NestMC: a prototype multicompartment neuronal network simulator for high-performance computing

Alexander Peyser, Simulation Laboratory Neuroscience –
Bernstein Facility for Simulation and Database Technology
Institute for Advanced Simulation, Forschungszentrum Jülich
Jülich, Germany

Description of the Code

NestMC [3] is a prototype simulator for neuronal networks composed of morphologically detailed neurons. This new code is being designed for the new generation of HPC infrastructure composed of massively parallel and heterogeneous architectures. Planned architectures include 'normal' non-vectorized CPUs, vectorized CPUs such as KNL, GPUs and other boosters such as FPGAs.

The code is composed of two fundamental elements (see Fig. 1C): the computation of multicompartment neurons as passive cables with active electrical elements including ion channels, and the exchange of resultant spike events between local and remote neurons connected at synapses. The computation of neural cable properties is amenable to the various HPC parallelization, vectorization and booster capabilities, while massive spike exchange is a problem of efficient communication, node local distribution and interleaving with computational tasks.

The code has been developed using 'Modern C++' [11], specifically the C++11 standard [10]. Using the metaprogramming capabilities of contemporary C++ in order to efficiently implement pluggable backends (Fig. 1A&B), we are developing a framework for switching solvers and developing kernels optimized for each targeted computational backend. This principle applies equally to threading and communication modules: serial, OpenMP [7], TBB [8] and HPX [5] threading models are possible, while communications can in principle be handled locally, handled using MPI [4], or in the future possibly handled by HPX [5]. This approach also limits the required external libraries to C++ STLs and the specific capabilities required for a given architecture. For BlueGeneQ (BGQ), this approach has limited us to the clang MPI compiler [2] which sufficiently implements the C++ 11 standard for the current NestMC codebase.

Since a functional TBB library is not currently available for BGQ, for the this Extreme Scaling Workshop we have used the OpenMP backend as well as tested a thread pool implementation developed using C++ STL threads. However, the current OpenMP backend does not implement the ability to interleave communications with computation, thus resulting in an extra barrier and a serialization bottleneck during computation. This motivated the development of a C++ thread pool library implementing a minimal subset of TBB task based

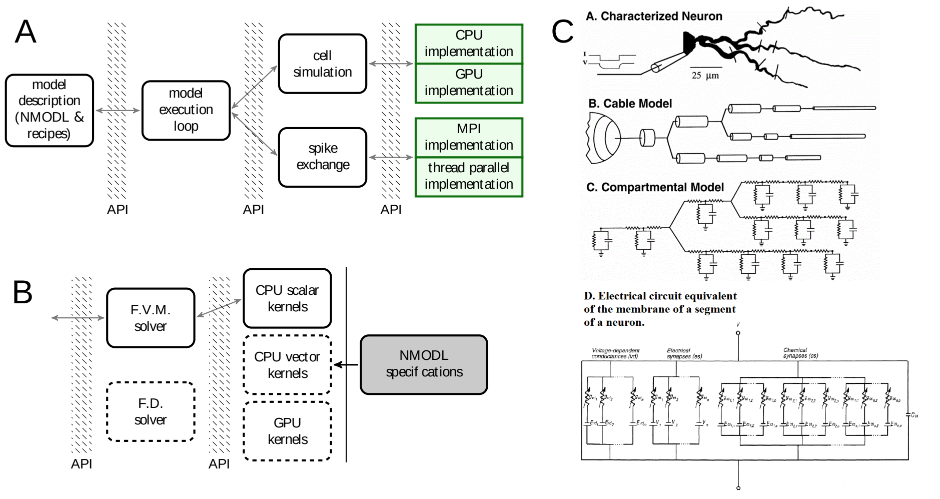

Figure 1: **A**: API architecture. NestMC models can be C++ recipes, NMODL compartment descriptions and other DSLs. The model execution loop is composed of a spike communication task and cell computation tasks; for OpenMP, communications and then computations are done, and for TBB or the threading pool model, all tasks are interleaved. **B**: Backend architecture. NMODL and other specifications are used to automatically build kernels for different architectures. Similarly, the architecture can use templated plugins for solvers and thread models. **C**: Underlying computational model for neurons. Each neuron is modeled as connected short cables that can be described via RC circuits with added active elements. Compartment models are implemented by the cell simulation in (A), and connections between neurons at synapses by the spike exchange in (A). [6], Creative Commons Attribution-Share Alike 3.0 Unported

parallelization. For communications, we have used the MPI-2 communication backend.

A randomly connected network for profiling purposes was defined using a small JSON file for each simulation. No parallel output has yet been developed; output was limited to a file describing aggregate behavior such as total number of spikes exchanged and a profiling output for analysis produced from the handwritten profiling framework within the application. Future extensions include a python frontend for describing networks and parallel IO for relevant measurements such as spike timings and voltage measurements.

Our goal with the workshop was to test whether our current architecture is scalable to 10k's of nodes. During development, profiling has been focused on node level and small network efficiency; code has been tested on clusters available at the Swiss Supercomputing Centre such as Piz Daint and Human Brain Project resources such as Julia and Juron available at the Jülich Supercomputing Centre at the Forschungszentrum Jülich. Before committing to a software architecture that will need to be scalable for the next generation of supercomputing equipment, testing the scalability of the communication algorithms for much larger systems using JUQUEEN was considered important, despite the limitations of available compilers and libraries.

Results

During the workshop, we began by comparing the pure MPI backend to the OpenMP+MPI backend. We used a network model with 8 cells/thread, 2000 synapses and 15 compartments/cell for relatively 'simple' Hodgkin and Huxley neuron models [1]. Thus the number of spikes communicated globally grows linearly with the number of nodes used. The timestep for integration was 0.025 ms, and the total simulation time was 50 ms. Since the minimum delay was 20 ms, this implies a total of 5 spike exchanges, each occurring every $1/2\,\mathrm{d}t$ (one half the minimum delay in the system for spike communication between neurons, which determines the communication timestep). Performance was measured with a built-in profiler for rank 0, assuming that rank 0 is characteristic of the system. For this profiler, the times for thread operations are divided by the total number of threads so that the sum of all times is equal to the total wall time.

During the communication phase for each time step, spikes events are exchanged with a call to `MPI_Allgather`. For pure MPI threading, this requires that every thread have a copy of the global spike buffer. Preliminary testing found that these memory demands outstripped available memory on JUQUEEN at around 8092 nodes.

The current OpenMP+MPI backend does not interleave communication with computations. Thus, the 'computational time' between communication time steps ($1/2\,\mathrm{d}t$) is the communication and spike exchange time plus the parallel computation time. For the tested configuration of 4 threads per rank, this means that 3 out of 4 threads wait for the spike exchange step. We found that under this configuration (see Table 1) good weak-scaling up to 2048 nodes, and better than 50% weak scaling up to 8092 nodes. The simulations continued to increase the number of neurons simulated faster than the increase in computation time for the simulation, up to 28672 nodes. However for the last increase from 16384 to full JUQUEEN, the gain in computation of 1.75-fold neurons costs a 1.52-fold increase in total computational time.

Time spent in MPI functions is negligible and 'event' time (computation time) is small and constant. This increase is dominated by 'communications' time, which in this case is the wait time required before the next parallel computation time begins. Thus, this $O(\text{nodes})$ term includes walking over all spikes produced in the simulation during the communication timestep (which grows with $O(\text{nodes})$). Communication time for the communication thread is scaled by 4 to account for computation threads' wait times.

Development has been focused on the TBB backend using task-oriented parallelization which allows interleaving communication with computation. The OpenMP measurements can be used as a 'worst-case scenario'. With interlaced jobs, the communication timestep is the maximum of communication time and the longest time spent by a thread doing cell simulation computations. For the

nodes	total	efficiency	communications	MPI	events
32	63.347753	1.000000	0.531771	0.000228	0.201864
64	63.659304	0.995106	0.829948	0.000225	0.200727
128	64.213774	0.986513	1.315700	0.000151	0.200540
256	65.061453	0.973660	2.294623	0.000391	0.201071
512	67.076769	0.944407	4.179481	0.000623	0.196835
1024	70.712083	0.895855	7.710518	0.001723	0.198596
2048	78.163655	0.810450	15.173421	0.000882	0.199653
4096	93.104669	0.680393	29.911602	0.002407	0.198081
8192	122.733964	0.516139	59.287230	0.013735	0.199668
16384	181.700285	0.348639	117.897971	0.018810	0.198835
28672	274.991168	0.230363	210.943898	0.018586	0.198643

Table 1: **OpenMP weak scaling tests**: 16 ranks/node, 4 OpenMP threads/rank. Efficiency is measured relative to 32 nodes. Communication time is scaled by 4 to represent the time spent by the communication thread and the 3 waiting computation threads. Since MPI times are negligible, the growth in total time comes from processing the global spike buffer, composed of all spike events from all nodes.

TBB backend, a more realistic worst-case scenario is interleaving communications with computations, using the limiting case of simple neurons with large numbers of spikes per time step which can thus outstrip the ability of the algorithm to hide communications. Our results during the workshop motivated the development during that time of a thread pool implementation using C++11 thread objects (underlain by 'pthreads') and implementing the subset of TBB used by NestMC currently, to allow testing such a scenario.

In Table 2, we were able to produce comparable results from 32 nodes up to 8192 nodes. For this case, we used 1 rank per node and 64 threads per rank, reducing memory demands 16-fold for the global spike buffers. In the table, we've scaled the communication time for the exchange task by 64 to represent the minimum computational time for the half minimum delay timestep. Again, MPI communication time is minimal, and event (computation) time is close to constant. Up to 1024 nodes, the results show that communication time is fully interleaved with computation time, leading to almost perfect weak-scaling. This interleaving effect as compared to OpenMP can be seen in Fig. 2: thread wait times as identified with the 'other' curve for OpenMP follows communication delay time for the entire range, but only begins to grow for C++ threads between 1024 and 2048 nodes. From 2048 nodes to 8192 nodes, weak-scaling falls off for C++ thread pool; at 8192 nodes, we find a scaling efficiency of 61% as compared to 52% for the OpenMP backend.

Conclusions

For OpenMP, the current architecture with 1 thread per rank handling all spike communications and exchange scales well up to 2048 nodes, and continues to give performance gains up to full JUQUEEN. Using threading pools that partially

nodes	total	efficiency	communications	MPI	events
32	66.923973	1.000000	4.906073	0.000050	0.189069
64	66.702474	1.003321	5.408203	0.000028	0.188245
128	66.976243	0.999220	5.839075	0.000061	0.189016
256	67.008755	0.998735	7.813766	0.000123	0.189148
512	67.444161	0.992287	10.579938	0.000145	0.188981
1024	69.349252	0.965028	14.346233	0.000110	0.188718
2048	75.911964	0.881600	20.477371	0.000165	0.189956
4096	87.261659	0.766934	33.028073	0.000113	0.189260
8192	110.194867	0.607324	57.930969	0.000185	0.190285

Table 2: **C++ threads weak scaling tests**: 1 rank/node, 64 pthreads/rank. Efficiency is measured relative to 32 nodes. Communication time is scaled by 64 to represent the minimum computational time between $1/2\,\mathrm{d}t$ (half the minimal delay, which determines the commiunication timestep), since the communication time cannot be distributed over all threads in the current implementation. If the time spent by all other (computation) tasks is exceeded by the serial communication time, all other threads must wait for the communication task to complete. Otherwise, $1/2\,\mathrm{d}t$ time is determined by the parallization of computation task. As in Table 1, since MPI times are negligible, the growth in total time comes from processing the global spike buffer, composed of all spike events from all nodes. However, the growth in timestep length starts with more nodes than with the OpenMP backend, given the ability to fully interleave computation with communications up to 1024 nodes.

implement the functionality of TBB, we see good weak-scaling up to 4096 nodes and can expect to see performance gains up to JUQUEEN scale. For more complex neuron models and morphologies which increase the ratio of computation time to communication time, weak scaling should be significantly improved; the cases tested are 'worst case scenarios' relative to production runs.

With this workshop, we identified the limits of weak-scaling on the current architecture. This motivated the development of a threading backend for architectures where TBB is not available. Since the communication time is dominated by processing the global spike buffers, a dry-run mode [9] has been developed taking advantage of this performance profile, which will allow us to estimate these results using negligible resources.

Acknowledgments

We would like to thank assistance from Yachao Shao and Wouter Klijn from the SimLab Neuroscience. Additionally, Ben Cumming from the Swiss Supercomputing Centre (CSCS) is the Lead Software Developer for NestMC, and Stuart Yates from CSCS is the Senior Software Developer and Mathematician for NestMC. Ivan Martinez Perez from the Barcelona Supercomputing Centre was responsible for the OpenMP backend.

The author would like to acknowledge the support of the Helmholtz Association through the portfolio theme “Supercomputing and Modeling for the

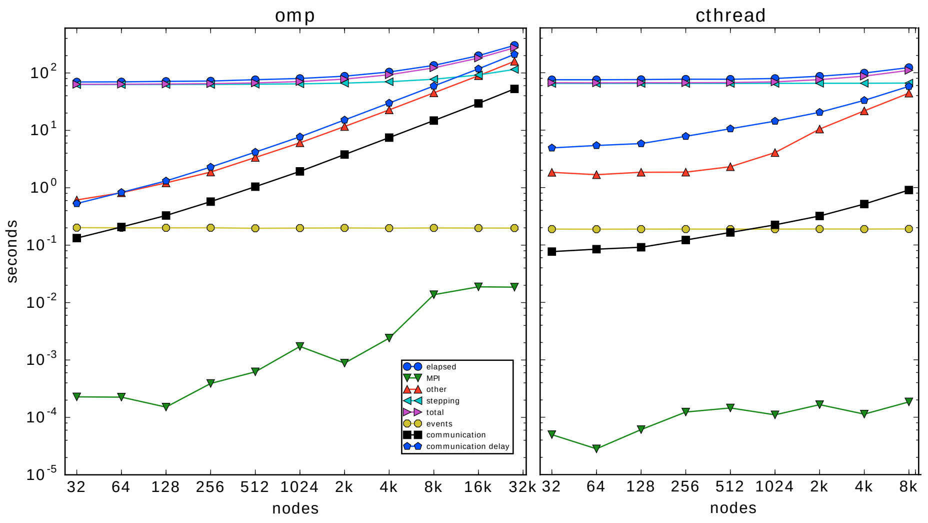

Figure 2: **Scaling comparison between OpenMP and C++ threads**: Total elapsed time (wall time for the entire job) is blue circles, total MPI time is green triangles pointing downward, other (thread wait times) is red triangles pointing down, total stepping time (time spent in computations and communications) is blue triangles pointing left, total simulation time is purple arrows pointing right, time spent in cell updates is yellow circles, unscaled communication time is black squares, and scaled communication time to predict possible communication delays is blue pentagons. For OpenMP curves on the left, the communication delay times are additive to all other parallel computations; for C++ threads on the right, communication delay times is the minimum for delays between communication steps. For OpenMP, 'other' follows 'communication delays'; for C++ threads, 'other' only follows 'communication delays' after 1024 nodes.

Human Brain" (SMHB). In addition, this project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No. 720270 (HBP SGA1).

References

[1] Proceedings of the physiological society. *The Journal of Physiology*, 117(suppl):1–14, 1952.

[2] Clang. `http://clang.llvm.org`.

[3] Benjamin Cumming, Sam Yates, Alexander Peyser, Wouter Klijn, Pedro Valero Lara, and Ivan Martinez. `https://github.com/eth-cscs/nestmc-proto`.

[4] Message P Forum. Mpi: A message-passing interface standard. Technical report, Knoxville, TN, USA, 1994.

[5] Hartmut Kaiser, Thomas Heller, Bryce Adelstein-Lelbach, Adrian Serio, and Dietmar Fey. Hpx: A task based programming model in a global

address space. In *Proceedings of the 8th International Conference on Partitioned Global Address Space Programming Models*, PGAS '14, pages 6:1–6:11, New York, NY, USA, 2014. ACM.

[6] Christof Koch and Idan Segev. *Methods in neuronal modeling: from ions to networks*. MIT press, 1998.

[7] OpenMP Architecture Review Board. OpenMP application program interface version 2.0, May 2002. www.openmp.org/mp-documents/cspec20.pdf.

[8] James Reinders. *Intel Threading Building Blocks*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, first edition, 2007.

[9] Wolfram Schenck and Susanne Kunkel. Dry-Run Mode for the Simulation Phase of NEST. NEST 100k Meeting (INM-6), Jülich (Germany), 2014.

[10] SO/IEC. Iso international standard iso/iec 14882:2014(e) – programming language c++. [working draft]. Technical report, Geneva, Switzerland: International Organization for Standardization (ISO), 2014. `https://isocpp.org/std/the-standard`.

[11] Bjarne Stroustrup and Herb Sutter. *C++ Core Guidelines*. 6 February 2017. `http://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines`.