MPI+OpenMP Tasking Scalability for the Simulation of the Human Brain

Human Brain Project

Pedro Valero-Lara, Raül Sirvent, Antonio J. Peña
Barcelona Supercomputing Center (BSC)
Barcelona, Spain
{pedro.valero, raul.sirvent@bsc.es, antonio.pena@bsc.es}@bsc.es

Xavier Martorell, Jesús Labarta
Universitat Politècnica de Catalunya
Barcelona, Spain
{jesus.labarta, xavim}@ac.upc.edu

***Abstract*—The simulation of the behavior of the Human Brain is one of the most ambitious challenges today with a non-end of important applications. We can find many different initiatives in the USA, Europe and Japan which attempt to achieve such a challenging target. In this work we focus on the most important European initiative (Human Brain Project) and on one of the tools (Arbor). This tool simulates the spikes triggered in a neuronal network by computing the voltage capacitance on the neurons' morphology, being one of the most precise simulators today. In the present work, we have evaluated the use of MPI+OpenMP tasking on top of the Arbor simulator. In this paper, we present the main characteristics of the Arbor tool and how these can be efficiently managed by using *MPI+OpenMP* tasking. We prove that this approach is able to achieve a good scaling even when computing a relatively low workload (number of neurons) per node using up to 32 nodes. Our target consists of achieving not only a highly scalable implementation based on *MPI*, but also to develop a tool with a high degree of abstraction without losing control and performance by using *MPI+OpenMP* tasking.**

***Index Terms*—MPI, OpenMP, Tasking, Simulation, Arbor, Human Brain**

I. Motivation

Today, we can find multiple initiatives that attempt to simulate the behavior of the Human Brain by computer simulations [1], [2], [3], [4]. This is one of the most important challenges in the recent history of computing with a large number of practical applications. The main constraint is being able to simulate efficiently a huge number of neurons (11 billions of neurons in the Human Brain) using the current computer technology. One of them is the called NEST Initiatives [5]. In particular, in this work, we focus on Arbor [6], [7], one of the tools of this initiative. The main motivation of this new initiative is to design and develop a modular brain simulator, which is able to adapt the simulator to the target platform. In the present paper, the authors explore for the first time the use of *MPI+OpenMP* tasking on homogeneous multi-core clusters. Although, we would like to see the work performed in this paper as a new back-end based on *MPI+OpenMP* of the Arbor simulator, it is important to note that we are not the developers of such simulator, and the *MPI+OpenMP* tasking is not integrated into the Arbor simulator.

One of the most efficient ways in which the scientific community attempts to simulate the behavior of the Human Brain consists of computing the next 3 major steps [8]: The computing of 1) the Voltage on neuron morphology, 2) the synaptic elements in each of the neurons and 3) the connectivity between the neurons. Previously to compute these steps, the network of neurons, the size and shape of the neurons and the connectivity between them, is created.

In this work, we focus on evaluating the *MPI* scalability for the simulation of the Human Brain. Our simulations include all the steps of the simulator. The contribution of the present work is twofold. i) we describe in detail the model followed by the simulator ii) we evaluate the use of *MPI+OpenMP* to minimize the impact of the *MPI* communication, and exploiting the efficiency of *OpenMP* tasking, not only to orchestrate the overlapping of communication and computation, but also to develop a tool with a high degree of abstraction without losing control and performance.

One of the most important challenges in this work is to implement an approach that can benefit from the strategies presented in the numerical model (simulator), with the minimal programing effort and the maximum performance. This is performed by using *MPI_AllGather* for the inter-node communication and *OpenMP* tasking for the intra-node computation and communication/computation overlapping. Both, *MPI_AllGather* and *OpenMP* tasking can introduce a non-negligible overhead regarding distributed-memory communication and thread scheduling on shared-memory. However, in this work, we will prove that this approach is not only an affordable and reduced-cost implementation in terms of programming, but also it is able to exploit efficiently the strategy implemented in the numerical model, achieving even ideal scaling.

This paper is structured as follows. Section II describes the physical problem at hand and the general numerical framework that has been selected to cope with it. In Section III, we present the specific parallel features for the resolution of the simulation, as well as the parallel strategies envisaged to optimally enhance the performance. Section IV shows the performance study to evaluate the scalability of our approach.

Section V goes through the state-of-the-art references and related work. Conclusions are outlined in Section VI.

II. Human Brain Simulator

The simulator is divided into two major tasks [9]: i) computation of neurons (Voltage capacitance and spikes), and ii) exchange of the spike events between neurons which are connected through synapses. In the model used by the simulator, the neurons can be seen as multi-compartment cables (see Figure 1) composed of active electrical elements. This model can benefit from several HPC capabilities such as vectorization, tasking, and communication/computation overlapping.

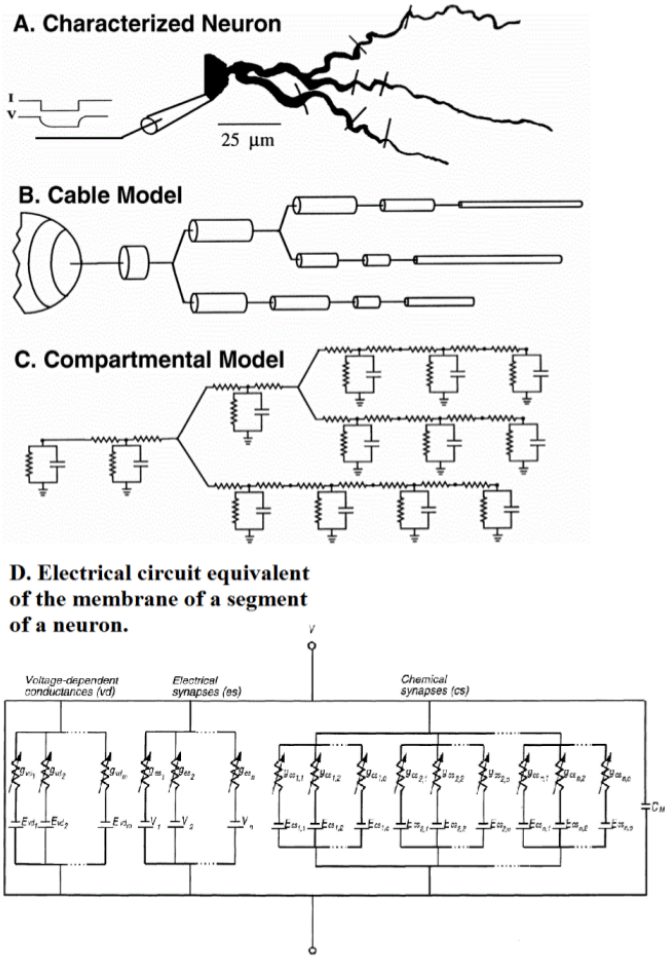

Fig. 1. Multi-compartment neuron model [9].

Next we describe the numerical framework behind the computation of the Voltage capacitance on neurons morphology [10], which is one of the most time consuming steps of the simulation. It follows the next general form:

$$C\frac{\partial V}{\partial t} + I = f\frac{\partial}{\partial x}(g\frac{\partial V}{\partial x}) \tag{1}$$

where *f* and *g* are functions on x-dimension and the current *I* and capacitance *C* [8] depend on the voltage *V*. Discretizing the previous equation on a given morphology we obtain a system that has to be solved every time-step. This system must be solved at each point:

$$a_i V_{i+1}^{n+1} + d_i V_i^{n+1} + b_i V_{i-1}^{n+1} = r_i \tag{2}$$

where the coefficients of the matrix are defined as follows:

$$\text{upper diagonal: } a_i = -\frac{f_i g_{i+\frac{1}{2}}}{2\Delta_x^2}$$
$$\text{lower diagonal: } b_i = -\frac{f_i g_{i+\frac{1}{2}}}{2\Delta_x^2}$$
$$\text{diagonal: } d_i = \frac{C_i}{\Delta_t} - (a_i + b_i)$$
$$\text{rhs: } r_i = \frac{C_i}{\Delta_t} V_i^n - I - a_i(V_{i-1}^n - V_i^n) - b_i(V_{i+1}^n - V_i^n)$$

The a_i and b_i are constant in time, and they are computed once at start up. Otherwise, the diagonal (d_i) and right-side-hand (rhs) coefficients are updated every time-step when solving the system.

The discretization explained above is extended to include *branching*, where the spatial domain (neuron morphology) is composed of a series of one-dimension *sections* that are joined at branch points according to the neuron morphology.

For the sake of clarity, we illustrate a simple example of a neuron morphology in Figure 2, divided by segments ($s_i, i = 1, \ldots, 4$) and nodes ($n_i, i = 1, \ldots, 6$) which connects the segments. It is important to note that the graph formed by the neuron morphology is an acyclic graph, i.e. it has no loops. The nodes are numbered using a scheme that gives the matrix sparsity structure that allows to solve the system in linear time.

To describe the sparsity of the matrix from the numbering used, we need an array ($p_i \;\; i \in [2 : n]$) which stores the parent indexes of each node. The pattern of the matrix which illustrates the morphology shown above is graphically illustrated in Figure 2.

The Hines matrices feature the following properties: they are symmetric, the diagonal coefficients are all nonzero and per each off-diagonal element, there is one off-diagonal element in the corresponding row and column (see row/column 7, 12, 17 and 22 in Figure 2).

Given the aforementioned properties, the Hines systems ($Ax = b$) can be efficiently solved by using an algorithm similar to Thomas algorithm for solving tridiagonal systems. This algorithm, called Hines algorithm, is almost identical to the Thomas algorithm except by the sparsity pattern given by the morphology of the neurons whose pattern is stored by the p vector. An example of the sequential code used to implement the Hines algorithm is illustrated in pseudo-code in Algorithm 1.

Once the voltage is computed, we compute the spikes. Basically, this consists of going through the different points on the neurons' morphology where there is a synapse (a connection between two neurons) and check if the voltage in these points is higher or lower than a given threshold to trigger or not a spike.

It is important not to forget one of the most important challenges into this model. This is the massive spike exchange

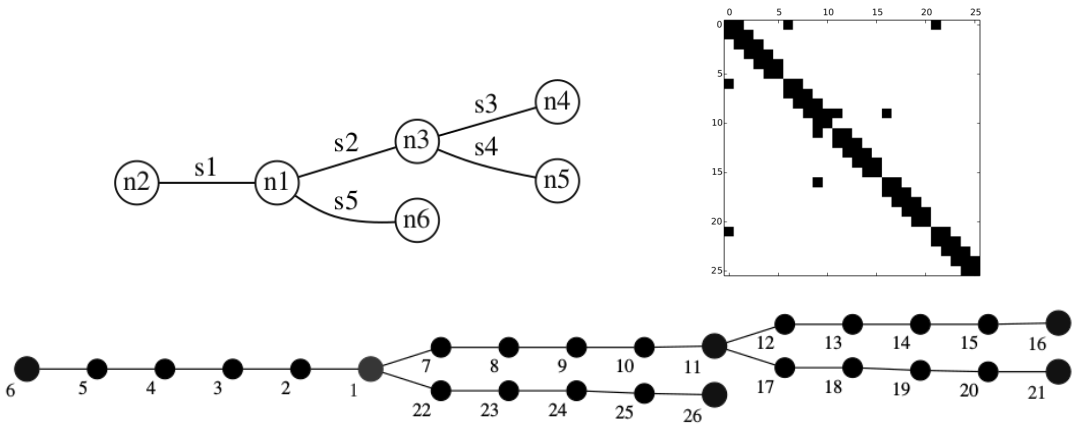

Fig. 2. Example of a neuron morphology and its numbering (left-top and bottom) and sparsity pattern corresponding to the numbering followed (top-right) [10].

Algorithm 1 Hines algorithm.

```
1: void solveHines(double *u, double *l,   double *d,
2:                 double *rhs,      int *p, int cellSize)
3: // u → upper vector, l → lower vector
4: int i;
5: double factor;
6: // Backward Sweep
7: for i = cellSize − 1 → 0 do
8:     factor    = u[i] / d[i];
9:     d[p[i]]   -= factor × l[i];
10:    rhs[p[i]] -= factor × rhs[i];
11: end for
12: rhs[0] /= d[0];
13: // Forward Sweep
14: for i = 1 → cellSize − 1 do
15:    rhs[i] -= l[i] × rhs[p[i]];
16:    rhs[i] /= d[i];
17: end for
```

between the neurons, which can be a problem on current distributed memory clusters due to the large difference between communication and computation speed, in particular if this communication has to be carried out using the *MPI_AllGather* routine, since one neuron can be connected (through the synapses) with a huge number of neurons. The strategy followed in this model to deal with this problem consists of the next ideas. The simulation time is divided into two different time-step factors, one local (*dt* in Figure 3) and one global (*Network delay* in Figure 3). In every local (*dt*) iteration, all the neurons are computed and the spike events are stored in one local buffer. There is no *MPI* communication at this level. After computing several local steps, we compute what we call global (*Network delay*) step. The MPI (*MPI_AllGather*) communication is carried out in this step. Basically, it consists of two tasks: first we store the spike events triggered, along one global step, into other local buffer, then we send/receive the information of the spikes to/from the rest of nodes using *MPI_AllGather*. In this way, all the nodes have the information about the spikes triggered along the simulation.

It is important to note that, although this model is in need

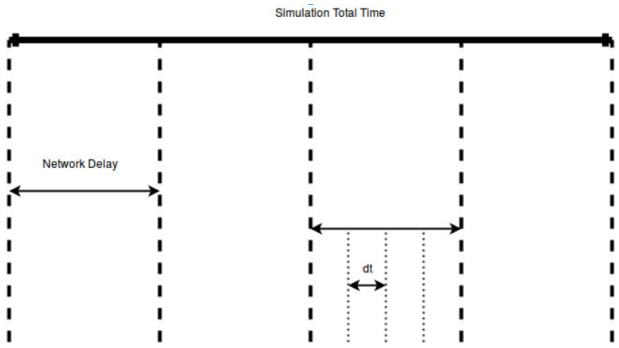

Fig. 3. Time-stepping used by the Arbor simulator [10].

of exchanging the information about the spikes every global step, there is an important difference between the data size sent in the spikes exchange (*MPI_AllGather*) and the operations computed along the local steps that compose one global step. For instance, let us assume that we have *N* neurons of size *M*, where every neuron has *S* synapses. Let us also assume that we compute *D* local steps per global step, and along these steps *Sp* spikes were triggered ($Sp <= S$). The operations performed every global step are $D \times (N \times ((8 \times M) + (S)))$, being $(8 \times M)$ the necessary operations to compute the Hines algorithm on a neuron of size *M* and *S* the operations performed for spike computation on synapses. While the data transfered is $N \times Sp$, the operations computed are $D \times N \times (8 \times M)$, which are musch bigger than the data size communicated. Depending on the simulation, these parameters can be very different; however, a commonly used value used for *M* [11] is about of $10 - 800$, for *S* is $200 - 1000$, and *N* depends more on the hardware (memory) limit and simulation time desired than on one specific range, but in our experiments we execute in the range of dozens to hundreds of thousands of neurons. *D* can be about of $10 - 20$.

III. Parallel MPI+OpenMP Tasking Simulator

After reviewing the main characteristics of the simulator, we focus on the parallelization. We decided to use *MPI+OpenMP*,

since both are standards, being the most extended and used programing model for distributed memory and shared memory computation, respectively. In fact, in the last years the concept of *MPI+X* is more and more popular, being the *OpenMP* standard the most popular and widely used candidate for the *X* unknown into the equation *MPI+X*.

As commented above, for *MPI* communication, we make use of the *MPI_AllGather* routine due to the particular nature of our target application (see above). Although this routine is among the least scalable routine in *MPI*, we will show that it is possible to achieve a good scalability by minimizing the cost of this routine thanks to both, the model used for the simulation and the parallelization implemented.

Recently, since *OpenMP 4.0* [12], it is possible to use tasking into *OpenMP*. Using tasking not only allows us to declare the dependences among tasks and let the compiler deal with the best distribution of the tasks on multi-core processors, but it also helps us to implement *MPI+OpenMP* codes very easily. For instance, we can encapsulate *MPI* routines into *OpenMP tasks*, which considerably simplifies the interoperability between both standards and the overlapping of *MPI* communication with *OpenMP* computing.

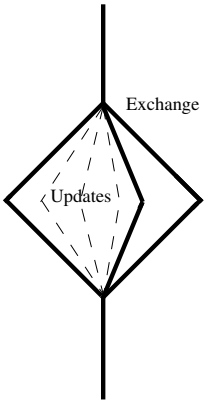

Fig. 4. Parallel model implemented for the simulator based on MPI+OpenMP tasking. Vertical/horizontal axes correspond to time/#cores.

Our target is to achieve the model graphically illustrated in Figure 4, overlapping the computation of the neurons (Updates in Figure 4) and the *MPI* communication (Exchange in Figure 4). Although there are many different ways to achieve this target, we intend an easy to implement and as much transparent (from the programmer's point of view) as possible approach that can yield a good scalability and performance. It is also important that our implementation makes use of standard programing models, minimizing (even avoiding) the programing effort to maintain, porting and/or tunning performance our application. Keeping this idea in mind, we parallelize our code using *OpenMP* tasking in the way that is illustrated by the Algorithm 2.

The parallelization is based on *OpenMP pragmas*. First we open (fork) a *parallel region* by using *#pragma omp parallel* every *global* time-step (*global_step* in Algorithm 2). After this, since we use *OpenMP* tasking, we must use *#pragma omp master*. At this level, we can create as many *OpenMP* tasks as we want. For the sake of clarity and due to the modularity of the code, every major step (*MPI* communication, Hines and Spikes computation) is implemented in separate files, we make use of nesting. In the first level of parallelism, we use two tasks, one for *MPI* communication and one for *OpenMP* computation. The last task (line 4 in Algorithm 2) instantiates one task (line 9 in Algorithm 2) per neuron, which computes the *Hines* algorithm and the *Spike* computation on one particular neuron. After the *#Neurons* for loop (line 14 in Algorithm 2) we must synchronize the tasks instantiated because of the data-dependences among different iterations of the *total_local_step* for loop. The parallel region is closed (join) every *global* time step. As we show in the next section, the overhead of fork-join, nesting, and tasks synchronization does not represent an important overhead with respect to the computational intensity to compute the Hines algorithm on a high number of neurons.

Algorithm 2 Parallel implementation.

```
1: while(global_step ¡ total_steps){
2:     #pragma omp parallel{
3:     #pragma omp master{
4:         #pragma omp task
5:         MPI_AllGather(Spikes);
6:         #pragma omp task
7:         for( local_step=0; local_step¡total_local_steps;
   local_step++){
8:             for( i=0; i¡#Neurons; i++){
9:                 #pragma omp task{
10:                Hines(Neuron[i]);
11:                Spikes(Neuron[i]);
12:                } //End omp task
13:            } //End for #Neurons
14:            #pragma omp taskwait
15:        } //End for total_local_steps
16:    } //End omp master
17:    } //End omp parallel
18:     global_step++;
19: }
```

As shown in Algorithm 2, "just" using 5 different *OpenMP* pragmas and one *MPI* primitive, we are able to have a portable, easy to maintain and optimized *MPI* code for the simulation of the Human Brain.

IV. Performance Analysis

The platform used in our experiments is a cluster composed of 39 NUMA nodes cluster with 2 sockets each, using Intel Xeon CPU E5649, see Table I for more details. Hyperthreading is not enabled.

In all the experiments we use one *MPI* process per node, and as many *OpenMP* threads as cores available (12 in our test platform). We use the default values for the parameters regarding the size of the neurons (450) and number of synapses per neuron (500) [1]. The simulations consist of computing 10 global iterations (*Network delay* in Figure 3) and 10 local

[1]For those experiments, which involve 100 neurons, the number of synapses per neuron is 100, achieving a fully-connected neuronal network

TABLE I
DETAILS OF THE ARCHITECTURE USED

Platform	Xeon E55649 (Westmere) at 2.53 GHz
Cores	2×6
On-chip Memory	L1 32KB (per core) L2 256KB (per core) L3 12MB (unified)
Main Memory	24GB DDR4
Compiler	gcc 6.2.0
Network	2 Infiniband QDR (4 Gbit/s each) non-blocking network

iterations (*dt* in Figure 3) per global iteration. The number of spikes triggered along the execution only depends on the parameters of the simulation, not on the number of nodes and cores per node. In particular it has a strong relationship with the number of neurons computed. In general the more neurons, the more spikes (see Figure 5). The number of spikes triggered (data size transferred) can be different every global iteration.

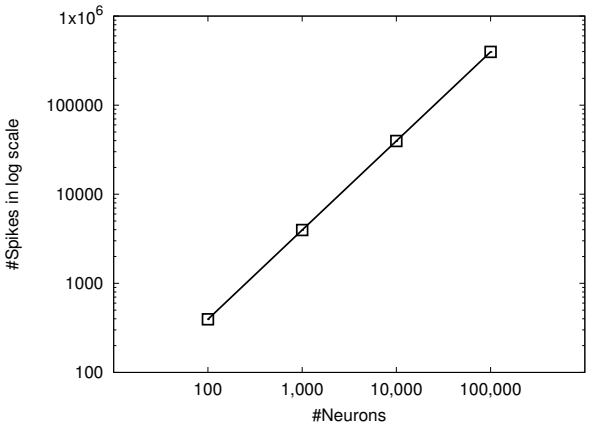

Fig. 5. Total number of spikes triggered in each of the experiments.

Next, we analyze the strong and weak scaling of our approach. First we focus on strong scaling analysis. In this case we have launched 4 test cases regarding the number of neurons computed (100, 1,000, 10,000 and 100,000). Figure 6 graphically illustrates the strong scaling analysis by increasing the number of nodes keeping constant the number of neurons computed in the simulation. The neurons computed per node depend on the number of nodes used. For instance, for a simulation composed of 100,000 neurons and executed on 2 nodes, half of the neurons (50,000 neurons) are computed on one node and the rest of neurons on the other node. In case of using 4 nodes, every node computes 25,000 neurons. As shown, our approach is able to achieve an ideal strong scaling, except in the case of computing 100 neurons on 32 nodes (384 cores), where the overhead of the *MPI* communication and *OpenMP* thread scheduling dominates against the computations on the neurons. This is because, in this case, the number of neurons per node is very low (3). Unlike the previous scenario, when computing 1,000 neurons on 32 nodes (31 neurons per node), we do not find this behavior, so that it is proven that it is not necessary to have a high workload per node to yield a good scaling.

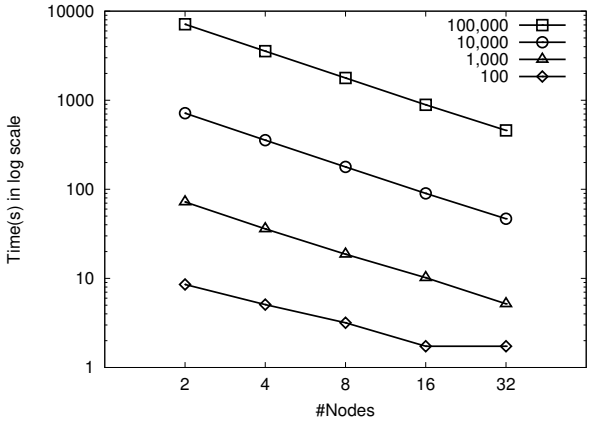

Fig. 6. Strong scaling analysis.

After analyzing the strong scaling, we analyze the weak scaling by increasing the number of nodes keeping constant the number of neurons computed in the simulation per node. In this case we have two test-cases regarding the number of neurons distributed per node, 1,000 and 10,000 neurons per node, respectively. As shown in Figure 7, our *MPI+OpenMP* tasking implementation is able to achieve an ideal weak scaling using up to 32 nodes.

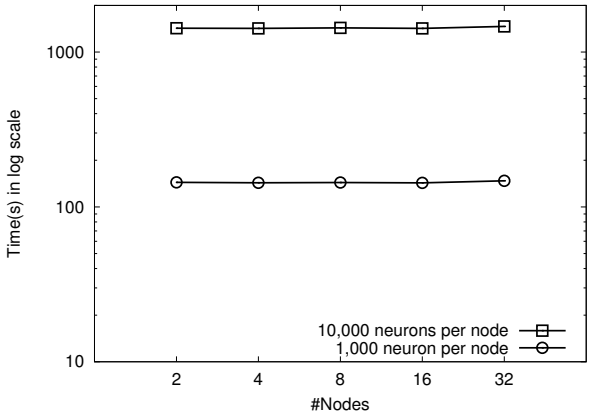

Fig. 7. Weak scaling analysis.

In order to perform a deeper analysis on the implementation presented, we have used the packages *Extrae+Paraver* [13]. *Extrae* is a dynamic instrumentation package to trace programs compiled and run using *OpenMP*, *OmpSs*, *pthreads*, *MPI* or a combination of the previous programming models (different *MPI* processes using *OpenMP* threads within each *MPI process*). *Extrae* generates trace files that can be later visualized with *Paraver*.

The first two traces (Figures 8 and 9) correspond to the execution of 10,000 neurons on 2 and 16 nodes, respectively. Both traces have the same time-scale to see the differences in time. In the traces, we can see three different colors which

Fig. 8. Trace of a simulation of 10,000 neurons on 2 nodes.

Fig. 9. Trace of a simulation of 10,000 neurons on 16 nodes.

correspond to *OpenMP* scheduling (in yellow), *OpenMP* execution (in blue) and *MPI* communication (in orange). In the traces, we see as many lines as cores used, 24 and 192 rows/cores for 2 and 16 nodes respectively (see Table I). Using *Extrae+Paraver* we are able to visualize easily the reduction in time achieved by increasing the number of nodes. In the 2 nodes trace (Figure 8) it is difficult to see the orange color (*MPI* communication), since the time consumed by *MPI_AllGather* is very low with respect to the time needed by the computation of the neurons just using 2 nodes. However, when using 16 nodes (Figure 8), it is easier to identify where the *MPI* communication is performed. Increasing the number of nodes, we increase the complexity of the *MPI* communication, the number of *MPI* calls, and then the time consumed by these calls is bigger. However, even when the use of *MPI_AllGather* supposes an increment in time when using a higher number of nodes, the time consumed by these calls is less than the 0.4% of the total execution time. Furthermore, these calls are overlapped (using *OpenMP* tasking) with the *OpenMP* execution, so that this increment does not affect the scalability. Only in extreme cases where we have a very low number of neurons per node (see Figure 6) the time of the *MPI* communication can affect the scalability. Due to the *OpenMP* scheduler, the MPI communication is done in one of the cores in each node, and the core responsible of the communication can change along the simulation. In both traces the predominant color is the blue (*OpenMP* computation).

To analyze the *MPI* communication and *OpenMP* schedul-

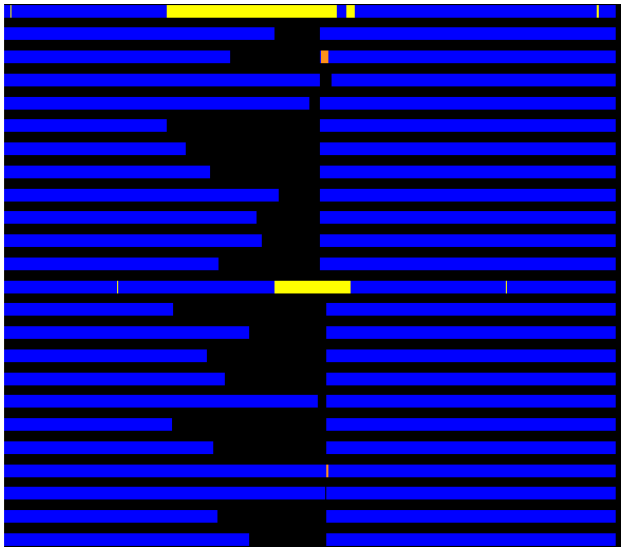

Fig. 10. Zoom on the trace for a simulation of 10,000 neurons on 2 nodes.

ing deeper, we zoom on in between the end of one global step and the beginning of the following global step for both traces (Figures 10 and 11). As in the previous traces, the time-scale is the same in these two traces. We can see more clearly the difference in time for *MPI* communication. The *MPI* calls are about $25\times$ bigger using 16 nodes with respect to using 2 nodes. However, this is still low with respect to the computing time and, as commented before, the *MPI* calls are overlapped with computation thanks to *OpenMP* tasking. Also, it is important to note that, unlike what we see in the previous traces (Figures 8 and 9), the first cores of each of the nodes also compute some operations on the neurons, so they are not only busy computing *OpenMP* instructions.

V. Related Work

It is possible to find many initiatives for the simulation of the behavior of the Human Brain by computers across the world, for instance in the USA [1], Europe [3], [2], [5] and Japan [4], [14]. Each of these initiatives is focused on the development of a set of tools for such target. In this work we have used one of them, Arbor [7]. In particular, we have focused on the scalability study using *MPI+OpenMP* tasking. This is the first study using both standards on Arbor code.

Most of the state-of-the-art references are focused on accelerating one of the most computationally expensive steps, that is the Voltage capacitance on neurons' morphologies. The standard algorithm used to compute the Voltage on neurons' morphology is the Hines algorithm [15]. This algorithm is based on the Thomas algorithm [16], which solves tridiagonal systems. Although the use of GPUs to compute the Thomas algorithm has been deeply studied [17], [18], [19], [20], [21], the differences among these two algorithms, Hines and Thomas, makes us impossible to use the last one as the last can not deal with the sparsity of the Hines matrix. Recently, in [11] it was proposed a new methodology to deal with a very high number of neurons on NVidia GPUs, achieving good scalability.

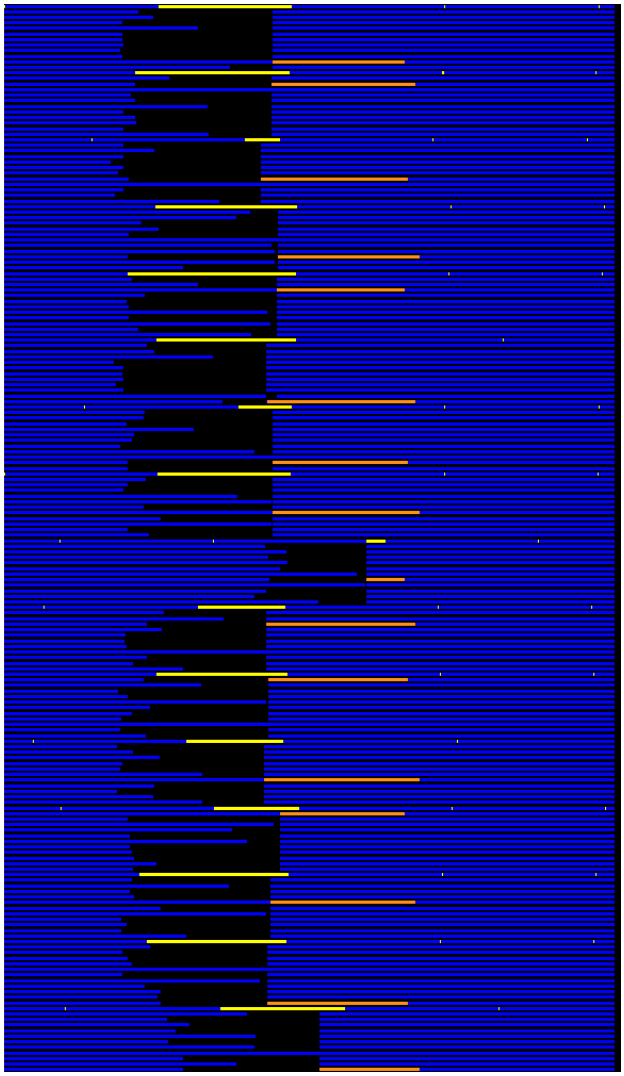

Fig. 11. Zoom on the trace for a simulation of 10,000 neurons on 16 nodes.

Unlike previous works, we focused on the parallelization and scalability on the *whole* application based on *MPI+OpenMP* tasking. In fact, this approach has been also used in other applications [22], such as the HPLinpack [23] tool used for the performance analysis of the TOP-500 list [24], achieving good results.

VI. Conclusions and Future Work

In the present work the authors proposed and evaluated an efficient (in terms of programmability) and optimized (in terms of performance and scalability) implementation for one of the most important challenges into the scientific computing community today, that is the simulation of the Human Brain. Given the results obtained and presented in this paper, the authors have proven the efficiency of using *MPI+OpenMP* tasking to achieve a good scaling.

Although *MPI_AllGather* and *OpenMP* tasking can introduce a non-negligible overhead regarding distributed-memory communication and thread scheduling on shared-memory, it was proven that this approach is not only an affordable and reduced-cost implementation in terms of programmability, but also it is able to exploit efficiently the strategy implemented in the Arbor tool, achieving even ideal scaling.

As future work, we plan to extend the work presented here to involve multi-morphology simulations, simulate a high number of neurons completely different, in terms of size and morphology, between them. This can suppose an additional challenge to deal with, in order to minimize the unbalancing found in the computation on the neurons, which can cause a non-neglected overhead, in particular for the *MPI* communication and the inter-node and intra-node synchronization.

ACKNOWLEDGEMENT

We would like to apreciate the valuable feedback and help provided by Benjamin Cumming and Alexander Peyser. This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 720270 (HBP SGA1 and HBP SGA2), from the Spanish Ministry of Economy and Competitiveness under the project Computación de Altas Prestaciones VII (TIN2015-65316-P) and the Departament d'Innovació, Universitats i Empresa de la Generalitat de Catalunya, under project MPEXPAR: Models de Programació i Entorns d'Execució Paral·lels (2014-SGR-1051). This project has received funding from the European Union's Horizon 2020 research and innovation program under the Marie Sklodowska Curie grand agreement No. 749516.

REFERENCES

[1] N. I. of Health, "Brain research through advancing innovative neurotechnologies (brain)," *https://www.braininitiative.nih.gov/about/index.htm*, 2018.

[2] E. P. F. de Lausanne (EPFL), "The blue brain project," *http://bluebrain.epfl.ch/*, 2018.

[3] E. C. Future and E. T. F. H. B. P. (HBP), *https://www.humanbrainproject.eu/*.

[4] H. Okano, E. Sasaki, T. Yamamori, A. Iriki, T. Shimogori, Y. Yamaguchi, K. Kasai, and A. Miyawaki, "Brain/minds: A japanese national brain project for marmoset neuroscience," *Neuron*, vol. 92, no. 3, pp. 582 – 590, 2016. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S089662731630719X

[5] F. Julich, "Nest initiative," *http://www.nest-initiative.org/*, 2018.

[6] ——, "Nestmc," *https://eth-cscs.github.io/nestmc/*, 2018.

[7] ——, "Arbor: A morphologically detailed neural network simulator for modern high performance computer architectures," *http://juser.fz-juelich.de/record/840405*, 2018.

[8] S. Diaz-Pier, M. Naveau, M. Butz-Ostendorf, and A. Morrison, "Automatic generation of connectivity for large-scale neuronal network models through structural plasticity," *Frontiers in Neuroanatomy*, vol. 10, p. 57, 2016. [Online]. Available: http://journal.frontiersin.org/Article/10.3389/fnana.2016.00057

[9] A. Peyser, "Nestmc: a prototype multicompartment neuronal network simulator for high-performance computing," 2017.

[10] B. Cumming, "Coreneuron overview," *CSCS - Swiss National Supercomputing Center*, 2010.

[11] P. Valero-Lara, I. Martínez-Perez, A. J. Peña, X. Martorell, R. Sirvent, and J. Labarta, "cuhinesbatch: Solving multiple hines systems on gpus human brain project*," in *International Conference on Computational Science, ICCS 2017, 12-14 June 2017, Zurich, Switzerland*, 2017, pp. 566–575.

[12] O. A. R. Board, "Openmp application programming interface," *http://http://www.openmp.org/*, 2018.

[13] G. Llort, H. Servat, J. Gonzalez, J. Giménez, and J. Labarta, "On the usefulness of object tracking techniques in performance analysis," in *International Conference for High Performance Computing, Networking, Storage and Analysis, SC'13, Denver, CO, USA - November 17 - 21, 2013*, 2013, pp. 29:1–29:11.

[14] RIKEN, "Brain/mids," *http://brainminds.jp/en/*, 2018.

[15] M. Hines, "Efficient computation of branched nerve equations," *International Journal of Bio-Medical Computing*, vol. 15, no. 1, pp. 69 – 76, 1984. [Online]. Available: http://www.sciencedirect.com/science/Article/pii/0020710184900084

[16] S. D. Conte and C. W. D. Boor, *Elementary Numerical Analysis: An Algorithmic Approach*, 3rd ed. McGraw-Hill Higher Education, 1980.

[17] P. Valero-Lara, A. Pinelli, and M. Prieto-Matías, "Fast finite difference poisson solvers on heterogeneous architectures," *Computer Physics Communications*, vol. 185, no. 4, pp. 1265–1272, 2014.

[18] P. Valero-Lara, A. Pinelli, J. Favier, and M. Prieto-Matías, "Block tridiagonal solvers on heterogeneous architectures," in *10th IEEE International Symposium on Parallel and Distributed Processing with Applications, ISPA, Leganes, Madrid, Spain*, July 2012, pp. 609–616.

[19] A. A. Davidson, Y. Zhang, and J. D. Owens, "An auto-tuned method for solving large tridiagonal systems on the GPU," in *25th IEEE International Symposium on Parallel and Distributed Processing, IPDPS, Anchorage, Alaska, USA*, May 2011, pp. 956–965.

[20] Y. Zhang, J. Cohen, and J. D. Owens, "Fast tridiagonal solvers on the GPU," in *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP, Bangalore, India*, January 2010, pp. 127–136.

[21] cuSPARSE, "Nvidia-cuda toolkit documentation," *http://docs.nvidia.com/cuda/cusparse/*, 2018.

[22] V. Marjanović, J. Labarta, E. Ayguadé, and M. Valero, "Overlapping communication and computation by using a hybrid mpi/smpss approach," in *Proceedings of the 24th ACM International Conference on Supercomputing*, ser. ICS '10. New York, NY, USA: ACM, 2010, pp. 5–16. [Online]. Available: http://doi.acm.org/10.1145/1810085.1810091

[23] I. C. L. (ICL), "Hpl - a portable implementation of the high-performance linpack benchmark for distributed-memory computers," *http://www.netlib.org/benchmark/hpl/index.html*, 2018.

[24] E. Strohmaier, J. Dongarra, H. Simon, M. Meuer, and H. Meuer, "Top-500 list," *https://www.top500.org/lists/*, 2018.