Report

Alessio Russo

[1]Dipartimento di Informatica – Università di Pisa

a.russo65@studenti.unipi.it

1. General information

The project consists in designing a parallel application solving the Odd-Even sort problem. In particular, providing two applications, one using only the C++ threads (C++ STD and Pthreads only), one using FastFlow and measuring the performance achieved on given target machine.

2. Work organization

The project has been developed with following steps:

1. writing the sequential code
2. analyzing possibilities for parallel solutions
3. evaluating alternatives using abstract performance models
4. implementing most promising solutions
5. evaluating the perfomance on the target machine

3. Sequential Odd-even sort

The problem takes in input a vector and gives in output the sorted vector. This kind of problem runs in two consecutive phases:

1. odd phase: comparing all odd indexed elements with the adjacent element in the vector and, if this pair is in the wrong order the elements are swapped.
2. even phase: comparing all even indexed elements with the adjacent element in the vector and, if this pair is in the wrong order the elements are swapped.

This phases are repeated until no swaps occur.

3.1. Vectorization

In this algorithm both the two phases are basically two loops. Analyzing the non functional properties of the for loop, I noticed that the number of iterations are known, independent and not mixing data, and so after rewriting the swap instructions in order to remove the branches, using the flags: $-fopt-info-vec$ and $-fopt-info-missed$, we can see that the loop is vectorized. I used the $pragmaGCCivdep$ to tell the compiler to not use versioning stuff because there is no aliasing.

```
[a.russo65@C6320p-2 work]$ g++ -std=c++17 -O3 seqv.cpp -o seqv -fopt-info-vec
seqv.cpp:13:18: optimized: loop vectorized using 16 byte vectors
seqv.cpp:13:18: optimized: loop vectorized using 16 byte vectors
```

Figure 1. Checking vectorization

3.2. Evaluation

As expected, the perfomance of sequential code with vectorization is better. The following experiments show the effects of the vectorization with a vector of 1×10^5 integer elements and a seed.

```
[a.russo65@C6320p-2 work]$ g++ -std=c++17 -O2 seqv.cpp -o seqv -fopt-info-vec
[a.russo65@C6320p-2 work]$ ./seqv 100000 1234
Seq:  computed in 30770015 usec
```

Figure 2. Not vectorized

```
[a.russo65@C6320p-2 work]$ ./seqv 100000 1234
Seq:  computed in 15736157 usec
```

Figure 3. Vectorized

In the vectorized solution, the completion time is more or less half time. This is due to the fact that it is vectorized using 16 byte vector and in this case with a single instruction are loaded four integers that are enough to execute two iterations in one shot.

4. Parallel Odd-Even sort C++ pthread

This section describes the design and development of the parallel solution using C++ threads with all the critical points and decision faced.

4.1. Probing the sequential code

To figure out the behavior of the sequential code, I put into code some extra instructions in order to measure the average completion time and the average time spent to complete each phase of the algorithm.

Example 4.1 *Execution: ./seqv 100000 1234 (size seed)*
Sequential time : computed in 15,782956 usec with iteration: 49820
avg odd: 157.348 usec
avg even: 157.236 usec

The first thing that I decided is not to use the fork/join model. In particular, not forking threads per each phase because all the overhead introduced to setup the concurrent activities at every phase can be greater than the computation.

4.2. Design of parallel solution

Clearly all the comparison of the odd pairs and even pairs respectively can be done in parallel. But the workers have to finish phase 1 (odd) before starting the phase 2 (even) and then iterate again. So each worker, after receiving its partition in the creation phase, executes on its partition of the vector the steps in figure 4.

4.2.1. Data partitioning

The partitioning is done by the thread main when creating the workers. The workers access, using pointers, on a portion of the vector according to the phase without having conflicts in the sense that the worker never accessess to the same elements accessed by another worker on the same phase. The semantic of the odd-even sort, in terms of odd/even indexes, in the vector is respected. Each worker will get more or less a portion of $size/nw$ elements supporting the distribution of the remaining.

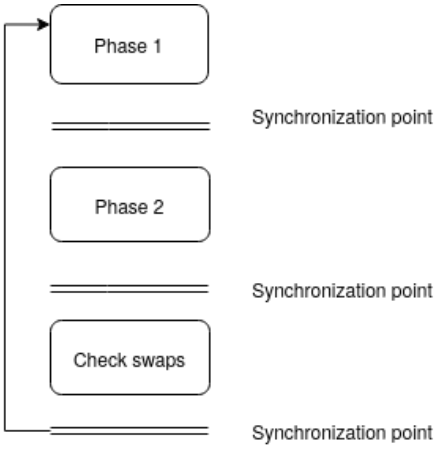


Figure 4. Abstract structures

It can be recognized as a sort of iterative data parallel pattern.
In particular, can be seen as a map/stencil data parallel pattern.

4.2.2. Synchronization point: barrier

When a computation is broken into phases, it is necessary to ensure that all threads complete all the work in one phase before any thread moves onto another phase, this is our case. A barrier is a form of synchronization that ensures this. Threads arriving at a barrier wait there until the last thread arrives, then all threads continue.

4.2.3. Exit condition

After the iteration i is finished before starting the iteration $i + 1$, the workers have to check if in the previous iteration no swaps in both phases occur. If no swaps occur then the workers stop, otherwise they iterate again. This phase can be implemented using an additional synchronization point and an atomic variables that keep track of the global swaps or avoiding this and using a vector of counters, in particular a counter per each worker.

4.3. Expected perfomance

Given a number of workers nw, each worker computes in parallel the phases on a portion of the original vector that is more or less $size/nw$. So the perfomance expected is $Tpar = Tseq/nw$ as T the completion time. Of course we have to consider the overhead introduced by the synchronization points and the communications. In particular, the time to safely update swaps and exit condition that can increase the serial fraction and so adding a potential Amdahl bottleneck.

4.4. Implementation

This section describes the actual implementation of the parallel C++ thread version.

4.4.1. Thread main

To setup the parallel activities, the thread main forks, only at the beginning, workers with their subproblem. Then the thread main waits until the workers finished their task with join. As mentioned before, the thread main divides the problem into subproblems according to the number of workers and assignes each subproblem to a worker.
A subproblem consists of:

1. (start, end): denotes the portion of the vector allowed to be modified by the worker.
2. even/odd mode: are added to the $start$ according to the phase in order to respect odd phase and even phase in terms of indexes swapped.
3. identification number
4. vector used to keep track of the swaps (size = number of worker)
5. vector of barriers to coordinate the threads

4.4.2. Worker

The workers execute the algorithm described above using essentially the same logic of the sequential version to execute the odd/even phase. Each phase returns the number of swaps executed and each worker accessess, using its own id, to a vector of counters accumulating the number of swaps for each worker. I decided to use a kind of $owner\ writes$ shared data structure (vector of counters) because it does not require the use of locks and to avoid the use of a global shared atomic variable that can reduce the parallelism according to the Amdhal law by increasing the serial fraction. So the critical point, described below, is how manage the synchronization and the termination of the workers minimizing as much as possible the overhead.

4.4.3. Barriers

A barrier, in my solution, is implemented using an atomic shared variable. First the atomic variable is initialized to the number of workers. Then when they arrive to that point, the threads try to decremet the shared variable initilizated and block if the value of that variable is not equal to 0. This is what the worker does after the first phase. Another barrier is needed after the second phase. The waiting on this barrier is managed in a different way. The $nw - 1$ decrement the shared variable and wait until is 0 instead the last thread doesn't decrement the variable but wait until the other threads finished the second phase and updated the private counter of swaps. So it waits until the shared variable is not 1. When it exits from the active waiting, only this thread will run and has the responsability to check the swaps, and decrement the shared variable, free the other threads and start a new iteration or eventually terminate. I decide to use a vector of barriers because implementing a $reset$ barrier, according to me, can require some extra atomic variables and some extra work and this can slow down the perfomance.

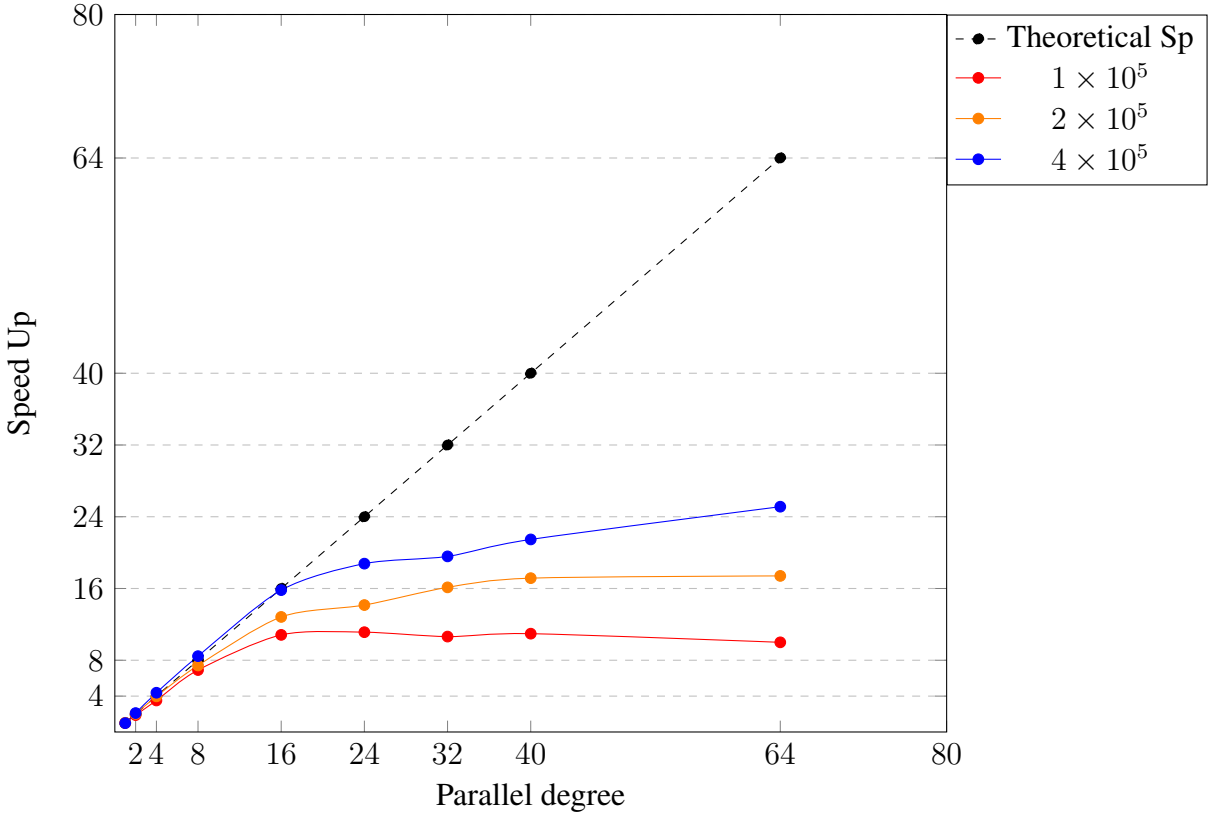
4.4.4. Termination

I decided to use a common kind of shared state access: owner writes. A number of concurrent entities access the variable for reading but only one of the concurrent entity is

allowed to modify it. In this case, the variable is shared among threads and so this means involving some kind of cache coherence algorithm on the target machine. The designated thread, after exit from the barrier, has the additional task of checking the vector of the swaps and if the total swaps is 0 then set the exit condition to true. Then it decrements the counter of the barrier. Before starting a new phase the threads read that state and eventually terminate. The vector of the swaps is shared among threads and this means that each time a thread made a change, the vector changes and all the other threads will have an old value. So I have some cache coherence protocol that moves the data but in this situation the other threads are not interested. Padding the vector is one possible solution to avoid the false sharing.

4.4.5. Evaluation

Once the implementation is finished, the evaluation phase started. The test phase is executed, as expected, in the shared remote machine used during the course. I made 3 experiments of the same problem with different sizes: 1×10^5, 2×10^5, 4×10^5. The perfomance indicator taken into account is the speed up. I decide to not provide the scalability plot since the $Tpar$ with $nw = 1$ is near $Tseq$.



Speedup gives a measure of how good is our parallelization with respect to the ”best” sequential computation. As we see in the plot, in all cases we reach some speed up. In particular, we almost reach a linear speed up with small parallel degree and a sublinear speed up with the increasing of parallel degree. The linear or optimal speed up cannot be achieved because the extra activities that are necessary for synchronization and communication and also, according to the Ahmdal’s law, the speed up is limited by

the serial fraction of the code. In our case, the computation of a single worker has two synchronization points that cause active waiting and the use of an atomic that is shared among threads. The use of shared atomic can cause a slight increase of the serial fraction. In addition, the last thread has an extra amount of work to manage the termination that cannot be parallelized and also means increasing the time spent into the barrier for the other threads. Also we have to take into account the communication, in particular the execution of some kind of cache coherence algorithm for the two shared data structure used (vector of swaps and exit condition), especially for the last thread that manages termination. According to the Gustafson's law increasing the size of the problem, the speedup increases. We can observe this behavior in the plot. Last, if we take into account the 4×10^5 size problem, with small parallel degree, we have a superlinear speed up.

5. Fast Flow Odd-Even sort

This section describes the design and development of the parallel solution using Fast Flow.

5.1. Design of parallel solution

Given the characteristics of the problem, a possible solution can be the use of a fast flow farm, in particular a master−worker.

- Emitter: the emitter has the task to synchronize the execution of the phases and manage the termination without using the explicit mechanism used in C++ thread version but using the implicit ones provided by Fast Flow. At the beginning, the emitter start the execution sending to the workers the phase to execute: odd or even. After this, it has the task to wait that each thread send a message to notify that it has finished the phase and the number of swaps done. The emitter accumulates the swaps. If all the threads finished the first phase, it sends a message that tells to starting the other phase. After completing both phases, the emitter has the task to check the swaps and eventually send a termination message (propagate EOS) for terminating the execution or starting again with the next iteration.
- Worker: in this version the structure of the worker is a little bit different. All the workers performs the same amount of work in a different portion of vector. So they have all the same logic. After finished to execute the current phase, it notifies to the emitter the number of swaps that occurs in that phase.

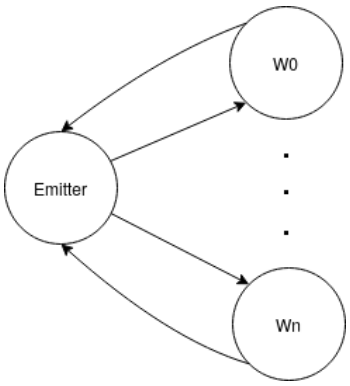


Figure 5. Master worker with feedback channel

5.2. Expected perfomance

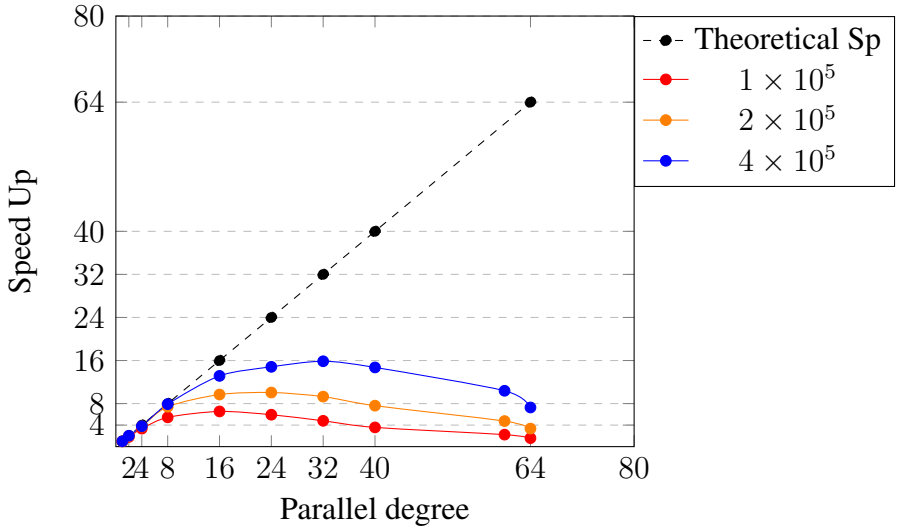
In order to model service time of the master/worker we have to observe that:

- emitter: is actually the result of merging synchronization and collection of swaps to manage termination.
- worker: complete a single phase on a portion of vector Tw.

So the service time of the farm is Ts (nw) = max (Tw, Te). Of course we have to take into account that the work of emitter increases with the number of workers to synchronize and so increase the communication cost.

5.3. Evaluation

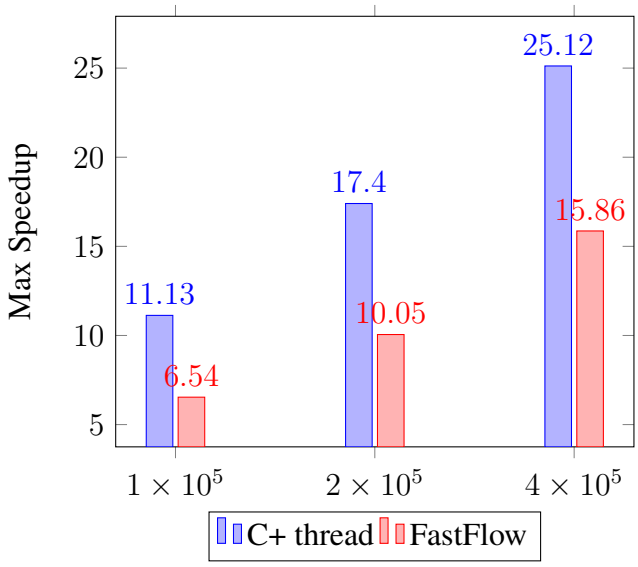
In this plot are reported the perfomance, in term of speedup, of the fastflow implementation. In the $x - axis$ is reported only the number of workers but we have also to consider the thread for the emitter.



With small parallel degree, we have almost a linear speed up but then it becomes sublinear with the increasing of the parallel degree. In this solution, we cannot achieve the optimal speed up because of the time spent to synchronize, coordinate the workers and manage the termination. All of this extra activities are pure overhead and all of them are done by the emitter (very frequently) per each phase computed by the workers. In fact the worker, after completing one phase has to communicate with the emitter and has to wait a message from the emitter before starting a new one. So the amount of work of the emitter slow down the perfomance.

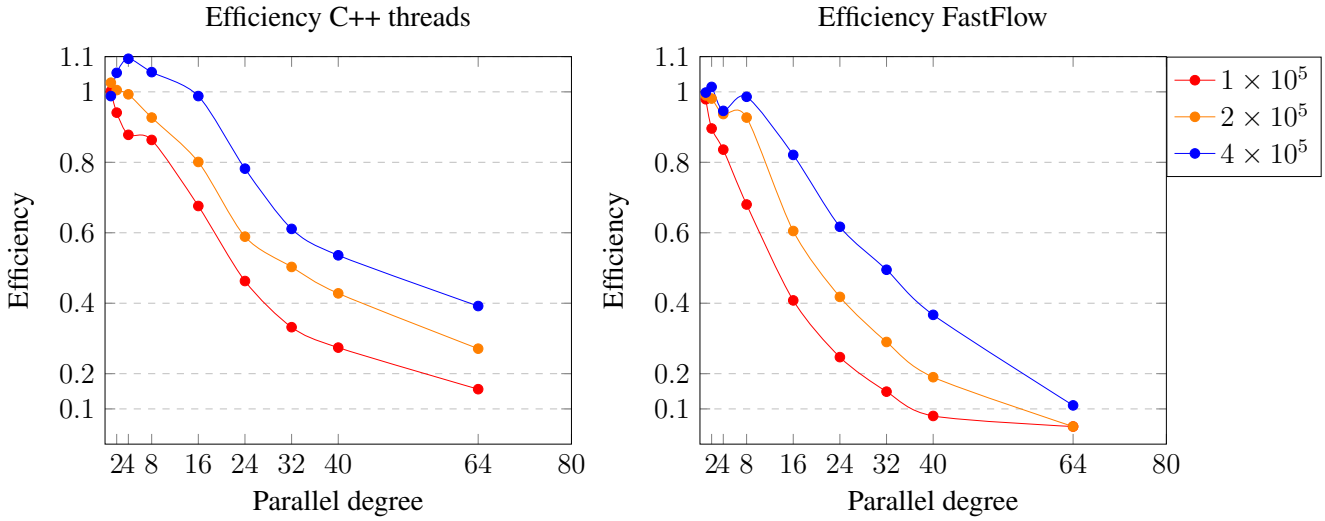
6. Comparison parallel solution

The plot below show the comparison between the maximum speed up reached by the two parallel solutions.



6.1. Efficiency

This section shows another kind of perfomance indicator: efficiency. The efficiency shows the ability of the parallel implementation in making a good usage of the number of processors.



7. Alternative solutions

The project has been developed using the knowledge acquired during the courses and during the assignments. Before implementing this final version, I implemented other kind of solutions using in a slightly different way the barriers or termination. For example, using a barrier with reset instead of a vector of barrier, or adding a third barrier and using a global atomic variable to manage the swaps and termination without adding extra work to a single thread. Thi solutions involves more communication among threads and the use of more atomics with respect to the final solution proposed above.