Parallel and Distributed Systems: Paradigms and Models

AUTONOMIC FARM

Framework Project
2018-19

Alessandro Berti
University of Pisa
Computer Science Department

Contents

1 Parallel Architecture Design - pthread

The structure of the implemented Farm is the classic one, therefore made up of *an Emitter*, *a Collector*, *nw Worker*.
Since the farm must dynamically manage the parallelism degree, it was therefore necessary to implement a *MAPE loop* introducing *a Manager* involved in analyzing the data and reacting changing the parallelism degree (Figure[1]).
In particular, the Manager, the Emitter and the Collector are executed on three distinct cores; this decision was taken to avoid that, if Emitter and Manager had been placed on the same thread, in the case of computationally lightweight tasks for the Workers, the Emitter would not have been able to guarantee at least one task for each Worker at any time. To sum up, I looked for the following equilibrium:

$$T_e * nw \simeq T_w$$

where:
T_e = Service Time Emitter, nw = number of worker, T_w = Service Time Worker

The tasks between Emitter, Workers and Collector are passed through data structures that exploit variable conditions for synchronization. The Manager, on the other hand, communicates with the other processing elements using lock and unlock mechanisms.
I also wanted to provide an implementation so that the farm was an Ordered Autonomic Farm. The Autonomic Farm has been implemented in order to manage the variation of the parallelism degree with the concurrency throttling.

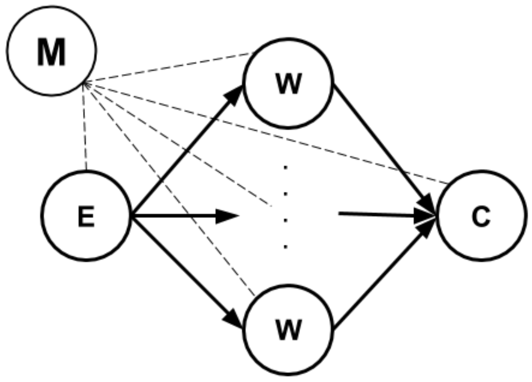

Figure 1: Parallel Design - Autonomic Farm - Pthread

2 Performance Modeling

The behaviour expected from the Autonomic Farm should be that as the worker's service times vary, the parallelism degree is adjusted so that the farm's service time remains constant, close to Ts_{goal}. This implies that in case there are sudden spikes on workers' service time, the ideal behavior is to ignore them in order to contain the stability problem.
The optimal solution would be to have a *Pro-Active* approach to varying the parallelism degree, so somehow to predict the arrival of heavier tasks computatively. But this solution is not easy to deal with, so the problem has been relaxed to an *Active* approach which modify the parallelism degree only if it detects a significant and constant variation of the service time. In particular, supponsing a collection made up by tasks following this distribution: 4L, L, 8L, the Autonomic Farm is expected to behave as in Figure 2.

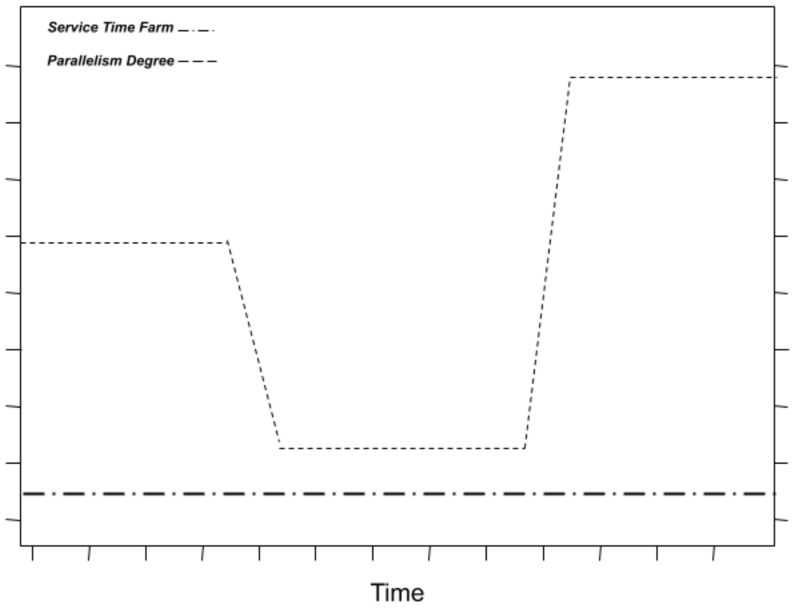

Figure 2: Performance Model
Collection distribution: 4L, L, 8L

3 Implementation Structure

A preliminary analysis was carried out to identify three objectives that would have determined the architecture: the management of the Stability Problem, a simple load balancing policy and the control of whether another application had overlapped with a core of the Autonomic Farm.
This would have required a full management of the threads, in particular it has been necessary to be able to trace at any time on what context was the thread and be able to move it as needed. So the Processing Elements were implemented so that they were sticky. To facilitate the management of the threads was defined a Context class whose objects would be as many as the number of contexts available on the machine. This choice would then allowed to ”sleep” all the threads on a context or ”wake” them up easily, as will be discussed below.

3.1 Thread Scheduling Policy

The choice of sticking the threads, in addition to being more performing and better controlling them, enabled me to experience a different policy for the sleep and wake of a thread, different from the use of variable conditions, i.e. to reduce the degree through the positioning of a thread of a core on another core in which there was already a thread (Figure 4). Doing this, if on the one hand it would bypass the use of variable conditions, on the other hand it would still affect the cache-unfriendliness. However, before showing how I tried to attenuate the cache-unfriendliness is necessary to illustrate what happens when a thread is woken up or asleep. Let’s consider for simplicity a machine with 4 contexts: if the user sets as initial degree 1 as a parameter of the Autonomic farm, it will spawn 4 threads all on the same context (Figure 3). In this way, remembering the choice taken on the fall asleep and wake up of the threads to change degree, is saved the overhead for the creation of new threads.

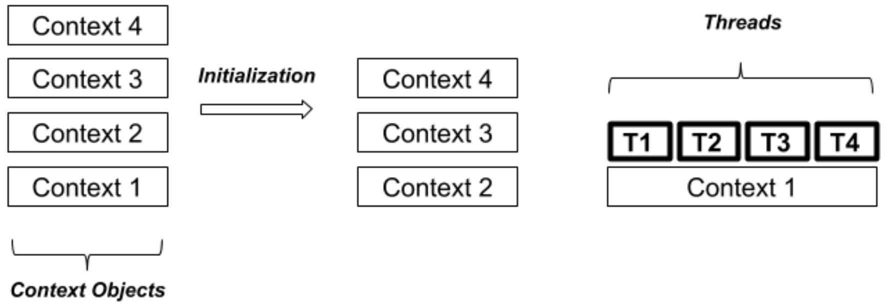

Figure 3: Autonomic Farm Initialization with Min Degree = 1

One concern was that by placing many threads on a single core, this would drastically reduce performance, but by executing some tests, comparing the 256 thread

solution on a single core and the sequential algorithm, the overhead paid for was neglectable.
As for the automatic management of which context to activate, two deques were used that allowed me to rotate between idle and active contexts, thus defining a scheduling policy (Figure 4).

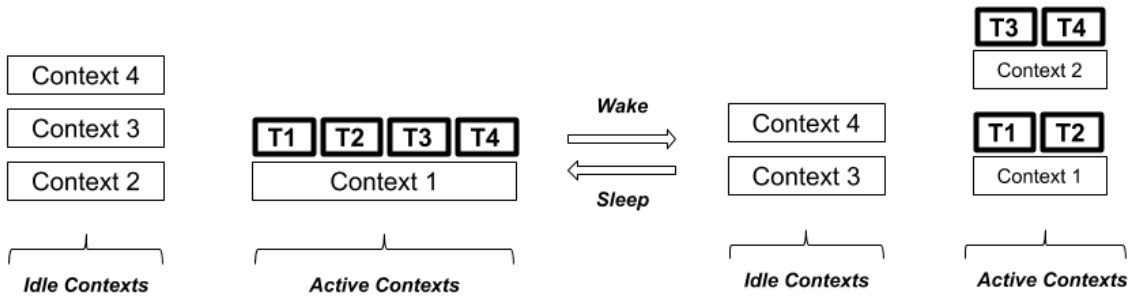

Figure 4: Increase/Decrease Parallelism Degree

With this approach, however, there would still be a problem with load balancing; in fact, threads with this simple policy, would have gone to distribute heterogeneously on the available contexts, affecting performance on long runs. To overcome this problem, every increase or decrease in the parallelism degree, a function is started whose purpose is to divide the number of threads in a homogeneous way on the active contexts. In this way only on a small portion of the thread would have been affected by a context-switch, thus supporting the cache-friendliness.

3.2 Addressing The Stability Problem

Concerning the Stability Problem it was not easy to mitigate it: initially a Pro-Active approach was experimented interpolating two successive TsFarm, but this, however, proved to be a too simple solution for the extent of the problem. So it was preferred to apply changes based on the average of the parallelism degree thus avoiding sudden changes. However, this was not enough because in the case of long runs, the value of the average would have stabilized so much that it would no longer be useful so I opted to use the Simple Moving Average with a time window (settable by the user) on which to calculate the average (Figure 5). This allowed to have an Autonomic Farm that would be affected more or less by sudden changes in the degree depending on the user's needs. In case the window size is 1, it would allow sudden changes of the parallelism degree while with a very high value it tends to become the calculation of the classic average.

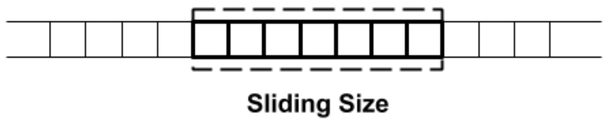

Figure 5: Simple Moving Average in this interval

3.3 Overlaying App Detection

A goal not reached was the one concerning the detection of a possible application that overlapped a context already in use by the Autonomic Farm. The idea in case of detection, was to move all the threads of the contexts on another context present in the deque of the idle contexts. However, this proved to be a problem that was not so easy to manage. First I tried to implement this feature considering the throughtput of a certain thread but then I preferred to relax the condition and consider the standard deviation of the workers service time. In particular have been experimented the following heuristics: *"if there is a lot of discrepancy between the standard deviation of the Ts related to different contexts then it means that another application is positioned on a context of the Autonomic Farm"*. Moreover, according to the forecast, assuming that the tasks were randomly distributed, it could not have been a context with a standard deviation very detached from the average of other contexts's standard deviations caused by the fact that all the threads in that context had gotten very heavy tasks compared to the threads of all the other contexts.
The heuristics has been validated, in fact, by overlapping an application on the same core used by a farm worker, there are changes in the worker's service time. The problem is that I would have expected a change in service time that would have remained constant over the application overlapping, but in practice only random spikes were generated. This did not allow me to establish with absolute certainty that such spikes had to be attributable to an overlapping application rather than to a computationally heavier task.

3.4 The Buffers

One of the first aspects that was thoroughly considered was the type of buffers. Right from the start has been opted for a Bounded buffer because because if on relatively short runs the unbounded would show better performance, on much longer runs it could lead to overflows that would damage the entire computation. Then fixed the decision for bounded data structures has been implemented the following:

- Lock-free data structure
- Non-lock-free data structure

For the first type I found the Lamport Ring Buffer and from there I came across to its implementation in FastFlow from which I referred. However, if on the variable conditions were cancelled, the processors were always busy and fairly energy-intensive. The

implementation of this lock-free structure was left within the project for completeness but disabled because it introduced some redundancies in the code that would weigh down the reading of the code without a real advantage.
As for the lock structure, I experimented with both a safe circular buffer and a safe-queue.
The safe-queue proved to be the most suitable structure because the safe circular buffer needed a pre-allocation of a vector of a required size; therefore as the size increased, on the one hand the startup time slowed down (buffer allocation overhead) and on the other hand it was going to store in memory in a constant and not dynamic way as with the queue.
The buffers has been implemented to manage pointers so, since tasks are passed by pointers, this speeds up the passage of tasks and it has allowed to implement the Autonomic Farm as an Ordered Autonomic Farm.

3.4.1 Bottleneck Detection

A bottleneck detection function was added in the Manager to try to keep the buffers not too large. This control is only activated if the Farm service time is in the range of Ts_{goal}, thus avoiding sudden, unsolicited degree increases.

3.5 Details

Regarding the T_W used in the formula for calculating the farm service time, it has been calculated in the following way: for each active context $T_{context_i}$ the average of the service times of each Worker (T_{w_j}) in a given context is calculated (1), then the average of the values obtained from the previous step (2) is recalculated. This value T_W is the one used in the farm service time formula.

$$T_{context_i} = \frac{\sum_{j=0}^{n_{workers}} T_{w_j}}{n_{workers}} \tag{1}$$

$$T_W = \frac{\sum_{i=0}^{n_{contexts}} T_{context_i}}{n_{contexts}} \tag{2}$$

Although it is an approximation, T_W is useful for containing sudden changes in service time in individual workers, which otherwise would only accentuate the stability problem.
Finally, for the sake of completeness, to calculate the number of Workers needed to keep the farm service time as close as possible to the Ts_{goal}, has been used the formula (3).

$$nw_{new} = \frac{T_W}{Ts_{goal}} \tag{3}$$

nw_{new} = number of workers needed
T_W = Worker service time computed in (2),
Ts_{goal} = service time on which to stay as close as possible.

4 Parallel Architecture Design - Fast Flow

Regarding the Fast Flow implementation has been experimented a solution without a Manager. In this solution, feedback channels between Collector-Emitter and Workers-Emitter were used to send service time to the Emitter of each processing element (Figure 6). In particular, during implementation has been kept the thread scheduling management and parallelism management policy the same as the pthread version. In the code, I referred the version of the farm presented in class [1].

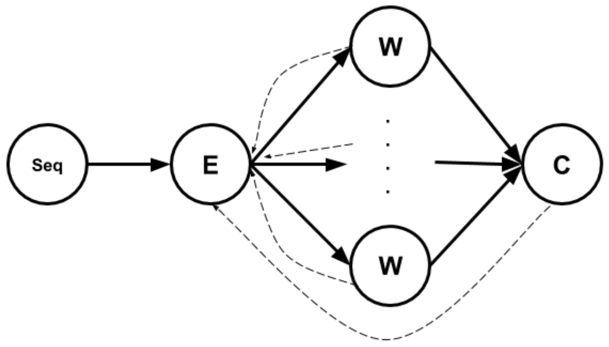

Figure 6: Parallel Design - Autonomic Farm - FastFlow

The choice to use the node ***Seq*** for the stream of the collection was necessary in order to allow the Emitter to manage the feedback of the processing element.

5 Experimental Validation

The experiments were performed on a Xeon Phi KNC equipped with 64 physical cores and mutithreading support at 4 contexts, so for a total of 256 contexts.

5.1 Concurrency Throttling

The Autonomic Farm pthread has been validated using a function and a collection of tasks so that the latency distribution is the following: 16000, 4000 and 32000 milliseconds. The Autonomic Farm was performed by setting Ts_{goal} = 100 milliseconds and SMA window = 20. The very same parameters have been used to test the pthread implementaton (Figure 7) and the FastFlow one.

Despite the first spike that I associate with a startup overhead, I found these results satisfactory because, despite some small degree adjustments in the stabilisation phase, there are no sudden unjustified degree changes. By comparing the optimistic behavior defined by the performance model defined in Figure 2, the results obtained, are in line with the expected result.

[1] `https://github.com/fastflow/fastflow/blob/master/tests/test_multi_output6.cpp`

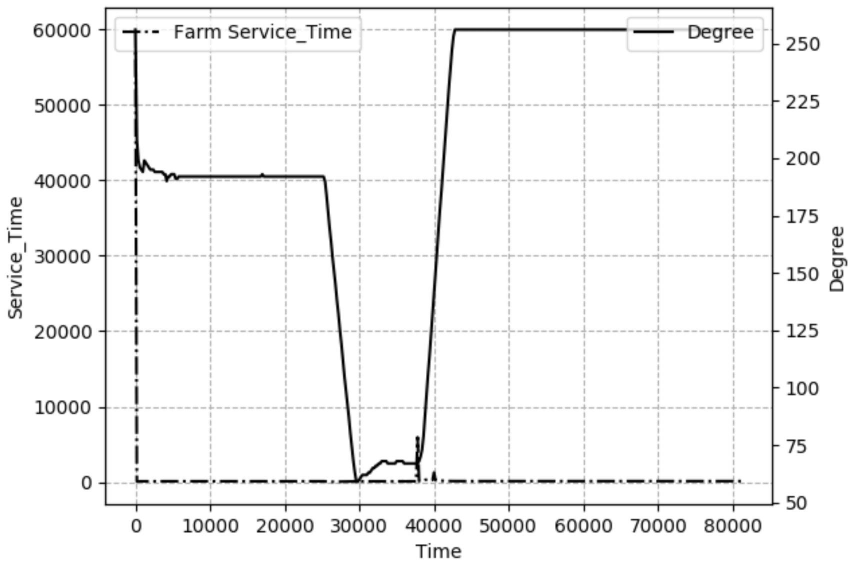

Figure 7: Concurrency Throttling - pthread
Ts Goal = 100 milliseconds
SMA window = 20

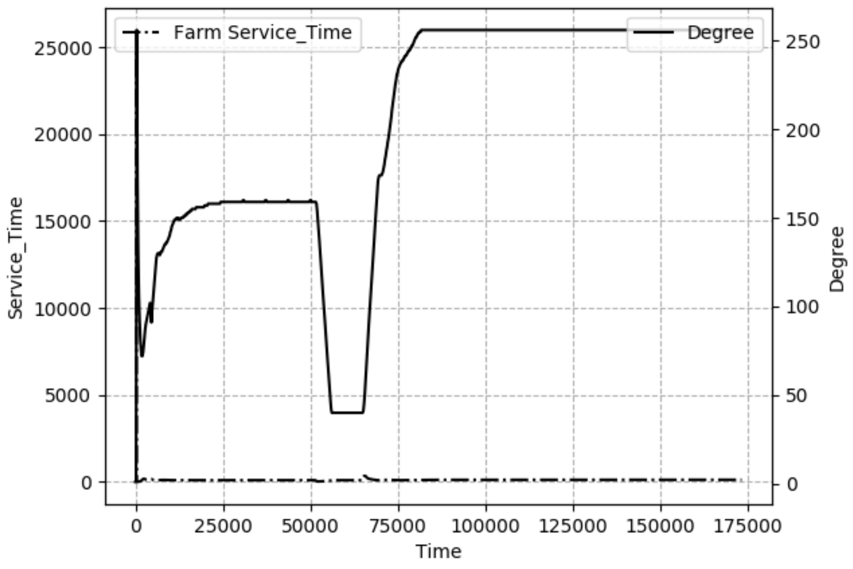

Figure 8: Concurrency Throttling - FastFlow
Ts Goal = 100 milliseconds
SMA window = 20

5.2 Spike Handling

It was also performed a test to observe how the degree change was handled in the case of a random spike. The result (Figure 9) shows that the spike does not make sudden changes to the degree. This is certainly due to the use of SMA (Section 3.2).

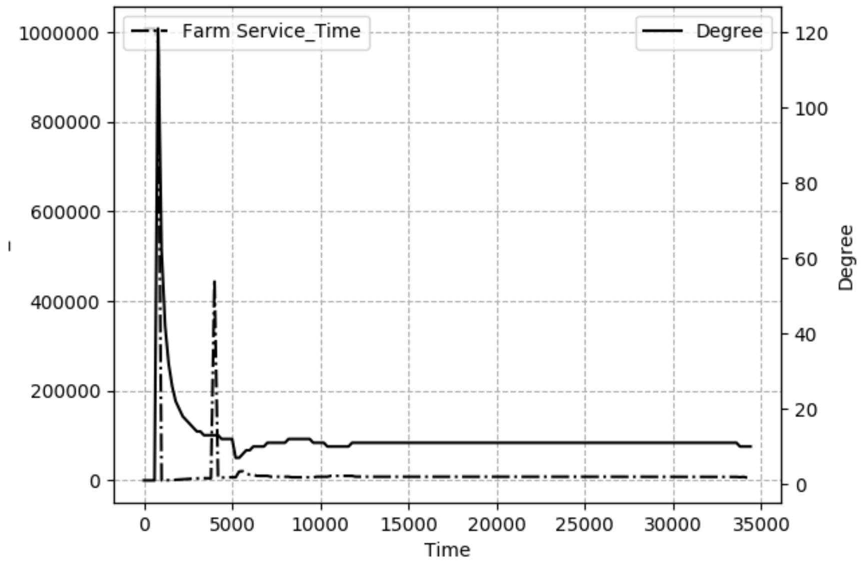

Figure 9: Spike Handling

5.3 Scalability, Speed Up, Efficiency

Finally Scalability (Figures 10), Speed Up (Figures 11) and Efficiency (Figure 12) were analized by comparing the pthread and FastFlow implementations.
From a first view it emerges that the pthread solution is comparable to the FastFlow implementation with respect to scalability and speed up until parallelism degree = 64. FastFlow starts behaving not very well in correspondence of the parallelism degrees 128 and 256; it is not exploiting hyperthreading (unlike the pthread implementation) and also decrease in performance.

From the point of view of efficiency (Figure 12), both solutions seem to behave well, despite the fact that the solution with FastFlow has a more stable trend that, however, compared to pthread, falls at the parallelism degree 128. Overlapping the speed up and efficiency graphs, for both implementations the best parallelism degree is 64, corresponding to the saturation of the physical cores (pthread 85% vs FastFlow 79%).

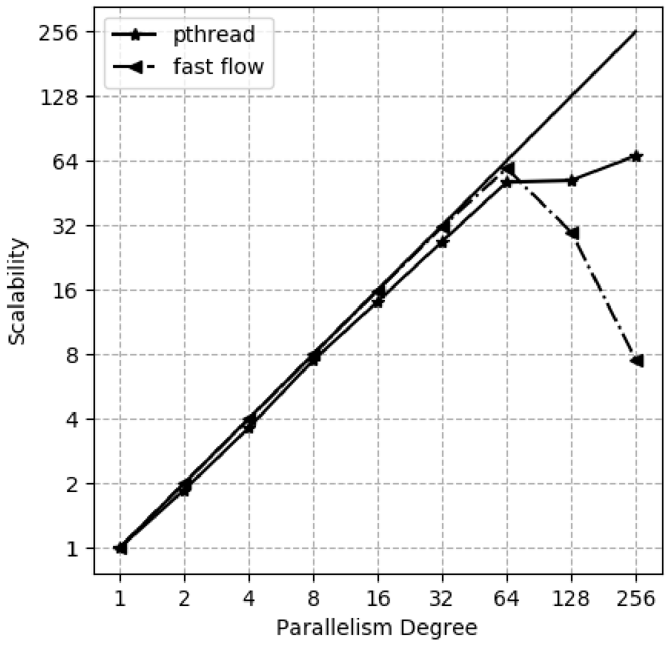

Figure 10: Scalability: pthread Vs FastFlow

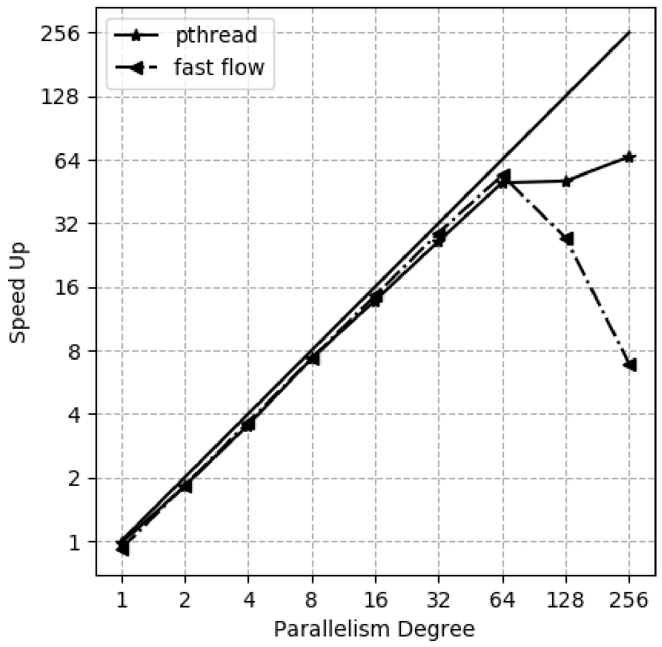

Figure 11: Speed Up: pthread Vs FastFlow

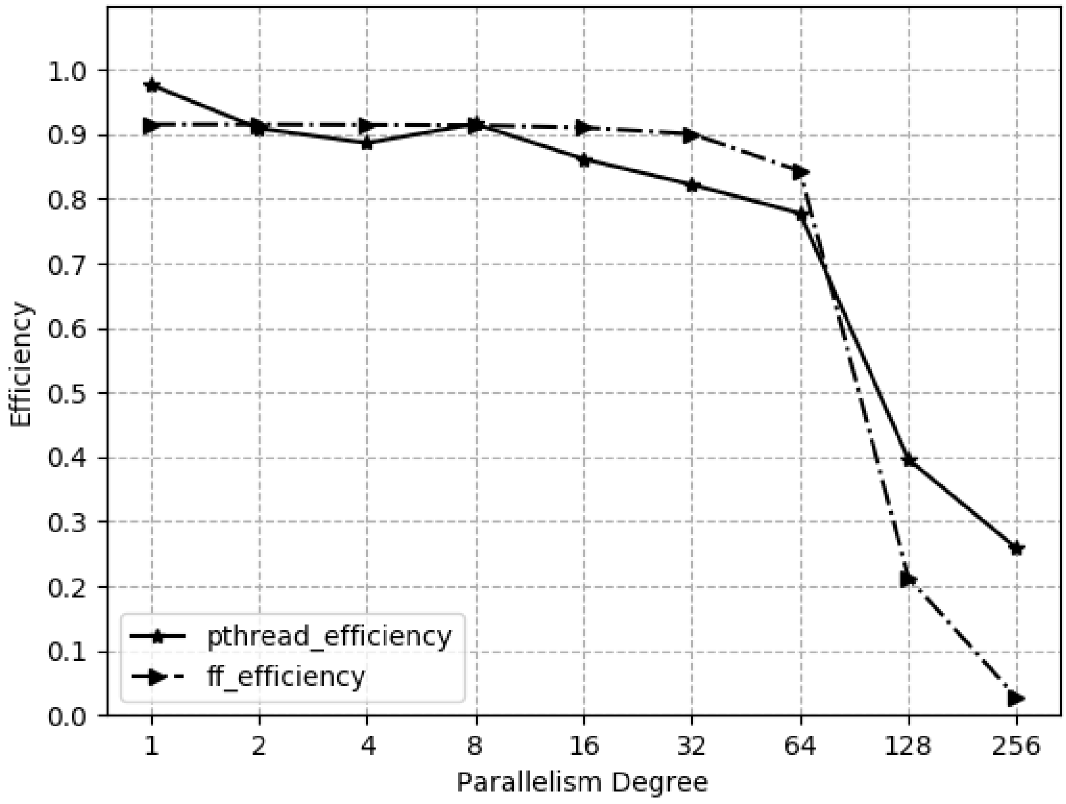

Figure 12: Efficiency: pthread Vs FastFlow