University of Plymouth

PEARL https://pearl.plymouth.ac.uk

Faculty of Science and Engineering School of Engineering, Computing and Mathematics

2020-07-31

Scaling the HTM Spatial Pooler

Dobric, D

http://hdl.handle.net/10026.1/19151

10.5121/ijaia.2020.11407
International Journal of Artificial Intelligence and Applications
Academy & Industry Research Collaboration Center (AIRCC)

All content in PEARL is protected by copyright law. Author manuscripts are made available in accordance with publisher policies. Please cite only the published version using the details provided on the item record or document. In the absence of an open licence (e.g. Creative Commons), permissions for further reuse of content should be sought from the publisher or author.

SCALING THE HTM SPATIAL POOLER

Damir Dobric[1], Andreas Pech[2], Bogdan Ghita[1] and Thomas Wennekers[1]

[1]University of Plymouth, Faculty of Science and Engineering, UK
[2]Frankfurt University of Applied Sciences,
Dept. of Computer Science and Engineering, Germany

ABSTRACT

The Hierarchical Temporal Memory Cortical Learning Algorithm (HTM CLA) is a theory and machine learning technology that aims to capture cortical algorithm of the neocortex. Inspired by the biological functioning of the neocortex, it provides a theoretical framework, which helps to better understand how the cortical algorithm inside of the brain might work. It organizes populations of neurons in column-like units, crossing several layers such that the units are connected into structures called regions (areas). Areas and columns are hierarchically organized and can further be connected into more complex networks, which implement higher cognitive capabilities like invariant representations. Columns inside of layers are specialized on learning of spatial patterns and sequences. This work targets specifically spatial pattern learning algorithm called Spatial Pooler. A complex topology and high number of neurons used in this algorithm, require more computing power than even a single machine with multiple cores or a GPUs could provide. This work aims to improve the HTM CLA Spatial Pooler by enabling it to run in the distributed environment on multiple physical machines by using the Actor Programming Model. The proposed model is based on a mathematical theory and computation model, which targets massive concurrency. Using this model drives different reasoning about concurrent execution and enables flexible distribution of parallel cortical computation logic across multiple physical nodes. This work is the first one about the parallel HTM Spatial Pooler on multiple physical nodes with named computational model. With the increasing popularity of cloud computing and server less architectures, it is the first step towards proposing interconnected independent HTM CLA units in an elastic cognitive network. Thereby it can provide an alternative to deep neuronal networks, with theoretically unlimited scale in a distributed cloud environment.

KEYWORDS

Hierarchical Temporal Memory, Cortical Learning Algorithm, HTM CLA, Actor Programming Model, AI, Parallel, Spatial Pooler.

1. INTRODUCTION

Popular Artificial Neural Networks employ commonly supervised learning based on strong mathematical principles.Such principals are efficient to solve specific kind of problems, but they operate in a way, which is not well aligned with the way the brain might work. Similarly, Recurrent Neural Networks are increasingly closing in to model of the biological functioning of parts of the brain. Unlike the brain, which typically operates in an unsupervised way, concepts like RNN and DNN apply explicitly supervised learning techniques. Hierarchical Temporal Memory Cortical Learning Algorithm (HTM CLA) [1] is an algorithm aiming to replicate the functioning of neocortex [2]. It incorporates theSpatial Pooler algorithm responsible for learning spatial patterns by using Hebbian learning rules [3]. The Spatial Pooler organizes population of neurons in structures called mini-columns. Every time the same or a similar pattern recurs, synapses between the input neural cells and mini-columns strengthen their permanence (weight)

DOI: 10.5121/ijaia.2020.11407

value [4]. With this principle, the Spatial Pooler can learn a spatial pattern after very few recurs of the pattern. Similarly, synapses can "forget" a learned pattern if it does not occur for a long enough time. The Spatial Pooler connects mini-columns in the higher-level entity called a cortical column. Similarly, to Spatial Pooler, the HTM CLA implements the algorithm called Temporal Memory, which is responsible for learning sequences. The Temporal Memory algorithm can also be activated inside of the cortical column.

Cortical columns can be connected into hierarchically organized network, which provide more cognitive capabilities like invariant representation, pattern- and sequence-recognition. The original HTM CLA was implemented in Python as a part of the NUPIC framework developed by Numenta [5]. C++ and JAVA implementations of HTM CLA are also available.[6]Because many of the modern enterprise applications are typically implemented in .NET with an increasing demand for cross-platform (Linux, Windows and MacOS) support, an implementation of this algorithm is required in the software industry to avoid inefficient and costly bridging between frameworks. As a preceding part of this work, HTM CLA was ported to C# .NET Core. [7] The current C# .NET Core version of HTM CLA aligns with JAVA implementations (which aligned with the original Python version [1]). It supports the singe-core Spatial Pooler and Temporal Memory algorithms. Processing of information in neurons inside of HTM is sparsely encoded as in biological neuronal circuits [8]. HTM CLA, in a nutshell, uses Hebbian learning rules [9] on binary sparse arrays represented as sequence of integers (0/1). In the context of memory consumption, the current representation of binary values in the form of integers is not the most efficient. Improving this is still work in progress and this kind of optimization is not in the focus of this paper. The assumption in the present work is rather that the current algorithm, when used with a high number of neurons, is highly CPU and RAM intensive. The simple Hebbian-Rule makes internal calculations efficient in comparison to other algorithms (i.e. back-propagation). However, the complex topology of the Spatial Pooler (SP) and the Temporal Pooler (TP) in HTM CLA with a high number of neurons and synaptic connections, internal inhibition, and boosting–algorithms, requires significantly more computing power and memory than available on a single commodity machine with multiple core processors and a GPU.

Current implementations across the mentioned frameworks maintains internally several matrices and maps (key-value pairs). For example, there is a matrix of synaptic connections between input neurons and cortical columns. To create a Spatial Pooler instance with 128x128 sensory neurons and 1024x1024 columns, the framework will have to create this matrix with 16,384 x 1,048,576 = 17,179,869,184 elements. In a .NET framework using a 64bit architecture operating system, the maximum possible array size of integers (32 bits) is 2,147,483,591, calculated as:

$$2147483591 \approx \frac{2^{\wedge}32}{2}$$

This is a half of the maximal integer value on 64 systems subtracted by a no significant internal framework overhead to hold and manage an array. This limit depends on many factors and fortunately, can be optimized. Nonetheless even with a very efficient optimization, the limitations from using a single node only will remain.

The current architecture of HTM CLA has, in this context, two limitations of interest: a limitation of the synaptic matrix size by available memory and long calculation times required for operations on synapses. Most papers related to HTM CLA indicate experimental work with 1024, 2048, 4096 (see [10]) and 16384 columns. As an example, in a case of 4096 columns and sensory input of 1024 neurons, which corresponds to an input image of 32x32 pixels, the SP algorithm will create 4,194,304 synapses when using global inhibition. The goal is therefore to design a distributed HTM CLA, which can run on multiple nodes and operate with any number

of columns (i.e. >100,000). HTM CLA is redesigned for flexible horizontal scale in highly distributed systems by using an actor model. Moreover, the concrete implementation should make usage of modern container server less technologies. Such a model should enable the flexible distribution of computation inside of a single HTM CLA unit or connecting multiple independent HTM CLA units in collective intelligence networks and provide an alternative to deep neuronal networks. This work is the first one about the parallel HTM Spatial Pooler on multiple nodes with the Actor Programming Model. With the increasing popularity of cloud computing and server less architectures, this work is the first step towards proposing interconnected independent HTM CLA units in an elastic cognitive network and can provide an alternative to deep neuronal networks, with theoretically unlimited scale in a distributed cloud environment. This paper specifically targets the redesign of a single Spatial Pooler unit.

Section 2 describes the current state of the Actor Programming model and the Spatial Pooler. Sections 3 and 4 describe how the Spatial Pooler was improved for scale with help of Actor Model reasoning.

2. THE ACTOR PROGRAMMING MODELAND CORTICAL COLUMN COMPUTATION

Object-oriented programming (OOP) is a widely accepted and approved and familiar programming model. One of its core pillars is encapsulation, which ensures that the internal data of an object is not accessible directly from the outside of the object’s context (object itself or external allowed objects). The context of the object is responsible for exposing safe operations that protect the invariant nature of its encapsulated data. Instructions of method invocations, when executed in parallel (i.e. on multiple threads or multiple physical machines) can be interleaved, which leads to race conditions. The consequence of this is that invariants will probably not remain in intact if threads are not coordinated. Common approach to solve this problem is to use locks around method execution. This programming technique includes a high number of concurrent threads and especially threads shared across multiple physical machines (in this work referred as nodes), very difficult to implement, it is very error prone and it shows bad performance [18].

In contrast, the Actor Programming Model a mathematical theory [6] and computation model, which addresses some of the challenges posed by massive concurrency. In this theory, the Actor is treated as the universal primitive of concurrent computation. An Actor is a computational unit that, in response to a message it receives, can concurrently run code. Motivation for this programming model in this work is the simple reasoning about concurrent computation. Moreover, both the HTM CLA and the Actor Model are biologically inspired models.

Designing distributed systems with this model can simplify compute balancing between actors deployed on different cores and physically distributed nodes. In this paper, a node is defined as a physical machine, which hosts multiple actors. There is also another issue today in this context. The CPUs are not getting faster. Most programming frameworks accepted in industry target multicore programming (single physical machine with multiple CPUs) and solve this problem efficiently. Multicore programming tasks are well integrated in modern languages and can be solved in a standardized way. For example, task/await pattern can be used in C#, promises in JavaScript etc.

However, distributing computational logic across many physical nodes running multiple CPUs remains a challenging task. This is where the Actor Programming Model in combination with

modern cloud technologies can be a promising approach, because of the ability to easily execute a huge number of independent well controlled computational units on remote physical nodes.

Because this work is related to the C# .NET Core implementation of Spatial Pooler and Temporal Memory, the Actor Model implementation must support the .NET Core platform. Following a review of several actor programming model frameworks, including Orleans Virtual Actors [12], Service Fabric Reliable Actors [13], and Akka.NET [14], it became apparent that none of them is suitable for this task. While they are very powerful, such solutions do not offer custom partitioning functionality [12][13], or they rely on some corporate-specific cluster [13]. As a result, the approach taken was to design and implement a lightweight version of the actor model framework. The most promising framework was Akka.NET [14], but it has shown insufficient results when it comes to networking under high CPU consumption. The Actor Model Framework proposed by this paper combines RPC and API style messaging to enable an easy and intuitive implementation. Message API style is used to physically decouple nodes in the cluster, which enables easier addition of new nodes while the system is running.

The lightweight Actor Model Framework was designed and implemented in C# .NET Core, because it integrates very easy with the named messaging system. Approach implemented in this framework provides a runtime for complex distributed computation on the set of distributed nodes. For example, in the context of HTM CLA a complex computation is defined by the set of partial computations for the cortical column (see chapter 2.3). The computation logic defined by should be spread remotely in the cluster of multiple nodes. By following biological principals described in chapter 2 every computation related to the cortical column runs isolated inside of the cortical column in the single layer. The result of the computation of the cortical column is then propagated to other cortical columns, areas, and so on. Because the cortical column behaves as a placeholder for mini-columns the computation set consists of mini-column specific partial computations

$$\mathrm{C}^m = \{c_1^m, c_2^m, .., c_Q^m\}, m \in \{0,1,..,\mathrm{M}\} \quad (1)$$

where Q defines the number of required computations for the mini-column and m denotes one of M mini-columns inside of the cortical column. Partial computations $c_i^m \in \mathrm{C}^m$ for the specific mini-column $m$are executed in parallel by using a function $\varphi(c_i^m)$ for all $m \in \{0,1,..,\mathrm{M}\}$.

One important feature of the Actor Programming Model is the location transparency. This is in general useful concept because it simplifies implementation. Unfortunately, complex algorithms, which require more control over the compute logic are difficult to align to this concept. As discussed, a computation C requires multiple computational steps, which can be executed sequentially, in parallel or both. Locally initiated computation part of an algorithm can be used orchestrate computation steps and correlate partial results to build the overall computational state. This concept enables the local part of the algorithm (client) to establish a computational session by placing the execution steps that belongs together on the dedicated node (server) in the cluster. This violates to some sort, the location transparency principal of the Actor Programming Model, but it introduces a partition concept, which is described in the next chapter. With this, the Actor Model algorithm will not automatically choose the physical node as previously named frameworks do. It rather uses a logical partition, which creates an isolated unit for execution of computational steps that belong together. Before the algorithm runs, the initialization process creates a logical representation of all partitions and sticks them to the physical nodes. The following algorithm illustrates how to run the execution of a cortical column computation, which

is used to calculate a single learning step of the Spatial Pooler for the given input of sensory neurons $\boldsymbol{i}$.

Algorithm 1

1 input: $\boldsymbol{i}$ // Set of neural cells. I.e. sensory input.
2 output: $\boldsymbol{o}$ // Set of neural cells. I.E. active cells of a mini-column.
3 configuration: N, M, P
4**begin**
5|$\boldsymbol{\varphi}$←actor($\{\mathrm{C}, \boldsymbol{i}\}, \mathrm{N}, \mathrm{P}, \mathrm{M}$); // Creates a orchestration function
6|**foreach**$\boldsymbol{c_i} \in \mathrm{C}$
7|| $r \leftarrow \varphi(c_i^m) \mid m \in \{0,1,..,\mathrm{M}\}$ //Runs the parallel compute remotely
8|| $\boldsymbol{o_i} \leftarrow S(\boldsymbol{o_{i-1}}, \boldsymbol{r})$ //Recalculate internal column state
9|**return**$\boldsymbol{o}$
10**end**

Algorithm 1 - Example of the HTM cortical column calculation

First, (line 5) the actor local system initiates the orchestration function $\boldsymbol{\varphi}$. During the initialization process, the local system will connect to the cluster of N nodes, associate P partitions to nodes and bind M mini-columns to partitions. Further (line 7), the function $\boldsymbol{\varphi}$willrun in parallel the remote computation on a dedicated physical node, which is defined by the associated partition. The function $\boldsymbol{\varphi}$does not implement the computation logic. It rather lookups the node of the partition and it routes the computation request to the concrete implementation by exchanging ordered messages and collects the partial result. The partitioning makes sure that every partial computation of the mini-column m runs in the same partition. With this, mini-column can keep the state in memory during computation time. Moreover, it implicitly creates a computational session that sticks to the node. Finally (line 8), the state building function S, which is usually not CPU intensive, recalculates the internal state of the cortical column.

The Actor Model Cluster is created by running a required number (theoretically any number) of multiple identical processes, which executes the same actor system code.

Algorithm 2

1 input: sys // Actor system initialization.
2 configuration: P, Ç, T
3 **begin**
4 | **do**
5 | | $(\boldsymbol{thread}, c_i, m) \leftarrow sys(\boldsymbol{msg}) | c_i \in \mathrm{C}, 0 < t < T$
6 | | $thread^t$(response, $\boldsymbol{c_i^m}$)
7 | | | **IF** $\boldsymbol{c_i^m} \notin Ç$
8 | |/$Ç \leftarrow Ç \cup [\, \boldsymbol{c_i^m}, \boldsymbol{\varphi}'(\, \boldsymbol{c_i^m})]$ // Move actor compute logic to the cache
9 | | /$actor \leftarrow Ç(\, \boldsymbol{c_i^m})$
10| | | $response \leftarrow actor()$ // Execute compute logic, send result.
11| **while**

Algorithm 2 - Execution of the compute logic in the cluster

Nodes in the cluster do not have any knowledge about each other. The system starts with the specified number of partitions P and maximal allowed concurrent actors T (lines 1 and 2). As a next, the listening for incoming computation requests is activated (line 5). Every received message must describe the requested computation c_i^m by providing the partial computation identifier i and the actor identifier m. The identifier m associates the computation with the partition. In the previously described example (see Algorithm 1), the m is the index of the mini-column. This suggests implementation of the mini-column as an actor. All messages sent to the actor with identifier m are ordered. As a next (line 6) the $thread^t$ is created. According to Actor Programming Model rules, there is a single instance of c_i^m running in a unique thread for every

m and i in the whole cluster. The compute actor with the partial computation logic defined by c_i^m is created in the line 8 by φ' or retrieved from the cache in the line 9. The result of the computation is returned to the orchestration function φ in the Algorithm 1, which has requested the partial computation.

With this approach, any computation (1) that can be partitioned, can be easily distributed across multiple partitions. The control over the placement of the computation is defined on the client side (Algorithm 1), where the set of all computation units (in this example set of mini-columns) is grouped into partitions. Partitions are associated to nodes, where they will be created and remain during the life cycle of the computation of C.

3. Redesign Of The Spatial Pooler

The original implementation of the Spatial Pooler (SP), as originally migrated from JAVA, supports in .NET Core the single threaded execution of computation. To be able to support parallel execution of HTM on multicore and multimode infrastructures, the SP algorithm was initially redesigned to support multiple cores on a single machine. The sensory input is defined as a set of neurons by input topology; Spatial Pooler uses an internally flattened version of input vector mapped to sensory neurons. Every input is represented as a set of values (0/1), where N is the number of all sensory neurons. This number is also known as the number of features. A flattened version of the input vector I_k is defined as:

$$I_k = \{0,1\}^{1xN} \qquad (2)$$

Columns are defined as a set of grouped cells, represented as a flat column array, where M is the total number of columns:

$$C = \{c1, c2, .., cM\}^{1xM} \qquad (3)$$

Most other neuronal networks typically connect every column to every input neuron. The Spatial Pooler connects to a subset of input neurons. This subset is defined by receptive field (RF) of the column. RF-array is defined as a subset of all column's synapses:

$$P_k^{1xCkq} | \; C_{kq} \in \{0,..., N\text{-}1\} \qquad (4)$$

The original design of SP maintains a single instance of the connection matrix λ. This matrix specifies whether the column ***Ci*** is connected to the sensory neuron ***Ij***. Indexes ***i*** and ***j*** are in the flattened versions of columns and sensory neurons respectively.

$$\lambda = \begin{pmatrix} x11 & x12 & \dots & x1N \\ x21 & & & \\ \dots & & & \\ xM1 & xM2 & & xMN \end{pmatrix} | x_{ij} \in \{0,1\}, i \in \{1, N\}, j \in \{1, M\} \quad (5)$$

Note, that the ***Ci*** column is connected to the sensory neuron ***Ij*** if the synapse permanence value is higher than the *proximal synapse activation* threshold. More details about the mathematical formalization of HTM CLA can be found in [15].

The matrix $\lambda$is one of artefacts inside SP, which tightly couples the design of algorithm to a single thread and prevents the SP to easily be redesigned for parallel execution. Refactoring this matrix will lead to different design of Spatial Pooler capable to run in distributed environment.

To be able to save memory and partition calculus of entire column space, this matrix has been was removed from original version of **SP** and semantically restructured as a graph of columns, in order to be able to save memory and partition calculus of the entire column space. Figure 1shows a single column inside of the column graph.

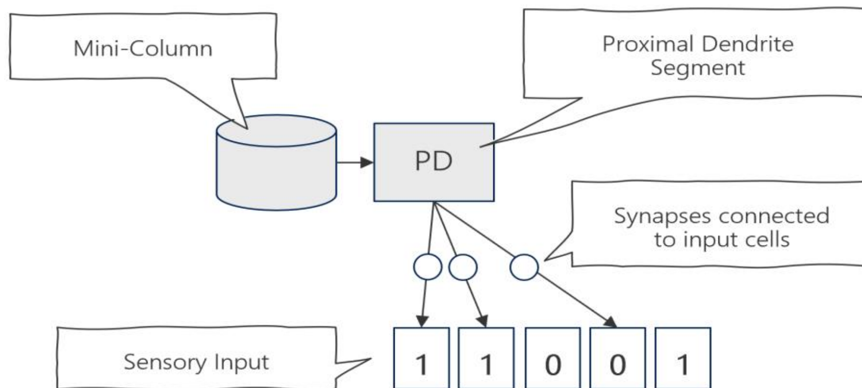

Figure 1 : Representation of a single column, which can be calculated on any node in the cluster. Every column holds its own dendrite segment with a set of synapses connected to sensory input defined by its Receptive Field.

Removal of this matrix enabled easier reasoning about single column calculus, as proposed by the Actor model approach. With this, it is possible to partition columns and to share memory across multiple nodes without of need to use distributed locks, which must be used to coordinate distributed calculation. Right now, three implementations of **SP** are implemented and considered:

- *Spatial Pooler single threaded* original version without algorithm specific changes.
- *SP-MT multithreaded* version, which supports multiple cores on a single machine and
- *SP-Parallel*, which supports multicore and multimode calculus of spatial pooler.

The Spatial Pooler algorithm consists in general of two stages inside of an application:

- Initialization
- Learning

Every named stage runs several specific calculations shown at *Figure 2*. For example, the Initialization stage performs a few initialization tasks internally. The Columns and synapse initialization stage creates a graph of columns with all required artefacts. The initialization stage is typically running once, and the learning stage is running for every input sample (online-learning). SP-MT and SP-Parallel versions of SP hold the parallel implementation of all listed algorithms as shown in Figure 2 at the right.

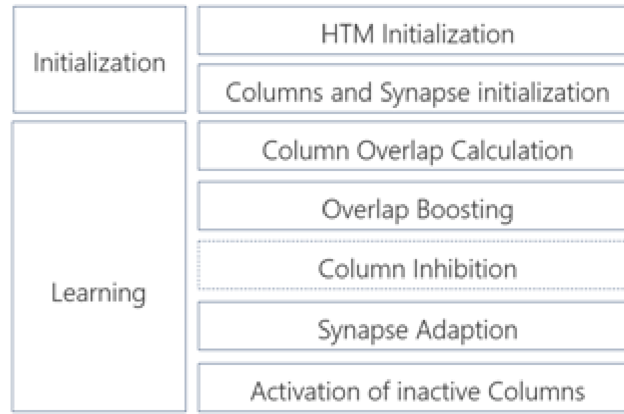

Figure 2. Spatial Pooler internal calculations shared across Initialization and Learning stage (left). Every stage consists of multiple calculations steps (right).

With the exception of *Column Inhibition*, which is currently shared across all three implementations, all other calculations are SP-version specific. In some future versions of SP, *Column Inhibition* and some other smaller algorithmic parts (not listed here) might also be redesigned. Redesign of **SP** targets two major tasks: partitioning of memory and partitioning of CPU usage. Memory consumption inside of SP is defined as follows

$$m = m(i_k) + \sum_u^M m_c(u) \quad (6)$$

$$m_c(u) = m_{s0} + \sum_w^S m_s(w) \quad (7)$$

where:

m – Overall memory consumption of a single SP instance, while calculation is running.
*i*k – Input vector sample: $k \epsilon \{0, N-1\}$
m(*i*k) - Memory occupied by an input sample.
m*c*(*u*)- Memory occupied by a single column.
***m*s0** - Memory occupied by column, excluding synapses. This memory is nearly the same across all columns. The difference is mainly influenced by holding references to a different number of synapses.

ms(w)- Memory inside of a column occupied by single instance of a synapse. Sum fraction of equation (6) corresponds to memory occupied by the proximal dendrite segment of a column with S synapses.

The original SP implementation and SP-MT both consume all memory ***m*** inside of a single process and it is therefore limited by physical memory of the hosting machine (node). For this reason, the first HTM Initialization step (see Figure 2) shares the same technique to allocate required memory in both named algorithms. SP-MT algorithm runs all calculations of every column on a single core by using C# technique called task/await.

4. THE PARTITION CONCEPT

The SP-Parallel algorithm performs partitioning of the column graph and distributes columns across multiple nodes. A partition is defined as a unit of computation with occupied memory.

$$P = p(Cpu, Mem) \tag{8}$$

Creating of partitions can be expressed by the following pseudo code:

```
createPartitions(numElements, numOfPartitions, nodes)

   destNodeIndx = 0

   numPerPart =
round(1+numElements /
   numOfPartitions);

   FOR partIndx = 0 to numOfPartitions
   OR min>=numElements

   min = numPerPart * partIndx;

   maxPartEl = numPerPart *
  (partIndx + 1) - 1;

    IF maxPartEl < numElements
     max = maxPartEl
    ELSE
     max = numElements - 1;

   destNodeIndx =
   destNodeIndx % nodes.Count;

destinationNode =
   nodes[(destNodeIndx++ %
   nodes.Count)];
  placement =
  (destinationNode,
   partIndx, min, max)
```

```
map.Add(placement)
ENDFOR

return map;
```

This code ensures that all columns (*numOfElements*) are uniformly shared across specified *numOfPartitions* and second, that all partitions are shared uniformly across all specified nodes. For example, if numElements = 90000 (columns), nodes = 3 and number of partitions = 35 then 34 partitions will contain 2572 elements and the last partition will be filled up with the remaining 2552 elements.

To understand how SP is changed to fulfil parallelization requirements, the following SP pseudo code must be refactored:

```
compute(input, learn)
   overlap = calculateOverlap(input)
   if(learn)
      boostedOverlap = boostFactors*overlap
  else
       boostedOverlap = overlap

  activeCols = inhibitColumns(boostedOverlaps)

  adaptSynapses(input, activeCols)

  activateInactiveCols()
```

To solve this, several parallelization approaches [16] have been analyzed. As a result, a dedicated and simple HTM column placement (distribution) algorithm has been designed based on the described partitioning mechanism.

Ideally, like neural parallelism [16] in the context of node parallelization, calculus for every column could be executed on a single core. For various practical reasons, placing of single column calculation on a core is understood as a special case. The partitioning algorithm rather places a set of columns in a partition, which is calculated on a single core across all nodes. In a generalized and simplified form, the overall calculation time can be defined as follows:

$$t = CN\, t_s + \frac{1}{N_c} \sum_u^{CN} t_u + CN\, t_g \mid m < m_\theta \qquad (9)$$

Equation (9) states that the theoretical amount of time required to calculate any step (see Figure 2) is defined as the sum of scatter time t_s needed to remotely dispatch calculation, the sum of all column-specific calculations t_u divided by the number of cores N_c and gather time t_g needed to collect results. Note that the calculation time for every column t_u is statistically different, depending on the number of connected synapses on the dendritic segment.

This equation holds as long overall memory consumption on the node does not exceed the maximally allowed threshold $m\theta$. If this limit is exceeded, the operation system will generate hard-page faults, which would cause memory reallocation to disk. Because this operational state would dramatically slow down performance, algorithms should take care of proper configuration to avoid this state.

Calculation time in such a distributed system is more complex as shown in the previous equation (9).

$$t = t_{rcv} + t_{sched} + t_{start} + t_{calc} + t_{persist} + t_{end} + t_{send} \mid m < m_{\theta} \quad (10)$$

trcv: Time for receiving of the message, which triggers calculation

tsched: Time required by system to schedule calculation. This is usually not a trivial task to coordinate lifecycle of partitioned calculations of columns in distributed system. All messages must be ordered and when possible, distributed locks shell be avoided. To solve this problem, already named a dedicated Actor Programming Framework was implemented on top of Microsoft Azure Service Bus [17], which provides messaging platform with many features required for this task. In this case message-session is used to ensure that all messages are ordered and dispatched to a single location, where calculation is performed. With this, no distributed locks are possible, and every partition calculation is running on a single thread. Because of this, *tsched* is taken out of algorithm and it remains a part of messaging system.

In this concept, one partition is defined as an Actor, sometimes called *partition Actor*. It physically owns 1-N columns (as shown in Figure 1) and it performs calculus over space P_k as defined by equation (3) owned columns only (see Figure 3). This space is much smaller than space defined by equation(4).

$$\{p_{k1}, p_{k2}, \ldots, p_{C_{kq}} \mid C_{kq} <= \mathrm{IN} \quad (11)$$

The Actor Model guarantees that there is only one calculation running on a single column partition across all cores in the cluster. Every partition Actor also holds the potential pool array as defined by the equation (3) and is capable of calculating the algorithm listed in Figure 2.

The distributed HTM algorithm **SP-Parallel** performs partitioning of Actors inside of the orchestrator node, which plays the role of a scatter operation. Running of calculations in actors on remote nodes is started and awaited on multiple threads inside of the orchestrator. Finally, the Actor model executes actors on nodes in the cluster and results are collected by the orchestrator node, which now plays the role of a gather operation.

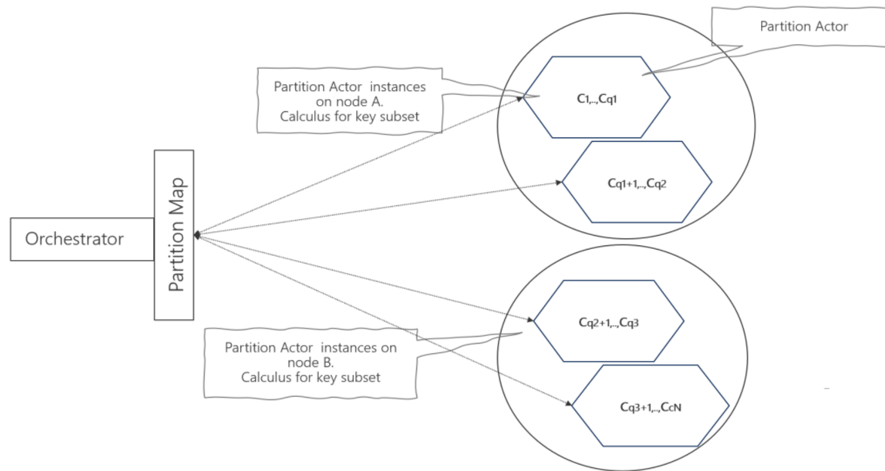

Figure 3 : Partitioned column space. Every partition is implemented as an Actor, which owns subsets of columns from the entire column graph. By providing the number of columns and nodes, the number of partitions can be automatically calculated or explicitly set.

To recap, in this partitioning concept the number of partitions and the number of nodes are known. That means, the SP-Parallel orchestrator code, which initiates placement of partitions must know nodes and can explicitly place a partition to the dedicated node. With this, theActor model framework can ensure that full calculation is executed as a sticky session on an initiated node. This improves performance and does not require a durable persistence of the calculation state, because the state is kept in the cache.

There is also another approach, which was tested, but it was excluded from this paper. In this (second) approach the orchestrator node does not have any knowledge about the number of nodes. This enables a simpler architecture of the system, but it requires to durably store the calculation state because, after every calculation step, the next step can be initiated on another node. For this reason, nodes must be able to save and load the state to and from durable storage, which adds significant performance costs. The second approach would perform better for shorter calculations with less memory.

5. EVALUATION

In this work several experiments have been created, which evaluate the performance of the modified Spatial Pooler algorithm. For all tests MNIST images of 28x28 pixels have been used. First, a single-threaded algorithm was compared against SP-MT (multicore single node SP) on different topologies (results shown for 32x32 columns).

Then the compute time of SP-Parallel was tested for a column topology 200x200 on one, two and three physical nodes. Finally, the performance of several column topologies was tested in a cluster of three physical nodes.

All tests have been executed on nodes with following "*commodity*" configuration on virtual machines in Microsoft Azure cloud:OS: Microsoft Windows Server 2016;
Processor: Intel(R) Xeon(R) CPU E5-2673 v4 @ 2.30GHz, 2295 MHz, 2 Core(s), 4 Logical Processor(s); Physical Memory (RAM):16.0 GB.

The first experiment was designed to demonstrate performance improvements of the SP-MT versus single-threaded algorithm. Remember, both algorithms were running on a single node. As input, MNIST test images with 28x28 pixels were used, and a cortical topology of 32x32 columns. Figure 4 shows the resulting sparse representation of the MNIST image.

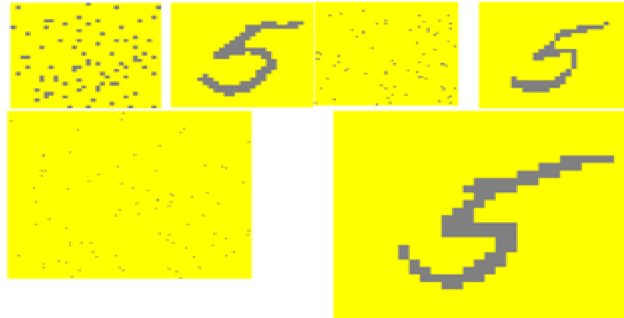

Figure 4. Sparse representations of an MNIST digit with different sparsity in column topologies 32x32 (top-left), 64x64 (top-right) and 128x128 (bottom). As an example, SDR on the top-right with column topology of 64x64 (4096 columns) occupies 2% (81) columns only.

Results shown in Figure 5 indicate that SP-MT is approximately twice faster than SP single-threaded on the indicated VM configuration.

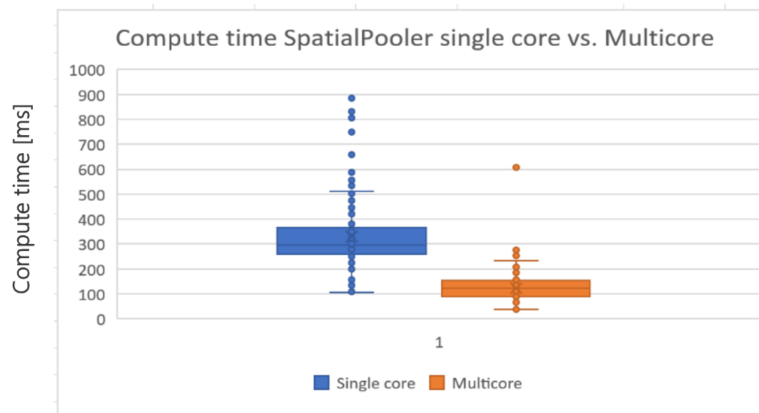

Figure 5 : Performance results, SpatialPooler single-core (SP) versus Spatial Pooler multicore (SP-MT) on a single machine. Tested on Surface Book2 with Microsoft Windows 10 Enterprise, Processor Intel(R) Core(TM) i7-8650U CPU @ 1.90GHz, 2112 MHz, 4 Core(s), 8 Logical Processor(s). MNIST 28x28 test image used 32x32 columns.

In the same experiment, the memory consumption (see Figure 6) and processing time in milliseconds in dependence of column topology were measured (see Figure 7).

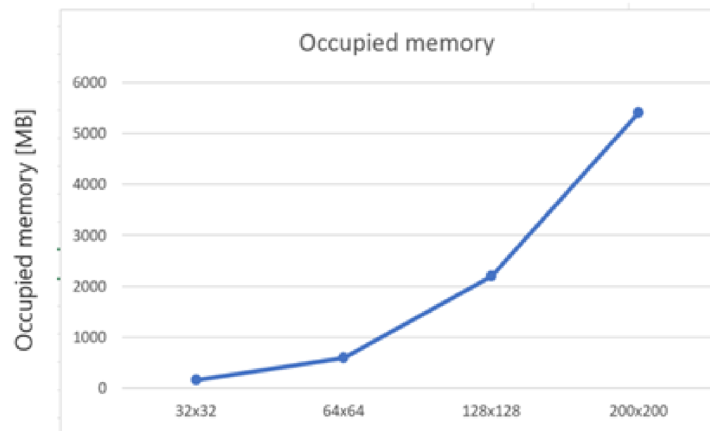

Figure 6 : RAM memory in MB, occupied during execution of the SpatialPooler-MT algoritm on a single node.

By using the same experiment with SP-Parallel instead of SP-MT, topologies with a higher number of columns and multiple nodes were tested. In this experiment learning of the MNIST image was measured on 1, 2 and 3 nodes. As shown in Figure 8 SP-Parallel on a single node needs nearly the same time as SP-MT. This is a good result because it approves that the Actor model framework does not spend significant time on the internal messaging mechanism.

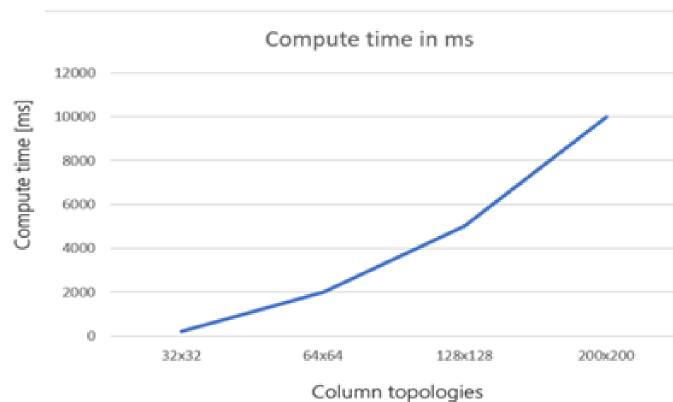

Figure 7. Spatial Pooler-MT compute time in milliseconds in dependence of column topology.

By adding more nodes to the cluster, performance increases as expected. The optimal number of partitions still must be investigated. As for now, to ensure that calculations on multiple partitions can run in parallel, it should be 2 or 3 times higher than the number of cores on all nodes.

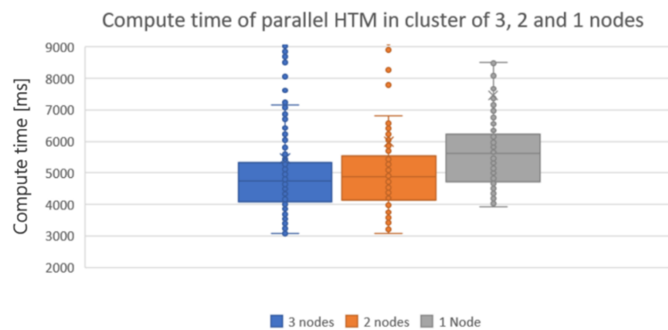

Figure 8. Learning time in [ms] of a MNIST image 28x28 with 200x200 Columns. Performance on a single node corresponds nearly to performance of SP-MT algorithm.

Figure 9 shows memory and CPU consumption on a single node, while calculation is running on multiple nodes. Independent of column topology, both memory and CPU consumption are shared across nodes in the cluster. As shown by the figure, during initialization time (step 1) memory is increasing, while allocating space for the columns and then it gets stable across the remaining repeating steps 2, 3 and 4 during the iterative learning process.

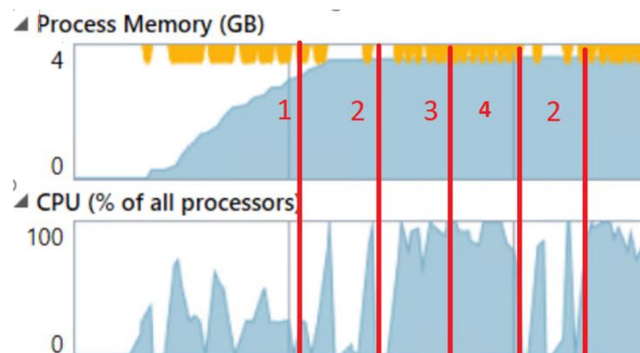

Figure 9. Process memory on a node while the computation of SP-Parallel is running. At the beginning 1 initialization stage is running, which allocates the required memory. Subsequently, stages 2,3 and 4 are related to overlap, synapse adaption and activation of inactive columns.

Finally, the system was tested to run up to 250000 cortical columns. This configuration allocates 196000000 synapses to sensory input (28x28 sensory neurons) on a proximal dendrite, by used global inhibition of the Spatial Pooler in this experiment.

In this experiment, every column connects to all sensory neurons, which corresponds to a potential connectivity radius of 100%. Topologies 200x200, 300x300 and 500x500 (250000 columns) columns were compared.

Additionally, the initialization time (see Figure 11) of the Spatial Pooler should not be underestimated. Allocating cortical columns and corresponding synapses takes significantly more time than compute time. Note that compute times for topology 200x200 with 20 and 15 partitions do not indicate significant differences. This is because the number of partitions is higher than the number of cores (3 nodes with 4 logical processors) in both cases.

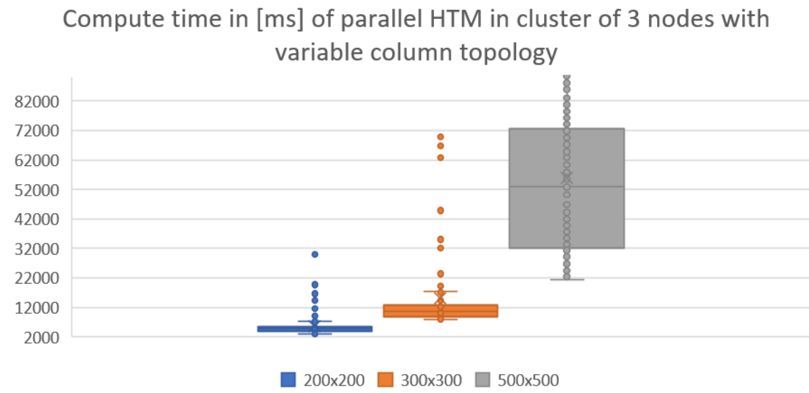

Figure 10. Compute time of SP-Parallel on three nodes in dependence of column topology.

Having a lower number of partitions than the number of cores would not use available CPU power and having a too high number of partitions would generate too many context switches and slows down the performance.

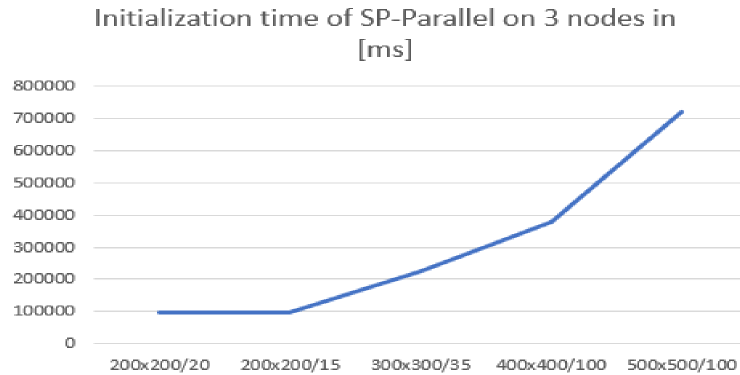

Figure 11. Initialization time in milliseconds of SP-Parallel in a cluster of 3 nodes in dependence of column topology. Used topologies are 200x200 with 20 partitions, 200x200 with 15 partitions etc.

All presented results were tested with the Actor model implementation, which sticks calculation to specific nodes without state persistence. Persistence of calculations would slow down calculation time. Some additional experiments show (not presented in this paper) that a single column takes approx. 700kb space persisted as JSON. Persisting of partitions described in this paper with thousands of columns would take gigabytes of space and would require a sophisticated algorithm to save and load such state in a very short time. This is one of the future tasks in this context.

6. Conclusions

Results in this paper show that HTML CLA can efficiently be scaled with an Actor programming model by using higher-level programming languages. Proposed algorithms SP-MT and SP-Parallel can successfully run on multicore and multi-node architectures on commodity hardware, respectively. SP-MT executes on a single node multicore architecture without the Actor Programming Model, which is rather used by SP-Parallel. The modified version of the Spatial Pooler can observe calculations for a high number of cortical columns in the simple Actor model cluster on commodity hardware. The building of algorithms natively in hardware by using lower-level programming languages might show better performance. However, using widely industrial

accepted and extremely productive higher-level programming languages enable easier use of compute power of modern cloud environments and enables this technology for use in a wide community of developers. The current version of SP-Parallel and SP-MT rely on the same code base, which will be step by step optimized, for example, in the way how internal sparse representations are implemented, especially when it comes to memory consumption. The goal of this work was to redesign Spatial Pooler for the Actor Programming Model by enabling it for easy horizontal scaling of the multiple nodes. The current implementation supports Windows, Linux and macOS on almost any kind of hardware.

With this approach, cortical regions can be widely distributed across many machines with acceptable costs and performance. Scaling of the Spatial Pooler algorithm is the first step in this research. **Spatial Pooler** produces sparse representations of inputs in the form of active columns. By following findings in neurosciences, generated *sparse representation* can be used as an input for the **Temporal Memory** algorithm. A next step in this research is the design of a distributed parallel version of the Temporal Memory algorithm and the design of a cortical network with the used Actor Programming Model approach. Such cortical networks will be capable to build highly interconnected cortical regions distributed in cluster. The high degree of connections should enable powerful sequence learning and more cognitive capabilities.

REFERENCES

[1] Hawkins Jeff, Ahmad, Subutai, Dubinsky Donna, "HTM Cortical Learning Algorithms," Numenta , 2011. [Online]. Available: https://www.numenta.com/htm-overview/education/HTM_CorticalLearningAlgorithms.pdf.

[2] Hawkins, Jeff and Ahmad, Subutai and Cui, Yuwei , "A Theory of How Columns in the Neocortex Enable Learning the Structure of the World," Frontiers in Neural Circuits, vol. 11, p. 81, 2017.

[3] Wulfram Gerstner1, Werner M. Kistler2, "Mathematical formulations of Hebbian learning," US National Library of Medicine National Institutes of Health, vol. 87, no. 5-6, 2002.

[4] J. H. Kaas, "Evolution of columns, modules, and domains in the neocortex of primates," PNAS, 2012. [Online]. Available: https://www.ncbi.nlm.nih.gov/pmc/articles/PMC3386869/.

[5] "NUPIC," NUMENTA, [Online]. Available: https://github.com/numenta/nupic.

[6] "HTM.JAVA," NUMENTA, [Online]. Available: https://github.com/numenta/htm.java.

[7] D. Dobric, "NeoCortexApi," 2019. [Online]. Available: https://github.com/ddobric/neocortexapi/blob/master/README.md.

[8] Luca A. Finelli, Seth Haney, Maxim Bazhenov, Mark Stopfer, Terrence J. Sejnowski, "Synaptic Learning Rules and Sparse Coding in a Model Sensory System," PlOS Computational Biology, 2007.

[9] Wulfram Gerstner, Werner M. Kistler, "Mathematical formulations of Hebbian learning," 2002. [Online]. Available: https://link.springer.com/article/10.1007/s00422-002-0353-y.

[10] M. W. W. Marcin Pietron, "Parallel Implementation of Spatial Pooler in Hierarchical Temporal Memory," Research Gate, 2016. [Online]. Available: https://www.researchgate.net/publication/301721307_Parallel_Implementation_of_Spatial_Pooler_in_Hierarchical_Temporal_Memory.

[11] C. Hewitt, "Actor Model of Computation," Cornell University, 2010. [Online]. Available: https://arxiv.org/vc/arxiv/papers/1008/1008.1459v8.pdf.

[12] Microsoft Research, “Virtual Actors,” [Online]. Available: https://www.microsoft.com/en-us/research/project/orleans-virtual-actors/.

[13] Microsoft Corporation, “Service Fabric Reliable Actors,” Microsoft, 2018. [Online]. Available: https://docs.microsoft.com/en-us/azure/service-fabric/service-fabric-reliable-actors-introduction.

[14] Akk.NET, “Akka.NET,” [Online]. Available: https://getakka.net/.

[15] James Mnatzaganian, Ernest Fokoué, Dhireesha Kudithipudi, “A Mathematical Formalization of Hierarchical Temporal Memory's Spatial Pooler,” 2016. [Online]. Available: https://arxiv.org/abs/1601.06116.

[16] M. L. W. H. Mark Pethick, “Parallelization of a Backpropagation Neural Network on a Cluster Computer,” ResearchGate, 2003. [Online]. Available: https://www.researchgate.net/publication/228549385_Parallelization_of_a_Backpropagation_Neural_Network_on_a_Cluster_Computer.

[17] “Cloud Messaging,” Microsoft, [Online]. Available: https://azure.microsoft.com/en-us/services/service-bus/.

[18] Y. a. T. G. Lubowich, "On the Performance of Distributed Lock-Based Synchronization," ecture Notes in Computer Science, vol. 6522, pp. 131-142, 2011.

AUTHORS

Damir Dobric CEO and lead software architect of DAENET Corporation specialized in software technologies with strong focus on Cloud Computing, IoT and Machine Learning. Microsoft Regional Director and Microsoft Azure Most Valuable Professional working with Microsoft on helping customers to adopt modern technologies. Teaching software engineering and cloud computing on University of Applied sciences in Frankfurt am Main. Researching at University of Plymouth at the School of Engineering, with the focus on artificial intelligence and distributed systems.

Andreas Pech Professor of Computer Engineering at Frankfurt University of Applied Sciences (Frankfurt, Germany), Dept. of Computer Science and Engineering, Head of Computational Intelligence Lab. He received his Diplom-Ingenieur degree in Electrical Engineering from TU Darmstadt, Germany in 1987 and his PhD (Dr. rer. nat.) from University of Mayence, Germany in 1990.Previously he worked as a group manager at Deutsche Klinik f Diagnostic, medical research department in Wiesbaden, Germany and as System Engineer at different companies.

Dr. Bogdan Ghita Associate Professor at the School of Engineering, Computing and MathematicsFaculty of Science and Engineering. Leader of network research within the Centre for Security, Communications, and Network research.

Thomas Wennekers Associate Professor in Computational Neuroscience. School of Engineering, Computing and Mathematics (Faculty of Science and Engineering)