# Parallel Odd-Even Sort

Lorenzo Beretta (lorenzo2beretta@gmail.com)

6th July 2020

## 1 Introduction

In this report we implemented two parallel versions of the Odd-Even Sort (OES) algorithm. The algorithm is pretty simple and works as follows: we proceed repeating identical iterations, each of which consists of two scans over the vector to be sorted; the first scan compare elements in even positions with their successors and, if out of order, swaps them, the same happens for odd-positioned elements in the subsequent scan. For the rest of this report we assume to sort a vector $v$ of length $n$; it can be proven (Wikipedia Odd-Even Sort) that $n$ such identical iterations are sufficient to sort the vector, leading to a sequential complexity of $\mathcal{O}(n^2)$, sub-optimal with respect to the well known $\mathcal{O}(n \log(n))$ alternatives. However this algorithm presents a strong data independence (i.e. its data flow graph is moderately interconnected), therefore we can aim at achieving interesting speedups with a parallel version.

## 2 Design Choices

The most trivial parallel implementation of OES exploits the data parallelism that is inherent to the inner for loops; we can simply run two parallel for (one for the even phase and one for the odd one) inside a sequential loop. An OMP implementation of this pattern is provided (`openmp.cpp`), however the experimental results showed that the parallel version introduce so much overhead that it is not even compensated by the speedup, so that the parallel version is slower than the sequential one. We implemented the exact same pattern implementing the barrier and the parallel for by myself (`pthread-barrier.cpp`) and again it yielded awful performances. Therefore

we decided to get rid of the barriers, it required some effort and we needed to add some additional structures and to disrupt the iterative form in which the sequential version has been describing. In the next section we describe two different patterns implemented respectively using native C++ threads and the Fastflow library.

# 3 Pthread Version

We refer to the file `pthread-async.cpp` that is widely commented, consider reading the source code for a thorough explanation of implementation details. The basic idea here is that, given $nw$ workers, we can partition our vector $v$ in $nw$ chunks, so that each worker only deals with a single chunk. This is more or less what happens in our pthread implementation of the "barrier pattern" (i.e. the one employing a parallel for). In that case we synchronized our workers so that they waited each other once they finished a single pass over their chunk, but now we want to avoid the overhead arising from that. If we suppose that threads act really independently many problems open up: first we need to deal with data races (that we previously neglected, since the odd-even structure coupled with barriers prevented any data race); second we need to provide some stopping condition (since the theorem mentioned in the introduction and stating that $n$ outer-loop iterations are enough holds only if the outer loop is sequential).
Now we explain how we managed to solve those problems, implementing a version of the algorithm in which each worker perform the OES iterations on its chunks until a global stopping condition is met.

## 3.1 Data Races

Consider the vector $v$ divided in chunks so that the $i$-th chunk is $v[st[i]] \dots v[en[i]]$ where $st$ and $en$ are suitable vectors of indices denoting chunk boundaries. Our algorithm swaps elements with their successors, hence the only elements of $v$ for which a data race occurs are $v[en[i]] = v[st[i+1]]$ for some $i = 1 \dots nw$. We can use a vector of mutexes (one for each of those values) without incurring in a substantial overhead, in fact each pass over a chunk will only need two locks and unlocks: one for the left side and one for the right side boundary. In the next subsection more structure will be introduced that will need some synchronization mechanism to avoid data races, we exploited

the fact that those structures are often accessed simultaneously to boundary elements of $v$ and employed the same mutexes. The only detail worth mention regards deadlocks, to prevent the deadlocks we adopted the "topological sort rule": given the natural numeration of mutexes associated to elements $v[st[i]]$, we only nested mutexes so that the inner one is associated with the lower number.

### 3.2 Stopping Condition

## 4 FastFlow Version

## 5 Experiments

## 6 Conclusions