Final Report:
Excitable Cell Simulation in Star-P

Motivation

My PhD research involves mathematical modeling of cells in the heart's "pacemaker." These cells (sinoatrial node cells, or SANCs) undergo spontaneous depolarization driven by membrane ionic currents as well as intracellular Ca^{2+} dynamics. The activity of the cells is also modulated by the autonomic nervous system (ANS), which controls how quickly the pacemaker cells depolarize, and as a result sets the heart rate.

Currently, models of SANCs exist, and their ability to replicate specific ANS inputs has been demonstrated. However, a unified model of cell function including ANS control has not yet been developed. I plan to use the existing models and combine them into a comprehensive one.

In doing this work, the complexity of the model is bound to increase drastically. The most important change will be that the model (described in terms of coupled ordinary differential equations) will become much more stiff since the electrical activity currently modeled occurs on a timescale of milliseconds, whereas some of the chemical regulation mechanisms elicit changes over tens of seconds. Therefore, before I get too deep into creating this unified model, I'd like to make sure that I can run the simulation in parallel and get a significant speed-up. I would also like to make the actual implementation and development time as short as possible, so this is why I am staying very high-level and using Star-P.

Plan

In general, the plan is to implement the most physiologically accurate SANC model in existence, and evaluate the time needed to run simulations in serial versus in parallel. Developing the actual final model that includes ANS modulation of SANC function would take a few months on its own, so for the purpose of this project, I will evaluate the performance of just the base model. I plan to do the following:

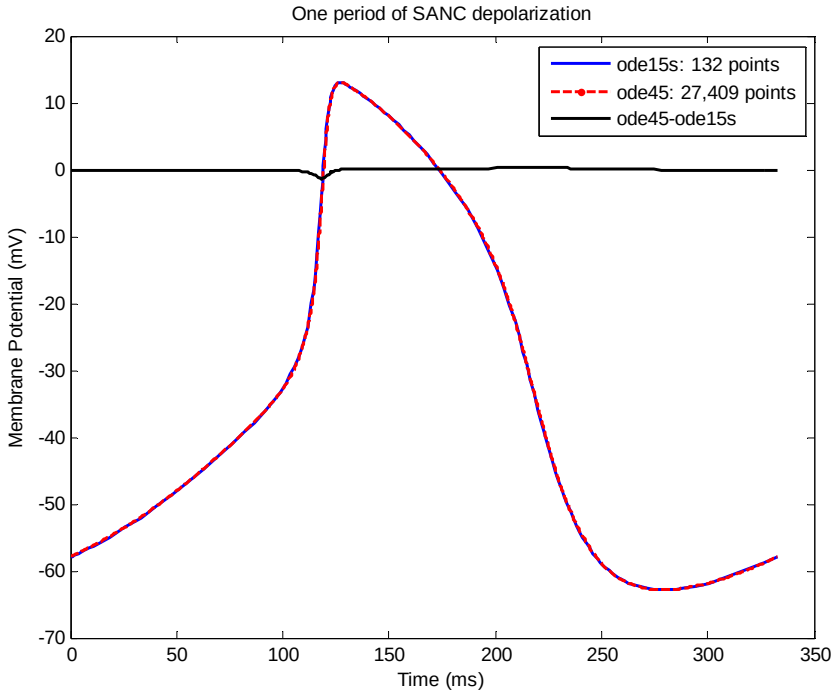
1. Implement the model in Matlab
2. Evaluate serial performance
3. Evaluate parallel performance for speed and accuracy
    a) for running one long simulation distributed across many processors
    b) for running multiple shorter, independent simulations, one simulation/processor
4. Summarize results to provide guidelines for future design of future simulation experiments

Results

1. I implemented the model. It's a nonlinear dynamical system with 29 coupled ODEs and a large set of parameters. When I do multiple simulations in parallel, I will run the simulations with different sets of initial conditions.

2. I ran some serial simulations on my processor: 2.6 GHz dual-core Intel Centrino. Although my processor is dual-core, Matlab can't use the cores appropriately for doing ODEs (there is no effect on timing when I turn the multithreading option on or off).

In running these simulations in parallel, the most interesting thing I learned is that the system is stiff. The figure below shows one period of the simulated action potential solved with two ode solvers. The results are nearly identical, but the stiff solver ode15s only evaluated the ode function at 132 points, compared to the 27,409 points evaluated by the non-stiff solver ode45. This implies that the stiff solver runs about 200 times faster! The two solvers conform to the same error tolerance, and the difference between the two simulations is shown with the black line, which verifies that the two solutions are fairly similar, with the largest deviation occurring during the quick up-stroke at around 110ms. A large error here doesn't count for much since the function is almost vertical, and for my purposes, I am really mostly concerned with *initiation* of the action potential and the location of the peak rather than the upstroke.

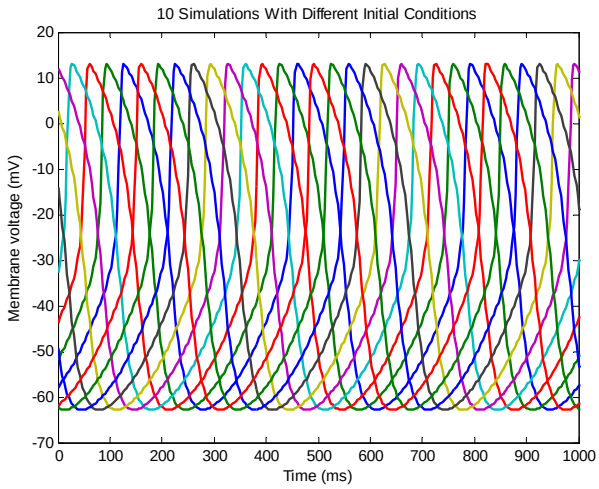


3. I had some difficulty doing the timing experiments for the parallel implementation. This is because ode15s was not implemented in Star-P yet! Fortunately, Dr. Edelman pointed me to Viral Shah at ISC, who brought some more people on board, and they were able to quickly make a task parallel ode15s solver. Because this was done very quickly, there are some limitations to using the solver in Star-P, which I will touch on below.

a) In order to do one long simulation distributed across many processors, the solver would have to have data-parallel capability. The ode15s that Viral et al. put together for me did not have this ability, and Viral referred me to Sundials, which is a suite of parallel ODE solvers developed at Lawrence Livermore National Lab. I was able to install and run the Sundials ode solver CVode on my computer (Sundials has a nice interface to Matlab, so the solver is actually pretty easy to use). I also tried to install it on the parallel machine I was using to run my experiments (Godzuki), but I couldn't do it in time. Thus, I could not do the data-parallel part of this project. This is not a huge loss because when I actually go to solve my problem, it will likely involve many shorter simulations (with various initial conditions and model parameters) rather than one very long one.

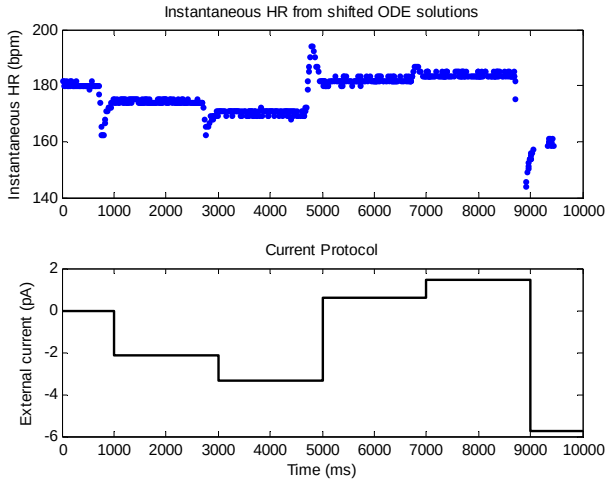
b) The bulk of my work was done on this part. I simulated my model for 10 seconds, running 10 simulations on each processor. I stored the simulation results and timing information so that I could compare accuracy and run time.

To run multiple simulations, I used the same equations each time, but with different initial conditions. To get various sets of initial conditions, I sampled the simulation over one cycle so that the different initial conditions would give time-shifted solutions. For example, the figure below shows 10 simulations:



To make the simulation procedure more relevant/interesting, I added an external current to the model. This external current would change the depolarization rate of the cell and thereby alter the heart rate. By using many shifted simulations on the same external current signal, it is possible to get a measure of instantaneous heart rate, as shown in the figure below. What's plotted in the figure is the instantaneous

heart rate, which is calculated by finding the times at which the maxima of the membrane voltage occur, say at some t1 and t2, and the instantaneous HR at any time t1 is 60/(t2-t1) beats per minute. By using many initial conditions, we get a better sampling of this instantaneous HR signal (this is a rabbit model, so the HRs are pretty fast), which allows us to see the transient overshot whenever the current steps. As an aside, this is an interesting phenomenon that I didn't expect to encounter since the cell acts like an integrator, so I thought it would just smooth out the current signal.



In order to distribute my simulations across the processors, I ran ode15s within a ppeval call. A limitation of running a function with ppeval is that the outputs of all the functions must be the same length. This is not necessarily guaranteed when using an ode solver since they utilize an adaptive time step. So, when useing ppeval to solve a system of equations with various initial conditions, one must specify the time points at which the values should be returned. By specifying all the time points, you do not change the step size that Matlab chooses as it's solving the system (because this would alter the error tolerances) but rather the simulation runs as it normally would, and then the result is interpolated at the specified time points. Therefore, it seems that this should add a linear cost (in terms of time) to the simulation. To quantify this, I ran some simulations on my (serial) computer using ode15s. To keep things fair, I chose the time points such that the length of the vectors returned by the adaptive timestep algorithm was the same as that returned by the algorithm with specified timepoints:

Table 1 : Cost of specifying time points for ode15s

Simulation Duration	Time **without** specified time points	Time **with** specified time points	Percent increase in time
10s	3.9	4.3	10.3%
30s	11.8	13.0	10.2%
60s	24.0	26.0	8.3%

Thus, the costs of running ode15s with specified time points is about 10% of the regular run time for my particular problem, and the interpolation becomes slightly less expensive as the simulation duration increases. However, I later found out that when I call ode15s on Godzuki using ppeval, what actually

runs is the program CVode from Sundials. When I did a similar comparison as above using CVode, I saw a similar trend (see table below). As simulation duration increases from 10 to 30 to 60 seconds however, the cost of interpolation goes down very quickly. This is probably because of memory allocation issues. When the function knows the desired time vector, memory can be allocated upfront, however when you let the algorithm use an adaptive step size, it's impossible to allocate memory at the beginning, so the results matrix has to be grown a couple of times during evaluation. The cost of growing the results matrix starts to close the gap. To answer the question of whether the cost of memory allocation ever *exceeds* that of interpolation, I ran the simulation for 150s, and the answer is no (specifying the time points was ~10% slower in this case). It's interesting that the percent increase in time is not monotonic as simulation length increases. This is probably because of the way I grow the results matrix when the time vector is not specified.

To grow the matrix, I begin with a size of 1000, and once this gets filled up, I calculate how much simulation time was covered using 1000 evaluations, and I estimate how many more evaluations need to be done to reach the end of the simulation interval. This is done each time the end of the allocated matrix is reached. This method may work well sometimes and badly at other times, leading to the observed timing fluctuation. This way of growing the matrix is worth it however, because growing the matrix line by line for each evaluation leads to about a 2x slowdown.

Table 2: Cost of specifying time points for CVode

Simulation Duration	Time **without** specified time points	Time **with** specified time points	Percent increase in time
10s	2.2	3.1	37%
30s	7.8	9.2	18%
60s	15.6	16.0	2.3%
150s	37.8	41.6	10.0%

From doing this study, I also realized that CVode is about 1.6 times faster than ode15s (the absolute and relative error tolerances were equal). This is good to know for solving ODEs in the future, and the implication for this project is that I should compare performance of CVode (instead of ode15s) in serial and parallel. The table below shows the timing data:

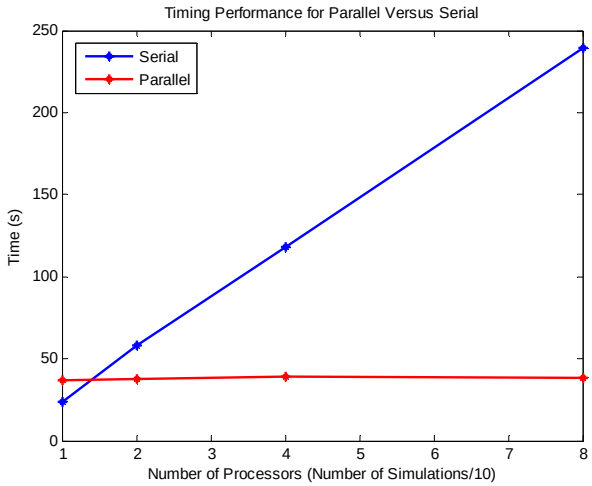
Table 3: timing for CVode solver serial versus parallel

# of Parallel Processors	# of Simulations	Simulation Duration	Parallel Time	Serial time (on my laptop)	Serial time (on godzuki)
1	10	10s	36.9	23.4	39.8
2	20	10s	37.8	58.4	Crash
4	40	10s	38.8	118.2	Crash
8	80	10s	38.7	239.2	Crash

The first line above is very instructive. In this case, all the computation was done on a single processor. Parallel time was obtained by running a single ppeval call. Serial time was either done on my laptop with 10 CVode calls, or on Godzuki with 10 ppeval calls (each one with just a single initial conditions vector).

Since my processor is 2.6GHz, and the processors on Godzuki are 2.4GHz, I can roughly estimate the time spent in communicating by scaling the times from my laptop by 2.6/2.4. This gives 25.4 seconds on my laptop, implying that the communication cost is about 11.5 seconds. Interestingly, the parallel times do not grow very quickly as the number of simulations increases eightfold, implying that the communication cost is mainly due to setting up the connection rather than the actual data transfer. Using pptic and pptoc to estimate communication time supports this idea, because sending 10 times more information does not take 10 times as long (but pptic and pptoc are pretty rough at estimating communication times, so I'm not sure if this can be trusted). Pptic and pptoc estimate about a 1-2 second communication time, so my GHz correction above probably isn't right (possibly due to other jobs running on Godzuki at the same time). Also, it is strange that the serial time on Godzuki isn't longer than what we see, since the client communicates with the server on 10 separate occasions. In all, it's most reasonable to go with the pptic/pptoc estimate of communication time, which is about 1-2 seconds regardless of data size within the range of data sizes I tested.

The figure below summarizes the run times:



In running the parallel simulations, I always loaded each processor with 10 simulations of 10 seconds duration each. In doing the serial counterpart, I just ran the same number of simulations (10*number of processors) but on a single processor. The figure shows the desired/expected result: as the number of simulations is increased, the runtime for the serial algorithm increases linearly, whereas the parallel time stays constant. This is an excellent result, showing that there are no significant communication costs building up that would make the parallel implementation worse than serial. It also shows that having N processors allows you to do N times as many simulations in the same amount of time. The result is expected given that the problem is embarrassingly parallel, but it's nice to have confirmation that performance matches theory. The offset between the serial and parallel case for the point at N=1 differs because of different processor speeds and the possibility that other tasks were sharing the processors on Godzuki, as mentioned above.

One interesting thing to note is that there were some problems with the ode solver implementation on Godzuki. Looking at Table 3, we can see that certain simulations crashed when run on Godzuki. This is

probably due to the fact that ISC put together the solver for me in a very short amount of time, and the problem can probably be overcome, but it illustrates an important drawback of parallel computing: things crash often and they're hard to debug. From my experience with parallel computers in this class, I've had the client/server connection be interrupted or stopped for no obvious reason and I've had programs cause errors that I was unable to debug because the errors were in some internal code on the server that I didn't have access to.

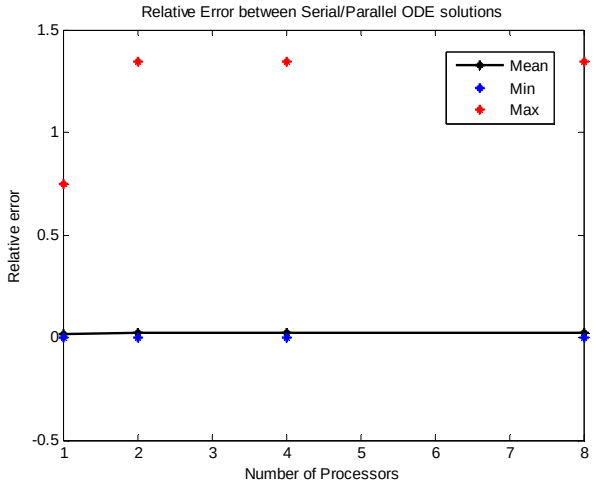
In running these simulations in particular, I know that my code is error-free, and I can run any number of simulations of any length on my laptop without a problem (using the same ode solver as Godzuki). But, as soon as I run the code remotely on Godzuki, I run into limitations on simulation number and duration that will run without crashing. For example, using a single processor on Godzuki, I can run 10 10s simulations, but not 20 or more. Similarly, on one processor, I can't run a single simulation if it's longer than ~20 seconds. Similar problems were encountered when using multiple processors. The error messages returned indicate that at some point in the numerical integration, the differential equation file is called with an empty state vector from within one of the ode solver functions (the same function does not generate any such problem on my machine). Also, the crashes occurred at random times throughout the set of simulations pointing to some processor instability/overload rather than a problem with my specific program (my program is deterministic; so if any crashes do occur due to numerical problems, they should happen at the same point in the program each time).

These reliability questions bring us to the issue of accuracy. Both the serial and parallel simulations were run using the same relative and absolute error tolerances. I calculated the absolute errors by subtracting the solutions obtained on Godzuki versus on my laptop. To get relative errors, I normalized the absolute error by the average of the absolute value of the signal. The error results are shown below.

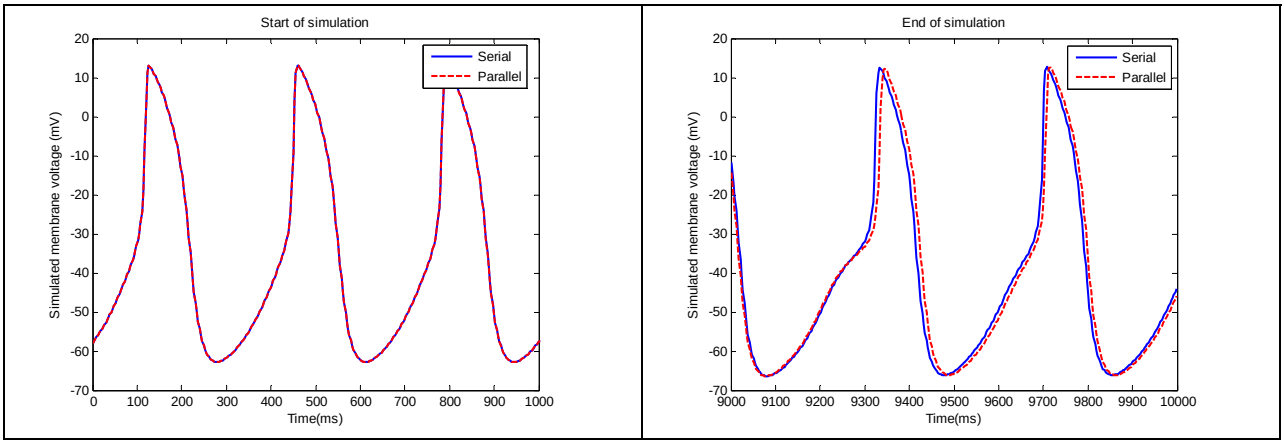
Table 4: Absolute and relative error between serial and parallel implementation

	Absolute Error			Relative Error		
Number of processors	Mean	Min	Max	Mean	Min	Max
1	0.6899	0	29.5148	0.0175	0	0.7508
2	0.8826	0	52.8492	0.0225	0	1.3447
4	0.8301	0	52.8492	0.0211	0	1.3446
8	0.8427	0	52.8492	0.0214	0	1.3446

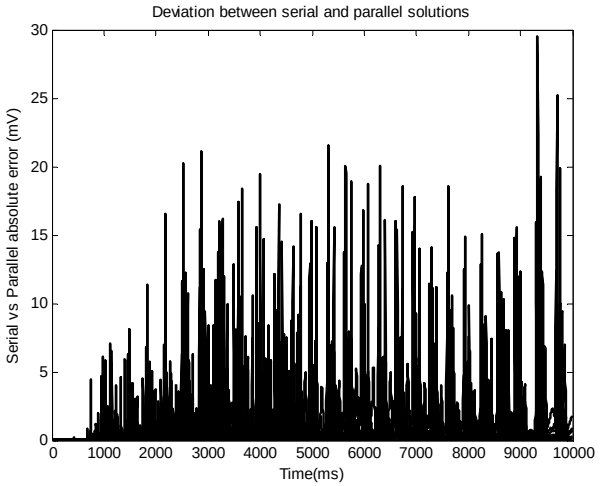
The relative error was no more than 2.25% on average, having a minimum error of 0 and a maximum error on the same order as the value of the function being integrated (relative error ~1). The figure below shows the Relative Error data from Table 4.



These average errors are acceptable, but the large maximum error is concerning. To get more insight into the nature of the error (is it just noise, or is it somehow systematic), we can look at a plot of one serial simulation versus the same simulation done on Godzuki at two points, at the start of the simulation and at the end:



We see that the serial and parallel simulations look exactly the same at the beginning, but by the time we get to the end, there is a lag. This lag contributes to the error, and because of the fast oscillatory nature of the function, this is why we get the large absolute errors. To illustrate how quickly the simulations go out of synch, the figure below shows the absolute difference between the 10 parallel and serial simulations done on one processor (on my laptop (serial) and on Godzuki (parallel)). The error plots for the other sets of simulations look the same.



This figure shows that the solutions are fairly similar for the first 700 ms, and after that, they fall out of synch, showing periodic large errors. The errors are periodic because of the shape of the action potential: even a small shift will manifest as a large error if the slope of the function is high, and this happens at the upstroke and repolarization phases of the action potential. Also, the fact that the error is 0 at the very beginning confirms that it isn't just a difference in initial conditions that causes the errors, but that the errors are actually a byproduct of the integration process.

What causes these errors? Since the problem is embarrassingly parallel, there is no data communication between processors during the integration, so no precision is lost there. Also, the programs used to integrate the solutions are identical, so there are no algorithmic differences. The only difference is that the solvers are run on different processors (my laptop is an Intel Centrino Pro, and Godzuki has AMD Opteron 880s). The differences we see probably arise due to different rounding/precision conventions of the two architectures. With that said, it's strange that the solvers claim to be have a relative error tolerance of 0.001, but the actual deviation between the solutions is about 10 times that much. This is an important point to keep in mind when running jobs on different machines, and especially when using a cluster of computers with dissimilar processors.

Finally, another interesting problem with running the parallel simulations was random number generation. I initially wanted to generate random values for the external current signal driving the simulation. In order to keep the results the same, I set the state of the random number generator to 0 at the start of each simulation. I thought that this would work and produce the same random vector each time I ran the simulations regardless of where the simulations were being run. This turned out to be wrong. I found that although I set the random number generator state to be the same on my laptop and on Godzuki before running each simulation, the two gave different random numbers. This may be because of different versions of Matlab installed on my laptop versus on Godzuki, or different implementations of the random number generator. I got around the problem by just hard-coding a random vector into the ODE function in order to keep things consistent.

4. This project illustrated the power as well as some limitations of parallel computing for solving systems of ODEs.

My main aim was to evaluate the ease of use and reliability of parallel computing with Star-P. My experience may have been a 'worst case' scenario because the function I wanted to use was not yet implemented in Star-P, so a lot of time went into enabling this capability in the first place. Once the function was in place and I could use it, I realized that there were some instabilities that prevented me from running arbitrarily long simulations on the parallel machine. Finally, when I compared the results generated on my laptop versus on the parallel machine, the results diverged over the course of a simulation. This observation brings into question the ode solver's error tolerances and illustrates once more the limitation of finite precision and different processor architectures, and the confusion that they may engender.

On the positive side though, I was able to get the expected speed-up by distributing the problem across multiple processors. Although the individual processors on the parallel machine were slower than the processor in my laptop, distributing simulations across 8 of those slower processors was about 8 times faster than my single processor.

Taking all the times into consideration, this is my takeaway:

- If you can run all your simulations in ~8 hours, it's better to just do it on your laptop overnight. The initial coding is faster, debugging is easier, and you don't have to worry about the effects of different processors on the data.
- If simulations take on the order of days, then it pays to distribute the work on a parallel machine. In this way, you can still get all the results overnight, and make any necessary changes if there are logical errors in the program. Just bear in mind that the parallel machine may crash through no fault of your own (especially if you don't have exclusive use of the machine) and you might have to save data periodically or restart the work from scratch.

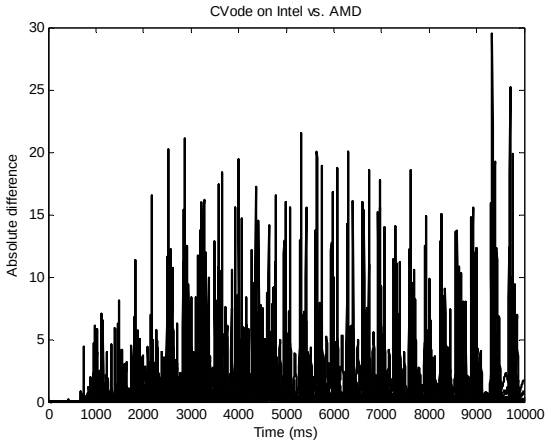
I am also glad that I was introduced to the Sundials suite of ODE solvers, because that gives me an almost 2x speed-up over ode15s in Matlab, and that itself can make the difference between tying up my laptop all day, or just overnight.

Finally, many thanks go to Viral Shah, Andy Greenwell and Raghunathan Sudarshan for putting the stiff solver together and giving me time on their parallel machine. One thing that puts me more at ease about one day relying on parallel computing is that the support I got from these folks was excellent, and I feel that given enough time, they can make any problem work.

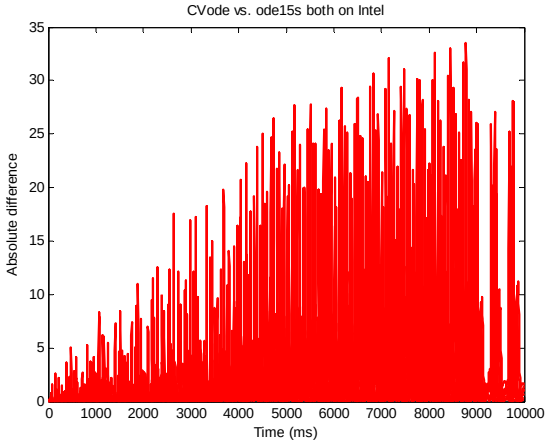
Appendix

Here is a comparison of the divergence of solutions for the two processors and two ode solvers:

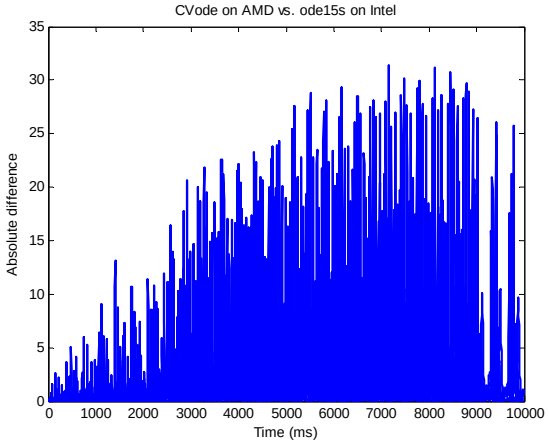
Same solver, different processors:



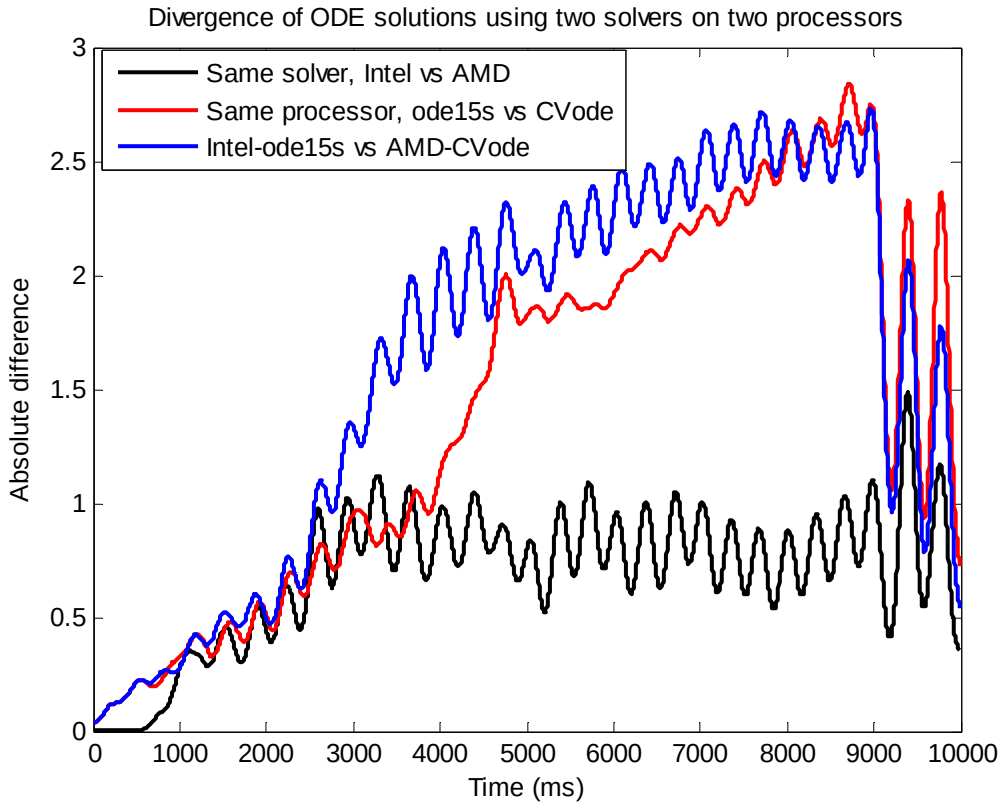
Same processor, different solvers



Different Processor, Different Solver



Summary: Mean of all 3 cases



In this figure, the mean was taken across 10 simulations, and this mean error was filtered with a Gaussian filter with standard deviation of 20 samples (60 ms). Because of the mean and filtering, the absolute differences are now on the order of 1-2 instead of 20-30 as in the previous 3 figures. It's interesting that the error for the (same solver, different processor) case starts at 0 and seems to plateau after 3000ms, whereas the other two cases start above 0 and continue rising. Perhaps they plateau too, just at a higher value. The error drops again around 9000ms because the last external current step is pretty negative (see figure on page 4), so it tends to reset the model.