Neuron splitting in compute-bound parallel network simulations enables runtime scaling with twice as many processors

Michael L. Hines[1], Hubert Eichner[2], and Felix Schürmann[3]
Computer Science[1], Yale University, New Haven, CT
Max-Planck-Institute of Neurobiology[2], Martinsried, Germany
Brain Mind Institute[3], EPFL, Lausanne, Switzerland

March 5, 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

Neuron tree topology equations can be split into two subtrees and solved on different processors with no change in accuracy, stability, or computational effort; communication costs involve only sending and receiving two double precision values by each subtree at each time step.

Splitting cells is useful in attaining load balance in neural network simulations, especially when there is a wide range of cell sizes and the number of cells is about the same as the number of processors. For compute-bound simulations load balance results in almost ideal runtime scaling. Application of the cell splitting method to two published network models exhibits good runtime scaling on twice as many processors as could be effectively used with whole-cell balancing.

Introduction

Load balance is a necessary condition for efficient parallel simulation of networks of neurons. That is, since the overall speed of the simulation is rate limited by the processor which has the most work to do, greatest efficiency is achieved when the maximum workload is as close as possible to the average workload.

Neuronal network simulations are often perfectly balanced by choosing the number of each cell type to be an integer multiple of the number of processors. If individual cell processing time does not vary too widely, or the number of cells is much greater than the number of processors, a simple round robin distribution algorithm analogous to card dealing is usually good enough to divide the load reasonably uniformly among the available processors (cf Migliore, et al, 2006). When the number of processors is such that only a small number of cells can be on each processor, it is often sufficient to iteratively choose the cell with the longest processing time and put it on the least used processor. More sophisticated algorithms such as the Complete Karmarkar-Karp algorithm (Korf, 1998) can also be employed.

Although investigators often choose to scale the network size with the number of available processors to keep runtime more or less constant (weak scaling), on occasion it is desirable to keep the problem size the same and reduce runtime through the use of as many processors as is consistent with strong scaling. Strong scaling is obviously impossible via whole-cell load balancing when the number of processors is greater than the number of cells. With heterogeneous size cells, strong scaling fails even earlier. Clearly, the strong scaling regime can be extended to larger numbers of processors only by splitting cells into smaller pieces.

Very strong coupling between adjacent compartment voltages demands implicit methods for numerical stability with reasonable time steps and therefore a matrix equation must be solved every time step. Though the matrix setup can obviously be parallelized down to the individual compartment level, optimal solution of the tree matrix via Gaussian elimination is normally accomplished by a recursive, and therefore serial, algorithm, e.g. Mascagni and Sherman (1996) or Hines (1984). This paper describes a method for splitting cells into two subtrees simulated on different processors with no change in accuracy, stability, and computational effort, and with a communication cost consisting of only two double precision values sent and received by each subtree at each time step. The method is closely related to one discussed

in Hines (1994) in the context of Cray YMP vectorization. Since a cell can be split at one of many locations, satisfactory load balance is routinely obtained by filling a processor with cells to just below the average workload and topping off with a split piece.

We compare the runtime scaling behavior vs number of processors for two whole-cell load balance methods and the cell splitting method using two published network models: a thalamocortical model by Traub et al (2005) and a dentate gyrus model by Santhakumar et al (2005).

Methods

All simulations were carried out with the NEURON v6.1 simulation program (Hines and Carnevale, 2007). The "splitcell" functionality is available when NEURON is configured with the `--with-paranrn` option which requires preinstallation of an implementation of the Message Passing Interface (MPI). Performance tests were carried out on the EPFL IBM Blue Gene/L.

Load balance and cell splitting algorithms were tested by modifying two published network models from the ModelDB section of the Senselab database (http://senselab.med.yale.edu): Traub et al (2005), and Santhakumar et al (2005). In order to focus on load balance with reduced use of computer resources we scaled down the Traub model 10-fold and turned off gap junction interactions. This left a minimal model of 356 cells of 14 types with all type ratios preserved. In all cases the parallel models produce quantitatively identical spike times as their original serial counterparts. Complete model code with modifications used in this paper is available from ModelDB with accession number 97917.

Numerical methods for cell splitting

The most important efficiency attribute of spatially discretized neuron equations is that the number of arithmetic 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 (cf Hines and Carnevale, 1997). Optimal Gaussian elimination triangularizes the matrix proceeding from leaves to the root of the tree and back substitutes in reverse order from root to leaves. At a branch point, one cannot continue the triangularization process till the subtrees at the branch have

been triangularized. Conversely, one cannot start on the back substitution of the subtrees of a branch point until after the parent cable has been back substituted. Any compartment can serve as the root of the tree. In the simplest case of an unbranched single cable, it takes exactly the same number of operations to triangularize simultaneously from the two ends to some middle point and back substitute from there as it does in the normal sequence of triangularization from one end to the other.

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 refer to the voltages of this, the unique parent, and all the child compartments respectively. The a coefficients are constants depending only on the shape of the compartments, capacitance, 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. After triangularization has eliminated the effect of child voltages on the current balance equations, each compartment equation contains only two terms, one involving this compartment voltage and one involving the parent voltage, with changed values for d_i and b_i

$$a_p V_p + d'_i V_i \;=\; b'_i$$

and, since the root compartment has no parent

$$V_r \;=\; b_r / d_r$$

which can then be substituted into each of the root's child equations.

It is clear that there is no impediment to simultaneous triangularization of each subtree of the root compartment. The only question is precisely how to handle the arithmetic operations that modify the root compartment equations since one of the child compartments is on a different processor. One possibility is to split a cell at the boundary between two compartments. In this case, there are really two root compartments and after triangularization on each processor, the problem is to solve the two root equations which are coupled by the conductance between the centers of those compartments. I.e.

$$\begin{aligned} d_{r1} V_{r1} + a_{12} V_{r2} &= b_{r1} \\ a_{21} V_{r1} + d_{r2} V_{r2} &= b_{r2} \end{aligned}$$

Here, to complete the triangularization, the processor handling root 1 needs d_{r2} and b_{r2} from the processor handling root 2 and vice versa.

The other possibility is more in keeping with NEURON's semantics of the "connect" statement. That is, imagine that two trees, simulatable in isolation have their roots connected by a zero resistance wire. This is tantamount to dividing the root compartment itself into two pieces and full triangularization ends up with two equations of the form

$$\begin{aligned} d_{r1}V_r + i_r &= b_{r1} \\ d_{r2}V_r - i_r &= b_{r2} \end{aligned}$$

where i_r is the current flowing in the (virtual) wire connecting the roots. In practice, i_r is not computed and uses no memory location. Instead, the two equations are added together by each machine exchanging its triangularized d and b and adding them to their corresponding quantities so that each machine redundantly solves

$$(d_{r1} + d_{r2})V_r = (b_{r1} + b_{r2})$$

Considering Gaussian elimination only, the number of operations required for the two methods is almost identical. However, for several reasons we prefer splitting the root node itself despite the apparent redundancy of both processes computing the same value for V_r. First, as already mentioned, the second method corresponds to the semantics of the NEURON connect statement

```
connect child(0or1), parent(x)
```

which represents the connection of either end of a child cable to any location of the parent cable without introducing any extra conductance at the connection point. The concept of connecting two trees together by a wire is somewhat simpler than requiring both processes to compute the constant coupling coefficient which depends on length and diameter of both compartments. Second, and more substantively, splitting at the boundary between compartments requires, in addition to the exchange of the triangularized d and b, an exchange of the values of V_{r1} and V_{r2} at the beginning of the time step since evaluation of b_{r1} during the setup phase depends on the previous step's value for V_{r2} and vice versa. It should also be mentioned that, although NEURON does not currently support splitcell (or gap junction)

simulation using the variable step methods, because synchronization of the event queues on different processors has not yet been implemented, the variable step method setup phase which involves evaluation of the right hand side of

$$cy' \;=\; f(y,t)$$

does require an estimate of the value of the i_r current and so it is not clear at present whether the strategy of splitting the root node will retain its setup advantage over splitting at a compartment boundary. The answer depends on details of the way NEURON handles evaluation of the voltage at the zero area nodes at the ends of sections.

NEURON support for splitcell computation.

In most cases the parent location in a connect statement is also the 0 or 1 end of the parent and that is the situation captured by an extension to the /tt ParallelContext class that NEURON uses to manage interprocessor communication. Assuming the object reference, `pc`, references an instance of the `ParallelContext` class, then a virtual interprocessor connection is made between two trees by a pair of corresponding

```
rootsection { pc.splitcell(otherhost) }
```

Here, it is an error if the `rootsection` is connected to a parent section, i.e. is not the root of its tree. The argument, `otherhost`, must be `pc.id` plus or minus 1 and the `splitcell` method call makes the connection to the location identified by the corresponding call to `splitcell` on the other host. It can be seen that we purposely limit a cell to be split between hosts i and i+1. Thus, a host can deal with at most two split cells, one part of one cell on host i-1 and one part of the other cell on host i+1. This allows MPI_Send and MPI_Recv to take advantage of the Blue Gene torus topology to minimize communication time. It sacrifices the obvious possibilities inherent in choosing a branch point as the root node and dividing a cell into three pieces. It also disallows the possibility of splitting a cell and simulating the two pieces in a single process. But the latter is more or less pointless and the same result can be obtained with the connect statement.

Splitcell is well suited to the practical issues of specifying cells in the NEURON environment. That is, most models define cell types that are instantiated as a whole and it is desirable to split different instances of the

same type at different locations. Thus it is proper to separate the code that defines a cell from the code that disconnects the two subtrees, destroys the portion of the cell not needed on a host, re-roots the remaining subtree so that the disconnect point is the root node, and calls the splitcell method on the root of the remaining subtree. A wrapper method has been provided in NEURON's tool library to carry out these subsidiary operations so that the cell is split properly if the same statement is executed on the two processes that have instantiated a cell. Clearly, there is a good deal of wasted effort in setting up an entire cell and then throwing away half of it. But in a network context, splitting is a small portion of the total network setup time, which in turn is very small compared to runtime. Also, if administrative issues can be overcome in regard to processes creating only the subtrees they need, there is no reason one cannot connect those subtrees with the `ParallelContext.splitcell` method.

Results

Load Balance

Cells differ in the amount of processing time they add to network simulation runtime due to differences in number of cell compartments, types of channels in a compartment, the number and complexity of synapses, and the number of spikes handled by the synapses on a cell.

The simplest whole cell load balance strategy ignores differences in individual cell processing time and distributes cells (each one identified by a global unique identifier, gid) according to the round robin or card dealing algorithm

```
for (gid = pc.id; gid < ncell; gid += pc.nhost) {
    // create cell associated with gid
}
```

where `pc.id` and `pc.nhost` refers to the MPI rank and total number of MPI processes respectively. This is the recommended default method for parallel NEURON simulations and, based on the measured computation time on each processor, provides an initial measure of load balance to assess whether it is worthwhile to try methods that are more complicated but might yield superior performance.

The simplest load balance algorithm which takes into account differences in cell processing time is the "longest processing time" (LPT) algorithm in which one iteratively chooses the largest cell and puts it on the currently least used processor (cf Korf, 1998). The use of LPT, as well as our splitcell load balance method described below, requires a weight assignment as close as possible to the actual computation time required by a cell during a simulation. Unfortunately, such fine grain measurement even at the whole cell level during a trial simulation introduces significant measurement artifacts and we have found it useful to use more easily measured proxies to estimate the relative computation time that would result from a given partition.

We have tested several proxies, e.g. number of NEURON sections, number of compartments, and number of equations. But the proxy or complexity measure that corresponds most closely to actual computation time is to measure the computation time of a 100 compartment cable with each membrane channel or mechanism inserted separately and normalize to the computation time of the cable with no channels present. We do this also for synapses by instantiating one per compartment with the caveat that since no spikes are handled, it may underestimate the computation time added by a synapse in the actual simulation.

With a complexity measure for each mechanism type and each elementary compartment (an empty compartment has complexity 1) it is straightforward to assign a total complexity value to each compartment, each cell as a whole, C_i, and the whole network, C_n. The load balance problem can then be expressed as partitioning the network so that the maximum complexity on any processor is as close as possible to $C_n/$`nhost`.

<Figure 1 about here>

Figure 1 shows the number and complexity range of the cells which are to be distributed among the processors for the Santhakumar and Traub models. In order to obtain the complexity data for each cell, we perform a complete setup of the entire network, including synapses, using the default round robin distribution. Each host appends to a file the complexity of each cell on that host. There are 5 distinct cell sizes for the Santhakumar model. The single cell with complexity 0 is an artificial cell used for stimulation. The 15 largest are 17 compartment Mossy cells and the most common are 9 compartment Granule cells. The location and number of synapses does not vary within a type. The Mossy cell complexity of 1487 means that we predict it will

take 1487 times longer to solve the equations for one of those cells than to solve a single empty compartment. The Traub model has 14 cell types. Within a type, number and locations of synapses are chosen randomly from type-to-type projection dependent distributions. For the particular instance shown here, there were 137 distinct complexity values and the 356 instances of those values are displayed in a histogram with bin size 50. Tufted deep IB cells with 61 compartments are the most common cell type, 80 of them. The 10 largest are thalamocortical relay cells with 137 compartments. The complexity information in the "whole-cell" file is used by the LPT algorithm to determine the distribution of cells on processors.

In addition to whole-cell complexity, each host also determines for each cell the set of complexity pairs for each possible split point. A branch point with a parent and two children adds three distinct subtree pairs, (parent alone, child1 + child2), (parent + child1, child2), and (parent + child2, child1) or 6 distinct complexity values to the set of possible split cell complexities. At branch points with more than three connected sections, typically at the soma which may have a dozen branches, nb, the $nb * (nb - 1)/2$ possible complexity pairs were arbitrarily reduced to $2 * nb$ examples consisting of successive addition of the least or greatest remaining complexity subtree to the parent subtree.

<Figure 2 about here>

The left side of figure 2 shows the distinct split piece sizes for the cell types of the Santhakumar model.

The total complexity of each cell along with their sets of split piece complexity data was used to decide upon a distribution of cells and split cells on `nhost` processors where the maximum complexity on a processor was as close as feasible to the average complexity and with the constraint that only one cell can be split between processors i and $i+1$. Because of this constraint as well as the ability to split a cell at one of many locations, LPT is not directly applicable and we instead attempted to find accurately balanced partitions using simplistic heuristics which, nevertheless, were reasonably successful on our present networks.

The principle heuristic is to fill one processor at a time with the largest remaining whole cells. When a whole cell would cause the processor to exceed its maximum complexity, top off the processor by choosing the optimum split piece of that whole cell such that the processor has less than its maximum complexity. Then begin filling the next processor with the left over piece.

The result of the heuristic is illustrated on the right side of figure 2 where the maximum complexity per processor was chosen to be 3% larger than the average complexity, $C_n/256$ ($C_n = 402493$), for the 528 cell Santhakumar network. In this case, the fill algorithm succeeded in that less than 256 processors were needed. The last piece (along with the 0 size stimulus) is placed on processor 252 and the last 3 processors are empty. Apart from those, processor 15 has the least load with a complexity of 1325. Processor 28 has the greatest load with a complexity of 1626. Notice that processor 0 is filled with a whole Mossy cell and topped off with the smallest piece of a second Mossy cell. Processor 1 is filled with the remaining (largest) piece of the second and the next to smallest piece of the third Mossy cell. If more than `nhost` processors are required by the fill algorithm, then the maximum complexity per processor has to be increased and we try again. For a 2% maximum above average complexity, the algorithm happens to need 262 processors, hence the acceptance of a 3% load imbalance for a 256 processor allocation.

The distribution information is written to an `nhost` specific file with a format such that it is a straightforward matter for each processor to quickly find the relevant section of the file and read only the information for processors $i-1$ and i to determine which cells are to be instantiated on processor i and, if split, which portion is to be retained.

Performance

Figure 3 shows the runtime performance of the 528 cell Santhakumar and 356 cell Traub models as a function of number of processors using the three load balance algorithms described above.

<Table 1 about here>

Table 1 shows the load imbalance of the distributions chosen by the Round Robin, Least Processing Time, and Split cell algorithms. The last two columns in Table 1 compare the predicted balance of the split cell method and the balance measured during a simulation and demonstrate that the complexity values derived from the processing time for individual channel and synapse types are reasonable proxies for total computation time. Of course, for 512 processors there are more processors than Traub model cells and so the Round Robin and Longest Processing Time algorithms yield identical run times.

<Figure 3 about here>

On up to 128 processors, the LPT and Splitcell algorithms generate distributions that are very well load balanced. The total complexity and maximum cell complexity for Santhakumar are 402493 and 1487 respectively so that whole cell load balance is impossible with more than 270 processors. For Traub, total complexity is 1.13e6 with maximum cell complexity 4740 so that the whole cell balance limit is below 238 processors. Given these total complexities along with the 300ms and 1000ms simulation times with time steps of 0.1ms and 0.025ms, one expects the Traub model to have a runtime 38 times longer than the Santhakumar model. That is consistent with the observed computation time ratio of 37. Although the spiking patterns of the two models are very different they have a similar network spiking rate, approximately 12 spikes/ms and 9 spikes/ms for the Traub and Santhakumar models respectively. Thus we expect spike exchange overhead to be similar for the two models and this, along with our complexity measure indicating the almost three fold greater complexity of the Traub model, explains the three fold greater (runtime/computation - 1) value for the splitcell balanced Santhakumar model for 256 and 512 processors. That is, for a balanced model, computation time is proportional to complexity, spike exchange time is proportional to spike rate, and splitcell matrix communication time is proportional to number of time steps per millisecond. The more complex Traub model is thus performance limited by its complexity at least up to 512 processors whereas Santhakumar spike exchange overhead is becoming noticeable at 256 processors

Discussion

We have shown that splitting a cell into two pieces on separate processors nevertheless allows Gaussian elimination to be carried out with almost no increase in computational effort and no change in accuracy or stability. Splitting cells into two pieces and doubling the number of processors can reduce runtime by a factor of two. This performance doubling is realized when computation time is much greater than spike exchange time and the overhead of per time step exchange of the triangularized diagonal and right hand side elements of subtree roots. In this regime, when load balance is achieved, runtime performance displays nearly ideal scaling behavior.

It is worth discussing similarities and dissimilarities of the presented split cell method with how gap junctions are treated in a simulation. Most importantly, gap junctions couple pairs of compartments so that the equations no longer have a tree topology. Yet, gap junctions have a relatively large resistance compared to the resistance between the centers of adjacent cable compartments so that the extra off diagonal Jacobian elements tend to be very small relative to the diagonal and, for reasonable time steps, can be ignored during Gaussian elimination without significantly affecting stability or accuracy. Thus, gap junction currents affect only the right hand side and diagonal of the current balance equation 1 and cells can continue to be split at any compartment. Interprocessor gap junctions require an exchange of voltages on either side of the gap at the beginning of each time step. However, strongly coupled split cells cannot be reconnected using the gap junction method without introducing serious stability and accuracy problems that can only be overcome through the use of much smaller time steps. Fortunately, the message payload doubling from one to two double precision values does not increase MPI Send/Recv time so the split cell method makes it possible to maintain accuracy and stability with no increase in communication time over that of a gap junction.

Longitudinal ionic diffusion presents a case of even weaker coupling than that of gap junctions and an exchange of ion concentration state variables is certainly sufficient without causing numerical instability. On the other hand, simulations involving extracellular fields involve very tightly coupled equations with a grid topology and are therefore outside the scope of the presented method. Mechanisms such as radial diffusion, most second messenger cascades, and channel gating states, do not exhibit inter compartmental coupling except indirectly through membrane potential and therefore simulations involving such methods are not affected by cell splitting.

Because of per time step communication overhead, practical use of the split cell method (as well as gap junctions) requires hardware with a high speed interconnect. Performance is likely to be poor on Beowulf clusters with Ethernet connections slower than 1 gigabit/sec.

A significant benefit of the double precision quantitative identity (modulo round off error due to different ordering for Gaussian elimination) between split and whole cell method compartment voltages during a simulation is the ease in diagnosing errors that result in non-identity of network spike patterns. Such errors typically occur during the setup of connections between source cell and target synapse because of the extra user logic involved in determining

which source cell split piece contains the source location and which target cell split piece contains the synapse location.

Minor differences between predicted load balance via the complexity proxy and measured load balance during a simulation (last two columns in Table 1 and the RR, LPT 512 processor Traub items) can be attributed in part to several additional factors that contribute to computation time such as spike queue management, and the synapse calculations that occur on delivery of a spike. Computation time is also sensitive to high speed cache misses that depend on the problem size and memory usage patterns but this may be less of a factor for the higher number of processors since in that case each processor's portion of the problem fits entirely into the cache.

The limitation of splitting into two pieces essentially limits the method's benefits to at most a doubling of the number of processors on which a given simulation can be usefully performed. In simulations with identical numbers of cells and processors, the most complex cell is the rate limiting step. Therefore, two-fold speed-up with the split cell method can only be achieved if the most complex cell indeed can be split into two equal pieces. Particularly, in the context of very detailed simulations, such as the neocortical simulations of the Blue Brain Project (Markram, 2006), certain cell types can be more than twice as complex as the average cell (e.g. detailed layer 5 thick tufted pyramidal cells exhibit substantially more complex dendrites than layer 2/3 cells). Thus, being able to split cells into more than two pieces is strongly desirable and motivated the extension of the presented method. Fortunately, Gaussian elimination is generally a small portion of the overall computation time so that an extension of the method to split a tree at many nodes into many pieces, even though it implies a significant Gaussian elimination complexity increase, means one can usefully increase the number of processors by up to a factor of 8 — even for large single cell models. A subsequent paper presents the extended method (Hines, 2008).

Acknowledgments

Research supported by NIH grant NS11613 and the Blue Brain Project.

References

Hines, M., 1984. Efficient computation of branched nerve equations. *Int J Biomed Comput. 15*, 69-76.

Hines, M., 1994. The NEURON simulation program. In: Skrzypek, J. (Ed.), *Neural Network Simulation Environments*. Kluwer Academic Publishers, Norwell, MA, pp. 147-163.

Hines, M. L. and Carnevale, N. T., 1997. The NEURON simulation environment. *Neural Comput. 9*, 1179-1209.

Hines, M. L. and Carnevale, N. T., 2007. Translating network models to parallel hardware in NEURON. *J Neurosci Meth.* (accepted)

Hines, M. L., Markram, H. and Schürmann, F., 2008. Fully implicit parallel simulation of single neurons. *J Comput Neurosci.* (submitted).

Korf, R. E., 1998. A complete anytime algorithm for number partitioning. *Artificial Intelligence. 106*, 181-203.

Markram, H., 2006. The blue brain project. *Nat Rev Neurosci. 7*, 153-160.

Mascagni, M. and Sherman, A., 1996. Numerical Methods for Neuronal Modeling. In: Koch, C. and Segev, I. (Eds.), *Methods of Neuronal Modeling: From Ions to Networks*. MIT Press, Cambridge, MA.

Migliore, M., Cannia, C., Lytton, W. W., Markram, H. and Hines, M. L., 2006. Parallel network simulations with NEURON. *J Comput Neurosci. 21*, 119-129.

Santhakumar, V., Aradi, I. and 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, R. D., Contreras, D., Cunningham, M. O., Murray, H., LeBeau, F. E., Roopun, A., Bibbig, A., Wilent, W. B., Higley, M. J. and Whittington, M. A., 2005. Single-column thalamocortical network model exhibiting gamma oscillations, sleep spindles, and epileptogenic bursts. *J Neurophysiol. 93*, 2194-2232.

Figure Legends

Figure 1: Cell complexity variation for the Santhakumar and reduced Traub model. For the Santhakumar model there are 500 cells with complexity 731.

Figure 2: Left: Distinct split piece sizes for the cell types of the the Santhakumar model. The number of cells for each type is indicated below the type label. Right: Example of filling 256 processors with 528 cells with 3% balance tolerance. The vertical lines with marks indicate the cell and piece sizes that add up to the load on selected (labeled by processor id) and adjacent processors. The horizontal line with complexity 1572 indicates average processor load.

Figure 3: Run time performance in seconds vs number of processors for the Santhakumar and Traub models and for three load balance algorithms. RR is Round Robin, LPT is Longest Processing Time, and Split is for some cells split into two pieces and simulated on different processors. Because the Traub model has 356 cells, RR and LPT methods have the same runtime for 512 processors. Open circles show the average computation time (same for all balance methods). Dashed line shows ideal speedup (slope is -1 on the log_2 vs log_2 plots) relative to average computation time with 32 processors.

Tables

Table 1: % load imbalance for Traub and Santhakumar models. Imbalance for the Round Robin (RR) and "Split CT" column is derived from the maximum and average processor computation time of a simulation. LPT and "Split CX" columns show the predicted imbalance derived from the complexity proxy.

Model	#CPU	RR	LPT	Split CX	Split CT
Traub	32	6	5	0	1
	64	15	6	1	2
	128	31	8	2	4
	256	88	37	3	6
	512	120	115	11	13
Santhakumar	32	5	4	0	2
	64	16	5	1	2
	128	37	16	1	2
	256	78	39	3	5
	512	158	89	4	6

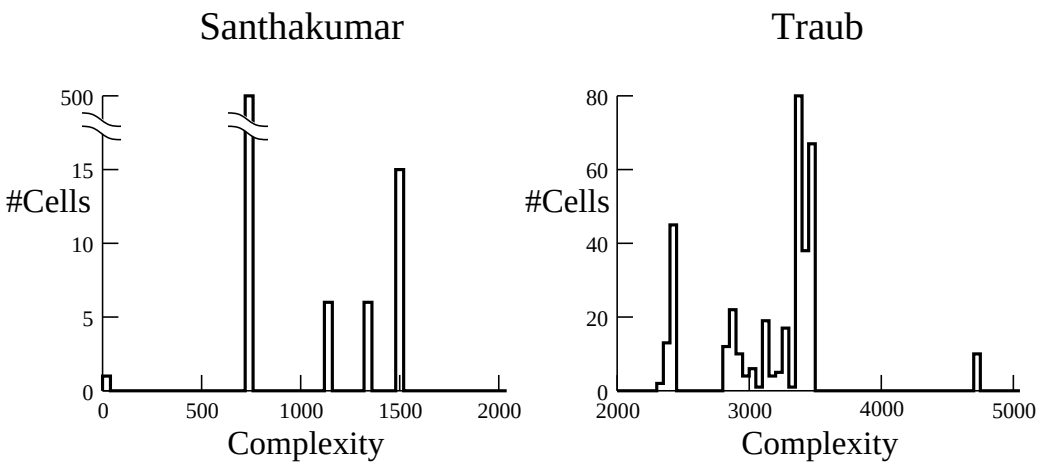
Santhakumar
Traub
500
15
10
5
0
#Cells
0
500
1000
1500
2000
Complexity
80
60
40
20
0
#Cells
2000
3000
4000
5000
Complexity

Figure 1

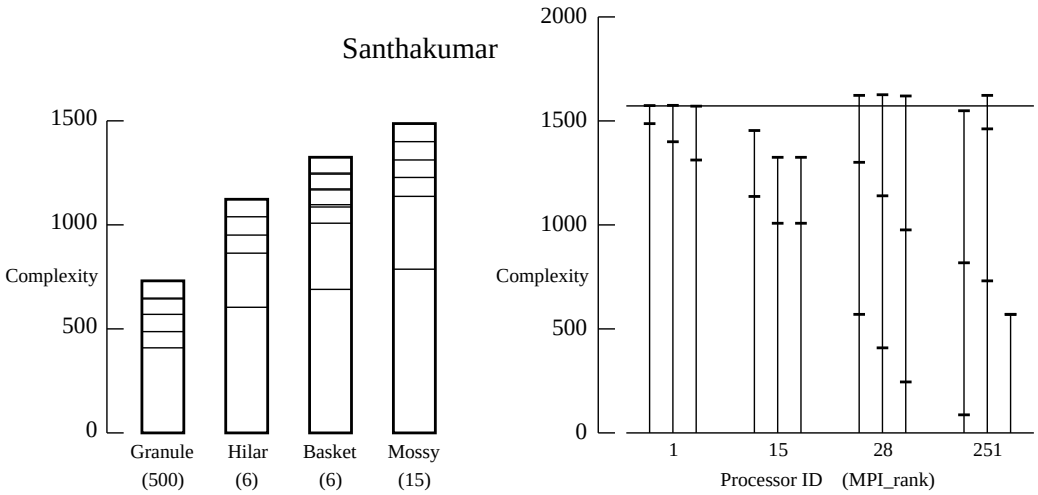
Santhakumar
1500
1000
Complexity
500
0
Granule
(500)
Hilar
(6)
Basket
(6)
Mossy
(15)
2000
1500
1000
Complexity
500
0
1
15
28
251
Processor ID (MPI_rank)

Figure 2

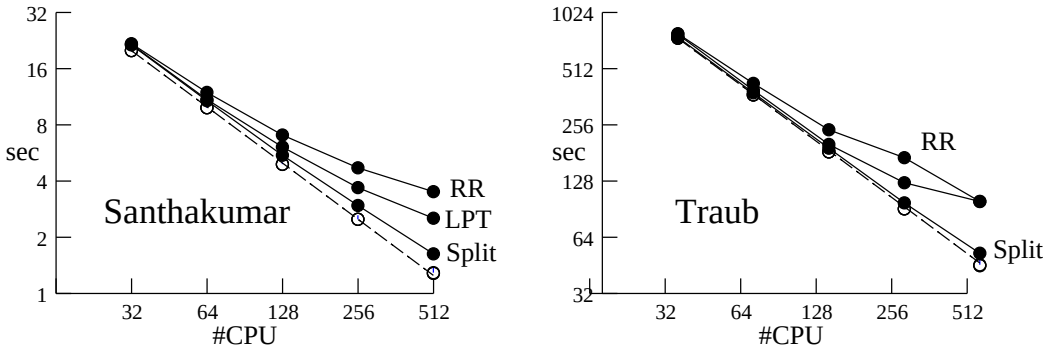
32
16
8
sec
4
2
1
Santhakumar
RR
LPT
Split
32
64
128
256
512
#CPU
1024
512
256
sec
128
64
32
Traub
RR
Split
32
64
128
256
512
#CPU

Figure 3