Università di Pisa, Master Degree in Computer Science
Course of "Parallel and Distributed Systems: Paradigms and Models" (SPM)

Yet Another Divide and Conquer Framework

Report of the Final Project for the A.Y. 2018/2019

Francesco Landolfi
fran.landolfi@gmail.com

April 2, 2019

1 Introduction

This report describes the framework developed to execute a Divide and Conquer (DAC) algorithm over a given input, possibly exploiting the parallel resources, where a DAC algorithm is identified by

- a *divide* function, which takes an problem and divides it into two or more (possibly easier) sub-problems;
- a *test* function, which, given a (sub-)problem, determines whether it needs to be divided further;
- a *conquer* function that for a given (sub-)problem returns its solution;
- a *combine* function that combines the results of two or more sub-problems.

The framework was implemented in C++14, and the above functions must adopt the same interface described in [1], which is the same provided by OpenMP, Intel TBB and FastFlow.

This report is organized as follows: section 2 will describe the structure of the model architecture, section 3 will briefly analyze the expected performances of the proposed model, section 4 will provide some details regarding its implementation and section 5 will show the results obtained adopting this model to solve the mergesort and quicksort algorithms, and comparing them with the ones obtained using other frameworks. Finally, section 6 adds a concluding remark.

2 Parallel Architecture Design

The core of the framework is the *scheduler module* (subsection 2.1), which is used to implement the *fork-join pattern* [2]. Two schedulers are used to enqueue the *fork* and *join* tasks, respectively, where

- a *fork* task has as input a problem and a *promise* of its solution. It consists in executing the *test* function over the given problem and, if it can be split, executes the *divide* function, otherwise the *conquer* one. Then
  - if the *divide* function was executed, assuming that the input was split in b sub-problems, enqueues in the join scheduler a join task having as input b *promises* of the solutions of the respective sub-problems and the same promise of the current fork task. Then, in the fork scheduler, enqueues $b-1$ fork tasks having each one a sub-problem and a *promise* (one of the bs given to the join task) as input. The remaining sub-problem will be used as continuation for the current fork task;
  - else, if the *conquer* function was executed, set the promise value with its result.

- a *join* task has as input b promises of the results of b sub-problems and another promise. It executes the *combine* function over the values contained in the *future* of each one of the b promises (once they will be ready), then sets the other promise value with its result.

To solve a DAC problem, it is sufficient to schedule a fork task having as input the original problem and a promise which will eventually be updated with the final result. A thread (or *worker*) should retrieve and execute tasks from the fork scheduler until it is empty, and only then start retrieving and executing tasks from the join scheduler. Once also the join schduler becomes empty, the original problem is solved.

Multiple threads can access these schedulers to retrieve a tasks, so it is crucial that the schedulers implement some kind of synchronization mechanisms to solve conflicts between concurrent accesses. Subsection 2.1 explains how this problem is tackled in the scheduler module, while subsection 2.2 shows how the two schedulers are combined together to form the final framework model.

2.1 The Scheduler

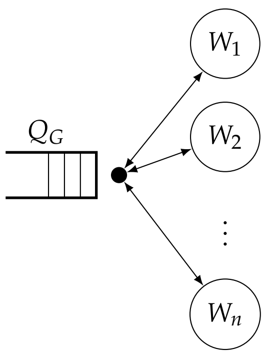
During the design of the scheduler module, there were taken into account the following three models:

1. a *base model* (Figure 1a), where the task were queued in a common (*global*) list. To retrieve a task, each worker had to access the global list in mutual exclusion (i.e., using a *mutex*). Although simple, due to this serialization of the accesses, this model can not scale up as the number of workers increases.

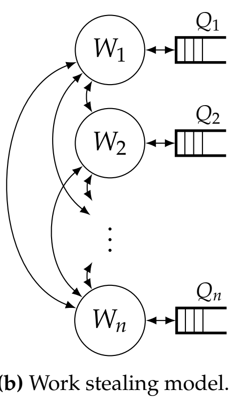
2. to overcome the previous problem, it was also considered a *work-stealing model* [2] (Figure 1b), where each worker has a *local* (but still synchronized) task queue. If a worker has no task in its local list, it "steals" a task from another worker's queue. To reduce the concurrent accesses, each worker extracts from its local queue the *top* task (i.e., the last one inserted), while, when in needs, it steals a job from the *bottom* of another random worker's queue (i.e., the first enqueued job), which, in the fork case, is expected to be more time consuming and to generate more sub-forks to be scheduled locally[1]. This model achieves a good load-balancing between threads, so that it is the model used to implement the map operation both in Intel Cilk Plus and Intel TBB [2], but has the drawback that, when the remaining tasks are few and they are being computed by some workers, the other ones stuck themselves in an active loop in which they select random workers, try to steal a task and fail.

3. the *proposed model* (Figure 1c) tries to find a compromise between the previous two models. Each worker owns a local queue, as in the work-stealing model, but, instead of "stealing" a task from another thread when the local queue is empty, it *actively* "passes the buck" to the

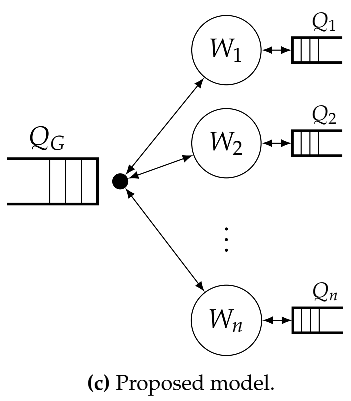
[1]This does not work in the join case, since stealing a job form the bottom of the queue will lock the thread and possibly produce a deadlock.



(a) Base model.



(b) Work stealing model.



(c) Proposed model.

Figure 1: Different models adopted to solve the Divide and Conquer problem using parallel threads.

other threads when the size of the local queue is too large, according to a heuristic. In this case, the worker extracts the bottom of its queue and inserts it in a global (*synchronized*) queue, as in the base model. When a worker has no task left in its local queue, it search for it in the global one. If also the global queue is empty, the worker halts until another worker inserts a task or "wakes up" all the other waiting workers after the last task was done. Differently from the work-stealing model, the queue of each worker is purely local and not synchronized. This model may achieve a good load-balancing, depending on the goodness of the heuristic function. This will be explained further below and in section 3.

The heuristic function To measure whether the local queue has too many tasks with respect to the remaining ones, the owner of the queue uses Pearson's *chi-squared test* (χ^2). More concretely, every time a task is scheduled in its local queue, a worker of index w computes the remaining task to be done, which are

$$R = |Q_G| + \sum_{i=1}^{n} |Q_i| \, ,$$

where Q_G is the global queue and Q_i the local queue of the worker of index i. Then, if the observed number of local tasks $O_w = |Q_w|$ are more then the expected $E_w = \frac{1}{n}R$, it computes the local χ^2_w value as follows

$$\begin{aligned}\chi^2_w &= \frac{(O_w - E_w)^2}{E_w} + \frac{(O_{\overline{w}} - E_{\overline{w}})^2}{E_{\overline{w}}} \\ &= \frac{n\big(|Q_w| - \frac{1}{n}R\big)^2}{R} + \frac{n\big(R - |Q_w| - \frac{n-1}{n}R\big)^2}{(n-1)R} \\ &= \frac{n\big(|Q_w| - \frac{1}{n}R\big)^2}{R} + \frac{n\big(\frac{1}{n}R - |Q_w|\big)^2}{(n-1)R} \\ &= \frac{n^2\big(|Q_w| - \frac{1}{n}R\big)^2}{(n-1)R} \, .\end{aligned}$$

If $\chi^2_w \geq \chi^2_{\max}$, where $\chi^2_{\max}$ is a user-defined value, the worker extracts a task from the bottom of the local queue and re-schedules it in the global one.

Load-balancing As will be discussed later in section 3, the value of $\chi^2_{\max}$ regulates the load-balancing of the scheduler. The user may set its value by choosing one of the following *balancing policies*:

- **`only_global`**: this option sets $\chi^2_{\max} = -1$. Since χ^2_w is always positive, every scheduled task will end up in the global queue. This option makes the scheduler act as the base model.
- **`strong`**: this option sets $\chi^2_{\max} = 0.455$. A worker will accept a scheduled task if the current distribution is observable with probability[2] higher than $p = 0.5$.
- **`strict`**: this option sets $\chi^2_{\max} = 3.841$. A worker will accept a scheduled task if the current distribution is observable with probability higher than $p = 0.05$.
- **`relaxed`**: this option sets $\chi^2_{\max} = 7.879$. A worker will accept a scheduled task if the current distribution is observable with probability higher than $p = 0.005$.
- **`only_local`**: this option sets $\chi^2_{\max} = +\infty$. Every scheduled task will remain in the local queue. A task scheduled to a given worker will never be reassigned.
- **`best`**: this option sets $\chi^2_{\max}$ with a value that depends on the number of parallel processors. In section 3 will be explained how this value is computed.

[2]This and every following p-value is evaluated using the chi-squared distribution with one degree of freedom, since we are considering the event that a task may or may not be assigned to a given worker.

Computing the number of remaining tasks Since the value R is computed every time a task is being scheduled, it needs a (possibly synchronized) method to be computed fast and without creating a bottleneck in the computation. To achieve this, the value of R is stored as an *atomic* and updated every time a task is scheduled or completed. Its content does not need to be perfectly coherent with the current status of the scheduler, since the value of χ^2_w can be just a rough approximation (e.g., a worker may compute χ^2_w with an old version of R). The value of R needs instead to be up to date when it comes to check whether all the remaining tasks have been executed. To do this, we need to follow the following rules:

1. The value of R must be (atomically) increased *only before* the scheduling of a task, and
2. The value of R must be (atomically) decreased *only after* having executing a task.

Following the previous rules, it can be proved that, if $R = 0$, there is no task to be computed.

2.2 The Divide and Conquer Model

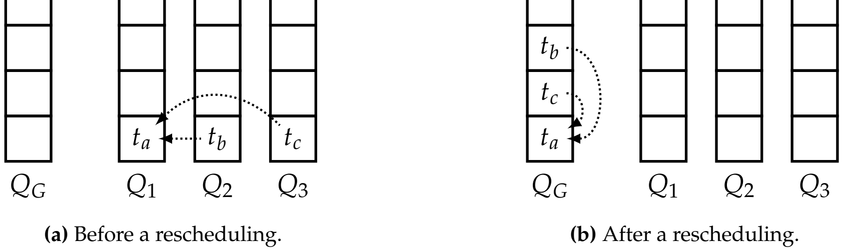
The final model of the framework is built using two instances of the scheduler described above: one used for the fork tasks, and one used for the join tasks. Since the join tasks are *data-dependent* with respect to their input, they need to be extracted following in reversed order of insertion.

The problem of task shuffling Using the heuristic load-balancing described above in the scheduler dedicated to the join tasks may lead to a deadlock, as depicted in Figure 2. In the example, Q_1, Q_2, and Q_3 have, respectively, t_a, t_b, and t_c, at their bottoms, with t_a dependent on t_b and t_c, as in Figure 2a. At some point in the execution, Q_1 may get too much tasks and move t_a to the global queue. Later, also Q_2 and Q_3 may move t_b and t_c in the global queue, as in Figure 2b. This may happen for a number of task greater than the number of workers, causing a deadlock.

Final model To solve this problem, the policy of the scheduler dedicated to the join task must be **`only_local`**. This will cause each worker to execute the join tasks in reversed order with respect to the forks the same worker have executed. The problem of this solution is that, the more the task were unbalanced in the fork phase, the more they will be in the join one. The final model can be synthesized as in Figure 3.

Other possible solutions To solve the problem of shuffling, using a `std::priority_queue` as global queue and using as priority the depth of the recursion of the fork seemed not to solve completely the problem: testing the framework on unbalanced DAC algorithms as quicksort eventually led to deadlocks.

Another possible solution that has yet to be tested is to optionally give higher priority to the global queue instead of the local ones, so that a worker, to retrieve a job in its local queue, has first to check whether the global queue is empty, making the model's behavior resemble the one in the work stealing model.



(a) Before a rescheduling. **(b)** After a rescheduling.

Figure 2: A example of task shuffling.

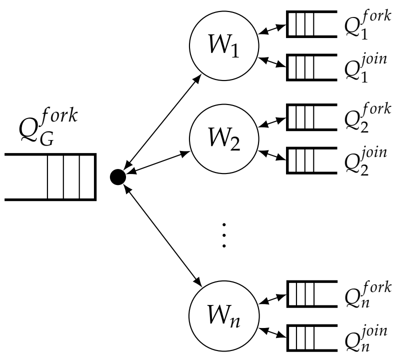


Figure 3: Divide and Conquer (DAC) model.

3 Performance Modeling

A time complexity analysis of the DAC parallel pattern over an ideal machine (with an infinite number of parallel processors) can be found in [2]. This analysis may also fit our model if we prove that a worker, as it computes a fork, schedules its sub-forks in the global queue instead of the local one. This is true if we set $\chi^2_{\max}$ such that, even assuming $n-1$ tasks have been evenly distributed among all the n workers, a busy worker that is about to schedule another task will give it to the global queue instead. More concretely, given w the index of the worker which is about to schedule its second task in its local queue, we have that

$$\begin{aligned}\chi^2_w &= \frac{(O_w - E_w)^2}{E_w} + \frac{(O_{\overline{w}} - E_{\overline{w}})^2}{E_{\overline{w}}} \\ &= \frac{(2-1)^2}{1} + \frac{(n-2-(n-1))^2}{n-1} \\ &= \frac{n}{n-1} \, .\end{aligned}$$

Since as $n \to \infty$, $\chi^2_w \to 1$, if we set $\chi^2_{\max} \leq 1$, all workers will receive a task at the beginning of the fork phase, whatever it is the number of processors.

Finite number of parallel processors If we relax the assumption of having an infinite number of parallel processors, we may still refer back to the general case just noticing that the fork phase will schedule the new sub-problems on the remaining free processors until every worker become busy, as shown in Figure 4. Assuming the size of the sub-problems are the same for every parallel executor, and that the new tasks will be spawned at regular intervals, the DAC computation will act as a *serial DAC algorithm* with respect to a single worker, since every new task generated by the fork phase will be scheduled in its local queue. As the local sub-problems are completed, a “reduced” join phase concludes the global computation.

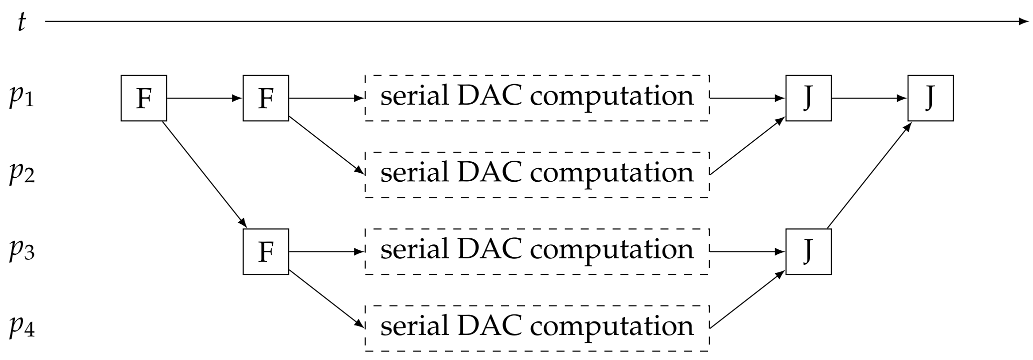


Figure 4: Example of how the tasks may be distributed among different processors p_i.

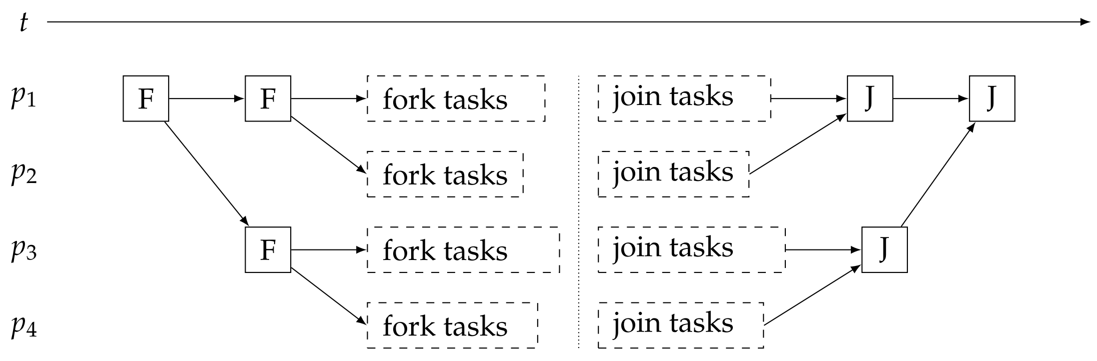


Figure 5: Example of how the tasks may be distributed unevenly among different processors p_i.

Unbalanced sub-problems If we relax the assumption of having uniform partitions of the problem among the parallel executors, we may expect to observe an unbalanced number of tasks executed among the workers. Moreover, since the join tasks will not be executed until all fork tasks have been completed, some workers may remain idle as the number of remaining fork tasks reaches zero, as depicted in Figure 5.

This problem is mitigated by the "buck-passing" of the workers done whenever the local queue becomes relatively unbalanced. Lowering the value of $\chi^2_{\max}$ will increase the load-balancing among the parallel workers but may also have an impact on the performances, in opposite ways. More concretely,

- lower values of $\chi^2_{\max}$ will lead to a better load-balancing among parallel workers, at the cost of having more tasks scheduled in the global (synchronized) queue, hindering the scalability of the overall process;
- higher values of $\chi^2_{\max}$ will lead to a worse load-balancing among parallel workers, but also making less tasks be scheduled in the global queue, allowing the process to scale up as the number of processors increases.

Setting $\chi^2_{\max} = \frac{n}{n-1}$, where n is the number of parallel processors, will ensure that the first (and, ideally, more time consuming) tasks will be scheduled in the global queue and then distributed evenly among the workers, thus encouraging a better load-balancing, at least in the first part of the fork phase. This is the value given by the **`best`** balancing policy option, which is set by default.

Notice that this value decreases with the number of parallel processor, which is counter-intuitive. One may choose to set a higher (fixed) value of $\chi^2_{\max}$, giving up the initial even distribution of the tasks in order to achieve a better scalability.

4 Implementation Details

The framework is implemented in C++14, and it is composed by:

- a **`DAC`** class, which implements the solver for a given divide and conquer algorithm, as described in subsection 2.2. This class has the same interface of the ones provided by Intel TBB and OpenMP, except that the input, the output and the number of parallel workers have been moved from the constructor to the **`DAC::compute`** method;
- a **`Scheduler`** class, which implements the scheduler module described in subsection 2.1. This class also contains the following two inner (private) classes:
  - a **`SyncJobList`** class, which implements a synchronized verison of `std::list` and contains the number of remaining tasks to be computed;
  - a **`Worker`** class, which manages the local queues of each parallel worker i.e., the scheduling and retrieval of the tasks.

5 Experiments

The framework was tested on the test cases provided by authors of [1]. In particular, the framework was used to compute the mergesort and the quicksort algorithms over arrays randomly filled with 10M, 20M, 50M and 100M elements, with a base-case cutoff of 2000 elements, for which the problem is solved with `std::sort` in both algorithms. The framework was compared with the ones provided by Intel TBB, OpenMP and FastFlow. If not specified, the tests were executed on a Intel Xeon PHI Knights Landing (KNL), over 2^i processors for $i = 0, \dots, 8$. Each test was repeated 5 times and the results depicted in the following images represent their average value.

Policy comparison As expected, by executing the tests adopting different balancing policies we can observe changes in scalability: as shown in Figure 6, we can see that, increasing the number of parallel workers, the test executed with **`relaxed`** policy overtakes the ones executed with **`strict`** or **`strong`** policies, while showing worse results with fewer parallel workers. The **`best`** option seems to achieve the best of both worlds, beating all other policy options up to 2^5 workers, where the **`relaxed`** option shows the best results. The **`only_global`** option, instead, always showed the worst results.

Queue sizes and concurrency Figure 7 shows the results obtained by executing the mergesort algorithm with 4 workers and **`best`** policy option over an array of 1M elements on a Intel Core i7-8550U processor, with details about the queue sizes, the value of χ^2_w of each worker, and the time spent scheduling or retrieving task in the global queue during the fork phase.

In the beginning of the execution (Figure 7a), the main thread (which will become later W_3) schedules the first original problem in the local queue of W_0, which will eventually fall in the global queue (near 35.9 *ms*), since[3]

$$\chi^2_0 = 4 \cdot \left(1 - \frac{1}{4}\right)^2 + \frac{4}{3} \cdot \left(-\frac{3}{4}\right)^2 = 3 > \frac{4}{3} = \chi^2_{\max} \,.$$

Then it start spawning the other three threads, which will become W_0, W_1, and W_2. In this case, by the time the three threads have been created, the main thread (now W_3) has already retrieved the first and unique task in the global queue, the same it had scheduled before[4]. The global queue is now empty, and the other workers are waiting for a task to compute. In the meantime, W_3 starts to generate three forks, which will fall in the global queue and will be retrieved by W_1, W_0 and W_2, in this particular order. Then all threads start working on their local tasks, generating new forks at

[3]Actually, the scheduler always assumes that a worker is already working on an retrieved task (not present in its local queue), summing 1 to $|Q_w|$. This will produce a wrong result the very first time a task is scheduled, as shown in Figure 7b, where $\chi^2_3 > 3$.

[4]This is a particular case, since the created threads may already work on the first task while the main thread continues spawning.

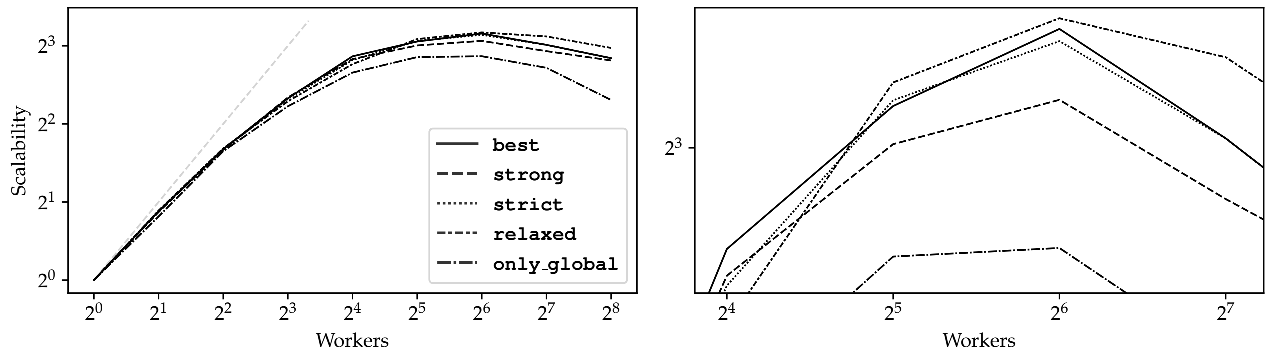


Figure 6: (left) Scalability of the mergesort algorithm, executed over an array of 10M elements, using different balancing policy options and **(right)** a detail of the previous figure. Both plots are in log-log scale.

regular intervals, preserving the load-balancing until after the 41st millisecond, but still accessing to the global queue sporadically, as showed in Figure 7b. The global queue never stores more than two tasks throughout the whole execution of the fork phase.

Performance and scalability Figure 8 shows the performance obtained with the proposed model compared to the ones of the other frameworks. We can see that its trend is somewhat analogous to the one obtained using Intel TBB in the execution of mergesort, reaching higher scalabilities than the ones observed using OpenMP or FastFlow. Quicksort produced worse results, instead. This may be caused by the nature of the algorithm: quicksort can produce uneven partitions of the problem in input, and since the load-balancing is based on the number of tasks (not on the dimension of their input), the resulting distribution can become highly unfair. Another side effect may be that, even if the combine phase of the quicksort is almost non-existent, a large number of data-dependent join tasks may force some worker to stay idle instead of computing other higher prioritized tasks. This is worsened by the fact that, while in the fork phase the balancing is made "on the fly", depending instant by instant by the current size of the queues, the overall number of computed tasks done by a single worker (which is the one that will be found in the join task queue), may be very much higher than average.

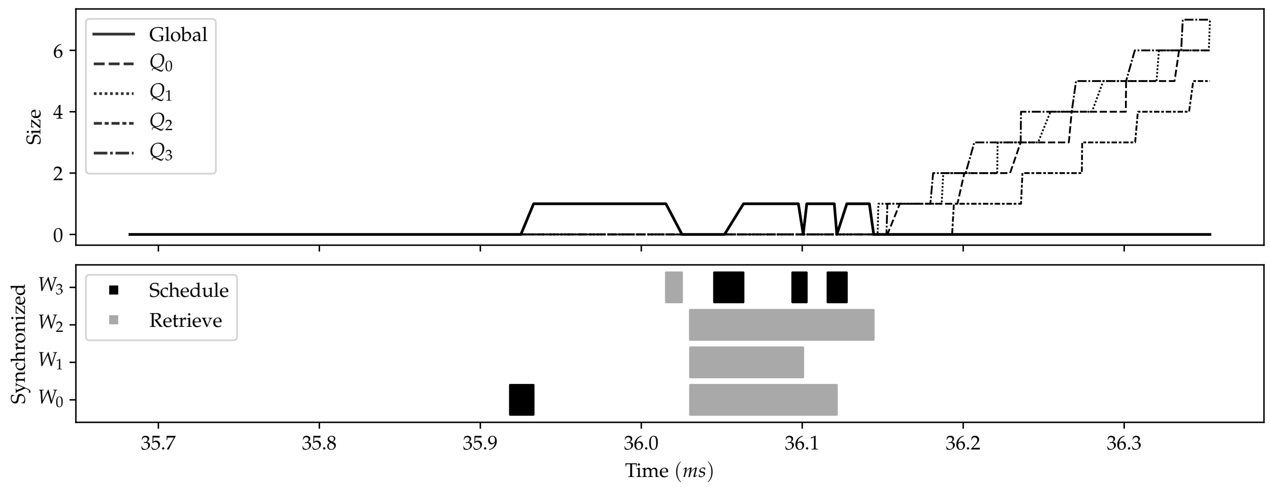
6 Conclusions

Even without load-balancing during the join phase, this model seem to compete with the other tested frameworks. Some improvements should be done in order to obtain better performances: besides the ones already proposed in subsection 2.2, we should aim to a better heuristic that renounces to a good load-balancing in order to achieve a better scalability with a higher number of parallel processors. Another feasible heuristic may take in considerations also the number of total tasks computed by a single worker, solving the problem of load-balancing in the join phase. These proposals are yet to be implemented and will be left as future work.

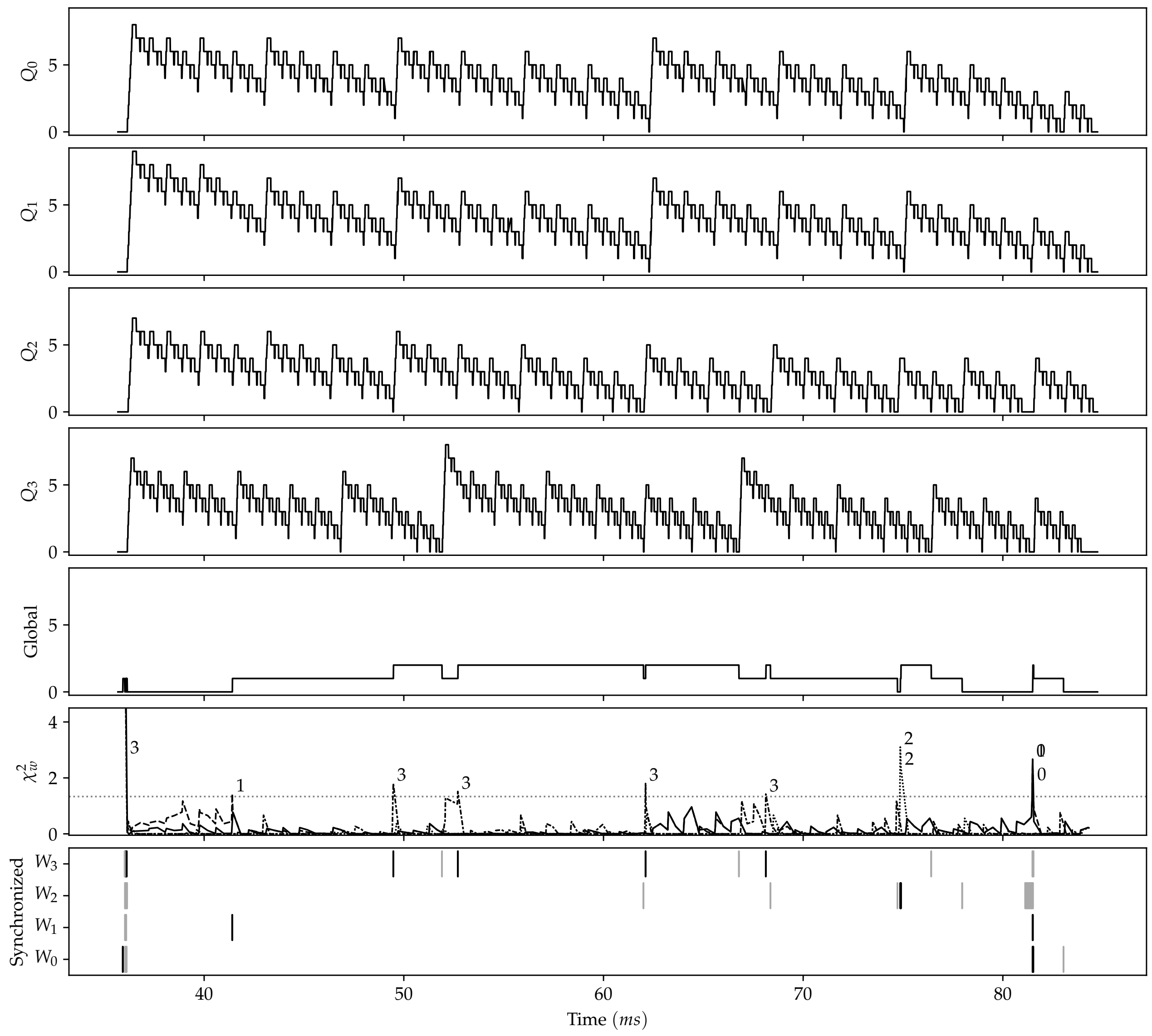
References

[1] Marco Danelutto et al. "A divide-and-conquer parallel pattern implementation for multicores". In: *Proceedings of the 3rd International Workshop on Software Engineering for Parallel Systems*. ACM. 2016, pp. 10–19.

[2] Michael McCool, James Reinders, and Arch Robison. *Structured parallel programming: patterns for efficient computation*. Elsevier, 2012.

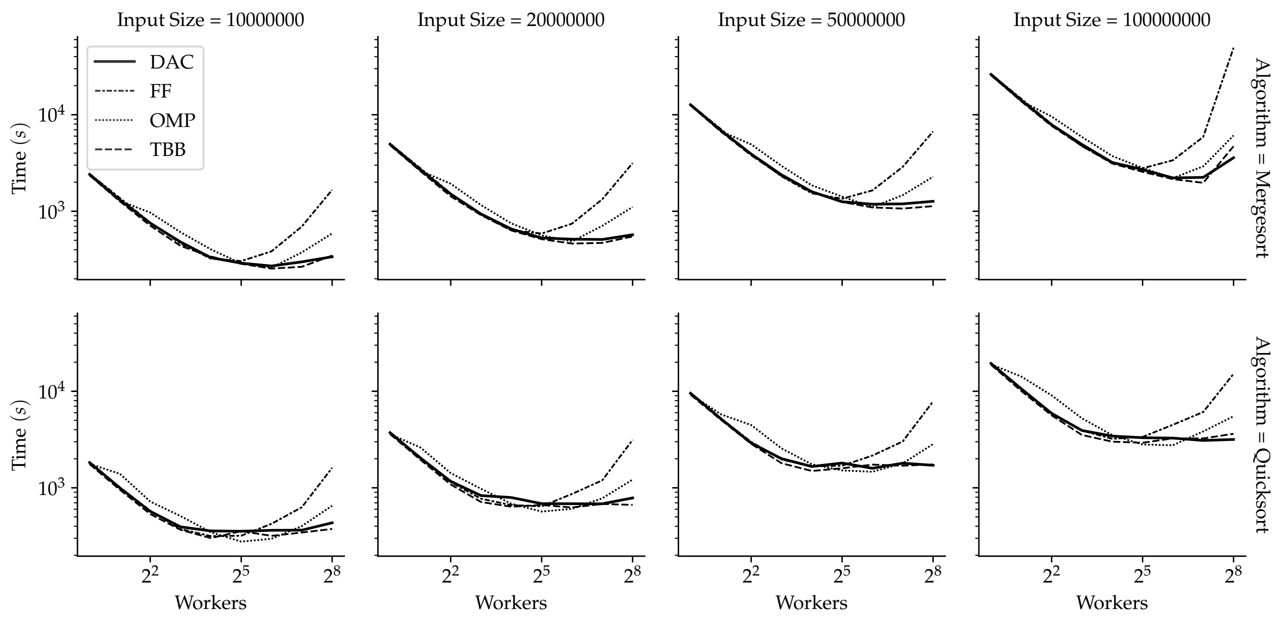


(a) Detail of the queue sizes and concurrent activities in the first 36.3 *ms* of execution.

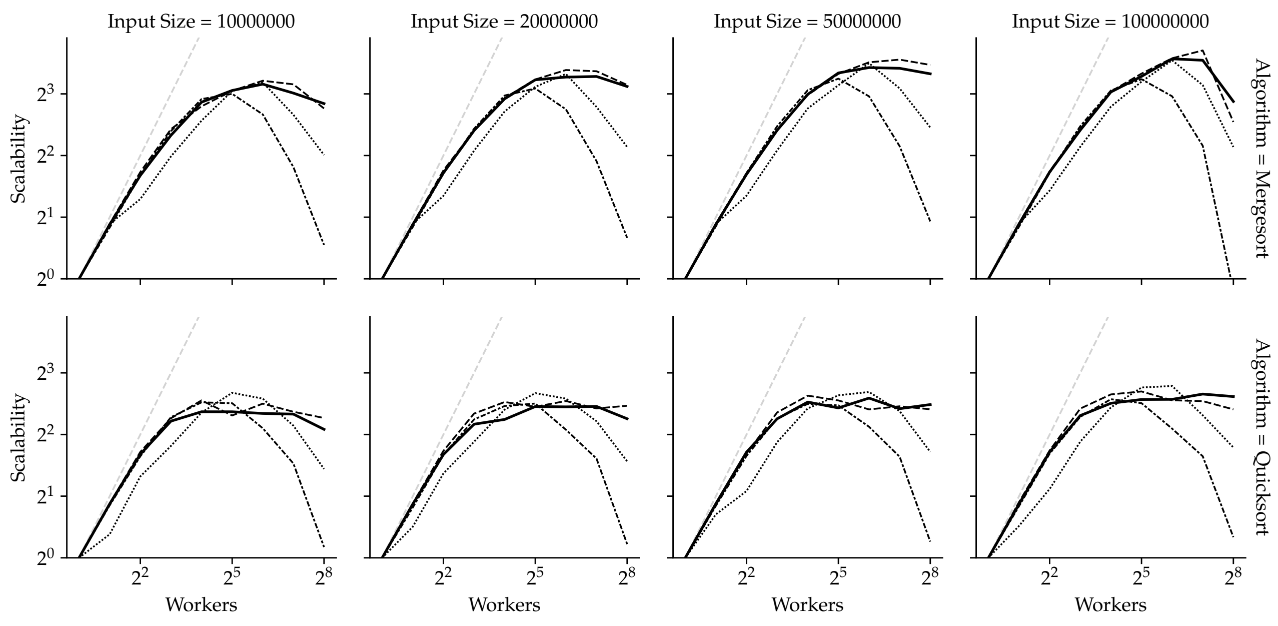


(b) The first five plots show the size of the local queues Q_i, for $i \in [0, 3]$. The sixth plot shows the value of χ^2_w of each worker, marking its index whenever it reaches a value higher than $\chi^2_{\max} = \frac{n}{n-1} = \frac{4}{3}$ (dotted gray line), and setting its value to 0 whenever the worker's local queue has size below expected. The last plot shows the times a worker is scheduling (in black) or retrieving (in gray) a task from the global queue, i.e., its concurrent activities.

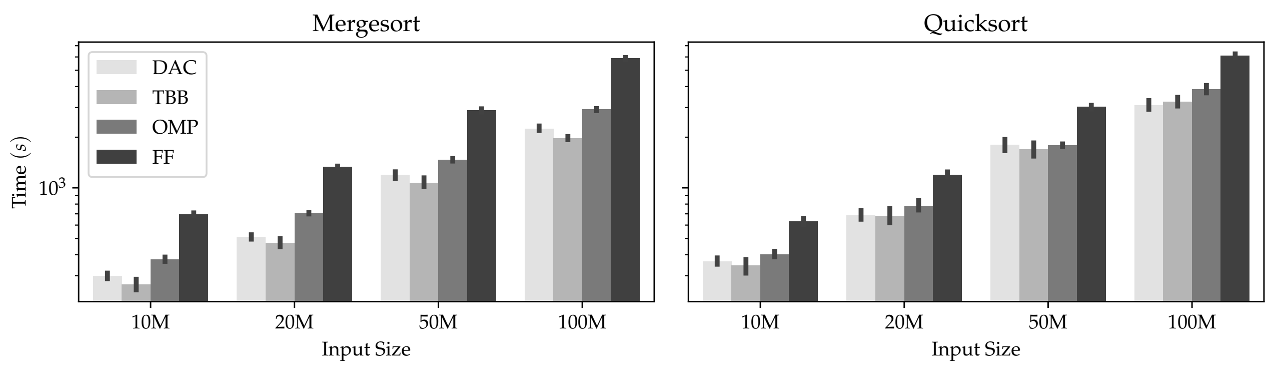
Figure 7: Queue sizes, χ^2_w of each worker, and concurrent activities during the fork phase.



(a) Completion time of mergesort and quicksort executed using different frameworks and input of different sizes. All plots are in log-log scale.



(b) Scalability of mergesort and quicksort executed using different frameworks and input of different sizes. The gray dashed line represents the ideal scalability. All plots are in log-log scale.



(c) Completion time of mergesort and quicksort executed with 128 parallel threads, using different frameworks and input of different sizes. Time is shown in log scale.

Figure 8: Completion time and scalability of the proposed framework, Intel TBB, OpenMP, and FastFlow.