Fully Implicit Parallel Simulation of Single Neurons

Michael L. Hines[1], Henry Markram[2] and Felix Schürmann[2]
Computer Science[1], Yale University, New Haven, CT
Brain Mind Institute[2], EPFL, Lausanne, Switzerland

January 1, 2008

Address correspondence to:
Michael Hines
michael.hines@yale.edu
1-203-250-0022

Keywords: computer simulation, computer modeling, neuronal networks, load balance, parallel simulation

Abstract

When a multi-compartment neuron is divided into subtrees such that no subtree has more than two connection points to other subtrees, the subtrees can be on different processors and the entire system remains amenable to direct Gaussian elimination with only a modest increase in complexity. Accuracy is the same as with standard Gaussian elimination on a single processor.

It is often feasible to divide a 3-d reconstructed neuron model onto a dozen or so processors and experience almost linear speedup. We have also used the method for purposes of load balance in network simulations when some cells are so large that their individual computation time is much longer than the average processor computation time or when there are many more processors than cells. The method is available in the standard distribution of the NEURON simulation program.

Introduction

Hines et al. (2008), from now on referred to as the previous paper, demonstrated the practical usefulness of splitting cells into two subtrees in compute-bound parallel network simulations when load balance is otherwise impossible. With a communication cost of a pair of MPI Send/Recv operations and with a message size of only two double precision values, per split-cell, per time-step, the neuron compartment equations could be solved with no change in accuracy or stability, and with essentially no increase in number of arithmetic operations during Gaussian elimination. The split-cell method is limited to at most a doubling of the number of processors on which a given simulation can be usefully performed but it is intriguing that computation time (as opposed to communication time) continued to dominate the runtime for our published test models even with the largest processor numbers for which load balance was attainable. Therefore the ability to split cells into more than two pieces is strongly desirable. The hope is that Gaussian elimination of the distributed compartment equations, which is necessary for numerical stability, will continue to be a small part of the runtime.

Rempe and Chopp (2006) present a predictor-corrector method which is very efficient but at the cost of an inexact solution at the branch points. Mascagni (1991) solves the linearized equations exactly by domain decomposition in which tridiagonal systems are solved for compartments between branch points and a dense linear system is solved for the branch points. Mascagni recommends the method for the simulation domain where there are few branch compartments relative to compartments between branches. Heglund (1991) presents a parallel algorithm for solving an unbranched cable and provides a comprehensive discussion of its complexity and numerical stability.

The parallel Gaussian elimination algorithm we have settled on is a fairly straightforward extension of the single split-cell algorithm presented in the previous paper. It differs from Heglund in its suitability to handling tree structures. It differs from Mascagni by including a constraint on splitting which reduces the size and complexity of the branch point matrix.

Our Gaussian elimination strategy divides the tree topology matrix into subtrees with the constraint that no subtree has more than two distinct connection points. In the numerical methods section we show that this constraint allows an attractive compromise between the flexibility of arbitrary division and the efficiency of the single split method. Since the computation and

communication cost is proportional to the degree of splitting, the method is most effective when the cell is split into the minimum number of pieces required for load balance.

Into how many pieces can a cell be usefully split? The increased communication cost imposed by the multisplit method is paid every time step. This is in very negative contrast to spike exchange communication, which needs to take place only at minimum spike delay intervals consisting of many time steps (e.g. Morrison et al., 2005). Furthermore, during a single cell simulation, all processors but one are necessarily idle while they wait for that processor to finish solving the reduced tree matrix. Clearly the method has to prove itself on a set of uncontrived models. This paper exhibits a practical heuristic for multiple piece cell division and load balancing for single cell simulation on multicore (shared memory) workstations and for network simulation on supercomputers when the number of cells is much less than the number of processors.

Methods

All simulations were carried out with the NEURON v6.1 simulation program (Hines and Carnevale, 2007). The "multisplit" functionality is available when NEURON is configured with the `--with-paranrn` option which requires pre-installation of an implementation of the the Message Passing Interface (MPI). On multicore machines we used MPICH2 (http://www-unix.mcs.anl.gov/mpi/mpich) configured with the `--with-device=ch3:nemesis` option which drastically reduces MPI communication time under shared memory compared with the default socket device.

Performance tests for single cell simulations were carried out on an Intel x86_64 dual-processor dual-core 3.2 GHz Dell Precision 490 and a SGI Prism Extreme with 32 1.5GHz Itanium2 processors and 300GB of shared memory. Network simulations were run on the 8192 processor (700MHz PowerPC 440) EPFL IBM Blue Gene/L.

Parallel single cell simulations to test the multisplit algorithm can be carried out on any single cell ModelDB model without modification and the examples we chose were the CA3 pyramidal neuron model of Lazarewicz et al (2002), the highly inhomogeneous CA1 pyramidal neuron model of Poirazi et al (2003), and a large Purkinje cell model by Miyasho et al (2001). The first network model used to test the multisplit algorithm in conjunction with

load balancing is that of Traub et al (2005) using code modified from the ModelDB section of the Senselab database (http://senselab.med.yale.edu) and is the same as described in the previous paper (gap junctions turned off). To compare the relative overhead of spike exchange, distributed multisplit neurons, and gap junction communication we also ran simulations with gap junctions turned on. The second network model is a version of the neocortical column simulations performed within the Blue Brain Project. Unlike the first network model, here, the multisplit algorithms help balance the processor load in the scenario where the number of cells essentially equal the number of processors but the cell complexities are varying substantially.

In all cases, the parallel models produce quantitatively identical spike patterns or voltage trajectories compared to their serial versions. Model code for parallelization of the single cell models and links to the network models are available from ModelDB with accession number 97985.

Numerical methods

Spatially discretized neuron equations have a tree topology in which the current balance equation of the i^{th} compartment has the form

$$a_p V_p + d_i V_i + \sum_c a_c V_c \;=\; b_i \tag{1}$$

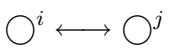
where the V_i, V_p, and V_c are the voltages at the end of a time step of this, the unique parent, and all the child compartments respectively. The a coefficients are constants depending only on the shape of the compartments and axial resistance. The d and b are evaluated using only parameters and variables known at the beginning of the time step in the i^{th} compartment. (A gap junction connecting compartments i and j solved by the modified euler method, adds a term to b_i that requires the value of V_j at the beginning of the time step and does not affect the topology of the equations.) On a serial machine, the number of operations required to solve the tree topology matrix equations is exactly the same as for a tridiagonal matrix representing an unbranched cable with the same number of compartments (Hines and Carnevale, 1997). The previous paper showed that this holds also when a cell is split into two pieces and solved on different processors.

To illustrate the Gaussian elimination operations we use a kinetic scheme diagram style in which an arrow (single or double sided) expresses the interaction directions, between the states, denoted as circles. Thus a two com-

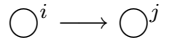
partment model with the equations

$$\begin{aligned} d_i V_i + a_{ij} V_j &= b_i \\ a_{ji} V_i + d_j V_j &= b_j \end{aligned}$$

is represented as



and, after a triangularization step that eliminates the effect of the V_j on equation i, we draw,

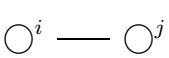


to express

$$\begin{aligned} d'_i V_i &= b'_i \\ a_{ji} V_i + d_j V_j &= b_j \end{aligned}$$

where $d'_i = d_i - a_{ji} a_{ij}/d_j$ and $b'_i = b_i - b_j a_{ij}/d_j$. Notice that equation i no longer depends on other compartments and back-substitution eliminates the effect of the V_i in equation j to result in the solved system

We need one other interaction notation due to a peculiarity of the NEURON program. That is, isolated subtrees are connected by a virtual wire instead of resistors so that we draw

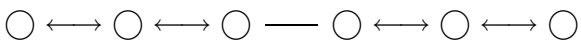


with no arrowheads to signify

$$V_i = V_j$$

Normally, when two NEURON sections

are connected together



the redundant nodes are combined

However, when the sections being connected are not on the same machine, both the connecting nodes are retained and eliminating the interaction term introduced by the wire is accomplished by adding the equations together (Hines et al, 2008).

Instead of attempting a mathematical description of the parallel Gaussian elimination method, it will be more understandable to use an example which shows how the major issues are resolved.

A prerequisite to the presented method is the division of the neuron morphology into subtrees so that no subtree has more than two connection points to other subtrees. This constraint follows only from practical considerations: Allowing only one connection point makes it impossible to divide a tree into many subtrees. It will become clear that allowing more than two connection points significantly increases the Gaussian elimination complexity.

[Figure 1 about here]

Figure 1 illustrates the compartment representation of a specific branched neuron along with its exemplary division into 8 subtrees (other divisions that satisfy the at-most-two-connection-points-to-other-subtrees rule are possible) connected by wires and which may be distributed in any manner on from 1 to 8 processes. As implied in the transition between Figure 1B and Figure 1C, when a subtree is disconnected from its parent tree, each subtree retains at least the portion of the shared node that includes the interaction term or terms from the adjacent nodes in the subtree. Terms in the split node involving channels and synapses may be distributed according to convenience. In practice they follow the section to which they belong.

[Figure 2 about here]

Gaussian elimination is carried out in several phases, the result of phase 1 is illustrated in Figure 2B and consists of triangularizing each subtree as much as possible using the optimum leaf to root ordering. For subtrees with only one connection point, this triangularization is complete. But for subtrees with two connection points, optimum triangularization stalls when it reaches any point on the path between the two connection points. The resulting submatrix along the path has, in fact, exactly the tridiagonal form of an unbranched cable and we call it the backbone path. The problem, of course, is that one cannot complete the triangularization from either end

of the backbone path unless one first eliminates terms from the connecting subtrees. This can be trivially done if all the subtrees connected to one end of the backbone path themselves have only one connection point (since their triangularization is complete) but not if one of the connecting subtrees has two.

The second phase is to transform the tridiagonal submatrix of each backbone path into a submatrix in which the backbone compartments are affected only by the two end compartments. The result is illustrated in Figure 2C and has an "N" topology in which interior compartments depend only on the ends but the two ends must be solved simultaneously.

During the third phase, each subtree sends the equations that still contain interaction terms, i.e., d' and b' of the triangularized root of each subtree that has only one connection point, and the 2x2 matrix elements and right hand sides of the backbone end points. Equations of compartments connected by wires, are added together and the result, illustrated in Figure 2D, is called the reduced tree matrix. The reduced tree matrix is solvable by classical optimal Gaussian elimination. The voltages are then sent back to the relevant subtrees, resulting in Figure 3E which is fully triangularized.

The final two phases are: back-substitute the two end compartments of the backbone path, Figure 3F, and complete any remaining tree back-substitutions.

The reasons for restricting the number of connection points on a subtree to two or less are now apparent. First, transforming the simple tridiagonal like tree into a topology with more than two endpoints greatly increases the number of operations needed. Two endpoints are bad enough — the transformation into an N topology of an n compartment backbone takes $2(n-2)$ divisions and $6(n-2)$ multiplications without moving triangularization forward at all. This, along with backbone back substitution, is the principle computational cost of the method, asymptotically increasing by 3/2 the number of divisions and by 8/3 the number of multiplications normally required by the backbone elements. Second, for m connection points on a subtree, a dense matrix equation of rank m must be transferred for addition to the reduced matrix. Third, the reduced matrix would not have a tree topology.

NEURON support for multisplit computation.

To support parallel simulation of individual neurons with the multisplit method described above, NEURON's ParallelContext class was extended

with the method

```
section ParallelContext.multisplit(x, sid)
```

which specifies that the compartment of the section containing position x ($0 <= x <= 1$) is connected by a "virtual" wire to all the other locations having the same positive integer split identifier, sid. The sid for a split point must be globally unique across all processes but is completely independent of other integer identifiers such as the spike source identifier or cell identifier used for spike exchange. Error messages are generated if the number of split points on a tree is greater than two or if the multisplit connection topology contains a loop, i.e., the reduced matrix is not a tree. Trees do not have to be in different processes in order to be connected by multisplit and that makes both load balancing and debugging easier. After every virtual connection is specified, the no-argument version of `ParallelContext.multisplit()` must be called by all processes to set up internal data structures and the MPI Send/Recv communication pattern. The no-argument call must be made even by processes with no multisplit cells in order to allow completion of several MPI collective calls needed to analyze the connection pattern.

To use multisplit, it is not necessary to change the code that defines cell types. It is usually easiest to create the entire cell in any process which needs any piece of the cell, disconnect the pieces, and throw away all except those pieces to be simulated in that process. For single cells, this roughly doubles the cell creation portion of the setup time and makes it independent of number of processes. But that increase is generally small compared to runtime savings. For networks, setup time usually continues to be inversely proportional to number of processors and is, in any case, normally dominated by the time taken to setup network connections.

A number of administrative issues become more complicated in a multisplit simulation, especially for networks, and some more or less general idioms have been developed to manage the extra complexity. First, divided cells make the notion of global cell identifier (Morrison et al., 2005) somewhat ambiguous. To make spike coupled network connections between undivided cells, the global cell identifier (gid, see Migliore et al, 2006, for an extended discussion of its role in a NEURON context) identifies both a whole cell and its normally unique spike detection location. In its role as spike detection location, the gid is used as the source of a NetCon and is internally critical in making sure a spike is delivered to the proper synapses. In its role as cell identifier, it is contingently employed by the user code to determine if the

cell exists in a process and, if so, to retrieve a reference to the cell object. For a multisplit cell, we use the old whole cell gid (call it the `base_gid`) for the spike detector and the pruned cell object where that spike detector exists. For the other pruned cell objects containing different subtrees, an idiom that has worked well in practice is to invent a

```
thisprocess_gid = base_gid + index*maxgid
```

where `maxgid` is greater than the largest `base_gid` and the `index` is the smallest piece index left in the pruned cell object. The virtue of this mapping is that it is easy to create functions that return the base gid from the associated gid on this process and vice versa. So, as spike detector, the base gid must be used in the call on the processor where the spike source location exists

```
pc.cell(base_gid, spike_source)
```

as well as the calls on the processors where the synapse targets exist

```
netcon = pc.gid_connect(base_gid, target)
```

where `pc` is a reference to a ParallelContext instance. On the other hand, retrieval of a target reference, or creation of the target synapse from `base_gid` and section name information, requires getting a reference to the pruned cell using the `thisprocess_gid` and, if it exists, checking whether the section exists in the pruned cell using one of the variants of

```
if (section_exists("section_name", cell_object)) { ... }
```

Other issues tend to have uniform solutions that have been encapsulated in standard library procedure calls. These include the determination of complexity, determination of split points and subtrees of a whole cell so that no subtree is larger than a maximum size, determination of a load balanced distribution of pieces on available processors, and pruning away subtrees not belonging to a given cell on a given process and performing the inter- and intra-processor reconnections with `pc.multisplit`. Note that an intra-processor reconnection could be accomplished in principle with the standard NEURON `connect` statement but at the cost of even further administrative complication.

Results

Load balance

As justified in the previous paper, load balance is the first priority for efficient compute-bound parallel simulation. We also showed that a simple proxy that assigns a complexity value to each compartment allows reasonably accurate prediction of processor computation time and thus makes static load balancing feasible. In that paper the severe restrictions on splitting required a special load balance algorithm. With the multisplit method, however, a cell may be split into pieces in a very large number of ways. Though the precise count is complicated by the constraint that a subtree can have no more than two connection points, the fact that a branch point with 2, 3, or 4 connected NEURON sections can be split (or left alone) in 2, 4, and 15 ways respectively, gives an indication of the degree of flexibility. Although the selection of cell split points during the process of load balancing is conceptually attractive, we chose the much simpler direct method of first dividing a cell into pieces so that every piece has less than some maximum complexity value, and then distributing the pieces according to the Least Processing Time (LPT) algorithm (Korf, 1998). LPT serially places the largest remaining piece on the processor with the least complexity. LPT is generally able to balance in the neighborhood of 1% if the maximum piece complexity is about a third of the average complexity per processor. The primary explanation for that success is that our maximum piece complexity cell division algorithm usually ends up with a good number of small pieces. Roughly, the cell division algorithm starts at the soma and examines the complexity of each subtree, keeping subtrees connected to the soma as long as the complexity is less than the maximum and disconnecting them if the complexity is greater. Disconnected subtrees are reconnected to each other if their sum is less than the maximum. For subtrees with greater than maximum complexity, we move one section away from the root and repeat the process.

[Figure 3 about here]

Figure 3 shows the cell division and LPT algorithm result for the CA3 pyramidal cell of the Lazarewicz model. The complexity of this 632 compartment neuron on an x86_64 processor is 76789. That is, the compartments contain many channels and the predicted computation time is that factor longer than that of a single empty compartment. The splitting algorithm

was limited to no piece larger than 0.1 of the total and ended up dividing the cell into 15 subtrees connected at 7 distinct locations. The largest subtree complexity is 7533 with the smallest subtree having a complexity of 122 and consisting of one short section. Seven subtrees are connected at the soma. The LPT algorithm that distributes the 15 pieces on 4 processors results in a maximum complexity on processor 3 of 19440 for a predicted load imbalance of 1%.

[Figure 4 about here]

Figure 4 shows the distribution of cell complexity for a neocortical column model of the Blue Brain Project. From the histogram listing the number of cells per estimated complexity, it can be seen that there is a tail of cells that is much larger than the 9236 average cell complexity. In the case of whole cell distribution the most complex cell of complexity 42385 will be the rate limiting step for any processor count larger than 2180, i.e. when the average load per processor is getting smaller than the load for the largest cell. With the multisplit algorithm we chose to divide cells so that no piece is larger than 0.8 times the average processor complexity, i.e. for 8192 processors there are 16694 pieces and no piece is larger than 9034. The distribution of resulting piece complexities is shown on the right hand side of Figure 4. The LPT algorithm that distributes those pieces on 8192 processors achieves a predicted load imbalance of 5%.

Single cell simulations

[Figure 5 about here]

Figure 5 shows that the multisplit method scales well on the test models on shared memory machines up to 8 processors and gives worthwhile reduction in runtime on 16 processors.

Average computation time (open circles) generally follows ideal scaling behavior though there is apparently a significant high speed cache effect for the Purkinje model on the 4 processor Dell machine. At first sight that is puzzling since the total complexity (the proxy used for load balancing) of the CA1, CA3, and Purkinje models are 33786, 93410, and 25674 respectively. (The similarity in runtimes for the CA3 and Purkinje models is an artifact of the different `tstop` and `dt` for the models such that the number of steps

in a run for the three models is 7500, 5600, and 20000.) The (section, compartment, mechanisms used) counts of the three models are (183, 331, 23), (211, 632, 19), (1088, 1088, 19) and the average complexity per compartment is 100, 150, 25. It seems reasonable to suppose that the complexity is not necessarily related to the number of unique memory accesses per time step. Modest evidence for this is that each of the dual-core processors on the x86_64 Dell machine has 2MB of cache and the change in memory size when each model is created is 3.8, 4.4, and 5.3 MB.

Associated with each open circle (computation time) is a vertical line whose extent shows the minimum and maximum computation times for the processors involved in the simulation. Most of these are too short to be easily visible, demonstrating the excellent load balance, often less than 1%, from the combination of many small pieces and the LPT algorithm. The worst predicted imbalance is 6% for the CA3 simulation on 2 processors and that had a computation time imbalance also of 6%.

For all 30-processor runs, the difference between computation time and runtime is dominated by MPI Send/Recv time with reduced tree computation time being always about 10% of the difference. The reduced-tree receive-buffer size can be calculated from the number of pieces, p, and number of split points, s. There are $s-1$ backbones, each sending 6 doubles, and $p-(s-1)$ leaf subtrees, each sending 2 doubles. (We ignore that the machine solving the reduced tree matrix does not have to send its own subtree information but merely copies the elements into the reduced tree.) Then the receive buffer size for the reduced tree is $2p+4s-4$ double precision values.

A small improvement in 30-processor runtimes, though not enough to justify that number of processors, can be achieved by reducing the degree of splitting. Figure 5 simulations were carried out with a maximum piece size of 0.3 of the average processor complexity. The idea was to have about 3 pieces per processor and in practice what happens is, in addition, there are quite a few much smaller pieces as well. That makes it easier for the LPT algorithm to successfully balance. It is clear that a smaller number of pieces will improve MPI Send/Recv time at the cost of worsened load balance. We re-ran the 30-processor simulations with a range (0.3 to 1.0 in increments of 0.1) of maximum piece size criteria and plotted the best runtime as a dash-mark on Figure 5. Table 1 shows the best runtime with associated statistics along with the original results for the 0.3 maximum piece size factor.

[Table 1 about here]

Network simulations

[Figure 6 about here]

Figure 6 shows runtime performance scaling for the 356 cell Traub model with and without gap junctions on the EPFL IBM Blue Gene computer. Except at 2048 processors the multisplit simulations used a 0.5 maximum piece size factor. At 2048 processors, although load imbalance was very good at 6% the runtime was 23.4 seconds, significantly more than the 18.7 second runtime shown on the graph that resulted from a maximum piece size factor of 0.8 with a load imbalance of 18%. The number of pieces with those two factors were 6811 and 4058 pieces respectively and the extra communication time was larger than the load balance savings (maximum computation time, 14.4 and 15.6 s respectively).

The scaling results for the 10,000 neuron neocortical column simulation of the Blue Brain Project are shown in Figure 7. Over 1000 ms, these simulations exhibit an average per cell firing rate of 6 Hz and, with 13 million connections between cells, generate 77 million synaptic events. For the 1024 and 2048 case the runtime for whole cell balancing and multisplit are essentially identical, i.e. the complexity of the most complex cells is smaller or equal to the average complexity of a processor. In the 4096 processor case, the scaling for the whole cell case breaks down (filled squares) as the most complex cell now becomes the rate limiting step. The multisplit algorithm with LPT balancing on the other hand shows ideal scaling and with respect to average computation time we see a significant cache effect. Though the predicted load imbalance is 5% for the 8192 processor case the measured maximum and average computation time exhibits a load imbalance of 18%. While these results are already very satisfying, further speedup may be achievable if the reasons for the deviation of the measured computation time balance and the expected balance from the complexity proxy can be identified.

Discussion

For the single cell models tested, the number of useful processors is related to the complexity per compartment. The CA3 model had an average complexity per compartment of 150 meaning that the computation time to setup its matrix equation was approximately 150 times longer than that equation's

contribution to Gaussian elimination. With this degree of compartment complexity, runtime scaling is very good up to 16 processors because each processor is computation bound by its set of pieces even though the 110 piece reduced tree matrix has a rank of 58. On the other hand, the Purkinje cell model had an average compartment complexity of 25 and ideal scaling was limited to 8 processors with a 46 piece, 27 rank reduced tree matrix and 1242 average processor complexity.

Migliore et al (2006) justified the use of a naive spike exchange method based on an MPI_Allgather every interprocessor-network-connection-minimum-delay-interval. For the network models presented here, spike exchange overhead is minimized through the use of several compression techniques introduced by Morrison et al. (2005). That is, the 4-byte global identifier is compressed to a single byte (allowed when there are at most 255 cells on any processor) and the 8-byte double precision spike event time is compressed to a single byte (allowed when there are fewer than 256 fixed size time steps within an integration interval). Furthermore, the spike buffer size was set large enough so that there was no occasion for a spike buffer overflow that would require a second MPI_Allgatherv. With these optimizations, which do not affect accuracy with the fixed step integration method, spike exchange is a small portion of the runtime; approximately 0.4 s for the 356 cell Traub model on 2048 processors and 5 s for the 10k cell BlueBrain model on 8192 processors.

Our first implementation of the multisplit method focused on minimizing transfers by allowing only backbones long enough to safely ignore coupling coefficients between the backbone endpoints. Unfortunately, we observed many 3-d reconstruction examples for which splitting into more than a few subtrees required very short backbones whose ends were tightly coupled. Allowing alternating short and long backbones preserved the analogy to the single split transfer pattern but turned out not to be satisfactory because of numerical instability problems when using the second order correct Crank-Nicholson method; despite the transformation of short backbones into an N topology matrix. Also, splitting into n subtrees at the soma required $n(n-1)$ transfers due to exchange between every pair of subtrees. The reduced tree reduces this to $2n$ transfers. The reduced tree method also provides a good practical compromise between splitting flexibility and fast Gaussian elimination. And that method's double precision quantitative identity to single process optimal tree Gaussian elimination has proved to be an extremely useful debugging aid.

Direct static load balancing analogous to the previous paper, which starts by filling processors with the largest possible pieces and splitting as needed to top off, has not been attempted and should do even better than the sequence of "maximum piece size splitting" followed by LPT. However, any replacement algorithm should avoid a wide range for the piece count on the processors since under those conditions we observed large discrepancies between predicted and actual processing times. LPT tends to keep the number of pieces on each processor reasonably similar. Although the general balance problem is np-complete, the non-optimal balance found by LPT is usually good enough, especially when there are many small pieces whose complexity is a small fraction of the average complexity per processor (cf Hayes, 2002). In the future, however, there are two reasons to consider combining splitting and balancing in one algorithm for Blue Gene load balance. First, each Blue Gene node (dual processor) is directly connected, in a 3-D torus topology, to 6 others. If the pieces of a cell are forced to be on these 7 nodes (14 processors) with the reduced tree in the center, one would expect MPI_Send/Recv overhead to be dramatically reduced. Second, the above notion implies local balance and that allows parallelization of the decisions about splitting and distribution, allowing scaling of this part of the problem to the next generation of very large cluster machines.

The CA3 model (Lazarewicz et al, 2002) comes from ModelDB with the variable time step integration method turned on. That method reduces the runtime for a single x86_64 process from 77 to 41 seconds. Presently, the implementation of the multisplit method is restricted to NEURON's fixed step integration method. It would clearly be beneficial to eliminate this restriction and, in principle, that can be easily accomplished since the variable step integrators (Hindmarsh and Serban, 2002) already support parallel equation solving. Unfortunately, the problem becomes complicated in a spiking network environment since, at first blush, NEURON's local variable time step method (Lytton and Hines, 2005) requires that the least-time element of the event and cell queues be identical in all processes that share a part of the same cell. This problem would be, of course, obviated in a threaded environment.

The use of threads is an attractive alternative to MPI in a shared memory environment. With threads, NEURON program launch would be the same as on a single processor and full GUI functionality would be immediately available. Since Gaussian elimination is normally a very small fraction of total simulation time, multisplit is not directly needed, but will be very

important for cache efficiency in order to avoid copying matrix equations from the cache in which they are set up to the cache where they are solved. There may be a savings in reduced tree communication as well since several MPI buffer copies would be avoided. It should also be mentioned that cache efficiency militates against using threads at the compartment level for network simulations where small cells are coupled by spike events in which the minimum delay interval between spike generation and spike delivery is many time-steps. Plesser et al. (2007) show that dividing a network into cell groups, each of which fits into cache, is very effective in avoiding cache misses over the entire minimum delay integration interval. An example of the significance of this for cache efficiency is the dentate gyrus model of Santhakumar et al (2005, also cf Migliore et al, 2006) simulated on a 3GHz i686 single processor machine with a 512KB cache. As a single process the simulation runtime is 290 seconds but when launched with 4 or 8 MPI processes (all on a single processor) the runtime is 253 and 247 seconds respectively.

However, it is not feasible to run large network models on "small" shared memory machines, and, for load balance on processor clusters, the use of multisplit is critical for performance scaling as the number of processors becomes similar to or greater than the number of cells.

Acknowledgments

Research supported by NIH grant NS11613 and the Blue Brain Project. We are grateful to Thomas Morse for his work on initial porting of the Fortran version of the Traub model into NEURON and to Rajnish Ranjan for incorporating the load balance and multisplit methods into the BlueBrain workflow.

References

Hayes B (2002) The easiest hard problem. American Scientist 90: Number 2, 113–117.

Heglund M (1991) On the parallel solution of tridiagonal systems by wrap-around partitioning and incomplete LU factorization. Numer. Math. 59: 453–472

Hindmarsh A, Serban R. (2002) User documentation for CVODES, An ODE solver with sensitiviy analysis capabilities. Tech. rep., Lawrence Livermore National Laboratory. http://www.llnl.gov/CASC/sundials/.

Hines ML, Carnevale NT (1997) The NEURON simulation environment. Neural Comput. 9: 1179–1209.

Hines M, Eichner H, Schürmann F (2008) Neuron splitting in compute-bound parallel network simulations enables runtime scaling with twice as many processors J. Comput. Neurosci.: (accepted)

Korf RE (1998) A complete anytime algorithm for number partitioning. Artificial Intelligence 106: 181–203.

Lazarewicz MT, Migliore M, Ascoli GA (2002) A new bursting model of CA3 pyramidal cell physiology suggests multiple locations for spike initiation. Biosystems 67: 129–137

Lytton W, Hines ML (2005) Independent variable time-step integration of individual neurons for network simulations. Neural Comput. 17: 903–921.

Mascagni M (1990) A parallelizing algorithm for computing solutions to arbitrarily branched cable neuron models. Journal of Neuroscience Methods 36: 105–114

Migliore M, Cannia C, Lytton WW, Markram H, Hines ML (2006) Parallel network simulations with NEURON. J. Comput. Neurosci. 21: 119–129.

Miyasho T, Takagi H, Suzuki H, Watanabe S, Inoue M, Kudo Y, Miyakawa H (2001) Low-threshold potassium channels and a low-threshold calcium channel regulate Ca2+ spike firing in the dendrites of cerebellar Purkinje neurons: a modeling study. Brain Res. 891: 106–115.

Morrison A, Mehring C, Geisel T, Aertsen AD, Diesmann M (2005) Advancing the boundaries of high-connectivity network simulation with distributed computing. Neural Comput 17:1776–801.

Plesser HE, Eppler JM, Morrison A, Diesmann M, Gewaltig M-O (2007) Efficient Parallel Simulation of Large-Scale Neuronal Networks on Clusters of Multiprocessor Computers. In Euro-Par 2007: Parallel Processing, Lecture Notes in Computer Science. 4641: 672–681. eds. Kermarrec, Bouge, Priol.

Poirazi P, Brannon T, Mel BW (2003) Pyramidal neuron as two-layer neural network. Neuron 37: 989–999.

Rempe MJ, Chopp DL (2006) Predictor-Corrector Algorithm for Reaction-Diffusion Equations Associated with Neural Activity on Branched Structures. SIAM J Scientific Computing 28: 2139–2161.

Santhakumar V, Aradi I, Soltesz I (2005) Role of mossy fiber sprouting and mossy cell loss in hyperexcitability: a network model of the dentate gyrus incorporating cell types and axonal topography. J. Neurophysiol. 93: 437-453.

Traub RD, Contreras D, Cunningham MO, Murray H, LeBeau FE, Roopun A, Bibbig A, Wilent WB, Higley MJ, Whittington MA (2005) Single-column thalamocortical network model exhibiting gamma oscillations, sleep spindles, and epileptogenic bursts. J. Neurophysiol. 93: 2194–2232.

Figure Legends

Figure 1: A: Kinetic scheme for the example neuron. B: Scheme is divided into 8 subtrees connected at 4 distinct compartments. C: Subtrees separated from each other each have some terms from the original shared compartment and are connected together by 0 resistance virtual wires.

Figure 2: The sequence of steps that carry out Gaussian elimination of the partitioned neuron in Figure 1C. A: Starting structure prior to any elimination steps. B: After phase1, single connection point subtrees are fully triangularized and two connection point subtrees are triangularized up to the backbone path. C: After phase 2, tridiagonal backbone paths are transformed so that all backbone compartments depend only on the end compartments. D: At the start of phase 3, a reduced tree is constructed. Here it consists of 4 coupled equations. The patterns inside each circle are meant to indicate that the d and b matrix elements are the sums of the corresponding elements from the connecting subtrees. At the end of phase three, the voltages are sent back to the subtrees. E: After phase 3, all subtrees are fully triangularized. F: After phase 4, all voltages along the backbone paths are known. In phase 5, not illustrated, the backsubstitution is completed.

Figure 3: Left panel: Lazarewicz model CA3 pyramidal cell is divided into 15 pieces with 7 distinct connection points. Middle panel: Complexity of each of the pieces ordered from greatest to least. Right panel: LPT algorithm chooses a processor for each piece based on the processor with the least cumulative complexity.

Figure 4: Complexity histograms for the 10000 cell model and when the cells are split into 16694 pieces suitable for load balanced simulation on 8192 processors. Maximum cell complexity is 42,385 and total network complexity is 92,360,610. The largest cell was split into 8 pieces with a reduced-tree matrix rank of 5. LPT load imbalance is 5%. Maximum piece size during splitting was chosen using 0.8*total_complexity/nhost.

Figure 5: Performance as a function of number of processors for three single cell models. The x86_64 machine is the 4 processor Dell. The ia64 machine

is the 32 processor SGI. For the latter, the last two points are for 24 and 30 processors. Filled circles are runtime in seconds. Open circles are average computation time. The dashed line is the "ideal" (-1) slope performance scaling in this log_2 vs log_2 plot relative to the one processor computation time. Open circles contain a vertical line, usually too short to be visible, that spans the minimum to maximum computation time for the processors that took part in each simulation. The dash mark for 30-processor runs is the best runtime as maximum piece size was varied.

Figure 6: Performance as a function of number of processors for the 356 cell Traub model. The style is the same as that of Figure 4 except that the filled squares are the runtime with gap junctions included. At 256 processors there is a switch from whole cell balance to the multisplit method.

Figure 7: Performance as a function of number of processors for the 10000 cell model. Style is the same as that of Figure 4. Filled squares are for whole cell balancing.

Tables

Table 1: Least Runtime in seconds for the three single cell models when the maximum piece size factor (Fac) is varied. Also shown are the results for the standard maximum piece size factor of 0.3. Other columns are % Load Balance, Number of multisplit subtrees (Pieces), Number of split points (Rank of the reduced-tree matrix), Reduced-tree MPI Recv buffer size (Buf) in number of double precision data values, and the Reduced-tree (RT) computation time in seconds.

Model	Runtime	Fac	% Bal	Pieces	Rank	Buf	RT
CA1	2.84	0.9	16	56	31	232	0.06
	3.39	0.3	4	151	79	614	0.15
CA3	5.17	0.5	5	121	64	494	0.09
	5.73	0.3	5	175	89	702	0.12
Purk	6.75	0.8	10	71	43	310	0.21
	8.75	0.3	3	205	124	902	0.57

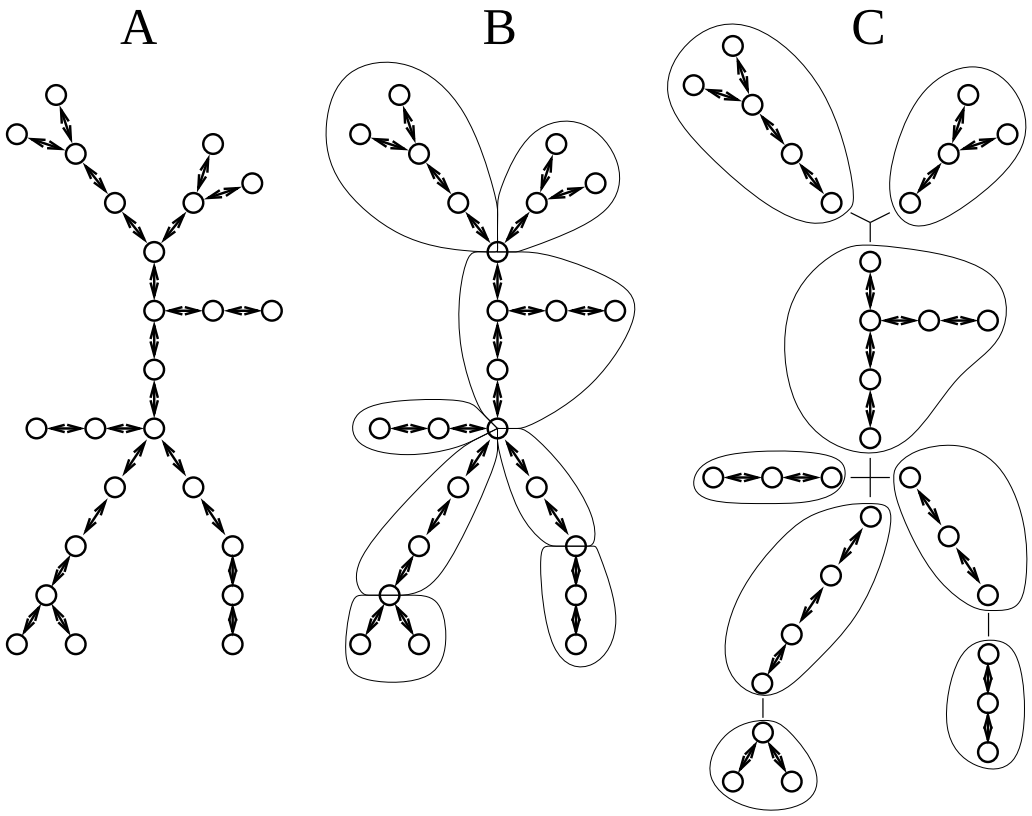


Figure 1:

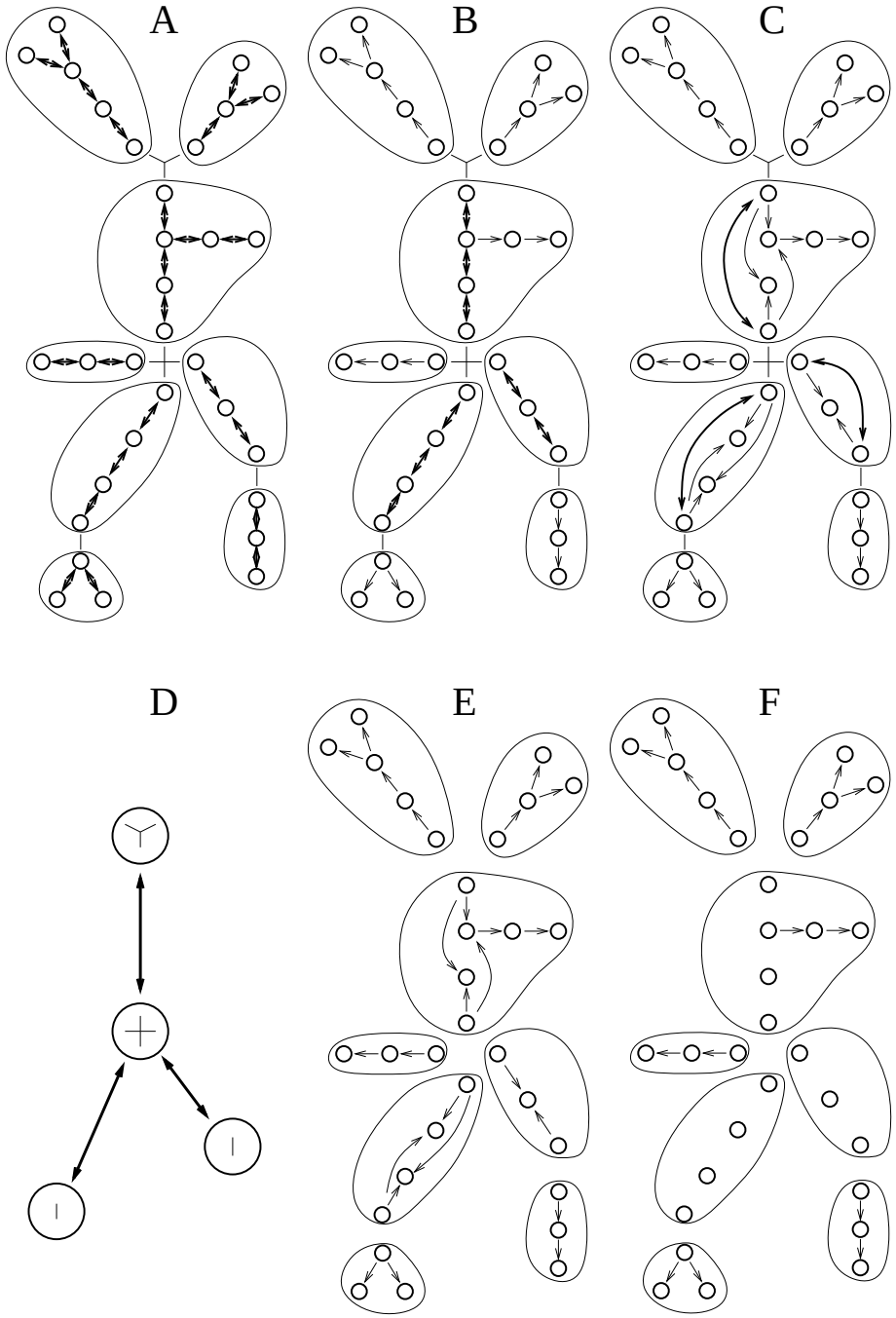


Figure 2:

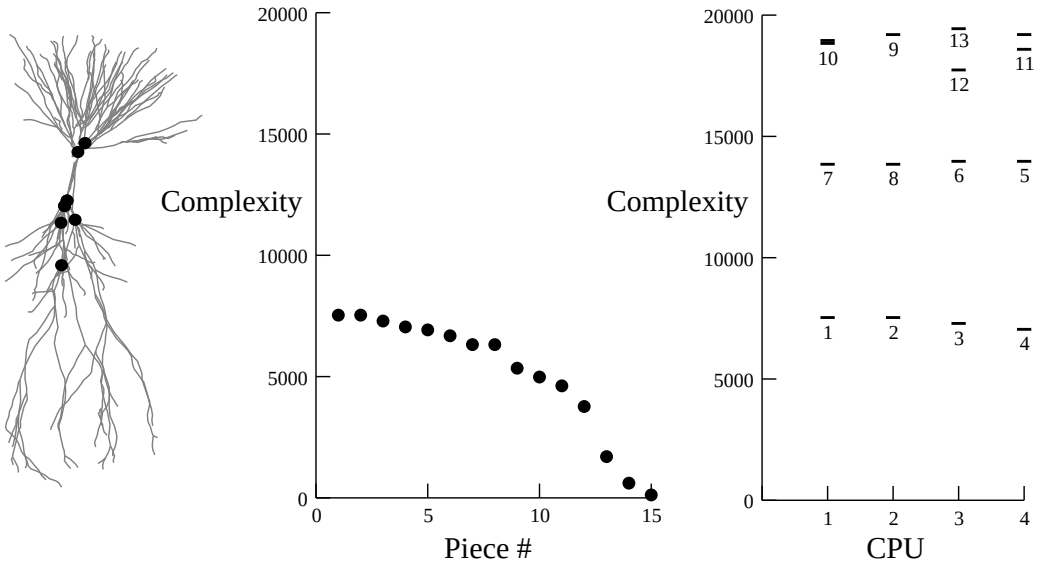

20000
15000
10000
5000
0
Complexity
0
5
10
15
Piece #
20000
15000
10000
5000
0
Complexity
10
9
13
12
11
7
8
6
5
1
2
3
4
1
2
3
4
CPU

Figure 3:

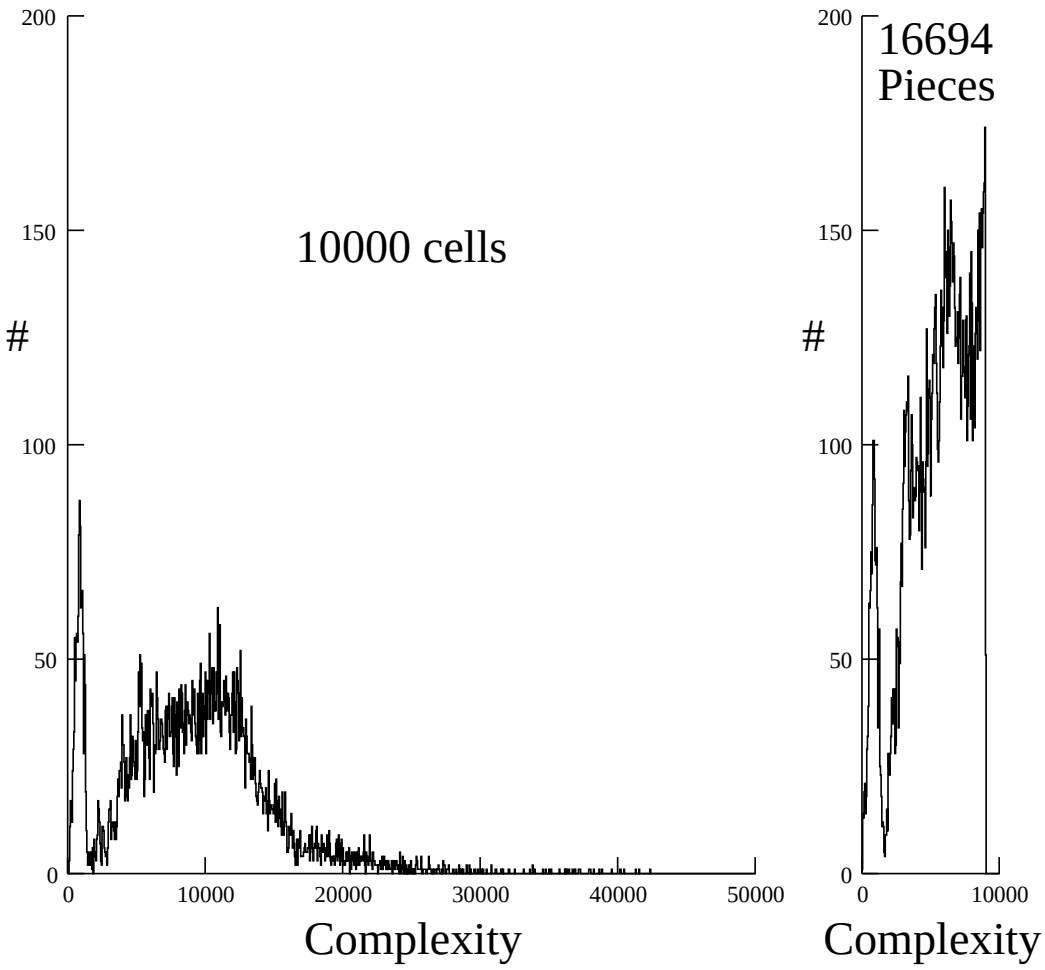

200
150
100
50
0
#
10000 cells
0
10000
20000
30000
40000
50000
Complexity
200
150
100
50
0
#
16694
Pieces
0
10000
Complexity

Figure 4:

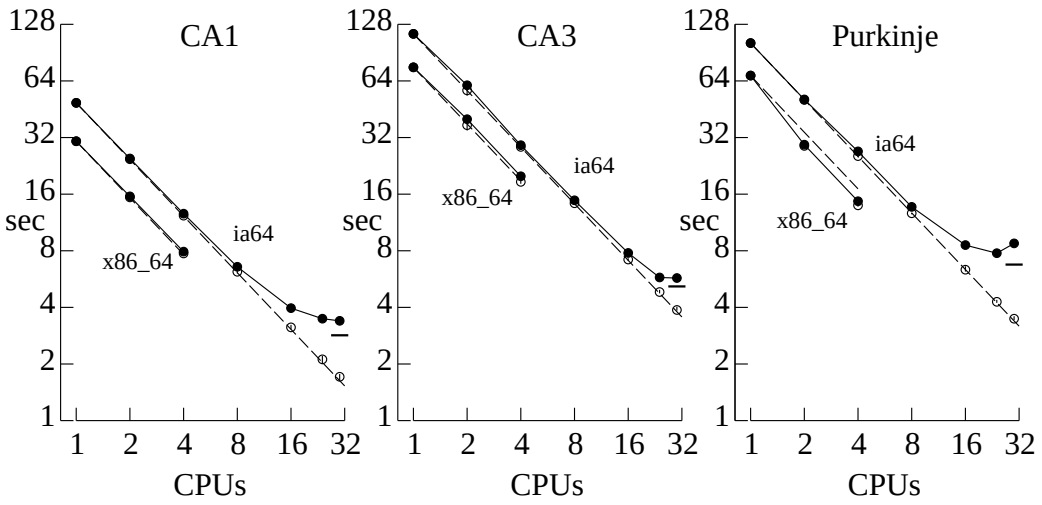

CA1
CA3
Purkinje
128
64
32
16
8
4
2
1
sec
ia64
x86_64
1
2
4
8
16
32
CPUs

Figure 5:

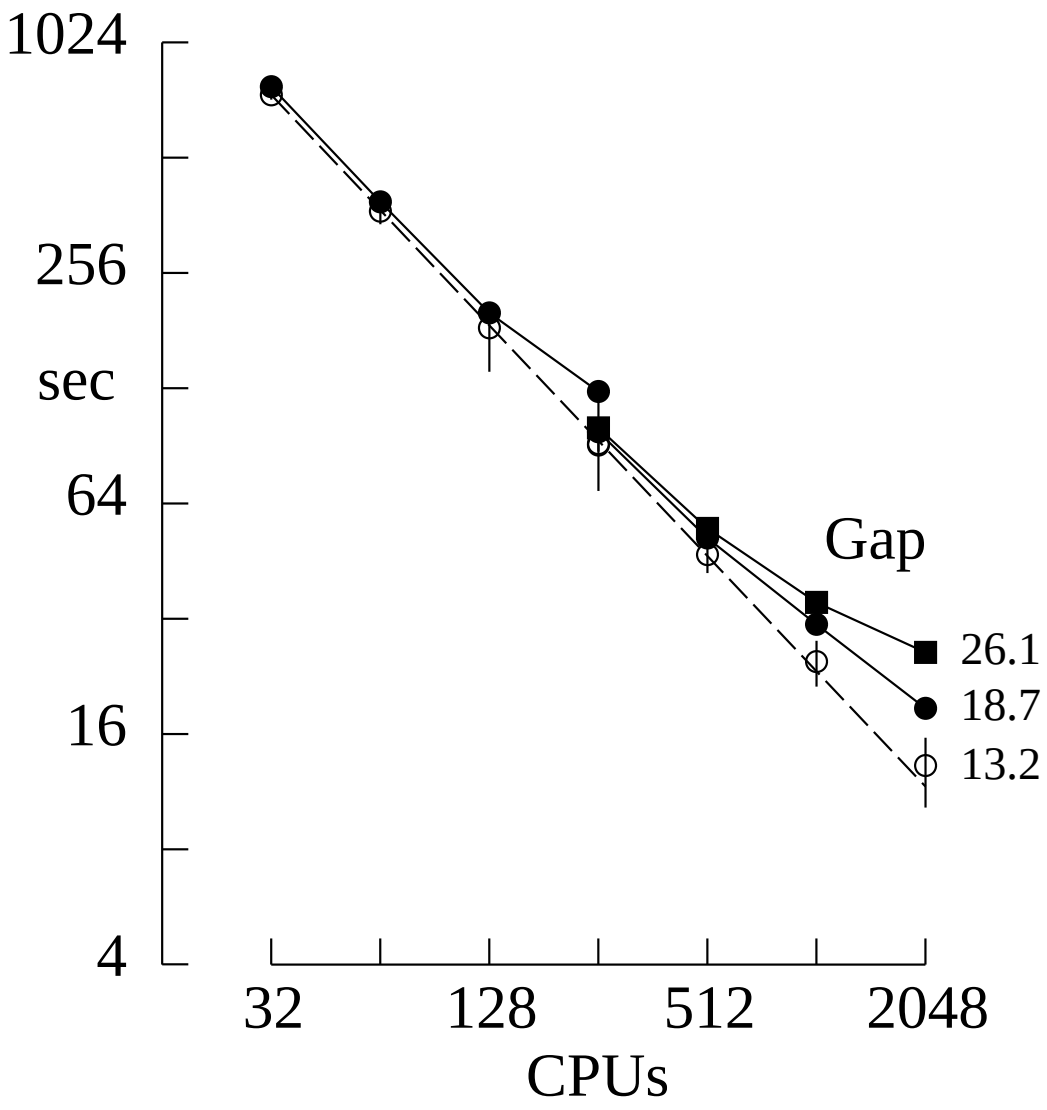


Figure 6:

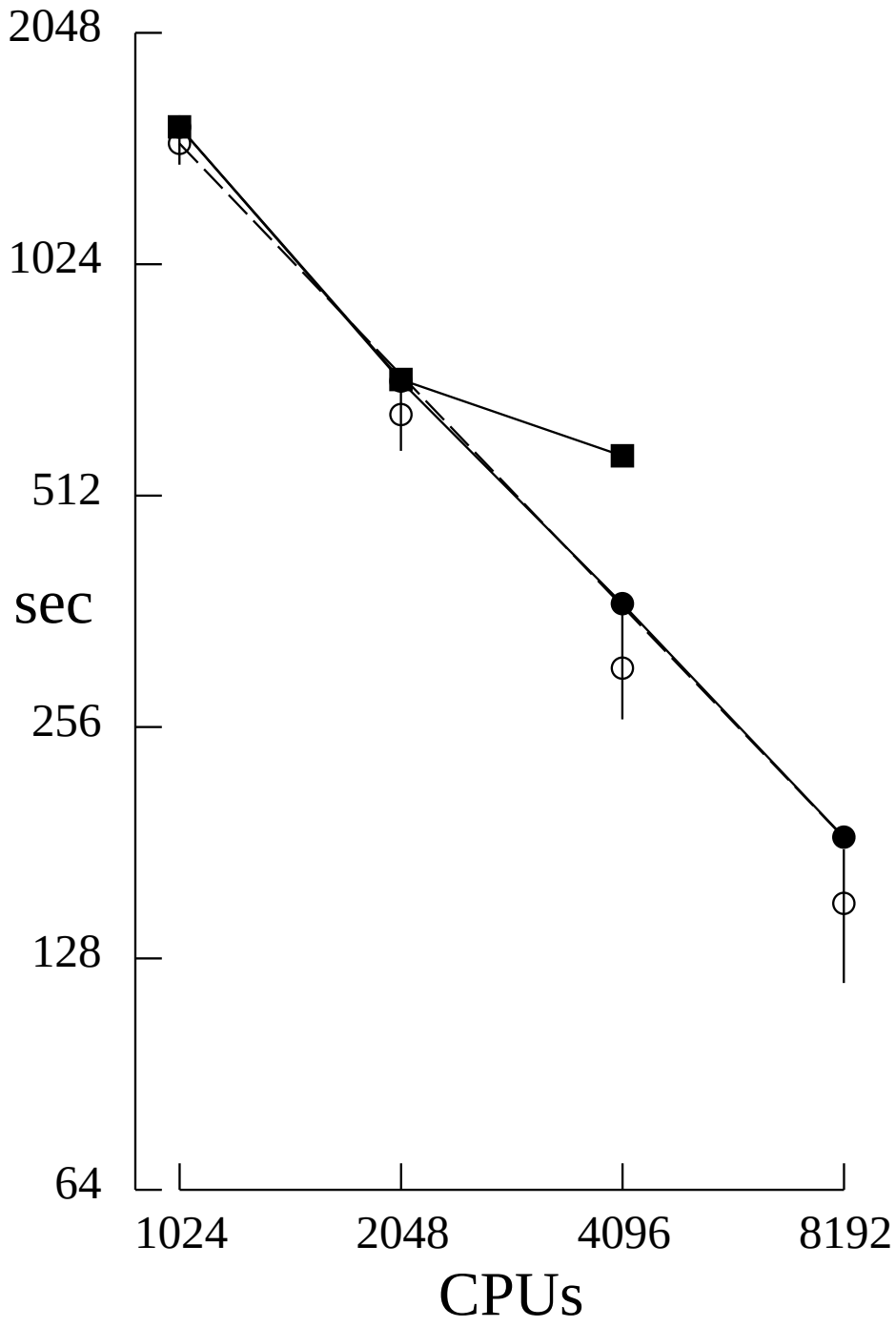


Figure 7: