Parallels and Distributed Systems: framework project - Autonomic Farm

Giacomo De Liberali - 580595

February 9, 2020

Contents

1 Introduction

This project involves the implementation of a farm that can increment or decrement the number of workers to achieve a constant service time, provided as input. This report analyzes the design decisions and the obtained performance of the framework on the Xeon Phi KNL machine.

The requirements of the project were asking to take as input the expected service time of the farm, but in the implementation I replaced it with the throughput, since it's easier to design the system around it and it is the inverse of the service time. In the next sections I will refer to throughput instead of service time.

2 Parallel architecture design

The parallel architecture design follow the classic *master-worker* pattern.

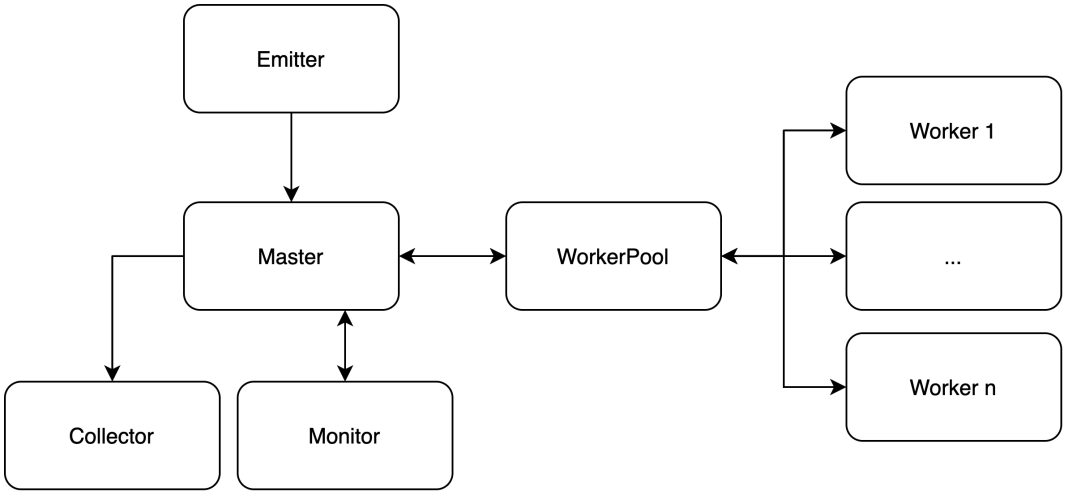

Figure 1: Parallel architecture: master worker pattern

The *master* worker schedules all the input tasks the workers, that once completed notifies it back. Every time a task gets collected by the *master*, it will notify the *monitor* and the *collector*. The *collector* simply stores all the computed results from workers, while the *monitor* takes care of computing the current throughput and deciding whether is better or not to add or remove a worker. If so, the *monitor* sends the command(s) to the master that will act accordingly. If a worker isn't needed anymore, it is removed from the pool of the available workers, so if the *monitor* decides that it will be needed in future it simply gets added again to the pool, without any need to spawn the worker again.

I decided to insert also another entity, the *WorkerPool*, that lies between the workers and the *master*. The *master* indeed does not interact directly with the workers, but rather with the *WorkerPool* which is in charge of finding a free worker and manage its termination together with adding or removing a worker. The *master* is actually a container that orchestrates the flow of data between all the other entities.

3 Performance modeling

The goal of the farm is trying to maintain a constant throughout, as close as possible to the one provided in input. The pace of the throughout would be influenced by input tasks, that could be balanced or not. If we consider the input indicated in the requirements, which consist in a sequence containing tasks that require 4L, 1L and 8L in this precise order, where L is a arbitrary unit of time, we expect a plot similar to the one in figure 2 with an expected throughput of 10.

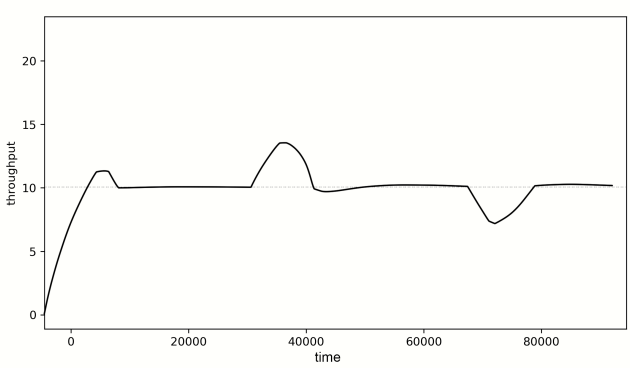

Figure 2: Expected throughput pace with default input 4L 1L 8L

The throughput starts from zero and grows up until it reaches the expected value, 10, and then it remain stable until the next section of the input comes, which require less time and thus the throughput increases. The last negative peak indicates the start of the last section, the one with tasks long 8L.

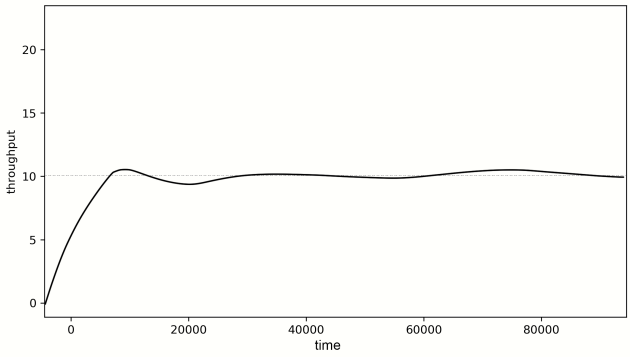

Figure 3: Expected throughput pace with constant input 4L

The figure 3 instead represent the expected throughput variation with an input of tasks that require the same amount of time (eg. 4L). Same as before, the throughput starts from zero and grows, and eventually it reaches a steady state until the end of the execution.

4 Implementation

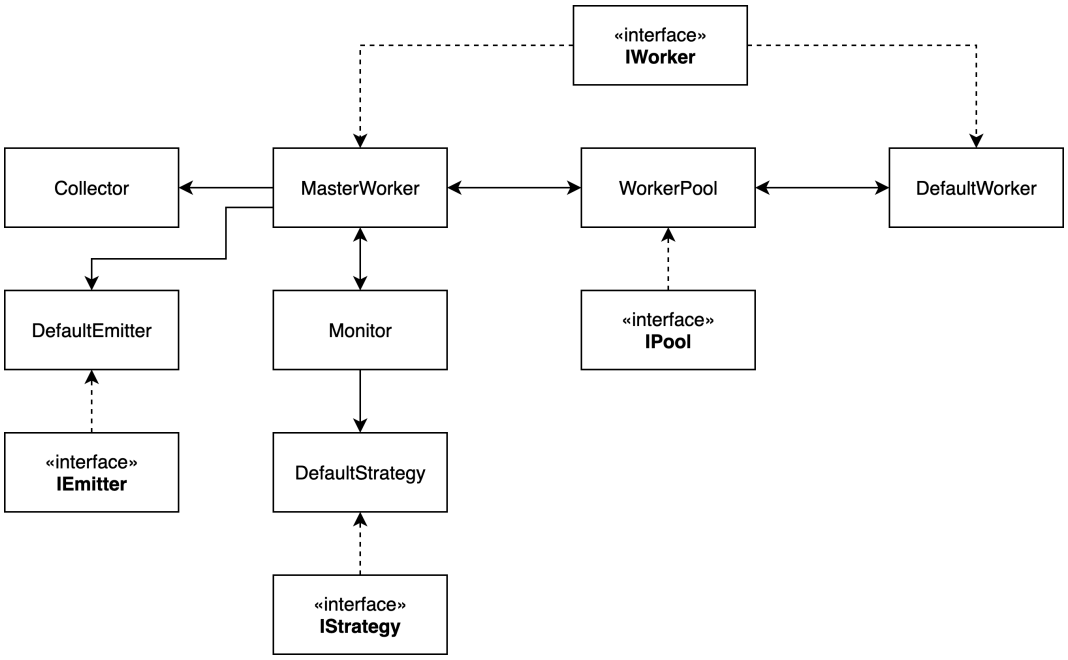

Figure 4: Implementation entities

4.1 Emitter and Collector

The *DefaultEmitter* generates a new task in an on-demand fashion every time the *MasterWorker* requests it. In this implementation it accepts a vector in the constructor and return the next item in each call. The *Collector* has a single *collect* method, called from the *MasterWorker* every time a task is completed.

4.2 Master worker

The *MasterWorker* role is to orchestrate other entities. It requests a new task from the *DefaultEmitter* and asks to the *WorkerPool* to assign it to a free worker. It there are not free workers in the pool, it waits until one comes. When the emitter has ended its stream of tasks, it notifies the *WorkerPool* to join all workers. Whenever a *DefaultWorker* terminates the computation of its current task, it will notify the *WorkerPool* that in turn notifies the *MasterWorker*. Every time it gets this notification it will alert both the monitor and the collector, providing the result of the computation.

4.3 Monitor and DefaultStrategy

The *Monitor* entity receives a notification every time a *DefaultWorker* completes its current task, and keeps a counter of the task collected so far. Once the next notification comes, the *Monitor* checks whether 5 milliseconds are passed from the previous call, and if so it computes the current throughput, otherwise it consider the current throughput equals as the previous one. Whereupon it invokes the *DefaultStrategy* providing the current throughput and the actual number of workers present in the farm. The *DefaultStrategy* has a window of throughputs that is filled at every

invocation by the monitor. Once the window is full and so some troughputs have been collected, the strategy computes two measures: the average of the window and the slope of the regression line between the window's elements. Based on this indicators, it decides if the farm should adjust the number of actual workers, either by adding or removing some. If the slope of the regression line is more than some thresholds, it might decide to add/remove more than a worker at time. Once the decision has been taken the strategy returns to the monitor a integer value representing the commands that the monitor should communicate to the *WorkerPool*. At this point the *Monitor* informs the *WorkerPool* of the decision of the strategy and the command is pushed into a queue of commands managed by the pool.

4.4 WorkerPool

This is the most critical subject of the system. It runs on its own thread, and the fist thing it does is to spawn the initial number of workers requested by the user. Once spawned, those workers are added in a queue of available workers, that contains the workers that are free for computing the next task. When the *MasterWorker* requests to the pool to assign the next task, the pool pop an element from the available workers queue, if any, otherwise it waits until one arrives. So every time a task is assigned, a worker is removed from the available workers queue and when the task is completed the worker that delivered it gets added again to this queue, and the master notified of this event. One more thing that is important to mention is that the *collect* operation of the *WorkerPool* is thread safe and thus cause a race to acquire the lock between all the workers. This is needed as this operation involves the increment of counters, the calculation of the throughput and the insertion inside the *Collector*.

The *WorkerPool* is also in charge of adding or removing workers. The thread of the pool is listening for new commands coming from the monitor, and once it gets one it can either be add or remove a worker. If the command is to remove a worker, the pool pop a worker from the available workers queue and push it to another queue, but of inactive workers this time. The worker in this case is not joined, but rather in a waiting state. In fact when an add worker command comes, before spawning a new *DefaultWorker* first the pool checks if the queue of inactive workers is empty. If it contains a worker it gets moved to the queue of available workers. If in the other case the inactive worker queue is empty, a new worker is spawned and added to the available queue.

In this way the workers are never joined during the execution of the farm, reducing the overhead of spawning multiple times the same thread.

When the stream of the emitter ends and the master notifies the pool to join all workers, the pool send to all workers the *END_OF_STREAM* command joining them. Before sending this command to the workers it must wait that the ones that are computing. To achieve this the pool has a counter of the total number of spawned workers and waits on a condition variable until the size of the available and inactive queues sums to the number of total spawned workers.

4.5 DefaultWorker

The workers run in their own thread. A worker, once spawned, is waiting on a condition variable that the task it has to compute is not null. When the *WorkerPool* assign a task to a worker it basically sets the value for the task and notify the condition variable, that awake the worker that proceeds with the computation. Once the computation is done the task is set to null again and the worker is waiting for the next.

A thing to highlight is that when a worker finishes a task the computation of the throughput made by the monitor is invoked in the worker thread and thus is computed in its own thread. Due to this fact the operation of printing out the current throughput to the standard output influence the throughput measured by the monitor. The plots in section 5 are created by redirecting the program output to a csv file, and since the redirection takes way less time than the print on the standard output the performances of the farm run from the command line may differ for the one provided in this report.

5 Experimental validation

The experimental validation has been done by using different types of input distributions:

- default: the collection suggested in the requirements, a vector containing tasks that requires respectively 4L, 1L and 8L units of time
- constant: a collection of tasks each requiring 4L
- reverse default: the reverse situation of the default one, a collection containing tasks of 8L, 1L and 4L
- low high: a collection containing half of tasks 1L and half 8L
- high low: a collection containing half of tasks 8L and half 1L

For every input collection we show both the plot representing the throughput and the one which shows the actual number of workers inside the farm. The throughput, on the y axis, is expressed in tasks per 5 milliseconds while the x axis shows time since the farm is started, expressed in microseconds.

5.1 Native implementation

We can notice how the farm is behaving correctly, trying to maintain the requested throughput by increasing or decreasing the number of workers.

5.1.1 Default input: 4L 1L 8L

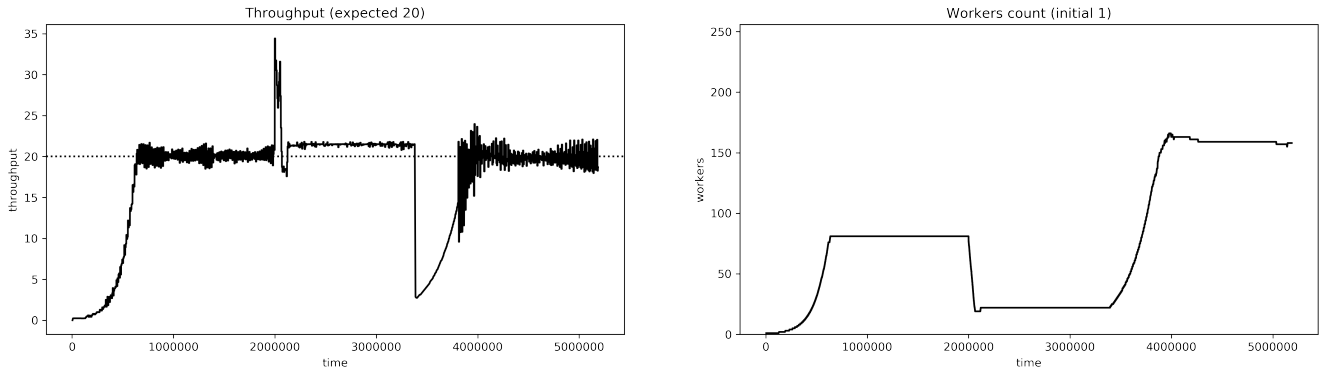

Figure 5: Default: 4L 1L 8L with nw=1, expected throughput = 20

In the right plot of figure 5 we see how the throughput grows up the the given value of 20 and how it bounce around the this value. The two spikes are the indicators of the changing of the input

task duration: the first one shows that the tasks duration has been shrank down to 1L (and thus the throughput increases), while the second shows that the tasks take more time (we notice also that the number of workers after the second spike are the double of the initial section, about 80 and 160).
We can observe also how the slope after the second spike is steeper than the initial one, due to the fact that the workers are already spawned and contribute to the throughput faster.

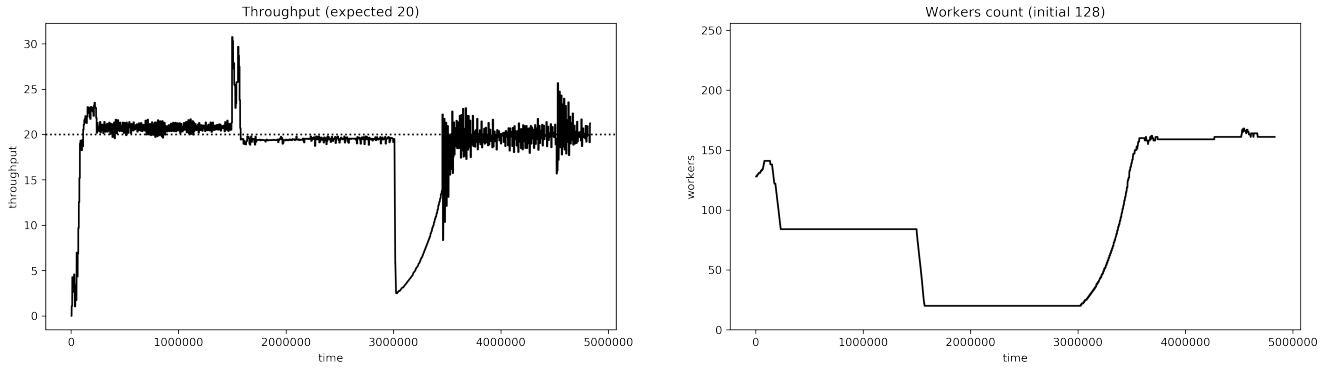

Figure 6: Default: 4L 1L 8L with nw=128, expected throughput = 20

5.1.2 Constant input: 4L

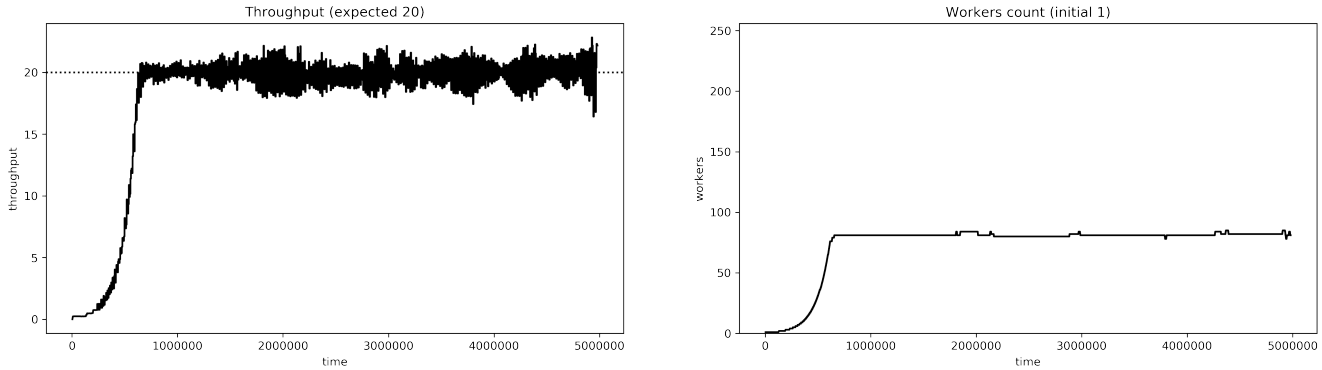

Figure 7: Constant: 4L with nw=1, expected throughput = 20

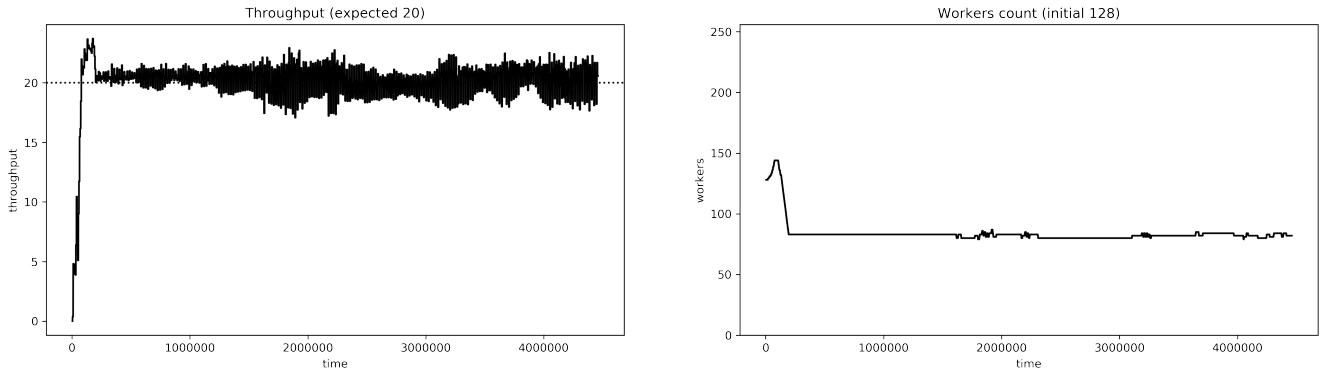

Figure 8: Constant: 4L with nw=128, expected throughput = 20

In figure 8 we set an initial number of workers equals to 128, but since it is too much for the requested throughput the farm scales it down until the throughput is in between the threshold (±10%).

5.1.3 Reverse default input: 8L 1L 4L

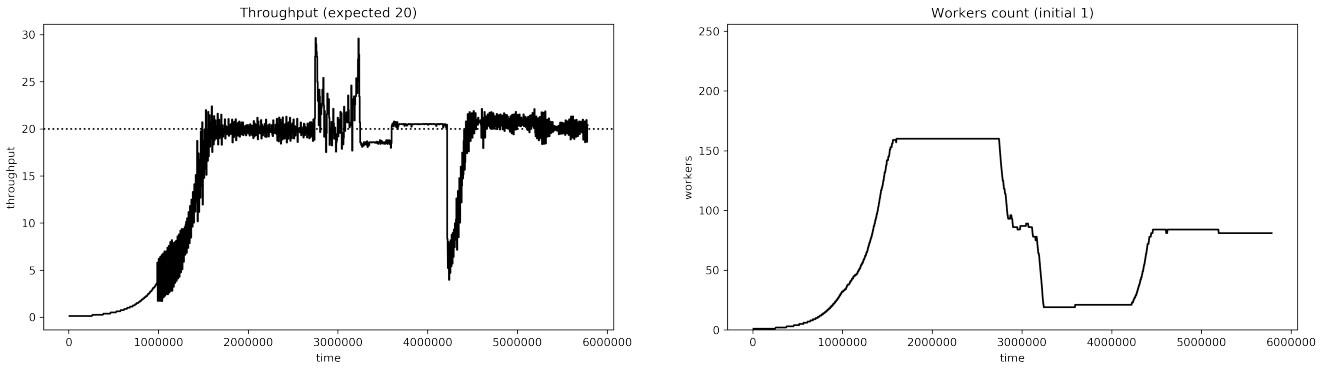

Figure 9: Reverse default: 8L 1L 4L with nw=1, expected throughput = 20

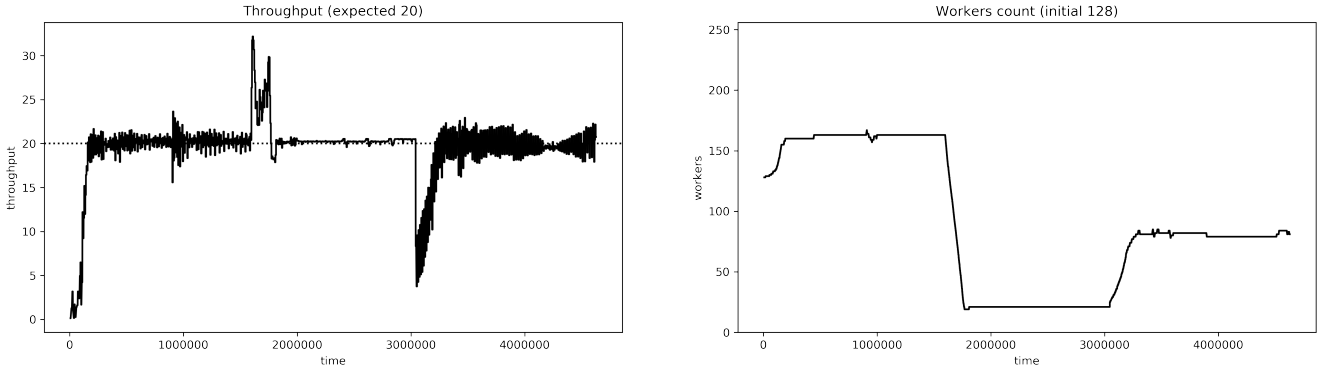

Figure 10: Reverse default: 8L 1L 4L with nw=128, expected throughput = 20

5.1.4 Low high input: 1L 8L

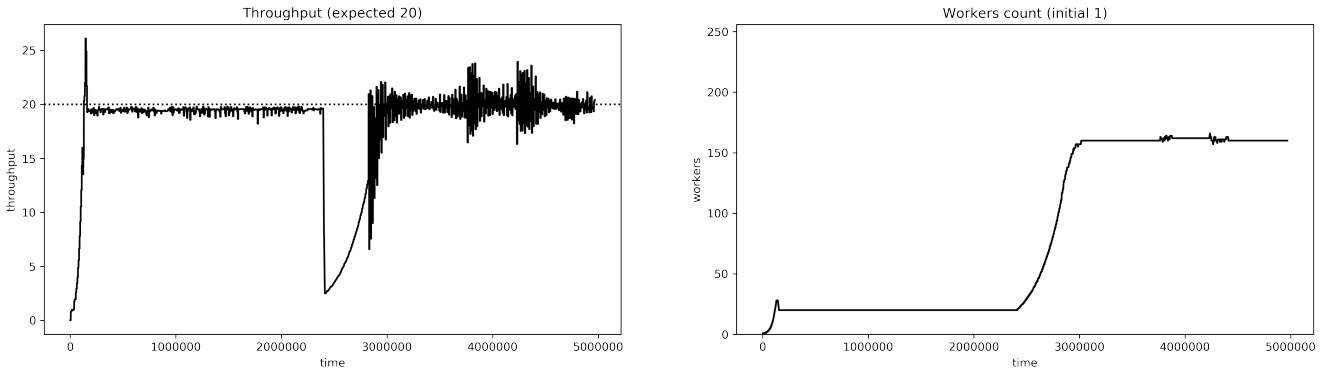

Figure 11: Low high: 1L 8L with nw=1, expected throughput = 20

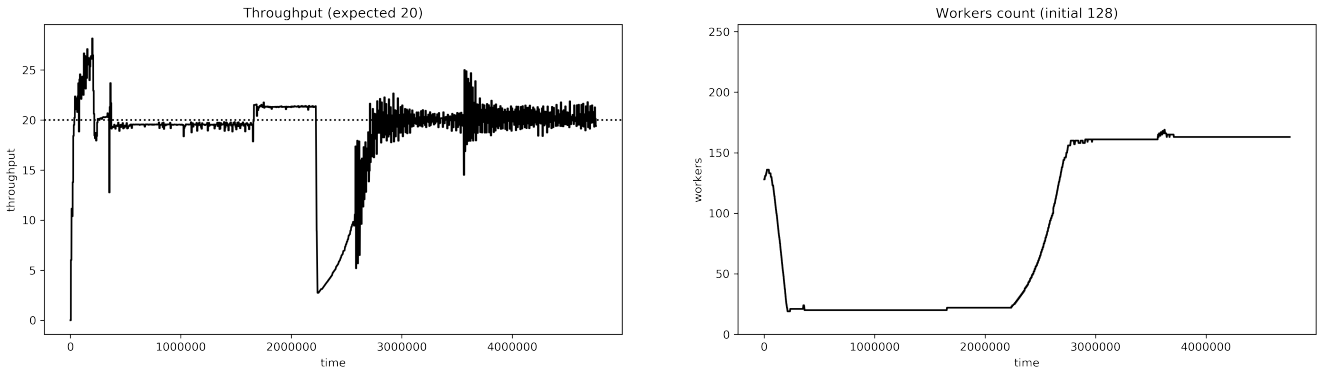

Figure 12: Low high: 1L 8L with nw=128, expected throughput = 20

5.1.5 High low input: 8L 1L

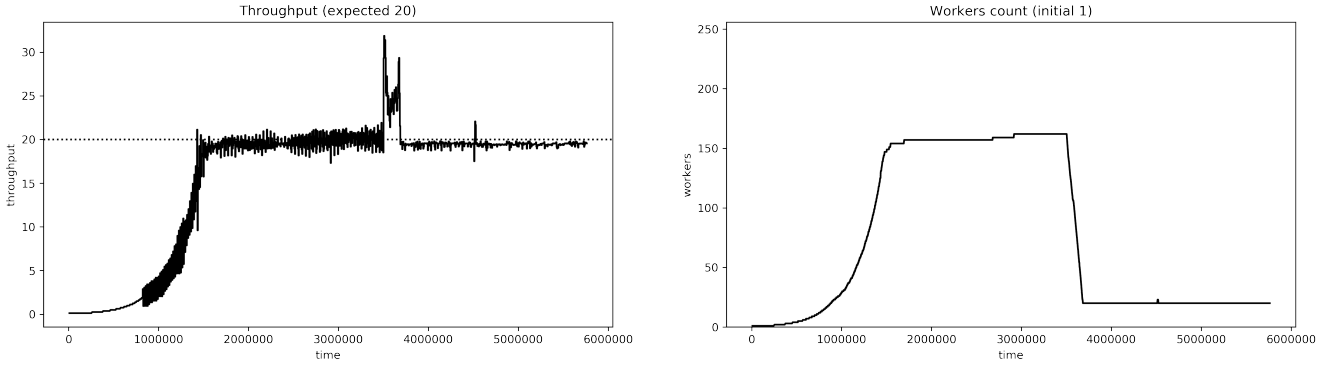

Figure 13: Low high: 8L 1L with nw=1, expected throughput = 20

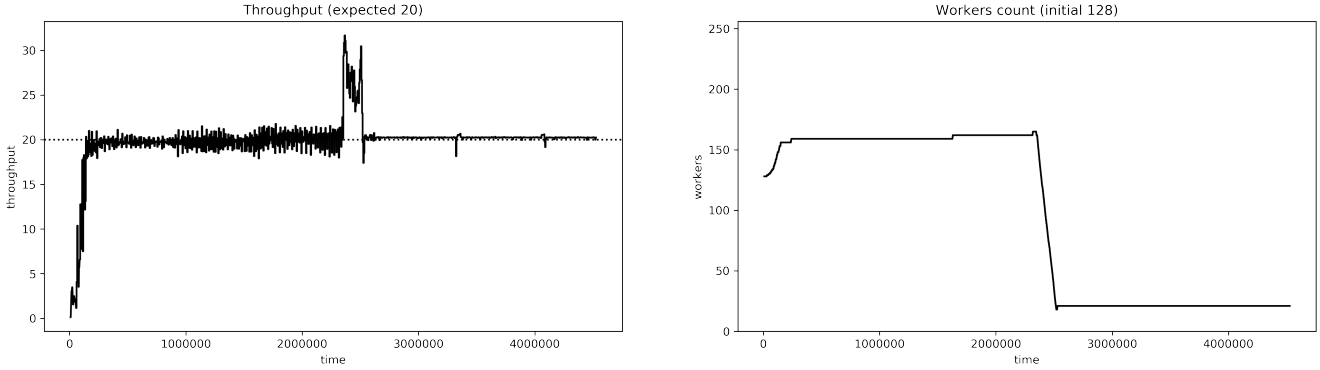

Figure 14: Low high: 8L 1L with nw=128, expected throughput = 20

5.2 Fast Flow implementation

The architecture used in FastFlow is again a master-worker pattern very similar to the previous one. The only big difference is that there is no more the *WorkerPool*, indeed all workers have a feedback channel directly connected to the master. The master worker is a multi-output *ff-node* that schedules tasks to the workers. A major difference from functional view point is that in

FastFlow we can not add more workers that the ones provided at the beginning, so we used as initial and max value for the parallel degree a value of 128 workers.

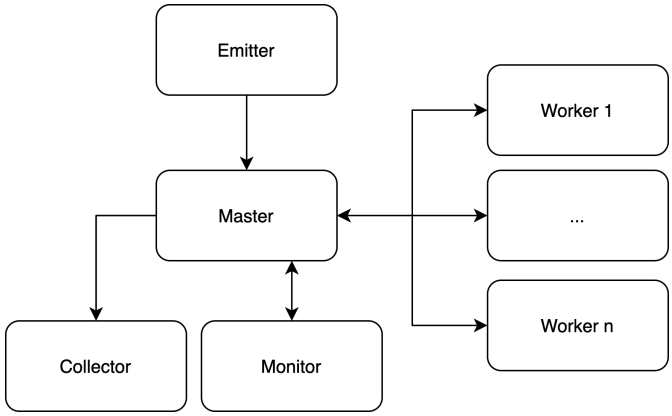

Figure 15: FastFlow architecture: master worker pattern

In this implementation the master worker incorporates all the functionalities that in the native implementation were in charge of the *WorkerPool*: finding a free worker and adding/removing a worker. From the FastFlow's view point the master acts as emitter and collector of the farm.

The monitor, the strategy, the emitter and collector are the same as in the native implementation since they follow the single responsibility principle.

My implementation of FastFlow includes at this moment a bug that prevents the program to terminate properly. The issue concerns the feature of adding/removing a worker: I wasn't able to find a way to properly select a worker that is not computing to freeze it. Currently I'm keeping an index to the last available worker to send to it the *GO_OUT* command to freeze it. But this doesn't take into account that that worker could be processing a task and thus it cannot be freezed yet. This behavior might make the farm starve forever waiting a result that will never be delivered.

5.2.1 Default input: 4L 1L 8L

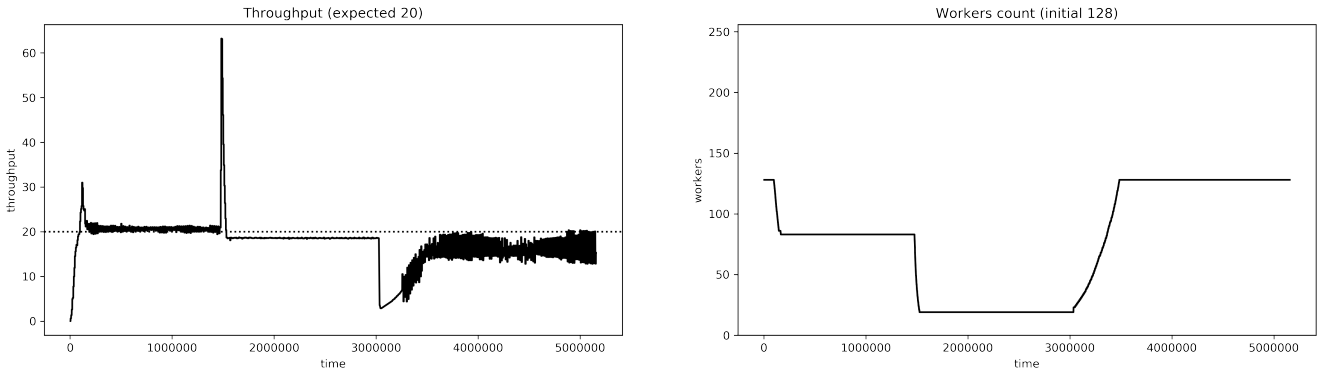

Figure 16: Default: 4L 1L 8L with nw=128, expected throughput = 20

5.2.2 Constant input: 4L

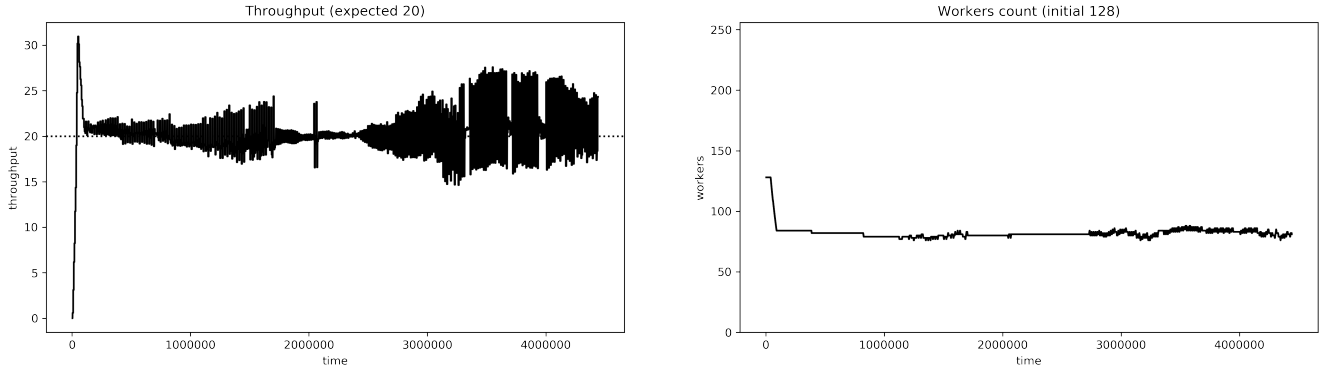

Figure 17: Constant: 4L with nw=128, expected throughput = 20

5.2.3 Reverse default input: 8L 1L 4L

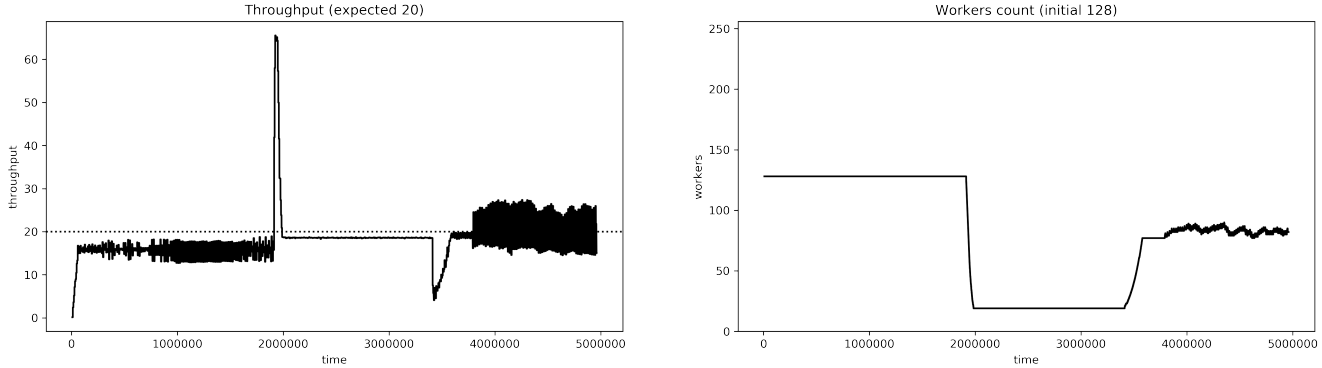

Figure 18: Reverse default: 8L 1L 4L with nw=128, expected throughput = 20

5.2.4 Low high input: 1L 8L

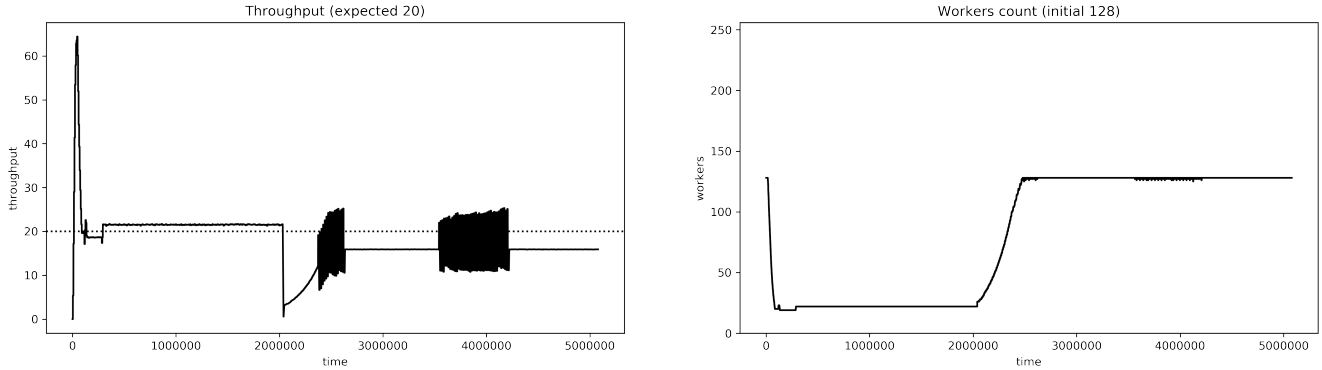

Figure 19: Low high: 1L 8L with nw=128, expected throughput = 20

5.2.5 High low input: 8L 1L

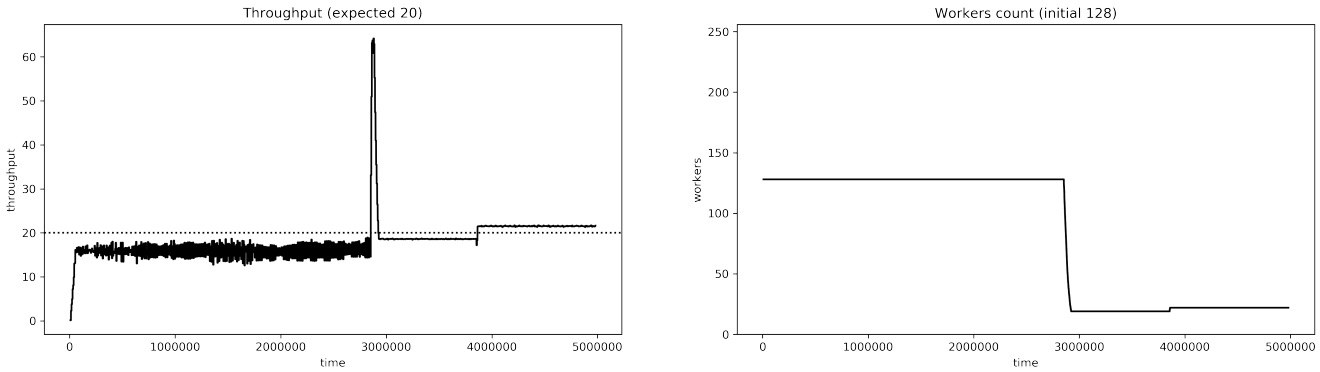

Figure 20: Low high: 8L 1L with nw=128, expected throughput = 20