SISSA - ICTP

Master in High Performance Computing

Master's Thesis

Performance-driven refactoring of Potts associative memory network model

Author:
Leonardo Romor

Supervisor:
Alessandro Treves
Giuseppe Piero Brandino

ACADEMIC YEAR 2014/2015

Abstract

Neural networks simulations have always been a complex computational challenge because of the requirements of large amount of computational and memory resources. Due to the nature of the problem, a high performance computing approach becomes vital, because the dynamics often involves the update of a large network for a large number of time steps. Moreover, the parameter space can be fairly large. An advanced optimization for the single time step is therefore necessary, as well as a strategy to explore the parameter space in an automatic fashion.

This work first examines the purely serial original code, identifying its bottlenecks and inefficient design choices. After that, several optimizations strategies are presented and discussed, exploiting vectorization, efficient memory access and cache usage. The strategies are presented together with an extensive set of the benchmarks and a detailed discussion of all the issues encountered.

The final part of the work is the design of a high throughput approach to the paramenter sweep, necessary to explore the behaviour of the network. This is implemented by means of a task manager that takes care of running simulations from a batch of predefined runs in an automatic way and collects their results. A detailed performance analysis of the task manager is reported.

The results of the work show a consistent speed up for the single-run case, and a massive productivity improvement thanks to the task-manager. Moreover, the code base is now reorganized to favor extensibility and code reuse, allowing the application of several of the present strategies to other problems as well.

Contents

Chapter 1

Introduction to the Algorithm

The Potts associative memory network model is an extension of Hopfield model to more than two states, intended as a simplified model of macroscopic cortical dynamics, in which each Potts unit stands for a patch of cortex. The internal neuronal dynamics of the patch is not described by the model, rather it is subsumed into an effective description in terms of graded Potts units.

The Potts network exhibits attractor dynamics similar to that of the Hopfield model. The network is able to store and retrieve a large number of global patterns (network attractors) when the connectivity matrix is suitably constructed. Here however we are not so much interested in the statistical properties of the network, rather we want to exploit dynamical properties that may be relevant to cortical processing.

Such dynamics are defined such that after retrieving an externally cued attractor, the network can continue jumping, or latching, from attractor to attractor, driven by adaptation effects.

The network is a directed graph made of N nodes with C connections. Each unit has a fixed amount of states S plus an inactive state (labeled from now on by the 0 index). For a given unit i the state of the unit is described by a vector with $S + 1$ components. This is represented by the symbol σ_i^k with $k = \{0..S\}$ and subject to the constraint:

$$\sum_{k=0}^{S} \sigma_i^k = 1 \tag{1.1}$$

The σ_i^k can be seen as the unit vector in a S+1 dimensional space, indicating how much the unit is pointing towards a specific state.

Each connection has a weight describing numerically the strength of that connection where only active states take part in the construction of J. Specifically J_{ij}^{kl} is the weight of the connection between the state k of unit i and the

state l of unit j. Moreover the J matrix is multiplied by a dilution matrix C which is a random matrix containing zeros and ones aiming at a more realistic representation of cortical connectivity. Throughout this work, we will use lower indices as unit indices and upper indices as state indices.

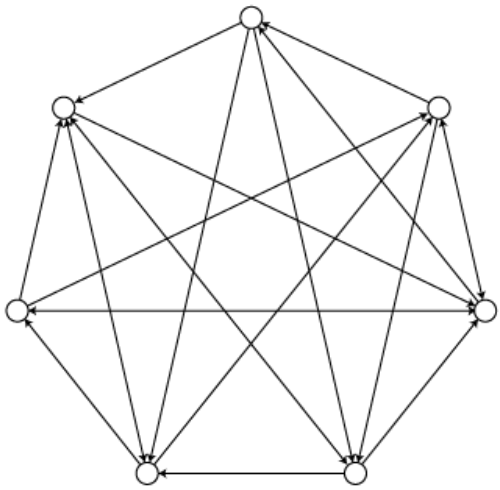

Figure 1.1: Example of the network topology with N=7 C=3

The aim of the simulation is to collect statistical data on the dynamics of the model, such as:

- behavior of the latching sequence using random or artificially correlated patterns.
- the dependence of the latching length on the network parameters.
- the study of limit cases (ex. $S >> 1$).

1.1 Patterns Generation

Before the initialization of the network, p patterns are generated using a patterns generating algorithm. These patterns will define, after being used in the weights generation, the attractors of the dynamic system.

The multifactorial pattern generating algorithm allows for the construction of patterns for various degrees of pairwise correlation.

To produce the patterns, a set of factors are generated. Each factor is a subset of the total set of units and may share part of its units with another factor. In the second step, global patterns(indexed by μ) are generated from

the factors which are indexed by r in order of decreasing mean importance. For each global pattern, a weight γ_r^μ describing the factor importance is assigned (eq.1.2)

$$\gamma_r^\mu = \epsilon e^{\xi r} \tag{1.2}$$

where ϵ is taken to be 0, with probability $1 - a$, with a the activity parameter, otherwise drawn with flat distribution between 0 and 1.

A value between $1...S$ representing the index of one of the states s_r is then drawn and a contribution γ_r^μ is added to the field onto each Potts unit over which factor r was defined, in the direction s_r.

After accumulating contributions from all factors, the direction in which each unit received the largest field is computed, and aN units receiving the largest maximal fields are assigned the corresponding direction s_r in pattern μ, while the remaining $(1 - a)N$ units are assigned the null state for pattern μ.

1.2 Network initialization

Prior to initializing the network, J and C are constructed. To initialize the J tensor we use the patterns generated by the patterns generating algorithm:

$$J_{ij}^{kl} = \frac{c_{ij}}{Ca(1 - \frac{a}{S})} \sum_{\mu=1}^{p} (\delta_{\xi_i^\mu k} - \frac{a}{S})(\delta_{\xi_j^\mu l} - \frac{a}{S}) \tag{1.3}$$

in which c_{ij} is defined in such a way that $c_{ij} = 0$ if the unit i is not connected to j while $c_{ij} = 1$ otherwise. The a is called the sparsity parameter and $\delta_{\xi_i^\mu k}$ is a Kronecker delta in such a way that it is 1 if if unit i of pattern μ is in the k-th state and 0 otherwise.

In the code the connectivity matrix C is represented by a $N \times C$ integers matrix where each column represents of set of units that are connected to the unit with the index of that column.

The initial active states are defined as follows:

$$\sigma_i^k \approx 0 \tag{1.4}$$

The initial inactive states instead, are computed in the following way:

$$\sigma_i^0 = 1 - S\sigma_i^1 \tag{1.5}$$

Here we introduce another quantity r_i^k, called the input. This quantity controls the dynamics of the state of the unit. The initialization of the active and inactive inputs are given by the following relations:

$$r_i^k = h_i^k = \sum_j J_{ij}^{kl} \sigma_j^l \tag{1.6}$$

$$r_i^0 = 1 - \sigma_i^0 \tag{1.7}$$

1.3 Dynamics

During the dynamics stage the whole network is updated until a predefined number of *total updates*(or *tupdates*)[1] is reached, or, an exit condition occours. At each time step, every unit of the network is updated, but the order in which the units are updated is selected randomly at each iteration. Moreover, the units update is strictly sequential because, as we will see in the next section, the dynamics of a given unit at time $t+1$ depends on the whole state of the network at time t.

1.3.1 Update Rule

A single unit is updated using the following equation:

$$\sigma_i^k = \frac{e^{\beta r_i^k}}{\sum_{l=0}^{S} e^{\beta r_i^l}} \tag{1.8}$$

r_i^k is the local field that the unit i receives from the other units. For $T = \beta^{-1} = 0$ corresponds to a strictly single active state being activated.

The $r^0(t)$, $r^k(t)$ and $\theta^k(t)$ obeys to three different differential equations, each with a time constant τ_1,τ_2,τ_3:

$$\tau_1 \frac{dr_i^k(t)}{dt} = h_i^k(t) - \theta_i^k(t) - r_i^k(t),\ \forall k \neq 0 \tag{1.9}$$

$$\tau_3 \frac{dr_i^0(t)}{dt} = \sum_{k=1}^{S} \sigma_i^k(t) - r_i^0(t) + U \tag{1.10}$$

[1] A *total update* here is defined as the set of N operations needed to update the whole network (all the units) exactly one time

with h_i^k computed at each time step using the equation below:

$$h_i^k = \sum_j J_{ij}^{kl}\sigma_j^l + w(\sigma_i^k - \frac{1}{S}\sum_{l=1}^{S}\sigma_i^l) + ge^{-(t-t_0)/\tau}\delta_{\xi_i^\mu k} \tag{1.11}$$

The middle term of the equation 1.11 is a self reinforcement term, introduced in order to model non linear convergence to the current active state, while the last term is the initial cue of the network, enabled when $t > t_0$, that pushes the network into the basin of attraction of one of the global patterns μ. The cue term exponentially decades with a time constant τ after being enabled.

The dynamic threshold θ affecting only active states is instead ruled by:

$$\tau_2\frac{d\theta_i^k(t)}{dt} = \sigma_i^k(t) - \theta_i^k(t),\ \forall k \neq 0 \tag{1.12}$$

The differential equations 1.9,1.12,1.10 are integrated with the time forward finite difference method in order to be evaluated in the code.

The network status is evaluated every fixed number of steps. During this phase, the cross correlation or what we call the overlap between the stored pattern and the network state is evaluated, with the rule (eq1.13).

$$m^\mu = \frac{1}{aN(1-a/S)}\sum_{i=0}^{N}\sum_{k=1}^{S}(\delta_{\xi_i^\mu k} - \frac{a}{S})\sigma_i^k \tag{1.13}$$

Using this measure, it is possible to track the evolution of the network, in and out of one of the many global attractors guided by input and the threshold variables. One of the major objectives is to study how the length of the sequential retrieval of each global pattern (latching) is affected by network parameters.

1.3.2 Exit condition

Depending on the network parameters that we set the dynamics may either cease after a finite number of time steps or go on indefinitely. In the former case the network will arrive to a quiescent state. To check whether the network is in a quiescent regime or not all the correlations are compared to a threshold value. If the all correlations are lower than this threshold value the simulation exits. In the latter case, a maximum number of time steps is defined. If the dynamics does not cease by itself and reaches this maximum value, again the simulation exits.

Chapter 2

Original code

In this chapter we will focus on the description of the original code in order to analyze its efficiency and identify the possible optimizations to be performed, with the aim to decrease the total simulation time. The original code is written in C++ and structured in three executables where one of them is a "manager" that runs them sequentially. The first executable produces the patterns and saves them into a file while the second one is focused on the initialization of the network, its dynamics and the data collection. The manager executable has the task to compile and run multiple times the two executables listed before. The initial parameters of the network are saved in a separate file using precompilation defines and are described in the table 2.1

Increasing the number of the stored patterns or lowering the connectivity increase the latching length and so the simulation time (figure 2.1).

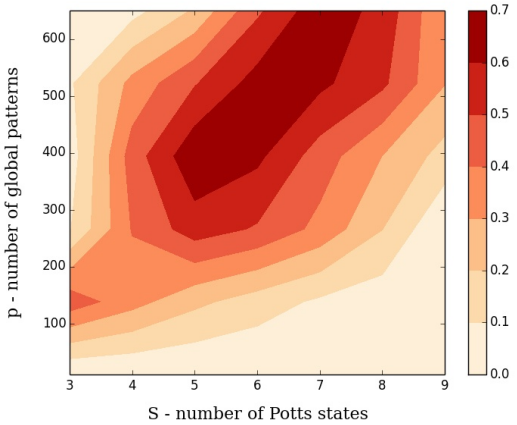

Figure 2.1: parameters dependence for the latching regime: red, high latching regime, white, quiescent regime.

Table 2.1: Set of initial parameters

N	total number of units	Trete	*tupdates*
Cm	connectivity	tempostampa	number of steps in order the correlation to be evaluated
p	number of patterns	NumSet	number of different random sequences
S	number of states for each unit	N_fact	Number of unit for each factor
a	sparsity parameter	Num_fact	number of factors
U	static threshold	Num_u	N
b1	time constant from the integration of eq1.9	Num_p	p
b2	time constant from the integration of eq1.12	Num_s	S
b3	time constant from the integration of eq1.10	a_mod	sparsity parameter
beta	inverse temperature	eps	noise parameter
w	self reinforcement term	a_pf	activity parameter
g	cue constant eq1.11	fact_eigen_slope	exponential slope in pattern generation
tau	cue exponential time constant		

For certain combinations of the input parameters, the network can reach a regime of infinite latching, making hard to envision a maximum number of operations needed to stop a simulation. For each set of network parameters the simulation is run multiple times with different cue patterns and since the latching length is unknown, the network has to be updated the highest possible number of times, in order to be able to spot long latching sequences that would otherwise be marked as infinite. To quantify the time spent in the various part of the code we can compare the number of time updates required to evaluate the pattern generation or the initialization. To do so we count the clock cycles elapsed for each of those code sections using the standard library clock function and then we divide by the number of clock cycles required to perform a whole network update (*timestep*).

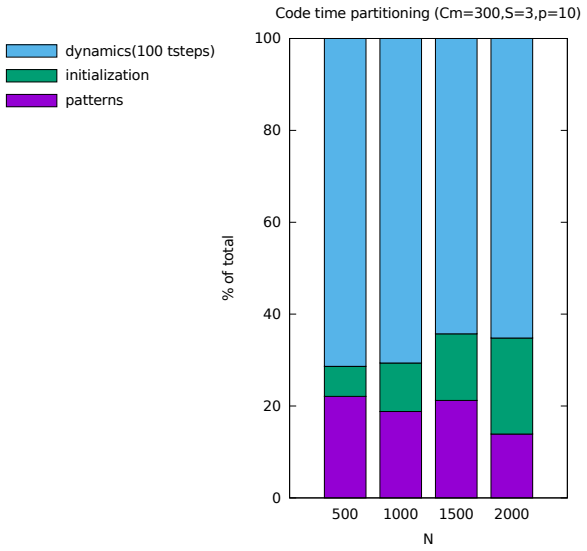

Figure 2.2: Percentage of single time steps required to simulate each stage of the code.

2.1 Pattern Generation

The pattern generation is a one-time task that occours at the start of every simulation and, in the original code, is placed in another executable. During the pattern generation the code generates an output file containing the patterns. The patterns are stored saving the state index for each unit for every pattern. We are interested in the relation between the time needed to perform the pattern generation task and the initial parameters to show how and when it could give an important contrbution to the whole simulation time. The results of this analysis are shown in 2.3a

As we can see, all the plots are decreasing functions of the corresponding parameters except the one involving the number of initial patterns. This means that the more we grow those parameters the less the pattern generation will affect the computational time compared to the dynamics.

On the other hand, increasing the number of patterns, the generation will take a larger and larger amount of time. However, in such a case the latching sequence is typically very large as well, such that the pattern generation time should be irrelevant also in the case of large p. Moreover given a certain set of patterns, the time evolution can be performed several times with different parameters. Since the patterns depends on the total number of unit N, the number of pattern p and number of stats S, it is possible to reuse the same pattern while modifing all the other parameters for example the connectivity,

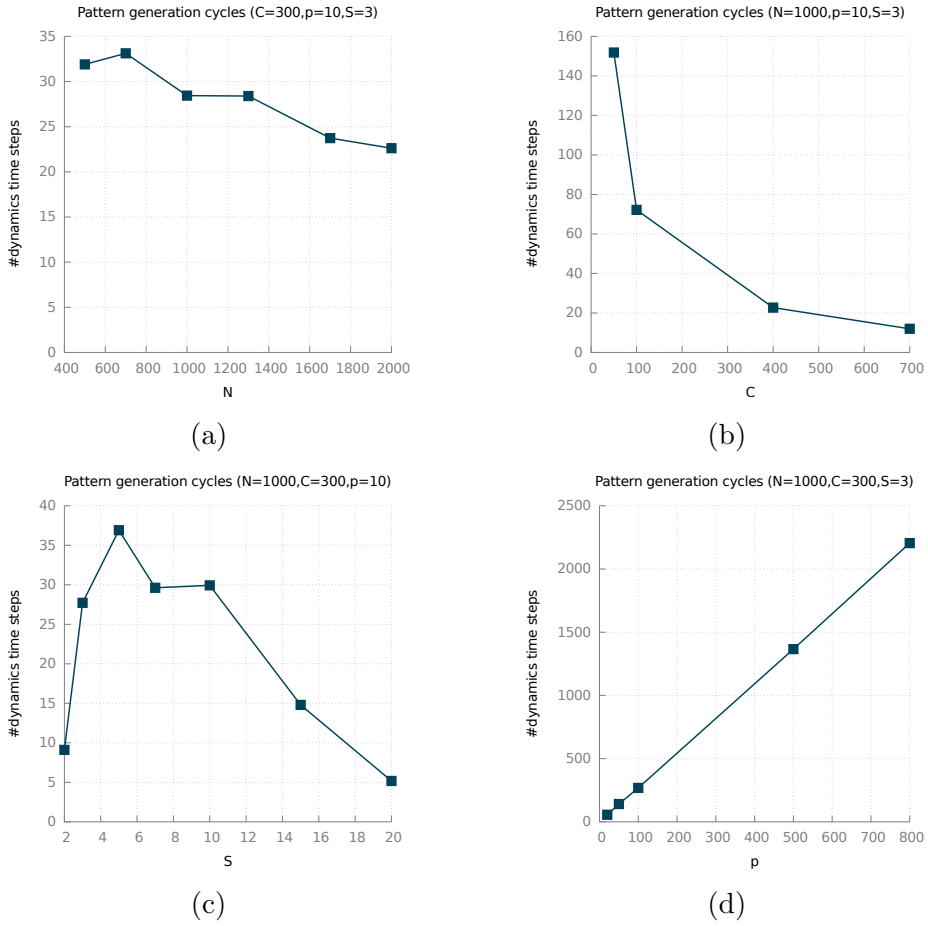

Figure 2.3: Dependence of the pattern generation time with the initial parameters N, Cm, S, p. The time unit of these plots is the time required to evaluate one-time step in the simulation.

the time integration constants, the static threshold, etc...

2.2 Initialization

The initialization consist in loading in memory the the generated patterns and then fill the array of states, the connection matrix and the J tensor. During this stage a sequences matrix containing different sets of shuffled integers from $0..N-1$ is generated. We remind that the dynamics of the unit at time $t+1$ depends on the whole state of the network at time t. This implies that the n-th unit being updated will have its dynamic effected by the n-1 unit updated so far. To prevent the introduction of a bias in the

dynamics, the sequence in which the units are updated at a given time step needs to be random. The sequence matrix introduced above meets exactly this need. At every time step one of the possible sequence is randomly selected and the update is performed using such a sequence. However, from the point of view of possible optimizations, this routine is irrelevant since the number of operations to construct it is of order N and so it will not be considered.

The number of operations required to fill the array of states is required a number of cycles proportional to $N \times S$ while for the connection matrix is proportional to $N \times C$. To build a more dense matrix in order to save memory and computation, the connection matrix is just filled with the indices each unit is connected to. For the J tensor instead a five-nested loop is used, requiring $N \times C \times S \times S \times p$ total number of operations.

The whole initialization task requires $N(S + C(1 + p \times S^2))$ operations. This is an expensive part of the code but since is a one-time task it has not been the center of the work. Also it applies what already said for the pattern generation: there are for sure cases where the pattern generation and the initialization may require a large time, but we are not concentrating on those cases because they are meant to be only a small part of the simulations the researchers are interested in.

2.3 Dynamic part

When the main loop of the dynamics starts it will cycle *tupdates* times or until the exit condition is reached. At each cycle a unit is picked using the pregenerated sequence matrix.

The tensor product of the update algorithm(the first term of eq.1.11) is technically implemented following the representation scheme in fig.2.4

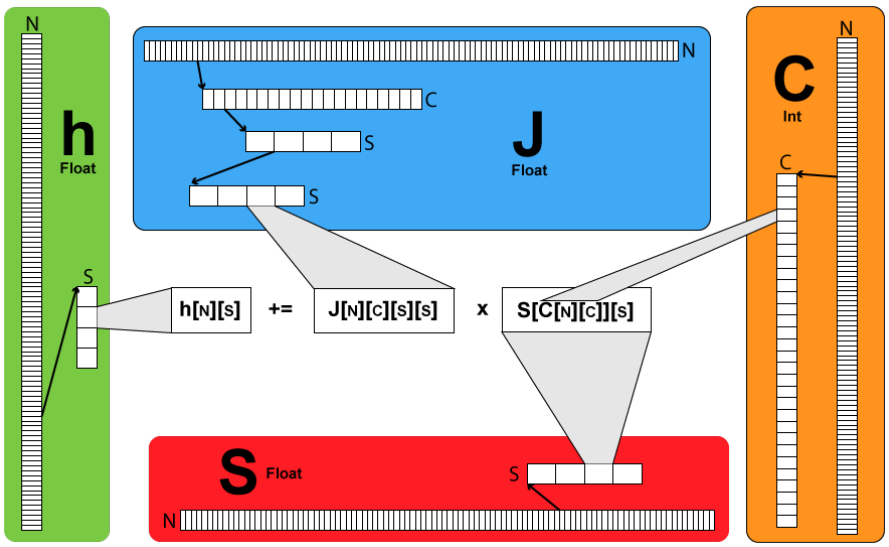

Figure 2.4: Data model of the original code. Shows how many arrays are involved, their structure in memory and how they are involved in the tensor product. The arrow shows that each sub-piece of memory is pointed by the parent array and how the array S retrieve their index from the connection matrix C.

Figure 2.4 is a representation of the arrays required to evalute the tensor product, which is, the most computational consuming part of the code:

$$h[i][k] += J[i][x][k][l] * s[C[i][x]][l]$$

The required states connected to the unit are retrived using the connection matrix (in the image denoted with the capital letter C) and, because C is randomly generated, consecutive indices will point to non contiguous locations in memory, making impossible for the compiler to perform any optimization like vectorization or to exploit cache effects. Also, the arrays are stored as nested pointer arrays, this means that their memory contiguity is not guaranteed, reducing the memory fetching efficiency.

When the exit condition is reached, the index variables of each loop are set to the limit of their loop condition in order to exit each loop simultaneously.

The number of operations of the biggest loop is of the order $tsteps \times N \times S \times C \times S$ times, where tsteps is the number of time steps after which the exit condition is met and in the simulation is bounded by the maximum value *tupdates*.

2.4 Memory load

The memory occupation of the dynamics is dominated by the J tensor which is of size

$$(4 \times N \times C \times S \times S)Bytes$$

as an example using the initial parameters N=1000, Cm=500, S=10, the memory occupied is roughly 200 MBytes making the memory growth not an issue for the parameters range we are interested in and for the memory avaiable per node in the ULISSE cluster(40GB).

2.5 Chaotic regime

In some cases when the network is embedded with a high number of patterns $p >> 1$ and the other parameters are such that the network is in a high latching regime, the basins of attraction start to merge giving rise to a complex energy landscape. Given that the simulation involves an enormous number of recursive operations, small changes in for example the summation order may lead to different latching sequences and lengths. Because of this reason, using a different optimization scheme such as the aggressive compiler optimizations (for example g++ or icpc compiler -O3 flags) may result in different simulation outputs. However, it is important to notice that both the original code as well as the optimized one yield initial sequences of patterns that are identical and it is only at a certain point that a deviation occurs. This deviation is a random and not a systematic error owing to the sensitivity of the system to the initial conditions.

Chapter 3

Optimization strategies

In this chapter we present a detailed report of the Potts neural network code refactoring. This is organized in three main parts

- Code restructure, in particular the new object-oriented design.
- High connectivity regime strategy
- Low connectivity regime strategy

One of the strongest limitations of the code is the stricly serial update rule of the network that denies any high level optimization with threads or processes for a single simulation reducing the allowed strategies to cache efficiency and vectorization.

3.1 Code restructure

The reorganization of the code is aimed to increase extensibility and readability, in order to allow new developers to easily work on the code. The new code is written in C++11 and is separated using the following folder structure:

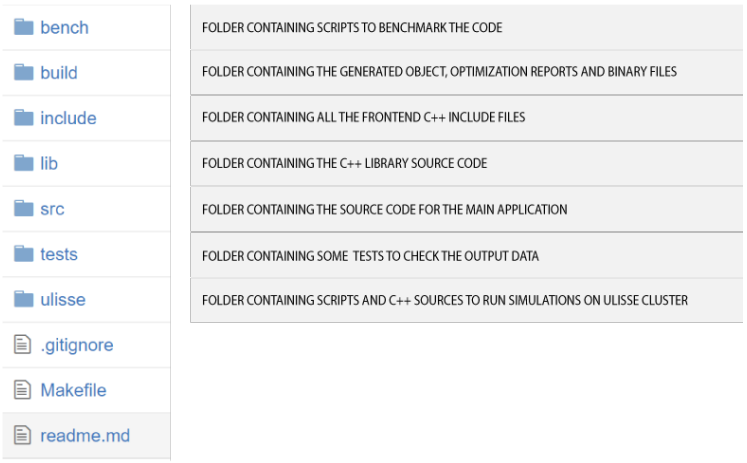

Figure 3.1: Project folders tree

The compilation is handled by GNU Make and can be compiled with any C++ compiler that supports C++11 standard.

In the new code the following object with their respective most important methods have been defined:

- **PatternGen**
  - PatternGen: initialize the patterns object and allocate the array for storing the number of patterns.
  - generate: generate the patterns
  - eval_stats: evaluate statistical data of the generated patterns
  - get_patt: retrieve the pointer of the generated patterns array
- **PNet** to setup the network object.
  - PNet: setup the network object and allocate its member arrays.
  - init_network: initialize the network, filling the connection matrix, the J tensor and the initial values as described in 2.2
  - start_dynamics: starts the network dynamics and stores in a member variable the latching sequence

3.2 Testing environment

3.2.1 Hardware environment

Both the original code and the optimized code have been developed and tested on the ULISSE SISSA cluster. The specifications of the compute nodes are:

Table 3.1: Ulisse node hardware specifications

Architecture	x86_64
CPU(s)	20
Core(s) per socket	10
Model name	Intel(R) Xeon(R) CPU E5-2680 v2
L1d cache	32K
L2 cache	256K
L3 cache	25600K
Total memory	41123748K

3.2.2 Regression tests

To check the coherence of the new code with respect to the original one, a total of five regression tests has been developed. For each test the new and the original code print in a text file part of the state of the simulation. After that, a simple diff[1] command is run between the two files.

The list of tests is:

1. **pattern generation test**: compares the generated patterns
2. **states initialization test**: compares the initialized states
3. **connection initialization test**: compares the connectivity matrix
4. **J test**: compares the J tensor
5. **states update test**: compares the states at the end of the simulation

[1] A bash command that outputs the differences between two files

3.2.3 Benchmarks setup

To benchmark the code we ran the simulation multiple times on the *ULISSE* cluster. Since the number of time steps required to hit the exit condition is strongly dependent upon the initial parameters, to be able to properly benchmark the new implementations we forced the dynamics to always execute a fixed number of time steps.

The benchmarks are run sweeping over the N and C parameters.

Every benchmark has been run at least 5 times in order to reduce fluctuations.

3.3 High Connectivity regime

The first optimization strategy adopted was developed specifically for the high connectivity regime, which means a N/C ratio lower than 3. Since the connectivity matrix is initialized at random at every simulation, when the connectivity gets large this implies that the access to the S array as well as the tensor J are rather erratic, causing a large number of fetches from the main memory. In this regime it is more efficient to hide the connectivity directly inside the J tensor, trading floating point operation for memory fetches.

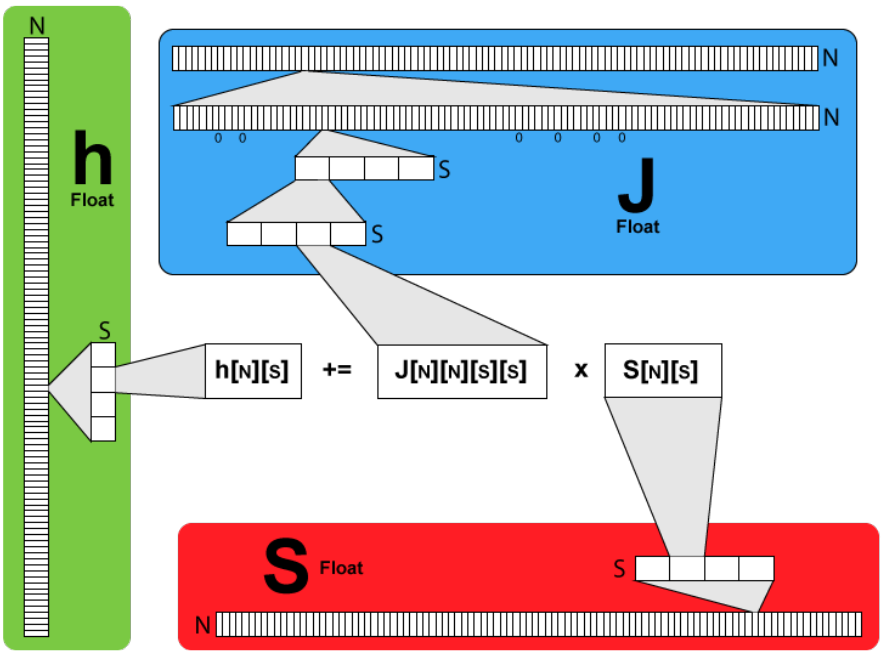

Figure 3.2: High connectivity new memory management. The connectivity matrix is hidden inside the J tensor that now has a size of $N \times N \times S \times S$

This means that if two units states are not connected, we set the corresponding value of the J tensor to zero.

$$\forall i, j \mid c_{ij} = 0 \implies J[i][j][k][l] = 0 \; \forall k, l$$

In this way, while looping on j and s, the access pattern is now contiguous, allowing faster fetches from the main memory and vectorization. A pictorial view of this strategy is reported in figure 3.2. This increase the final size of the J tensor, that now is an array of $N*N*S*S$ floats but we completely remove the non contiguity of the original code in the fetching process of the states. However the drawbacks are rather irrelevant in this regime ($C/N > 1/3$): first the increase in the memory load is at most a factor of 3. Since the typical memory load is in the order of few hundred Megabytes, the extra memory requirement is not an issue. Second the increase of the number of operations, once again, at most a factor of three, is completely overtaken by the speedup of vectorization. Using this strategy, the number of operations and so the update time depends just on N and S, but not on the number of connections, such that increasing the connectivity of the network yields exactly the same simulation time.

3.3.1 Benchmarks

The first effect of this strategy should be a reduction of the cache reads and cache misses. In the following table we report a comparison between the original code and the high connectivity strategy, using cachegrind tool from the valgrind suite.

Table 3.2: Cache reads

Code version	Dr	D1mr(miss rate%)	DLmr
original code	51,751,792,425	8,570,583,711(16%)	1,983,237,597(%3)
high connectivity	20,414,022,668	1,803,628,810(8%)	1,694,415,417(%8)

Table 3.3: Cache writes

Code version	Dw	D1mw(miss rate%)	DLmw(miss rate%)
original code	8,722,172,354	14,308,771(%0.2)	4,126,601(%0.04)
high connectivity	1,796,819,369	33,574,766(%2)	1,227,975(%0.06)

The new data model shows a substantial reduction of cache misses. This is due to the new memory organization which has increased the spatial locality, thanks to the usage of contiguous arrays. Regarding the time profiling, we can see the scaling of the new strategy compared with the original code:

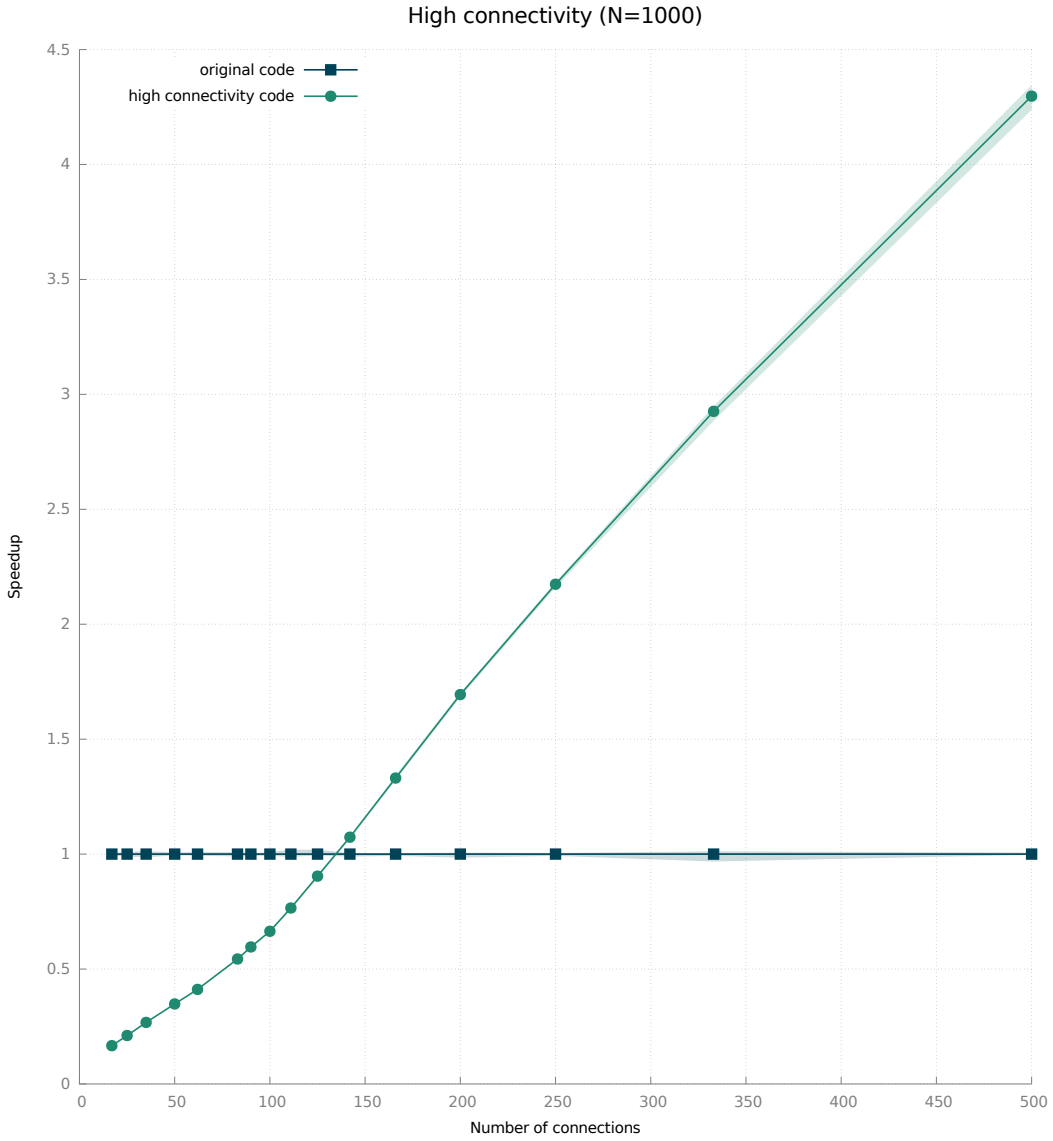

Figure 3.3: Speedup of the high connectivity strategy compared with the original code. Results of the benchmark on the ULISSE cluster

The plot in figure 3.3 has been made by fixing the number of units to 1000 and then running the simulation for differenct values of the C parameter.

Since low connectivities are in general scientifically more interesting, the sampling gets denser for small C.

The speedup of the high connectivity strategy with respect to the original code is almost linear. The execution time is independent on C, instead the execution time of the original code goes roughly as C resulting in the linear speedup as seen in fig 3.3.

For a number of connections higher than ≈ 140 the algorithm scales up to 4.2x times faster than the original one. The speedup is due to the combined effect of both cache efficiency and vectorization. However as the C gets small the overhead of multiplying a lot of zeros starts to take over the memory contiguity effects, and we have a decrease in performance compared to the original code.

3.4 Low Connectivity regime

Since for low connectivity the previous strategy proved to be unsuccessful, we devised a new one specifically designed for this regime. The low connectivity strategy N/C ($\gtrapprox 3$) exploits both temporal and spatial locality plus the vectorization. To achieve this, it was necessary to change the retrieval of the states for each update: to perform a unit update at a given time step we need to re-use several times the states a unit is connected to. This means that collecting in a contiguous buffer all these states could improve spatial and temporal locality and allow vectorization. The original code instead was fetching the same states for S times inside the update subroutine. The new memory organization is shown in fig 3.4:

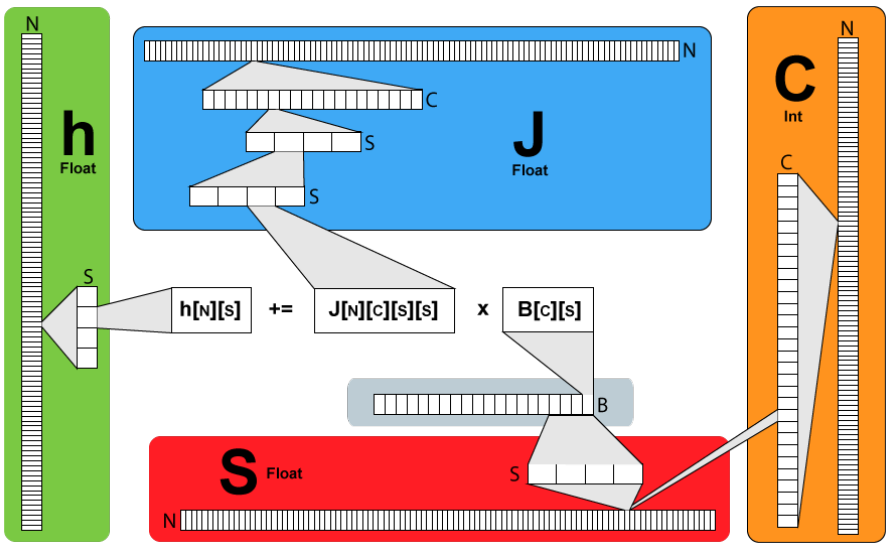

Figure 3.4: Low connectivity strategy memory organization. We can see that now we use a buffer as intermediate means to evaluate the tensor product

This approach allows vectorization, reduce the memory reads and an increased cache affinity even in the case of low connectivities. To show the efficiency of the buffer strategy in the case of repeated reads from a random index array we have written a sample code reported in Appendix A.

3.4.1 Benchmarks

To analize the performance, we first report the cache usage (table 3.4,3.5)

Table 3.4: Cache reads

Code version	Dr	D1mr(miss rate%)	DLmr
original code	51,751,792,425	8,570,583,711(16%)	1,983,237,597(4%)
low connectivity	15,610,241,799	717,318,361(5%)	568,800,527(4%)

Table 3.5: Cache writes

Code version	Dw	D1mw(miss rate%)	DLmw(miss rate%)
original code	8,722,172,354	14,308,771(%0.2)	4,126,601(%0.04)
low connectivity	4,455,908,387	25,825,680(%0.6)	846,748(%0.02)

As we can see the number of cache reads has decreased more than 3 times and the number of the relative number of cache-misses is one order of magnitude lower than in the original code. Also all the other parameters have decreased except for write misses which are anyway rather low. The performance of the low connectivity strategy is shown in figure 3.3.

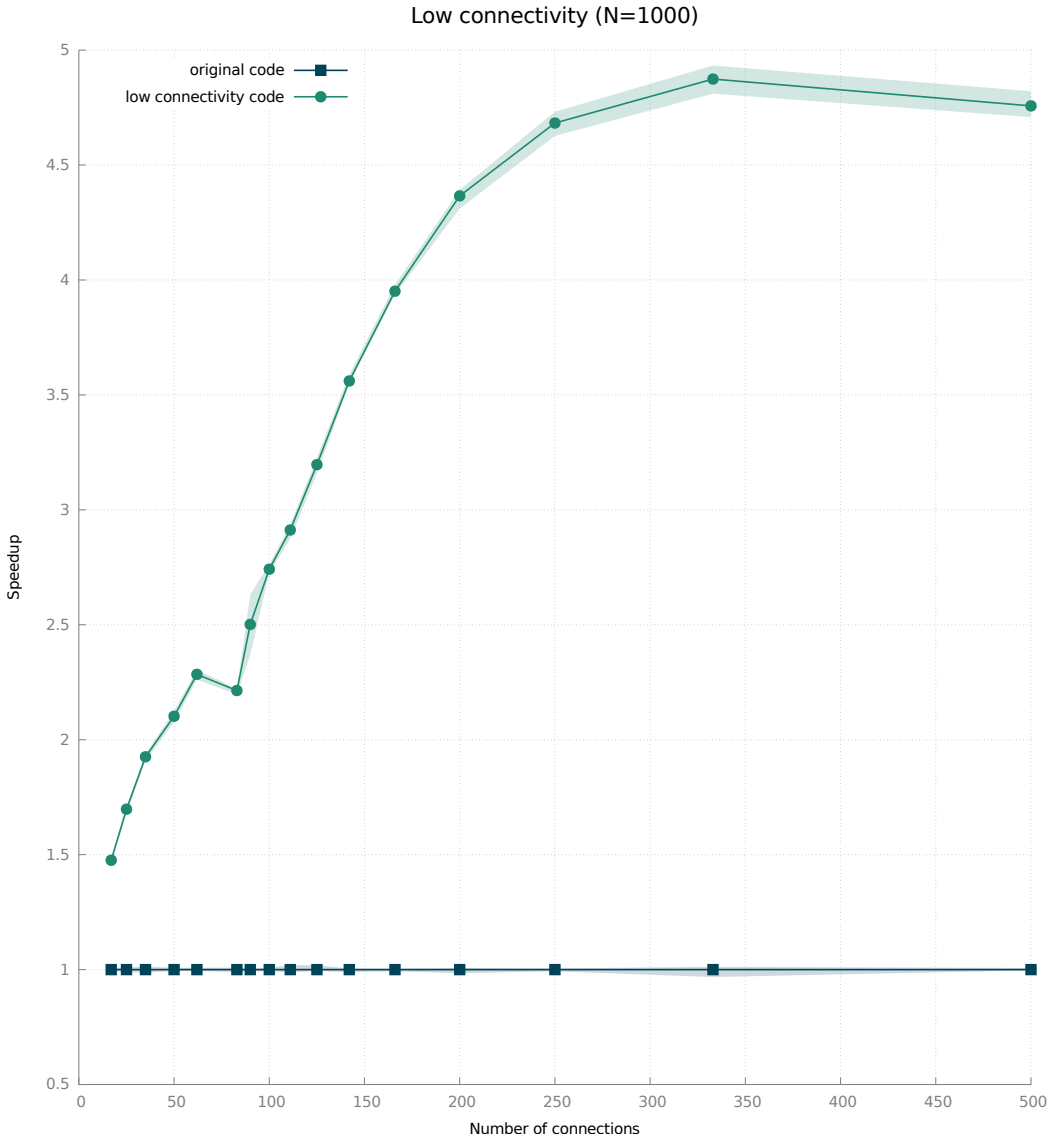

Figure 3.5: Low connectivity strategy benchmark ($N = 1000$ and $S = 3$)

From figure 3.5 it is clear that the buffer strategy gives a consistent improvement for every value of the connectivity C. The improvement goes from

1.5x for very low connectivies to 5x for $(N/C) \approx 2$.

3.5 Results comparison

In this section, we compare the performance of the different strategies implemented.

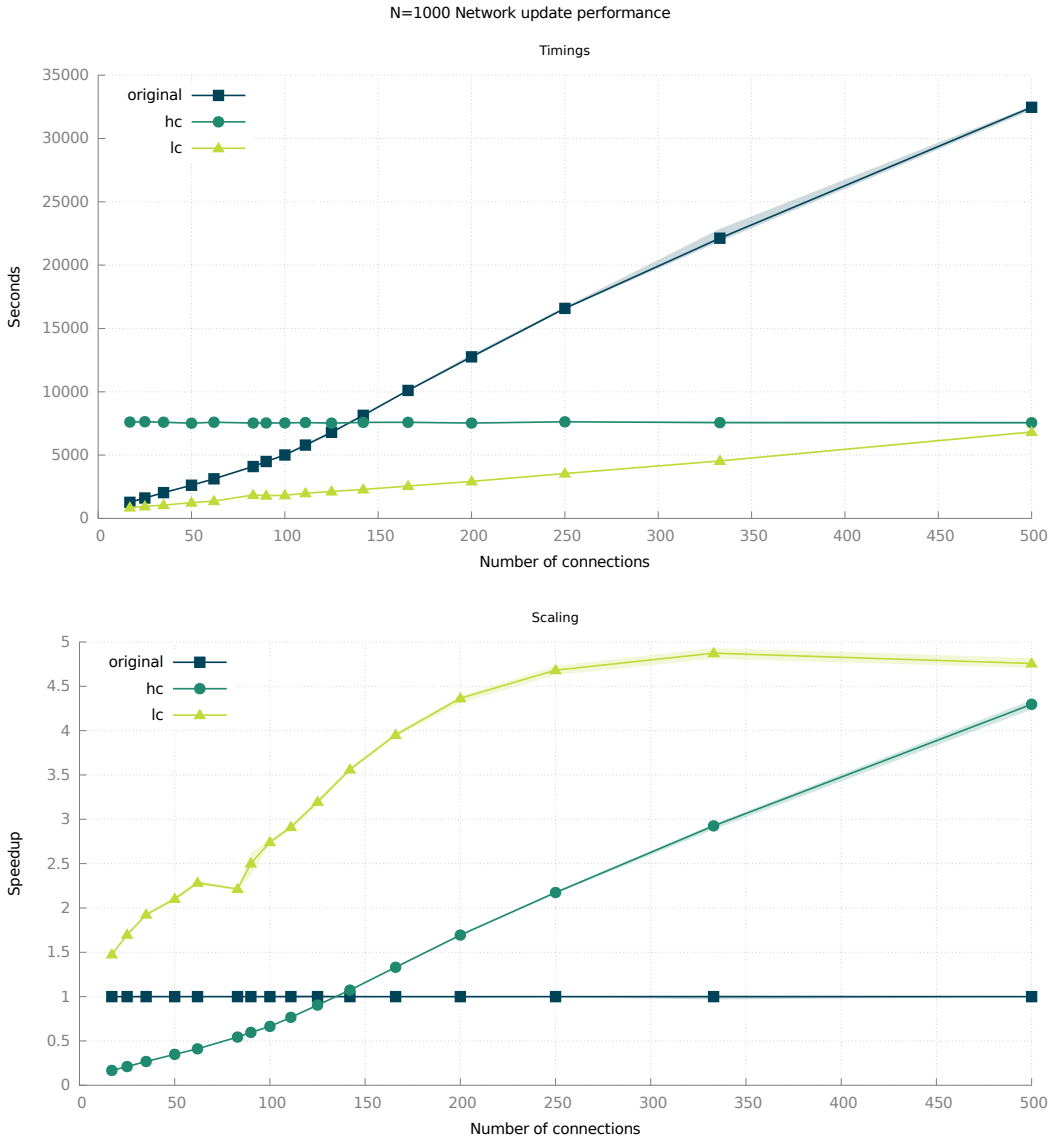

Figure 3.6: Benchmark comparison (N=1000)

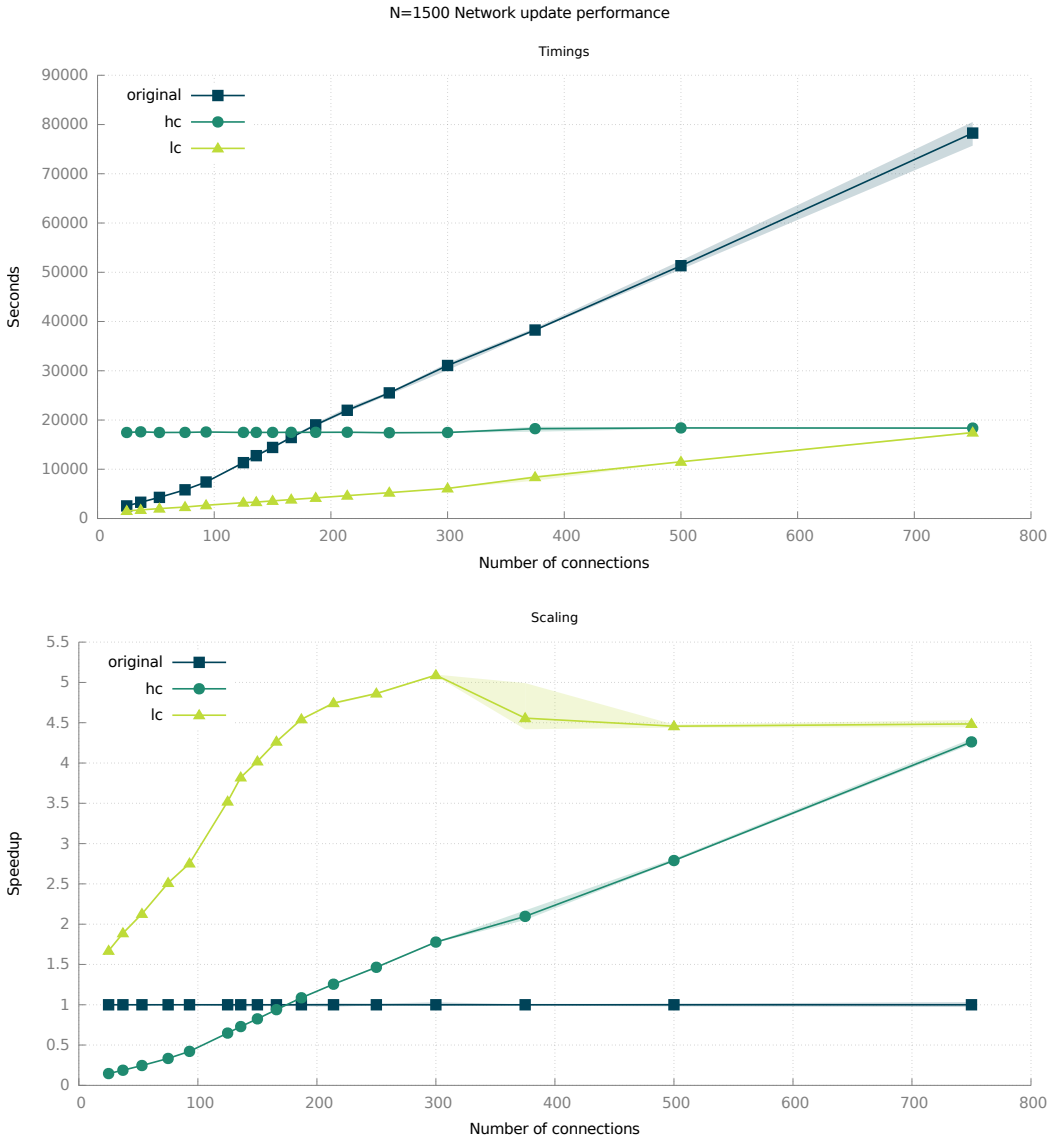

Figure 3.7: Benchmark comparison (N=1500)

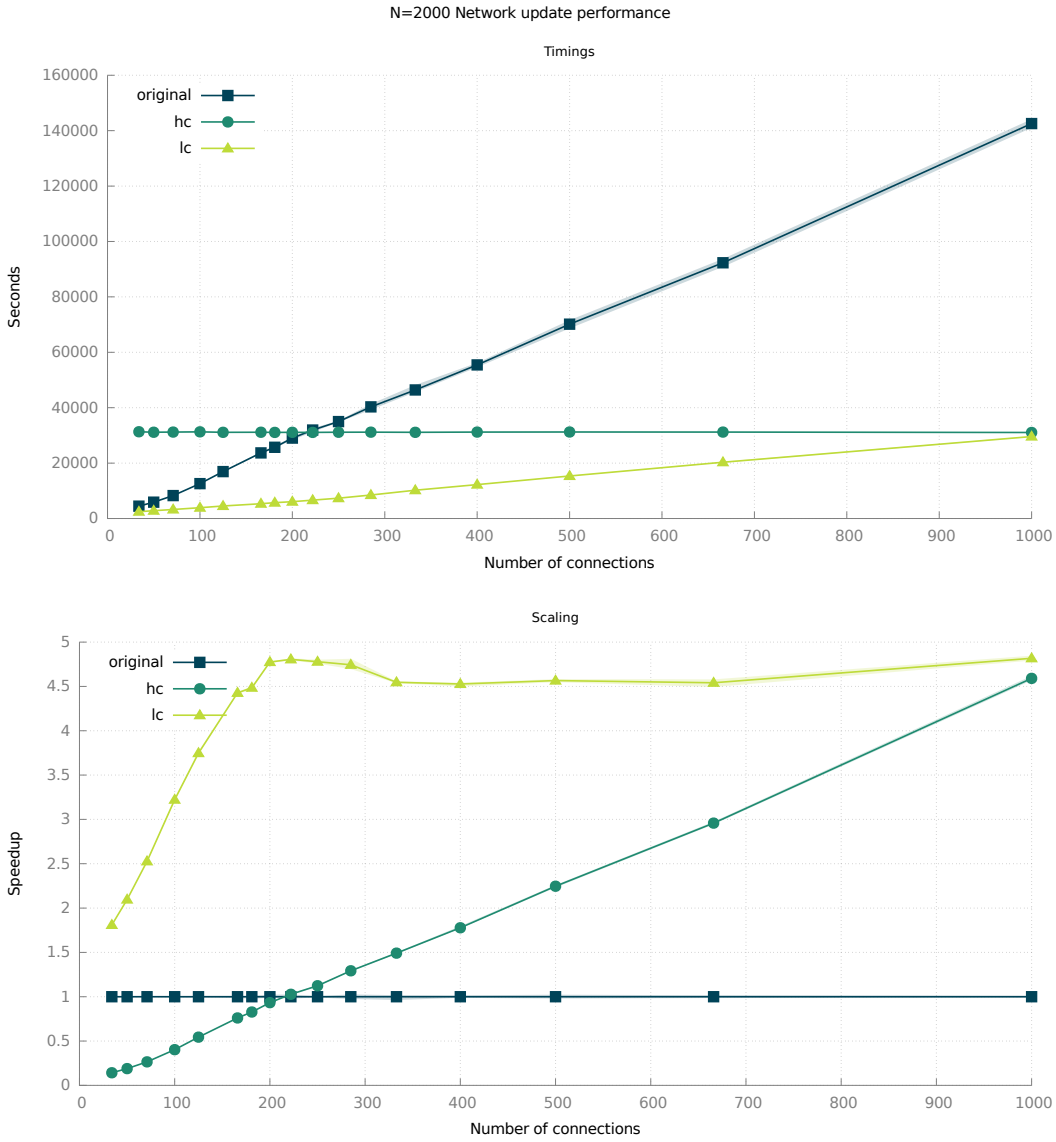

Figure 3.8: Benchmark comparison (N=2000)

From figure 3.6,3.7,3.8 it is clear that the low connectivity strategy show the best performance up to $N/C \approx 2$. However increasing C in such a strategy always shows a plateu in the speedup. This is due to the fact that for large C, the buffering requires too many operations compared to the gain given by vectorization. In this regime, the high connectivity strategy becomes more and more efficient.

Chapter 4

Task manager

In the previous chapter we focused on the efficiency of a single simulation, however, one of the main needs from the scientific point of view is to collect data from a large number of simulations in order to have a reliable statistics. The solution devised for this problem was the development of a task manager.

The idea is to place different simulations in different processes in order to run them simultaneusly, achieving a potential speedup limited only by the number of cores.

4.1 Algorithm

The task manager is developed using the MPI framework and a master/slave paradigm.

The master process is meant to deliver to each of the slave processes the set of initial parameters to start their own different simulations, and to keep a stack of the parameters sets to assign to the first avaiable slave process. To be able to communicate parameters data through MPI a custom MPI datatype has been created, which contains all the parameters required by a single simulation.

The task manager logic is schematized in the following way:

1. on startup, each slave notifies the master that is ready to receive a set simulation parameters.

2. the master deliver to all the avaiables slaves a different set of parameters from the parameters stack.

3. the slave execute their own simulation and notifies the master when done.

4. the master keeps distributing parameters sets until the stack is empty.

5. when the stack is empty, the master send a signal to all the slaves in order to kill them and then the code exits.

4.2 Tests

Figure 4.1 and 4.2 show the time taken to run a large number of simulatons concurrently. For small system sizes (figure 4.1) the overhead of the task manager is almost negligible. This can be seen because the time taken for 1 or 39 simulations is basically the same. In figure 4.2 instead the single simulation time increases and reaches a plateu after around 20 processes. This is actually not due to the task manager overhead but to shared L3 cache as we will discuss in the following section.

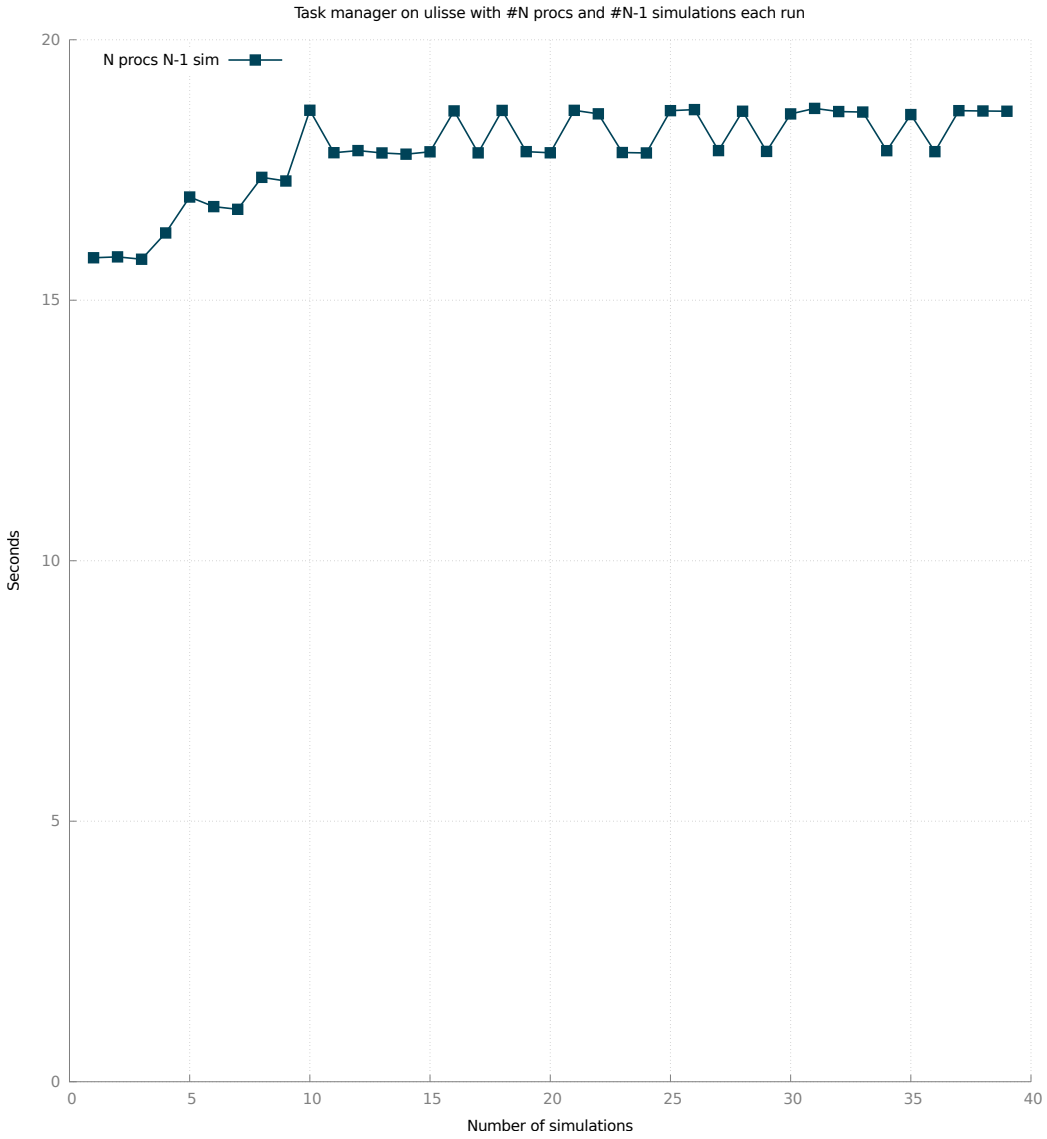

Figure 4.1: Task manager sweep, starting from one simulation to 39 with a step of 2. N=200,C=5,p=3,S=3, tupdates=500000

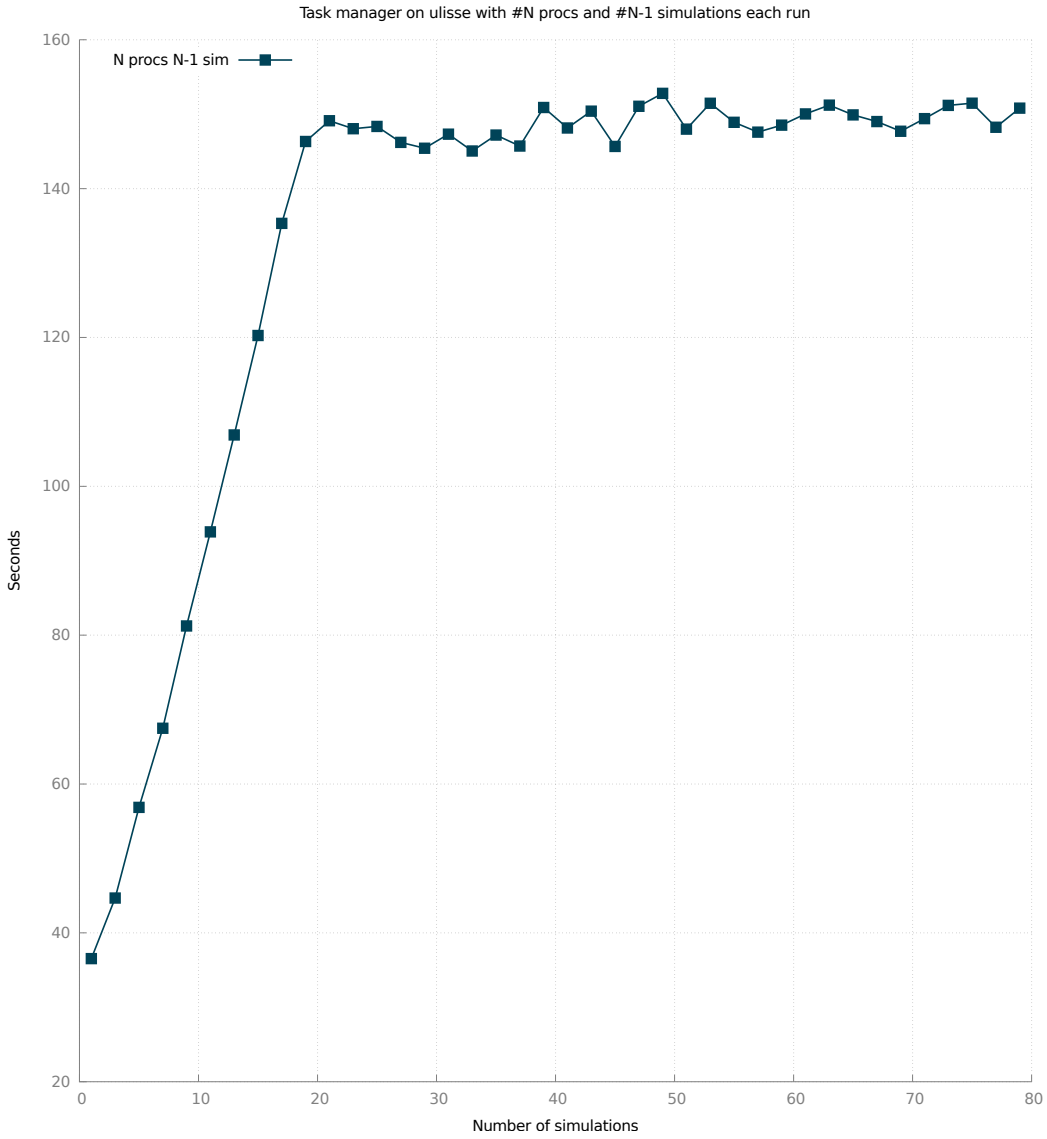

Figure 4.2: Task manager sweep, starting from one simulation to 79 with a step of 2. N=2000,C=500,p=20,S=6, tupdates=3000

4.2.1 The shared memory problem

As seen in the previous section, for large N the task manager shows a slowdown of each single simulation process. The reason behind this slowdown is the shared L3 cache between the cores inside a socket (we remind that in our case each ULISSE node is composed by a two sockets of ten cores each. All the cores in a socket share an L3 cache of 25MBytes).

Increasing the number of processes per node, the shared L3 caches starts to be used by more and more processes concurrently, leading to cache trashing.

The slowdown saturates at 20 cores which is the size of a ULISSE node. We remind that in these tests we are just running one simulation per core meaning that the global time reported always indicates the time of the slowest of the group.

However given the system size and so the amount of cache required, it is possible to identify an optimal number of simulations to be run per socket (using for example numactl, or MPI –npersocket) such that the balance between number of simulation run and time of single simulation yields the highest productivity.

Conclusions

In this work we showed the performance optimization of the Potts associative network model. Due to the very nature of the system and of the algorithm, a standard parallelization strategy was not possible, so we had to rely on low level optimizations, exploiting cache efficiency and vectorization.

The most time consuming part of the algorithm is the network update function, where a tensorial product takes place, requiring a large number of both memory and computational operations. A complete code reconstructure has been performed and two different strategies have been developed. One the strategies targets the simulations with a high number of connections and the other one with low connectivities.

Both of the strategies showed remarkable speedups up to 5x for the low connectiviy regime and even higher for the high connectivity regime. Because of the need of the reaserch team to collect a large amount of data to perform statistical analysis, we developed a parallel task manager. The task manager by itself brings negligible overhead and the overall time to solution for a large set of simulations is only limited by the avaiable amount of L3 cache per socket. All these optimizations are expected to increase the productivity of the research team by greatly reducing the time taken to perform statistics or large parameters sweep.

Acknowledgements

I would like to thank Dr. Giuseppe Piero Brandino for his expert advices and encouragement, the Professor Alessandro Treves and Vezha Boboeva for their encouragement and patience for teaching me the subject. Thanks to all the master's professors and students, especially Moreno Baricevic and Stefano Piani for their friendship and help. Thanks to my parents and my girlfriend Valentina, for all their love and support.

Appendices

Appendix A

In this section we show a proof of concept of the low connectivity algorithm. The following C code shows the efficiency gain by using a buffer strategy, if the buffer itself is reaused several times. The code evaluates a scalar product between two arrays of size *tsize*, but randomizing the access order to the elements. This operation is done multiple times but in two different ways. The first one, mimicking the original code, collects the values by retrieving them at each loop iteration while the second one saves the values (that are the same at every iteration) in two buffers, allowing vectorization and a better cache efficiency.

```
1  #include <stdlib.h>
2  #include <math.h>
3  #include <time.h>
4
5  int main(int argc, char ** argv) {
6
7      typedef  float __fpv;
8      int i,j,k,tsize;
9      if(argc == 2) {tsize = atoi(argv[1]);} else {tsize = 1024;}
10     int repetitions = 100000;
11
12     clock_t start_t, end_t, total_t;
13
14     __fpv * a = _mm_malloc(sizeof(__fpv) * tsize,64);
15     __fpv * b = _mm_malloc(sizeof(__fpv) * tsize,64);
16     __fpv * c = _mm_malloc(sizeof(__fpv) * tsize,64);
17
18     __fpv * buffer1 = _mm_malloc(sizeof(__fpv) * tsize,64);
19     __fpv * buffer2 = _mm_malloc(sizeof(__fpv) * tsize,64);
20     int * indexes1 = _mm_malloc(sizeof(int) * tsize,64);
21     int * indexes2 = _mm_malloc(sizeof(int) * tsize,64);
22
23     for(j = 0; j < tsize; ++j){
24         a[j] = j;
25         b[j] = j;
26         indexes1[j] = rand() \% (tsize);
27         indexes2[j] = rand() \% (tsize);
28     }
29
30     __fpv h = 0;
31
32     clock_t start, end;
33     start = clock();
34
```

```
35      for(k = 0; k < repetitions; k++){
36          h++;
37          //#pragma novector
38          for (j = 0; j < tsize; j++){
39              h += a[indexes1[j]] * b[indexes2[j]];
40          }
41      }
42
43
44      end = clock();
45      printf("NOT BUFFERED\nTIME: \%f ms\nSOLUTION = \%f\n",
46              1000.0*((double)(end - start)) / CLOCKS_PER_SEC, h);
47
48      h = 0;
49      start = clock();
50
51      __assume_aligned(buffer2, 64);
52      __assume_aligned(buffer1, 64);
53
54      for (j = 0; j < tsize; j++) {
55          buffer2[j] = a[indexes1[j]];
56          buffer1[j] = b[indexes2[j]];
57      }
58
59      for(k = 0; k < repetitions; k++){
60          h++;
61          //#pragma novector
62          for (j = 0; j < tsize; j++) {
63              h += buffer2[j] * buffer1[j];
64          }
65      }
66
67
68      end = clock();
69      printf("BUFFERED\nTIME: \%f ms\nSOLUTION = \%f\n",
70              1000.0*((double)(end - start)) / CLOCKS_PER_SEC, h);
71
72
73      _mm_free(a);
74      _mm_free(b);
75      _mm_free(buffer1);
76      _mm_free(buffer2);
77      _mm_free(indexes1);
78      _mm_free(indexes2);
79
80      return 0;
81  }
```

For the parameters set in the code (tsize=1000, repetitions=100000) the execution time is the following:

```
**********NOT BUFFERED**********
TIME: 68.912000 ms
SOLUTION = 25849778667520.000000
**********BUFFERED**********
TIME: 6.789000 ms
SOLUTION = 25849778667520.000000
```

As we can see the results are the same but the required time to compute the sum is different by an order of magnitude, showing that even if the number of loops and operations are the same, the fetching operations in the main memory plays a foundamental role, and the buffering strategy can then greatly increase the overall performance.

Bibliography

[1] Russo, Eleonora and Treves, Alessandro, *Cortical free-association dynamics: Distinct phases of a latching network*, American Physical Society, 10.1103/PhysRevE.85.051920

[2] Eleonora Russo and Vijay M K Namboodiri and Alessandro Treves and Emilio Kropff, *Free association transitions in models of cortical latching dynamics*, New Journal of Physics, 2008, http://stacks.iop.org/1367-2630/10/i=1/a=015008

[3] Alessandro Treves, *Frontal latching networks: a possible neural basis for infinite recursion*, Cognitive Neuropsychology, 10.1080/02643290442000329, http://dx.doi.org/10.1080/02643290442000329

[4] Intel ®Corporation, *A Guide to Vectorization with Intel ®C++ Compilers*, https://software.intel.com/en-us/articles/a-guide-to-auto-vectorization-with-intel-c-compilers