

Parallel And Distributed Systems:
Paradigms and Models
a.y 2018/2019

Final Project
Autonomic Farm Pattern

Leonardo Frioli
580611

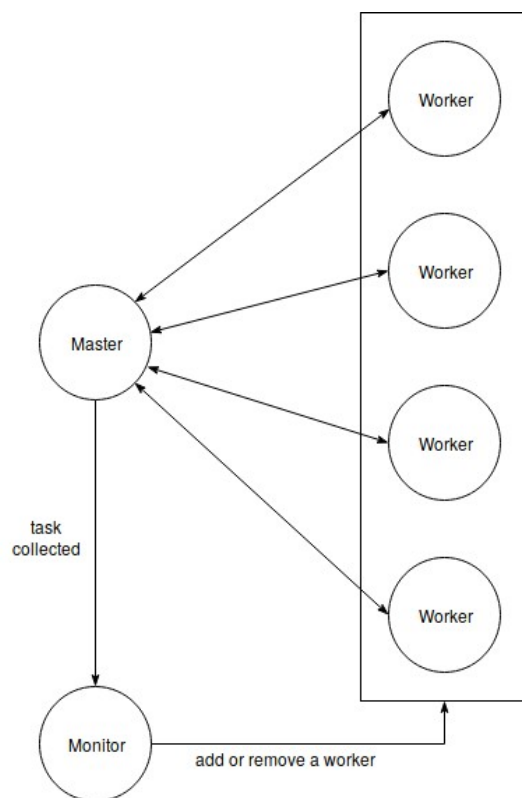
Introduction

In this report we will discuss about the implementation of the Autonomic Farm Pattern. First we will present the parallel architecture and the expected performances; then we will talk about the most important implementation choices and we will show the results of the experiments performed on the Xeon Phi KNL.

Before starting it's important to discuss just one thing.

In the requirements was asked to take in input the expected service time T_s ; in the implementation, instead, we take as input the expected **throughput** because it's easier to reason in term of it and because the service time and the throughput are one the inverse of the other. Also during this report we will talk about throughput instead of service time.

Parallel Architecture Design



The architecture is a Master-Worker pattern with an additional entity that is the Monitor.

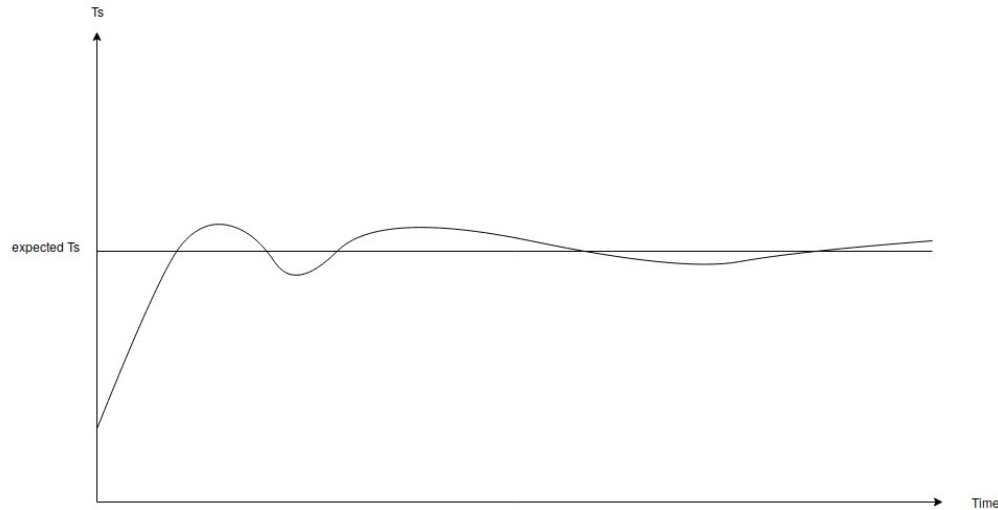
The Master gives the tasks to each worker and, as soon as it collects the result, it will inform the Monitor, that may decide to add or remove a Worker. The Worker will be initialized according to the parallelism degree given by the user and it won't be physically removed: if we do not need a Worker, it will be frozen.

The Monitor gets from the Master the number of task collected so that it can compute the throughput and decide what to do. If it thinks that a Worker should be removed that one will be frozen so that it can be resumed later on if the Monitor decides to add a Worker. This mechanism reduces the overhead due to thread initialization and Worker set up.

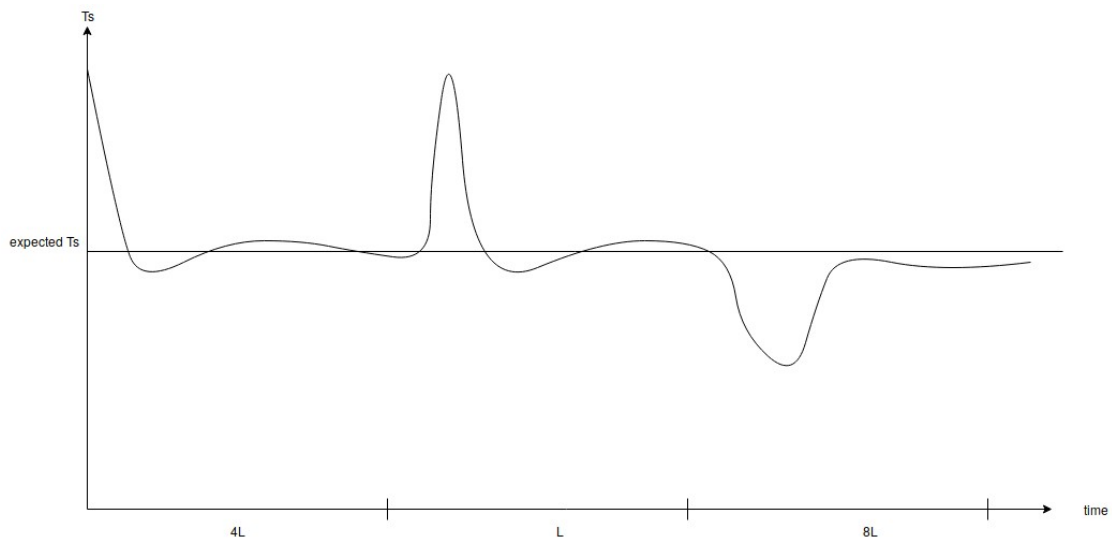
The most relevant implementation details will be explained later in the **Implementation structure and (notable) implementation details** section.

Performace Modelling

The farm will try to maintain a throughput as close as possible to that one taken in input from the user. The performances will depend also on how balanced the workload is: if in input we have a vector with tasks that take more or less the same time to be computed, the farm should stabilize itself around the expected throughput and it won't have many variations.

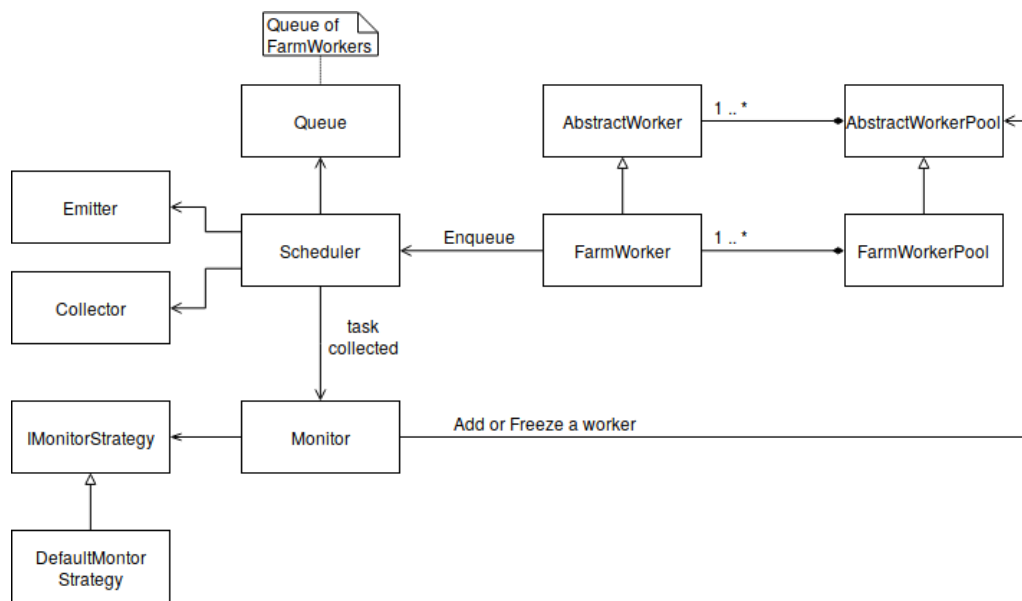


If, instead, the workload is not constant, there will be spikes in the throughput when the unbalanced tasks start to be computed, but after a while the farm will be stabilized again.



In this last figure we can see the expected performances having as input the collection suggested by the requirements.

Implementation structure and (notable) implementation details



Scheduler and Scheduler Queue

The Scheduler is the core of the architecture. The Queue is used by the FarmWorkers to inform the Scheduler that they are ready to take a new task and to give back the result. The scheduler pops the Workers from the Queue gives the result to the Collector and takes the next task from the Emitter to give it to the Worker.

It has been chosen to have just one single Queue to both notify that the worker is ready and to return the results because having one Queue for the result and another to notify the Scheduler would have brought more overhead due to the FarmWorkers synchronization.

The Scheduler runs on the main thread and, after having collected some results, it calls a method of the Monitor (which doesn't have its own thread, but runs in that one of the Scheduler).

Worker Pool thread

The Worker Pool just manages the set up and the freezing of the Workers. It has been needed to run the WorkerPool on a separate thread because when it receives a request of adding a new worker (different from defrosting an already existing one) it can take care of the overhead of spawning a new thread with blocking the Scheduler. The Scheduler needs to collect the task as soon as possible because this can compromise the throughput computation, so it's not good to add to the Scheduler the duty of spawning a new thread.

FarmWorker Locks

Beside the Queue locks there are a couple more mutexes (implemented in the FarmWorker): one it's used to freeze the worker, the other is used to synchronize it with the Scheduler when receiving a task (actually there is another lock implemented in the WorkerPool but it's not too relevant). These locks are needed to synchronize the threads, freeze them, defrost them etc...

We want to point out that only the Queue lock may represent a relevant overhead because it is shared between all the FarmWorkers, instead the other two locks of the FarmWorker (related to the freezing and the task fetching) are just shared between one Worker and the Scheduler or between one Worker and the WorkerPool so that they are contended only between two entities and not between all the Workers.

Default Monitor Strategy: thresholds, window size, trend and average

The MonitorStrategy provided tries to keep some metrics and parameters related to the throughput to take a better decision. When the Monitor receives the amount of task collected by the scheduler it computes the throughput and stores it in a vector. The decision is taken only after having collected some “samples” of the throughput. Here it comes the need of a WINDOW_SIZE that allows to keep the recent history of the computed throughput. Each time the vector is filled (every WINDOW_SIZE samples) the **trend** and the **average** are computed from the past throughputs. The trend is used to understand if the throughput is increasing or decreasing while the average is used to take a decision based also on the previous measures. When deciding which action to take also some **threshold** values are considered to have a margin of tollerability to avoid applying changes every time. Both the WINDOW_SIZE and the threshold values are derived from the input size and the expected throughput.

The MonitorStrategy may also decide to “refresh” the throughput so that it’s computation is not affected too much by the hold computations. By issuing a REFRESH command the monitor will recompute the starting time point used to derive the elapsed time and will consider only the new task collected.

Experimental validation

Inputvectors

Beside the collection suggested by the requirements, several other collections are provided to test the farm with different input workload. The function given to the workers is always the same: it actively waits the ammount of millisecond taken as input and returns it. Some bash scripts are provied to compile and run the experiments with different input values. We will now explain the notation used in the executable to better understand the plots: each main-* has associated a particular kind of vector

main-default	[4L 1L 8L]	(this is the one suggested in the requirements)
main-constant	[4L]	
main-halfdefault	[2L 1L 4L]	
main-updown	[4L 1L 8L 4L 1L 8L 4L 1L 8L]	
main-highlow	[8L 1L]	
main-lowhigh	[8L 1L]	

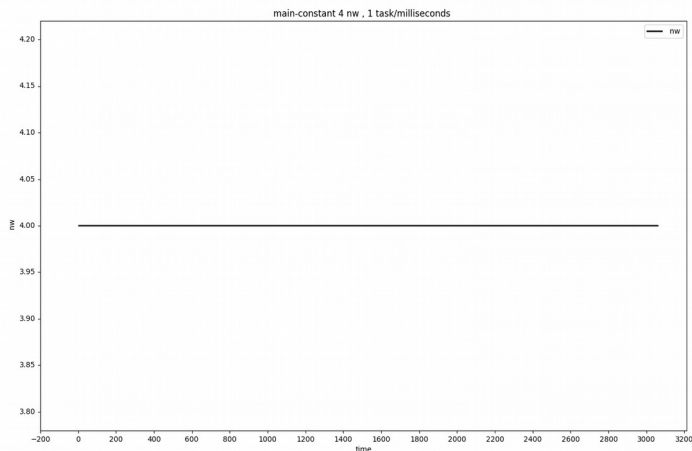
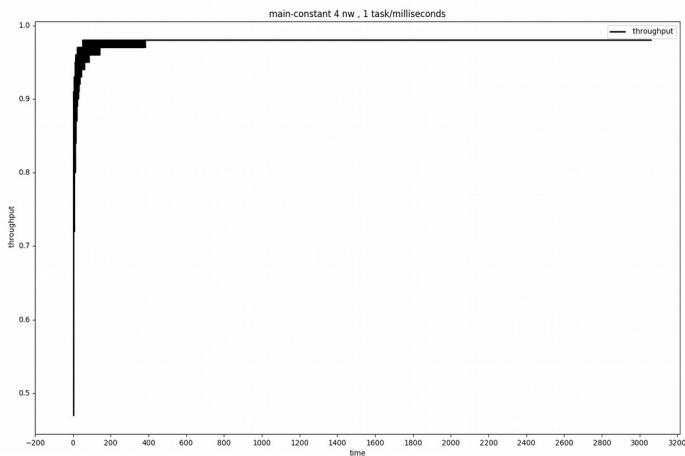
The same notation is used to store the results (in .csv files) and to plot the charts.

Plots

We will se the obtained results running the experiments on the Xeon Phi KNL. For each experiment we will show two charts: one taking into account the throughput variation and aggiustment, and the other showing the increasing and decreasing of the number of workers. Since the specification requires to test the framework with the “default” collection, more experiments have been performed using this vector in input and changing the parameters.

main-constant

We show first the performances with a constant workload to compare them with the first chart in the **expected performances** section.

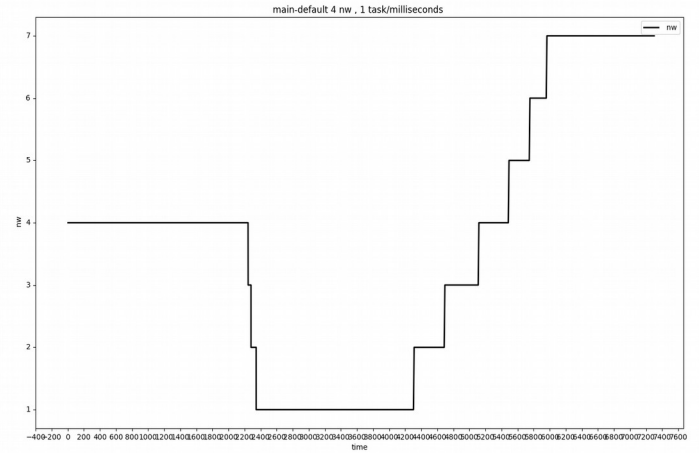
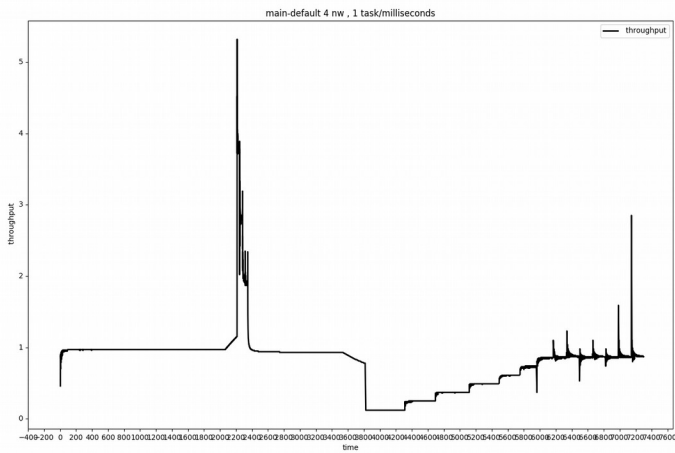


As expected the throughput (left plot) is stable around the given value of 1 task/millisecond and the number of workers is constant.

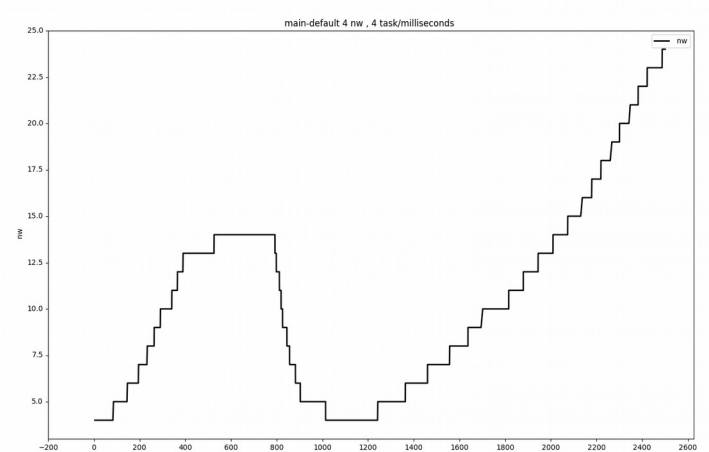
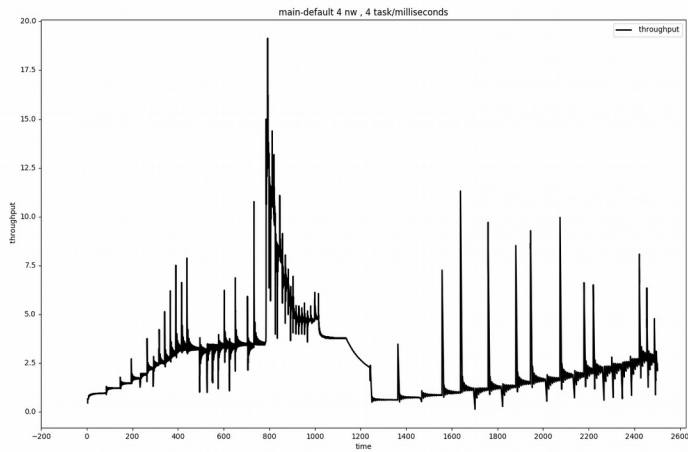
main-default

We will see different results obtained by calling the main-default with different parameters.

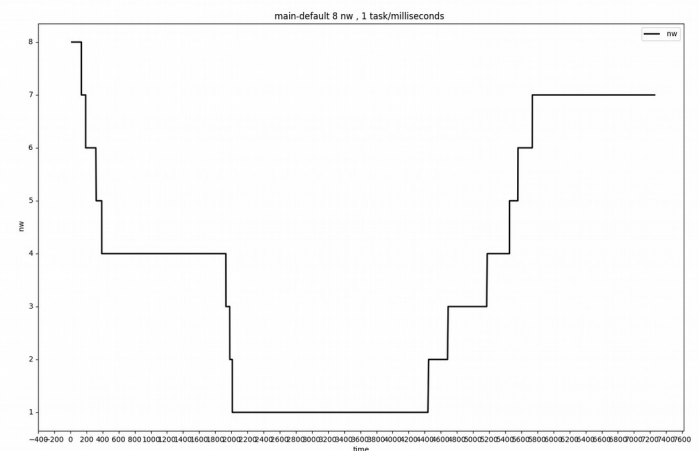
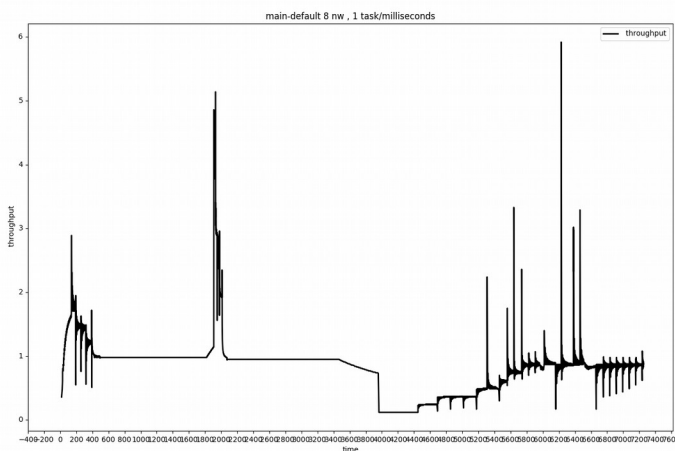
$nw = 4$, expected throughput = 1 task/millisecond



$nw = 4$, expected throughput = 4 tasks/millisecond



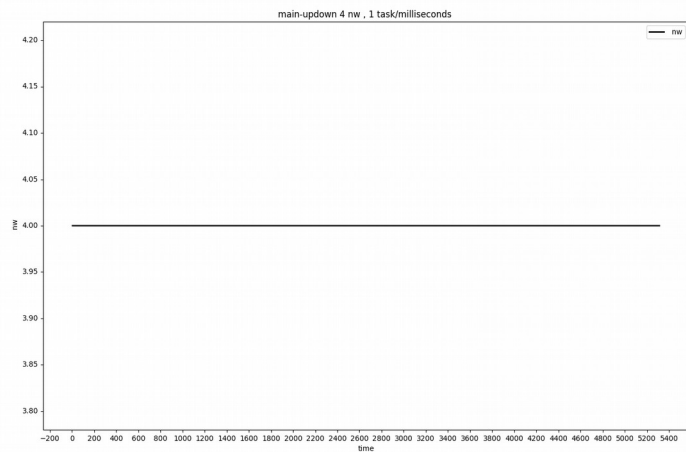
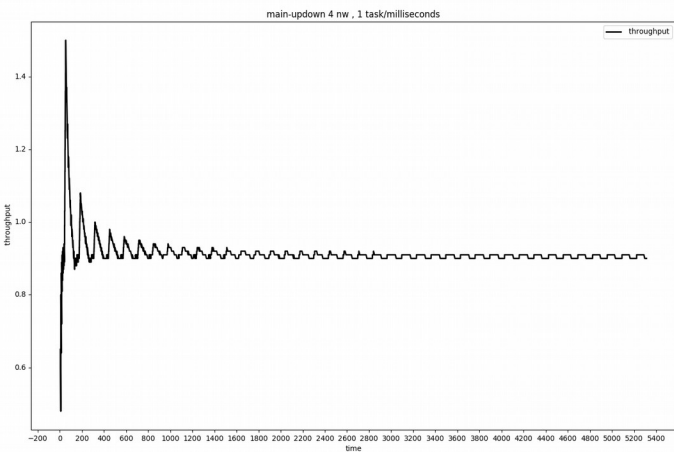
$nw = 8$, expected throughput = 1 tasks/millisecond



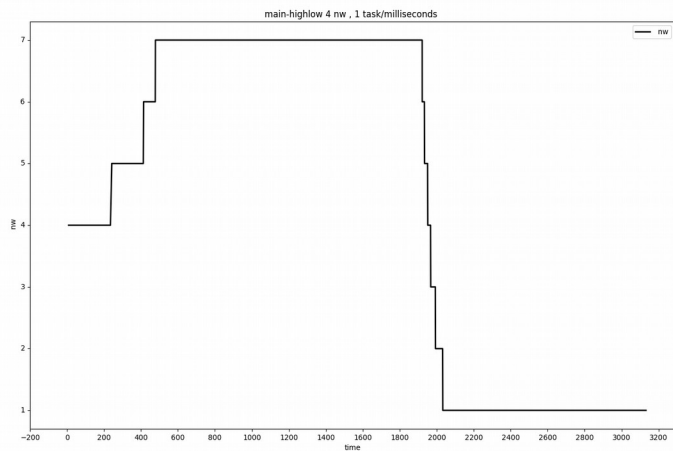
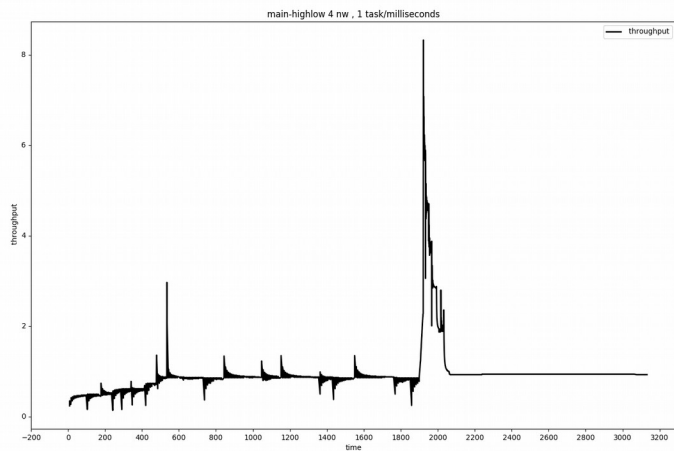
We can say that the trend achieved is the right one: in the above plots it's easy to see that when the throughput is higher than the expected one, the farm reduces the number of workers; when it starts to decrease, the farm increases the number of workers.

We will now show three more experiment results:

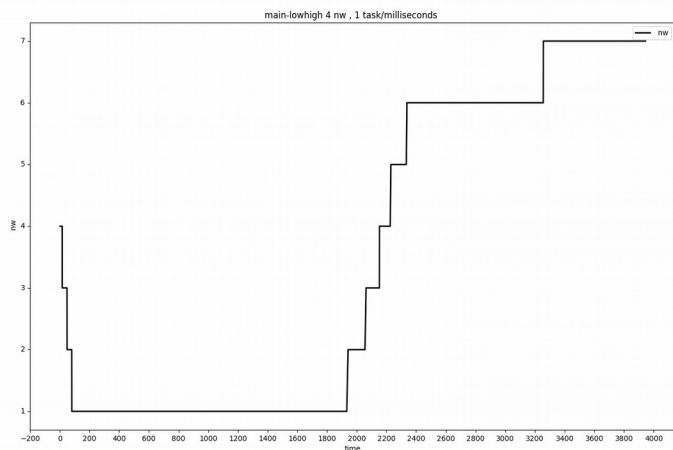
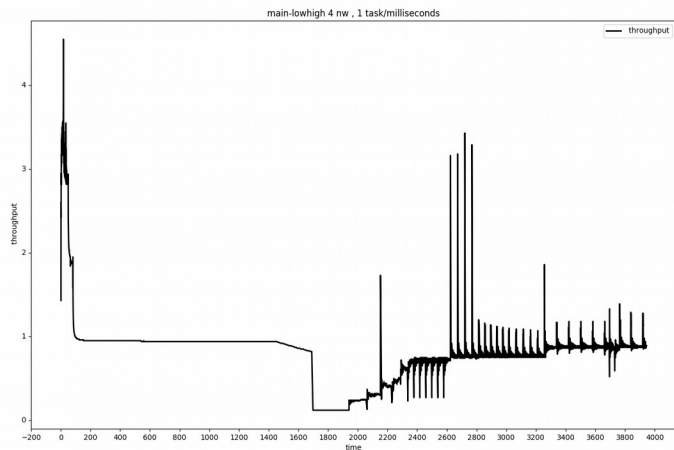
main-updown $nw = 4$, throughput = 1 task/millisecond



main-highlow $nw = 4$, throughput = 1 task/millisecond



main-lowhigh $nw = 4$, throughput = 1 task/millisecond



main-halfdefault

nw = 4, throughput = 2 task/millisecond

