Autonomic Farm Pattern

Parallel and Distributed Systems: Paradigms and Models

Carlo Alessi

September 20, 2019

I Introduction

An autonomic computing system is a program that is capable of self-managing the allocated resources on a parallel (distributed) architecure, adjusting to the impredictability of the requests complexity, with the goal of achieving a steady desired performance.

This project consists in (i) providing an implementation of the farm stream parallel paradigm, which supports a concurrency throttling mechanism; (ii) performing experiments on a shared-memory multi-core CPU; and (iii) analyzing the results.

In section II the autonomic farm model is formalized, and the design of the parallel architecture is described. Moreover the main performance measures used in the experiments are reviewed and the expected results are reported. The relevant implementation details are also summarized. The results of the experimental validation are reported in section III. Finally the results are analyzed in section IV. Section V concludes.

II Methods

II.1 Farm pattern background

A farm system Σ is a form of stream parallelism that replicates the computation of a module on a set of n_w identical *workers*, $\{W_1, \dots W_{n_w}\}$. The farm receives in input a data stream $X = \{x_1, \dots x_m\}$ of length m, and outputs a stream $Y = \{y_1, \dots, y_m\}$ of the same size. Each worker computes a function $f : X \to Y$, producing values $y = f(x)$ for the output stream. The input stream is usually managed by the farm *emitter*, E, which distributes the items of the stream to the workers according to opportune scheduling policies. The output stream is managed by the farm *collector*, C, which collects the results elaborated by the workers and sends them to the output stream. Thus there is a E-W-C pipeline effect. Moreover in the case of an autonomic farm, the number of workers can be changed by the farm *manager*, M. Therefore the total parallelism degree of the farm pattern is $n = n_w + 3$ when emitter, collector and manager are realized as single modules.

II.2 Parallel architecture design

This section describes the design and functioning of the parallel architecture used, shown in Figure 1, considering the communication between modules, the control flow (i.e. through which phases each module evolves), and the (concurrent) data/state access.

Emitter-Worker communication The communication between emitter and workers plays a key role on the final performance. The emitter schedules the tasks to each worker and, to optimize the performance, it has to guarantee that the workload of each worker is balanced. To cope with the variability of the execution time as a function of the value of different tasks, the scheduling was implemented with the *on-demand* strategy, keeping track of the free workers[1]. The on-demand strategy was implemented in a deterministic way: the emitter (i) receives a new element from the input stream X; (ii) waits that a worker sends a message to the channel Q_{req} declaring its disponibility to accept tasks; (iii) sends the task to that worker via the channel Q_{task} relative to the same worker.

[1] If the tasks had all the same execution time, a circular round-robin assignment would have been sufficient.

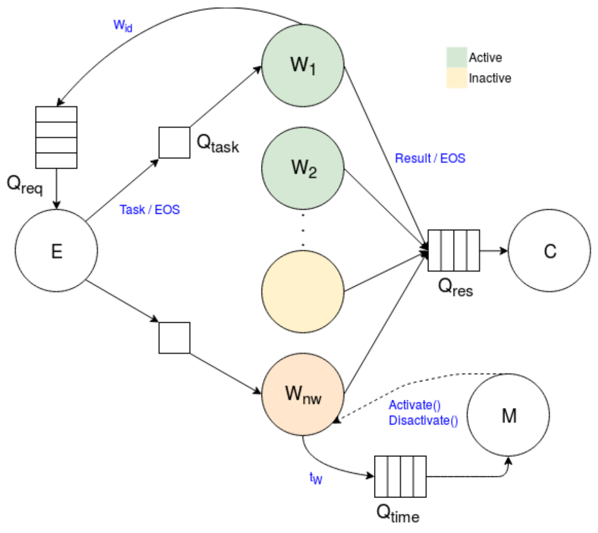


Figure 1: Autonomic farm design. Each node in the graph corresponds to a single thread of execution. The archs connecting the nodes define the interaction between modules. Continues arrows implicitly indicate *send()* and *receive()* operations via channels, and the archs are labeled with the data transferred. Discontinues arrows are labeled with functions that access the state of another module. Active and inactive workers are shown.

Worker-Collector communication The results computed by the workers are sent to a common channel Q_{res} to the collector. The collector can (i) simply function as an interface toward the output stream, or (ii) guarantee the order of the results as the respective requests (ordered farm), which is useful when the execution time of the tasks differ.

In the latter case, the emitter generates a unique identifier that is associated to the message sent.

Farm Manager & concurrency throttling The councurrency throttling mechanism is controlled by the farm manager, which specifies its own policy to update the parallelism degree in order to match the desired service time. The average execution time of the function replicated by the workers, t_W, is approximated by an exponentially weighted moving average (EWMA), defined below:

$$t_W(i) = \begin{cases} t_W(1) & , \quad i = 1 \\ \alpha t_W(i) + (1-\alpha) t_W(i-1) & , \quad i > 1 \end{cases} \tag{1}$$

where the smoothing factor $\alpha \in [0,1]$ specifies how much the system recalls the past execution timings. Higher values of α discount old values faster, whereas lower α values keep track of old execution timings for longer[2].

The farm manager receives from each worker the elapsed time of the computation, via the channel Q_{time}, and uses (1) to update the parallelism degree, as described in section II.3, by activating/disactivating the proper number of workers.

II.3 Performance model

This section first reviews the main performance measures in the context of the farm paradigm, then describes how some of this measures are used to build the concurrency throttling mechanism. The section concludes with a discussion of the expected results.

Let Σ be a farm structure with n_w workers, plus an emitter, collector and manager. The total parallelism degree, i.e. the number of modules, is $n = n_w + 3$, independently from the hardware architecture and their capacity to operate simultaneously.

Service time The service time, $T_s(n_w) = t_i - t_{i-1}$, is the average time between the deliver of two consecutive results. Assuming that computation and communication do not overlap, each individual worker has an ideal service time

$$T_s^W = t_W + L_{comm_1} + L_{comm_2} \tag{2}$$

where t_W is the average execution time of the function replicated by the worker, L_{comm_1} and L_{comm_2} are respectively the latency of the communication between emitter and worker, and worker and collector[3].

The service time of the emitter and collector can be both approximated with L_{comm}, because their internal computation is negligible with respect to the communication; therefore

$$T_s^E = T_s^C = L_{comm} \tag{3}$$

Let T_A be the inter-arrival time of a task to the farm. The optimal parallelism degree n_w^*, which

[2]With $\alpha = 0$ it will consider only the first elapsed time, with $\alpha = 1$ it will consider only the new time.
[3]Usually $t_W \gg L_{comm_i}, \quad i = 1, 2$.

guarantees the work-load balance among the workers and minimizes the service time, is given by $n_w^* = \lceil \frac{t_W}{T_A} \rceil$. Setting $n_w \leftarrow n_w^*$ and $T_s^W(n_w) \leftarrow T_A$, we obtain $T_s^W(n_w) = \frac{t_W}{n_w}$.

Therefore, considering the pipeline-like nature of the farm, the overall service time is given by

$$T_S(n_w) = \max\{T_s^E, T_s^W(n_w), T_s^C\} \tag{4}$$

Assuming that emitter and collector are not the bottleneck of the system, that is $T_A \geq L_{comm}$, which is almost always satisfied in practice, the service time of the farm can be very well approximated with

$$T_S(n_w) \simeq T_s^W(n_w) = \frac{t_W}{n_w} \tag{5}$$

Latency The latency, $L = t_1 - t_0$, is the average time needed to compute a single element of the stream. In the sequential case, the latency coincides with the execution time t_W. In the parallel case, it also has to take into account the time spent during communication, thus obtaining

$$L_{farm} \simeq t_W + L_{comm_1} + L_{comm_2} \tag{6}$$

Completion time The completion time, $T_c = t_m - t_0$, is the average time needed to compute and deliver all the elements of the stream. As in the pipeline paradigm, it is possible to distinguish between the three phases: (i) filling transient; (ii) a regime phase; (iii) emptying transient. Therefore the completion time can be approximated as[4]

$$T_c \simeq m \cdot T_s^W(n_w) \quad , m \gg n_w \tag{7}$$

Speedup Let T_{seq} be the execution time of the best sequential application. The speedup, Sp, measures how much the parallel execution is faster than the best sequential application. It takes values in the interval $[0, n]$, and is defined as

$$Sp(n_w) = \frac{T_{seq}}{T_c(n_w)} \tag{8}$$

[4]In the experiments the condition $m \gg n$ is enforced.

Efficiency The relative efficiency ϵ, measures how much the actual performance match the ideal performance. Let $T_{id}(n_w) = \frac{T_{seq}}{n_w}$ be the ideal parallel time (without considering the overheads). The efficiency is defined as

$$\epsilon(n_w) = \frac{T_{id}(n_w)}{T_c(n_w)} = \frac{Sp(n_w)}{n_w} \tag{9}$$

therefore $\epsilon(n_w) \in [0, 1]$. The efficiency is an important measure because it gives an idea of the percentage of usage of the single modules of the parallel system, and thus tells if the resources are being exploited properly.

Scalability Similarly the scalability, Sc, is a measure of how much the performance scales by adding more and more workers, with respect to using a single worker, given by

$$Sc(n_w) = \frac{T_c(1)}{T_c(n_w)} \tag{10}$$

Parallelism degree update The parallelism degree is changed by considering the approximation of the actual approximation of the service time defined in (5), $T_s(n_w) = \frac{t_W}{n_w}$. It is then possible to derive the required number of workers, by setting the actual service time equal to the desired service time as below

$$\frac{t_W}{n_w} = T_s^* \Longrightarrow n_{w_{new}} = \frac{t_W}{T_s^*} \tag{11}$$

Since $n_w \to \infty$ when $T_s^* \to 0$, and $n_w \to 0$ when $T_s^* \to \infty$, the farm manager ensures that the new number of workers is between 1 and the maximum number of workers specified, $n_{w_{max}}$, performing the following assignment

$$n_w \leftarrow \max(1, \min(n_{w_{new}}, n_{w_{max}})) \tag{12}$$

Finally, the manager activates/deactivates the proper number of workers depending on the difference between the old and new number of workers required.

The accuracy of the concurrency throttling mechanism is evaluated by computing the average relative error between the desired and actual service time, i.e.

$$e_{rel} = \frac{1}{3m} \sum_{i=1}^{3m} \frac{\hat{T}_{s_i} - T_s^*}{T_s^*} \tag{13}$$

where $3m$ is the total lenght of the task collection, which is divided in three equal-sized partitions, and $\hat{T}_{s_i}$ is the approximation of the service time of the workers computed at each step[5].

Expected results The expected performance are analyzed assuming a CPU with n_{core} physical cores, each with $n_{context}$ contexts, for a total of n_{ht} hyper-treads.

Ideally, the speedup is expected to increase more or less linearly for $n_w \in [1, n_{core}]$, because up to n_{core} threads can be executed simoultaneously. Moreover, thanks to the hyper-threading technology, the speedup can increment further for $n_w \in (n_{core}, n_{ht}]$, although with a smaller slope.

However, the amount of communication overhead may completely compromise the performance of the parallel application. Indeed, if the function to be computed by the workers is trivial, the communication overhead dominates the service time of the worker, i.e.

$$t_W \ll L_{comm_1} + L_{comm_2} \tag{14}$$

As a result the parallel application will scale poorly. Conversely, when the tasks are computationally heavy, the communication overhead becomes negligible with respect to the computation time, i.e.

$$t_W \gg L_{comm_1} + L_{comm_2} \tag{15}$$

Therefore the parallel completion time, $T_c(n_w)$, will approach the ideal parallel time $T_{id}(n_w)$, resulting in a greater speedup and efficiency. In summary, speedup and efficiency are directly proportional to the task size.

Also, speedup and efficiency are positively improved by increasing the input stream length m, because the farm stays longer in the a regime phase. The discussion above holds for the scalability as well.

The scalability plays an important role on the concurrency throttling mechanism. If the actual service time is lower than the desired service time, $T_s(n_w) < T_s^*$, the farm manager will increase n_w according to (12), but if the application has a poor scalability it may not be able to reach the desired service time, even though it was analitically and physically possible.

Another important consideration is the fact that, if the emitter or collector are the bottlenecks of the system, changing n_w does not alter the service time. Otherwise there will be a transient period where the actual service time adapts to the desired service time, more or less quickly, depending on the value of α.

[5]Equation 13 was computed also as the average of the absolute value of the relative errors.

II.4 Implementation details

Overall design The software architecture is organized as follows. The main class is the *AutonomicFarm*, which contains instances of *FarmEmitter*, *FarmCollector*, *FarmManager*, and a vector of *FarmWorker*, each implemented as an independent class.

Each of the latter classes has a *std::thread* field, and implements methods such as *run()*, *join()* and *body()*, respectively to start and wait the termination of the thread, and define the main job of the farm module.

Farm initialization The farm is instantiated by specifying the maximum number of workers, $n_{w_{max}}$, the function f replicated by the workers, the α parameter used by the farm manager, and a *concurrency_throttling* flag telling whether the concurrency throttling mechanism should be active. At the time of the AutonomicFarm constuctor, all the queues used for communication are allocated and assigned.

Farm execution The farm is executed by calling the method *run_and_wait()*, which first starts all the modules then waits for their termination, and takes as parameters the *data* collection that simulates the input stream X, the initial number of workers $n_{w_{init}}$, and the desired service time T_s^*.

Workers activation/deactivation The activation and deactivation of workers is managed using locks and condition variables, used to access in mutual exclusion the worker status by the FarmWorker and FarmManager.

Each instance of FarmWorker has a *std::mutex* and *std::condition_variable*, which are coupled with the variable *worker_status* of type *Enum*, that can take values *ACTIVE* and *INACTIVE*.

When the farm is constructed, the variables *worker_status* are set to *INACTIVE*. When the farm is started, all the threads of the workers are spawned independently of the value of $n_{w_{init}}$. However, only $n_{w_{init}}$ workers start consuming tasks, which are activated by the FarmManager by setting the value of *worker_status* to active, and notifying the corresponding thread that was waiting on the condition variable. The FarmManager keeps track of the number of active workers and knows $n_{w_{max}}$, and uses this information to access the vector of FarmWorker and to activate/deactivate the proper number of workers.

Communication channels All the communication channels are implemented with safe queues that are accessed in mutual exclusion. In particular, the communication between emitter and workers is implemented with $n_w + 1$ bounded queues. The feedback task request queue, Q_{task}, has a maximum capacity of $n_{w_{max}}$, in order to possibily obtain requests from all workers. The n_w feedforward task queues, Q_{task}, can contain only one item at a time, so that the manager can disactivate a worker without leaving tasks left undone. The queue between workers and manager, Q_{time}, and between workers and collector, Q_{res}, are both unbounded because the length of the stream is unknown at the time of the creation of the farm.

Type of messages All the messages are sent by reference (pointers) to avoid unnecessary memory copies. The messages that flow through the E-W-C pipeline are implemented via a template-based structure. The structure is composed of three fields (*data*, *task_id*, *is_EOS*). The *data* field can be the element of the input stream, or the result for the output stream; *task_id* can be used by the collector to reorder the results; *is_EOS* is a boolean variable to manage the end-of-stream.

The emitter encapsulates the data items of the input stream into a structure of type *Task*⟨*Tin*⟩, where *Tin* is the data type. Each worker receives a *Task*⟨*Tin*⟩, and if it is not an EOS, computes the function on the data obtaining a result of type *Tout*. The worker then wraps the results into a *Task*⟨*Tout*⟩ structure and sends it to the collector. The collector unwraps the results accessing the *data* field, and finally pushes it to the output stream.

The messages between workers and manager are simply pointers to *std::chrono::microseconds*.

Termination and EOS The termination phase starts when the input stream is finished. The emitter terminates after sending $n_{w_{max}}$ EOS messages to all the workers, regardless of how many workers were active at that time. When a worker receives an EOS, first it sends an EOS to the manager[6], then it sends an EOS to the collector, and terminates. When the farm manager receives an EOS from a worker it activates all the inactive workers, so that they can too read the EOS received by the emitter. The farm manager and collector terminate after receiving $n_{w_{max}}$ EOS.

Statistics collection The FarmManager collects statistics such as the execution time of the workers, and the history of the actual service time and the number of active workers. The FarmEmitter and FarmCollector collect their own execution time.

III Results

In this section the results of the experimental validation are reported. The experiments were performed on a CPU Xeon Phi KNC with 64 cores, each with 4 contexts, for a total of 256 possible hyperthreads. We referred to *hyperthreading zone* when $n_w > n_{core}$. The pattern was tested on one synthetic application that busy waits for a specified amount of l microseconds.

III.1 Task collection

The task collection used to evaluate the concurrency throttling mechanism is composed of three sub-partitions of m tasks each, i.e the total collection length is $3m$. The tasks to be computed are identical within the same partition, respectively of length $4l$, l and $8l$.

[6]The EOS to the manager is an execution time of -1.

The task collection used to evaluate the other measures such as completion time, speedup and scalability use a monolitic collection of size m, where each task has length l. For these kind of experiments, the concurrency throttling is turned off, so also the communication between workers and manager does not take place.

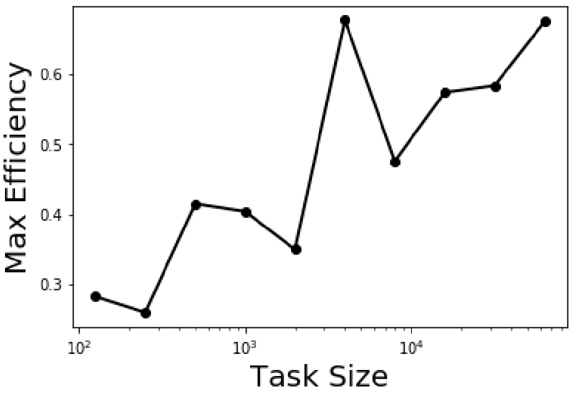


Figure 3: Maximum efficiency vs task size. Maximum efficiency $\epsilon = 0.68$ at task size $l = 4000$.

III.2 Performance measure evaluation

Task size Figure 2 and Figure 3 respectively show how the maximum obtained speedup and efficiency changes for $n_w \in [1, 256]$ and tasks size $l \in \{125, 250, 500, 1k, 2k, 4k, 8k, 16k, 32k, 64k\}$. The collection size was fixed at $m = 1000$. All the speedup curves are shown in Figure 4.

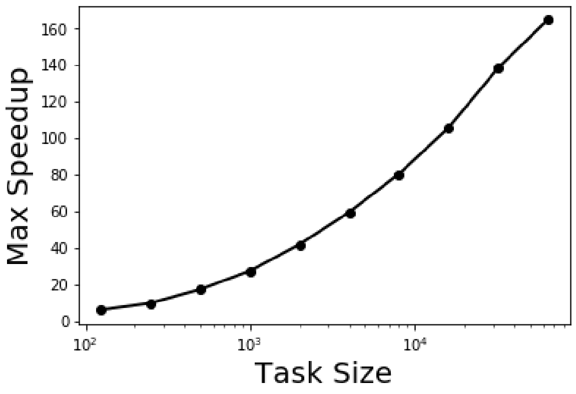


Figure 2: Maximum speedup vs task size. Maximum speedup $Sp = 164.77$ at task size $l = 64000$.

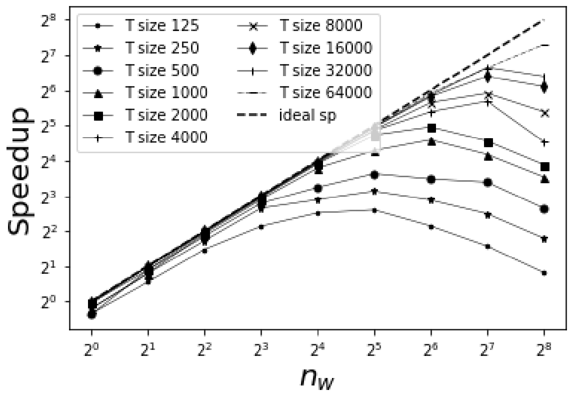


Figure 4: Speedup curves for different task sizes.

Collection size Similarly, Figure 5 and Figure 6 respectively show how the maximum obtained speedup and efficiency changes for $n_w \in [1, 256]$ and collection size $m \in \{100, 200, 400, 800, 1.6k, 3.2k, 6.4k, 12.8k, 25.6k\}$. The task size was fixed at $l = 1000$. All the spedup curves are shown in Figure 7.

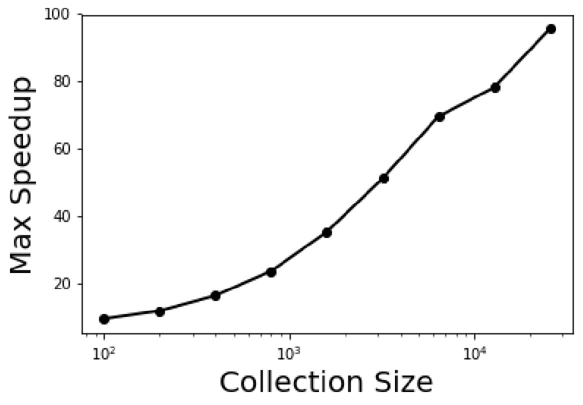


Figure 5: Maximum speedup vs collection size. Maximum speedup $Sp = 95.48$ at collection size $m = 25600$.

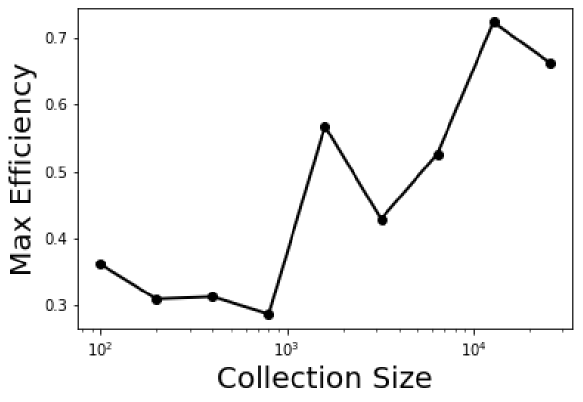


Figure 6: Maximum efficiency vs collection size. Maximum efficiency $\epsilon = 0.72$ at collection size $m = 12800$.

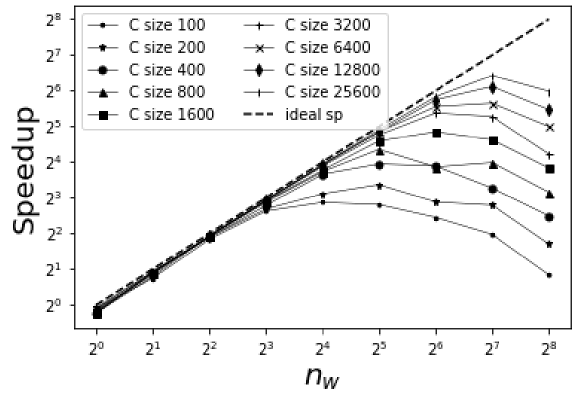


Figure 7: Speedup curves for different collection sizes.

Completion time and scalability The ideal and actual completion time as a function of the number of workers is shown in Figure 8. The scalability curve for $n_w \in [1, 256]$ is shown in Figure 9. The collection size was $m = 1000$, the task size was $l = 64k$. Results are summarized in Table 1.

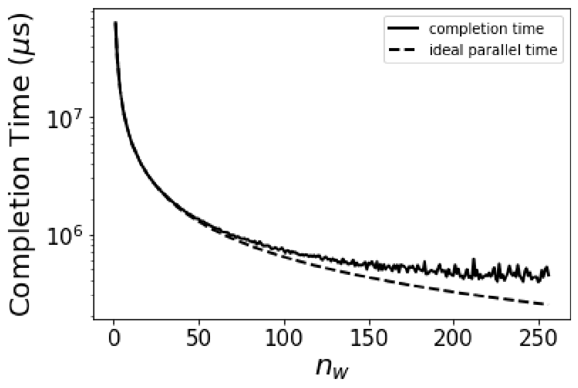


Figure 8: Completion time and ideal parallel time vs number of workers.

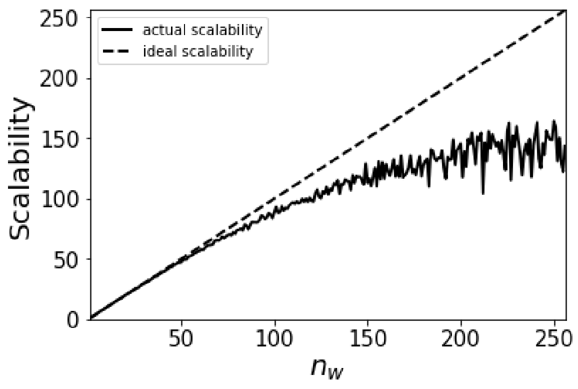


Figure 9: Scalability vs number of workers.

n_w	scalability
64	61.26
128	109.62
256	143.47

Table 1: Scalability summary. Maximum scalability $Sc = 164.33$ obtained with $n_w = 250$.

Speedup and efficiency The curves of speedup and relative efficiency for $n_w \in [1, 256]$ are shown

n_w	speedup	efficiency
64	61.08	0.95
128	109.3	0.85
256	143.06	0.56

Table 2: Speedup and efficiency summary. Maximum speedup $sp = 163.86$ with $n_w = 250$. Maximum efficiency $\epsilon = 1$ with $n_w = 4$, minimum efficiency $\epsilon = 0.48$ with $n_w = 255$.

in Figure 10 and Figure 11. The collection size was $m = 1000$, the task size was $l = 64k$. Results are summarized in Table 2.

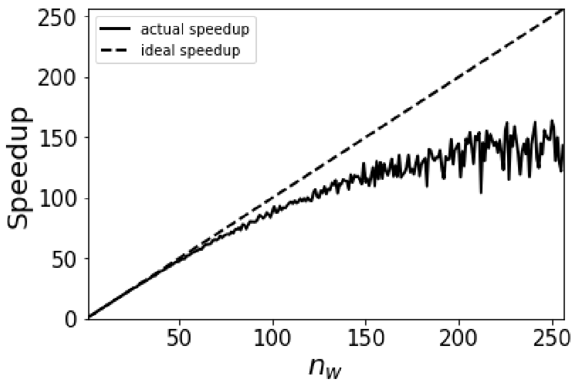


Figure 10: Speedup vs number of workers.

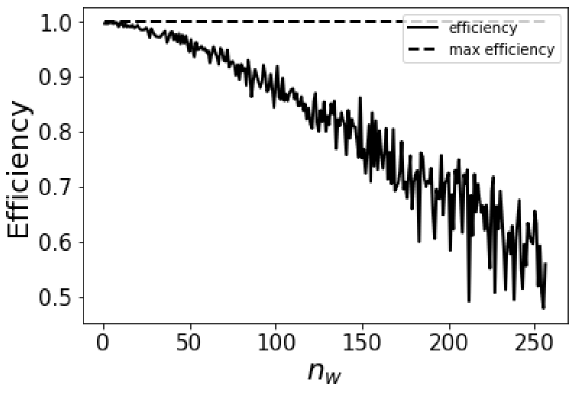


Figure 11: Efficiency vs number of workers.

III.3 Concurrency throttling evaluation

Service time history In Figure 12 it is shown how the actual service time and paralellism degree was adjusted by the farm manager in order to meet the desired service time. The experiments were performed for $l \in \{1000, 2000, 3000\}$, $T_s^* = 200$ microseconds, $\alpha = 0.05$, and a collection partition size of length $m = 500$[7].

For the three values of l the average absolute service time error was respectively 0.04, 0.051 and 0.028.

Alpha parameter The average relative errors between actual and desired service time for $\alpha \in [0, 1]$, $l = 1000$, and $m = 500$ is shown in Figure 13.

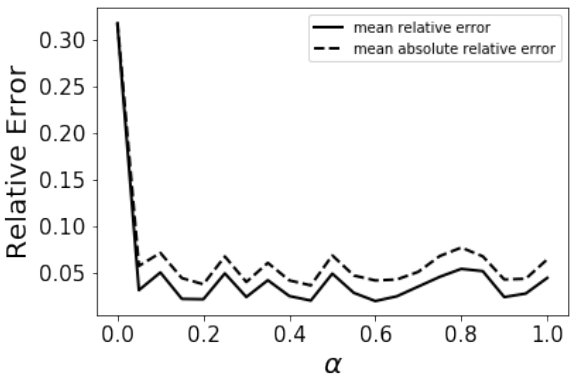


Figure 13: Relative service time error vs alpha.

The parallelism degree history for three runs with $\alpha \in \{0.25, 0.5, 0.75\}$ are shown in Figure 14. Each partition size was $m = 500$, with task length $l = 1000$ and $T_s^* = 200$.

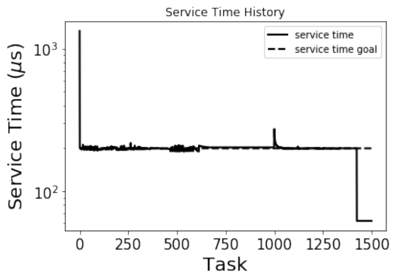
For the three α values, the average absolute service time error was 0.056, 0.054, 0.146.

IV Discussion

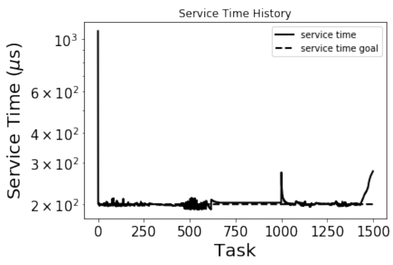
The first observation is that the synthetic application does not deal with large data structures, so there are not possible cache problems.

From Figure 2 it is clear to see that increasing the task size results in obtaining a greater maximum speedup. This is because when the tasks

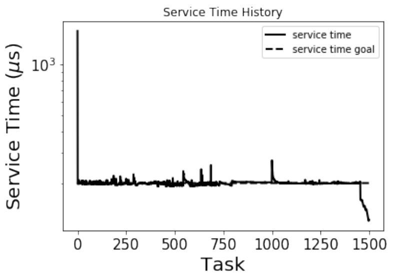
[7]The total length of the collection was $3m = 1500$.



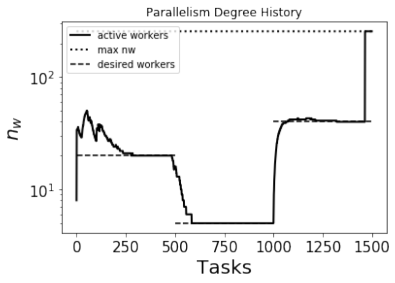
(a) $l = 1000$



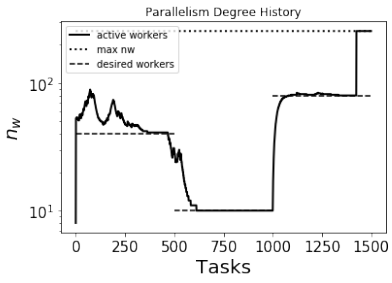
(b) $l = 2000$



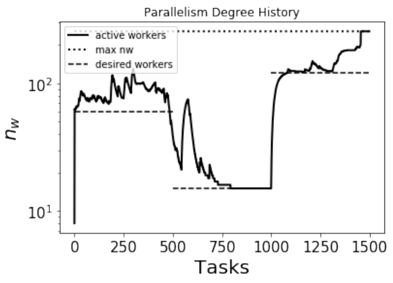
(c) $l = 3000$



(d) $l = 1000$

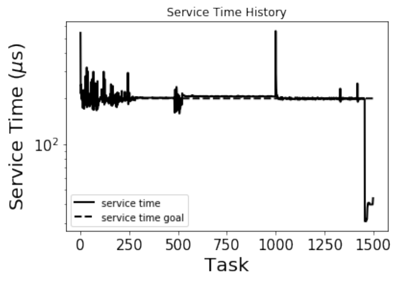


(e) $l = 2000$

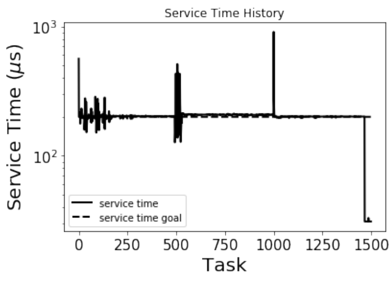


(f) $l = 3000$

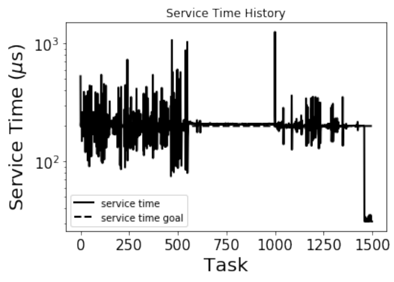
Figure 12: Service time history (top) and parallelism degree history (bottom) for different values of l.



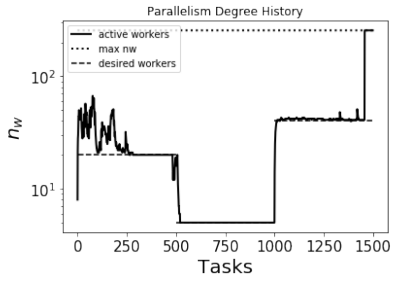
(a) $\alpha = 0.25$



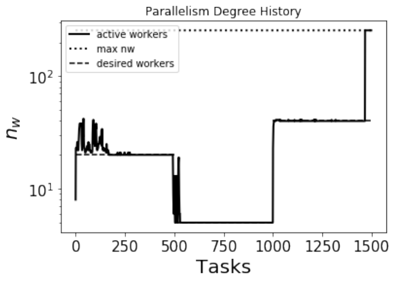
(b) $\alpha = 0.50$



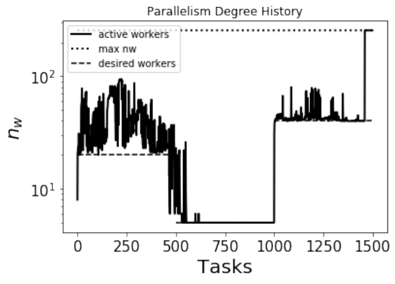
(c) $\alpha = 0.75$



(d) $\alpha = 0.25$



(e) $\alpha = 0.50$



(f) $\alpha = 0.75$

Figure 14: Service time history (top) and parallelism degree history (bottom) for different values of α.

are more computationally heavy, the overheads for communication and hyper-threads rescheduling become more and more negligible with respect to the computation time. As a result $T_c(n_w)$ approaches $T_{id}(n_w)$, therefore $Sp(n_w)$ approaches the ideal speedup.

Figure 5 confirms that the maximum speedup also increases as the collection size increases, because the a regime phase becomes longer as a result of better enforcing the condition $m \gg n_w$ in (7).

Observe that for higher task and collection sizes, the peak of the speedup curve is reached for higher n_w. Regarding the collection size, Figure 5, for $m = 800$ the peak is reached at $n_w = 2^5 = 32$ (no hyperthreading); for $m = 3200$ the peak is reached at $n_w = 2^6 = 64$ [8]; for $m = 6400$ onward the peak is reached at $n_w = 2^7 = 128$ (hyperthreading zone).

A similar observation applies for the task size in Figure 2, except that for $l = 64000$ the speedup did not even reach the peak.

In Figure 8 it is clear to see that the completion time well follows the trend of the ideal completion time until $n_w \sim 50$, then the gap starts to increase due to the communication overhead and the hyperthreading zone. This corresponds to a slower increment of the scalability curve in Figure 9.

From Figure 11, it is achieved a very good efficiency for $n_w \leq 64$, good efficiency for $n_w \in (64, 128]$ and relatively bad efficiency for higher n_w. The relationship between efficiency with task and collection size is not clear as shown in Figure 3 and Figure 6.

For what concerns the concurrency throttling mechanism, from Figure 13 it is clear to see that the value of α plays a role in the average service time error, but also that for any value $\alpha > 0$, the difference is not substantial[9].

For the service time and parallelism degree history in Figure 12, it is observed that when the lenght of the tasks of different collection partions differ a lot, the parallelism degree takes more time to adjust, for fixed values of α.

In particular notice that for $l = 3000$, during at the beginning of the second partition, the parallelism degree does not monotonically decrease to the new desired value. This is because the last tasks of the previous partition took longer to finish and, their latency still influenced the farm manager. By the end of the second partition, the parallelism degree completely adjusted to the desired value. In the third partition, notice how the parallelism degree tends to increase above the desired value. This is because for $l = 3000$ and $T_s^* = 200$, the desired parallelism degree is $n_w^* = 150$. Since n_w^* lies in the hyperthreading zone, and the application may scale not perfectly, the approximation of the service time performed by the manager is higher than expected, resulting in a higher number of workers being activated.

Finally, Figure 14 shows how different values of α play a role in how smoothly and quickly the parallelism degree and service time adjust to the desired values.

V Conclusion

The aim of this project was to provide an implementation of the autonomic farm pattern, and perform and analyze its perfromance on a benchmark application. The main limitation of the proposed solution is the wide usage of lock-based queues as communication channels. A possible improvement could be to use lock-free queues as proposed in [1, 2], especially in the unbounded channels between workers with manager and collector.

References

[1] Marco Aldinucci, Marco Danelutto, Peter Kilpatrick, Massimiliano Meneghin, and Massimo Torquati. An efficient unbounded lock-free queue for multi-core systems. In *European Conference on Parallel Processing*, pages 662–673. Springer, 2012.

[2] John D Valois. Implementing lock-free queues. In *Proceedings of the seventh international conference on Parallel and Distributed Computing Systems*, pages 64–69, 1994.

[8] Already slightly in hyperthreading since the machine has 64 cores, and 3 threads are wasted by the emitter, manager and collector

[9] For $\alpha = 0$ only the first time t_W is considered, so the parallelism degree never changes.