UNIVERSITÀ DI PISA

DIPARTIMENTO DI INFORMATICA

Master Degree Computer Science - AI Curriculum

Parallel and Distrubuted Systems:

paradigms and models

Autonomic Farm Pattern

Prof:

Marco Danelutto

Candidato:

Niko Paterniti Barbino
Matricola: 638257

ANNO ACCADEMICO 2022/2023

Indice

1 Introduction **3**

2 Architecture **4**

3 Performance Modeling **5**

4 Implementation **7**
4.1 Emitter . . . 7
4.2 Collector . . . 7
4.3 MasterWorker . . . 8
4.4 Monitor and DefaultStrategy . . . 8
4.5 WorkerPool . . . 8
4.6 DefaultWorker . . . 9

5 Experiments and Results **10**
5.1 Native Implementation . . . 10
5.1.1 Default input: . . . 10
5.1.2 Constant Input: . . . 11
5.1.3 Reverse default Input: . . . 12
5.1.4 Low high Input: . . . 12
5.1.5 High low Input: . . . 13
5.2 Fast Flow Implementation . . . 13
5.2.1 Default Input: 4L 1L 8L . . . 14
5.2.2 Constant Input: 4L . . . 15
5.2.3 Refverse default Input: 8L 1L 4L . . . 15
5.2.4 Low high Input: 1L 8L . . . 15
5.2.5 High low Input 8L 1L . . . 16

6 Problems **17**

Capitolo 1

Introduction

The focus of this project is the implementation of a farm pattern that can adapt the number of workers to achieve the best service time according to the input pressure provided as input. We will analyze the design decisions and the results obtained using the "NUMA multicore nod AMD EPYC 7301 32 cores 64hw threads 256G memory"

Capitolo 2

Architecture

The parallel architecture used follows the master-worker pattern. The master worker schedules all the worker's input tasks and once completed, notifies it back. Every time a task gets collected by the master, it will notify the monitor and the collector. The collector stores all the computed results from the 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 this case, the monitor sends the command 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 needing to spawn the worker again. The WorkerPool can be viewed as a sort of bridge between the workers and the master, in fact the master doesn't 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. So the master is actually a container that manages the flow of data between all the other entities.

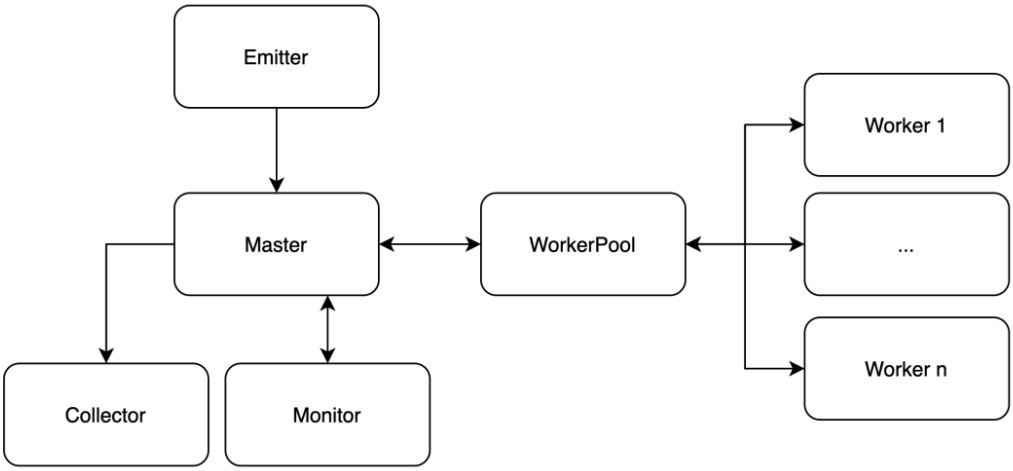

Figure 1: Parallel architecture: master worker pattern

Capitolo 3

Performance Modeling

The farm is tries to maintain a constant throughout, as close as possible to the one provided as input. The steps 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 (L being a arbitrary unit of time, we can expect a plot similar to the one in figure 2 with an expected throughput of 10. 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. 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.

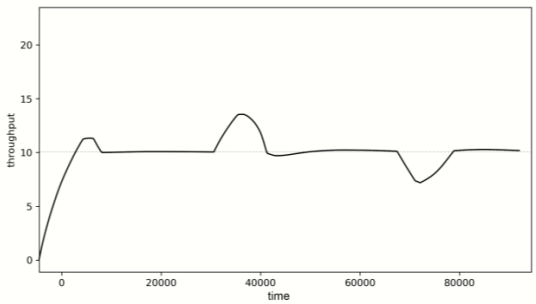

Figure 2: Expected throughput pace with default input 4L 1L 8L

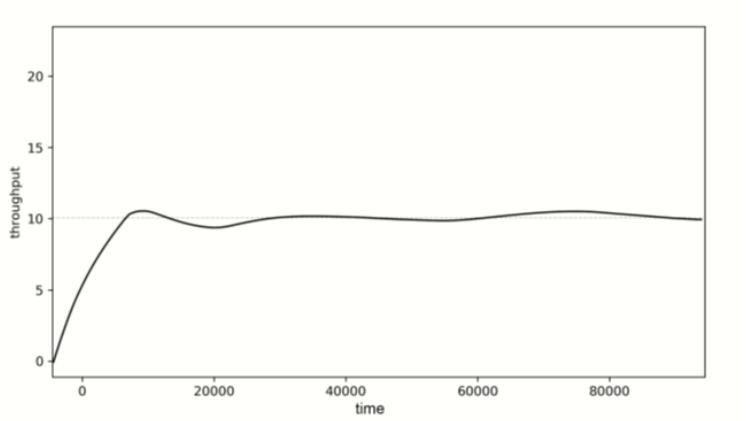

Figure 3: Expected throughput pace with constant input 4L

Capitolo 4

Implementation

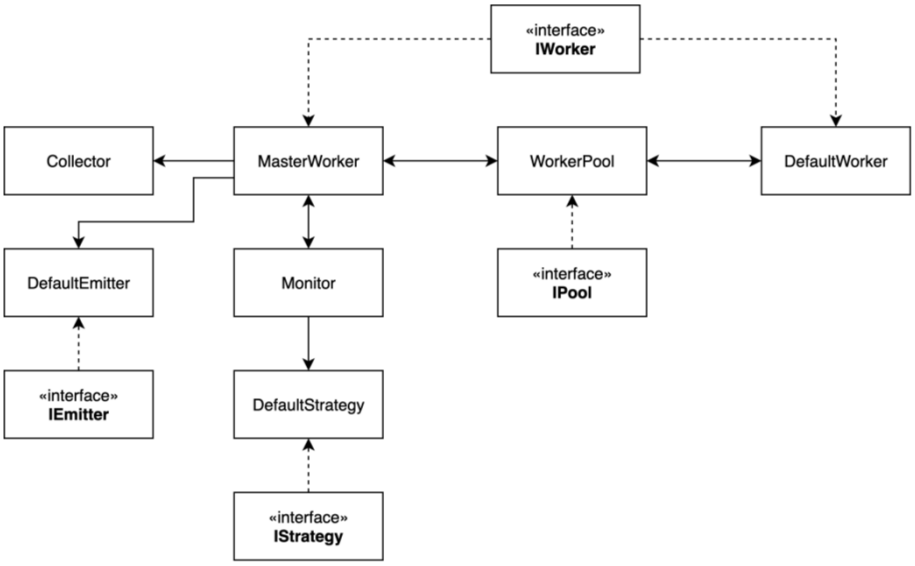

Figure 4: Implementation entities

4.1 Emitter

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

4.2 Collector

The Collector has a single collect method, called from the MasterWorker every time a task is completed.

4.3 MasterWorker

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.

4.4 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.

4.5 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 worker.

4.6 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.

Capitolo 5

Experiments and Results

We made our experiments using different configurations of input:

- default: Vector containing tasks requiring 4L, 1L and 8L units of time
- constant: Collection of tasks each requiring 4L
- reverse default: Collection containing tasks of 8L, 1L and 4L
- low high: Collection in which half tasks are 1L and the other half 8L
- high low: Collection in which half of tasks are 8L and the other half 1L

5.1 Native Implementation

The farm correctly tries to maintain the requested throughput by increasing or decreasing the number of workers.

5.1.1 Default input:

The right plot of figure 5 shows how the throughput grows up given the value of 20 and how it bounces around this value. The two spikes are showing the variation of the input task duration:

1. the first one shows that the tasks duration has been shrank down to 1L (so the throughput increases)
2. the second shows that the tasks take more time (N.B. the number of workers after the second spike are the double of the initial section, about 80 and 160).

Concerning the slope we can see that after the second spike it becomes faster than the initial one due to the fact that the workers are already spawned and contribute to the throughput faster.

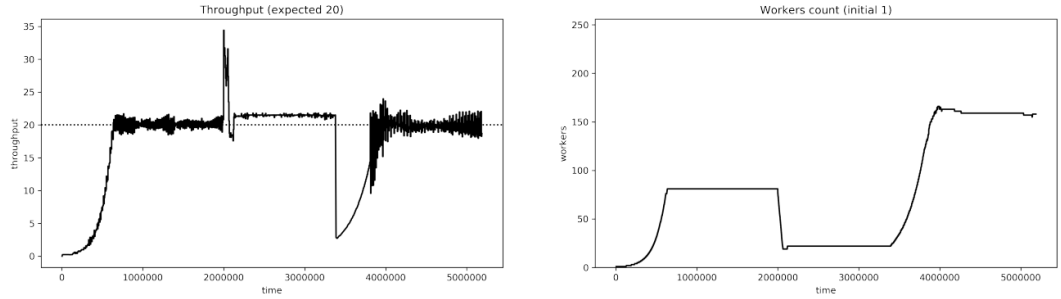

Figure 5: Default: 4L 1L 8L with nw=1, expected throughput = 20

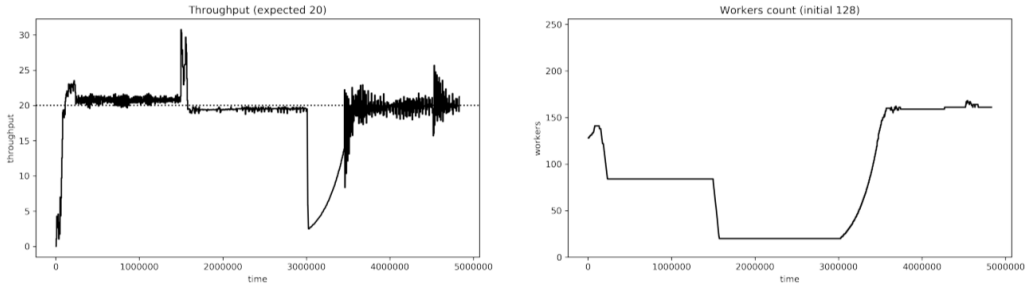

Figure 6: Default: 4L 1L 8L with nw=128, expected throughput = 20

5.1.2 Constant Input:

We set the 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

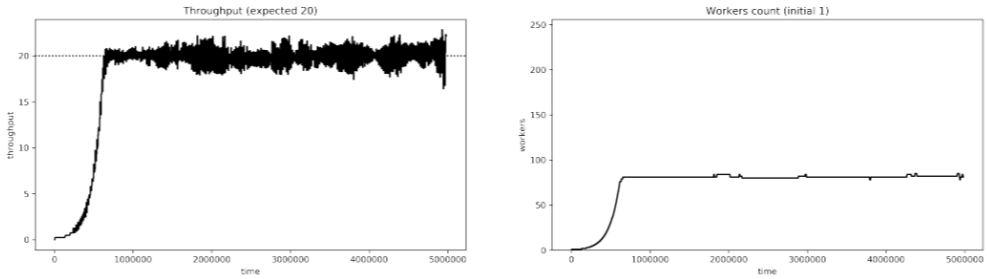

Figure 7: Constant: 4L with nw=1, expected throughput = 20

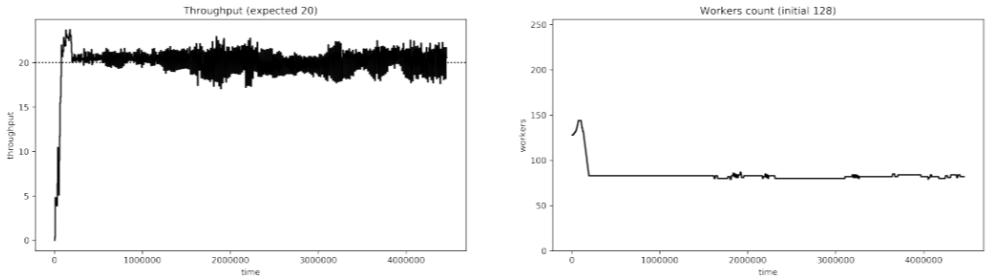

Figure 8: Constant: 4L with nw=128, expected throughput = 20

5.1.3 Reverse default Input:

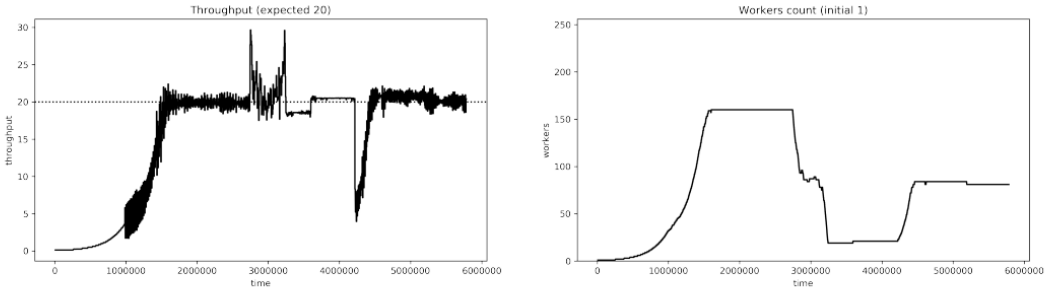

Figure 9: Reverse default: 8L 1L 4L with nw=1, expected throughput = 20

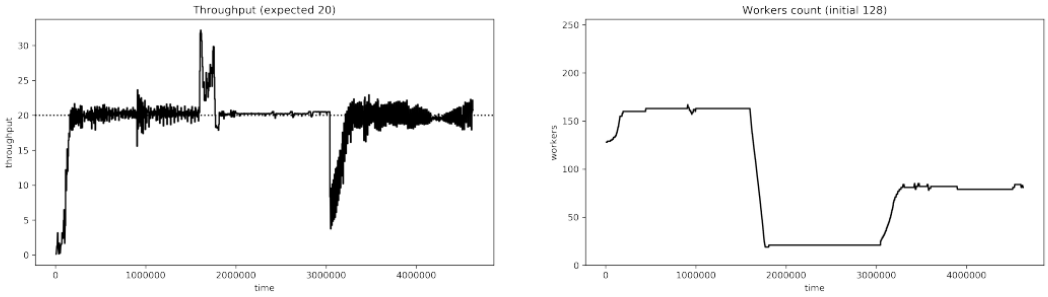

Figure 10: Reverse default: 8L 1L 4L with nw=128, expected throughput = 20

5.1.4 Low high Input:

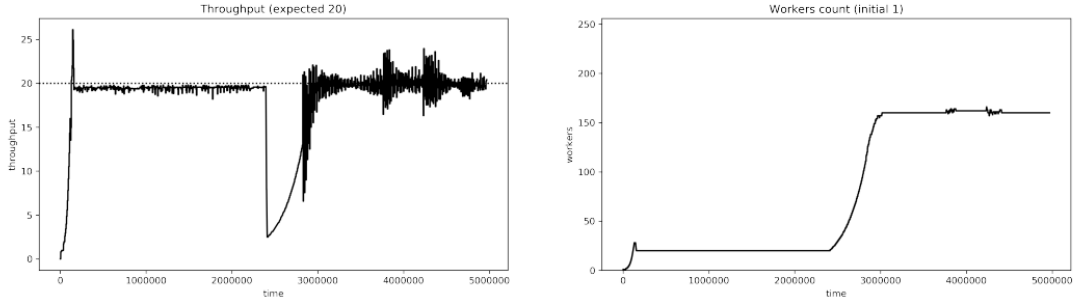

Figure 11: Low high: 1L 8L with nw=1, expected throughput = 20

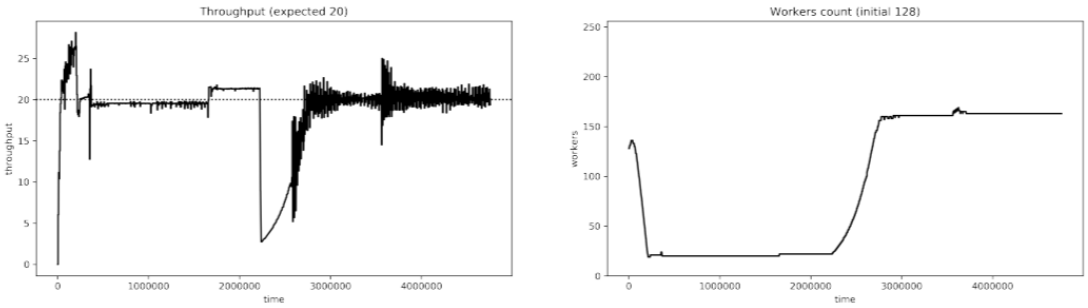

Figure 12: Low high: 1L 8L with nw=128, expected throughput = 20

5.1.5 High low Input:

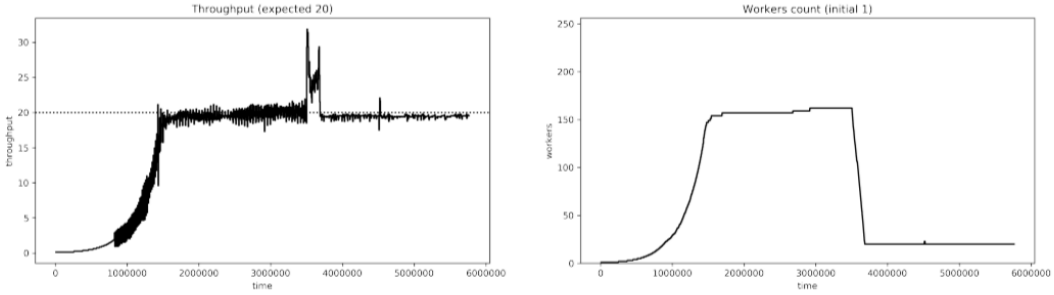

Figure 13: Low high: 8L 1L with nw=1, expected throughput = 20

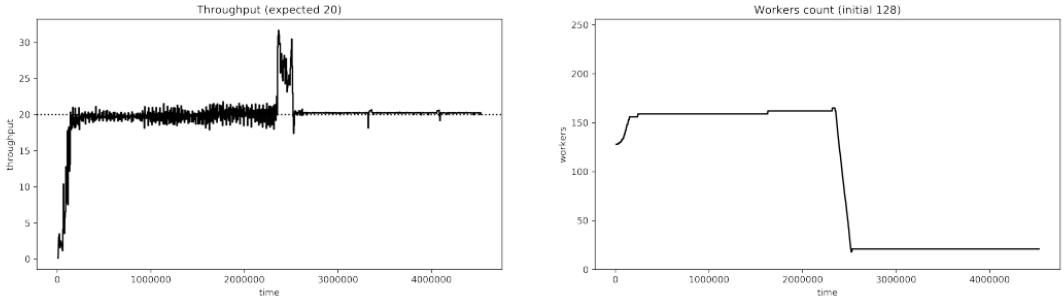

Figure 14: Low high: 8L 1L with nw=128, expected throughput = 20

5.2 Fast Flow Implementation

The architecture used for the FastFlow implementation is a master-worker pattern similar to the naive approach, but there are some differences:

- Absence of WorkerPool
- 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.

- in FastFlow we can't add more workers than the ones provided at the beginning, so we used as initial and max value 128 workers for the parallel degree.
- In this implementation the master worker incorporates all the functionalities that in the native implementation were in charge of the WorkerPool, for example, finding a free worker and addingremoving it).

In FastFlow 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 following the single responsibility principle.

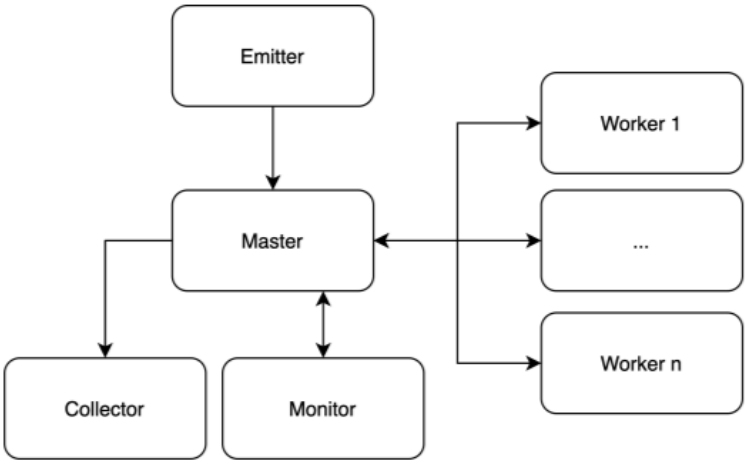

Figure 15: FastFlow architecture: master worker pattern

5.2.1 Default Input: 4L 1L 8L

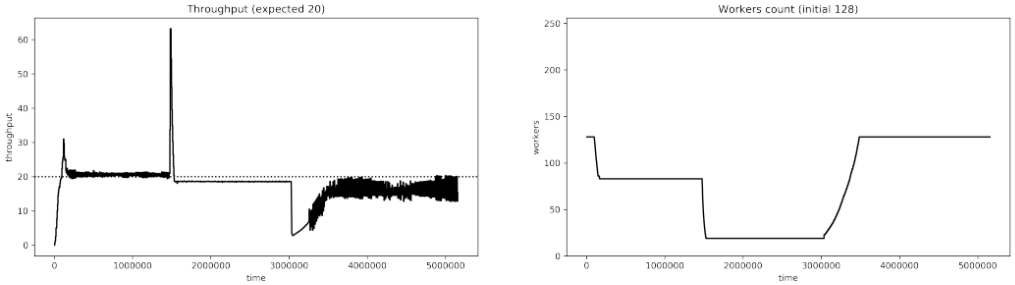

Figure 16: Default: 4L 1L 8L with nw=128, expected throughput = 20

5.2.2 Constant Input: 4L

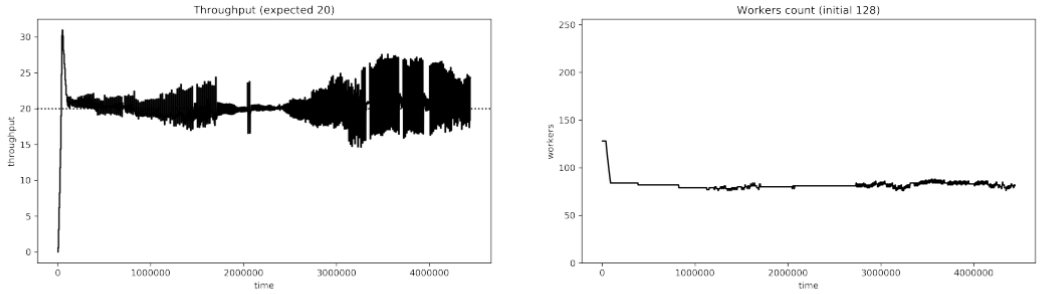

Figure 17: Constant: 4L with nw=128, expected throughput = 20

5.2.3 Refverse default Input: 8L 1L 4L

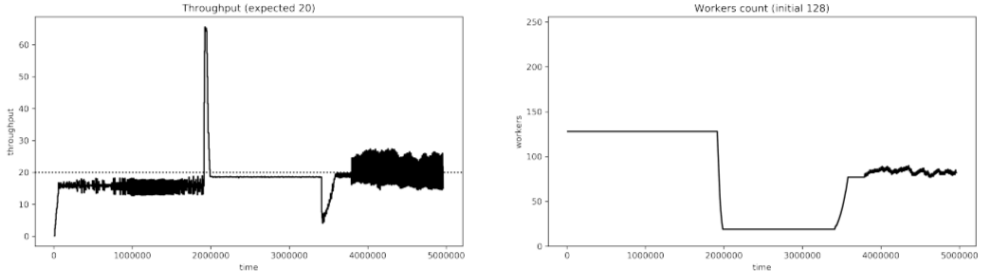

Figure 18: Reverse default: 8L 1L 4L with nw=128, expected throughput = 20

5.2.4 Low high Input: 1L 8L

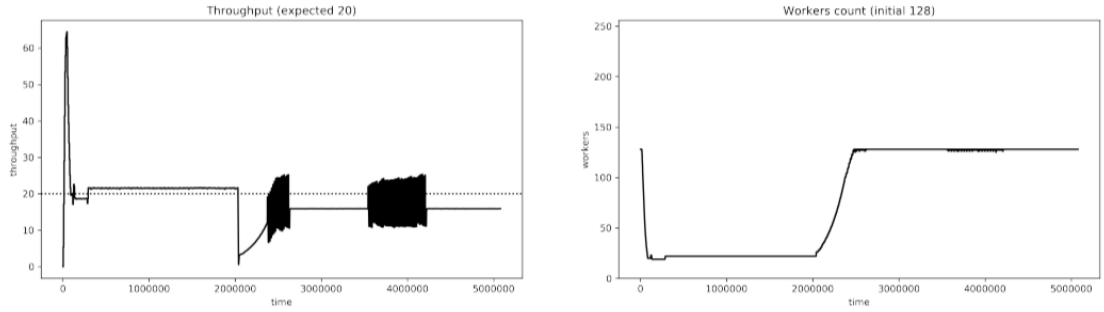

Figure 19: Low high: 1L 8L with nw=128, expected throughput = 20

5.2.5 High low Input 8L 1L

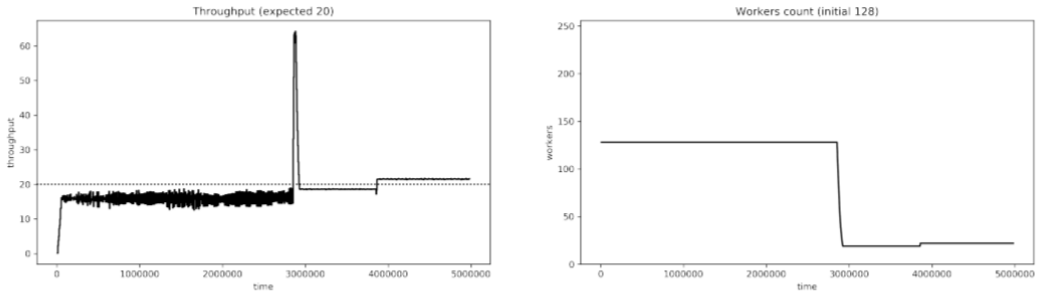

Figure 20: Low high: 8L 1L with nw=128, expected throughput = 20

Capitolo 6

Problems

While working on the FastFlow implementation i found a bug that sadly i wasn't able to solve. After running the script *"FastFlowRun.sh"* the program is unable to terminate. I wasn't able to find the reason as for why this problem occurs still it might cause the farm to wait indifinitely for a result.