# SPM Project Report
Parallelized online training of Echo State Networks

Valerio Mariani, 560014

**Abstract**

Objective of this project is to give some parallel implementations of the Recursive Least Square Algorithm (RLS) [2] for Echo State Networks (ESN) [3].
The main reason for implementation is that I want the online training algorithm to work as fast as possible because the application I want to use it for - namely financial prediction with adaptive filtering - requires filter adaptation to be as fast as possible to allow for proper exploitation of the given prediction.
In this report I will provide description of 4 different parallel implementations of the RLS algorithm, of which one uses only c++ threads and classical instruments provided by c++ language, while the other three use a library called fastflow [1].
In this report, I will first give a synthetic description of the problem at hand in section 1 - mainly presenting the algorithm to be parallelised - then I will go through the reasoning I did to get to the architectures I made and review their theoretical properties in section 2, and finally I will expose the results of the tests I made about the improvement in computation performances in section 3. Code for this project can be found here

# Contents

# 1 Introduction

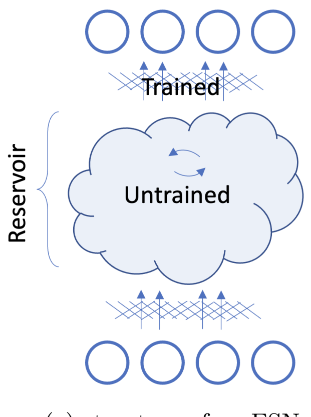
## 1.1 Problem description

An Echo State Network (ESN) [3] is a randomized, recurrent neural network. A recurrent neural network is a neural network that is provided with an internal state, giving it an intrinsic predisposition to process temporal sequences. The ESN extends the concept of recurrent neural network in two main ways: 1) It can be seen as made out of two components, a dynamical system called Reservoir and a (generally linear) readout attached to the reservoir, see Figure 1a. The Reservoir serves to encode the history of the inputs he received in its state (a multidimensional vector) while the readout serves to project vectors from the Reservoir's state space onto the output space (this basically means that it reads the state of the reservoir and uses it to spit out an output, that can e.g. be a prediction or a classification). 2) its Reservoir is generally initialized in a (semi-)randomic manner and is not trained, allowing us to call the ESN a **randomised** recurrent network, and to only train its readout part.

Now, the readout part of the network is generally trained only once by a direct (not iterative) closed-formula method, and that can be seen as the classical, offline training algorithm for this type of network - offline meaning that the training set is used altogether during each training iteration - although, the concern here is to use an iterative, **online** training algorithm, that better adapts to the task of predicting very long temporal sequences (e.g. stock market prices). In particular, online training differs from offline training in that, at each iteration, a single training sample (element of the training set) is provided to the network and readout is slightly changed in order to improve its prediction capabilities.

The task taken into consideration here is to train the network (its readout) to try and predict prices of the stock market one minute ahead using past price values, and this is supposed to be done by means of an online training algorithm called Recursive Least Square (RLS) [2] (or rather a numerically stable version called Symmetric Recursive Least Squares). From a very high-level point of view, we have a stream of price-values, and the main routine looks like this:
1)submit an element to the network and compute new state and prediction
2)compute the error committed
3)update the parameters of the network.
Where point 3) is the part where the RLS algorithm is used to update the parameters of the readout.

The RLS algorithm is described in detail in algorithm 1, it is taken from [4] but notation is re-adapted to fit the discussion. The first three lines of the algorithm update the state of the network using last state and current input and deposit it into the vector $x$. Line 4 computes the output of the network by multiplying the state vector $x$ by the matrix that represents the linear readout and deposits the result into the vector $y$, then there is some computation needed by the algorithm and finally line 9 computes the new entries in the readout. Note that the algorithm needs to keep updated another $N_r$x$N_r$ matrix called P (inverse of correlation matrix of the state vector), where $N_r$ is the number of neurons in the reservoir. As for **computational complexity**: number of flops to be done is dominated by the calculations in line 10, in which the algorithm recomputes every one of the $(N_r)^2$ entries in matrix P.



(a) structure of an ESN

**Algorithm 1:** RLS algorithm on one element of the stream

```
   Input:
   $u$: input to the network
   $d$: desired output
   $N_r$: number of neurons in the reservoir
1  Procedure RLS($u$, $d$, $N_r$):
2      $x^{rec} = W\ x$ // dot product ($N_r$x$N_r$) x ($N_r$x1)
3      $x^{in} = W^{in}\ u$ // dot product ($N_r$x4) x (4x1)
4      $x = tanh(x^{rec} + x^{in})$ // elemntwise tanh of ($N_r$x1)
5      $y = W^{out}\ x$ // dot product (4x$N_r$) x ($N_r$x1)
6      $z = P\ x$ // dot product ($N_r$x$N_r$) x ($N_r$x1)
7      $k_{den} = x^t\ z + l$ // dot product (1x$N_r$) x ($N_r$x1)
8      $k = z/k_{den}$ // elementwise division of ($N_r$x1)
9      $W^{out}_{ij} = W^{out}_{ij} + (d_i - y_i)k_j$ for i=1...4, j=1...$N_r$
10     $P_{ij} = (P_{ij} - k_i z_j)1/l$ for i=1...Nr, j=1...Nr
```

(b) RLS algorithm. full listing in 1

## 1.2 Problem analysis

The following analysis works on two levels: at an higher level I had to work on the organization of macro instructions in order to be able to execute them in parallel; then I had to analyze the macro-instructions one by one to insert some parallelism degree into the macro-instructions themselves. For simplicity I start from analysis of single macro instructions.

As stated in algorithm 1, there are three main types of macro instructions that have to be executed, each of which can be executed in embarrassingly parallel way:
- Matrix-dot-vector instructions
- element-wise algebraic operations on vectors
- element-wise algebraic operations on matrices
I have looked at Matrix-dot-vector instructions as composed by a sequence of independent vector-dot-vector operations. Of the two vectors read by each vector-dot-vector operation (vdv), one of them is the same for every vdv. This can facilitate parallelism because half of the total space read by a single vdv can be kept in memory by all physical threads working on the macro instruction, but it has to be said that cooperation can at the same time be impaired by the fact that each vdv consumes memory to fit at least two times the length of the vectors multiplied.

Then there are element-wise basic algebraic operations on the entries of a matrix/vector (see lines 4,8,9,10 from algorithm 1). These can be computed in parallel, but cooperation of the threads can be partially impaired by the low "calculation-per-entry" rate.

As for organization of the sequence of macro-instructions to be executed, I arranged the lines in algorithm 1 to come up with the work-flow in figure 2, where I indicated macro-instructions with circles and their arguments with squares.

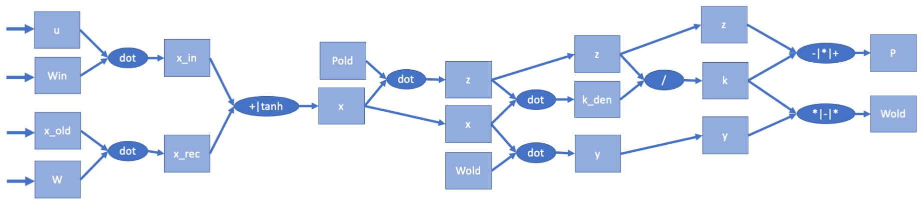


Figure 2: Graphical explanation of the work-flow for an iteration of training

Figure 2 shows that some macro-instructions can be executed in parallel, allowing for better cooperation of the threads involved. In more, it can be used to get a work-span model for estimation of upper bounds to the speedup (estimations that I did not really find to be of interest in this particular case).

It is important to stress that workflow in figure 2 is only one possible arrangement of the macro-instructions in algorithm 1, and that others are possible, indeed I will be letting the fastflow Run Time System decide the order itself in the final model proposed, see section 2.

Details on how I parallelized macro-instructions and organized the order of operations in practice can be found in section 2

## 2 Models proposed and Implementation details

### 2.1 Thought process

As stated in section 1, all macro-instructions in algorithm 1 can be executed in embarrassingly parallel way. This led me to the conclusion that each of them can be efficiently computed by some parallel map function, so the most simple yet conceptually less efficient solution I came to was a spurious sequence of parallel for instructions, like the one described in section 2.4. Soon, i realized that this type of solution had at least some communication problems as each parallel for has to wait for termination of the precedent one to start working as, again, can be seen in figure 3.

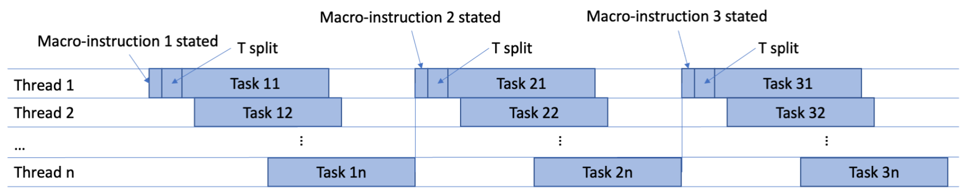


Figure 3: sketch of evolution of computation with parfor model, time goes left to right, Task1n is n-th task of macro instruction 1

So I started thinking of some way to improve load balancing and introduce some better work stealing policy. I realized that I had to mix execution of tasks from different map operations in order not to leave workers idle and waiting for each other to finish their tasks, and after some thought I realized that one simple way to do it was to abandon the idea that after launching a map operation the caller thread had to synchronously wait for completion of all the tasks, so i decided to create a thread pool class and to give it an asynchronous map method, in such a way that I could possibly submit multiple map operations to the thread pool, and then wait for the termination of the map operations altogether, like shown in figure 4. This way, if I notice that two or more macro-instructions can be executed in parallel, I can submit both to the thread pool and then possibly impose a synchronization barrier to wait until completion of all of the maps i submitted to the thread pool.

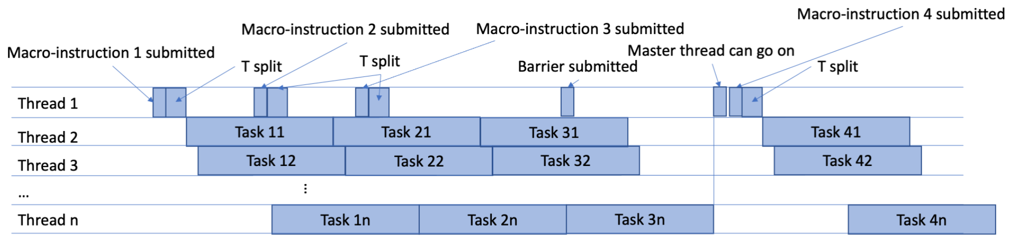


Figure 4: sketch of evolution of computation with pool+map model, time goes left to right, Task1n is n-th task of macro instruction 1

Even after implementing this solution though, I was not satisfied with the work because there still was some useless synchronization overhead due to waiting on the barriers, as can be seen e.g. in figure 4 between completion of tasks from macro-instruction 3 and beginning of execution of macro-instruction 4. At this point I started thinking of some way to mix my implementations of map functions with a Macro Data Flow (MDF) model and I came to the solution that, a-priori, I thought was my best guess: with the architecture described in section 2.6 I use a combination of a thread pool equipped with synchronous map function and an MDF run time system with three threads. This way each thread given to the MDF can submit maps to the thread pool and then wait for termination of the tasks he submitted without caring about global barriers. In other words, I wanted the MDF model to orchestrate my thread pool and work like a load balancer, like shown in figure 6.

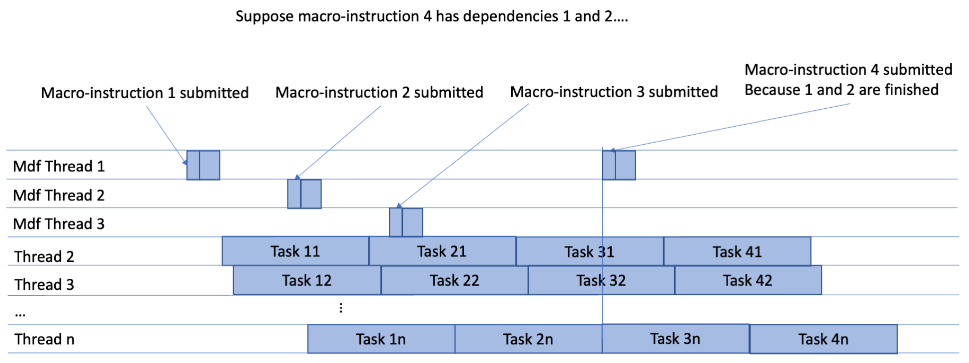

Suppose macro-instruction 4 has dependencies 1 and 2....
Macro-instruction 1 submitted
Macro-instruction 2 submitted
Macro-instruction 3 submitted
Macro-instruction 4 submitted
Because 1 and 2 are finished
Mdf Thread 1
Mdf Thread 2
Mdf Thread 3
Thread 2
Thread 3
...
Thread n
Task 11
Task 21
Task 31
Task 41
Task 12
Task 22
Task 32
Task 42
Task 1n
Task 2n
Task 3n
Task 4n

Figure 5: sketch of evolution of computation with MDF+pool+map model, time goes left to right, Task1n is n-th task of macro instruction 1

Finally, some thoughts on the map functions: tasks from the map functions deposit results into the entries of a vector, and this can let the problem of false sharing arise. To avoid this problem, when the map function splits the macro instruction into tasks, it does it in such a way that a single task works on a slice of the resulting vector that is long at least enough to fit an entire line of shared cache. This way, if processor $i$ is working on task $j$ to deposit results into a slice of a vector that goes from index $k$ to index $k + offset$, I can be quite sure that entries to be written will be stored on a line of shared cache that is written by no other threads concurrently, possibly avoiding false sharing. This is also to improve locality, allow for vectorization (that I don't think would have been easy if I were using other techniques), and to be sure that tasks are not too small (that would have caused some sync problems).

What follows in the next subsections is an overview of the architectures I developed to implement the ideas discussed up to here.

## 2.2 The Task class

As before mentioned, the map functions I wrote divide embarrassingly parallelizable macro-instructions into tasks and defer the computation of the task to some parallel thread of execution. In practice, I wrote a Task class that provides:
- member objects to remember whether a task has been executed or not in a protected way - a virtual execute method that has to be written to implement the business logic of the task, this should operate on member objects of child classes.
Then, what one needs to do to use this class is to write a derived class, implement the business logic in the execute method, and save arguments to the execute method through the constructor.

## 2.3 Model 1: thread pool + map

This implementation uses just the standard tools given from standard c++ language, in particular threads and classical synchronization methods like mutexes, locks and condition variables. Here I brought a thread pool class (pool) with associated map function.

Instances of the pool class posses a given number of worker threads each with a protected queue of tasks. For the protected queue I took an implementation given in the PDS course and slightly modified it to better fit the purpose, while for worker threads I built a very simple function to implement a consumer thread that just pops from his queue of tasks and executes what he gets.

The pool class exposes two fundamental methods:
- A submit method to asynchronously assign tasks to the worker threads,
- A barrier method to wait for termination of all submitted tasks.
Tasks are submitted to the workers following round robin scheduling policy, while the barrier method waits for the queue to become empty.

In more, the pool class exposes the map function, that takes the extremes of an interval (indices to iterate), the type of task to be executed and its non-indices arguments, then divides the given interval into sub-intervals, and for each sub-interval creates a task object with given arguments plus the indices that the task instance must iterate through. Finally, it submits the tasks created to the worker threads.

Given this implementation, I was able to compute the DAG in figure 2 following that same order of execution (macro-instructions on the same column are executed in parallel) but still have the synchronization problems depicted in figure 4 due to the synchronization barriers.

### 2.4 Model 2: fastflow parallel_for

This is the implementation of the first idea I got: A simple sequence of map operations through parallel for loops. In this case I just had to instantiate an ff::ParallelFor object and use it to compute the macro-instructions sequentially. Situation here is depicted in figure 3.

### 2.5 model 3: thread pool made with ff + map

For this model I implemented a thread pool class with fastflow, it can be found under the name of ff_pool together with its implementation of the parallel map function.

For this implementation I designed the pool in such a way that it encapsulates an ff:ff_Farm object with custom emitter. This time it is the emitter that pops tasks from a protected queue, then he sends them out to the worker thread that is going to execute them and mark them as terminated.

Overall, the situation here is the same as in subsection 2.3, except that the thread pool is made with fastflow.

### 2.6 Model 4: MDF + thread pool made with ff + map

This is the final, most complex model I brought. In this case I wanted an MDF run time system to orchestrate a thread pool, in order to work like some sort of very complex load balancer. To do it, I made up the taskgen function of the ff::ff_mdf in such a way that tasks posted in the DAG require the MDF's threads to submit map operations to a thread pool. This architecture is depicted in figure **??** and the expected behaviour is depicted in figure 6.

As a side-note, to build this model I had to change the logic of the map method and turn it synchronous (it has to be synchronous because MDF's threads must block until execution of their macro-instruction is completed for the MDF to work properly), so this time we have no synchronization barriers, and we have instead a synchronous submit method that waits for termination of all the tasks he submitted instead of waiting for the tasks queue to be empty.

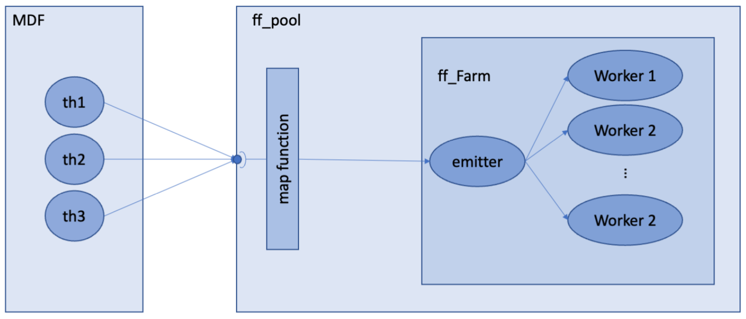


Figure 6: architecture of the MDF+pool+map model

## 2.7 Theoretical overview of the proposed models

First thing first, I have been seeing a thread pool as kind of a wrapper around a farm object since the beginning, fact being clear when I literally implement the ff_pool class as an encapsulated ff_farm object. This consideration comes particularly useful when the time arrives to model the performances of the pool. We know that service time of a farm object employing $n_w$ workers - each of them computing a task in time $T_w$ - can be approximated using equation 2 [PDS notes, page 51]:

$$T_s(farm(W)) = max\{t_e, t_c, t = \frac{T_w}{n_w}\}, \tag{1}$$

with $t_e$ and $t_c$ being communication times for emission of the task and collection of the result.I think this says a lot about the results I got from experiments, in particular about the importance of communication channels: once the worker threads have become enough, service time of the thread pool (farm) depends a lot on the communication overhead due to the synchronization between emitter and workers, in particular on latency needed to push a task onto the worker's queue and then pop it. This becomes very evident when comparing my pool, in which worker threads use blocking methods from locks, mutexes and condition variables, with the one made with fastflow, in which worker threads use spinwaiting uSPSC queues instead. This also tells why it is important to use separate queues for each worker instead of a single, global queue, exposing the greatest weakness in the ff_pool class: the fact that emitter works by its own slow queue, that results to be quite a bottleneck. Probably some better way to implement the solution could have been to expose an interface directly from the emitter object instead of encapsulating the whole farm object.

As for the MDF idea, I think the intuition was somehow in the right direction but it could definitely be implemented in a number of better ways:
- The bad news is that the original purpose of the MDF model is for implementing algorithmic skeletons, not for making them run (even if I really think there must be some ways to build up this kind of achitecture - or at least something that works on the same chords - in somewhat of an efficient way).
- In my defence although, for this application I am using a lot of maps, and those produce lots of tasks of the same "size", so technically the static, round robin scheduling policy is OK for the macro-instructions per-se. My aim here was at finding some type of "higher level" load balancing policy to be able to reduce synchronization overhead between a map operation and the other, and that is the reason I came up with the MDF+pool architecture.

This being said, an approximation of the optimal time that can be achieved from these type of solutions can be given based on the time needed to split a macro-instruction into tasks and submit the tasks to the workers plus the time needed for the workers to complete the tasks (Service Time).
I am speaking about approximations because 1) the tasks in which the various macro-instructions are divided into are not all of the same exact size, even though they are at least of same complexity, so what I can do is I can get a worst-case time-to-compute-a-task 2) I have been working on modelling architectures that mask communication overheads, so if that was to work perfectly I could technically take away the $t_e$ term in equation 2 for everyone but the first task submitted. Using points 1) and 2) I can obtain the following approximations.
I already stated a formula for the Service time $T_s$ of the tread pool (farm) in equation 2, but now, to take into consideration the different sizes of the tasks, let us take $T_w^{max}$ - instead of $T_w$ - to be **maximum** time that a single worker spends to compute a task, and let us eliminate the communication time from service time computation since, in the optimal case, it is masked like in figure 6 except for the first submission. then, equation 2 becomes:

$$T_s(farm(W)) \leq \frac{T_w^{max}}{n_w}, \tag{2}$$

assuming -as before- that we have a certain number of workers $n_w$.
Then, given the 9 macro-instructions listed in algorithm 1 (all maps), if we suppose for simplicity (I also think w.l.o.g) that they are divided into $n_w$ tasks, we will have a total of 9 $n_w$ tasks, so we can get a lower bound for the optimal time for an iteration using:

$$T_{it}^{opt} \leq t_e + (9\ n_w\ \frac{T_w^{max}}{n_w}) \tag{3}$$

that is, maximum service time times total number of tasks to be done plus time for splitting and submitting the tasks (only counted once, like in figure 6).

## 3 Experiments

### 3.1 Experimental setup

To evaluate the performances of the architectures I built with respect to the performance of the sequential version, I followed the following steps:

- I created an ESN with a reservoir of $N_r$=2000 neurons,
- I trained it with the sequential version of the RLS algorithm for $n_trials$=5 times, each time doing $n_{samples}$=100 iterations, and computed the mean training time,
- then, for each architecture, i trained the same network for the same $n_{trials}$=5 times, doing again $n_{samples}$=100 iterations per training, every time with increasing parallelism degree, up to a maximum of $max\_par\_degree$=100 workers (note that when training with the mdf+pool architecture described in subsection 2.6, I have always assigned three workers to the MDF run-time-system) and for each setup saved mean training time.

At the end, I dumped all the time-results i got into .txt files, saved them into my local machine and used a python notebook to derive the desired metrics (speedup, scalability, efficiency) and plot the results.

I have said before I am interested in time for an iteration, but here I took mean training time to evaluate the performances because I thought it could be more stable of a measurement, while still being representative of time for an iteration (If training in general is faster, and it is done by consecutive iterations of the same exact procedure, then an iteration of training must be faster in turn).

**Algorithm 1:** RLS algorithm on one element of the stream

```
Input:
$u$: input to the network
$d$: desired output
$N_r$: number of neurons in the reservoir
1 Procedure RLS($u$, $d$, $N_r$):
2     $x^{rec} = W\ x$ // dot product ($N_r$x$N_r$) x ($N_r$x1)
3     $x^{in} = W^{in}\ u$ // dot product ($N_r$x4) x (4x1)
4     $x = tanh(x^{rec} + x^{in})$ // elementwise tanh of ($N_r$x1)
5     $y = W^{out}\ x$ // dot product (4x$N_r$) x ($N_r$x1)
6     $z = P\ x$ // dot product ($N_r$x$N_r$) x ($N_r$x1)
7     $k_{den} = x^t\ z + l$ // dot product (1x$N_r$) x ($N_r$x1)
8     $k = z/k_{den}$ // elementwise division of ($N_r$x1)
9     $W^{out}_{ij} = W^{out}_{ij} + (d_i - y_i)k_j$ for i=1...4, j=1...$N_r$
10    $P_{ij} = (P_{ij} - k_i z_j)1/l$ for i=1...Nr, j=1...Nr
```

## 3.2 Results

Results of the experiments I run are plotted in figure 7.

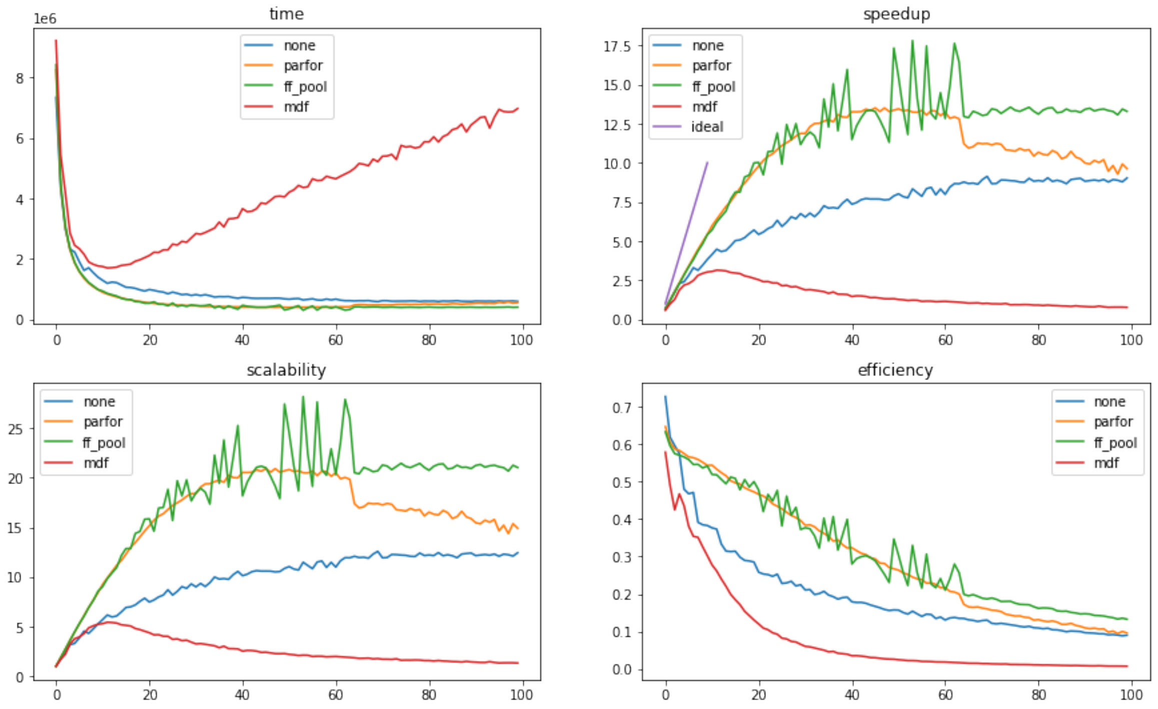


Figure 7: Results of the experiments. Parallelism degree on the horizontal axes. In the legend we have: none for thread pool with c++ thread, parfor for ff::ParallelFor, ff_pool for thread pool made with ff::ff_Farm, mdf for ff::ff_mdf + thread pool made with ff::ff_Farm

Results from figure 7 are quite clear in indicating that:

- the MDF+pool architecture described in subsection 2.6 revealed itself to be quite a failure in practice, I do not really have a clue of the reason.
- apart from that, fastflow library did not fail the expectations of resulting faster than my implementation with c++ threads. In general, the best results were achieved by the parfor and ff_pool methods.
- As can be seen e.g. from the speedup graph in figure 7, the parfor method had a sharp decline in performance when working with more than 60 worker threads, while the ff_pool method showed quite an unstable behaviour up to the the same 60 worker threads.
- efficiency of the "none" method seems to decline at an exponential rate, while - apart from the "mdf" method - efficiency of the methods that use the fastflow library declines at a closer-to-linear rate.

In more, I computed that:

- the parfor method got its maximum speedup value of 13.50 when working with 45 worker threads, achieving a mean speedup of 13.35 when working with a number of worker threads between 40 and 50.
- the ff_pool method got its maximum speedup value of 17.80 when working with 53 worker threads (note that this value is quite an outlier, as can be seen from the speedup graph in figure 7), achieving a mean speedup of 13.33 when working with a number of worker threads between 80 and 100.

## 4 Conclusions

For this project I brought 4 parallel implementations of the Recursive Least Square (RLS) online training algorithm for Echo State Networks (ESN). Of these, one uses standard instruments from c++ programming language, while the other three use a library for parallel computation known as FastFlow.
The simplest of the implementations is the one made with ff::ParallelFor, that just uses a sequence of parallel map computations, one after the other. Then there are two versions based on the concept of thread pool, and those rely on the flexibility of operations-scheduling on this type of architecture, that allows to mix execution of tasks from different macro-instructions in order to partially mask the time needed for synchronization between completion of a macro-instruction and beginning of execution of the consecutive one. Of those two versions that work on thread pools, one is built using c++ threads while the other works on top of fastflow farm. Last but not least, we have an implementation that uses fastflow's macro-data-flow run-time-system to orchestrate the thread pool built with fastflow's farm in order to further reduce the synchronization overhead.
Of the proposed architectures, the one using the mdf rts did not perform as expected, while the others obtained some nice performance-improvement results, the ones built with fastflow in particular.
As for possible future works, I have noticed that, first of all, there is much space for improvement in terms of communication channels, as I came up with a standard blocking queue to let worker threads communicate with emitters of tasks. In more, the mechanism I used to structure submission of macro-instructions to the thread pool using fastflow farm is quite inefficient, and it could be implemented e.g. exposing an interface that directly relates with a spinwaiting emitter thread to make it send-out tasks to the worker threads.

## References

[1] Marco Aldinucci et al. "Fastflow: high-level and efficient streaming on multi-core". In: *Programming multi-core and many-core computing systems, parallel and distributed computing* (2017).

[2] Yaakov Engel, Shie Mannor, and Ron Meir. "The kernel recursive least-squares algorithm". In: *IEEE Transactions on signal processing* 52.8 (2004), pp. 2275–2285.

[3] Herbert Jaeger. "The "echo state" approach to analysing and training recurrent neural networks-with an erratum note". In: *Bonn, Germany: German National Research Center for Information Technology GMD Technical Report* 148.34 (2001), p. 13.

[4] kucukemre uygar. "Echo state networks for adaptive filtering". In: ().