last for ∼35-40 microseconds, before a synchronization is needed. An overhead of even 4 microseconds means the 10% of the total time is spent synchronizing the threads. Therefore, good speedups with a large number of workers are expected in case of vectors with tens of millions of elements, while for few hundreds of thousands of elements, the efficiency is expected to decrease quickly as the number of threads grows.

# 4 Parallel Analysis

## 4.1 Static Scheduling

The first parallel version considered uses a static division of the workload among the threads: each thread computes, internally, the portion of the vector it has to work on, both for the Even phase and for the Odd phase.
The threads are activated only once at the beginning by the main thread, which will remain active, acting like the *master* in a *master-slave* fashion. The threads are then synchronized using active wait barriers (file `Include/ActiveBarrier.cpp`).
Two barriers are used: one to synchronize the execution of the two phases within the same iteration and one to synchronize the threads after every iteration. In the first case, the activated threads wait for all of them to reach the barrier and then continue the execution, without any intervention of the main thread. In the second case, the threads wait for the main thread to reset the barrier. This is done because the shared structures, including the barriers but also the reduction variable, need to be reset before starting the next iteration. The reduce phase is handled simply letting the single threads update a shared, atomic variable, adding their private, partial results computed along the two phases, rather than letting the main thread compute a final reduce on the workers' partial results. Table 2

| Problem | nw | barrier1 | barrier2 | update |
|---|---|---|---|---|
| N=100K, seed=-1 | 2 | 0.3 - 0.8 $\mu$s | 1.2 - 2.5 $\mu$s | $\sim$0.3 $\mu$s |
| | 4 | 0.5 - 1.3 $\mu$s | 1.7 - 4.1 $\mu$s | $\sim$0.3 $\mu$s |
| | 8 | 1.8 - 2.1 $\mu$s | 2.5 - 6.1 $\mu$s | $\sim$0.3 $\mu$s |
| N=100K, seed=42 | 2 | 0.8 - 0.9 $\mu$s | 1.5 - 1.8 $\mu$s | $\sim$0.3 $\mu$s |
| | 4 | 0.7 - 28 $\mu$s | 2.0 - 27 $\mu$s | $\sim$0.3 $\mu$s |
| | 8 | 6.1 - 25 $\mu$s | 7.4 - 26 $\mu$s | $\sim$0.3 $\mu$s |
| N=1M, niter=1000, seed=42 | 2 | 1.5 - 5.4 $\mu$s | 1.7 - 5.2 $\mu$s | $\sim$0.6 $\mu$s |
| | 4 | 2.4 - 8.8 $\mu$s | 2.7 - 9.1 $\mu$s | $\sim$0.5 $\mu$s |
| | 8 | 3.3 - 9.8 $\mu$s | 3.7 - 9.4 $\mu$s | $\sim$0.3 $\mu$s |

Table 2: Overheads for the static parallel version

shows the impact of the overheads in three different problems. It is clear how the major sources of overhead are the two barriers, while the update time for the shared variable is negligible, thanks to the atomic operations than avoid the usage of mutexes.
The overheads for the barriers are usually in the range of few microseconds, although it reaches much higher value in some cases. This, together with the fact that the average waiting time can be very different depending on the considered thread, are the effect a bad load balancing. Indeed, those threads that have a lower average phase completion time also have an higher waiting time at the barriers, which is clear from the results in case of 100K elements. This shows how, in the general case, the workload is not balanced if a static partitioning is used which heavily effects the efficiency of the solution.
To overcome this problem in evaluating the impact of overheads, made up problems have been used, so that the workload is perfectly balanced. One of these is the case of the reversed vector, showed in table 2 in the first lines, for which all the elements are swapped, in all the phases and iterations. In this case we can see how the barriers' overheads never exceeds ten microseconds, although it seems to increase with the number of threads. The second barrier usually introduces a longer overhead, which is coherent with the fact that the threads wait for the main thread to reset the shared structures.

Therefore, the main problem and limitation of this solution is the load balancing. Figure 1 shows the speedup in case of a general problem with $N = 100K$. Because of the small size of the problem, the poor speedups for more than 5 threads are expected. However, already with 3 threads the speedup reduces, showing how the load balancing is not achieved and some threads waste a lot of time waiting at the barriers. This behaviour is not observed for problems like those in figures 2 and 3. This is both because the problems are much larger, which let the solution achieve good speedups even with many threads, but also because of the way the problems are build, in order to be possible to solve them in the specified number of iterations. Therefore, they should not be taken as general results as in the general case, the load balancing is rarely achieved.

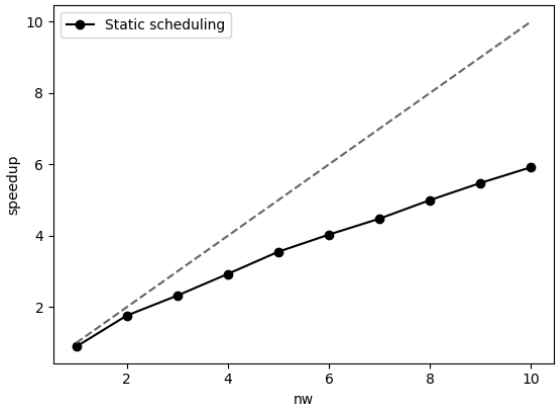


Figure 1: N=100K

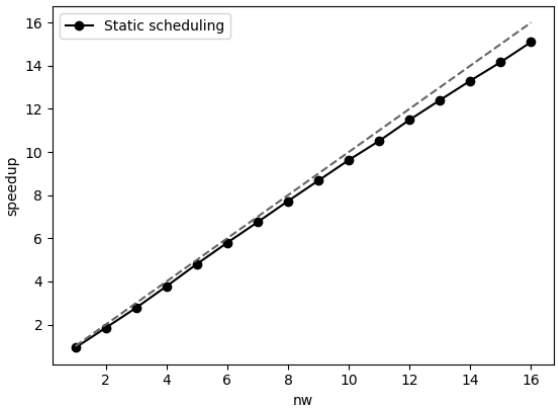


Figure 2: N=1M, niter=1000

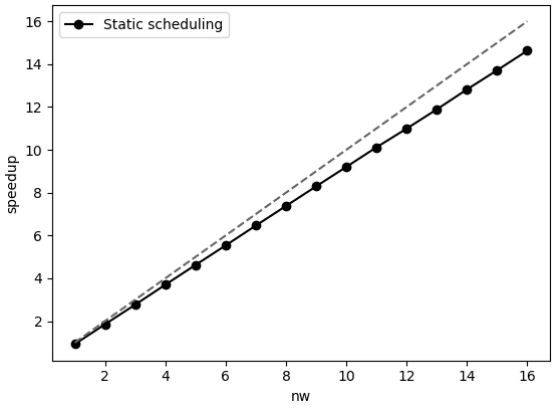


Figure 3: N=10M, niter=100

### 4.2 Auto Scheduling

In order to prevent the load balancing issues, one idea could be to use a dynamical scheduling policy. In this case, an auto-scheduling policy is used: the threads access a shared data structure from which they get the next task to execute. Here, the tasks correspond to chunks of the vector on which the map-reduce function need to be executed. The chunksize is then a further parameter of the solution. In order to maintain the scheduling overhead as small as possible, the shared data structure is implemented with atomic variables and is accessed and modified using atomic operations.
Even in this case, the threads are activated once at the beginning and are then synchronized using active wait barriers. The final reduce is still handled using a shared, atomic variable updated by the single threads. A difference with the static case is that now the activated threads wait for the main thread to reset the shared structures after each phase, instead of going on independently until the end of the iteration. This is supposed to increase the barriers' overhead w.r.t the previous case.

Table 3 shows the scheduling overheads and barriers overhead for 1 million elements and 1000 iterations with two different chunksizes: 10K and 100K.The update overhead is omitted as it is in the same range as in the static case.

| Problem | nw | Avg task exec. | Avg task retrieve | Avg phase | Avg barrier |
|---|---|---|---|---|---|
| chunksize=10000 | 2 | ∼39 $\mu$s | ∼0.42 $\mu$s | ∼1960 $\mu$s | 3 - 7 $\mu$s |
| | 4 | ∼41 $\mu$s | ∼0.46 $\mu$s | ∼1060 $\mu$s | ∼15 $\mu$s |
| | 8 | ∼42 $\mu$s | ∼0.53 $\mu$s | ∼538 $\mu$s | ∼23 $\mu$s |
| chunksize=100000 | 2 | ∼385 $\mu$s | ∼0.54 $\mu$s | ∼1920 $\mu$s | 2 - 5 $\mu$s |
| | 4 | ∼411 $\mu$s | ∼0.69 $\mu$s | ∼1030 $\mu$s | ∼190 $\mu$s |
| | 8 | ∼418 $\mu$s | ∼1.2 $\mu$s | ∼520 $\mu$s | ∼290 $\mu$s |

Table 3: Overheads for the auto-scheduling version

Thanks to the mutex-free implementation of the shared data structure, the single task retrieving overhead is kept, on average, around or under 1 microsecond, although it seems to depend also on the number of threads. In both cases, this represents a small fraction of the time spent for a single task, making the overall scheduling overhead negligible if compared to the overall phase completion time. Of course, this depends on the chunksize, since making it smaller means having a larger number of task to retrieve.
The barrier overhead seems to vary a lot, depending on the number of threads and on the chunksize. This is not surprising if we think that, increasing the number of threads, it is more likely that the workload cannot be evenly divided among all of them. Therefore, some are going to wait at the barrier while others are still in the middle of the computation. The waiting time depends, of course, also on the chunksize, since the threads waits, in the worst case, as much time as it is needed to process a single chunk.

Therefore, the chunksize represent a crucial parameter of this implementation as the overall performance deeply depends on its value. Figures 4 and 5 show the speedup in the case of $N = 100K$ with chunksize=5000 and in the case of $N = 10M$ with chunksize=100K. Again, the small size of the first problem does not make the parallelization worth the effort and makes it difficult to observe the effect of the load distribution.
In the second case, the chunksize corresponds to the 1% of the total size. This keep the overall waiting time at the barriers small, in percentage, allowing the solution to achieve good speedups, no matter the large number of tasks each thread retrieves. However, the

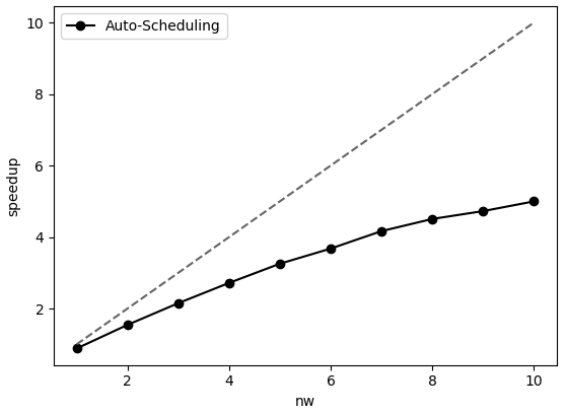


Figure 4: N=100K

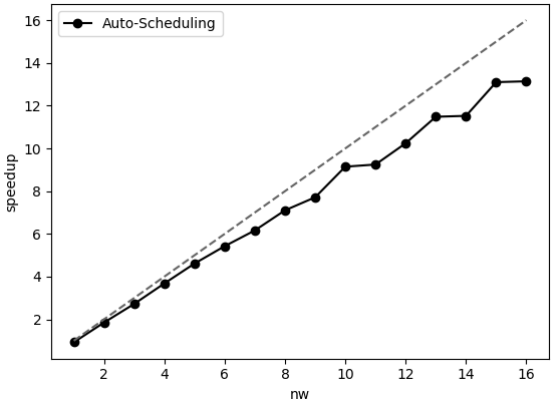


Figure 5: N=10M, niter=100

strange behaviour with $nw > 10$ shows how the workload cannot always be evenly divided among the threads and that some resources will be wasted. Therefore, the chunksize should be chosen taking into account also the number of thread used for the computation and not only the total length of the vector.

### 4.3 FastFlow version

Both the version with static scheduling and the version with auto-scheduling can be easily implemented using FastFlow library. In particular, this version uses the `ParallelForReduce` object in order to set up a number of threads to be used for the computation. Even in this case, the object, and so the threads, are created once at the beginning. In order to match this implementation with the previous two, the scheduler for the object is disabled. Each phase is then implemented as a call to the `parallel_reduce` method of the above object, each phase with the right starting index and ending index.
To control whether using the static scheduling or the dynamic one, the chunksize parameter is used: a chunksize equal to zero corresponds to a static scheduling, while a chunksize greater than zero corresponds to the auto-scheduling with that chunksize.

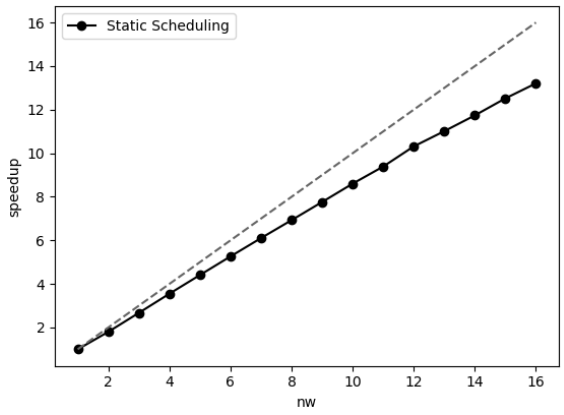


Figure 6: N=100K

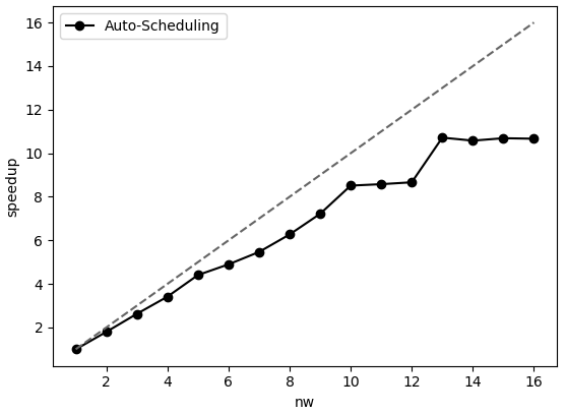


Figure 7: N=10M, niter=100

Figures 6 and 7 show the speedup for $N = 10M$ and 100 iterations. In the case of the auto-scheduling, a chunksize of 100K is used.
The static FastFlow version behaves more or less as the plain `c++` one. The difference in the speedup could be due to the fact that in the first one the method is called by the main thread while, in the second case, the execution of the two phases happens independently from the main thread, whose intervention is needed only at the end of the iteration.
The auto-scheduling FastFlow version maintains a fairly good speedup up to $nw = 10$, although having some problems in scaling well with any number of threads. This could be explained by the workload not being perfectly divided among the workers. More difficult to explain is the behaviour for $nw > 10$ as increasing the number of threads seems not to increase the performance in all the cases. This might be because of the actual implementation of FastFlow or more simply because of a very bad load balancing. Indeed, trying to vary the chunksize and/or the size of the array seems to lead to better results, although not completely solving the odd behavior for larger number of threads.

# 5 Conclusions

We looked at the Odd-even sort algorithm and investigated some possibilities for its parallelization. In particular, the algorithm was seen as the sequential computation of two map-reduce operations each iteration. Two main solution were presented: one that uses a static partitioning of the workload among the workers and one that uses the auto-scheduling technique for the tasks division. These solutions introduce different kind of overhead and problems which were analyzed and discussed.

In the general case the biggest problem seems to be the load balancing. In fact, the workload is not fixed, but vary as the algorithm proceeds with the iterations. This leads to a poor performance of the static partitioning version, which however achieves very good speedups in case of more balanced problems, thanks to the small overheads that it introduces.
The auto-scheduling solution, although it solves, at least in theory, the load balancing problem, heavily depends on the chosen chunksize. This have to be a good tradeoff between a large enough size, so that the scheduling overheads are neglected by the single task completion time, and a small enough size, so that the workload is evenly divided more easily and less resources are wasted.
Finally, we looked at a possible implementation of the two solutions using FastFlow library. Despite the fact that these versions does not seem to achieve results as good as those obtained with plain `c++` and `pthreads`, the library allows an extremely easy parallelization of the algorithm if compared to the "handmade" implementation of all the mechanisms and structures. This is surely a great advantage in case of more complex problems than this simple algorithm.
However, some results are still not easily explained and might require a deeper analysis or investigation in order to fully understand them.