Advantages of the distributed Implementation of the Monte Carlo method in neurobiology simulation

Authors:
Alexander Kovalenko[1]; Oleg Kovalenko[1]; Dmitry Rusakov[2]; Kaiyu Zheng[2]
[1] AMC Bridge LLC, http://www.amcbridge.com
[2] University College London, http://www.ucl.ac.uk

1. Introduction

Matlab is often the tool of choice for scientist when it comes to creating and prototyping of numerical models. It's easy to use and powerful scripting language combined with rich library of functions and debugging capabilities makes for a platform that allows researchers to receive results quickly. At the same time when tasks are moved from the initial prototyping stage to a stage when more massive data should be processed Matlab may have its limitations especially when it comes to performance optimization, ability to run on a cluster or cost associated with multiple licenses needed for clustering.

This paper's goal is to demonstrate the performance benefits from using distributed algorithms and computational power of clusters and GPUs for Monte-Carlo method in neurobiology simulation.

2. Scientific background

Information processing and storage by the brain neural networks involve rapid release of signaling molecules such as glutamate from a neuron which diffuse across the interstitial space and bind specific receptors on the membrane of the target neuron. In most cases, this signal transfer occurs at specialized microscopic contacts, termed synapses, which thus constitute basic elements of the brain neural network "wiring", akin to computer circuits. However, there has been an increasing body of experimental evidence implicating other modes of information transfer in the brain, such as volume transmission of molecular signals where the diffusing signaling molecules binds not only to the target neuron but also other neurons and cells in the neighbouring vicinity. Organizational principles and basic biophysical determinants of this non-conventional type of signaling are only beginning to emerge.

In many neural circuits, activity-dependent changes in the network connectivity (which are hypothesized to be the basis of memory formation) involve NMDA receptors (NMDARs), a membrane protein which can sense the excitatory neurotransmitter glutamate with high affinity. It has emerged that NMDARs could be activated by glutamate escaping from relatively remote synaptic connections, thus enabling spatiotemporal integration of rapid signals in the brain, in parallel to the classical "wired" transmission. To understand fundamental properties of this mode of information

transfer we have been exploring a Monte Carlo model of a realistic, three-dimensional brain neuropil fragment which incorporates multiple synaptic connections embedded in a tortuous extracellular space. Our preliminary results predict that discharges of glutamate from individual synapses are very efficient (10-30 times more efficient than glutamate "leakage") in providing sustained activation of NMDARs across the space. The level of such activation could therefore retain information about local excitatory activity in the network.

3. Problem statement

Matlab implementation of Monte-Carlo method in neurobiology simulation is quite good only for relatively small simulations. The performance of Matlab application for large simulation is very poor. The purpose of the project was creating a fast and scalable application starting from existing Matlab application.

4. Solution

Two approaches were used to obtain the performance gains. The first one uses C++, OpenMP and MPI to run the application on the cluster, the second – CUDA to run the application on the GPU.
Each of the above approaches can be divided into the following steps:
1. Converting existing Matlab code to the C++ without any algorithm changes.
2. Algorithm optimization and modification to suite the chosen technology.
Since an existing serial algorithm of Monte-Carlo method for neurobiology simulation did not fit the paralleled model of execution, it was redesigned in the step 2.

5. Performance results

The below performance results are obtained from the sequential implementing the described in the above paragraph steps.
The Figure A shows the performance improvements after the direct translation from m-code to C++. It can be seen, that for relatively small simulations the speed-up is up to 4x, for larger simulations, the speed-up is 2x-2.5x.
The algorithm optimization from the step 2, gave additional 4x speed-up. As a result, the optimized C++ application is about 10x faster than initial Matlab simulation.
The Figures B, C demonstrate the scalability of the distributed implementation using both: C++ MPI and C++ OpenMP + MPI schemes for two different simulations (100K iterations of 21 sites and 5K particles; 100K iterations of 21 sites and 200K particles).
As it can be seen from Figure B, the parallelization gives the following speed-up compared to the serial C++ code: 1.8x on 2 cores, 2.8x on 4 cores, 2.1x on 8 cores for C++ MPI scheme and 1.4x on 2 cores, 1.7x on 4 cores, 2.4x on 8 cores for C++ OpenMP

MPI scheme. Figure C gives the following speed-up: 1.4x on 2 cores, 1.5x on 4 cores, 2.4x on 8 cores for C++ MPI scheme and 1.4x on 2 cores, 2.4x on 4 cores, 4.6x on 8 cores for C++ OpenMP MPI scheme.

The Figure D demonstrates the performance comparison of running the C++ MPI, C++ OpenMP MPI and CUDA applications for the larger configuration file – CUDA application is 4.2x faster than C++ OpenMP MPI app and 8.1x faster than C++ MPI app.

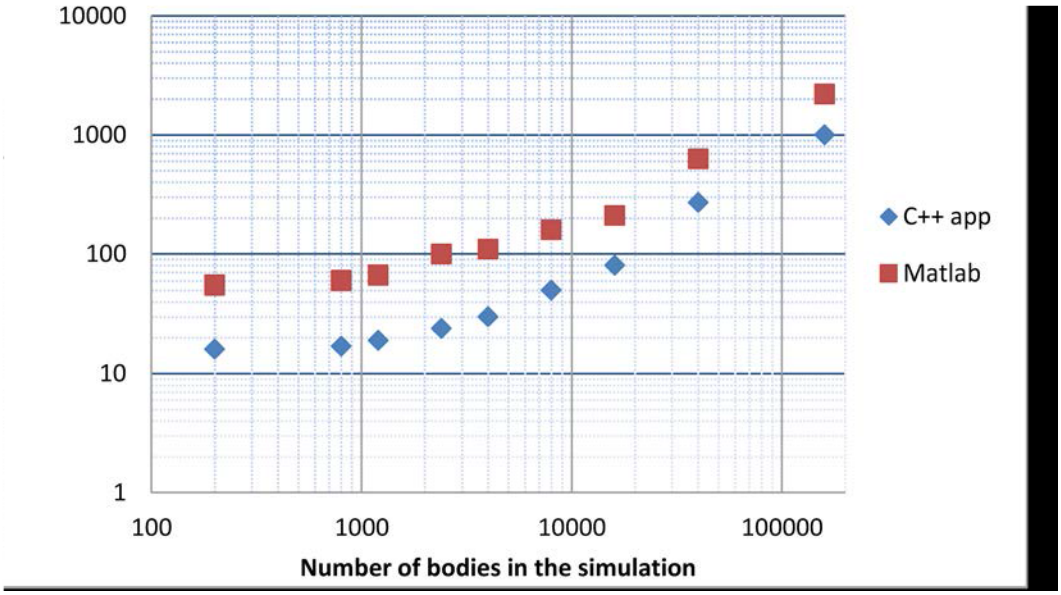
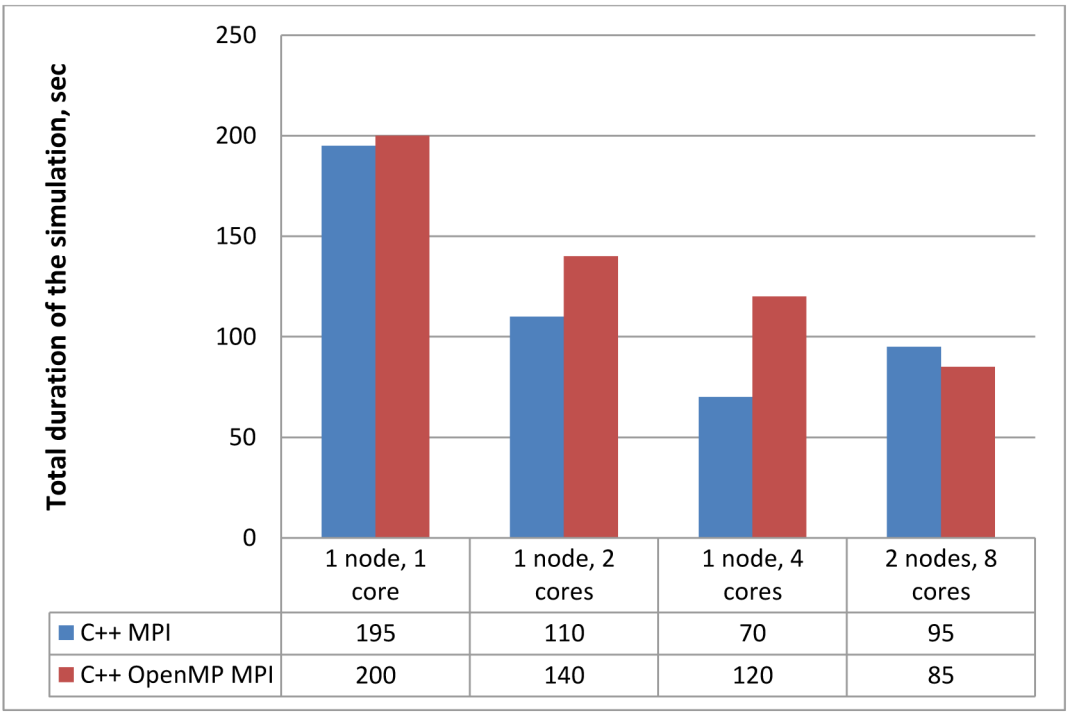


Figure A



	1 node, 1 core	1 node, 2 cores	1 node, 4 cores	2 nodes, 8 cores
C++ MPI	195	110	70	95
C++ OpenMP MPI	200	140	120	85

Figure B

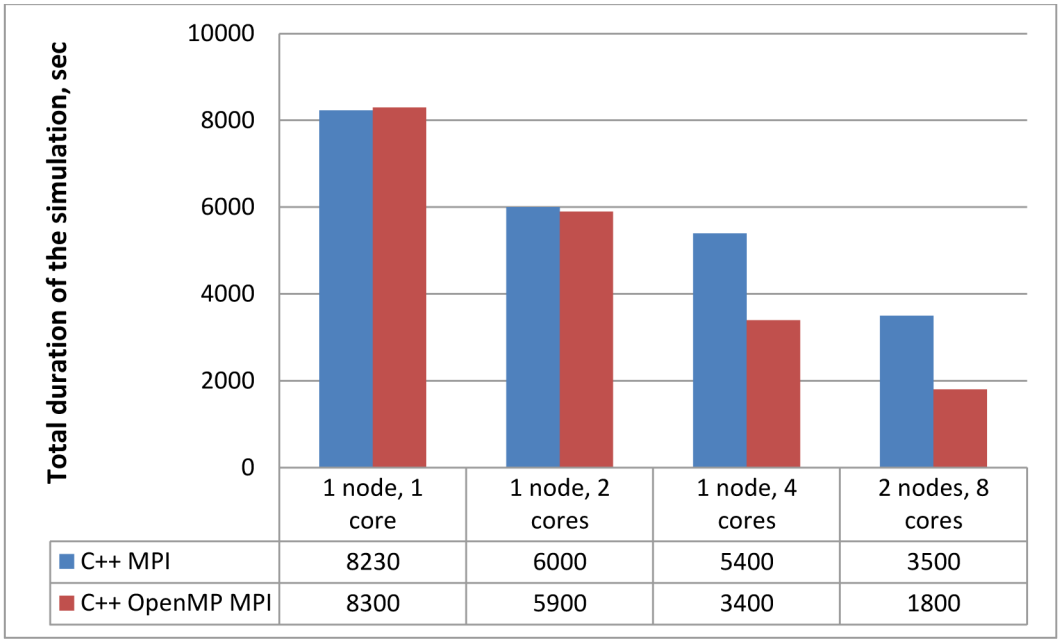


Figure C

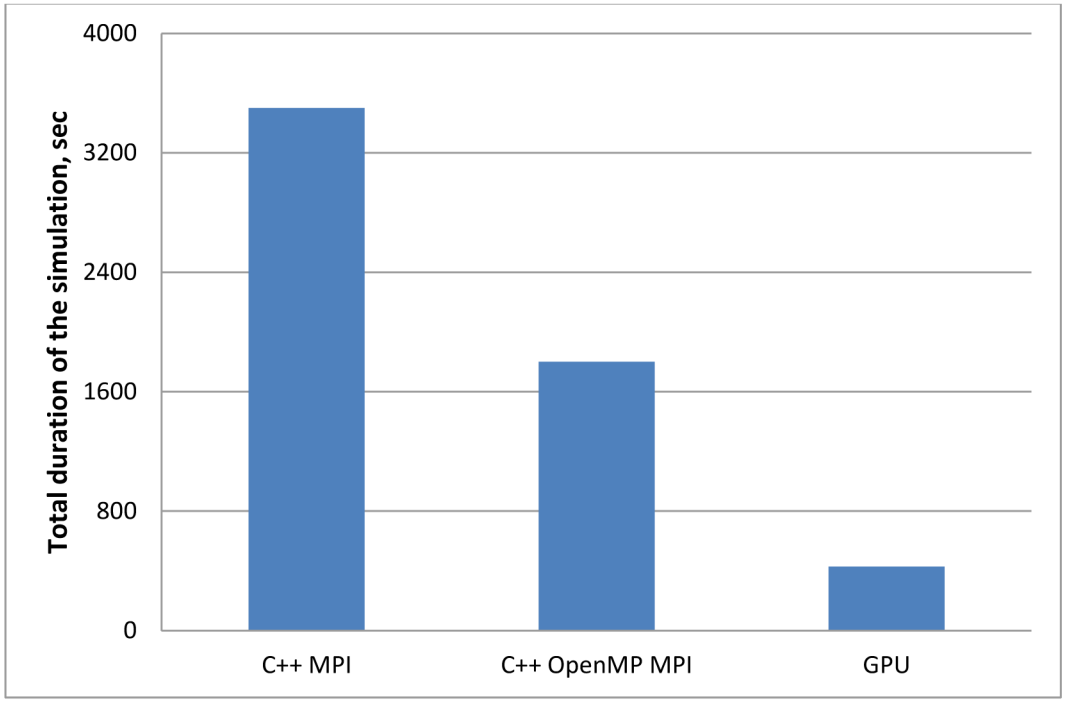


Figure D

The implemented C++ MPI and C++ OpenMP MPI applications were run under dual node quad core Intel Xeon E5440 2.83GHz 4 GB RAM cluster. The CUDA application was run under NVIDIA GeForce GTX 560M. Test execution time was measured over multiple runs and then averaged.

6. Conclusions

Since Matlab provides a very powerful scripts for mathematical modeling it is a very common tool to create any prototype. But very often such simplicity has its drawback – the created application is implemented in a suboptimal way.

Algorithm optimization and implementation it using C++ can give a huge speedup. The further parallelization and running on cluster gives even more performance gains.

For relatively small simulations we can see that for C++ MPI application the execution on single quad core node is even faster than execution on both nodes. This can be explained by the fact that MPI communication is much faster on the shared memory system (if the program is executed only under single node, then all used memory is shared) than under distributed memory system (case, when both nodes are used). Since each iteration of the first simulation is very light, the overhead of MPI communication kills the performance gains from using 4 extra cores. However, for large simulation, the distributed algorithm scales quite well.

Using OpenMP (to parallelize the work inside the node) together with MPI (to provide communications between nodes) reduces the number of communications between nodes. In the case when the number of particles is large (and hence the number of communication between nodes is large) the performance improvement is tangible: for the large simulation usage of the OpenMP + MPI scheme gives 2x speed-up on 2 nodes, 8 cores cluster comparing to C++ MPI implementation. Note, when the number of communications is low, MPI – based application can be more efficient than the OpenMP + MPI.

For the simulation with relatively small number of particles the GPU version gives poor results. But when the number of the particles grows, GPU usage can give the power of a cluster in a single desktop.

As a result, below is the list of recommendations for choosing approach for implementing the app:

- Select Matlab for prototyping and/or for applications which are not performance critical. This approach allows you to minimize the total time needed for creating the app, but requires expenses on Matlab license(s).
- Select sequential C++ for applications which will be actively using by lots of users and require good performance. This approach is cheap for experienced developer, but requires more time for creating the app comparing to the above one.
- Select C++ MPI for applications which will work with a large amount of data and use algorithms easy enough to be parallelized. This is expensive development but sometimes this can be the only option.
- Select GPU if your app needs to have extremely high performance and will not work with large amount of data and third-party libraries. This approach is extremely expensive for development and has lots of limitations.