insilicoSim: an Extendable Engine for Parallel Heterogeneous Biophysical Simulations

Eric M. Heien
Department of Computer Science, Graduate School of Information Science and Technology
Osaka University, Japan
e-heien@ist.osaka-u.ac.jp

Masao Okita
Department of Computer Science, Graduate School of Information Science and Technology
Osaka University, Japan
okita@ist.osaka-u.ac.jp

Yoshiyuki Asai
The Center for Advanced Medical Engineering and Informatics
Osaka University, Japan
asai@bpe.es.osaka-u.ac.jp

Taishin Nomura
Department of Mechanical Science and Bioengineering
Graduate School of Engineering Science
Osaka University, Japan
taishin@bpe.es.osaka-u.ac.jp

Kenichi Hagihara
Department of Computer Science, Graduate School of Information Science and Technology
Osaka University, Japan
hagihara@ist.osaka-u.ac.jp

ABSTRACT

Recently, several multidisciplinary projects have begun to model and simulate human physiological systems. However, the simulators for these models are often limited in terms of simulation type and lack of parallel computing support. In this paper we describe *insilico*Sim, an extendable simulation engine for performing parallel large scale biophysical simulations. We present three key components of the simulator for improving extensibility and performance. First, we demonstrate how a standardized plugin interface allows for easy extension of the simulator to new types of input, output and simulation methods. We detail a technique for improving simulation performance by simplifying and compiling simulation related mathematical expressions into an internal byte code representation for fast evaluation. Finally, we describe the simulation object manager which allows for shared object access between simulation interfaces while transparently performing parallel synchronization. We demonstrate the effectiveness of these methods by simulating several models on both serial and parallel computing platforms.

Categories and Subject Descriptors

I.6.7 [**Simulation and Modeling**]: Simulation Support Systems—*environments*; I.6.8 [**Simulation and Modeling**]: Types of Simulation—*continuous, parallel*

General Terms

biophysical simulation, physiome, parallel computing

Keywords

ACM proceedings

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
SIMUTools 2010 March 15–19, Torremolinos, Malaga, Spain.
Copyright 2010 ICST, ISBN 78-963-9799-87-5.

1. INTRODUCTION

In the past decade, research efforts such as the Human Genome Project have demonstrated the successful application of computation to the field of biology in order to map the human genome. Recent projects are working to extend this to the human physiome, and include the Virtual Physiological Human [4], the IUPS Physiome project [11] and the in silico Medicine initiative [17, 1]. These efforts aim to create multiphysics and multiscale models of human physiological systems and perform computer based simulations with the models. The simulations help researchers understand complex physiological phenomenon, for example, the effect of drugs in causing cardiac arrest [16].

Currently available simulators for these types of models are often limited in their scope and functionality. Simulators are available for molecular level simulations [8] or cellular particle simulations [12] that offer parallel computing support. However, these focus on physical phenomenon at a much smaller level than required for many physiological simulations, which can operate at tissue/organ level or higher. Physiologically oriented parallel simulators tend to be very domain specific and focus on one type of model. These are usually specific to neuronal networks [18] or cardiac modeling [15]. Other simulators allow general models described with domain specific markup languages [3] [14], but do not offer parallel computing needed for large scale models. To the best of our knowledge, the simulator described in this paper is the first generalized physiome oriented simulator with parallel computing abilities.

In this paper we describe the structure and function of the *insilico*Sim program which performs parallel simulations of heterogeneous biophysical models. In Section 2 we present background about the simulator and what problems it solves, then detail the target biophysical models in Section 3. The core simulation engine and interfaces are described in Section 4. Techniques for improving simulation performance and enabling parallel simulations are described in Section 5. We experimentally demonstrate the effectiveness of these techniques using several models in Section 6 and offer discussion and conclusions in Section 7.

2. BACKGROUND

In this section we describe the background behind *insilico*Sim and related simulators. We present some of the problems with biophysical simulations and how *insilico*Sim attempts to solve these problems.

*insilico*Sim is designed for use in conjunction with a biophysical model development environment called *insilico*IDE. Previously, simulations were performed by generating C++ source code for a given model from *insilico*IDE, then compiling and executing it to view the results. In previous work [7] we investigated methods for performing parallel simulations with the generated source with some success. However, as models become larger the compilation time grows significantly. In addition, this type of simulation made it difficult to add support for new data types, simulation techniques and input model types.

To solve these problems, we developed *insilico*Sim. In designing *insilico*Sim there were three main goals, each of which presented difficulties:

1. Make the system easily extendable to additional input/output data formats and simulation techniques.

2. Allow parallel simulation of large scale models over multiple processors.

3. Achieve high performance, comparable to model compilation and execution in an interpreted language.

The primary aim of *insilico*Sim is to allow simulation of heterogeneous biophysical models. By heterogeneous, we mean the models may be represented using different formats, and may contain multiple interacting elements simulated as algebraic functions or ordinary differential equations with many parameters. Furthermore, there are a variety of methods available to simulate the models and the simulator must allow easy addition of other methods. To solve the problem of allowing simulator extensions, we developed a common interface to the simulator (Section 4.2). To enable common access to the model across all interfaces, we implemented a shared object manager (Section 5.2).

The next aim of *insilico*Sim is to enable parallel simulations while allowing heterogenous models and extensions. Parallel simulations are necessary because models may become very large (thousands or millions of elements) and using a single computer may be impossible due to time or hardware constraints. The majority of programs and libraries used for biophysical simulation are serial, and large scale parallel simulations must often be implemented by hand and tailored to simulate a specific model. Parallelizing an extendable simulator is difficult because different extensions will access different simulation objects in different ways. To implement this we use the shared object manager (Section 5.2) to create global communication plans then transparently perform parallel synchronization during the simulation.

Finally, if *insilico*Sim cannot achieve relatively high performance simulations then it may be wiser to stay with the previous technique of compiling model source files. This is difficult because the internal tree structure of the model in *insilico*Sim is suited to easy manipulation rather than fast computation. To achieve high performance, we use standard compilation techniques to convert this internal representation into an internal byte code, and remap abstract values onto arrays for fast access (Section 5.1).

3. EXAMPLE MODEL

In this section we describe the representation of a model in *insilico*Sim using an example. This section is intended as a background for readers unfamiliar with physiological modeling. Figure 1 shows the dependency graph of the example model - a Luo-Rudy model of ventricular cardiac action potential [13] with an external stimulus. We will use this model to describe data and calculation in the simulator.

Models in *insilico*Sim consist of objects and values. The objects represent the calculations to be performed and the values represent the results of those calculations. The simulator supports four types of values in calculations (double, vector, matrix, undefined), but for simplicity we focus only on 64-bit double precision values in this paper. *insilico*Sim supports three types of objects:

1. Ordinary differential equations (ODE) - represent equations of the form $\frac{dy_i}{dt} = f(y_0, \ldots, y_n, t)$. In biophysical models these represent things such as the rate of change of intracellular ion concentration.

2. Function expressions - represent standard function expressions of the form $f(y_0, \ldots, y_n, t) = \ldots$. These expressions cannot be self-referential. They represent, for example, ion gate currents at a particular moment in time given surrounding ion concentrations.

3. Static parameters - represent static simulation parameters, such as cell membrane permeability or other physical constants.

In Figure 1, these object types are represented as boxes (ODEs), ovals (function expressions) and trapezoids (static parameters). In addition, ghost objects are used in parallel simulations to represent objects on other nodes. Each of these objects (except the ghost type) have associated mathematical expressions stored in a tree structure in the simulator. This model has 8 ODEs, 31 function expressions and 27 static parameters. There are a few points about the model worth noting.

First, there are a large number of static simulation parameters towards the top of the graph representing things such as stimulus timing, membrane permeability, and so on. Roughly one third of the model objects are static parameters, which is consistent with many biophysical models. Usually these parameters are specified at the start of a simulation to test their effect on the overall system behavior.

Second, most ODEs and functions reference values from several other ODEs and functions to calculate their own value. This means the model cannot be cleanly separated

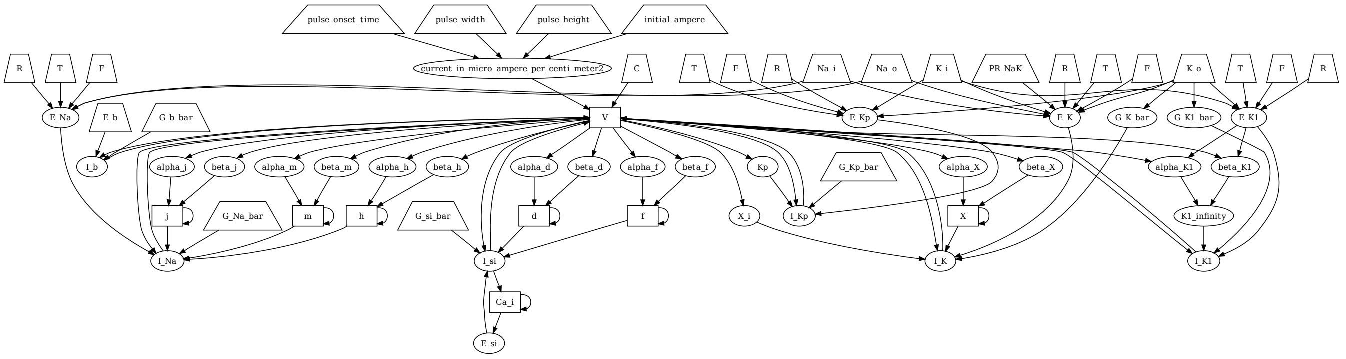


Figure 1: Dependency graph of the Luo-Rudy model of ventricular cardiac action potential with external stimulation. Boxes represent ODEs, ovals represent function expressions and trapezoids represent static parameters. Arrows represent data dependencies between objects.

Table 1: Calculations needed to approximate ODE for V in Luo-Rudy model.

#	Type	Calculation	References
1	Param	K_i	None
1	Param	K_o	None
1	Param	T	None
1	Param	F	None
1	Param	R	None
2	Function	G_K1_bar	K_o
2	Function	E_K1	K_i, K_o, T, F, R
3	Function	alpha_K1	V, E_K1
3	Function	beta_K1	V, E_K1
4	Function	K1_infinity	alpha_K1, beta_K1
5	Function	I_K1	V, K1_infinity, E_K1, G_K1_bar
...	...	...	...
...	ODE	V	I_K1, ...

into different types of calculation. Neither can the function expressions be easily substituted into the ODE expressions, because they have distinct physical meaning and their values can be referenced by multiple other expressions.

Finally, functions can iteratively refer to other functions, for example, I_K1 (lower right of Figure 1) refers to the value of K1_infinity, which refers to alpha_K1 and so on. This means that evaluating functions in the correct order is critical to simulation correctness.

Table 1 shows a subset of the calculations the simulator performs when approximating the value of V and the relative order they must be calculated in. Calculating V requires the value of several functions, which in turn have dependencies on other functions, ODEs and parameters. There is an order of calculation that must be followed to obtain the correct result. For example, alpha_K1 requires the value of E_K1, which in turn requires four parameters. Therefore, these parameters must be evaluated before calculating E_K1, which must be evaluated before calculating alpha_K1.

The simulator must therefore create an internal representation of the model, analyze the representation to determine the correct calculation ordering and variable relationships, and perform the required calculations in the correct order during each simulation step.

4. INSILICOSIM

*insilico*Sim is a program written in C++ designed to perform simulations of biophysical models specified by markup languages such as SBML (Systems Biology Markup Language) [10], CellML [5] and ISML (in silico Markup Language) [2]. It is designed to allow easy addition of functionality through interfaces, while transparently performing parallel synchronization and achieving high performance.

*insilico*Sim performs simulations by using interfaces which specify the actions to be taken at each simulation step. Examples of interfaces and their behavior are shown in Table 2. Interfaces are enabled by the user depending on what sort of simulation behavior they desire. The base interface is designed to be easily extended by users to support other types of computation and input/output formats. A core simulation engine connects these interfaces and provides common methods to access and manipulate objects related to the simulation.

In Section 4.1 we describe the core engine, how it is called by the user and how it interacts with the interfaces. The interface concept is described in Section 4.2, with some implemented interfaces described in Section 4.3.

4.1 Core Engine

The core engine of *insilico*Sim is responsible for managing simulation objects and values, and performing synchronization and load distribution in parallel simulations. It also calls the requested interfaces during the simulation. It uses the Zoltan library [6] for parallel load balancing and MPI for parallel communication.

The pseudocode for initializing and running the core engine is shown in Algorithm 1. The simulation length and time step size are defined when the simulator is created. Before initializing the simulator, all relevant interfaces must be registered using registerInterface(). The simulator also calls init() and finalize() on each interface during the respective simulator calls. The simulator advances the current simulation time during each call to advanceStep() until it has reached the total simulation time.

4.2 Interface Concept

To allow easy extension of *insilico*Sim, we designed a base interface class which is inherited by derived classes. Depending on the desired functionality of the derived class, it will override one or more functions which are called by the core engine. This allows developers to add functionality to the simulator while smoothly interacting with other interfaces

Table 2: List of available interfaces for *insilico*Sim.

Interface	Category	Function
Random	Import	Generates a random model for testing.
ISML	Import	Parses an ISML file, and converts it to the internal simulation object model.
SBML	Import	Parses an SBML file, and converts it to the internal simulation object model.
CVODE	Computation	Performs ODE approximation using the CVODE library with adaptive time stepping.
Euler	Computation	Performs ODE approximation using the Euler method.
RK4	Computation	Performs ODE approximation using the 4th order Runge-Kutta method.
DOT	Export	Exports the simulation model as a DOT graph file.
Print	Export	Prints simulation results to a file in a user specified format.
Progress	Misc	Periodically displays simulation progress and estimated time to completion.

Algorithm 1 Pseudocode to Run Core Engine

1: simulator = new *insilico*Sim(total simulation time, time step)
2: **for all** interfaces **do**
3: simulator→registerInterface(interface)
4: **end for**
5: simulator→init()
6: **while** !simulator→isSimDone() **do**
7: simulator→advanceStep()
8: **end while**
9: simulator→finalize()

and the core engine. We describe the interface functions below and how they interact with the core engine.

```
void init(InitOptions &init_options,
   set<ObjSetRequest> &req_set)
```

This interface function is called during the simulator `init()`. In this function the interface performs necessary initialization, for example, opening a file for writing data. The interface also should provide details to the core engine about its function. The `init_options` object lets the interface notify the core engine about its computational load (for parallel load balancing), if it uses compiled objects and if it should be timed by the core engine. Through the `req_set` object, the interface notifies the core engine of what type of objects it will read/write, which is used by the object manager to order calculations and perform parallel synchronization.

```
double getObjectWeight(DataObj *obj)
```

This function is used in parallel computing to perform load balancing. When called, the interface returns the computational weight of the given object for this interface. This allows more accurate load balancing without being specific to certain types of computation.

```
void createInitialObjects(DataObjSet &obj_set)
```

Any interfaces that create objects (ISML, SBML, random) do so in this routine and pass them back to the core engine through the DataObjSet object manager. The DataObjSet is used to assign globally unique identifiers to new objects. The core engine also provides graph and model analysis tools to help create the simulator representation of a model.

```
void objectsRebalanced(DataObjSet &new_obj_set,
   DataValSet &new_val_set, SimCurState &params)
```

This function is called during initialization and after load balancing is performed by the core engine. This can be used to reset interface specific structures for the new set of objects on the processor.

```
void advanceStep(DataObjSet &obj_set,
   DataValSet &val_set, SimCurState &params)
```

This function is called once during each simulation step. The interface performs relevant calculations using the objects in DataObjSet and the values in DataValSet and stores the results back in the DataValSet. This way the results from one interface are available to the others. The SimCurState object contains information about the current simulation state, such as simulation time and MPI processor rank.

4.3 Example Interfaces

Next we describe some of the currently available interfaces for *insilico*Sim. These range from an interface to construct a simulator model from ISML model files (4.3.1) to different types of solvers for the model ODEs (4.3.2 and 4.3.3).

4.3.1 ISML

The ISML interface parses ISML (in silico Markup Language) model files and creates an internal representation of the model. An explanation of ISML is outside the scope of this paper, details are available in the specification [2].

This interface uses the Xerces XML parsing library to read an ISML file. It then constructs a corresponding model in the simulator. Because the core engine uses a common structure to represent expressions, there is no fundamental difference in which markup language is used to represent a model. In the future, users will be able to specify how components of different models relate, and perform simulations combining models from different modeling languages.

4.3.2 Euler and Runge-Kutta

The Euler and Runge-Kutta (RK4) interfaces use the corresponding approximation techniques to approximate ODEs. Each of these were written without regard to the type of model or expressions being evaluated. They use the object manager interface to iterate through the objects on a processor, perform the necessary evaluations, and store the results back in the shared value manager.

The Runge-Kutta method provides a good test of the object manager for two reasons. First, it must perform communication multiple times during a single time step. Second, it must communicate different values depending on which midpoint calculation it is performing. Details of how the object manager achieves these are described in Section 5.2.

4.3.3 CVODE

This interface uses the SUNDIALS library [9] CVODE solver to approximate ODEs using adaptive time stepping. Because of the adaptive time stepping technique, the interface may require multiple communications during each time

step where the number of communications cannot be predicted ahead of time. This type of communication is handled correctly by the object manager. The implementation of this interface demonstrates how the simulator can easily incorporate other libraries to perform simulations.

5. OPTIMIZATION TECHNIQUES

Next we describe two techniques used in the simulator to improve performance and functionality. Section 5.1 describes a method to improve simulation speed by compiling objects from their normal tree structure to a simplified byte code representation. In Section 5.2 we explain how data dependencies and parallel computation are transparently managed by using an object manager.

5.1 Compiled Objects

In this section we describe the technique of compiling objects to internal byte code in the core engine to improve simulation speed. Mathematical expressions in *insilico*Sim are represented by tree structures, with variables referenced by a global identifier (GID). The global identifier in turn references an abstract data value object, which allows for different types of values (double, vector, matrix). This organization allows easy creation, manipulation and evaluation of the expressions.

However, evaluating expressions using this tree structure is slow for three reasons. First, traversing the tree structure is inefficient in terms of memory access because the tree nodes may be spread throughout memory. Second, referencing variables and calculating values in the expression is slow because data values are stored as an abstract class. Third, static parameters in expressions cause unnecessary references to variables that never change during the simulation. To improve the simulation speed, expression trees in the simulator can be compiled into internal byte code thereby ameliorating these three inefficiencies.

Compiling the expression trees is performed in three steps as shown in Figure 2. The original expression trees are shown in Figure 2(a), representing the function expression $z = x+2y$ and the static parameter $x = 1+2$. The GIDs refer to the global identifiers of each variable. Variables, static values and intermediate calculation values are all stored as instantiations of an abstract data value class.

The first step, shown in Figure 2(b), replaces references to static parameters with the computed parameter value. In this example, $z = x + 2y$ is replaced by $z = 3 + 2y$. Because of the tree structure of the expression, performing this replacement is simple and fast.

Next, Figure 2(c) shows the tree after the data values are converted from an abstract class to a normal C++ array of doubles and renumbered using local identifiers (indices in the double array). The DataObjSet object manager creates a mapping from the global IDs to the array, and uses this to remap the value references.

Finally, the modified expression tree is converted to an array as shown in Figure 2(d). Each element of the array describes what sort of instruction it is and what operand it uses. A side benefit of compiling this array is that the program can easily predict the maximum required stack size and preallocate memory to evaluate an expression.

It is worth noting that some simulators export source code for a model, compile it using an open source compiler such as GCC, then execute this to perform the simulation. This

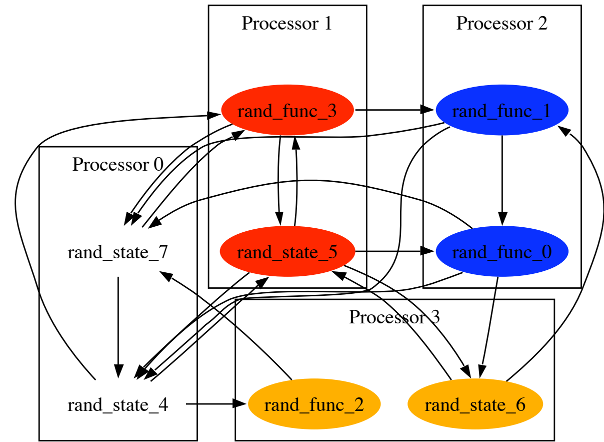


Figure 3: Random model graph produced by the Random and DOT export interfaces. The coloring represents model partitioning over 4 processors.

technique is fine for small scale calculations, but as we discovered in previous work [7] it quickly becomes intractable for parallel computation due to the complexity of object dependencies and parallel synchronization. In addition, for large models the compilation time can be much greater than the simulation time. Allowing different data types and simulation methods makes it even more difficult to generate this type of source, and packaging a compiler with the simulator presents a host of other problems, so we opted for a separate simulation engine.

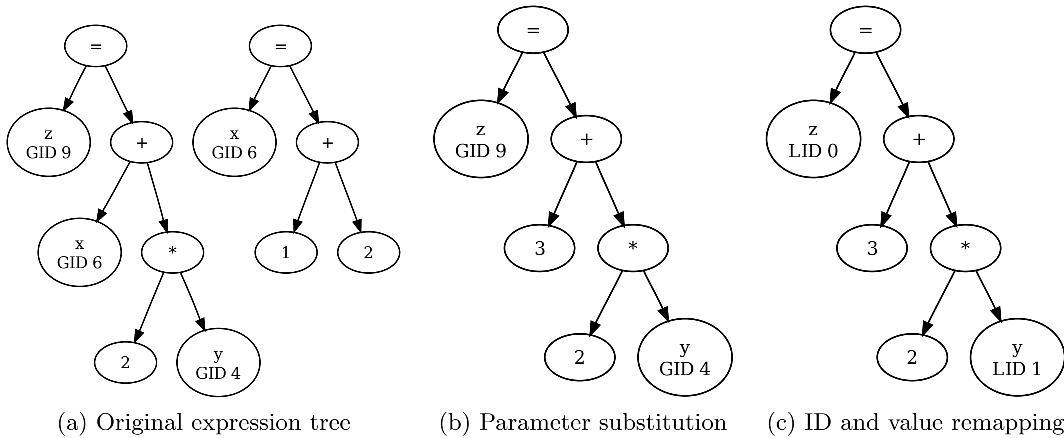
5.2 DataObjSet Object Manager

A significant difficulty in performing these types of biophysical simulations in parallel is caused by object dependencies as described in Section 3. This is particularly true for chained function dependencies because they cannot be easily simplified to a single expression. In previous work [7] we explored the idea of using redundant computation to reduce communication for function expressions, but this will not work for all models.

Therefore, when performing a simulation there are two issues that must be resolved. First, the evaluation of objects on a given processor must occur in the correct order. Second, if the simulation is divided over multiple processors, synchronization must occur such that a processor has all the values needed to evaluate an object. In *insilico*Sim we developed the DataObjSet object manager to handle both of these issues. In this section, we describe the algorithm used by the DataObjSet object manager to correctly order objects for computation and transparently perform communication.

As described in Section 4.2, each interface reports to the core engine what types of objects it will read/write. Based on this, the DataObjSet object manager creates a set of ordered computation and communication steps that will ensure correct simulation of the model in a parallel environment.

The DataObjSet object manager is called by the interface using the standard C++ iterator concept. An interface starts accessing objects by creating an iterator with a call to



(a) Original expression tree (b) Parameter substitution (c) ID and value remapping

Instruction #	0	1	2	3	4	5	6
Type	OPERATOR	VARIABLE	OPERATOR	CONSTANT	OPERATOR	CONSTANT	VARIABLE
Value	EQUALS	0	PLUS	3	MULTIPLY	2	1

(d) Compilation to byte code

Figure 2: The three stages of simplifying and compiling an expression tree into byte code.

Algorithm 2 Pseudocode of algorithm to generate communication/calculation steps

Require: Object type T
1: G = dependency graph for objects on this processor of type T
2: ObjsAvailForComm = [objects not of type T or ghost]
3: StepList = []
4: **while** any processor needs objects **do**
5:   **while** G has an object O with no unresolved dependencies **do**
6:     Add "Calculate(O)" to StepList
7:     Add O to ObjsAvailForComm
8:     Remove O from G
9:   **end while**
10:   **if** Any other processor has ghost objects in G **then**
11:     AllAvailObjs = ObjsAvailForComm lists over all processors
12:     ObjsToGet = (ghost objects in G) ∩ AllAvailObjs
13:     **for all** O in ObjsToGet, O is on processor P_i **do**
14:       Add "O ← P_i" to StepList of P_j
15:       Add "O → P_j" to StepList of P_i
16:       Remove O from G
17:     **end for**
18:   **end if**
19: **end while**
20: **return** StepList

Table 3: Expressions for example random model.

Initial Value	Expression
0.005	$F_0 = sin(S_5 + F_1)$
0.016	$F_1 = sin(S_6 + F_3)$
0.010	$F_2 = sin(S_4)$
0.030	$F_3 = sin(2S_4 + S_5 + 2S_7)$
0.562	$\frac{dS_4}{dt} = sin(2S_7 - S_5 - 2F_0 - 2F_1)$
0.250	$\frac{dS_5}{dt} = sin(S_6 - 2S_4 + F_3)$
0.869	$\frac{dS_6}{dt} = sin(S_5 + F_0)$
0.163	$\frac{dS_7}{dt} = sin(F_3 - F_2 + F_1 + 2F_0)$

the manager. This iterator points to an object that should be evaluated by the interface. Each increment of the iterator returns the next object that should be evaluated. The object manager transparently performs communication such that when the iterator references an object, all the dependencies needed for its evaluation have been resolved.

The pseudocode for the computation/communication ordering algorithm is shown in Algorithm 2. Orderings are created for each type of object combination requested by the interfaces. Object types may be mixed, for example, "function expressions and ODEs". The algorithm works by creating an ordered list (StepList) of objects to evaluate and communications to perform.

The algorithm first creates a dependency graph of the relevant objects. This allows the algorithm to track which dependencies remain to be fulfilled for a given object. It then loops until all processors have taken care of their necessary objects and the graph is empty. The first part of the loop removes objects with no unresolved dependencies from the graph, and adds them to StepList and ObjsAvailForComm. The ObjsAvailForComm list is used to notify other processors which objects have been evaluated and are available for communication.

Next the algorithm creates communication steps for any processors that require them. This involves swapping lists

Table 4: Computation and communication steps for the functions of the example random model.

	Processor			
Step	0	1	2	3
0	$S_4 \rightarrow P1$	$S_4 \leftarrow P0$		
1	$S_7 \rightarrow P1$	$S_7 \leftarrow P0$		
2	$S_4 \rightarrow P3$	$S_5 \rightarrow P2$	$S_5 \leftarrow P1$	$S_4 \leftarrow P0$
3		$Calc(F_3)$	$S_6 \leftarrow P3$	$S_6 \rightarrow P2$
4		$F_3 \rightarrow P2$	$F_3 \leftarrow P1$	$Calc(F_2)$
5			$Calc(F_1)$	
6			$Calc(F_0)$	

of all available objects between processors, finding which processor P_i has the value of object O needed by P_j and adding the communication to each of their StepLists.

Finally we present an example of how the DataObjSet object manager iterator is generated for a sample model using the algorithm described above. In this example, we use a model generated by the Random interface. The model dependency graph is shown in Figure 3, with different coloring for objects representing a division of the model over 4 processors. The expressions for the model are shown in Table 3.

Table 4 shows the steps to compute only the functions for the random model, as generated by Algorithm 2 for each processor. The communication steps have been separated for clarity - in the actual simulation these are grouped together into MPI_Request objects for efficiency. As an example, look at the expression for F_0. This requires the result from F_1, which is on the same processor. However, it also requires S_5, and F_1 requires S_6 and F_3. Therefore, steps 2, 3 and 4 consist of receiving the needed values before beginning the computation of F_0 and F_1. At the same time, F_3 must be calculated on processor 1 before being sent to processor 2, as is performed in steps 3 and 4.

This demonstrates how Algorithm 2 creates computation and communication plans for arbitrary models that ensure the correctness of the result. And because the communication occurs entirely within the DataObjSet object manager, the interface is completely hidden from the complexities of parallel synchronization.

6. EXPERIMENTS

To confirm the effectiveness of the optimization techniques proposed in Section 5 and demonstrate the accuracy of the different approximation interfaces, we performed a series of experiments. Serial experiments were performed on a MacBook 2.6GHz dual core Intel with 4GB RAM. Parallel experiments were performed on a Mac Pro 2.26 GHz Xeon 8-core machine with 8GB of RAM.

We used three different models in the experiments with characteristics shown in Table 5. The average dependencies represents the average number of references each object has to other objects. The first model is the Luo-Rudy model described in Section 3 with an average of 2.58 dependencies per object. Next we used a model of a spinal neural network for locomotor rhythm generation based on Rybak's central pattern generator [19]. This model is representative of the large scale models targeted for parallel simulations and has more connected components, with an average of 5.66 dependencies per object. Finally, we used the random model shown in Figure 3.

Table 5: Model characteristics.

Model Name	Luo-Rudy	Rybak	Random
ODEs	8	560	4
Functions	31	1000	4
Static Parameters	27	1402	0
Average Dependencies	2.58	5.66	2.5

In this section, we discuss three types of experiments. First, we examine the accuracy of each type of solver in Section 6.1. Next, we demonstrate the effectiveness of byte compiling objects in Section 6.2. In Section 6.3 we show that simulating models in parallel environments has good efficiency.

6.1 Result Accuracy

One important point when performing biophysical simulations is confirming that different solvers return approximately the same results. Because the simulator deals with differential equations which often do not have an analytical solution, the interfaces described in Sections 4.3.2 and 4.3.3 are used to obtain approximate solutions.

Each solver was activated by changing a command line option to the simulator. Because of the shared object manager and interface design described in Sections 5.2 and 4.2, no other changes to the code were necessary. For the Luo-Rudy model, the accuracy is computed for variable V (representing the membrane potential in millivolts) and for variable S_7 in the random model. The baseline for both models was computed by CVODE with a timestep of 10^{-3} and the solver interfaces each used a timestep of 10^{-2}.

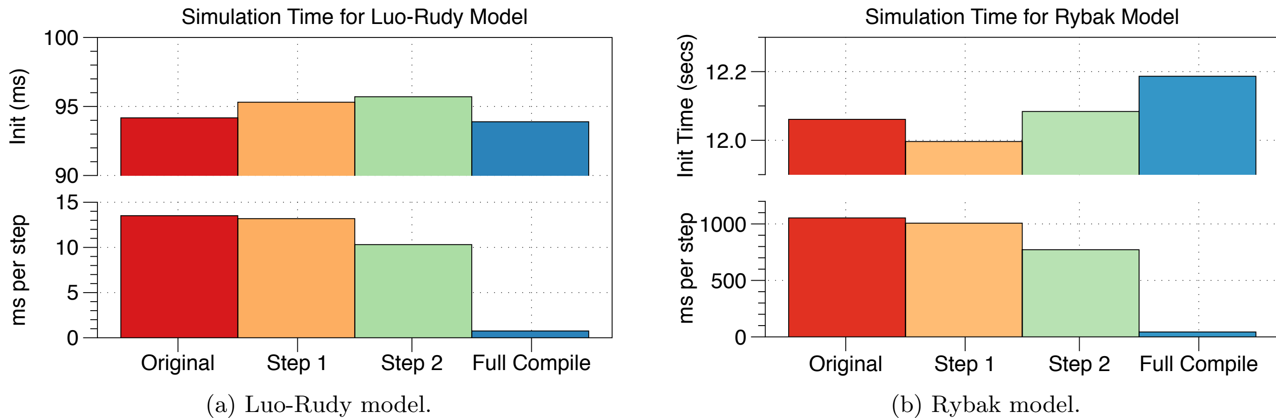
Figure 4 shows the accuracy of the three available ODE solvers with respect to a baseline solution. As shown, the error can depend on the model and point in the simulation. For the Luo-Rudy model, sudden changes in V cause high error in all the interfaces, but this soon stabilizes and all solvers finish with very low error levels. The random model is essentially an oscillating system where tiny errors will quickly accumulate. As shown in Figure 4(b), all the approximation methods end up with relatively high error, but the Euler method becomes inaccurate much sooner than RK4 or CVODE.

6.2 Compiled Objects

In Section 5.1 we described a technique for compiling calculations into a simulator specific byte code and remapping global identifiers with abstract objects into local identifiers in an array. Here, we examine the effect of each of the three compilation steps on simulation performance. We examine overall computation speed as well as cache performance as measured by the Cachegrind utility. The Cachegrind utility simulates how a program interacts with the cache and reports statistics on reads and writes to each cache level.

Figure 5 shows the simulation initialization time and time per simulation step for the Luo-Rudy and Rybak models using the Runge-Kutta interface. Figure 6 shows the number of cache references during simulation of the Luo-Rudy model. These figures show the effects on execution time and cache references after applying each step of compilation.

The figures first show the times and references for the original expression trees using abstract data values and retaining static parameters. Next, we run the simulations after sub-



(a) Luo-Rudy model.

(b) Rybak model.

Figure 5: Comparison of execution time for different levels of compilation.

stituting static parameter references with their actual values (Step 1 of compilation). Even though static parameters comprise a large number of simulation objects, this has only a minor effect on both models. This step decreases simulation time by 2.4% for the Luo-Rudy model and 4.4% for the Rybak neuron model and has a similar effect on cache references in Figure 6. This is not a significant improvement because static parameters are only calculated once at the beginning of the simulation, so the savings only comes from avoiding a lookup in the object map.

The second step of compiling objects changes the global identifier reference to a local one, and uses an array of doubles for object values rather than manipulating an abstract data value class. This step further decreases simulation time by 21.8% for the Luo-Rudy model and 23.4% for the Rybak model, for a total improvement of 23.7% for Luo-Rudy and 26.8% for Rybak. There are several reasons for this improvement. First, by using double values rather than abstract classes the simulator can avoid class type checking when performing calculations. Also, intermediate values do not need to be represented as an abstract class so the allocation and freeing of data values is avoided. However, as seen in Figure 6 there are still a large number of cache references due to traversing the tree structure.

The final step involves converting the tree structure to an array of interpreted instructions. This further decreases simulation time by 92.6% for Luo-Rudy (total decrease of 94.3%) and 94.4% for Rybak (total decrease of 95.9%). There is also a significant decrease in both instruction and data cache references, roughly 80% less than the original. There is also very little change in initialization time, showing that object compilation to interpreted byte code can produce fast simulations with little compilation time. It is worth noting that all steps in compilation result in the same simulation results.

To put this in perspective, we compared these results to execution times of a compiled simulation. Simulation with fully compiled objects gives a speed of 0.754 ms/step for the LR model and 43.5ms/step for the Rybak model. Since the Runge-Kutta interface evaluates all functions/ODEs 4 times per simulation step, this gives an average of $4.83\mu s$ to evaluate an object in the LR model and $6.97\mu s$ to evaluate an object in the Rybak model. In previous research [7] we used a model similar to Rybak that was converted into source, compiled and executed on similar processors. In this case, the average time to evaluate an object was $1.61\mu s$. This means our simulator is roughly 3-4 times slower than a compiled equivalent, on par with many interpreted languages. This difference is less significant when incorporating compilation time for large models, which took up to 10 minutes for the Rybak model.

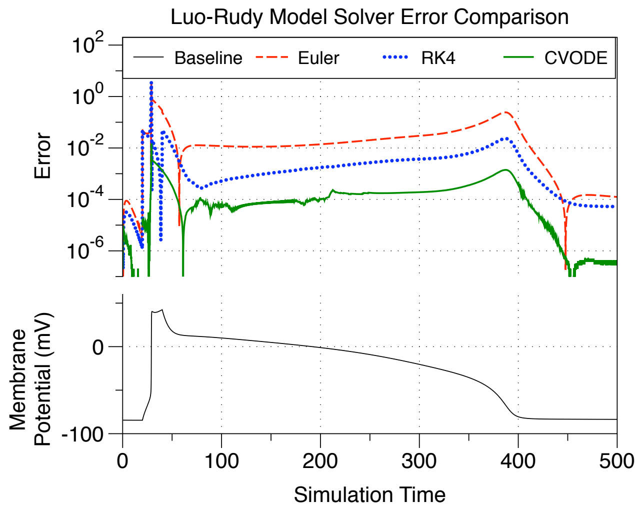
6.3 Parallel Computing Accuracy and Speed

Next we examine the parallel computing capabilities of the simulator. For this experiment, we used the Rybak model with the Runge-Kutta interface and compiled objects. The Luo-Rudy model is too small to benefit from parallel simulation. In these experiments we want to confirm that calculation scales well with more processors and communication does not grow excessively.

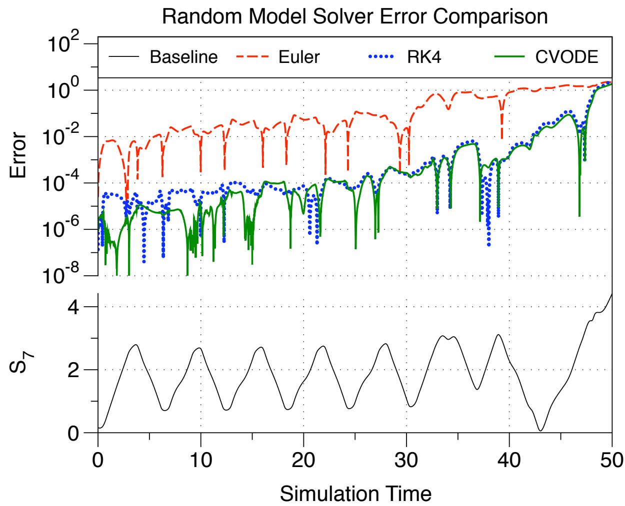
The simulation times for this model on an 8 core machine are shown in Figure 7. The figure shows the total simulation time on the bottom, broken into four components. Initialization consists of creating the model and communication plan, while partitioning involves distributing the objects over multiple processors. These are performed only once at the start of simulation. Communication and Runge-Kutta are performed during every simulation step. The top part of the figure shows the speedup of the entire simulation, the non-initialization parts (since initialization is not parallelized), and the Runge-Kutta calculation.

The results show that the program achieves reasonable speedup for parallel simulation. The Runge-Kutta calculation alone does well, with 8 processors providing 6x speedup. Including partitioning and communication in the measurements decreases performance, limiting speedup to roughly 3.5x for 8 processors. However, this will depend on the model and partitioning method used. The total speedup is worse due to the initialization time. This can be improved by optimizing model creation or allowing parallel model importation. For large models though, initialization will be a relatively small part of total simulation and therefore is a lower priority.

It is worth noting that no special optimizations were made for this parallel simulation, besides grouping communications into MPI_Request objects. In future work, we believe parallel simulations can be further improved by overlapping communication and computation, and using redundant cal-



(a) Luo-Rudy model.



(b) Random model.

Figure 4: Solver accuracy comparisons.

culation to decrease communication.

7. CONCLUSIONS

In this paper we introduced *insilico*Sim, an extendable parallel simulator of heterogenous biophysical models. We demonstrated three key aspects of the simulator. First, we showed how a standardized interface concept allowed for extension of simulation functionality. Next we demonstrated how simulation performance is improved by simplifying and compiling expressions. Finally we demonstrated how parallel synchronization and simulation is achieved transparently through a data object manager.

As multicore processors and parallel computing platforms become more prevalent, it will be necessary to create simulators that are easily run in parallel even while they can be extended to customized solution methods. We believe the techniques presented in this paper may be applicable to simulations to aid in this regard.

In the immediate future, we plan to extend the simulator

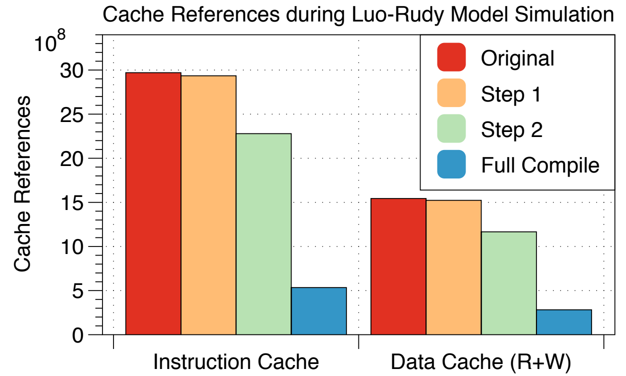


Figure 6: Comparison of cache references for different levels of compilation.

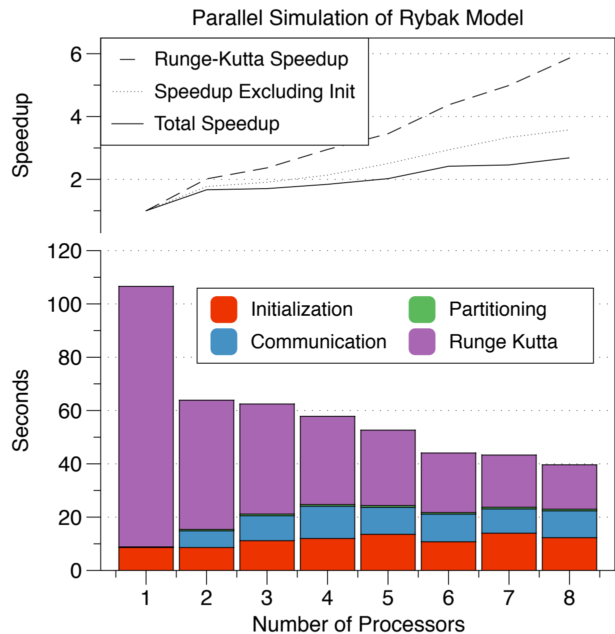


Figure 7: Parallel simulation of Rybak model.

to support additional import formats such as CellML and FieldML, while adding support for agent based simulation and finite element simulations specified in insilicoML. Because of the interface concept described in this paper, it is possible to make these extensions function with pre-existing interfaces and parallel simulations.

8. ACKNOWLEDGMENTS

The authors would like to thank Yasuyuki Suzuki and Keisuke Tominaga for assistance in creating the models, and to Braden Pellett for programming assistance. This work was supported in part by Research Fellowship (19·55401) and Grant-in-Aid for Scientific Research (A)20240002 from the Japan Society for the Promotion of Science, and by the Global COE Program "in silico medicine" at Osaka University.

9. REFERENCES

[1] Physiome website - http://physiome.jp/. 2009.

[2] Y. Asai, Y. Suzuki, Y. Kido, H. Oka, E. Heien, M. Nakanishi, T. Urai, K. Hagihara, Y. Kurachi, and T. Nomura. Specifications of insilicoml 1.0: A multilevel biophysical model description language. *The Journal of Physiological Sciences : JPS*, 58(7):447–58, Dec 2008.

[3] D. A. Beard, R. Britten, M. T. Cooling, A. Garny, M. D. B. Halstead, P. J. Hunter, J. Lawson, C. M. Lloyd, J. Marsh, A. Miller, D. P. Nickerson, P. M. F. Nielsen, T. Nomura, S. Subramanium, S. M. Wimalaratne, and T. Yu. Cellml metadata standards, associated tools and repositories. *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences*, 367(1895):1845–67, May 2009.

[4] G. Clapworthy, M. Viceconti, P. V. Coveney, and P. Kohl. The virtual physiological human: building a framework for computational biomedicine i. editorial. *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences*, 366(1878):2975–8, Sep 2008.

[5] A. Cuellar, C. Lloyd, P. Nielsen, D. Bullivant, D. Nickerson, and P. Hunter. An overview of cellml 1.1, a biological model description language. *Simul-T Soc Mod Sim*, 79(12):740–747, Jan 2003.

[6] K. Devine, E. Boman, R. Heaphy, B. Hendrickson, and C. Vaughan. Zoltan data management services for parallel dynamic applications. *Computing in Science & Engineering*, 4(2):90–96, 2002.

[7] E. Heien, Y. Asai, T. Nomura, and K. Hagihara. Optimization techniques for parallel biophysical simulations generated by insilicoide. *IPSJ Online Transactions*, 2 IS:149–161, 2009.

[8] B. Hess, C. Kutzner, D. van der Spoel, and E. Lindahl. Gromacs 4: Algorithms for highly efficient, load-balanced, and scalable molecular simulation. *Journal of Chemical Theory and Computation*, 4(3):435–447, Jan 2008.

[9] A. Hindmarsh, P. Brown, K. Grant, S. Lee, R. Serban, D. Shumaker, and C. Woodward. Sundials: Suite of nonlinear and differential/algebraic equation solvers. *Transactions on Mathematical Software (TOMS)*, 31(3), Sep 2005.

[10] M. Hucka, A. Finney, B. J. Bornstein, S. M. Keating, B. E. Shapiro, J. Matthews, B. L. Kovitz, M. J. Schilstra, A. Funahashi, J. C. Doyle, and H. Kitano. Evolving a lingua franca and associated software infrastructure for computational systems biology: the systems biology markup language (sbml) project. *Systems biology*, 1(1):41–53, Jun 2004.

[11] P. J. Hunter. The iups physiome project: a framework for computational physiology. *Progress in Biophysics and Molecular Biology*, 85(2-3):551–69, Jan 2004.

[12] M. Jeschke, R. Ewald, A. Park, R. Fujimoto, and A. Uhrmacher. A parallel and distributed discrete event approach for spatial cell-biological simulations. *ACM SIGMETRICS Performance Evaluation Review*, 35(4), Mar 2008.

[13] C. H. Luo and Y. Rudy. A model of the ventricular cardiac action potential. depolarization, repolarization, and their interaction. *Circulation Research*, 68(6):1501–26, Jun 1991.

[14] I. I. Moraru, J. C. Schaff, B. M. Slepchenko, M. L. Blinov, F. Morgan, A. Lakshminarayana, F. Gao, Y. Li, and L. M. Loew. Virtual cell modelling and simulation software environment. *IET systems biology*, 2(5):352–62, Sep 2008.

[15] D. Nickerson, M. Nash, P. Nielsen, N. Smith, and P. Hunter. Computational multiscale modeling in the iups physiome project: modeling cardiac electromechanics. *IBM Journal of Research and Development*, 50(6):617–630, 2006.

[16] D. Noble. Computational models of the heart and their use in assessing the actions of drugs. *J Pharmacol Sci*, 107(2):107–17, Jun 2008.

[17] T. Nomura. Challenges of physiome projects. *IEEJ Transactions on Electronics, Information and Systems*, 127(10):1491–1497, 2007.

[18] H. Plesser, J. Eppler, A. Morrison, M. Diesmann, and M.-O. Gewaltig. Efficient parallel simulation of large-scale neuronal networks on clusters of multiprocessor computers. *Euro-Par 2007 Parallel Processing*, pages 672–681, 2007.

[19] I. A. Rybak, N. A. Shevtsova, M. Lafreniere-Roula, and D. A. McCrea. Modelling spinal circuitry involved in locomotor pattern generation: insights from deletions during fictive locomotion. *J Physiol (Lond)*, 577(Pt 2):617–39, Dec 2006.