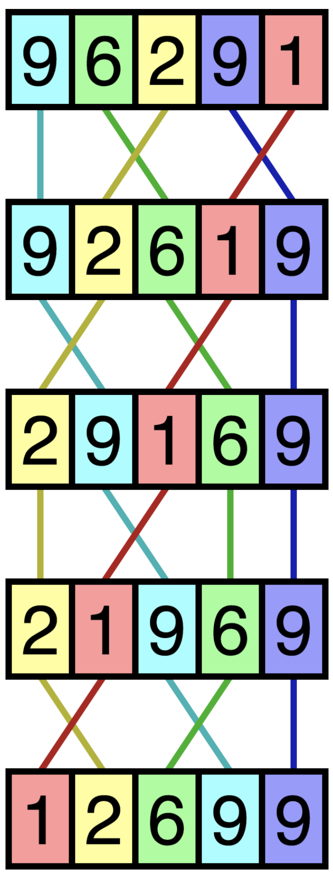
Parallel Odd-Even Sort

Andrea Bruno

SPM 2019/2020



1 Introduction

In this report, we analyze the odd-even sort algorithm and the possible ways to implement it in a parallel form. The structure of this document is the following:

- A brief introduction to the algorithm.
- The parallel architectures implemented.
- The description of the experiments.
- The analysis of the results.
- Final observations and conclusions.

2 Odd-Even Sort

The odd-even sort compares adjacent pairs of items in a list and exchanges them if they are out of order. The algorithm starts by pointing first at even indexes and in the subsequent phase it points to odd indexes. The procedure terminates when no swaps occurred. Here the pseudocode of the algorithm:

Algorithm 1 Odd-Even Sort Pseudocode

```
procedure OddEvenSort($A[\,]$)
    $n \leftarrow len(A)$
    $sorted = false$
    while $!sorted$ do
        $sorted = true$
        for $i \leftarrow \mathbf{0}$ to $n-2$ , $i \leftarrow i+2$ do            ▷ Even Phase
            if $(A[i] > A[i+1])$ then
                swap $A[i]$ and $A[i+1]$
                $sorted = false$
            end if
        end for
        for $i \leftarrow \mathbf{1}$ to $n-2$ , $i \leftarrow i+2$ do            ▷ Odd Phase
            if $(A[i] > A[i+1])$ then
                swap $A[i]$ and $A[i+1]$
                $sorted = false$
            end if
        end for
    end while
end procedure
```

As one may notice, the two inner loops share the same structure, apart from the starting index. This aspect has been captured through the implementation of an abstract version of the loops. The following code explains the modus operandi.

Algorithm 2 Abstract Loop

```
...
for $i \leftarrow$ start to end , $i \leftarrow i+2$ do
    if $(A[i] > A[i+1])$ then
        swap $A[i]$ and $A[i+1]$
        $sorted = false$
    end if
end for
...
```

This new version creates opportunities to exploit machines parallelism. Indeed, we can split the array in chunks and let each worker compute part of the solution. Further details will be discussed in the following paragraphs.

3 Parellel Architectures

The nature of the problem does not bring with it a clear idea on which are the right design choices for parallelism. The main problem resides on the outer while-loop. At each iteration, we must check whether a swap occurred in one of the two inner loops. In the concurrent universe, this implies the inclusion of some synchronization mechanism. If we look at the code without considering the abstract loop version, a naive design could be:

$$Pipe(Farm(Odd), Farm(Even)) \equiv Farm(Pipe(Even, Odd))$$

Alternatively, by exploiting the loop abstraction described before, we can lighten the architecture by using a simple $Farm(AbstractLoop)$.
In both cases, there must be a feedback channel between the ending and the starting entity to ensure the correctness of the algorithm, or in other words, to check if a swap occurred. In this work, we implemented two versions, the former using a *Farm-with-Feedback* template, and the latter following the *Master-Worker* template.

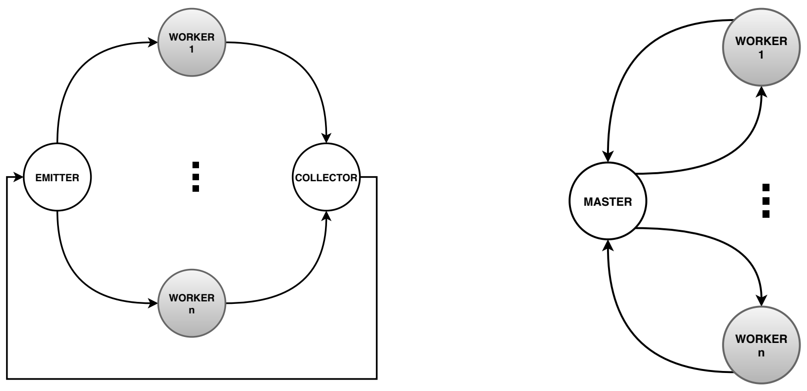


Figure 1: Farm-with-Feedback (left) and Master-Worker (right)

3.1 Farm-with-Feedback

The *Farm-with-Feedback* version implements all the classes in the header IFarm[1]. There is:

- an Emitter producing units of computations (called OUT).
- some Worker applying a function over the given chunk and returning a result.
- a Collector which merges (reduce) the partial results and produces a final value (feedback).

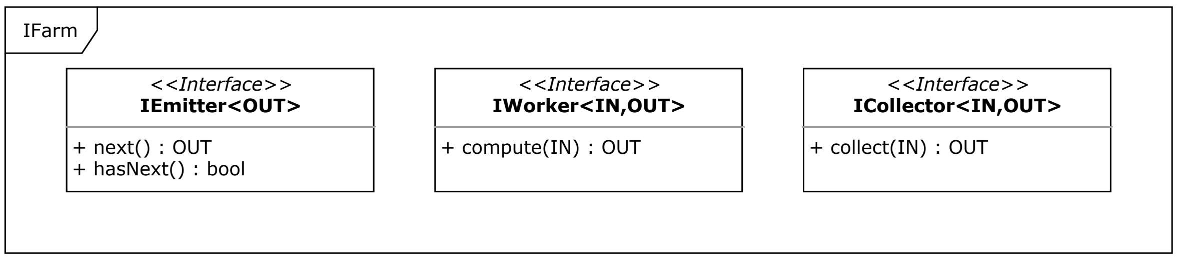


Figure 2: IFarm header

To make an exhaustive analysis, we developed two versions of this template to cover as much as possible, all the potential forms of cutting of the initial array. There is one using *Tasks* and one using *Chunks*. The details of the implementations are explained in the following two paragraphs.

3.1.1 Tasks[2]

Each task is composed of two *int*s and one *bool*. The two integers within each task, point respectively towards the start and the end of the abstract loop, whereas the flag merely keeps track of the eventual exchanges. The sampling[3] of those indexes is in charge of the Emitter, who in turn, will equally distribute the parts among the workers. Then, each Worker will check the sequence within the given limits and possibly will update the flag. Finally, the Collector will perform a logic-OR with all the flags received and if necessary will restart the Emitter.

```
struct Task
{
    int begin;
    int end;
    bool swap;

    Task(int a = 0, int b = 0, bool s = false) : begin(a), end(b) {}

    inline bool operator==(const Task &rhs)
    {
        return (begin == rhs.begin) && (end == rhs.end);
    }
};
```

Figure 3: Task struct

[1] core/IFarm.hpp [2] core/FarmT.cpp [3] lib/tools.cpp

3.1.2 Chunks[4]

The Chunk data structure contains two *ints*, a *bool* and a *pointer* to a vector. Both the two integers and the boolean are semantically the same to the Task idea. The vector instead, is generated on the fly by the Emitter by copying the values from the original array in the range indicated by the two integers. Hence, the logic of the Worker is quite the same. The one who needs some correction is the Collector. Indeed, it must write back the results into the original array in case a swap occurred.

```
struct Chunk
{
    std::vector<int> *list;
    int begin;
    int end;
    bool swap;

    Chunk(std::vector<int> *list = nullptr, int a = 0, int b = 0, bool s = false)
        : list(list), begin(a), end(b) {}

    inline bool operator==(const Chunk &rhs)
    {
        return (begin == rhs.begin) && (end == rhs.end) && (swap == rhs.swap);
    }
};
```

Figure 4: Chunk struct

3.1.3 Communication channels

To be sure of getting the best performances without incurring into undetectable bugs, we chose to reuse the Buffer[5] library presented during the SPM course. Although its logic is very easy (few lines of code), its results are very satisfying. Therefore, even if there is the possibility of using a passive queue (that is also greener), for the sake of getting the best results as possible, we discarded that approach.

3.2 Master-Worker[6]

The *Master-Worker* version implements all the classes in the header IMaWo[7]. There is:

- a Master acting as an Emitter/Collector.
- some Worker applying a function over the given chunk and returning a result.

The logic is roughly equal to the Farm-with-Feedback version, indeed it makes use of Tasks and Buffers. In this case, the Master besides generating and providing tasks, it must collect feedback from the Workers to decide whether to continue sorting.

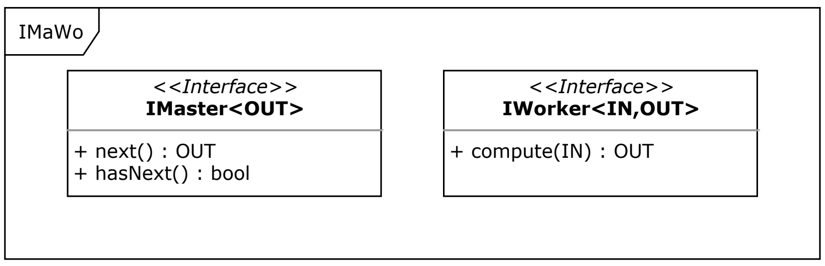


Figure 5: IMaWo header

3.3 FastFlow

In FastFlow, we implemented the very same architectures described above by introducing just a couple lines of code. Hence, there are a Farm-with-Feedback[8] version and a Master-Worker[9] one. The communication mechanisms are, of course, the one underlying FastFlow, whereas the unit of computation is the Task.

3.4 Extra: OpenMP

Even though it was not required, we built an OpenMP[10] version of the code.

[4] core/FarmC.cpp [5] lib/buffer.cpp [6] core/MaWo.cpp [7] core/IMaWo.hpp [8] ff_farm_tasks.cpp
[9] ff_master_workers.cpp [10] openmp.cpp

4 Experiments

To analyze and understand the advantages and disadvantages of the proposals, we did many stress tests on each architecture. In detail, for each of them, we varied the number of workers from 1 to 50 by fixing to 100K the items in the array. To guarantee fairness, both the seed and the range of the random, are identical in each experiment. Furthermore, before starting the actual tests, we assessed the algorithmic correctness by using the standard library function *is_sorted()*.

To start the experiments, please compile all the files using the command: *make all*, then launch the script *experiments.sh*.

4.1 Tasks vs Chunks

We tested deeper our best native C++ architecture (Farm-with-Feedback, see Section 5) to observe the differences between the adoption of Tasks and Chunks. In particular, we measured:

- the execution time of each entity.
- the communication overhead.

We provide in the folder *time*, both the *Makefile* and two scripts to replicate the experiments.

5 Results

In this section, we explain the data obtained by examining the issues encountered during the experiments.

5.1 Time

The plot in Figure 6 shows the time trend as the number of workers increases. Up to 20 workers the curve has a negative slope in all the architectures. In the range [20,30] there is a general flattening, the only version whose time is still decreasing is the OpenMP one. Interestingly enough, the two FastFlow versions after 30 workers, reach a mediocre time performance.

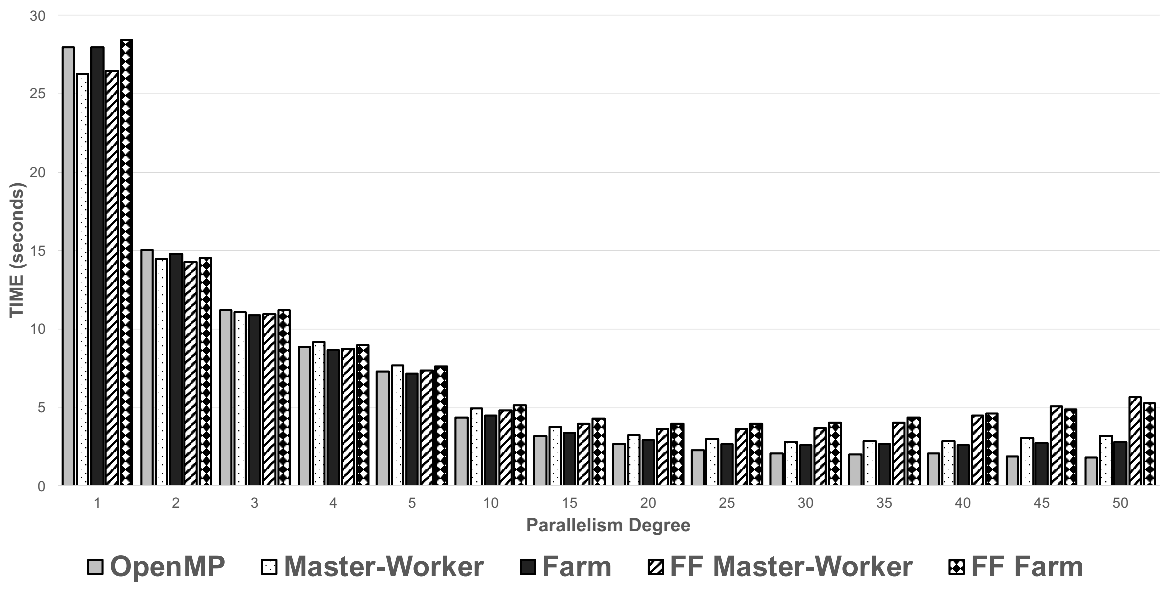


Figure 6: Time plot

5.2 Speedup, Scalability and Efficiency

The hypothesis of a flattening after 20 workers, is confirmed by the plot in Figure 7. The speedup curve grows in all the variants up to 20. Next, the FastFlow versions obtain a negative slope, whereas the two native versions grow up to 35 and then they flatten. Only the OpenMP version continues growing, and although it can seem to be a positive result, the ideal curve is still far away.

The scalability (Figure 8) and the efficiency (Figure 9) plots are quite the same, hence the same considerations are legitimate.

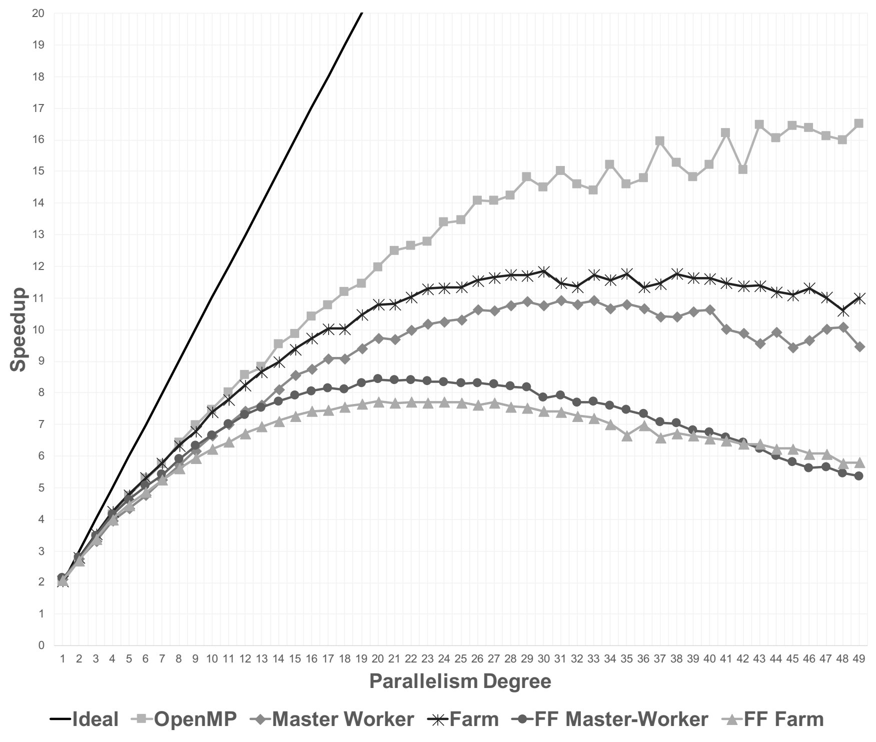


Figure 7: Speedup plot

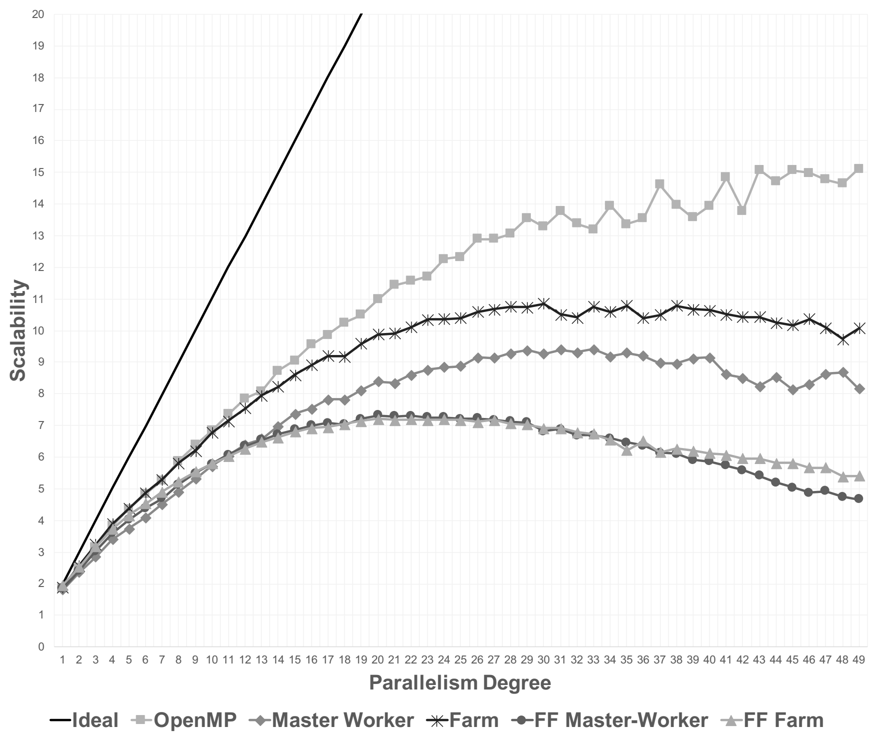


Figure 8: Scalability plot

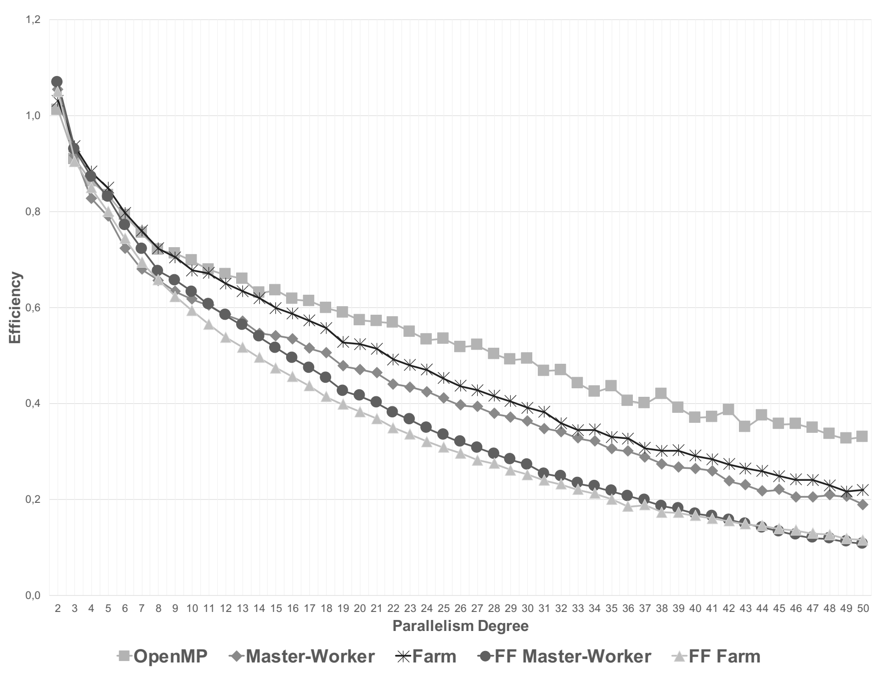


Figure 9: Efficiency plot

5.3 Tasks vs Chunks

As previously described in Section 3.1.1 and Section 3.1.2, we implemented two units of computations and we tested them (Section 4.1). In Figure 10, Figure 11 and Figure 12 is shown the average execution time of each entity. Despite the well-known problem of false sharing, the version using a global array (Tasks) performs better. The difficulty resides in the creation and the collection of these partial results, who finally, require to be reassembled.

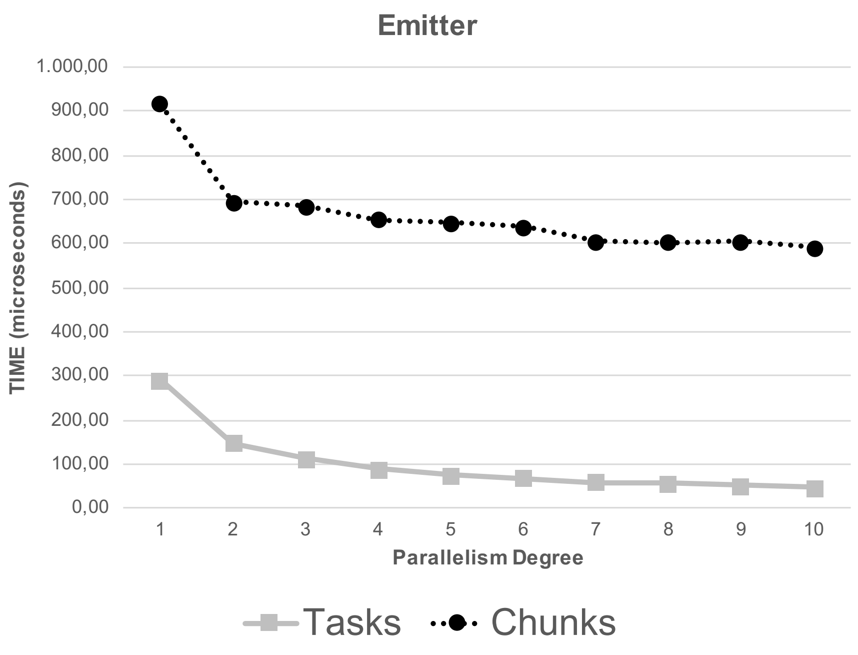


Figure 10: Emitter

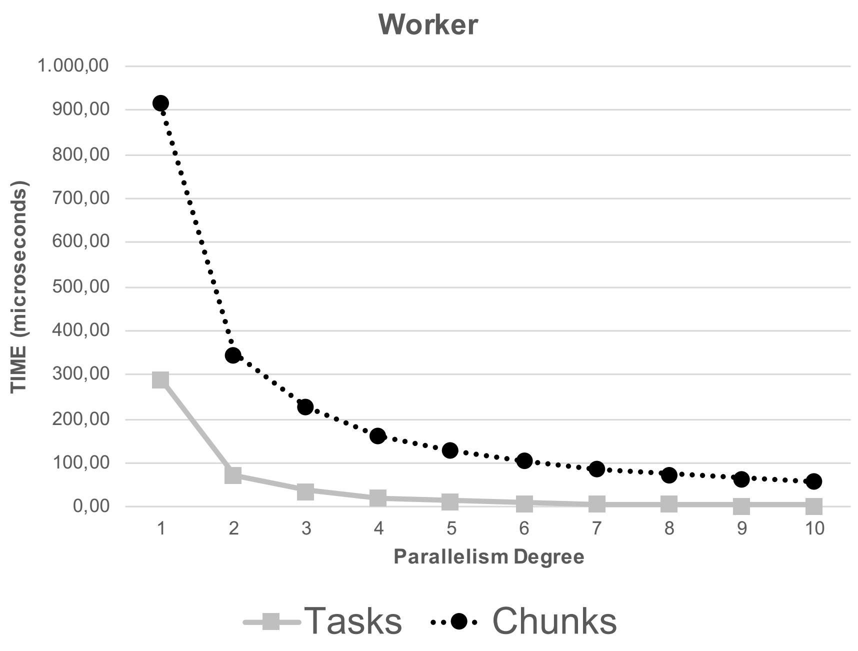


Figure 11: Worker

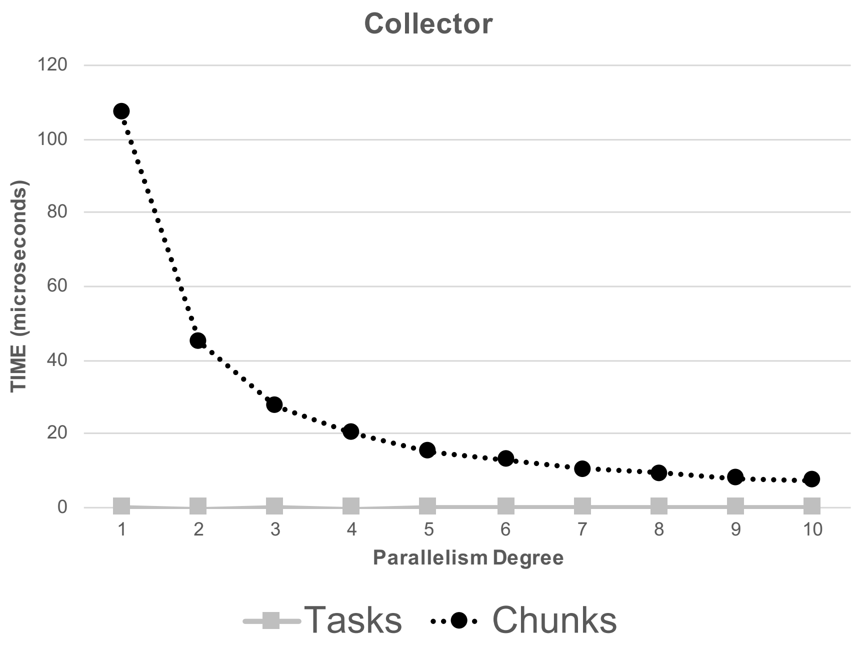


Figure 12: Collector

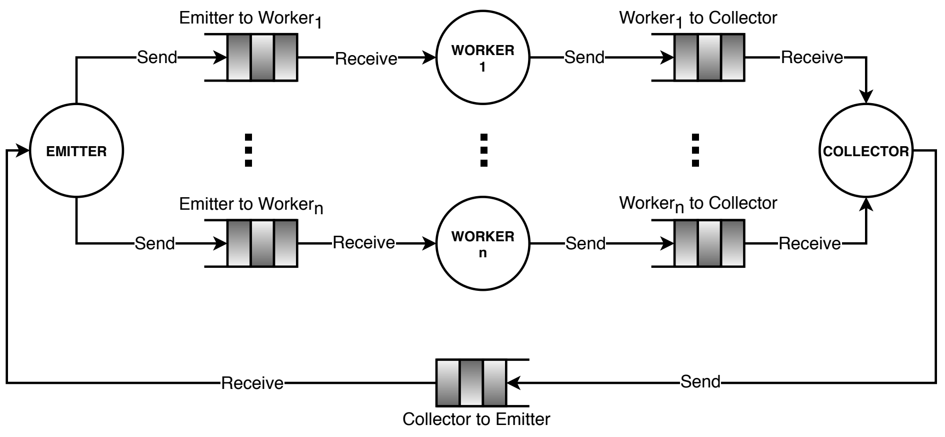


Figure 13: Schema of the communication channels

Even communication is lighter in case of Tasks. Indeed the overhead introduced by the copy and the recomposition of the list, makes the version using the vector pointer (Chunk) slower. This phenomenon can be appreciated by looking at Figure 14 and Figure 15 [11]. Hence, most of the time, the Emitter is waiting for something from the Collector, but also the Workers are in idle because the Emitter spends a lot of time in creating copies of the array.

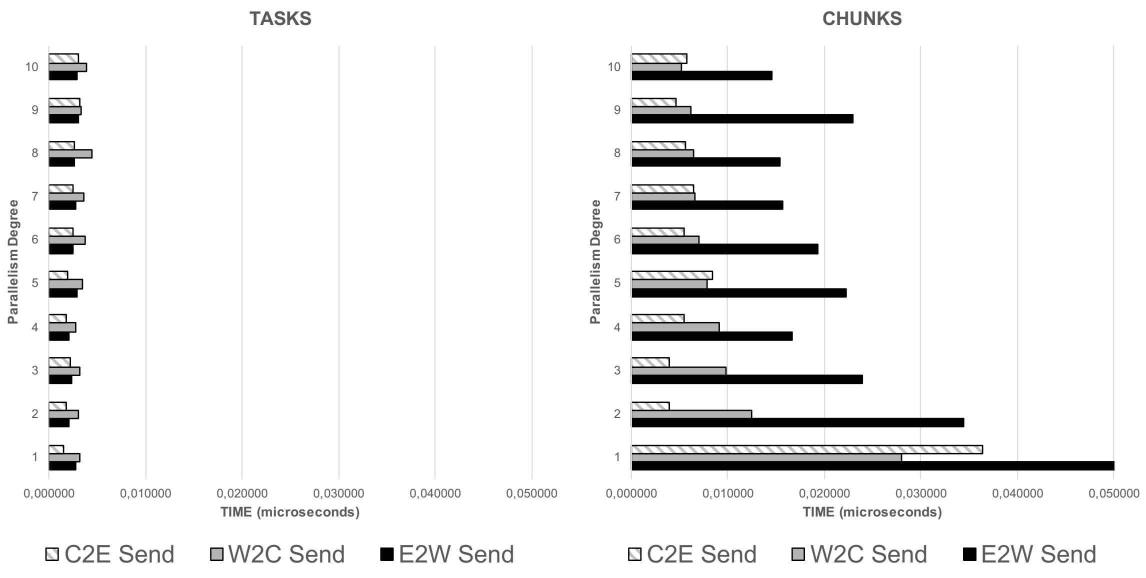


Figure 14: Tasks vs Chunks (Send)

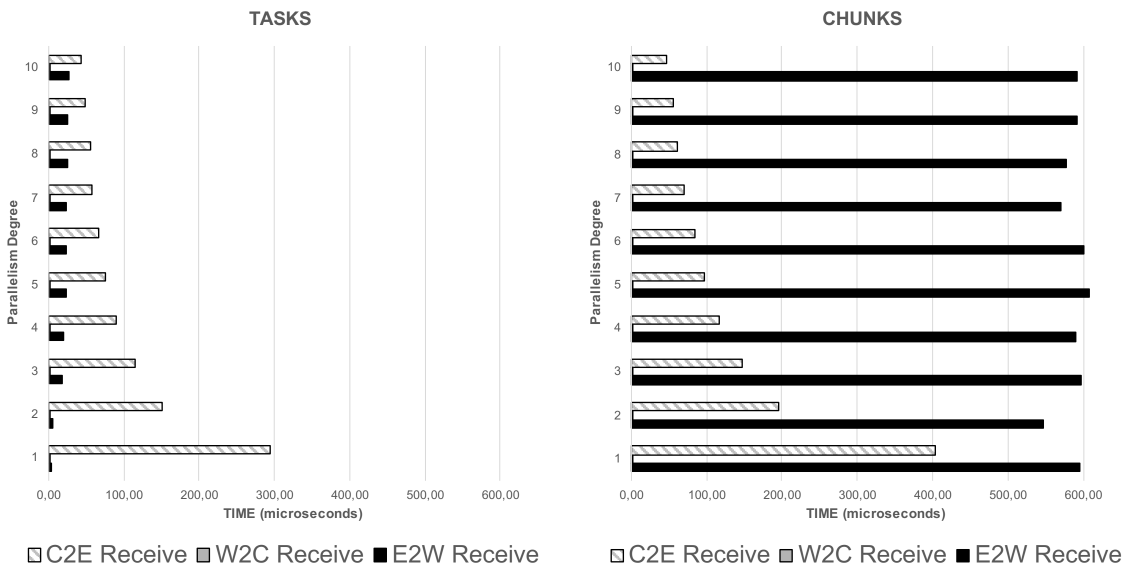


Figure 15: Tasks vs Chunks (Receive)

[11] E2W means Emitter to Worker. W2C means Worker to Collector. C2E means Collector to Emitter

6 Conclusions

In general, the speedups obtained are quite acceptable. The OpenMP version is the one that scales best between all the implementations developed. The main reason is that OpenMP can guarantee a smaller serial fraction than other architectures. This is thanks to communication mechanisms that are certainly more efficient, as well as better cache management. Moreover, being able to perform a static analysis on the code, it can generate the graph of the architecture, managing to optimize the code.

As expected from the Amdhal Law, the serial fraction, imposes great limits in the parallelization of the code. Indeed our best architecture gets better results when the Emitter and the Collector are limited to perform few operations (see Figure 10, 11, 12).

As far as the FastFlow versions are concerned, it is evident that the speedups obtained are not so satisfactory. The reason could lie in the greater complexity of the underlying mechanisms, in fact, the code used is almost identical to the version with native threads.