Techniques for Parallel Simulation of Large Scale Heterogeneous Biophysical Systems

Eric Martin Heien

Contents

1 Introduction **6**

1.1 Physiome Modeling . . . 7

1.2 Current State of Biological Simulation . . . 8

1.3 Goal, Motivation and Approach . . . 12

1.4 Contributions . . . 13

2 Biophysical Models **15**

3 Source Based Biophysical Simulation **21**

3.1 Introduction . . . 21

3.2 Source Based Biophysical Simulation . . . 21

3.3 Parallel Code Generation . . . 22

3.3.1 Module Dependencies . . . 24

3.3.2 Improving Simulation Speed with Dependence Information . . . 26

3.3.3 Work Partitioning . . . 27

3.4 Experiments . . . 30

3.4.1 Setup . . . 30

3.4.2 Simulation Communication . . . 31

3.4.3 Redundant Function Computation . . . 34

3.4.4 Simulation Execution . . . 36

3.5 Conclusion . . . 42

4 *insilico*Sim Biophysical Simulator **43**

4.1 Introduction . . . 43

4.2 *insilico*Sim . . . 44

4.2.1 Core Engine . . . 45

4.2.2 Interface Concept . . . 45

4.2.3 Example Interfaces . . . 47
4.3 Optimization Techniques . . . 48
4.3.1 Compiled Objects . . . 48
4.3.2 DataObjSet Object Manager . . . 50
4.4 Experiments . . . 53
4.4.1 Result Accuracy . . . 54
4.4.2 Compiled Objects . . . 55
4.4.3 Parallel Computing Speed . . . 58
4.5 Conclusion . . . 60

5 Volunteer Computing Scale Simulation **61**
5.1 Introduction . . . 61
5.2 Computation Model . . . 63
5.3 Analysis of Volunteer Computing Workers . . . 65
5.3.1 Worker Trace Data . . . 65
5.3.2 Analyzing Worker Availability . . . 66
5.3.3 Modeling VC Worker Communication . . . 68
5.3.4 Modeling VC Worker Computation . . . 72
5.4 Task Distribution Algorithms . . . 74
5.4.1 Fully Reliable Homogeneous Workers . . . 76
5.4.2 Semi-Communication-Reliable Homogeneous Workers . . . 77
5.4.3 Semi-reliable Heterogeneous Workers . . . 79
5.5 Experiments . . . 80
5.5.1 Semi-Communication-Reliable Workers . . . 80
5.5.2 Semi-Reliable Workers . . . 83
5.6 Conclusions . . . 85

6 Conclusions **86**
6.1 Summary of Contributions . . . 86
6.2 Future Work . . . 87

List of Figures

2.1 Hierarchy of modules in the Luo-Rudy model. 16

2.2 *insilico*IDE model of 8 connected Rybak brain stem respiratory neurons. 18

2.3 Part of the dependency graph of the Luo-Rudy model. 19

3.1 Sample model with four modules A, B, C and D, divided between two nodes, X and Y. 23

3.2 An example of redundant function calculation. 27

3.3 The model from Figure 3.1 after using the described techniques. 29

3.4 Comparison of communication between original and modified models. 32

3.5 Communication comparison between cardiac models. 33

3.6 Average redundant function evaluations per simulation time step. 35

3.7 Example partitioning of respiratory model over 16 processors. 37

3.8 Motor and respiratory simulations performed on multicore machine. 38

3.9 Cardiac simulations performed on multicore machine. 39

3.10 Motor and respiratory simulations performed on cluster. 40

3.11 Cardiac simulations performed on cluster. 41

4.1 The three stages of simplifying and compiling an expression tree into byte code. 49

4.2 Random model graph from the Random and DOT export interfaces. 52

4.3 Solver accuracy comparisons. 56

4.4 Comparison of execution time for different levels of compilation. 57

4.5 Comparison of cache references for different levels of compilation. 58

4.6 Parallel simulation of Rybak model. 59

5.1 Graphic depiction of tasks and batches in the low latency computing model, and three example workers with their availability intervals. 64

5.2 Worker availability and lifespan characteristics. 67

5.3 Simulation results. 69

5.4 QQ-PP plots. 70

5.5 Simulation results for worker availability and unavailability prediction. 73

5.6 Results of simulations for validating worker performance prediction. 75

5.7 Two results from semi-communication reliable worker experiments. 82

5.8 Fraction of tasks in a batch that were distributed before the deadline. 83

5.9 Task and batch satisfaction rates from the experiments. 84

List of Tables

1.1 A list of programs and libraries for performing biophysical simulations. 9

2.1 Calculations needed to approximate ODE for V in Luo-Rudy model. 20

3.1 Calculations to perform one simulation time step for the model in Figure 3.1. 24

3.2 Steps for simulation of the model in Figure 3.3. 28

3.3 Simulation model details. 30

4.1 List of available interfaces for *insilico*Sim. 45

4.2 Expressions for example random model. 53

4.3 Computation and communication steps for the functions of the example random model. 53

4.4 Model characteristics. 54

5.1 Task request simulation parameters. 68

5.2 Simulation parameters. 80

Acknowledgments

I would like to start by expressing gratitude to my advisor Professor Kenichi Hagihara for his guidance and support. I learned a great deal from him about how to do research, write a paper and how to think about my work.

I would also like to thank other professors in my lab for their assistance in my research, including Professor Noriyuki Fujimoto, Professor Masao Okita and Professor Fumihiko Ino. I would also like to thank the department staff, especially Mitsuyo Kondo, without whom I would be spending most of my time navigating paperwork.

I also offer my thanks to my committee members - Professor Toshimitsu Masuzawa, Professor Hideo Matsuda and Professor Taishin Nomura - for their helpful questions and comments throughout the course of my research.

Because this work goes outside the computer science discipline, I have had the chance to work with many talented individuals in the biosciences. I would especially like to offer thanks to Professor Yoshiyuki Asai for including me in his research and to Dr. Rachid Ait Haddou for many discussions of theoretical biology. Also, many thanks to the students of Professor Nomura's lab for helping create models on short notice and discussing their work, especially Yasuyuki Suzuki, Yosuke Yumikura, Keisuke Tominaga and Masao Nakanishi.

Outside of Osaka University I have received help from numerous people. I am deeply grateful to Dr. Derrick Kondo (INRIA), Braden Pellett (UC Davis), Dr. David Anderson (UC Berkeley), Dr. Dan Werthimer (UC Berkeley) and Jeremy Cowles (UC Berkeley) who all supported my work and gave me excellent advice.

Finally, I am very grateful to my wife Haruko who has supported me throughout my work.

My 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.

Abstract

Recently several multidisciplinary projects have begun to model and simulate human biophysical and physiological systems. These projects aim to create databases of models which can be combined to generate larger and more complex biophysical models. These large scale models can then be used to perform *in silico* (computer based) simulations to accurately predict the effects of drugs over a range of circumstances and patients. The ultimate goal of these projects is to realize a system for predictive medicine. This uses patient specific models to determine treatment for an ailment in a particular patient or predict side effects of a drug on patients with certain genetic traits.

Though this research area has advanced in recent years, there is still a lack of tools to effectively perform these types of simulations, particularly in regards to simulating large scale models on parallel computing platforms. These types of large scale simulations are required to effectively solve the problems posed by predictive medicine. Although many software packages exist to perform biophysical simulations, few of these support parallel simulation of large scale models, and the ones that do generally support only a particular type of model. This dissertation discusses techniques and tools that are applicable to performing parallel simulations of these kinds of large scale models.

This subject is addressed on multiple scales. First, we examine techniques for performing parallel simulations by converting models into C++ source code which is compiled and executed. This method is common because of its simplicity and ease of simulation generation. Also, there are tools such as MATLAB and Mathematica which may be used as a back end to perform the computation. However, these types of simulations are difficult to parallelize because of the complexity of the models and difficulty of writing metacode for parallel simulations. In this case, we use model analysis and redundant computation to simplify the parallel simulation yet maintain good performance and calculate the same results. However, this technique is limited because of the time required to compile individual simulations and the complexity as the model becomes more heterogeneous.

Next we describe *insilico*Sim, an extendable simulation engine for performing parallel large scale biophysical simulations. Rather than creating source code for each model, this engine imports models, converts them to internal data structures and performs simulations. This solves the difficulties of source based simulations in dealing with large heterogeneous models and allows new types of models and 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 calculations into a 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.

Finally, we demonstrate a method for utilizing large scale unreliable computing resources to perform parallel computations such as those used in modeling biomolecular dynamics. We propose algorithms for computing batches of medium grained tasks with deadlines in pull-style volunteer computing environments. First we develop models of unreliable workers based on analysis of trace data from an actual volunteer computing project. These models are used to develop algorithms for task distribution in volunteer computing systems with a high probability of meeting batch deadlines. We develop algorithms for perfectly reliable workers, computation-reliable workers and unreliable workers. Finally, we demonstrate the effectiveness of the algorithms through simulations using traces from actual volunteer computing environments.

Publication List

Journal Papers

1. Eric M. Heien, David P. Anderson, and Kenichi Hagihara. “Computing Low Latency Batches with Unreliable Workers in Volunteer Computing Environments.” Journal of Grid Computing, Vol. 7, No. 4, pp. 501-518, (2009-12). **(Chapter 5)**

2. Eric Heien, Yoshiyuki Asai, Taishin Nomura, and Kenichi Hagihara. “Optimization Techniques for Parallel Biophysical Simulations Generated by insilicoIDE.” IPSJ Transactions on Advanced Computing Systems, Vol. 2, No. 2, pp. 131-143, (2009-07). **(Chapter 3)**

3. Yoshiyuki Asai, Yasuyuki Suzuki, Yoshiyuki Kido, Hideki Oka, Eric Heien, Masao Nakanishi, Takahito Urai, Kenichi Hagihara, Yoshihisa Kurachi, Taishin Nomura. “Specifications of insilicoML 1.0: A Multilevel Biophysical Model Description Language.” The Journal of Physiological Sciences, vol. 58 (7) pp. 447-58 (2008-12).

International Conferences

4. Eric M. Heien, Masao Okita, Yoshiyuki Asai, Taishin Nomura, Kenichi Hagihara. “insilicoSim: an Extendable Engine for Parallel Heterogeneous Biophysical Simulations.” Submitted to SIMUTools 2010 **(Chapter 4)**

5. Eric M. Heien, Adam Kornafeld, Yusuke Takata, and Kenichi Hagihara. “PyMW - a Python Module for Desktop Grid and Volunteer Computing.” In Proceedings of the 3rd Workshop on Desktop Grids and Volunteer Computing Systems (PCGrid 2009), Rome, Italy, (2009-05). 7 pages (CD-ROM).

6. Eric M. Heien, Adam Kornafeld, Yusuke Takata, and Kenichi Hagihara. “PyMW - a Python Module for Parallel Master Worker Computing.” In Proceedings of the 1st International Conference on Parallel, Distributed and Grid Computing for Engineering (PARENG 2009), 15 pages, Pécs, Hungary, (2009-04).

7. Eric Heien, Yoshiyuki Asai, Taishin Nomura, Kenichi Hagihara. "Techniques for Automatic Parallelization and Optimization of Biological Simulations from insilicoIDE." 3rd MEI International Symposium (2008-12). **(Chapter 3)**

8. Yasuyuki Suzuki, Yoshiyuki Asai, Toshihiro Kawazu, Masao Nakanishi, Yoshiyuki Taniguchi, Eric Heien, Kenichi Hagihara, Yoshihisa Kurachi, and Taishin Nomura. "A Platform for in silico Modeling of Physiological Systems II. CellML Compatibility and Other Extended Capabilities." In Proceedings of the 30th International Conference of the IEEE Engineering in Medicine and Biology Society (EMBC 2008), pp. 573-576, Vancouver, British Columbia, Canada, (2008-08).

9. Eric Martin Heien, Noriyuki Fujimoto, and Kenichi Hagihara. "Computing Low Latency Batches with Unreliable Workers in Volunteer Computing Environments." In Proceedings of the 2nd Workshop on Desktop Grids and Volunteer Computing Systems (PCGrid 2008), Miami, FL, USA, (2008-04). 8 pages (CD-ROM). **(Chapter 5)**

10. Eric Martin Heien, Noriyuki Fujimoto, and Kenichi Hagihara. "Static Load Distribution for Communication Intensive Parallel Computing in Multiclusters." In Proceedings of the 16th Euromicro International Conference on Parallel, Distributed and Network-based Processing (PDP 2008), pp. 321-328, Toulouse, France, (2008-02).

11. Eric Heien, Masao Nakanishi, Yoshiyuki Asai, Taishin Nomura, Kenichi Hagihara. "Automatic Parallelization of Biological Simulations from the in Silico IDE for Execution in Cluster Environments." 2nd MEI International Symposium (2007-12).

12. Eric Martin Heien, Tomoyuki Hiroyasu, and Noriyuki Fujimoto. "Investigation of Mutation Operators for the Bayesian Optimization Algorithm". In Proceedings of the 2007 Genetic and Evolutionary Computation Conference (GECCO 2007), p. 625, London, UK, (2007-07).

Domestic Conferences

13. Eric Heien, Yoshiyuki Asai, Taishin Nomura, Kenichi Hagihara. "Optimization Techniques for Parallel Biophysical Simulations Generated by insilicoIDE." 9th High Performance Computing Symposium (HPCS 2009), pp. 1-8, Tokyo, Japan, (2009-01).

Oral Presentations

14. Jeremy Cowles, Eric Heien, Adam Kornafeld, Yusuke Takata, Kenichi Hagihara, Nicolás Alvarez. “PyMW: Master Worker computing with Python.” 5th BOINC Workshop, Barcelona, Spain (2009-10).

15. Yusuke Takata, Eric Heien, Masao Okita, Kenichi Hagihara. “MapReduce Implementation in Python for multiple Parallel Computing Environments.” Information Processing Society of Japan Report, 2009-ARC-185, 7 pages, (2009-10).

16. Yasuyuki Suzuki, Yoshiyuki Asai, Eric Heien, Masao Nakanishi, Takahito Urai, Hideki Oka, Kenichi Hagihara, Yoshihisa Kurachi, Taishin Nomura. “insilicoIDE: An Integrated Development Environment for Physiological Modeling.” Biology Medicine and Engineering Symposium 2008, Osaka, Japan, pp. 378-381, (2008-09).

17. Yoshiyuki Asai, Yasuyuki Suzuki, Yoshiyuki Kido, Hideki Oka, Eric Heien, Masao Nakanishi, Takahito Urai, Kenichi Hagihara, Yoshihisa Kurachi, Taishin Nomura. “Development of insilicoML 1.0: A Language for Multilevel Physiological Models.” Biology Medicine and Engineering Symposium 2008, Osaka, Japan, pp. 383-384, (2008-09).

18. Yoshiyuki Asai, Toshihiro Kawazu, Masao Nakanishi, Yasuyuki Suzuki, Eric Heien, Kenichi Hagihara, Yoshihisa Kurachi, Taishin Nomura. “The insilicoIDE Integrated Platform and Architecture for Multilayer Object Oriented Biodynamics Modeling.” The 47th Annual Conference of Japanese Society for Medical and Biological Engineering, 2008, Kobe, Japan, p. 652, (2008-05).

19. Masao Nakanishi, Yoshiyuki Asai, Eric Heien, Kenichi Hagihara, Taishin Nomura. “The Massive Simulation of the Spinal Neural Network Dynamics using Biodynamics Modeling Integration Platform (insilico IDE).” The 47th Annual Conference of Japanese Society for Medical and Biological Engineering, 2008, Kobe, Japan, p. 506, (2008-05).

Chapter 1

Introduction

Computer based simulation traces its roots back to Monte Carlo simulations used to model neutron diffusion in nuclear reactions [1]. Since then, computer simulation has been used in a variety of applications ranging from the optimization of air flow in car design to population dynamics of ecosystems. The design of the Boeing 777 aircraft was completely tested by computer simulation before being flown for the first time [2]. Simulation is also used in numerous other areas, including layout of wireless communication networks and other distributed systems [3]. Traditionally, simulation tends to focus on fields such as physics, chemistry and engineering. However, in recent years scientists have increasingly realized the benefits of computer simulation in the fields of biology, physiology and pharmacology.

One of the early computer based biological simulations was performed by Denis Noble, who used an analog computer to simulate a model of cardiac cell action potential. From this simulation he demonstrated that the oscillatory nature of the heart is not caused by a single oscillator but is rather a systems-level emergent property [4]. More recently, improvements in computational technology have added further successes in the use of computers for biological simulations. The Blue Brain project uses the Blue Gene supercomputer architecture to perform biologically realistic simulations of neuronal activity in order to better understand brain function [5]. The Folding@home project [6] uses volunteered computing resources to perform molecular scale simulations of protein dynamics. This improves understanding of how proteins fold incorrectly and cause ailments such as Alzheimers and Parkinsons disease [7]. Pharmaceutical companies routinely use molecular level computer simulations to screen potential drug candidates before performing costly trials, albeit with varying degrees of success [8].

Thanks to the exponential rise in computing power as described by Moore's Law, biological models and simulations are becoming more realistic and useful. An electromechanical whole heart simulation of cardiac function has already been implemented [9], and researchers predict full scale modeling and simulation of the human brain will be possible within the next 10 years [10, 11]. Naturally these models push the boundaries of computer performance and require specialized software running on supercomputers.

Encouraged by the success of these projects and others, international efforts to model the human physiome are beginning to take shape. The physiome of an individual describes its functional behavior on multiple scales of time and size. This can range from the microsecond timescale behavior of proteins in cells to the multiyear development of large tissues and organs. These models may be highly heterogeneous, combining partial/ordinary/stochastic differential equations, algebraic functions, static model parameters, agent based representations, morphological representations, etc., to properly capture aspects of the physiome. The techniques used to simulate the models may also be heterogeneous, for example, using the Euler method, Runge-Kutta method or adaptive time stepping to approximate the solution of an ordinary differential equation. Finally, the representation of the models themselves may be heterogeneous, with several markup languages and standards for these purposes already in use or under development.

1.1 Physiome Modeling

The current major efforts in physiome modeling include work by the International Union of Physiological Sciences (IUPS) Physiome Project [12], National Simulation Resource (NSR) Physiome Project [13] and the *in silico* Medicine initiative [14]. The goal of these projects is to create *in silico* (i.e. computer based) multi-level and multi-timescale integrated models of cells, organs and systems [15].

Part of these efforts includes work to define standards for creating and sharing biological models [16]. These standards are used to facilitate model sharing between researchers and allow easy model reuse and extension. We discuss some standards here that relate directly to this thesis, but this is by no means a comprehensive list.

The Systems Biology Markup Language (SBML) [17] standard is a popular format for representing models of biological processes. SBML uses MathML to represent parameters, function expressions, ordinary differential equations and discrete events which model a biological process. SBML is commonly used to represent biochemical rate reactions involving multiple species and parameters. It is a widely adopted standard, with over 150 programs and libraries supporting it.

CellML [18] is a standard proposed by the IUPS project. CellML is similar in scope to SBML, using MathML to encode ordinary differential equations representing biochemical reactions. The main difference between CellML and SBML is that CellML has a hierarchical structure of model components. This makes model building easier, and allows for large complex models to be built from simple components. The primary goal of CellML is modeling biological systems, but it can be applied to other fields as well (such as materials science [19]).

More recently, the *in silico* Markup Language (ISML) [20] standard was proposed and is currently under development in the *in silico* Medicine initiative. This standard aims to support a wider range of model representations and structures. In addition to the sort of rate reactions supported by SBML and the hierarchical structure of CellML, ISML also encodes morphological data, rules for creating systems of interacting agents and support for external sets of time series data. It is being developed in conjunction with a database system to promote model reuse and sharing of morphological/time series data.

One goal of modeling biophysical systems is to perform numerical simulations based on the models. These simulations help researchers understand complex biological phenomenon and make predictions by altering model parameters. There is a wide variety of simulation libraries and software packages available to researchers to assist in modeling and simulating biophysical systems.

1.2 Current State of Biological Simulation

There are numerous software packages currently available for performing biological modeling and simulation. A subset of these packages is shown in Table 1.1. These programs and libraries vary in terms of the scale of their simulation target, as well as the computing platforms and capabilities they support. Here we will examine the different options and discuss what sort of simulations they are suited to.

In Table 1.1, the "Parallel" column refers to whether the program/library can decompose the simulation of a single model for parallel execution on a cluster or multicore machine. This is different from a parameter sweep simulation, where many simulations of the same model are performed with different parameters. Parameter sweeps or parameter optimization techniques are often used in designing a model to fit experimental data. However, as used in biophysical modeling, this functionality can be easily achieved by using parallel computing libraries and therefore is not considered in the scope of this dissertation.

The different simulation scales are indicated in the "Scale" column. Molecular level simulations are concerned with the movement and interaction of biological molecules, often proteins or viruses. Subcellular simulations are concerned with the processes and components within a cell, while cellular simulations are concerned with the whole cell and its interactions with nearby cells. Tissue simulations are used for large groups of cells and their function as a whole. Finally, organ simulations are concerned with the large scale function of entire organs and organ systems. Multiscale simulations involve the interaction of different levels of time or space in a simulation. It is difficult to exactly delineate the scope of each simulator (e.g. with enough molecules, a molecular simulator becomes a cellular simulator), so these are intended only as rough guides.

At the molecular scale, GROMACS (GROningen MAchine for Chemical Simulations) [21] is used for fast simulation of molecular systems consisting of hundreds to millions of particles. It is designed primarily for simulations of biochemical molecules and can run efficiently on parallel platforms using MPI and processor specific optimizations. It is also used in the Folding@home [6] and EvoGrid [60] projects to simulate biomolecular dynamics. NAMD (NAnoscale Molecular Dynamics) [22] is a similar package for parallel molecular simulation,

Table 1.1: A list of programs and libraries for performing biophysical simulations.

Simulator Name	Target Simulation	Scale	Parallel
GROMACS [21]	Biomolecular dynamics	M	Yes
NAMD [22]	Biomolecular dynamics	M	Yes
BioNetS [23]	Stochastic biomolecular	M/S	No
StochSim [24]	Stochastic biomolecular	M/S	No
AgentCell [25]	Stochastic biochemical	S/C	No
CellDesigner [26]	Biochemical networks	S/C	No
Cellerator [27]	Biochemical networks	C/T	No
Cell Illustrator [28]	Biochemical networks	S/C/T	No
COPASI [29]	Biochemical networks	S/C	No
DBSolve [30]	Biochemical networks	S/C	No
GEPASI [31]	Biochemical networks	S/C	No
Jarnac [32]	Biochemical networks	S/C	No
JigCell [33]	Biochemical networks	S/C	No
ScrumPy [34]	Biochemical networks	S/C	No
BioSPICE [35]	Cellular processes	M/S/C	No
Cellular Open Resource [36]	Cellular processes	S/C	No
CompuCell3D [37]	Cellular processes	C/T	No
E-Cell [38]	Cellular processes	M/S/C	No
iSimBioSys [39]	Cellular processes	S/C	No
PySCeS [40]	Cellular processes	S/C	No
SBW [41]	Cellular processes	S/C/T/O	No
simBio [42]	Cellular processes	M/S/C	No
Virtual Cell [43]	Cellular processes	S/C	No
MCell [44]	Cellular processes	S/C	Yes
BeatBox [45]	Cardiac simulation	C/T/O	Yes
CESE [46]	Cardiac simulation	C	No
iCell [47]	Cardiac simulation	C	No
LabHEART [48]	Cardiac simulation	S/C	No
C2 [49]	Neuronal simulation	C/T	Yes
GENESIS [50]	Neuronal simulation	C/T	Yes
NEST [51]	Neuronal simulation	C/T	Yes
NEURON [52]	Neuronal simulation	C/T	Yes
PCSIM [53]	Neuronal simulation	C/T	Yes
SPLIT [54]	Neuronal simulation	C/T	Yes
FLAME [55]	Multiscale agent simulation	S/C/T/O	Yes
CMISS [56]	Multiscale physiome	S/C/T/O	Yes
*insilico*IDE/*insilico*Sim[57]	Multiscale physiome	S/C/T/O	Yes
JSim [58]	Multiscale physiome	S/C/T/O	No
PCEnv/OpenCell [59]	Multiscale physiome	S/C/T/O	No
Scale Key: M = molecular, S = subcellular, C = cellular, T = tissue, O = organ			

with more of a specific focus on biological simulations than GROMACS and better scalability to more processors.

Because of the complexity of molecular interactions involving large numbers of particles, stochastic simulators are often used to perform probabilistic simulations of biomolecular processes. BioNetS (Biochemical Network Stochastic Simulator) [23] represents the uncertainty in biochemical reactions with stochastic differential equations. StochSim [24] calculates molecular interaction probabilities, then works by randomly selecting pairs of molecules and simulating their reaction. AgentCell [25] performs agent based stochastic simulations of a bacterium in a 3D environment to understand the process of chemotaxis.

Many simulators aim to understand the biochemical networks that affect the function of cells. The CellDesigner [26] program implements a graphical interface that lets users create and edit models of biochemical pathways. This program is compatible with SBML and uses the SBML ODE solver library to perform simulations. Cellerator [27] takes a different approach by writing biochemical reaction equations in the Mathematica framework. This simplifies simulation by using a common backend and allows model conversion to SBML, MathML or a variety of other formats.

Cell Illustrator [28] (previously known as Genomic Object Net) is part of a larger software package including a visualization program called Cell Animator and a model conversion program called BioPACS (BioPathway Automatic Convert System). It has been successfully applied to several simulations involving biochemical feedback loops and regulation. DBSolve [30] works in a similar vein by providing a GUI interface to manipulate and simulate biochemical networks.

COPASI (COmplex PAthway SImulator) [29] and GEPASI (GEneral PAthway SImulator) [31] are also biochemical pathway simulators. COPASI is based on GEPASI and is backwards compatible. COPASI features a graphical user interface and can import and export SBML. Both of these programs use ODEs to represent biochemical pathways. COPASI can perform parameter sweeps of models, but cannot parallelize individual models.

Jarnac [32] uses a custom scripting language to define and simulate models of biochemical pathways. This gives it an advantage in being able to support iterative and conditional statements, and allows for matrix manipulations. Cellular Open Resource [36] (COR) is a Windows environment for text based editing of CellML models and simulation with a variety of solvers. COR is functionally similar to OpenCell [59], but lacks features for composing complex models and does not use the CellML API.

BioSPICE (Biological Simulation Program for Intra- and Inter-Cellular Evaluation) [35] is an open source framework from cellular biology that is extended by numerous modules. For example, the JigCell module [33] allows users to import SBML models, edit them with a graphical user interface and perform simulations and parameter estimation. There are numerous modules available for BioSPICE, ranging from agent based simulation tools to libraries for time series data manipulation. SBW (Systems Biology Workbench) [41] is similar to BioSPICE in being extendable. It offers model translation and visualization tools, and simulators for deterministic and stochastic models.

The E-Cell [38] system aims to perform whole cell simulations. It supports deterministic and stochastic elements of models ranging from gene level to the entire cell. iSimBioSys [39] uses discrete event based simulation to model gene expression and regulation. PySCeS [40] and ScrumPy [34] both provide a high level interface for researchers to create Python based models.

simBio [42] is a Java based cellular modeling tool with a custom XML data format. It uses ordinary differential equations for the models and aims to allow easy composition of modules representing biological functions. The Virtual Cell Modeling and Simulation Framework [43] uses a Java web-based interface which allows users to create complex multi-layered models.

MCell [44] and CompuCell3D [37] are for modeling and realistic simulation of 3D biological environments. MCell targets cellular signaling and uses a Monte Carlo style algorithm to track the behavior of individual molecules. CompuCell3D uses partial differential equations and cellular automata to model morphogenesis in tissues, though it does not support parallel computing for large scale models.

CESE (Cell Electrophysiology Simulation Environment) [46], iCell [47] and LabHEART [48] are specifically designed to perform simulations of electrocardiac cell activity. They are oriented more towards student education than research, though iCell can also export Java code for simulations. BeatBox [45] is a planned framework to enable high performance parallel simulations of cardiac tissue, but is still under development. Other work [56, 61] has also touched on this area but there are still no definitive open source solutions for large scale cardiac simulation.

There are also several libraries and programs aimed specifically at the simulation of neurons and networks of neurons. These generally support parallel computing because of the large size of the neuronal models. The C2 cortical simulator [49] is designed for efficient, large scale parallel simulations of neuronal models on the Blue Gene supercomputer. GENESIS (GEneral NEural SImulation System) [50] targets highly realistic and detailed neuronal models, but generally of smaller scale. PGENESIS allows for parallel simulation on networked computers. NEST (NEural Simulation Tool) is another parallel simulation tool that focuses on heterogeneous networks of neurons. It is more suited to simulating overall systems dynamics rather than individual neuronal biophysics. NEURON [52] is an educational and research tool used for simulation of neurons and neuronal networks, often based on empirical data sets. PCSIM [53] provides a Python interface over a C++ library to perform neuronal circuit simulations. SPLIT [54] is a C++ library for constructing and simulating large scale neuronal networks, which scales very well on large computer clusters.

FLAME (Flexible Large-scale Agent-based Modelling Environment) [55] is an agent based modeling and simulation framework for multiscale biophysical simulations. It does not provide simulation tools itself, but allows outside tools to be combined to perform agent based simulations. CMISS/OpenCMISS [56] is a tool for multidimensional continuum mechanics simulations, such as in a full heart simulation. OpenCMISS is currently under development, and uses CellML to describe parts of the model.

Although there are many tools for biological modeling, few offer parallel computing support for handling large scale models outside of specific areas in cardiac and neuronal simulation. There are several tools more oriented towards general physiome simulation. JSim [58] is a Java based simulation system that integrates ODEs, 1D PDEs and discrete events in the simulations. JSim uses a scripting language to specify models, though it can import SBML and CellML models. It offers parallel computing support for parameter sweep style model runs, but not for decomposing large models.

PCEnv (now known as OpenCell) [59] is another platform for physiome oriented modeling and simulation. It is similar to Cellular Open Resource but is open source, supports more platforms and uses the CellML API for model manipulation. It is supported by a database of models in CellML format.

The *insilico*IDE program [57, 62] is a modeling and simulation program oriented to the ISML markup language, though it can import CellML and SBML models. With older versions of *insilico*IDE, simulations were performed by exporting the biophysical model and control code as a C++ source file. The recently released *insilico*IDE 1.0 uses a separate simulator program, *insilico*Sim, that supports large scale parallel physiological simulations. Both the current and previous versions support efficient parallel simulation of decomposed models.

1.3 Goal, Motivation and Approach

As shown in the previous section, few biophysical programs or libraries support the type of parallel computing needed for simulation of large scale models. The few packages that support parallel simulation are too confined to a specific area to be of general use in physiome simulations. For example, the neuronal simulation packages [49, 50, 51, 52, 53, 54] use techniques specific to certain types of neuronal models to optimize their simulations. In the C2 cortical simulator, if a neuron fires and activates neurons on other processors, the notification to these processors may be bundled with other firings to improve simulation speed. These sorts of techniques may not be applicable to more general physiological simulations.

With the development of ISML and similar markup languages, simulators for heterogeneous large scale biophysical models are required. Because of memory and speed constraints, as well as the growing importance of parallel computing in general, techniques for parallel simulation of these models are needed. Although enabling parallel simulation and improving performance are key goals, it is equally important to develop methods for easy extension of simulations to new model formats and simulation methods that can interact with preexisting models.

In addition, it is important to consider the scale of the simulation. Many simulation packages use source based simulations, where the simulation of a model is created by exporting and compiling a separate program based on the model. These simulations will have good performance because of the machine language code, and although it is difficult to parallelize this method can provide benefits for medium-sized models. However, as the model size grows the compilation time and memory requirements become overwhelming and a different technique is needed.

For larger models, it is generally better to use a separate program or library where the simulation is performed using internal representations of equations and states. This is because of the difficulty in writing a program to correctly generate another program, especially when the generated program is complex. Since the types of simulations in bioscience may combine different elements and scales, a source based approach may not be feasible. Furthermore, compiling the generated simulation can require excessive time for large models and may not be possible due to memory limitations. A separate simulation program can be run on a cluster or supercomputer and should be designed to achieve good parallel performance while running at speeds not significantly slower than compiled source.

Finally, grid or volunteer computing scale resources may be required for very large simulations. The advantage of using grid or volunteer computing resource is the massive amounts of available CPU time. However, because of the unreliable nature of the computing resources, heuristics are needed to achieve good performance of the simulation. Heuristics are needed to handle both communication and computation unreliability among workers, as well as scheduling of tasks on workers to ensure computational deadlines are met.

1.4 Contributions

The main thrust of this dissertation can be summarized as follows:

"Parallelization techniques for source based simulations, program or library based simulations and large scale volunteer computing simulations can be effectively implemented so as to perform heterogeneous parallel biophysical simulations that scale well and are efficient."

To achieve this goal, the contributions of this dissertation are as follows:

1. **A technique for modifying source based biological simulations to be efficiently and correctly executed on parallel platforms.**

   A key difficulty in implementing source based parallel biophysical simulations is the synchronization required between computation steps. When the model consists exclusively of ODEs this is not much of a problem, but for more heterogeneous models with separate functions and parameters it becomes more difficult. To solve this, we describe methods for performing parallel simulation through model analysis and redundant computation. This helps avoid unnecessary communication and efficiently simulates the model on a parallel platform. These techniques are also applicable to simulations performed by a separate program.

2. **A program for parallel large scale biophysical simulations using heterogeneous input, calculation and output.**

   Source based simulations are acceptable for medium sized models, but as model size increases the compilation and memory requirements become intractable. In addition, not all computers have the required compilation tools and it can be legally and technically difficult to include these with models. A separate simulation program is necessary to properly perform simulations of large scale heterogeneous models. To this end, we have developed and implemented an extendable simulator for large scale parallel simulations of heterogeneous biophysical models, such as those described by ISML.

3. **A heuristic for performing barrier synchronous biophysical simulations on large scale volunteer computing platforms.**

   Very large biophysical simulations with loosely coupled computations may be performed on large scale volunteer computing platforms, such as in Folding@home. These types of simulations have periodic barrier synchronizations where the current best results are used to generate the next set of simulation inputs. We present a heuristic for scheduling and selecting computational resources in a volunteer computing environment aimed at improving performance of loosely synchronous barrier computations such as these.

This dissertation is structured as follows. In Chapter 2 we discuss the structure and composition of the target biophysical models used in *insilico*IDE. We also provide a general overview of the *insilico*IDE modeling program and how it is used to create and manipulate models. The details of a source based parallelization technique are described in Chapter 3. The *insilico*Sim program is described in Chapter 4 along with the techniques it uses to improve simulation speed and allow for easy parallel simulation. For larger scale simulations in volunteer computing environments with barrier synchronization, we demonstrate a heuristic for improved performance in Chapter 5. We close with a review of the results and discussion of future work in Chapter 6.

Chapter 2

Biophysical Models

In this section, we describe the modeling of a biophysical system using an example in the ISML markup language. Most aspects of the example are common to models in the scope of physiome simulations. Elements that are peculiar to ISML are noted where appropriate. For this example we use a Luo-Rudy model of ventricular cardiac action potential [63] with an external stimulus. This is a common biophysical model with many important elements.

ISML uses "modules" to organize functional units of a model in a hierarchical fashion. This technique is also used in CellML and is being introduced to SBML. Each module represents a functional biological element such as a membrane, cell, organ, etc. Modules may contain other modules in a capsular fashion. An example would be a tissue module which contains multiple cell modules, each of which contain modules for ion pumps, cell membranes, etc. Modules are connected by "edges" representing structural or logical relationships. Edges allow modular, hierarchical, and/or network representations in a model. For example, a module representing intracellular ion concentration may be connected to a cell membrane module, which connects to an extracellular ion concentration module. This would represent the flow of ions between the inside and outside of the cell through the membrane.

Each module contains any number of user specified functions, static parameters and/or dynamic states. They also may include morphological data describing, for example, the bone structure or heart topology related to a model. Parallelization involving the morphological data is currently not supported and we ignore it in this discussion. More details on the structure and organization of ISML models are available in [64].

Modules enable model reuse by specifying clear divisions of abstraction among the different elements of a model. Figure 2.1 shows the module capsulation hierarchy of the Luo-Rudy model. The peaked boxes indicate modules within the model. The dashed lines indicate module encapsulation where an arrow from module A to module B indicates module B is encapsulated within module A. For example, E_Si is encapsulated by I_Si, which is in turn encapsulated by LR91_I_Si.

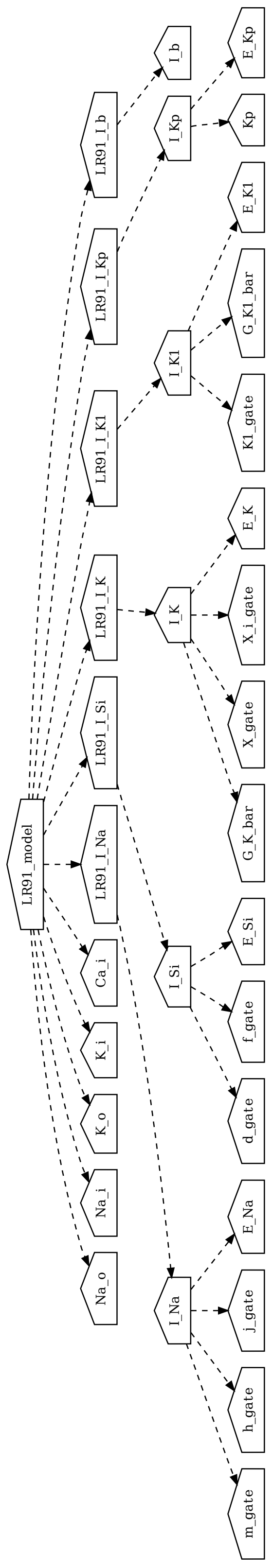


Figure 2.1: Hierarchy of modules in the Luo-Rudy model.

This model contains a main module (LR91_model) at the top of the hierarchy that encapsulates all other modules. Some of the sub-modules are separate models that have been incorporated into this model (LR91_I_Na, LR91_I_Si, etc). This mechanism allows for easy insertion and replacement of model elements to suit the desired model characteristics. For example, if a researcher wants to use a different representation of the fast sodium current they can replace LR91_I_Na with another module without changing anything else in the model.

The modules are also important in terms of simulation. Module encapsulation is important in determining data dependencies between modules in the simulation. In source based simulations from *insilico*IDE each module is exported as a separate class. The classes are combined with control code and compiled to create a model specific simulation. This simplifies code reuse and cleanly separates different calculations. However, this causes difficulties when performing parallel simulation as we describe in Chapter 3.

For the purposes of this dissertation, the key module elements are states, functions and parameters:

1. **Ordinary differential equations (States)**

   These represent values that change over time governed by first order ordinary differential equations (ODEs) of the form $\frac{dy_i}{dt} = f(y_0, \ldots, y_n, t)$. In biophysical models these represent changes in system elements with respect to time or other system elements. An example of a dynamic state is the ion concentration in a cell, changing based on the inflow and outflow of ions.

2. **Function expressions (Functions)**

   Represent standard function expressions of the form $f(y_0, \ldots, y_n, t) = \ldots$. These expressions cannot be self-referential. An example of module functions are the ionic channel current functions in the Hodgkin-Huxley model [65].

3. **Static values (Parameters)**

   Represent static simulation parameters, such as cell membrane permeability or other physical constants. Static parameters are values that do not change during the course of the simulation, for example, the viscosity of water or the Faraday constant. These may also be used to fit a model to experimental data by altering parameters over successive simulations.

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

Biophysical models are created by constructing and linking modules together. A state or function in one module may use the value of a state or function in any module it is linked to. Figure 2.2 shows a sample model with 8 modules representing Rybak brain stem respiratory neurons [66]. In this model, each neuron module (represented by the yellow squares) contains numerous submodules representing ion channels, membranes, etc.

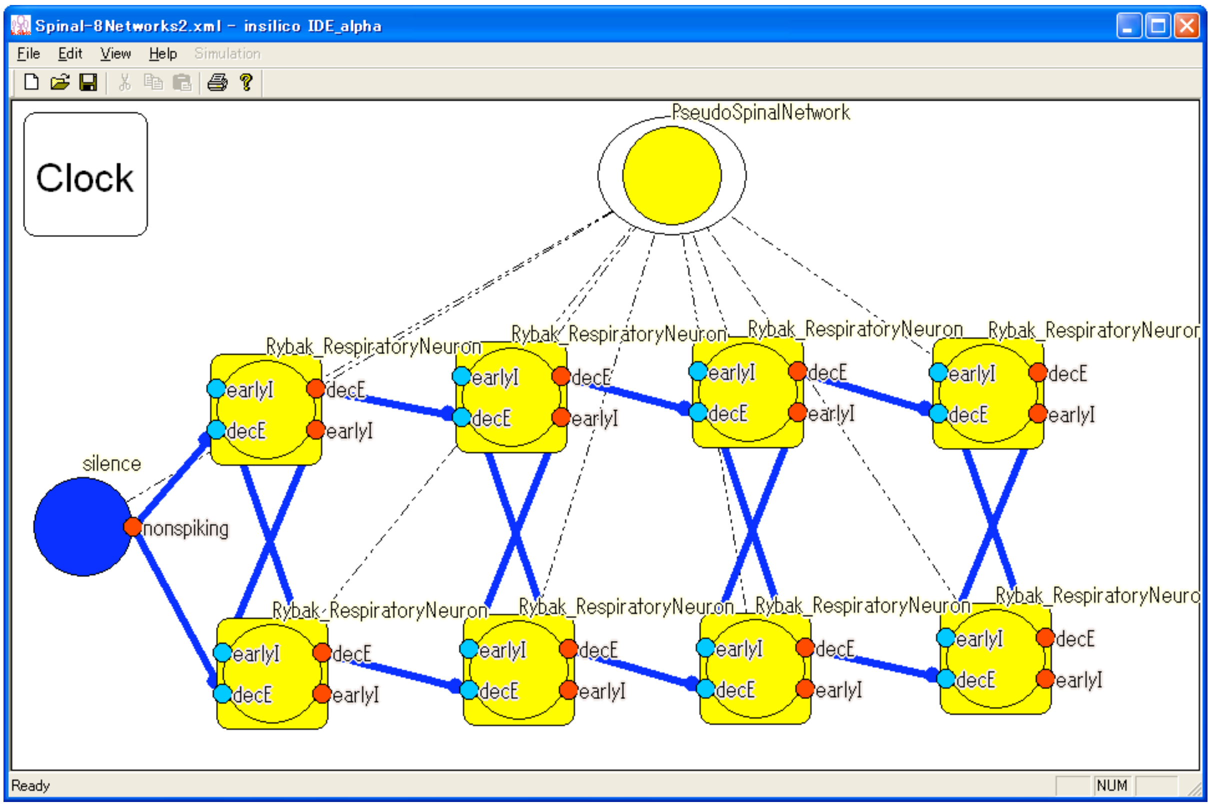


Figure 2.2: *insilico*IDE model of 8 connected Rybak brain stem respiratory neurons.

The connections (represented by the bold blue lines) represent synapses between axons and dendrites. The entire model is contained in the PseudoSpinalNetwork module, with the dotted lines representing the encapsulation of each module.

Figure 2.3 shows the dependency graph of the example model. This is only part of the full dependency graph, some elements were removed for clarity and are shown in the dashed circle. In this figure, boxes represent states (ODEs), ovals represent function expressions and trapezoids represent static parameters. The arrows represent data dependencies, for example, alpha_K1 requires the values of E_K1 and V, and it is used in turn to calculate K1_infinity. Each of these objects has associated mathematical expressions, which are output as a C language expression in source based simulations and stored as a tree structure in the *insilico*Sim simulator program. This model has a total of 8 ODEs (1 shown in Figure 2.3), 31 function expressions (6 shown in Figure 2.3) and 27 static parameters (5 shown in Figure 2.3).

It is worth noting the differences between Figures 2.1, 2.2 and 2.3 to avoid confusion. Figure 2.1 shows only the hierarchy of modules in the Luo-Rudy model. This is important because it demonstrates how model reuse is possible and gives an example of how a model is decomposed. Figure 2.2 shows the IDE with mixed aspects of a model. This shows the first level of the model hierarchy (represented by the dashed lines) as well as the connections between modules (represented by the bold blue lines). Finally, Figure 2.3 shows the Luo-Rudy model

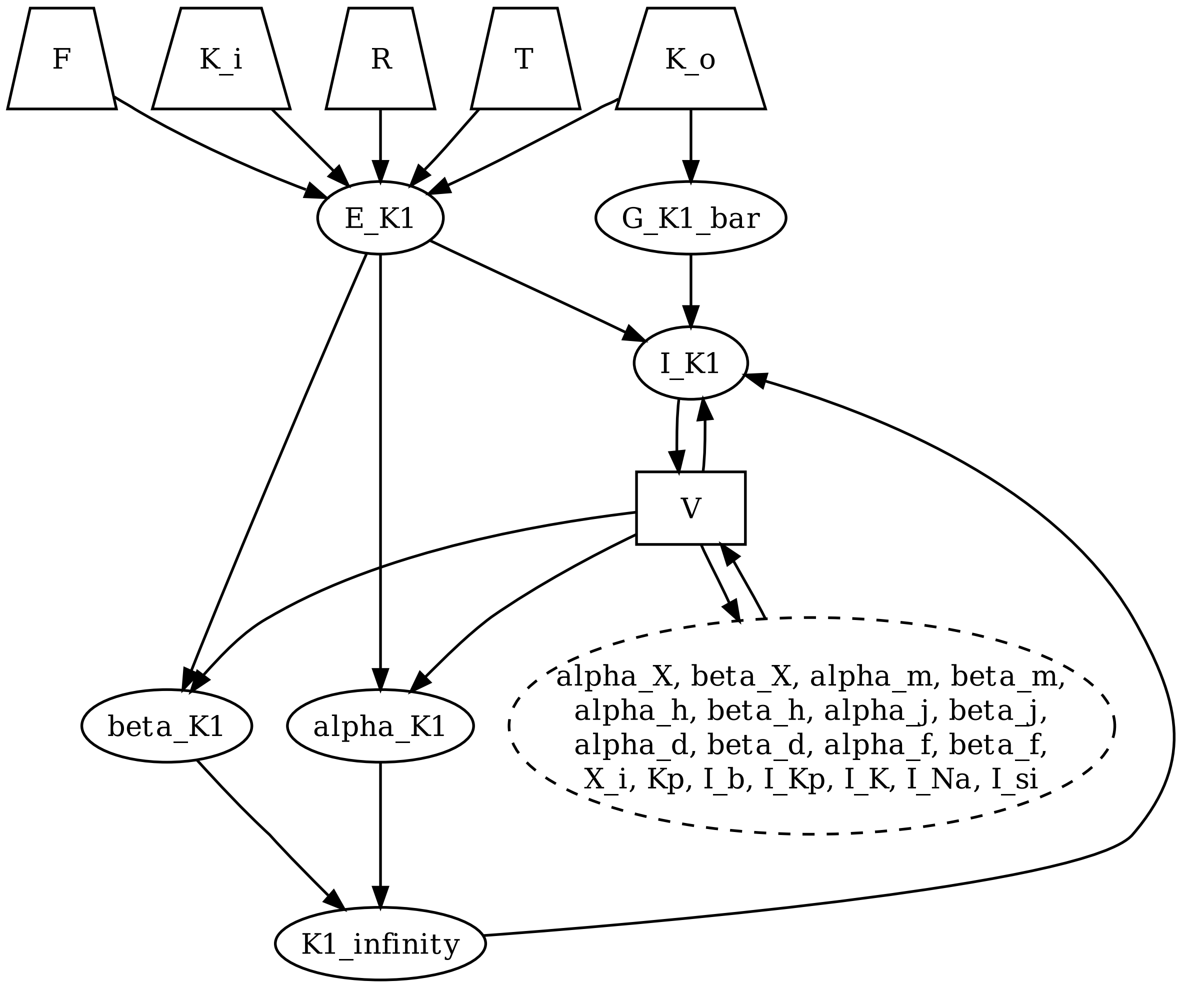

F
K_i
R
T
K_o
E_K1
G_K1_bar
I_K1
V
beta_K1
alpha_K1
alpha_X, beta_X, alpha_m, beta_m, alpha_h, beta_h, alpha_j, beta_j, alpha_d, beta_d, alpha_f, beta_f, X_i, Kp, I_b, I_Kp, I_K, I_Na, I_si
K1_infinity

Figure 2.3: Part of the dependency graph of the Luo-Rudy model.

Table 2.1: Calculations needed to approximate ODE for V in Luo-Rudy model.

Object Type	Calculation	Value References
Static Parameter	K_i	None
Static Parameter	K_o	None
Static Parameter	T	None
Static Parameter	F	None
Static Parameter	R	None
Function Expression	G_K1_bar	K_o
Function Expression	E_K1	K_i, K_o, T, F, R
Function Expression	alpha_K1	V, E_K1
Function Expression	beta_K1	V, E_K1
Function Expression	K1_infinity	alpha_K1, beta_K1
Function Expression	I_K1	V, K1_infinity, E_K1, G_K1_bar
...	...	...
ODE Approximation	V	I_K1, ...

from the perspective of simulation where the module hierarchy is effectively ignored. Figure 2.3 is most relevant for this paper because it demonstrates the data dependencies in a model.

There are a few points about the model worth noting. First, there are a large number of static simulation parameters in the model representing 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 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 refers to the value of K1_infinity, which refers to alpha_K1 and so on, which is also shown in Table 2.1. This means that evaluating functions in the correct order is critical to simulation correctness. This can present a challenge when simulation elements are spread over multiple computation nodes.

Table 2.1 shows a subset of the calculations the simulator performs when approximating the value of V. 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.

A valid simulation of a model 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. In the following chapters we will discuss methods to do this in the context of parallel computing systems, and techniques to improve simulation speed.

Chapter 3

Source Based Biophysical Simulation

3.1 Introduction

As described in Section 1.2, there are numerous types of simulators for different biophysical models. Here we describe the *insilico*IDE 0.3 modeling environment and the simulations it generates. As of this writing, the current version of *insilico*IDE is 1.0. This version no longer uses these type of source based simulations. However, many of the techniques described here are applicable to the new simulator program *insilico*Sim, and are applicable to source based parallel simulations in other software packages.

*insilico*IDE is a multiplatform software package for modeling and simulating biophysical systems of varying timescales and sizes. Figure 2.2 shows an example model being edited in *insilico*IDE 0.3. *insilico*IDE can import models from CellML or ISML and export them as ISML. *insilico*IDE supports editing for a variety of models expressed by different means - ordinary and partial differential equations, morphological data, agent based models, etc. For the purposes of this discussion, we focus on the types of models described in Chapter 2.

In this chapter we describe *insilico*IDE, a graphical environment for creating and editing multi-scale models of biophysical systems for the purpose of physiome modeling. We also describe how these models are converted to source code for simulation, and how the source code is modified to run on parallel platforms. The structure of the biophysical models was described in Chapter 2. Section 3.2 details how *insilico*IDE generates simulation code for a model and what type of calculations the simulation performs.

3.2 Source Based Biophysical Simulation

To perform a simulation, *insilico*IDE generates C++ source code based on a biophysical model. This source code contains classes representing each module and control code to perform the simulation. Each module class contains variables representing states and parameters. The classes also contain functions that calculate the states and functions of the module for given inputs. The source code also contains control code that calls class functions in the correct order and performs the necessary approximation for ODEs.

*insilico*IDE simulations update states using either the Euler or fourth-order Runge-Kutta approximation method. In this dissertation we focus on the Runge-Kutta method because of its substantially higher accuracy and computational requirements. The Runge-Kutta method is used to approximate the solution over a series of T time steps each of length h, both specified by the user. This guarantees a bound of order h^4 on the accumulated solution error.

Algorithm 1 shows pseudocode of the simulation execution. After initialization, the program performs T simulation steps. Each step involves computing the Runge-Kutta values k_1, k_2, k_3, k_4 and using these to approximate the solution to the state ODEs. The calculation F in the pseudocode included function computations in the necessary ordering.

Algorithm 1 Pseudocode of *insilico*IDE source based simulations.

```
1:  Create modules with functions $F$ and states $S_0$
2:  $step$ = 0
3:  **while** $step < T$ **do**
4:      $k_0 = S_{step}$
5:      **for** $i$ in $[1, 4]$ **do**
6:          $k_i = F(k_{i-1})$
7:      **end for**
8:      $S_{step+1} = S_{step} + \frac{1}{6}k_1 + \frac{1}{3}k_2 + \frac{1}{3}k_3 + \frac{1}{6}k_4$
9:      $step$ += 1
10:     Write $S_{step}$ values to a log file
11: **end while**
```

3.3 Parallel Code Generation

Here we describe the techniques used by *insilico*IDE to perform fast parallel source based simulations for arbitrary heterogeneous models. A simple way of performing a parallel simulation is to evenly divide modules among compute nodes. Sophisticated methods involve minimizing communication by using a graph partitioning scheme to divide the simulation. However, these methods can be inefficient because of complex interactions not immediately apparent from the model.

Figure 3.1 shows a graph of an example model with four modules and their data dependencies. This graph contains vertices representing state and function calculations (2 and 5, respectively), and edges representing dependencies between these calculations. For simplicity, each function and state is considered to require the same amount of calculation. In the actual simulator, the calculation volume can be approximated by counting the number of operations.

A possible work division for two nodes (X and Y) based on graph partitioning would assign modules A and

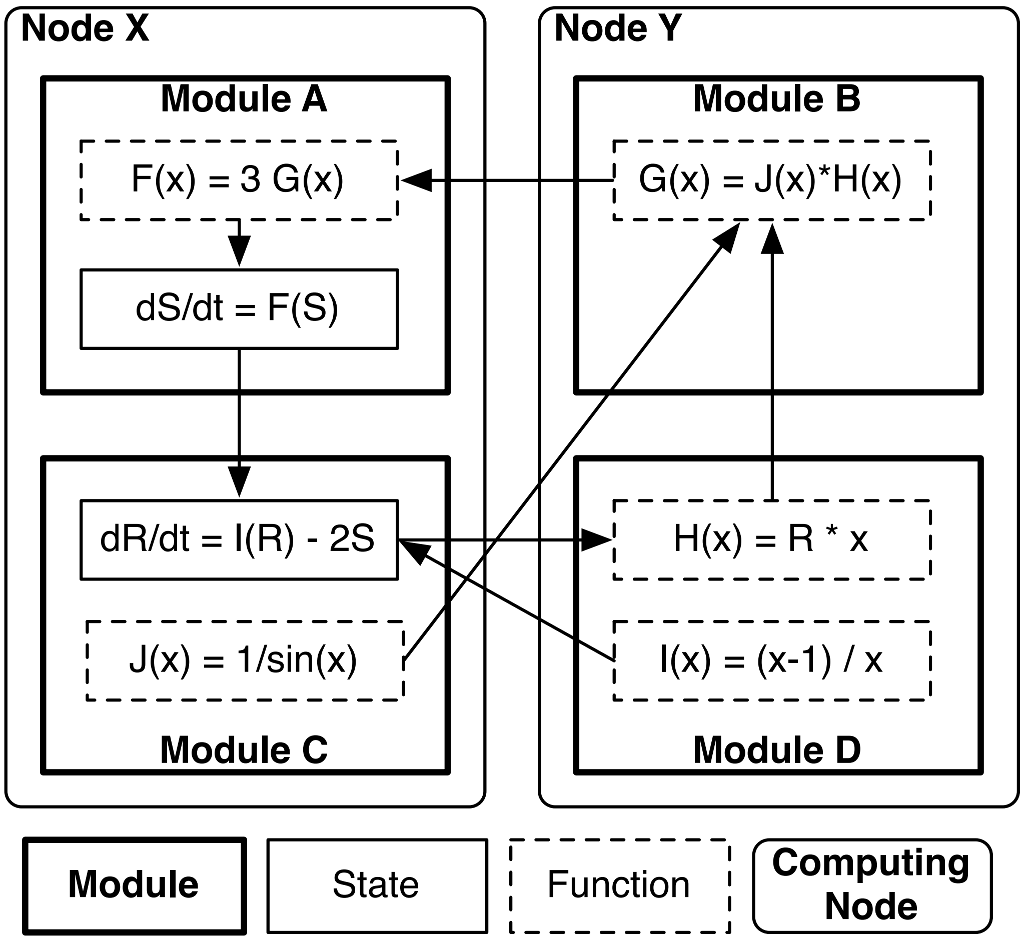


Figure 3.1: Sample model with four modules A, B, C and D, divided between two nodes, X and Y.

Table 3.1: Calculations to perform one simulation time step for the model in Figure 3.1.

Step	**Node X**	**Node Y**
1	Send R, S to Y	Receive R, S
2	Evaluate J(S)	Evaluate I(R), H(S)
3	Receive I(R)	Send I(R) to X
4	Send J(S) to Y	Receive J(S)
5	(none)	Evaluate G(S)
6	Receive G(S)	Send G(S) to X
7	Evaluate F(S)	(none)
8	Approximate $\frac{dS}{dt}$, $\frac{dR}{dt}$	(none)

C to node X and modules B and D to node Y. This is because the goal of most graph partitioning algorithms is to balance the number of vertices and minimize the number of edge cuts between nodes. This partitioning results in 2 modules on each node, and a total of 4 edge cuts. Each edge cut represents data that must be communicated from one node to the other.

Table 3.1 shows the pseudocode to perform one simulation step of this model. There are several problems that affect the efficiency of this simulation. First, due to the nature of the model, multiple communication phases occur (steps 1, 3, 4 and 6). Second, several steps (5, 7 and 8) leave one of the processors idle. Because of this, the overall simulation speed is affected. This problem occurs because the work partitioning does not distinguish between functions and states, and does not capture the true dependencies of each module. To improve the simulation, we modify the graph to distinguish dependencies between state and function vertices.

In this section, we present techniques for improving computational speed based on model analysis. First in Section 3.3.1 we examine the types of module data dependencies. Section 3.3.2 illustrates how this dependence information is used to improve speed by minimizing computation and communication. The division of work among multiple machines for minimum communication is described in Section 3.3.3.

3.3.1 Module Dependencies

In the parallel simulation, the fundamental unit of work distribution among computing nodes is the module. The reasons for this are the organization of the source code, and the desire for reusable code. We optimize the simulation speed through analysis of data dependencies among modules. As described in Chapter 2, a module may depend on an arbitrary number of other modules in multiple ways. *insilico*IDE analyzes these dependencies when generating the simulation, and uses the analysis to optimize the parallel simulation. The analysis requires no user assistance and takes a small fraction of the total time to generate a simulation. In the source based simulations, parameters are not part of the parallelization and can be ignored for the purposes of this chapter. Dependencies can be classified into four categories described as follows:

Function $\leftarrow$ Function

Function $\leftarrow$ Function dependencies occur when the evaluation of function F requires the output value of function G, denoted as $F \leftarrow G$. Naturally there can be no circular dependencies among functions. However, there may be nested dependencies spanning multiple modules, in which case all the dependent functions are recorded. In Figure 3.1, one Function $\leftarrow$ Function dependence is $F \leftarrow G, H, J$. This calculation spans modules A, B and C.

To record Function $\leftarrow$ Function dependencies, *insilico*IDE creates a directed acyclic graph (DAG) based on the immediate dependencies of each function. Since functions are used to update states, the DAG is referenced when determining State $\leftarrow$ Function dependencies. This gives a better estimate of the computational cost of a state.

State $\leftarrow$ Function

State $\leftarrow$ Function dependencies occur when the update of a state S requires the output of a function F, denoted $S \leftarrow F$. Because of Function $\leftarrow$ Function dependencies, a state update may depend on more functions than are immediately obvious in the ODE. Once immediate State $\leftarrow$ Function dependencies are determined, the DAG generated from Function $\leftarrow$ Function dependence analysis is referenced to determine the entire set of functions needed for a state update. For example, if $S \leftarrow F$ and $F \leftarrow G$ then the complete dependence relation is $S \leftarrow F, G$. In Figure 3.1, $S \leftarrow F$ is a direct dependence, but the entire set of dependencies is $S \leftarrow F, G, H, J$.

Function $\leftarrow$ State

Function $\leftarrow$ State dependencies occur when evaluation of a function F requires a state S as input, denoted as $F \leftarrow S$. Using this information can decrease computation and communication time. For example, if $F \leftarrow S$ and $T \leftarrow F$ for states S and T, then communication and computation can be reduced by placing the modules containing S and T on the same computing node. In Figure 3.1, an example Function $\leftarrow$ State dependence is $I \leftarrow R$.

State $\leftarrow$ State

State $\leftarrow$ State dependencies occur when the update of state S requires the value of state T, denoted as $S \leftarrow T$. Unlike Function $\leftarrow$ Function dependencies, State $\leftarrow$ State dependencies can have circular relationships because state values are updated simultaneously. In other words, if at time t there are states S_t and T_t with $S \leftarrow T$ and $T \leftarrow S$, then the value of $S_{t+1} = f(T_t)$ and $T_{t+1} = g(S_t)$. In Figure 3.1, an example State $\leftarrow$ State dependence is $R \leftarrow S$.

3.3.2 Improving Simulation Speed with Dependence Information

The dependence information described in Section 3.3.1 is used to ensure the correctness of the simulation and to improve simulation speed. Two techniques based on the model analysis are used to decrease total communication and thereby improve simulation speed.

The dependence information is recorded in the simulation C++ source file. State $\longleftarrow$ Function (implicitly including Function $\longleftarrow$ Function) and State $\longleftarrow$ State dependencies are recorded in each module class. This effectively records what states and function evaluations are needed to update the states in each module. Function $\longleftarrow$ State dependencies are recorded separately from the modules. This is to distinguish Function $\longleftarrow$ State dependence from State $\longleftarrow$ State dependence, which is necessary to perform the optimization described below. Techniques to improve simulation speed are described in the following two sections.

Redundant Function Evaluation

For every simulation time step, each node must evaluate the functions necessary to update the states of its modules, or receive the value of these function evaluations from another node. Function dependence information recorded in the modules is referenced so that only the necessary functions are calculated by each node. This way each node will avoid evaluating functions not needed for its state updates.

Ideally, states and their dependent functions will be on the same compute node, though this cannot be guaranteed. Function dependencies may span multiple modules, therefore the issue arises of how to handle functions needed by multiple compute nodes. As shown in Table 1, communicating module function evaluations between nodes can slow the simulation. Because function evaluation time is relatively fast compared to communication time in most networks, multiple nodes can redundantly evaluate functions to increase overall speed.

Figure 3.2 gives an example of this. Although Modules A and B are on different compute nodes (nodes 1 and 2, respectively), they both require function F in Module C. In turn, function F requires function G in Module D. To avoid excess communication, both nodes 1 and 2 evaluate F and G independently. By only evaluating necessary functions and redundantly computing functions on multiple nodes, communication can be reduced and total simulation time decreased.

State Communication

The State $\longleftarrow$ State and Function $\longleftarrow$ State dependencies correspond to state communication between modules. An even division of modules among compute nodes can yield balanced computational load to each node, but may result in unbalanced and excessive communication. This is because some modules communicate many states while others communicate relatively few.

By recording State $\longleftarrow$ State and Function $\longleftarrow$ State dependencies, modules with similar state requirements can

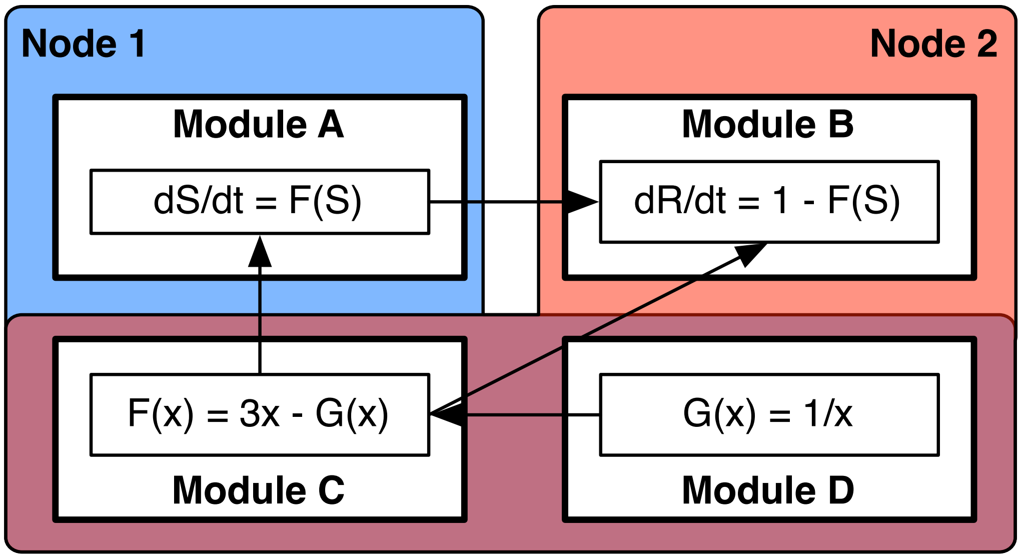


Figure 3.2: An example of redundant function calculation.

be grouped on compute nodes to minimize communication between nodes. In the example from Section 3.3, the state dependencies are not completely known because of the chained function dependencies. This in turn leads to poor performance due to excessive communication. For example, through the chain of dependencies from Function F to Function H, we see that module A implicitly requires state R for an update of state S, which is not shown in the figure. The technique for resolving this is detailed in the next section.

3.3.3 Work Partitioning

The unit of work division for a source based parallel simulation is the module. More specifically, it is the task of updating the states of a module. This includes function calculation and state retrieval from other modules. At the start of the parallel simulation, modules must be partitioned among compute nodes. In this section we explain how modules are divided among compute nodes to maximize simulation speed.

Let there be P compute nodes $N_0, N_1, \ldots, N_{P-1}$ performing the simulation. The root node N_0 calculates a mapping of D modules $M = \{M_0, M_1, \ldots, M_{D-1}\}$ to the compute nodes, where each module is assigned to one compute node. To compute this mapping, we represent the simulated model as a graph, with graph vertices corresponding to modules and graph edges corresponding to state dependencies between modules.

The graph representing the modules and their state dependencies is generated as follows. The vertices V of the graph correspond to the modules M. A module M_i with state S is connected to a module M_n with state R if $S \leftarrow \ldots \leftarrow R$ where $\ldots$ represents zero or more Function $\leftarrow$ Function dependencies (using the dependency information described in Section 3.3.1). This means the new module graph explicitly includes Function $\leftarrow$ State dependencies which are not directly evident from the model. This also effectively deletes the Function $\leftarrow$ Function

Table 3.2: Steps for simulation of the model in Figure 3.3.

Step	**Node X**	**Node Y**
1	Send S to Y	Receive S
2	Receive R	Send R to X
3	Evaluate H, J, G, F	Evaluate I
4	Approximate $\frac{dS}{dt}$	Approximate $\frac{dR}{dt}$

and State $\leftarrow$ Function dependencies, but only for the purpose of work partitioning (the data are retained in the simulation file). To estimate the true computational cost, each vertex is assigned a weight representing the number of functions needed to update the states at that vertex, or in other words the calculation cost of the module. This is slightly inaccurate because some functions are shared by multiple state calculations, but in our experience it is a reasonable approximation.

To obtain a mapping of the modules to the P compute nodes, we use the function $METIS\_PartGraphKway$ from the serial graph partitioning library METIS [67]. This function takes as input a graph and number of partitions P, and returns a mapping of each vertex in the graph to a partition. The mapping is computed with two goals: 1) the sum of weights of vertices in each partition is roughly equal and 2) the total number of edges crossing partitions is minimized. In the simulation, these mapping goals respectively correspond to 1) equal computation at each compute node and 2) minimal communication between nodes.

Some applications use a scheduling algorithm to divide work in a parallel computing environment. In this case, such an algorithm will not help for two reasons. First, scheduling algorithms are computationally intensive, and will take significant time given the the large numbers of elements in a model. Second, the goal of these algorithms is most commonly to minimize makespan (total execution time) when the computation times of each element and communication times between elements are known. Given that computation times are on the order of nanoseconds and that communication times will vary wildly on the expected timescale due to network effects, it is far more worthwhile to minimize communication rather than trying to synchronize computation and communication between different processors.

The output of METIS is equivalent to a mapping of each module to a compute node. Given this mapping, the root node determines for each compute node: 1) which states it must evaluate 2) which functions it must evaluate 3) which states it must send/receive to/from which processors. Finally, the root node sends this information to every node and the simulation begins.

Figure 3.3 shows the effect of applying the techniques to the original example model. The graph in Figure 3.1 becomes the graph in Figure 3.3 through a series of steps which preserve simulation correctness. First, State $\leftarrow$ State and State $\leftarrow$ Function dependencies are left intact. Next, Function $\leftarrow$ Function dependencies are removed and noted implicitly through the weights of each module. The dependency information is retained in the simulation to ensure correct calculation, but for communication and graph partitioning it is ignored. In Figure 3.3, Module

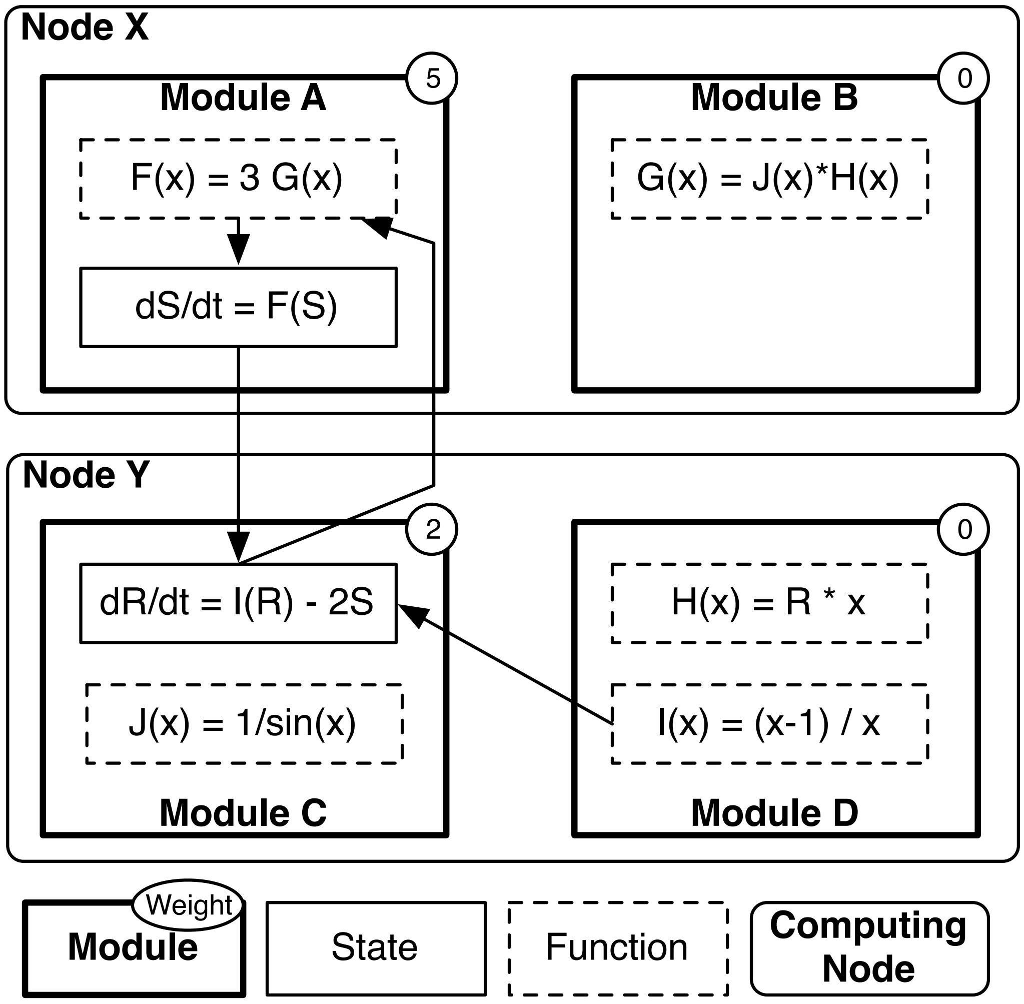


Figure 3.3: The model from Figure 3.1 after using the described techniques.

Table 3.3: Simulation model details.

Model Name	Modules	States	Functions
Respiratory	1209	921	2264
Motor	3781	2244	3960
Cardiac5	740	200	830
Cardiac10	2980	800	3360
Cardiac20	11960	3200	13520

A has weight 5 because it represents the calculation of 1 state and 4 functions (F, G, H, J). Module C has weight 2 because it represents the calculation of 1 state and 1 function. Finally, Function $\leftarrow$ State dependencies are handled. We can treat the state dependencies of function F as the union of all state dependencies of the children of F (functions G, J, H). This means that any node which calculates F must receive the states needed to calculate the children of F and ensure simulation correctness. In turn, state S (which uses the result of F) also requires these states. Therefore, the previous dependency H $\leftarrow$ R is transformed into the dependency F $\leftarrow$ R.

In the new graph, only state communication edges exist because function calculation is performed locally on each node and function evaluation results are not communicated between nodes. Although modules C and D belong to node Y, the evaluation of functions J and H actually occurs on node X. Modules B and D have weight 0 because they do not perform any state updates. The resulting simulation steps in Table 3.2 show a decrease in complexity, communication and idle processor time. With the techniques proposed in this dissertation, the simulation is greatly simplified and can be performed in 3 basic stages: communication, function evaluation and state calculation.

3.4 Experiments

To confirm the effectiveness of our techniques for optimizing source based simulations from *insilico*IDE, we performed experiments in parallel computing environments. In Section 3.4.1 we describe the setup for the experiments and the models used in the experiments. Section 3.4.2 details the effect of our techniques on communication in the simulations, while Section 3.4.3 examines the cost of redundant function computation. Finally, in Section 3.4.4 we show the runtime of the simulations in parallel computing environments.

3.4.1 Setup

The simulations were performed using five biophysical models. The first model is of eight connected neurons based on the Rybak model of respiratory neurons [66] and is referred to as “Respiratory”. The second model is of a spinal motor rhythm generator network [68] and is referred to as “Motor”. Models three through five represent grids of myocardial (heart muscle) cells [69] in a 5x5 grid (“Cardiac5”), a 10x10 grid (“Cardiac10”) and a 20x20 grid (“Cardiac20”). Details of each model are shown in Table 3.3.

For each model, the simulation was run for $T = 10000$ time steps. To test the parallel simulation, we used two environments - a multicore machine and a networked cluster. The multicore machine is a dual quad core processor Intel Xeon 2.83 GHz with 4 GB of RAM. The cluster consists of 16 HP ProLiant G2 machines, each with dual 2.8 GHz Xeon processors and 2 GB of RAM connected by Gigabit ethernet.

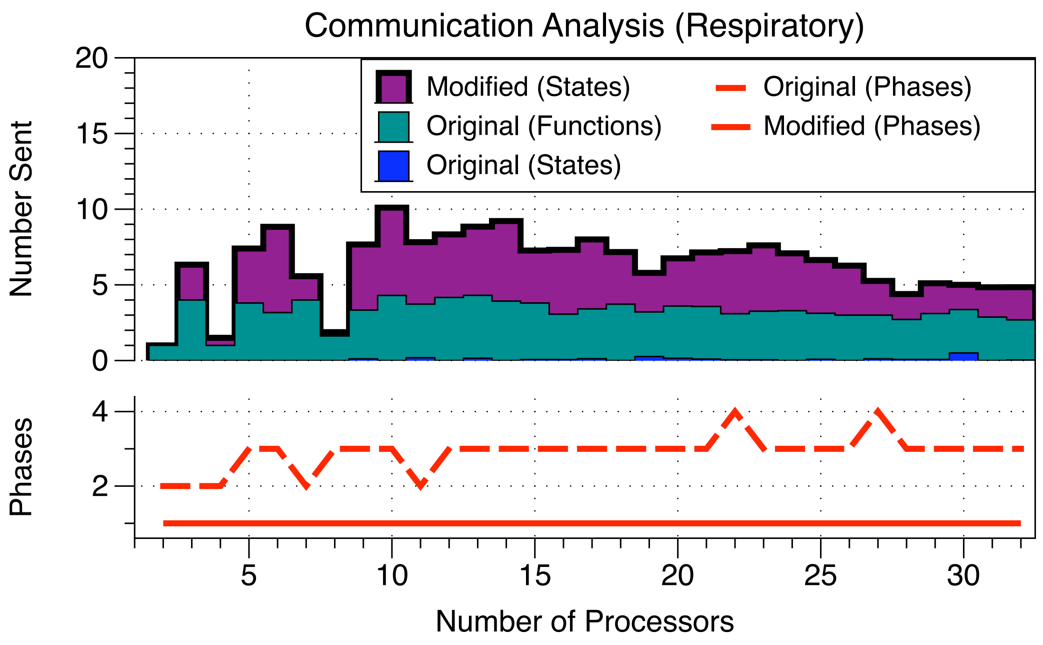
3.4.2 Simulation Communication

One key argument in this dissertation is that using the techniques described in Section 3.3 will reduce communication by simplifying the model (e.g. Figure 3.1 to Figure 3.3). To confirm this, we performed an analysis of communication patterns using the original model versus the modified model. The results of this analysis are presented in Figures 3.4 and 3.5. For each model, these figures show the average number of states and functions communicated between nodes, and the number of communication phases needed per simulation time step. In this dissertation, a communication phase is a set of MPI sends and receives between nodes that is uninterrupted by calculation. In all simulations, the communication data size of a state and function result are the same (8 bytes).

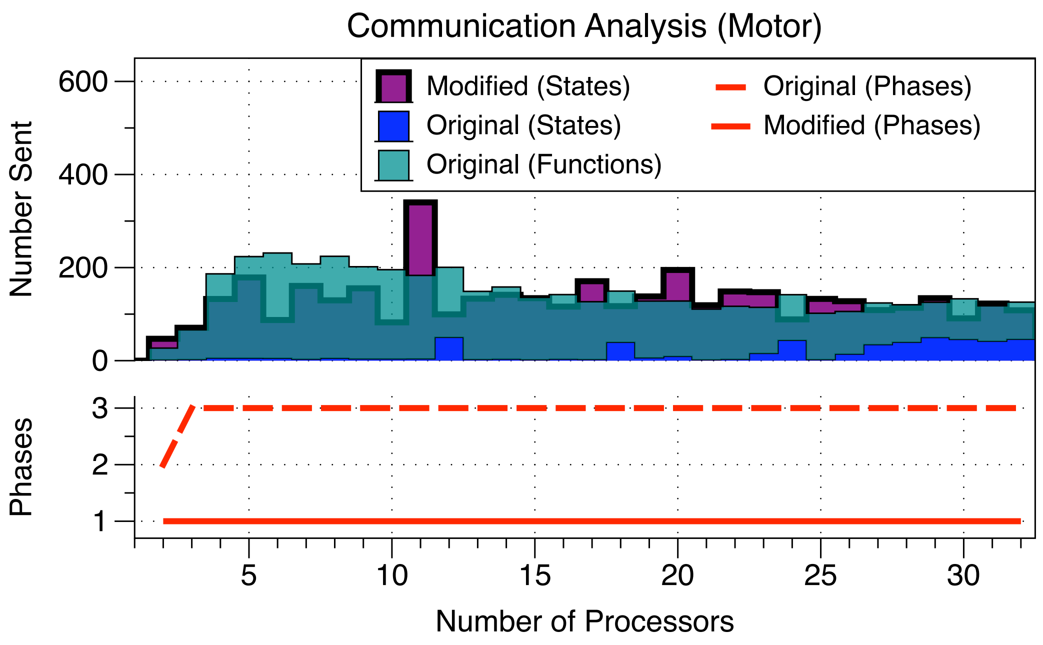
As seen in Figure 3.4(a), the modified model communicates more states than the original model for most configurations when simulating the respiratory neuron model. However, the tradeoff is that the original model must perform up to 4 communication phases per simulation time step. By simplifying the model, some additional state communication is added, but overall communication time is decreased through fewer communication phases. It is also worth noting the reduced communication when using 2, 4 or 8 nodes - this is because the respiratory model breaks cleanly into 8 pieces.

Figure 3.4(b) shows the effect on communication for the motor network simulation. This model is different from the respiratory model in that many of the components are strongly connected by states. In this case, the original simulation is optimally partitioned such that there is little state communication, though this in turn causes multiple communication phases due to function communication. Also, the number of communication phases stays relatively low regardless of the number of computing nodes, indicating that there is less inter-module functional dependence than in the respiratory model. This also explains why function communication is preferred in the original model simulation, specifically because the modules are not strongly connected by functions. The sudden increase in states when using 11 nodes is due to the structure of the model.

Figure 3.5 shows the effect on communication in the three cardiac cell models. In this case, the grid nature of the model means few communication phases are required because of the independent nature of each grid element (heart cell). However, by eliminating function communication in the modified model, model division follows the grid better than in the original model. This is noticeable, for example, in the Cardiac5 model where using 5 nodes reduces communication in the modified model but increases it in the original model. Using more compute nodes tends to increase the number of communication phases because of the higher likelihood of a Function $\leftarrow$ Function

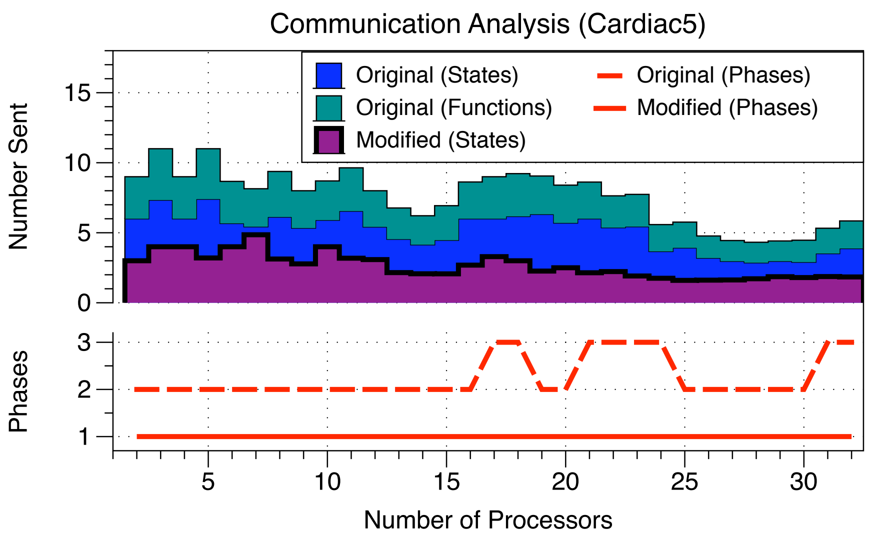


(a) Respiratory model communication.

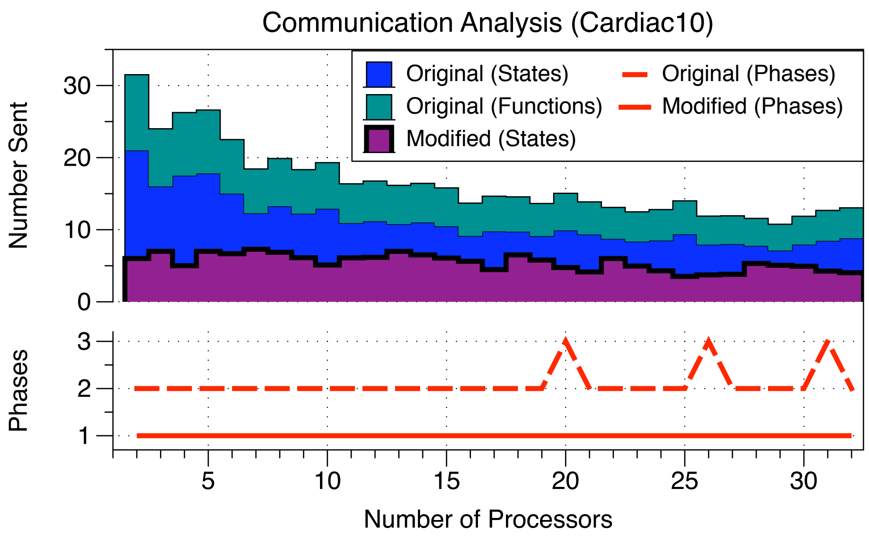


(b) Motor network model communication.

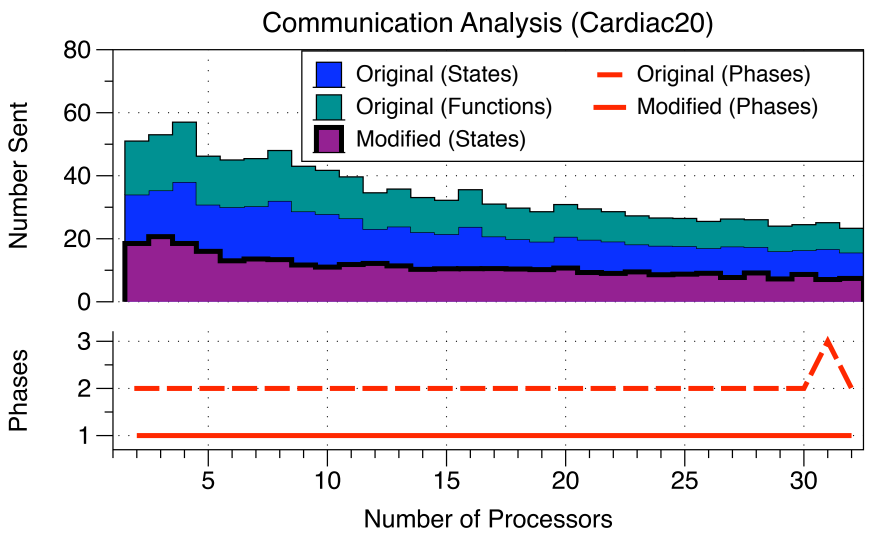
Figure 3.4: Comparison of communication between original and modified models.



(a) Cardiac5 model communication.



(b) Cardiac10 model communication.



(c) Cardiac20 model communication.

Figure 3.5: Communication comparison between cardiac models.

dependence being split across two nodes.

These figures demonstrate that by removing function dependencies from the model, the described techniques may increase the number of state dependencies (e.g. the respiratory model). However, because these states can be calculated independently the number of communication phases is reduced to 1. Furthermore, in many cases the increased state communication is offset by the elimination of function communication, particularly in the cardiac models and the motor network model. Of course the actual effect of the techniques on a given model will vary depending on the model characteristics. If a model is composed exclusively of states, the techniques will do nothing. However, in our experience there are few if any models that are so homogeneous.

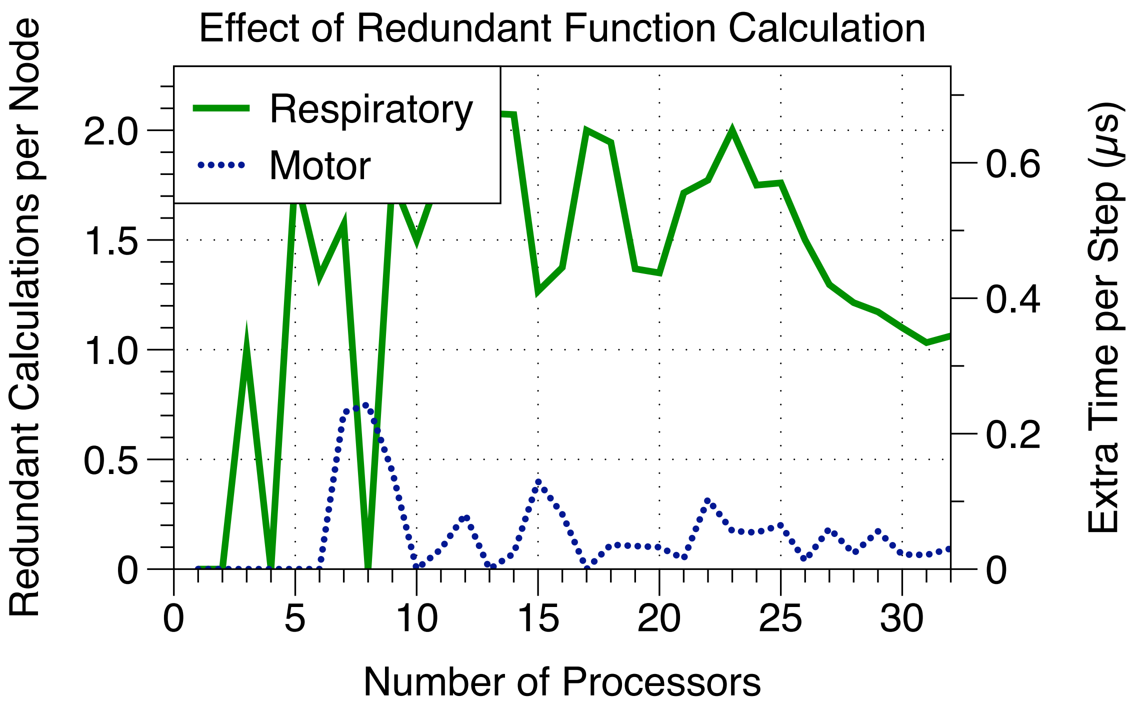
3.4.3 Redundant Function Computation

Section 3.3.2 describes a technique to reduce communication with redundant calculation of functions. Naturally, excessive redundancy will cancel the gain made by decreasing communication. To confirm the gain from redundant calculation, we examined the average number of excess function calculations per node. Figure 3.6 shows the effect of redundant calculation in terms of extra functions calculated and additional computation time per simulation step.

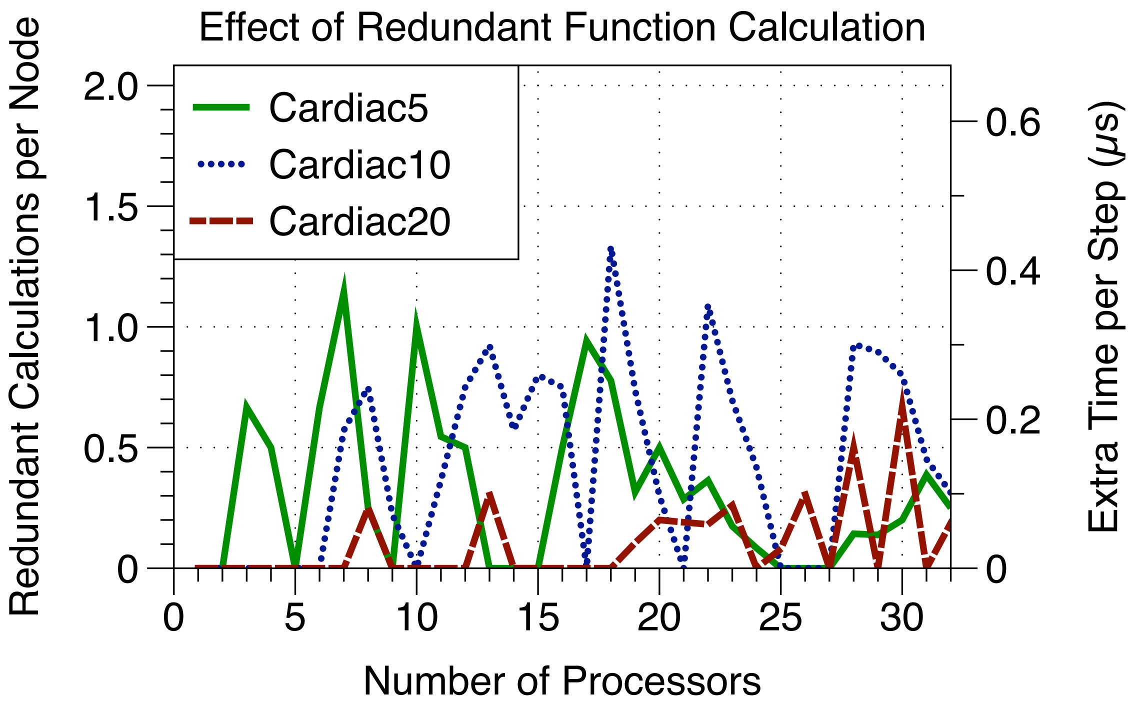
Depending on the model, the effect can vary widely. The respiratory model breaks cleanly into 8 pieces, so there is no redundant computation when using 2, 4 or 8 nodes. Functions in the motor model are not strongly connected so there is generally little or no redundant function calculation. The cardiac grid models also have little or no redundant function calculation when divided among a number of nodes that fits the grid well (e.g. 5 nodes for Cardiac5, 10 nodes for Cardiac10, etc).

Based on the execution of a serial simulation, we estimate the average time to compute a function as 0.32 μs, which means redundant function calculation adds at worst 0.68 μs of computation to each simulation time step. This is well under the latency of most communication channels and indicates a clear benefit of using redundant function calculation. For example, Myrinet has a minimum latency of 2.6 μs and Infiniband has a minimum latency of 1.1 μs, both of which are well over the average time spent on redundant calculation.

It is worth noting that this technique will not always provide a benefit. If a model has highly redundant usage of functions between modules, redundant calculation of these functions may be worse than communicating them, especially on fast networks. However, given the results from these tests we feel confident that this shouldn't be a problem for the vast majority of biophysical models. This is particularly true because these models are often of physical phenomenon. For example, if a 3D biophysical model was being simulated, an increase in model size along all dimensions would correspond to cubic growth in computation. However, since the interface between model elements is a surface, this increase will generally cause only a quadratic increase in communication. Therefore, for models of biophysical phenomenon we feel this technique should remain effective.



(a) Redundant computation in the respiratory and motor models.



(b) Redundant computation in the cardiac models.

Figure 3.6: Average redundant function evaluations per simulation time step.

3.4.4 Simulation Execution

Next we examine the running time of these simulations on multicore and cluster environments. Figure 3.7 shows an example partitioning of the respiratory model over 16 processors, with the assignment of each module to a processor indicated by color. This partitioning was created by the METIS library, and required less than 0.1 seconds on the above mentioned multicore machine. This demonstrates that graph partitioning takes relatively little time compared to the entire simulation, even for large models partitioned over many nodes.

Figures 3.8 and 3.9 show the results of performing the model simulations in the multicore environment. Figures 3.10 and 3.11 show the results for the cluster environment. These figures show the total simulation execution time for different numbers of nodes, the speedup of the computational part of the simulation, and the total speedup. Initialization time (including graph partitioning and work division) was negligible and is not included. We found little deviation in the run times for multiple executions, so these figures represent only a single execution.

Figures 3.8 and 3.9 show simulation performance in the multicore environment. The simulations scale very well on multicore machines, achieving a nearly linear speed increase in both function and state computation. Total simulation speed approaches a linear speedup for all but the Cardiac5 model, likely because of its relatively small size. The large myocardial simulation (Cardiac20, Figure 3.9(c)) shows a notable superlinear speed increase, likely due to improved cache performance. As expected, communication time is minimal on multicore machines, though for smaller models it can become problematic for large numbers of nodes. With larger models, communication accounts for a relatively small part of simulation time and more nodes will yield better performance.

Figures 3.10 and 3.11 show simulation performance in the cluster environment. In this environment there is greater sensitivity to communication time because of the network latency and thus simulations do not scale as well. For simulations performed on the cluster, communication dominates as the number of nodes increases, particularly for smaller models. As seen in Figure 3.10(a) and Figure 3.10(b), performance peaks around 16 nodes for the neuron models. The cardiac models scale poorly on the cluster due to their size, especially Cardiac5. However, the calculation of functions and states scales well for larger models as shown in the bottom graphs. The Motor and Cardiac20 simulations achieve a slightly superlinear computation speedup with multiple nodes.

From these experimental results, we have demonstrated the effectiveness of model analysis/reorganization and redundant function calculation when performing parallel source based simulations of heterogeneous biophysical models. These techniques are applicable to other source based simulations, and have also been applied to the *insilico*Sim biophysical simulator described in the next chapter.

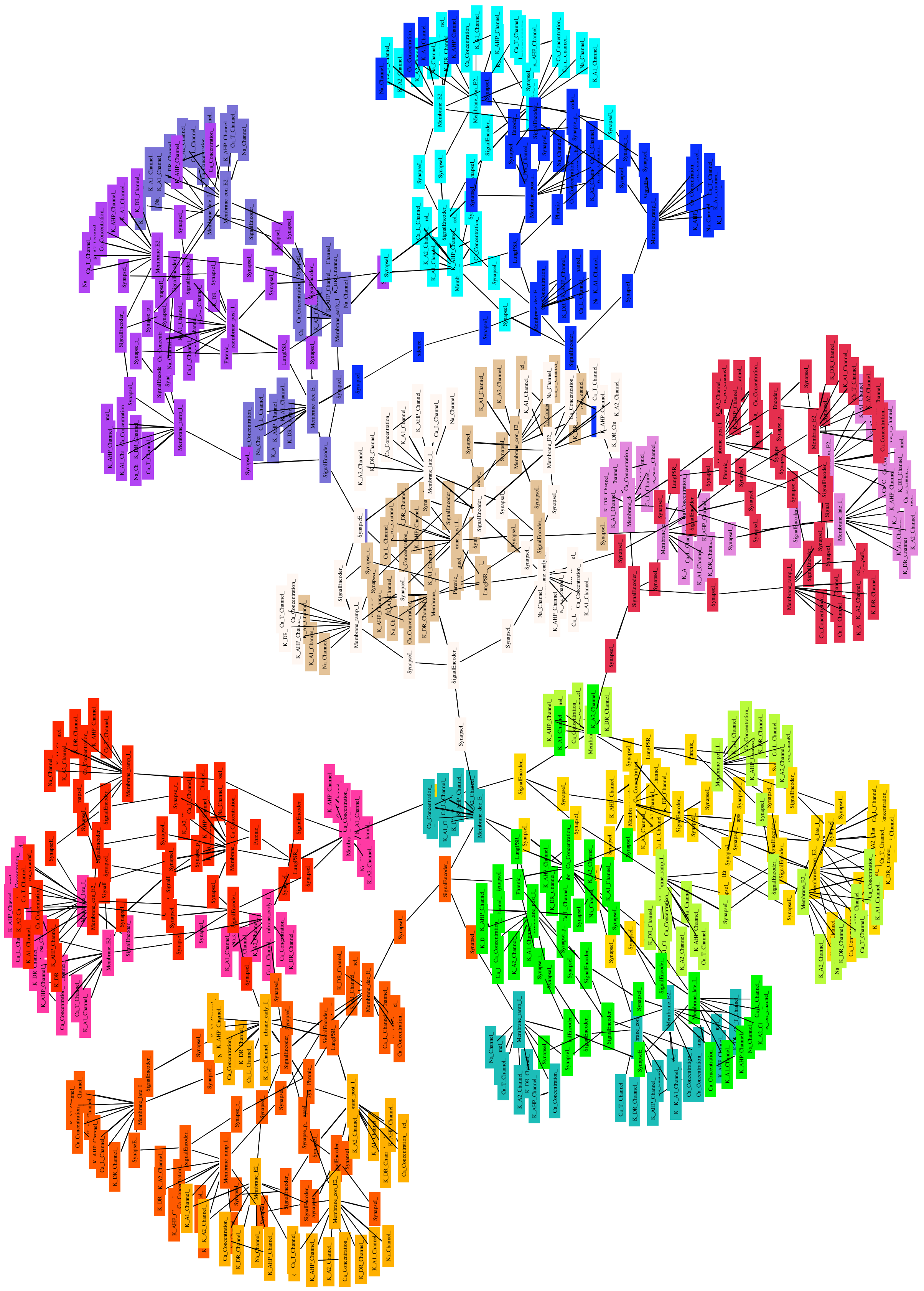
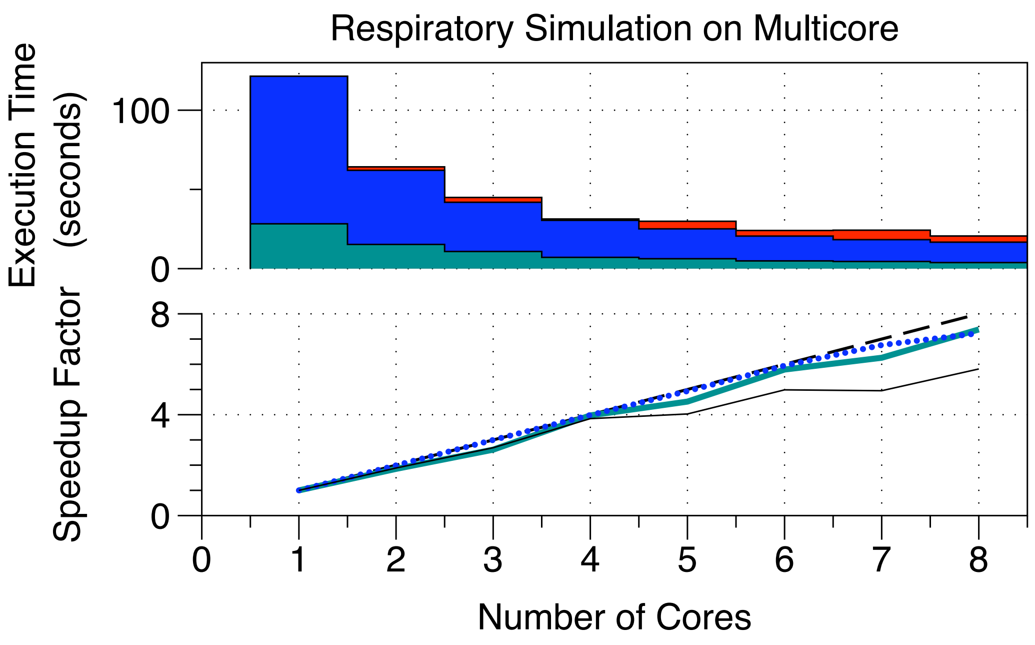
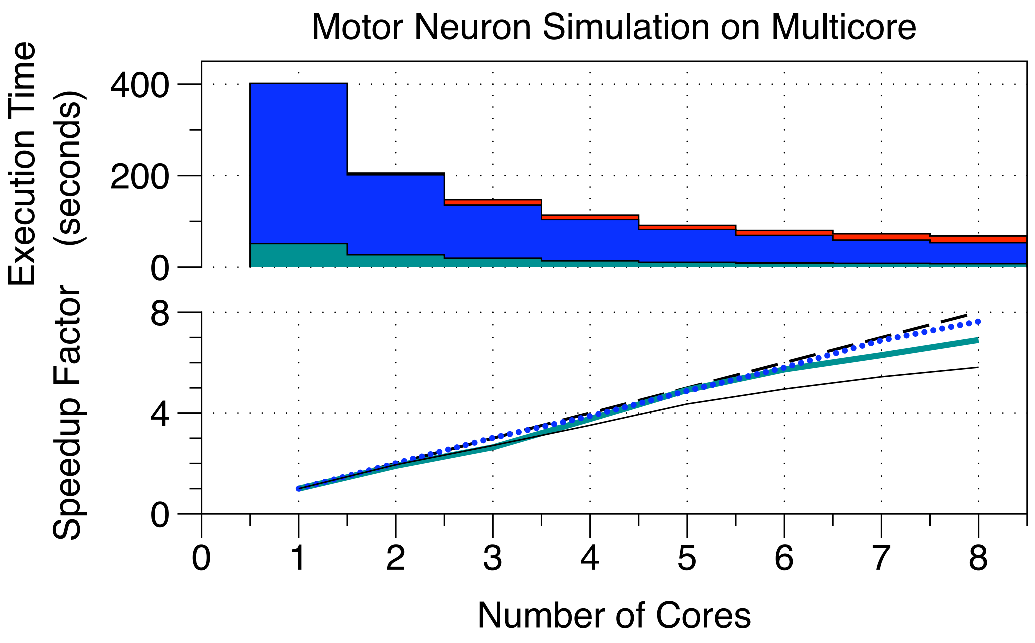


Figure 3.7: Example partitioning of respiratory model over 16 processors.



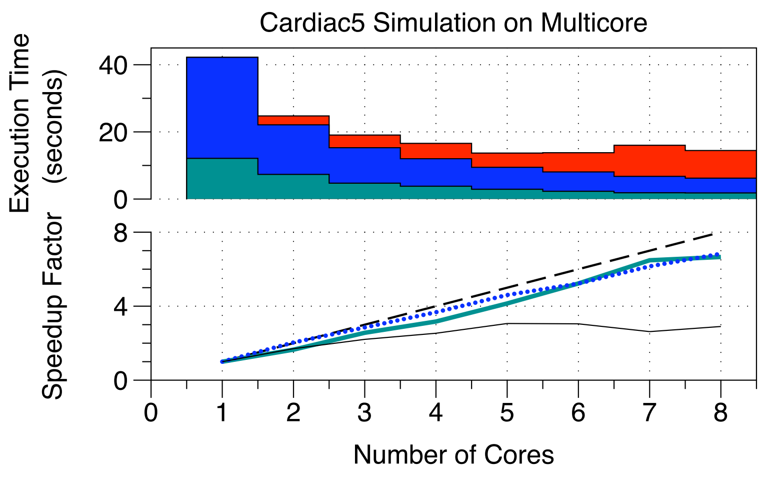
(a) Respiratory simulation times.



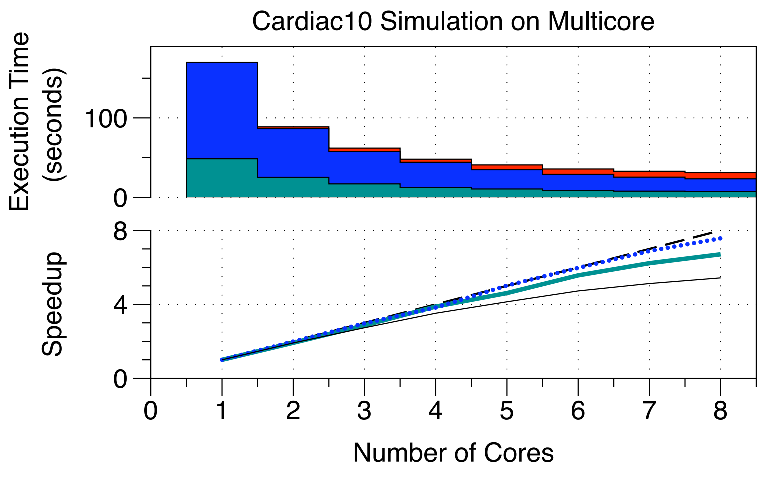
(b) Motor network simulation times.

🟥 Communication	— Function Speedup	— Linear Speedup
🟦 State Calculation	⋯ State Speedup	
🟩 Function Calculation	— Total Speedup	

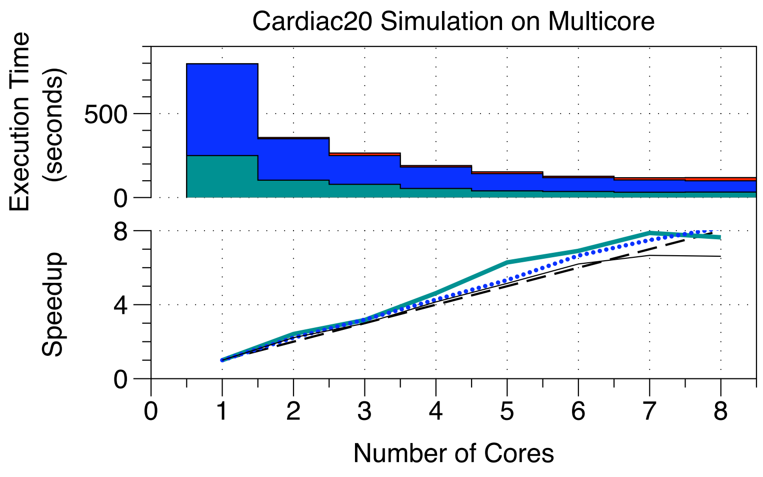
Figure 3.8: Motor and respiratory simulations performed on multicore machine.



(a) Cardiac5 simulation times.



(b) Cardiac10 simulation times.



(c) Cardiac20 simulation times.

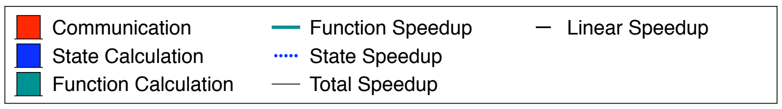
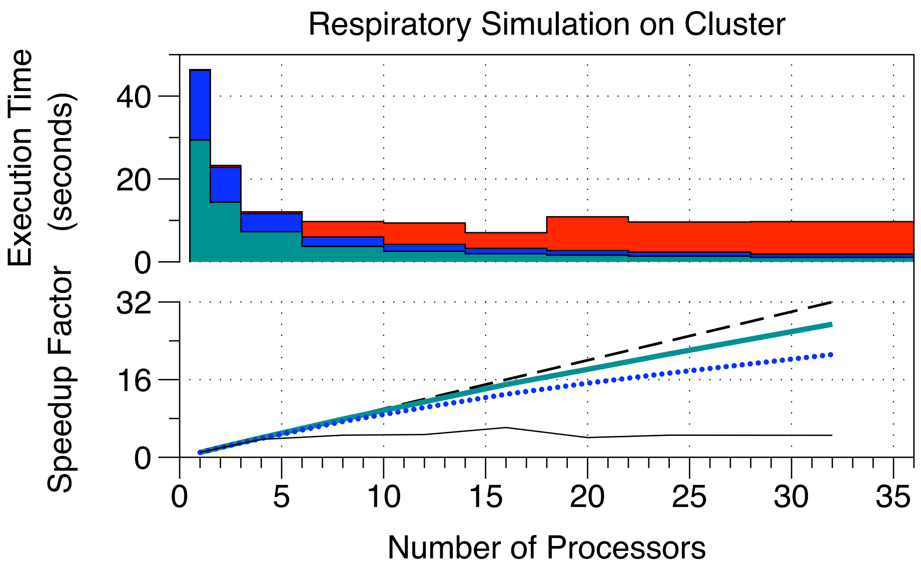
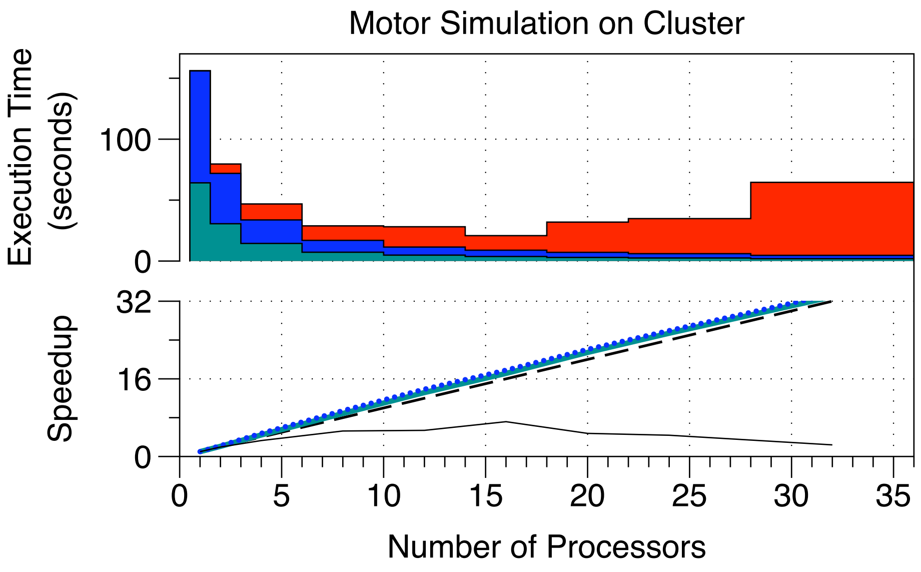


Figure 3.9: Cardiac simulations performed on multicore machine.



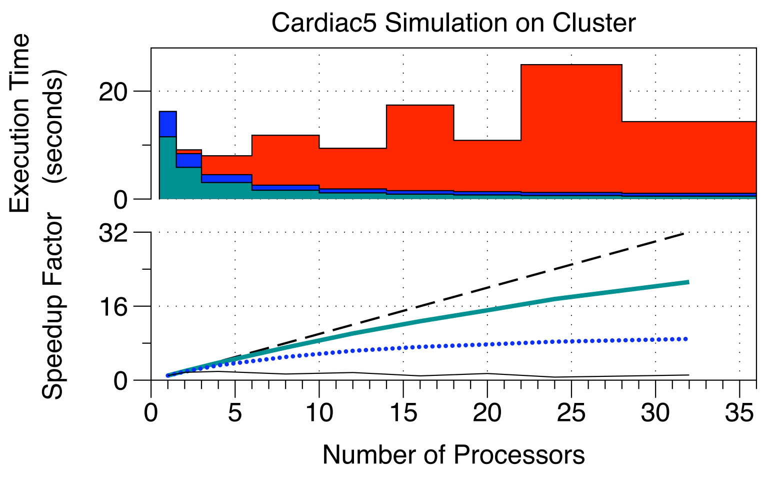
(a) Respiratory neuron simulation times.



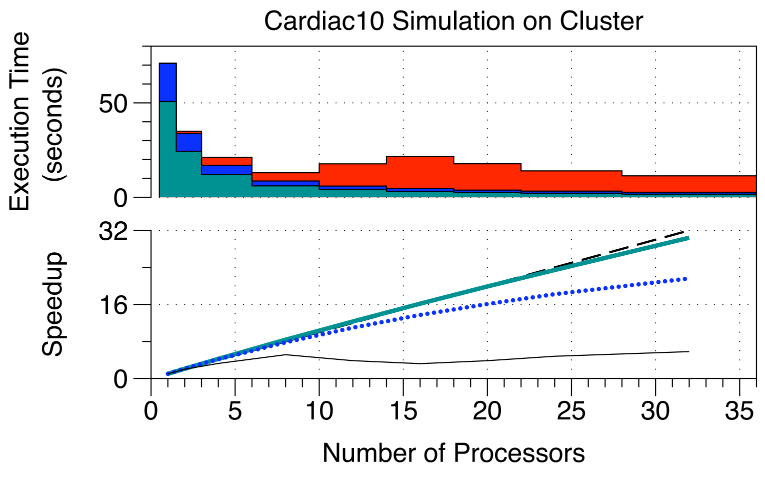
(b) Motor network simulation times.

🟥 Communication	— Function Speedup	– Linear Speedup
🟦 State Calculation	····· State Speedup	
🟩 Function Calculation	— Total Speedup	

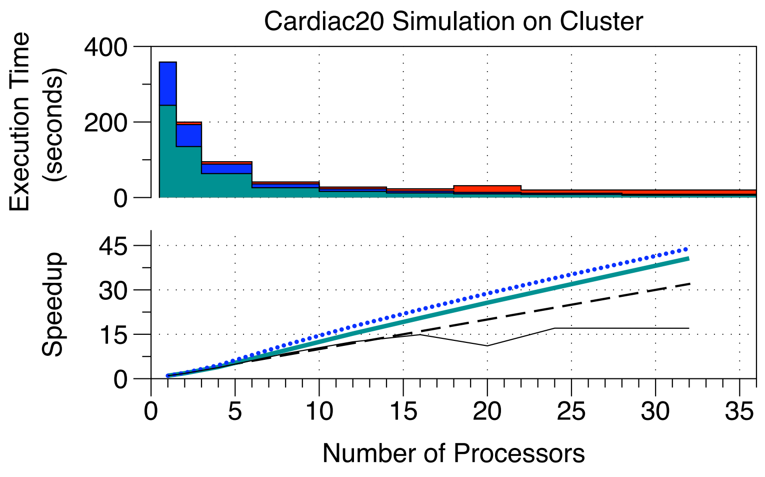
Figure 3.10: Motor and respiratory simulations performed on cluster.



(a) Cardiac5 simulation times.



(b) Cardiac10 simulation times.



(c) Cardiac20 simulation times.

🟥 Communication	Function Speedup	Linear Speedup
🟦 State Calculation	State Speedup	
🟩 Function Calculation	Total Speedup	

Figure 3.11: Cardiac simulations performed on cluster.

3.5 Conclusion

In this chapter, we introduced the *insilico*IDE program for developing biophysical models. We demonstrated how this program generates source based simulations of biophysical models and the problems involved in running these simulations in parallel. In Section 3.3 we described techniques to improve parallel simulations by reconfiguring the model and using redundant computation to reduce communication steps in the simulation. These techniques were experimentally verified in Section 3.4. As mentioned previously, *insilico*IDE 1.0 no longer uses source based simulation but rather a separate simulation program. However, the techniques described here still apply to simulations performed by this program, and potentially to irregular simulations in other fields.

Chapter 4

*insilico*Sim Biophysical Simulator

4.1 Introduction

Currently available simulators for the types of models common to physiological modeling are often limited in their scope and functionality. Simulators are available for molecular level simulations [21, 22] or cellular particle simulations [70] 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. They are also unable to simulate models of the sort common to physiome modeling. Physiome oriented parallel simulators tend to be very domain specific and focus on one type of model. These are usually specific to neuronal networks [49, 50, 51, 52, 53, 54] or cardiac modeling [56]. Other simulators allow general models described with domain specific markup languages [59] [71], but do not offer parallel computing needed for large scale models. To the best of our knowledge, the simulator described in this dissertation is the first generalized physiome oriented simulator with parallel computing abilities.

Here we describe the structure and function of the *insilico*Sim program which performs parallel simulations of heterogeneous biophysical models. In designing and implementing *insilico*Sim, there were three main goals:

1. Make the system easily extendable to additional input/output data formats and simulation techniques.
2. Allow efficient parallel simulation of large scale models over multiple processors.
3. Achieve high performance, comparable to model compilation and execution in an interpreted language.

The main difficulty in implementing *insilico*Sim is that the first goal is in opposition to the second and third goals. Creating an extendable system makes it more difficult to perform parallel computation because there must be a common method for simulator extensions to share data with each other and between processors. Also, because there are a wide variety of models and simulation techniques, it is necessary to make the simulation tools more generic. Because of this they cannot be tailored to a particular type of simulation, and it is difficult to achieve high performance. We describe the techniques used in *insilico*Sim to simultaneously achieve each of these goals.

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 heterogeneous 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.

The other goals of the program include allowing parallel and relatively high performance simulations. Parallel high performance 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. This is because large scale parallel simulations are difficult to implement, and are often limited to simulating a particular model.

In this chapter we describe design decisions that allow for heterogeneous simulations, including the use of user enabled interfaces and a shared object and value manager. Interfaces allow the user to extend the simulator to smoothly integrate additional libraries or tools. An object manager allows interfaces to ignore heterogeneity and focus exclusively on the set of objects they want to work with, while sharing these objects with other parts of the simulator. The object manager also allows for efficient transparent parallel synchronization through runtime dependency analysis on the model.

High performance simulations are achieved by internally compiling objects into byte code, and remapping abstract simulation values onto local arrays for fast access. Simulation calculations are represented using tree data structures which are necessary for model creation and editing, but relatively slow to traverse. Simplifying these structures and compiling them into byte arrays improves simulation performance by up to a factor of 24x with negligible impact on simulation initialization time.

This chapter is structured as follows. The core simulation engine and interfaces are described in Section 4.2. The techniques for improving simulation performance and enabling parallel simulations are described in Section 4.3. We experimentally demonstrate the effectiveness of these techniques using several models in Section 4.4.

4.2 *insilico*Sim

*insilico*Sim is a program written in C++ designed to perform large scale parallel simulations of biophysical models specified by markup languages such as ISML, CellML and SBML. 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

Table 4.1: 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 object model.
SBML	Import	Parses an SBML file, and converts it to the internal object model.
CVODE	Computation	Performs ODE approximation using the CVODE library.
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	Displays simulation progress and estimated time to completion.

step. Examples of interfaces and their behavior are shown in Table 4.1. The interfaces can be roughly grouped into 3 categories - import, computation and export. Import interfaces create internal model representations, such as the ISML interface making an internal representation of an ISML model file. Computation interfaces perform simulation related calculations based on the model. Export interfaces output representations of objects or values, such as writing the value of variables to a file during each time step.

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 4.2.1 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 4.2.2, with some implemented interfaces described in Section 4.2.3.

4.2.1 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 [72] for parallel load balancing and MPI for parallel communication.

The pseudocode for initializing and running the core engine is shown in Algorithm 2. 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(). During the simulation, each interface will be called in the order it is registered. 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.

4.2.2 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

Algorithm 2 Pseudocode to Run Core Engine

```
1: simulator = new insilicoSim(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()
```

by the core engine. This allows developers to add functionality to the simulator while smoothly interacting with other preexisting interfaces 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 different types of objects (for parallel load balancing), whether 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 and/or 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 each time after load balancing is performed by the core engine. It should be used to reset interface specific data 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 the 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 state of the simulation, such as the simulation time and MPI processor rank.

```
void finalize(void)
```

The calling of this function indicates the end of the simulation. The interface should free data structures it allocated and close any open files.

4.2.3 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 to different types of solvers for the model ODEs to an interface for outputting simulation results.

ISML

The ISML interface parses ISML (in silico Markup Language) model files and creates an internal representation of the model. This is accomplished with a MathML parser that creates simulator specific data structures, and the model tools to analyze data dependencies across modules. A detailed explanation of ISML is outside the scope of this dissertation, details are available in the specification [20].

This interface uses the Xerces XML parsing library to read an ISML file. It then constructs a corresponding model in the simulator. One of the difficulties in dealing with biophysical model files is the hierarchical nature of the model. An expression may refer to the value of another expression that is connected through multiple layers and pieces of the model, and these must be correctly analyzed to construct the internal model representation. *insilico*Sim provides graph related tools to analyze hierarchical models such as those used in ISML and CellML.

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 future versions of *insilico*Sim, it will be possible to specify how to relate components of model files in different notations, and perform simulations by combining models from different modeling languages.

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. In parallel simulations, the required data synchronization for these calculations is performed transparently by the managers.

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 4.3.2.

CVODE

This interface uses the SUNDIALS library [73] 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.

Output

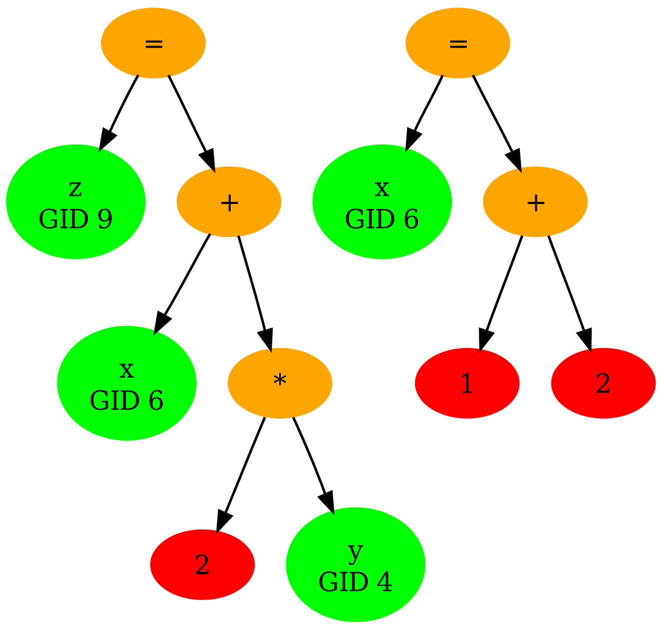
This interface writes values for objects on each processor to files. The output format can be specified by the user and includes the time, object value, and various types of object identifiers. In parallel simulations, each node writes its values to a separate file to avoid synchronization issues.

4.3 Optimization Techniques

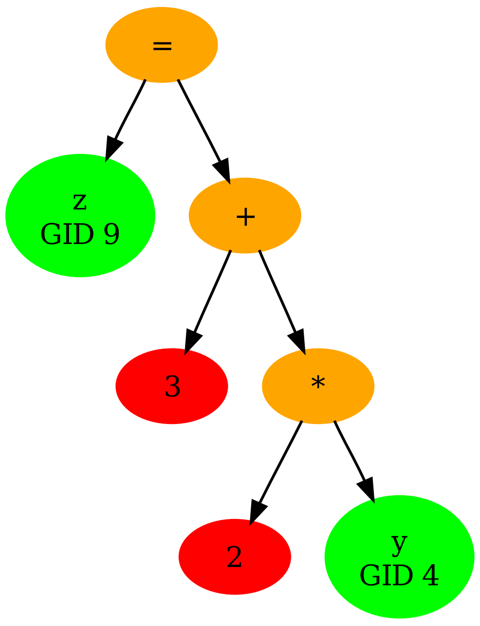
Next we describe two techniques used in the simulator to improve performance and functionality. Section 4.3.1 describes a method to improve simulation speed by compiling objects from their normal tree structure to a simplified byte code representation. In Section 4.3.2 we explain how data dependencies and parallel computation are transparently managed by using an object manager.

4.3.1 Compiled Objects

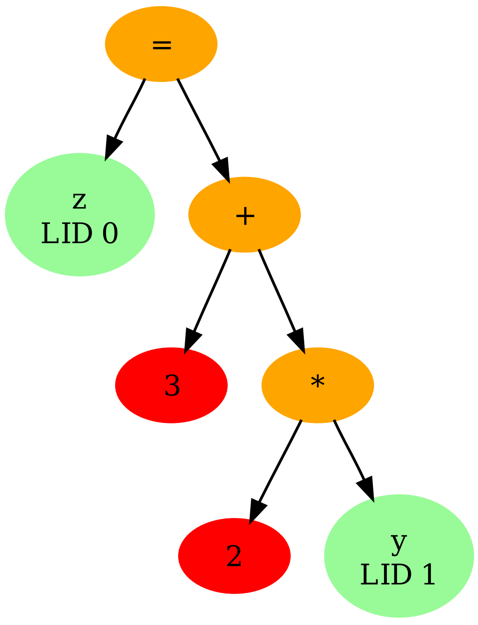
In this section we describe the technique of compiling objects to internal byte code in the core engine to improve simulation speed. Mathematical expressions in *insilico*Sim are represented by tree structures, with variables referenced by a global identifier (GID). The global identifier in turn references 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.



(a) Original expression trees



(b) Parameter substitution



(c) ID and value remapping

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

(d) Compilation to byte code

Figure 4.1: The three stages of simplifying and compiling an expression tree into byte code.

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 codes thereby ameliorating these three inefficiencies.

Compiling the expression trees is performed in three steps as shown in Figure 4.1. The colors indicate the different types of nodes in the tree structure. Red nodes represent constant values, orange nodes represent operators and green nodes represent variable references (light green for local ID references). The original expression trees are shown in Figure 4.1(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 4.1(b), replaces references to static parameters with the computed parameter value. Since $x = 1 + 2 = 3$ and will never change during the simulation, we can replace references to x with the value 3. 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 4.1(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 4.1(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 technique is fine for small scale calculations, but as we discovered in other work [74] 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 legal and technical problems, so we opted for a separate simulation engine.

4.3.2 DataObjSet Object Manager

A significant difficulty in performing these types of biophysical simulations in parallel is caused by object dependencies as described in Chapter 2. This is particularly true for chained function dependencies because they cannot be easily simplified to a single expression. In previous work [74] and in Chapter 3.2 we examined 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 4.2.2, each interface reports to the core engine what types of objects it will read/write. Based on this, the DataObjSet object manager creates a plan consisting of a set of ordered computation and communication steps that will ensure the correct simulation of the model in a parallel environment. These plans are created at the start of simulation and after each load rebalancing, and require relatively little time to generate.

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 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 and orders objects such that when the iterator references an object, all the dependencies needed for its evaluation have been resolved.

Algorithm 3 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: ObjsAvailForCommunication = [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 ObjsAvailForCommunication
 8:         Remove O from G
 9:     end while
10:     if any other processor has ghost objects in G then
11:         AllAvailObjs = Union of ObjsAvailForCommunication over all processors
12:         ObjsToGet = (ghost objects in G) $\cap$ AllAvailObjs
13:         for all O in ObjsToGet, O is on processor $P_i$ do
14:             Add "O $\leftarrow P_i$" to StepList of $P_j$
15:             Add "O $\rightarrow P_j$" to StepList of $P_i$
16:             Remove O from G
17:         end for
18:     end if
19: end while
20: return StepList
```

The pseudocode for the computation/communication ordering algorithm is shown in Algorithm 3. Orderings are created for each type of object combination requested by the interfaces. Object types may be mixed, for example, "function expressions and ordinary differential equations". 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 the processors have created plans for the 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 ObjsAvailForCommunication. The ObjsAvailForCommunication list is used to notify other processors which objects have been evaluated and are available for communication from this processor.

Next the algorithm creates communication steps for any processors that require them. This involves swapping lists 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. If an object has chained dependencies over multiple processors, this algorithm will generate several phases of communication as necessary.

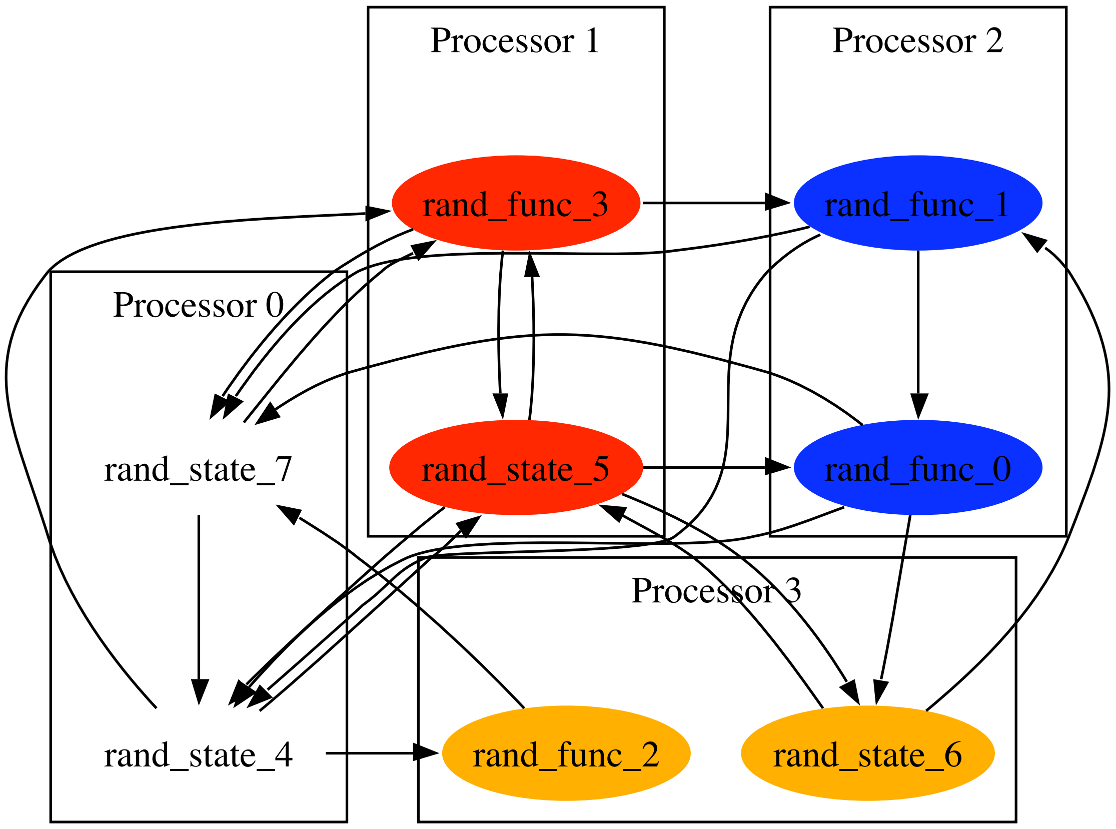


Figure 4.2: Random model graph from the Random and DOT export interfaces.

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 4.2, with different coloring for objects representing a division of the

Table 4.2: 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	$\frac{dS_7}{dt} = sin(F_3 - F_2 + F_1 + 2F_0)$

Table 4.3: 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			$Calc(F_0)$	

model over 4 processors. The corresponding expressions for the model are shown in Table 4.2.

Table 4.3 shows the steps to compute only the function expressions for the random model, as generated by Algorithm 3 for each processor. The communication steps have been separated for clarity - in the actual simulator 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_1 and F_0. 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 3 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.

4.4 Experiments

To confirm the effectiveness of the optimization techniques proposed in Section 4.3 and demonstrate the accuracy of the different calculation 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 4.4. The average

Table 4.4: Model characteristics.

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

dependencies represents the average number of references each object has to other objects.

The first model is the Luo-Rudy model described in Chapter 2 with an average of 2.58 dependencies per object. The scale of this model is along the lines of those commonly used in other simulators. Next we used a model of a spinal neural network for locomotor rhythm generation based on Rybak's central pattern generator [75]. 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 4.2. This model has relatively few dependencies per object, but as model size grows the number of dependencies will also grow significantly.

In this section, we discuss three types of experiments. First, we examine and compare the accuracy of each type of solver in Section 4.4.1. Next, we demonstrate the effectiveness of byte compiling objects in Section 4.4.2. In Section 4.4.3 we show that simulating models in parallel environments has good efficiency.

4.4.1 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 4.2.3 and 4.2.3 are used to obtain approximate solutions. If these solutions are significantly different, there is likely a problem with either the model or the simulator. This section also provides support to our claim that the simulator can easily support new types of calculation while smoothly integrating pre-existing code by using the interface concept.

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 4.3.2 and 4.2.2, no other changes were necessary to the code. 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} milliseconds and the solver interfaces each used a timestep of 10^{-2} milliseconds.

Figure 4.3 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 4.3(b), all the approximation methods end up with relatively high error, but the Euler method becomes inaccurate much sooner than RK4 or CVODE.

4.4.2 Compiled Objects

In Section 4.3.1 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.

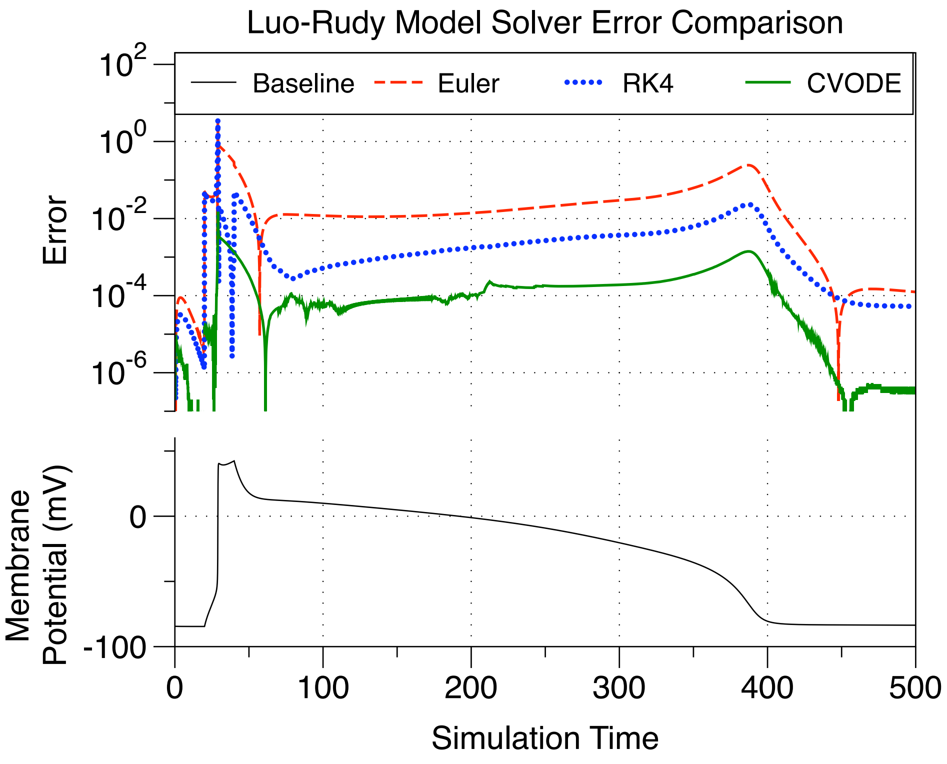
Figure 4.4 shows the simulation initialization time and time per simulation step for the Luo-Rudy and Rybak models using the Runge-Kutta interface. Figure 4.5 shows the number of cache references during simulation of the Luo-Rudy model. These figures show the effects on execution time and cache references after applying each step of compilation.

The figures first show the execution times and number of references for the original expression trees using abstract data values and retaining static parameters. Next, we run the simulations after substituting 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 4.5. 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.

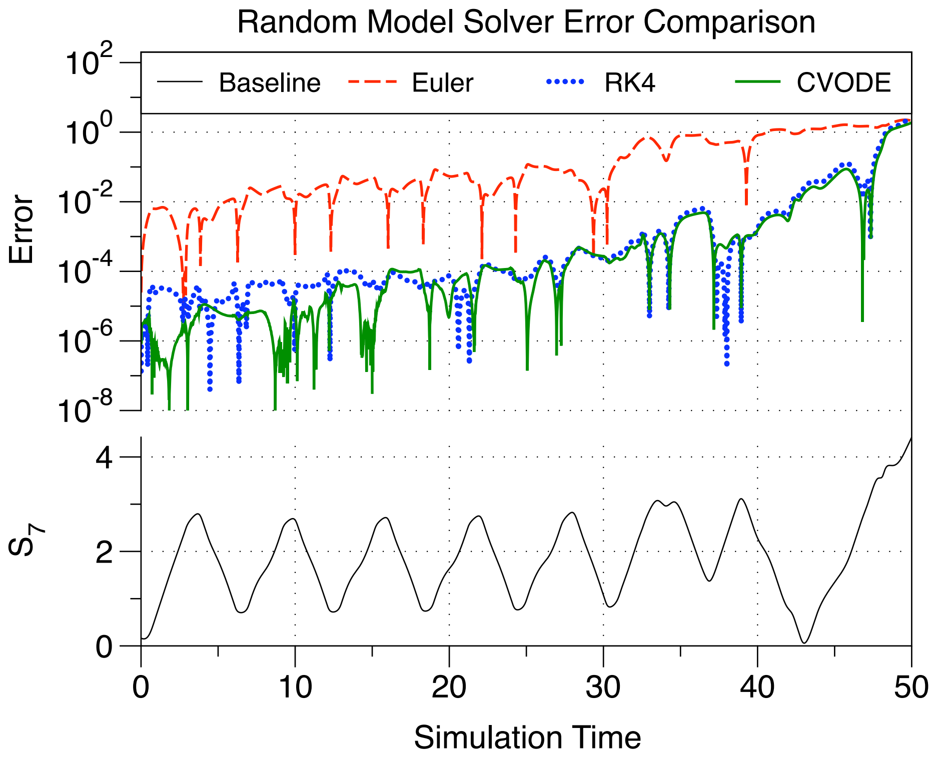
The second step of compiling objects changes the global identifier reference to a local one, and uses an array of doubles for object values rather than manipulating an abstract data value class. This step further decreases simulation time by 21.8% for the Luo-Rudy model and 23.4% for the Rybak model, 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 abstract data values is avoided. However, as seen in Figure 4.5 there are still a large number of cache references, likely 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 very little change in initialization time, showing that object compilation to interpreted byte code can produce fast simulations while requiring little compilation time.

To put this in perspective, we compared these results to execution times of a compiled simulation. The final

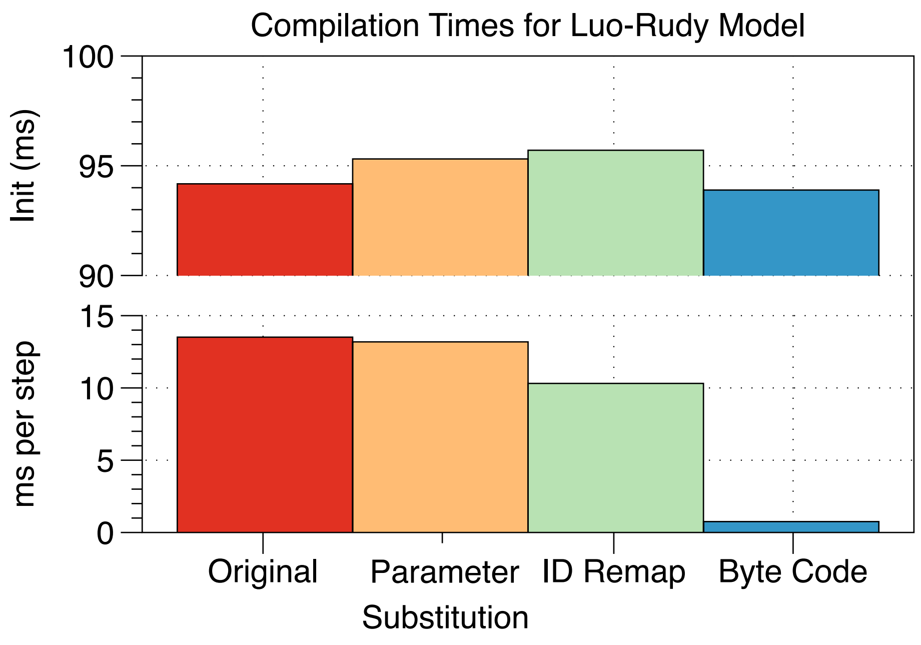


(a) Luo-Rudy model.

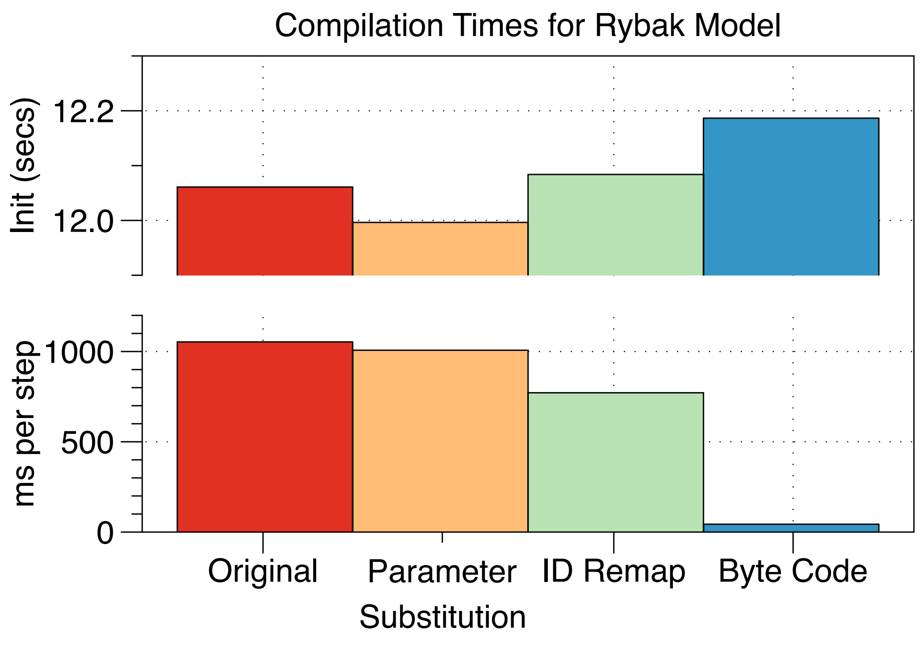


(b) Random model.

Figure 4.3: Solver accuracy comparisons.



(a) Luo-Rudy model.



(b) Rybak model.

Figure 4.4: Comparison of execution time for different levels of compilation.

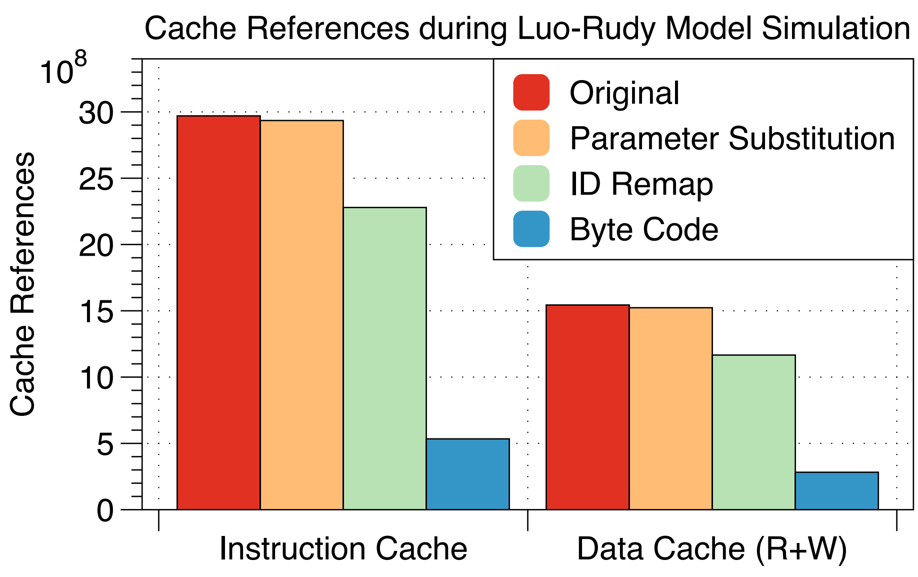


Figure 4.5: Comparison of cache references for different levels of compilation.

compilation step 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 performs 4 function/ODE evaluations per step, this gives an average of 4.83μs to evaluate an object in the LR model and 6.97μs to evaluate an object in the Rybak model. In Chapter 3 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 1.61μs. 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.

4.4.3 Parallel Computing 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 and random models are 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 4.6. 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 approximation are performed during every simulation step. The top part of the figure shows the speedup of the entire simulation, the non-initialization parts which are 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 is 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 calculation to decrease communication.

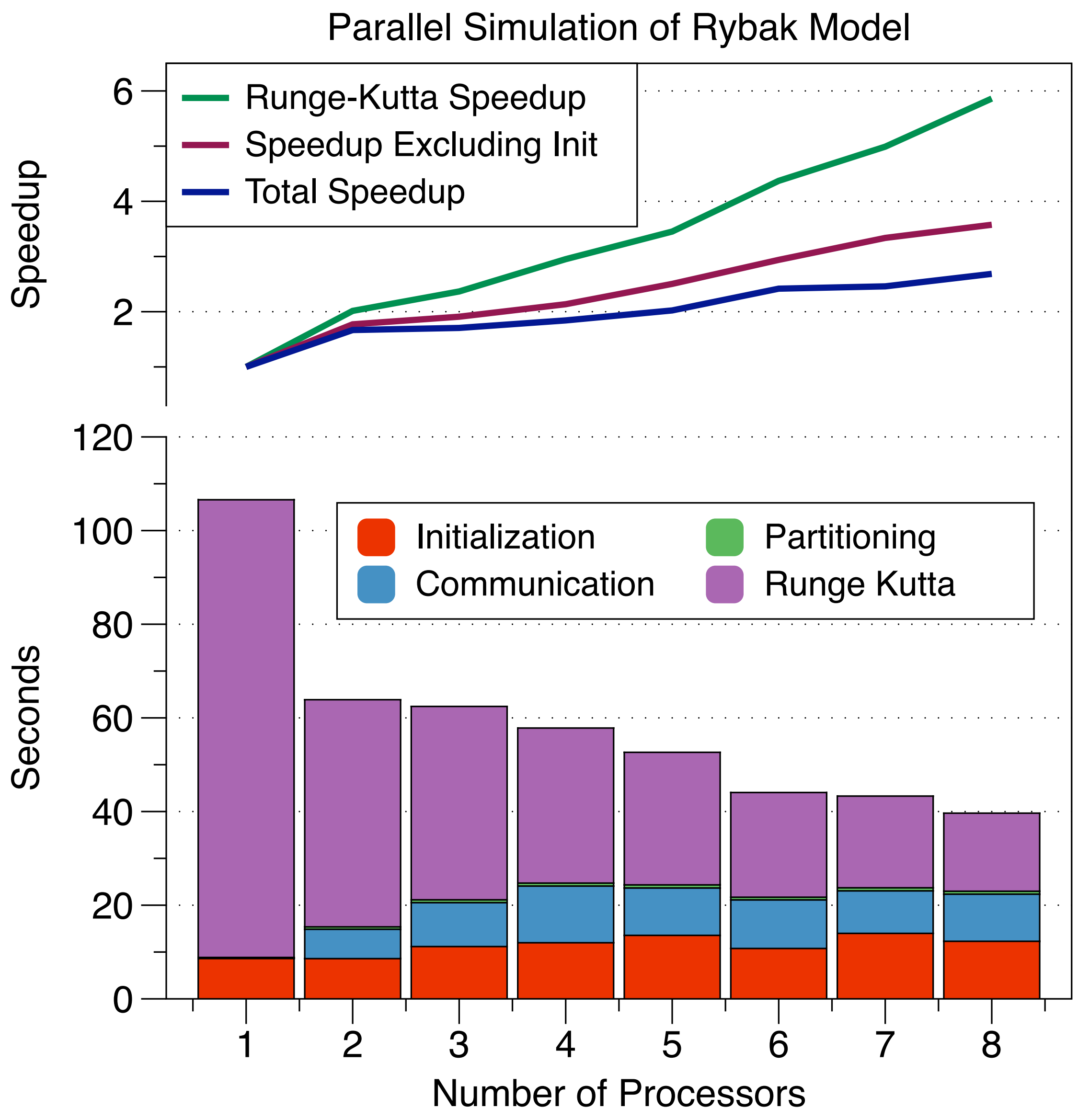


Figure 4.6: Parallel simulation of Rybak model.

4.5 Conclusion

In this chapter 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 chapter may be applicable to simulations to aid in this regard.

Chapter 5

Volunteer Computing Scale Simulation

5.1 Introduction

In recent years, public-resource computing or "volunteer computing" projects have demonstrated the power of performing distributed computation using donated resources over the Internet. Projects such as SETI@home [76] and Folding@home [6] sustain computation speeds of tens or hundreds of teraflops, comparable with high end supercomputers. In these projects, independent computational tasks are distributed and executed on donated computer resources around the world. These types of projects are often used to performing biophysical simulations, ranging from the molecular level protein folding simulations in Folding@home to the simulation of malaria disease transmission in Malariacontrol.net. The techniques described in this chapter are applicable to these kinds of simulations, as well as more general barrier synchronous computations.

Volunteer computing (VC) systems use a master-worker style of computing, where tasks are distributed from a master machine to worker machines and executed. Because these systems are composed of donated resources, they can make few guarantees about network or machine reliability. Therefore VC is usually applied to coarse grained embarrassingly parallel computation with tasks that require hours or days to complete. Task completion deadlines are generally on the order of days or months because of the volatile nature of the donated resources.

VC environments differ from traditional grid computing environments in several important ways. First, because the worker machines in VC systems are owned by private individuals, communication and computation reliability is significantly lower than in most Grid systems. A worker machine may often be disconnected from the network, used for other purposes or completely quit the computation without advanced warning. Second, worker machines are often behind firewalls and use NAT (network address translation) techniques which only allow one-way worker to master connections. This means that a "pull" model of task distribution must be used, instead of the common "push" model where the master distributes tasks to arbitrary workers. Finally, worker machines provide virtually no resource reservation or querying capabilities, thereby making task scheduling difficult.

In this chapter we propose algorithms for computing batches of tasks with deadlines in VC systems given varying types of worker reliability. Rather than normal VC deadlines of days or months, we deal with deadlines of minutes or hours. We call this “low latency computing.” Low latency computing in a VC system is appropriate for executing submitted batches of tasks with quick turnaround, or performing large scale barrier synchronous computations such as in the bulk synchronous parallel model [77]. Examples of such applications include molecular dynamics simulations with multiple trajectories, evolutionary based optimization algorithms with periodic swapping of solutions and any other problem with medium grained tasks and periodic barrier synchronizations. Applications such as Folding@home should have higher computing efficiency if they use a low latency style scheme. In addition, the guaranteeing of high performance for barrier computations will open these volunteered resources to a wider range of computations. For pull-style task distribution in a VC system, the key to meeting batch deadlines is ensuring that all tasks are distributed to workers in a timely manner and that workers complete the tasks before the deadline. Because of the nature of VC systems, we use techniques similar to those from stochastic scheduling [78] to handle worker unreliability.

Related studies examined task distribution in grid and VC environments [79, 80, 81], though some of these assume a task push model and are not valid for pull-style VC environments. However, some of these assume work may be distributed to arbitrary workers, which is not valid for pull style environments. Others describe methods to maximize total system throughput rather than meet specific task deadlines. There are also several works [82, 83, 84, 85, 86, 87, 88] analyzing the characteristics of VC and desktop grid environments which are applicable to high throughput computing but do not target low latency style VC. To the best of our knowledge, this dissertation is the first to investigate methods for computing low latency batches in a pull-style VC environment.

There is work similar to our study in regards to completing batches of tasks with deadlines [89], though this focused on desktop grid environments rather than volunteer computing. This study viewed the system as a buffer with batches of tasks periodically entering and expiring from the buffer. The authors analyzed the appropriate buffer size to ensure maximum task completion rates in a desktop grid environment, where tasks can be assigned to arbitrary workers. Future work in low latency batch computing could use this type of buffer, especially for computations performing multiple simultaneous batches.

One technique for handling unreliable workers is the use of checkpointing or “heartbeats” to the master server [90, 91]. In this technique, workers periodically report and/or save their progress to the master server. This allows for duplication of tasks which are unlikely to meet the deadline. However, we believe that this would not be useful in low latency because of the short task computation time. For short running tasks, intelligent scheduling will provide better results than checkpointing.

Other research related to VC scheduling includes using evolutionary algorithms to develop scheduling algorithms [92] and using P2P to perform load balancing in VC systems [93]. A possible future field of research is to use genetic algorithms in predicting worker availability for low latency batch computing, though other methods [84, 87] appear to already be very successful and more applicable to low latency computing.

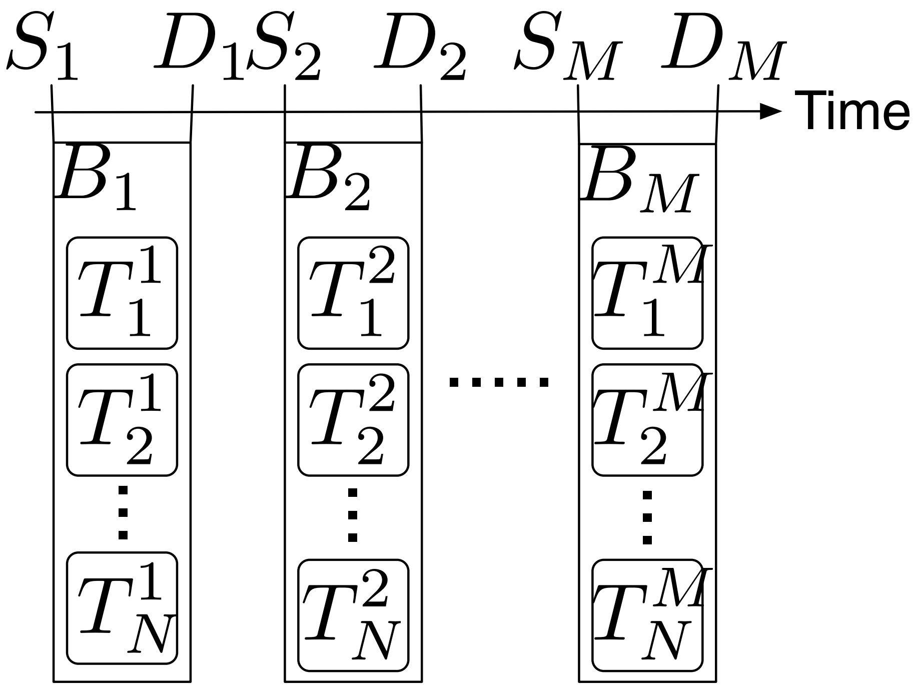
To develop suitable algorithms for low latency VC, we first define the environment and worker characteristics in Section 5.2. Using trace data from an actual VC environment, we show how worker task requests can be modeled as a Poisson process and task computation time can be predicted from past worker behavior in Section 5.3. From these models, we develop algorithms for task distribution in Section 5.4. The algorithms are verified using trace-driven simulations in Section 5.5.

5.2 Computation Model

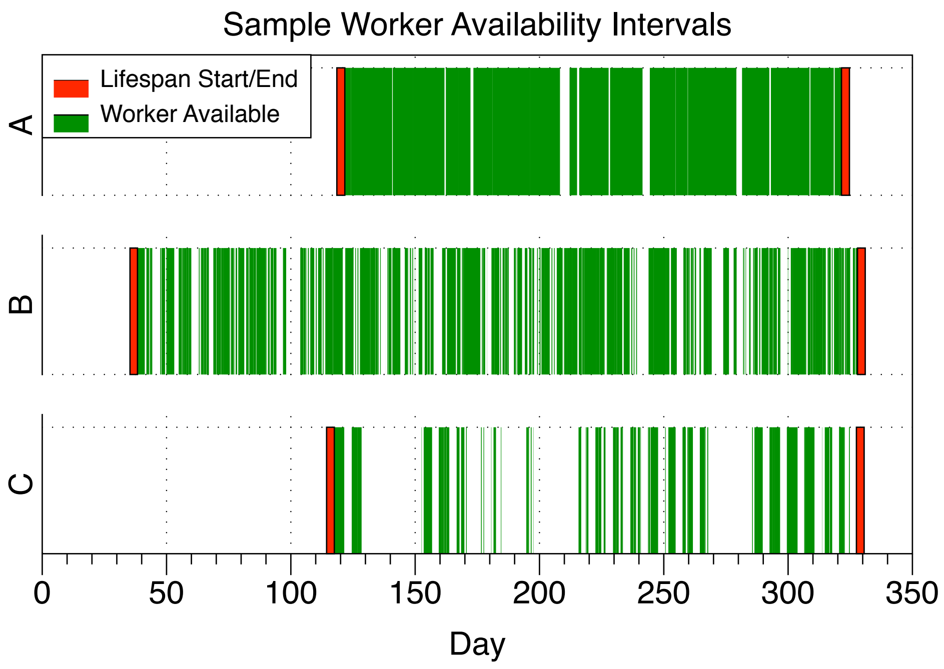
In this model for VC low latency batch computing, there are M batches of work, denoted $B_1, \ldots, B_M$. Each batch B_i has N independent tasks of equal computational size denoted $T_1^i, \ldots, T_N^i$, a submission time S_i and a deadline D_i with $S_i < D_i$. All tasks in batch B_i are available for distribution at time S_i. Batches are sequential and do not overlap, i.e. $\forall i, n, (i < n) \Rightarrow D_i < S_n$. Figure 5.1(a) graphically shows the model of tasks and batches used in this dissertation.

All tasks are initially on a single master server, which tries to assign tasks to workers so as to minimize the number of tasks completed after their deadline. A task that is completed before the batch deadline is called a satisfied task, and a batch whose tasks are all satisfied is called a satisfied batch. Note that batch submission and deadline times need not be predetermined, meaning that creation of the tasks in B_i can depend on completion of the tasks from B_{i-1}. In fact, due to the unpredictable nature of VC systems it is often difficult to predict S_i and D_i for batches far in the future.

To compute the tasks of a batch, there are P workers $W_1, \ldots, W_P$. In standard VC environments, connections may only be made from a worker to the master, not vice-versa. A connection is made when the worker initially becomes active, and at specified times afterwards called “reconnection times”. At any given time, a worker is in one of two states - available or unavailable. The master is always available. Previous studies [82] [83] have shown this to be a good approximation of actual systems, rather than measuring availability as a fraction of available CPU power. A period where a worker is in an uninterrupted available state is called an availability interval, and a period where a worker is in an uninterrupted unavailable state is called an unavailability interval. The lifespan of a worker is defined as the time between the start of the earliest availability interval and the end of the latest availability interval. Average availability is the fraction of the worker lifespan spent in the available state. Figure 5.1(b) shows availability intervals for three workers with different average availability (A: 95%, B: 65% and C: 33%). Black sections indicate the worker is available and white sections indicate the worker is unavailable. The red bars show the beginning and end of the worker lifespan. As seen in this figure, worker availability may be erratic and difficult to predict.



(a) Batch diagram.



(b) Availability intervals and worker lifespan.

Figure 5.1: Graphic depiction of tasks and batches in the low latency computing model, and three example workers with their availability intervals.

A worker in the available state may perform computation or communication, an unavailable worker may do neither. Various factors may cause transition between these states - user activity/idleness, machine reboot/shutdown, machine/network failure, etc. If a worker transitions from available to unavailable while executing a task, the task is resumed at the same point when the worker returns to the available state. This behavior can be achieved through task checkpointing. Each worker W_i also has a task computation time C_i, which is the number of seconds the worker requires in the available state to complete a task. This can be thought of as the inverse of the computation speed.

In this dissertation we treat C_i as a deterministic variable, though in real systems it might be more accurately treated as a probability distribution depending on the worker and task characteristics. For simplicity, we assume C_i is constant and that non-deterministic effects, such as swapping due to insufficient memory, can be avoided by using standard worker selection techniques. In addition, malicious or malfunctioning workers occurring in VC systems are not explicitly considered in this dissertation. These can be handled by computing redundant tasks or using other techniques such as ringers and magic numbers [94] or worker reputation measurement [95], though the usage of these is outside the scope of this discussion.

In terms of communication and computation, workers can be termed either reliable or semi-reliable. A worker is "communication-reliable" if it can guarantee communication with the master at an arbitrary time R. A worker is "semi-communication-reliable" if it cannot guarantee communication at time R, but behaves like a standard VC worker. A worker W_i is "computation-reliable" if it can guarantee task completion within the task computation time of the worker (C_i). A worker is "semi-computation-reliable" if it cannot guarantee task completion within C_i, but behaves like a standard VC worker. Communication-reliable and computation-reliable workers are only theoretical constructs, but we use them as a basis to develop algorithms in later sections.

5.3 Analysis of Volunteer Computing Workers

In this section we examine workers from an actual VC environment and create models of them based on analysis. In Section 5.3.1 we describe the experimental data taken from a VC system used to model the workers. Analysis of worker availability is performed in Section 5.3.2. In Section 5.3.3 we examine communication reliability in workers and demonstrate that connections from VC workers can be modeled using a Poisson process. In Section 5.3.4 we examine computation reliability in workers, and derive a model for predicting worker computation reliability based on prior worker availability.

5.3.1 Worker Trace Data

To perform the analysis and experiments in this dissertation, we used a set of worker availability trace data. This trace data was measured using the Berkeley Open Infrastructure for Network Computing (BOINC) [96]. The

BOINC middleware is used to perform large scale VC over the Internet. The BOINC client software currently runs on over 1 million computers throughout the world. The BOINC client was augmented to record the start and stop times of CPU availability on each worker machine. In BOINC, CPU availability is determined by user preferences and whether the machine is running. For example, some users only allow BOINC to run when the computer is idle or only on weekends, while other users shut down the machine every night.

Volunteer participants downloaded this BOINC client, which recorded the CPU availability intervals and reported the intervals back to the main server. A total of 112,268 worker machines ran the client during the period between April 1, 2007 to February 12, 2008, though none of the worker lifespans covered the entire period. In total, the modified client recorded 16,293 years worth of CPU availability. Among the worker operating systems, 66% ran Windows XP, 12% ran Windows Vista, 9% ran Mac OS X, 7% ran a variant of UNIX/Linux, and the remaining 6% ran a variant of Windows. Further analysis of the trace data is available in [84].

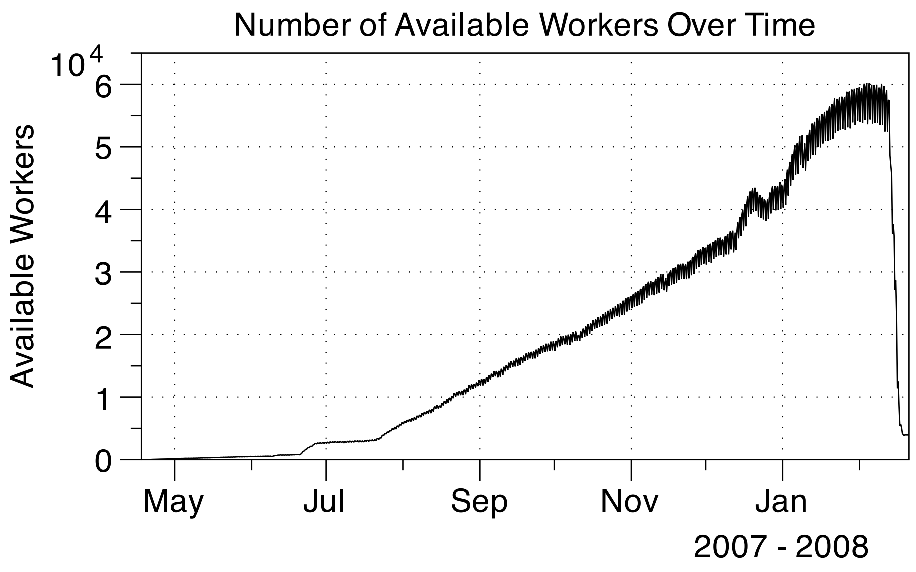
5.3.2 Analyzing Worker Availability

To learn more about the typical worker machine in a VC system, we performed several analyses of the trace data in regards to worker availability. These analyses were performed to confirm the validity of the trace data compared with previous studies, and to examine the characteristics of worker availability.

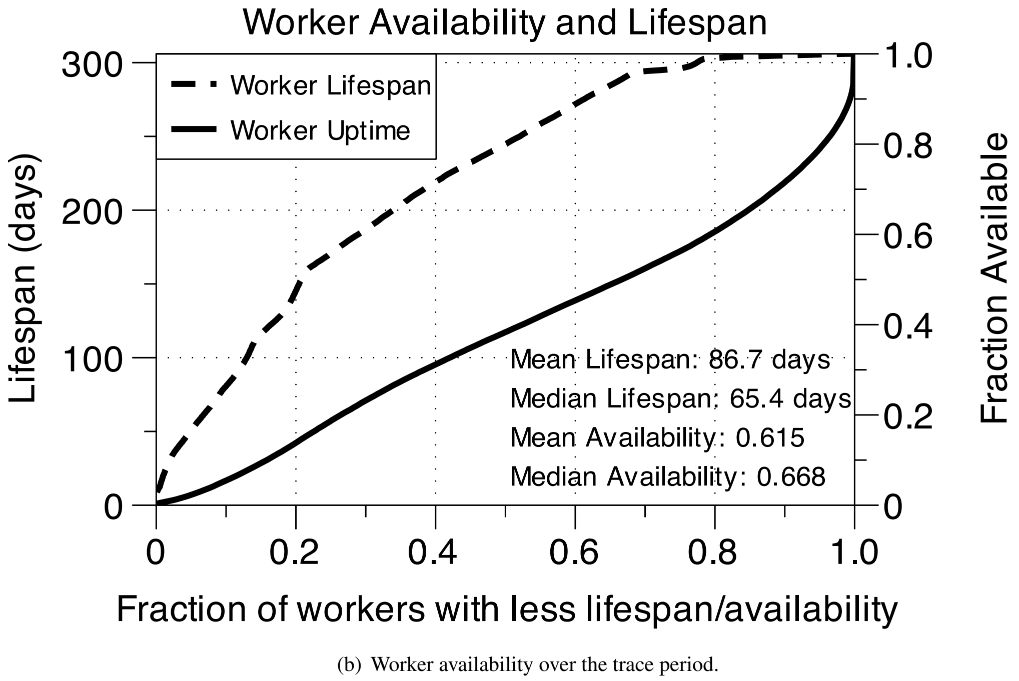
Figure 5.2(a) shows the number of available workers over the trace recording period. This was obtained by measuring the number of workers in the available state every 8 hours over the entire trace period. The oscillation of available workers is caused by computers being shut down at night and on the weekends. Although this oscillation grows over time, the oscillation relative to the total number of workers stays relatively constant and does not affect the algorithms described here.

Figure 5.2(b) shows cumulative distribution plots of worker lifespan and worker availability. The point (x,y) on the dotted line means that x fraction of workers have lifespans less than y, and on the solid line means that x fraction of workers spend less than y of their lifespan in the available state. From this graph we see that only a small fraction of workers (roughly 5%) are available over 80% of their lifespan and that half of the workers are available for less than 40% of their lifespan. This environment of high unreliability and downtime makes traditional scheduling algorithms difficult to use.

In this dissertation we assume that worker unavailablity is transient rather than permanent, in other words workers do not become permanently unavailable. For actual systems this is not a correct assumption due to machine failures, volunteers ending their participation, etc. An analysis of VC projects indicates worker lifespan roughly follows an exponential distribution with a mean of 3 months [97]. As seen in Figure 5.2(b), the mean and median worker lifespan are on the order of 2 to 3 months. This means that for low latency tasks less than 15 hours in length, fewer than 1% of tasks will fail due to permanent worker unavailability. Therefore, we can simplify our



(a) Number of available workers.



(b) Worker availability over the trace period.

Figure 5.2: Worker availability and lifespan characteristics.

Table 5.1: Task request simulation parameters.

Parameter	Values
Number of workers (P_{total})	11320
Reconnection time (T)	5m, 15m, 1h, 2h, 4h, 12h, 1d, 2d, 4d, 10d
Interval lengths (L)	1m, 5m, 15m, 1h, 2h, 4h
Number of intervals (M)	200
Test period start times (S_0)	Sun, 01 Jul 2007, Wed, 01 Aug 2007, Mon, 01 Oct 2007, Sat, 01 Sep 2007, Thu, 01 Nov 2007, Sat, 01 Dec 2007, Tue, 01 Jan 2008

model by ignoring permanent worker unavailability without significant effect.

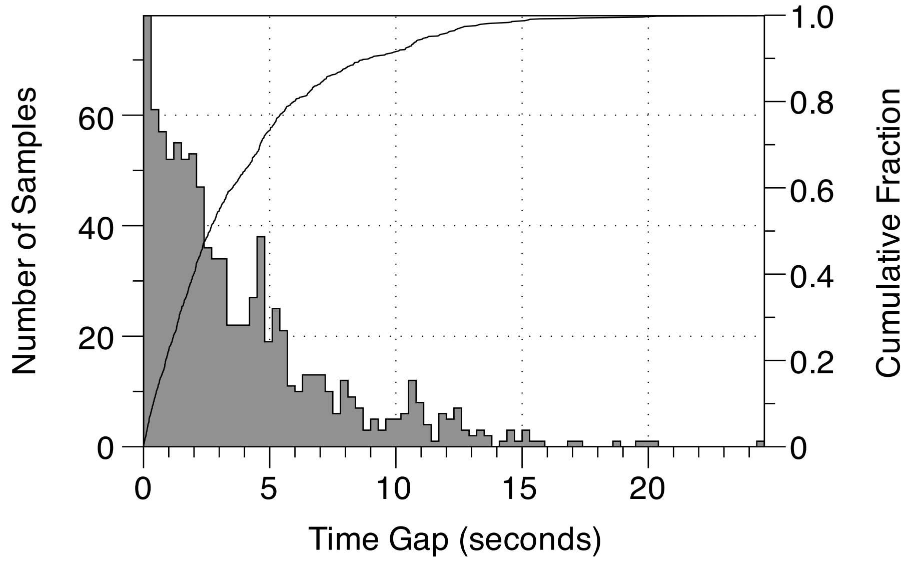
To create algorithms for performing low latency computation in a VC environment, it is crucial to understand and model the behavior of workers in terms of availability and unavailability. Many studies have investigated worker availability in desktop grids [82, 83, 85, 86] and in VC type environments [84, 87, 98]. In the following two sections, we develop models of VC workers partly based on our own analysis and partly based on the work in these studies.

5.3.3 Modeling VC Worker Communication

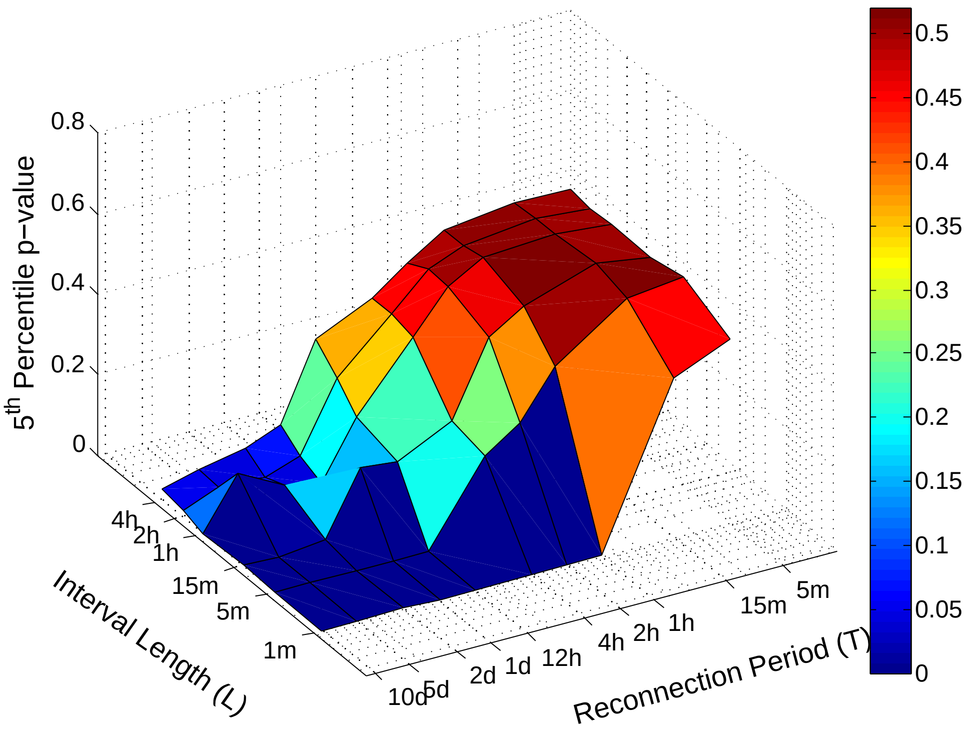
In this section we examine the effect of worker unavailability on task requests and propose a model for task requests from VC workers. To develop a model of task requests from VC workers, we perform simulations using worker trace data. The simulation results indicate that task requests from VC workers can be modeled as a Poisson process. Furthermore, the task request rate of this process can be controlled through the worker reconnection period T. This means that rather than scheduling communication for individual workers, we treat the entire worker pool as a tunable stream.

To perform the simulations, we used a subset of 11,320 randomly selected workers from the entire trace data set. We performed a wide range of simulations to test multiple reconnection periods, worker pools and stretches of time. In a simulation, the workers transition between available and unavailable based on the trace data. Each worker W_j connects to the master at the start of their lifespan. After each connection, the worker is assigned a reconnection time $R_j = CurTime() + T$. If a worker is unavailable at time R_j, the connection is initiated when the worker next becomes available. Because the number of active workers changes over the course of the trace data, we define the number of recently active workers (P_{active}) as the number of workers which have connected in the last $2T$ seconds.

To analyze the results of the simulations, we recorded each time a worker connected to the master. These times were recorded during intervals $[S_0 + XL, S_0 + (X + 1)L]$ for $X \in [0, M - 2]$. This effectively emulates the task request behavior of the workers performing M batches. The parameters for the simulations are shown in Table 5.1.



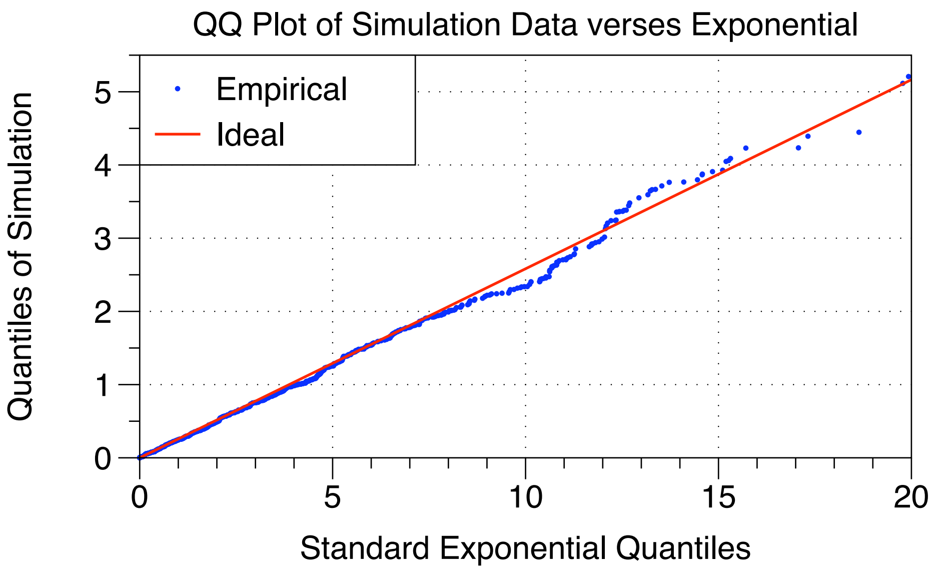
(a) Time Gaps Between Connections to the Master.



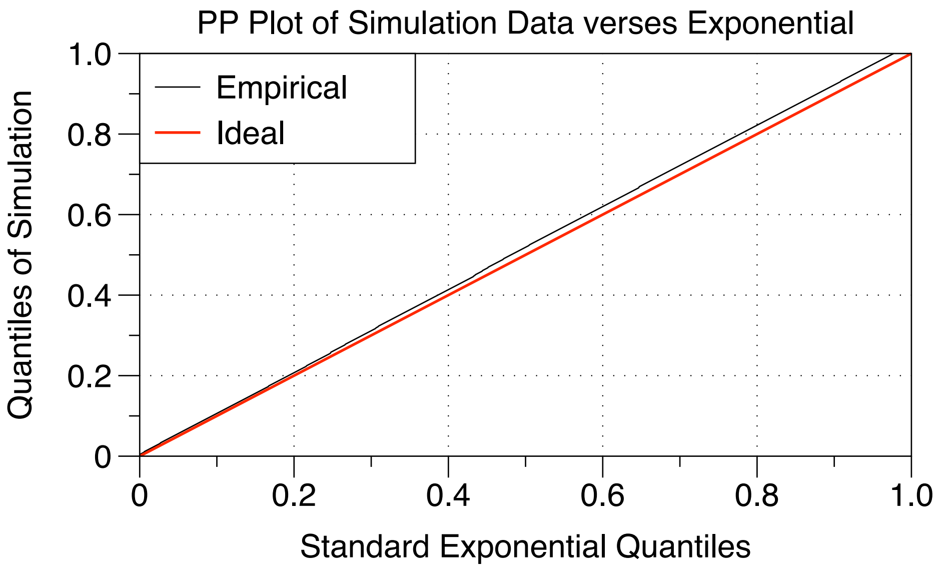
(b) 5th percentile p-vals from Kolmogorov-Smirnov test of EDF fit.

Figure 5.3: Simulation results.

The results of one simulation are shown in Figure 5.3(a) with the time gaps between consecutive connections to the master plotted as a histogram and cumulative fraction. The parameters of this simulation are $T = 4$ hours, $L = 1$ hour, $S = 08$ Dec 2007, and at the start of the simulation $P_{active} = 4024$. Note that this represents connections from all workers, not from a single worker. For a reconnection time T with P_{active} active workers, the average expected rate of connection is one connection every T/P_{active} seconds. In Figure 5.3(a), the average time gap between connections of 3.76 seconds closely matches the expected time gap of 4 hours$/4024 = 3.58$ seconds. The cumulative distribution of time gaps in this simulation and others appears similar in shape to that of an exponential distribution function.



(a) QQ Plot.



(b) PP Plot.

Figure 5.4: QQ-PP plots.

Figure 5.4 shows QQ/PP plots for the simulation shown in Figure 5.3. As seen, there is a close fit between the simulation results and an exponential distribution. There is a slight deviation for larger values as seen in the QQ plot of Figure 5.4(a). The PP plot in Figure 5.4(b) also shows a good fit. Based on these results we offer Hypothesis 1.

Hypothesis 1 *Let T be a positive time period. P workers with availability characteristics common to VC system workers are assigned reconnection times (T) as described above. Then the time gap between connections to the master can be modeled as an exponential distribution function (EDF).*

To confirm Hypothesis 1, we apply the Kolmogorov-Smirnov (KS) test [99] to all simulation results. This allows comparison of the simulation results with EDFs using the parameter based on the expected time between connections $\lambda = P/T$. The KS test is sensitive to imperfections in large data sets, so each KS p-value is based on the average from 100 random samplings each of 100 data points from each data set. Generally, the minimum acceptable p-value for the KS test is 0.05, so p-values higher than this indicate good fit with the EDF and therefore the validity of Hypothesis 1.

Figure 5.3(b) shows the results of applying the KS test to the simulation results. In this figure, the x and y axes represent the reconnection time T and interval length L. The z axis represents the 5th percentile p-value over all simulations for a given T and L (i.e. 95% of simulations had a higher p-value than indicated in the graph). In other words, the graph shows the validity of Hypothesis 1 for a range of T and L. For example, with T = 4 hours and L = 2 hours, 95% of simulation results had a p-value greater than 0.22.

In this figure, the accuracy of modeling worker connection time gaps as an EDF varies depending on the ratio of T to L. If T is much greater than L, then few workers will connect in a given interval and the KS test will show a poor fit. For example, with T = 4 hours, L = 5 minutes and P = 1000, an average interval will have only 4.2 connections even with full worker availability. For a lower ratio of T to L, worker connections fit an EDF very well.

A Poisson process [100] is defined as a process with time between events following an EDF. Since Hypothesis 1 showed that time gaps between worker connections follow an EDF, we offer Corollary 1.

Corollary 1 *Let T be a positive time period. Then connections from P volunteer computing workers can be modeled as a Poisson process with rate parameter $\lambda = P/T$.*

Fundamentally, this means that we can handle worker communication unreliability by treating the group of all workers as a Poisson process with a tunable connection rate parameter. Based on the results shown in Figure 5.3, we maintain that Corollary 1 is correct for short reconnection periods ($\leq$ 12 hours) with long interval lengths ($\geq$ 5 minutes). These are well within the range expected in low latency batch volunteer computing. With greater numbers of workers, Corollary 1 may apply for longer T and shorter L. The idea of a tunable Poisson process is used in Section 5.4 to develop algorithms for maintaining a continuous stream of workers for computing low latency batches.

5.3.4 Modeling VC Worker Computation

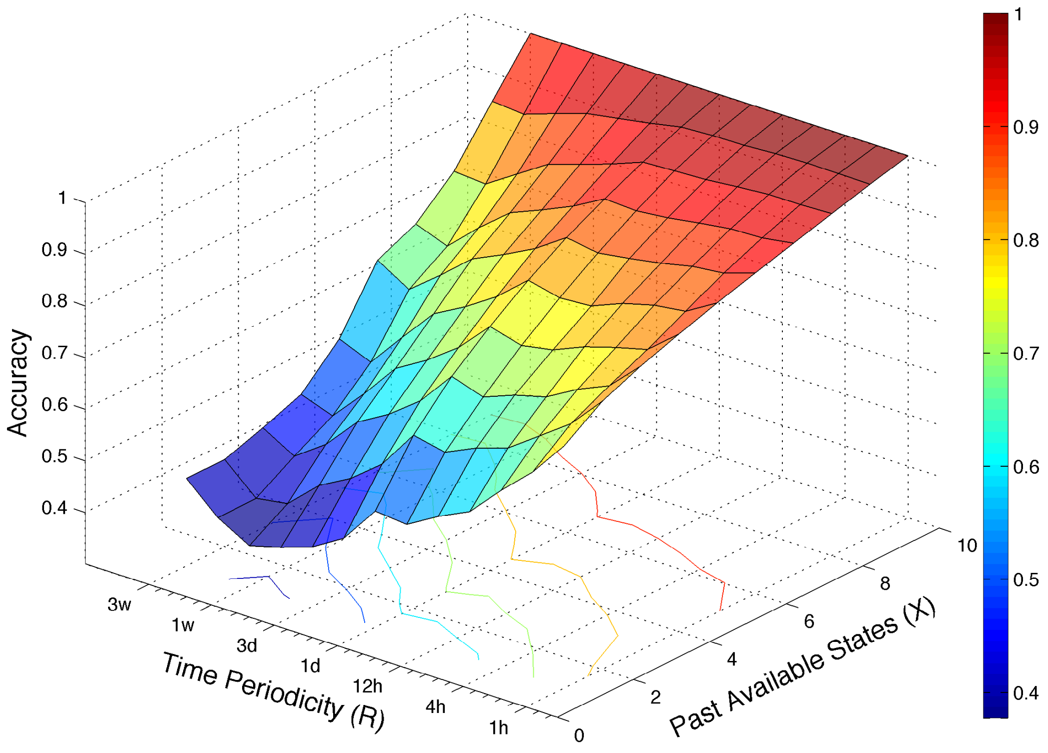
Here we propose a model of worker computation reliability based on worker availability prediction. Previous studies [84] [87] demonstrated techniques to ensure future worker availability based on correlating worker behavior, Bayesian classifiers and other techniques. These techniques provided means of organizing workers into groups based on training data for performing simultaneous computation. In this study, we use a simpler method of recent worker availability, which is more suited for low latency batches where workers need not be simultaneously active.

Although the other methods could possibly be used for low latency batch computing, we chose not to for a couple reasons. First, the other two methods discard roughly half of the worker population and require weeks of training data, which can discourage many VC projects from using them. Also, by using only highly available and predictable workers, the algorithms can cause cyclical unavailability in workers which share CPU time between projects in a round-robin fashion. Therefore, in this dissertation we focus on the technique described below.

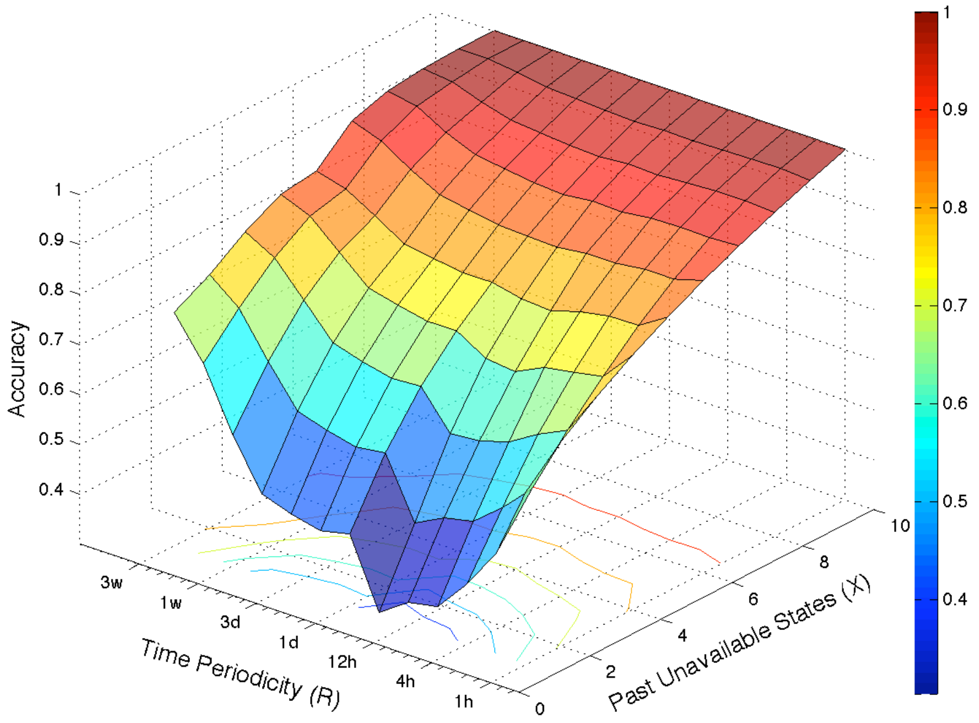
To examine worker predictability, we performed 1 million simulations using the trace data subset described earlier. The goal of these simulations is to determine how well a workers recent past state can predict the future availability. Our hypothesis is that worker availability/unavailability can be predicted based on periodic worker behavior. Each simulation involves randomly selecting a worker with a chance proportional to its lifespan. The availability state of the worker is examined at a randomly selected time R in the workers lifespan. Next, for a range of time lengths T we examine worker availability at times $R - TX$ for $X \in [1, 10]$. The same analysis was performed for worker unavailability prediction based on previous unavailable states. The results of these analyses are shown in Figure 5.5.

Figure 5.5(a) shows the accuracy of using the recent past availability to predict current availability and Figure 5.5(b) shows the same for unavailability. For a given time period T and number of recent (un)available states X, these graphs show how accurate a prediction of (un)available is. For almost all of the simulations, (un)availability at all recent time periods is a strong indicator that the worker will be (un)available at time R. For example, in Figure 5.5(a), there is more than 90% accuracy in predicting a worker to be available when it has been available for the last 10 periods. This accuracy stays high when the time period T is 1 hour, but the accuracy falls to less than 50% as the time period increases.

In both graphs, shorter time periods yield better accuracy for predicting worker state. Figure 5.5(a) shows that using older states gives a poor prediction of current worker availability. An interesting feature in this graph is the jump in accuracy for time periods of 1 day, implying that daily usage patterns exist for workers. Worker unavailability is also well predicted with time periods of 1 week and 3 weeks, which would capture night and



(a) Worker availability prediction results.



(b) Worker unavailability prediction results.

Figure 5.5: Simulation results for worker availability and unavailability prediction.

weekend downtime. We use these results in Section 5.4.3 to predict whether a worker will finish an assigned task before a deadline.

To confirm these results, we performed 100,000 simulations to examine the accuracy of using past worker availability to predict task completion success. In these experiments, we randomly select a worker and a time R when the worker is active. We simulate the worker receiving 10 tasks at times $R - TX$ for $X \in [1, 10]$ with each task having deadline $R - TX + 2C$ (twice the task computation time). We then check whether the worker completes the task by the deadline and how well the past task completion rates correlate with predicting the current task completion probability.

Figure 5.6 shows the results of the simulations. The top graph shows the accuracy of the past simulation results in predicting task completion in the present. In this graph we show the accuracy of predicting deadline satisfaction given the number of tasks which met their past deadlines. The lower axis shows the minimum number of past simulations that must have satisfied the deadline (out of 10) in order for a current prediction of "Will Meet Deadline". As seen, the accuracy of this method increases as the bounds on past failures get tighter.

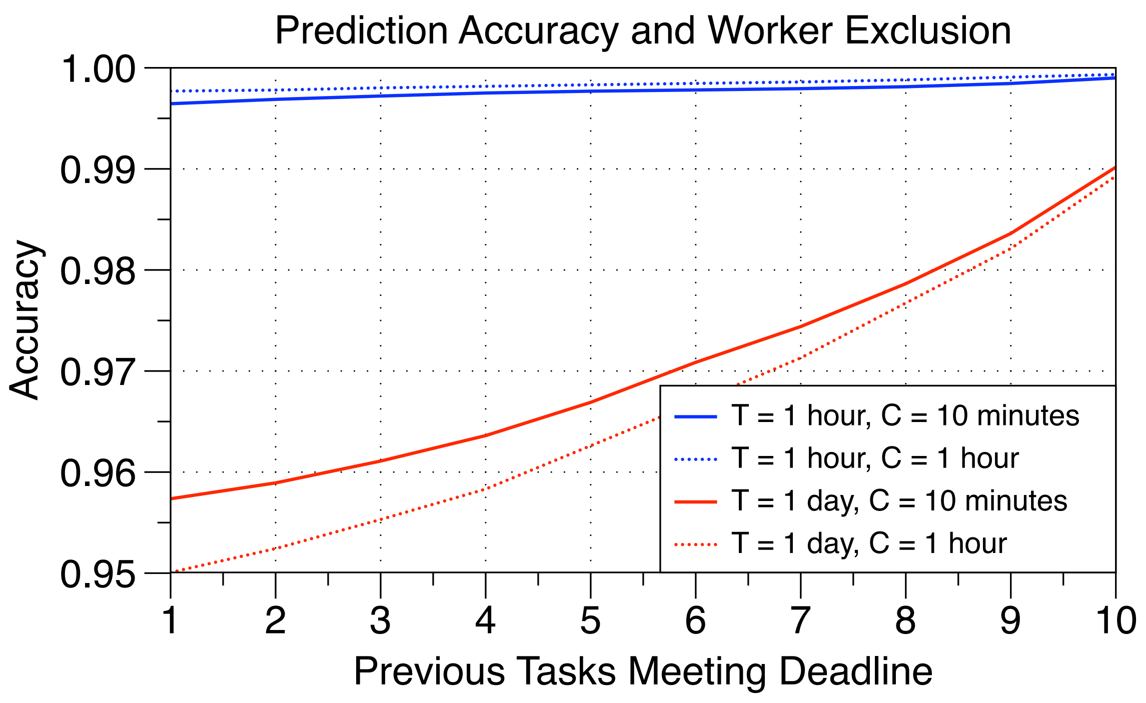
The bottom graph shows the tradeoff of tightening the bounds for prediction. As the bounds get tighter, the fraction of failure predictions and thus the number of denied work requests grows. This is equivalent to a worker connecting to the master but not receiving a task due to poor past performance. If this bound is too tight, not enough workers will be deemed reliable and the computation will fail to meet the deadline.

5.4 Task Distribution Algorithms

In order to meet batch deadlines, tasks must be distributed to workers in a timely manner. Pull-style VC task requests go from workers to the master. Thus, a sufficient number of task requests must occur between the batch submission and deadline. In a VC system, this is done by requesting workers to connect at certain times. This is known as the "reconnection time" and is denoted R_j for worker W_j.

A simple method for performing low latency batches is to estimate the submission time S_i of each batch B_i and request a number of workers to connect soon after that. However, even for synchronous batch computing, due to the unreliable nature of VC workers it is nearly impossible to predict when a given batch will complete and the next batch will begin. This is particularly true when a worker fails to complete a task and delays the generation of the next batch. For this reason, we assume that submission times cannot be known accurately far in advance.

In this section, we describe algorithms for ensuring a high probability of sufficient task requests to complete all batches before their deadlines. We start by developing an algorithm for fully reliable workers, then modify it to handle communication and computation unreliability. Section 5.4.1 presents an algorithm for fully reliable (communication-reliable and computation-reliable) workers, and proves that it satisfies all deadlines in certain conditions. In Section 5.4.2 we provide an algorithm for semi-communication-reliable workers with a probabilistic



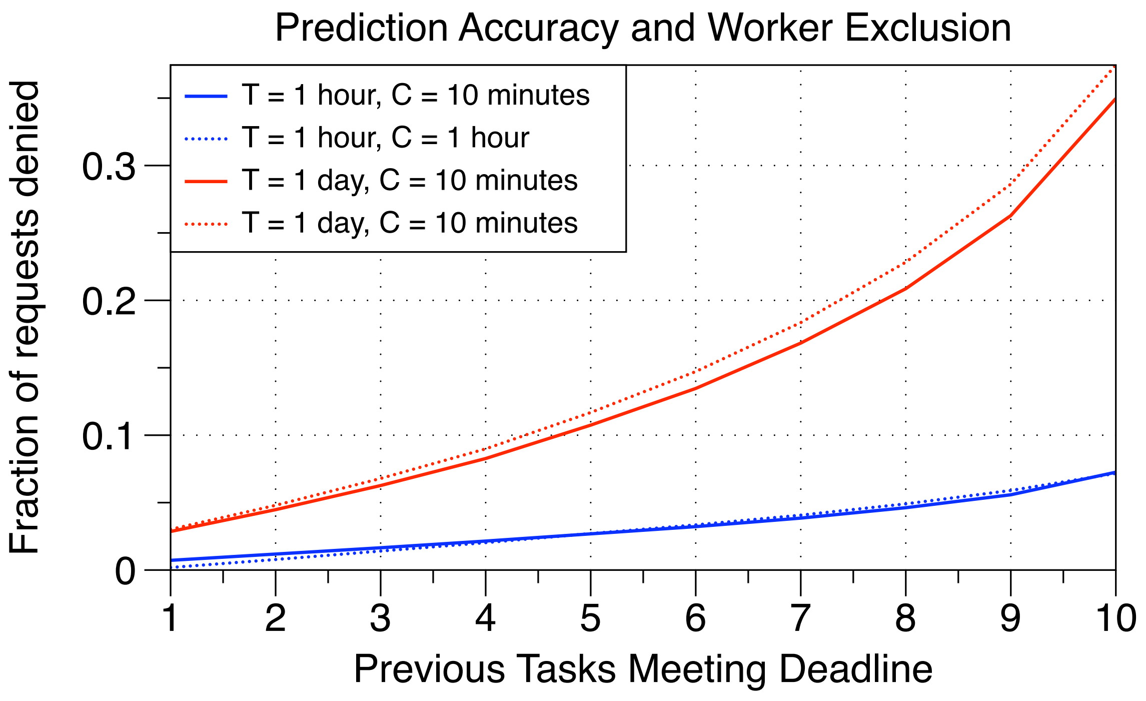


Figure 5.6: Results of simulations for validating worker performance prediction.

bound on failure. The algorithm for semi-reliable (semi-communication-reliable and semi-computation-reliable) workers is given in Section 5.4.3, and also provides a probabilistic bound on failure. The effectiveness of these algorithms is demonstrated by simulation in Section 5.5.

5.4.1 Fully Reliable Homogeneous Workers

In this section we consider fully reliable workers with guaranteed communication and computation times. Using fully reliable workers we develop an algorithm for task distribution that provides a basis for later algorithms with unreliable workers. In this section, workers are considered computationally homogeneous and reliable, meaning that a task always takes C seconds and finishes at time $R_j + C$. To meet the deadline D_i, all tasks in B_i must be distributed to workers before the distribution deadline $L_i = D_i - C$. We assume that all workers connect before the start time of the initial batch, S_1.

Algorithm 4 shows the algorithm for the master when dealing with fully reliable workers. The algorithm works by pre-assigning a task to a worker using the variables $AssignBatch$ and $AssignTask$. Worker W_j connects at time R_j or immediately if the current time is past R_j. Upon connection the worker receives a task (line 5) and next connection time, then executes the task. The master gives workers reconnection times that ensure the workers will receive tasks at a time necessary to meet the batch deadline. Because all workers are homogeneous and reliable, task assignment is in a simple round-robin fashion and the algorithm needs only ensure that N task requests occur during each batch. It is worth noting that the calculation of R_j (line 12) does not simply assign the batch submission time S_i to a worker. Instead, the algorithm spreads worker connections over the entire batch. This prevents the master from being overwhelmed by connections, and is also necessary in later algorithms for semi-reliable workers. Given this algorithm and constraints on the task length, deadline/submission times, and number of tasks/workers, Theorem 1 proves that all batches will meet their deadlines.

Algorithm 4 Fully Reliable Homogeneous Workers

```
1: $AssignBatch \leftarrow 1, AssignTask \leftarrow 1, SendBatch \leftarrow 1, SendTask \leftarrow 1$
2: while $SendBatch < M$ do
3:     Get connection from $W_j$
4:     if $CurrentTime() \geq S_{SendBatch}$ then
5:         Send task $T_{SendTask}^{SendBatch}$ to $W_j$
6:         $SendTask \leftarrow SendTask + 1$
7:     end if
8:     if $SendTask > N$ then
9:         $SendTask \leftarrow 1, SendBatch \leftarrow SendBatch + 1$
10:    end if
11:    if $AssignBatch \leq M$ then
12:        $R_j \leftarrow S_{AssignBatch} + (AssignTask - 1) * (L_{AssignBatch} - S_{AssignBatch})/(N - 1)$
13:        $AssignTask \leftarrow AssignTask + 1$
14:    end if
15:    if $AssignTask > N$ then
16:        $AssignTask \leftarrow 1, AssignBatch \leftarrow AssignBatch + 1$
17:    end if
18: end while
```

Theorem 1 *If, for all batches, the deadline time is greater than the submission time plus the maximum execution time among workers ($\forall i, D_i \geq S_i + C\lceil \frac{N}{P} \rceil$) then Algorithm 4 results in all batches meeting their execution deadlines.*

Proof 1 *Proof by induction. At time S_1, by definition no workers are executing a task and all P workers have connection times. Next, assume that no workers are executing a task and all workers have been given connection times by time S_i. We now demonstrate that if $D_i \geq S_i + C\lceil \frac{N}{P} \rceil$, then all tasks in B_i will be finished at or before D_i and at time S_{i+1} there will be no workers executing tasks.*

If $P \geq N$ then at time S_i all tasks for batch B_i have been assigned, and each worker will receive at most 1 task from the batch. The latest task distribution time will be $S_i + (N-1)\frac{L_i - S_i}{N-1} = L_i = D_i - C$ and the latest task completion time will be D_i, meaning that no workers are executing a task after D_i. If $D_i < S_i + C$, then $L_i < S_i$, which is a contradiction because no tasks from B_i can be distributed before S_i.

If $P < N$ then at time S_i only P tasks from batch B_i have been assigned. During the execution of batch B_i, a worker will request and execute either $\lfloor \frac{N}{P} \rfloor$ or $\lceil \frac{N}{P} \rceil$ tasks. Because a worker executes tasks one by one, the latest a worker will request a task is at time $max(L_i, C(\lceil \frac{N}{P} \rceil - 1))$ and the latest task completion time will be $max(D_i, C\lceil \frac{N}{P} \rceil)$. If $D_i < S_i + C\lceil \frac{N}{P} \rceil$, the batch completion time will be $C\lceil \frac{N}{P} \rceil > D_i$ and the deadline will not be met. In this case, there can be no guarantees about the execution state of workers at time S_{i+1}. If $D_i \geq S_i + C\lceil \frac{N}{P} \rceil$, the batch completion time will be D_i and no workers will be executing tasks at S_{i+1}.

Therefore all batches will meet their execution deadlines if and only if $\forall i, D_i \geq S_i + C\lceil \frac{N}{P} \rceil$.

5.4.2 Semi-Communication-Reliable Homogeneous Workers

Algorithm 5 Semi-Communication-Reliable Homogeneous Workers

```
1:  Calculate $\lambda$ from $K$ and $N$; estimate $P$; $T \leftarrow \frac{0.9PL}{\lambda}$
2:  $SendBatch \leftarrow 1, SendTask \leftarrow 1$
3:  while $SendBatch < M$ do
4:      Get connection from $W_j$
5:      if $CurrentTime() \geq S_{SendBatch}$ then
6:          Send task $T^{SendBatch}_{SendTask}$ to $W_j$
7:          $SendTask \leftarrow SendTask + 1$
8:      end if
9:      $R_j \leftarrow CurrentTime() + T$; Send $R_j$ to $W_j$
10:     if $SendTask > N$ then
11:         $SendTask \leftarrow 0, SendBatch \leftarrow SendBatch + 1$
12:         $P_{active} =$ num connections in last $2T$ seconds, $T \leftarrow \frac{0.9P_{active}L}{\lambda}$
13:     end if
14: end while
```

In this section we consider homogeneous workers that are computation-reliable and semi-communication-reliable. In other words, they behave like VC workers when requesting tasks, but will always complete a task on time once it is received. Here we modify Algorithm 4 to use the model of worker requests from Section 5.3.3 and ensure enough task requests for a given batch.

Predicting the availability state of an arbitrary VC worker at a specific time is extremely difficult, especially for times far in the future. Algorithm 4 cannot be used in such environments because it requires each worker W_j to be available at R_j. As previously mentioned, Algorithm 4 does not request all workers to connect at time S_i but instead maintains a constant rate of task requests between S_i and L_i. In the same way, computing low latency batches with semi-communication-reliable workers is possible by maintaining a stream of task requests.

In Section 5.3.3, we demonstrated that task requests from VC workers can be modeled as a Poisson process. Given this model, we now determine how to calculate the reconnection period T so as to distribute all tasks before the batch deadline L. Given Corollary 1, we can control the probability K of at least N task requests being sent to the master from P workers in a time period L. This probability is controlled by specifying a reconnection period T based on P, L and N. Although in this algorithm we use P to calculate T, there is no intrinsic dependence of T on P. Other techniques for counting active workers are just as suitable, as long as the active time period is greater than T.

The number of task requests occurring in a Poisson process follows the Poisson distribution. This gives the probability of exactly N task requests occurring in a given time period. Because this is a probabilistic model we can only put a bound on the probability K of a specified number of task requests occurring. For a probability K of at least N task requests in a given time period L, λ must satisfy:

$$K \geq 1 - \sum_{i=0}^{N-1} \frac{e^{-\lambda}\lambda^i}{i!} \tag{5.1}$$

Based on our observations, roughly 10% of workers become inactive in a daily cyclical pattern. λ is calculated assuming large numbers of workers do not become inactive simultaneously. To compensate, we calculate the reconnection period using 90% of the active worker count. In Section 5.3.3 we showed worker connections can be modeled as a Poisson process with rate parameter $\lambda = P/T$. Given λ from the above equation we can rewrite this as the reconnection period:

$$T = \frac{0.9PL}{\lambda} \tag{5.2}$$

The number of active workers P can change over long time periods, though λ and L are assumed to be constant. Therefore the value of T must change during the course of the computation. One way to track active workers is to count the number of workers which connected recently, in this case, the number of unique workers which connected in the last $2T$ seconds.

Algorithm 5 demonstrates how to use the active worker count to distribute tasks to semi-communication-reliable workers. This algorithm ensures sufficient task requests to the master before the distribution deadline even with individual worker unreliability and daily fluctuations. Algorithm 5 differs from Algorithm 4 in that it does not track the pre-assignment of tasks to workers. Instead, the goal of N task requests is implicitly achieved by altering the reconnection rate (line 12). We demonstrate the effectiveness of this algorithm in Section 5.5.

5.4.3 Semi-reliable Heterogeneous Workers

Finally, we propose an algorithm to replicate and distribute tasks to semi-reliable heterogeneous workers. These are semi-communication-reliable and semi-computation-reliable workers as described in Section 5.2 and modeled in Sections 5.3.3 and 5.3.4. As demonstrated, worker availability at time R can be estimated based on the number of times the worker was available at past times $R - TX$.

To determine whether a worker will complete a given task by the deadline, we estimate the probability of the worker providing the required amount of computational power between task distribution and the deadline. Suppose a worker receives a task at time R with deadline D. For a worker with task computation time C, we want to estimate the probability of the worker being in the available state for more than C seconds in the time interval $[R, D]$. If $C > D - R$ then the probability is 0. Otherwise, we estimate the probability based on the time to compute the task if it were started at past times $R - TX$ using $T = 1$ day and $T = 1$ week. If more than half of the tasks computed at the previous time periods would have missed the deadline, $Pr_i^{Success} = 0$ and the worker is not assigned a task. Otherwise, the probability is estimated based on mean worker task computation time $\overline{C}$ relative to this workers computation time C_i using the equation $Pr_i^{Success} = min(0.99, 2\overline{C}/C_i)$. The justification for this is that for faster workers, small unpredictable periods of unavailability will have less effect and the task will more likely finish before the deadline. This type of speed based scheduling is also investigated in [85].

Algorithm 6 shows the master task distribution algorithm for semi-reliable heterogeneous workers. This algorithm is similar to Algorithm 5, except we create replicas of some tasks that have a low probability of finishing before the deadline. To decide which tasks to replicate, we keep an estimate of the probability Pr_i^{Fail} of missing the deadline for each task T_i. This estimate starts at 1 for all tasks, then is updated (line 9) based on the estimated probability of success (line 6) as the tasks are assigned to workers. Also, because the workers are heterogeneous, L is computed using the mean task completion time $\overline{C}$.

Each of the algorithms described here distributes N tasks from each of M batches. At the end of each batch, the number of active workers is calculated, which takes $O(P)$ (or less, depending on the method). For Algorithms 1 and 5, distribution of a task takes constant time $O(1)$, so these algorithms have time complexity $O(M(N+P))$. For Algorithm 6, the task with the highest probability of failure is sent to the worker. Using a tree structure, maintaining a list of tasks sorted by probability of failure has time complexity $O(logN)$. Therefore Algorithm 6 has time complexity $O(M(NlogN + P))$.

However, it is worth pointing out that for the numbers of tasks and workers typically involved in VC systems ($N < 10^5$ per day, $P < 10^7$ [76]), each task distribution requires very little CPU time. In fact, the limiting factor

Algorithm 6 Semi-Reliable Heterogeneous Workers

```
1: Calculate $\lambda$ from $K, N$; estimate $P$; $T \leftarrow \frac{0.9PL}{\lambda}$
2: $SendBatch \leftarrow 1, SendTask \leftarrow 1$
3: $\forall i \in [1, SendTask], Pr_i^{Fail} = 1$
4: while $SendBatch < M$ do
5:     Get connection from $W_j$
6:     Calculate probability $Pr^{Success}$ of worker finishing task before deadline
7:     if $CurrentTime() \geq S_{SendBatch}$ and $Pr^{Success} > 0$ then
8:         Send task $T_i$ with highest $Pr^{Fail}$ to $W_j$
9:         $Pr_i^{Fail} \leftarrow Pr_i^{Fail}(1 - Pr^{Success})$
10:    end if
11:    $R_j \leftarrow CurrentTime() + T$; Send $R_j$ to $W_j$
12:    if All tasks finished then
13:        $SendBatch \leftarrow SendBatch + 1$
14:        $\forall i \in SendTask, Pr_i^{Fail} = 1$
15:        $P_{active}$ = num workers in last $2T$ seconds; $T \leftarrow \frac{0.9P_{active}L}{\lambda}$
16:    end if
17: end while
```

Table 5.2: Simulation parameters.

Parameter Name	Parameter Value
Number of Batches (M)	256
Tasks per Batch (N)	1024, 2048, 4096
Target Success Probability (K)	0.99
λ derived from K, N	1100, 2155, 4247
Batch Deadline (D)	1 hour, 4 hours
Simulation Start (S_0)	Sep 1, 2007; Nov 1, 2007; Jan 1, 2008
Batch Submission (S_i)	$S_{i+1} = S_i + D$
Task Computation Time $(C$ or $\overline{C})$	30 minutes, 1 hour, 2 hours

in VC computations is generally bandwidth to the master rather than CPU time.

5.5 Experiments

We conducted a series of simulation experiments to test the algorithms in Sections 5.4.2 and 5.4.3 and compare them to alternate algorithms. Simulations were implemented using a custom event-driven simulator program that emulates a VC master-worker environment based on trace files using double precision for all times. To improve simulation time, workers are skipped forward to a week before the simulation start. Master operation depends on the algorithm being executed. Workers execute tasks until completion - they do not abort a task if the deadline has past. The trace files for all simulations in this section consisted of 37,472 randomly selected workers from the trace data set in Section 5.3.1.

5.5.1 Semi-Communication-Reliable Workers

To confirm that Algorithm 5 provides sufficient task requests from workers, we performed experiments using the trace data described above. The parameters for the experiments are shown in Table 5.2. These parameters represent a range of possible low latency applications, from small to large batches with short to long tasks. Simulations with impossible parameter combinations (e.g. $C \geq D$) were not performed. Dates for simulation start (S_0) were chosen to get a good range of active workers.

Figure 5.7 shows two sample results from the experiments. The figures show the active worker count at the start of each batch, and the number of task requests received during each batch. The light histogram represents the number of task requests that arrived before the batch computation deadline, while the dark histogram represents the number of task requests that arrived before the batch distribution deadline. The horizontal line is the minimum number of task requests needed to successfully complete the batch. Daily and weekly worker unavailability cycles can be clearly seen in the active worker count for both graphs.

These results show that Algorithm 5 maintains a steady stream of worker task requests for batches with varying characteristics. Even though the number of active workers changes significantly over time, the required number of task requests arrive before or occasionally slightly after the distribution deadline. It is worth noting that task requests are spread evenly before and after the distribution deadline. This is important for systems where the submission time of a future batch is unknown, and a constant stream of task requests is required to ensure batch satisfaction.

For comparison, we also implemented a simpler algorithm that adjusts the reconnection rate using a feedback loop based on the fraction of successful task requests in the previous batch. We refer to this algorithm as the "shifting" algorithm. If a batch fails to receive enough task requests before the distribution deadline, this algorithm decreases the reconnection rate proportional to the number missed. If a batch receives over 10% more than the needed task requests, it increases the reconnection rate by a proportionate amount.

Figure 5.8 shows a comparison of the shifting reconnection rate algorithm with the the Poisson based algorithm described in Section 5.4.2. This shows the percent of batches in the experiments which received a given fraction of task requests before the distribution deadline. In over 70% of batches, both algorithms distributed all of the tasks before the deadline. However, the shifting algorithm resulted in many more batches with very few task requests. This is because the shifting algorithm can only react to changes in the active worker count after they have affected a batch, as opposed to the Poisson method which adapts to the worker count as well as the expected fluctuation. We found that with long reconnection periods, the shifting algorithm increases the reconnection rate for multiple successive batches, then suddenly encounters a lack of workers as the previous adjustments take effect. This helps explain the batches with very few task requests for the shifting algorithm in Figure 5.8.

From these results, we feel confident that Algorithm 5 and the model described in Section 5.3.3 are useful and valid for ensuring sufficient task requests to meet low-latency batch deadlines in VC systems. Although the algorithm failed to provide complete reliability in terms of task requests, we demonstrated that it can adapt to changes in worker availability better than a simpler algorithm.

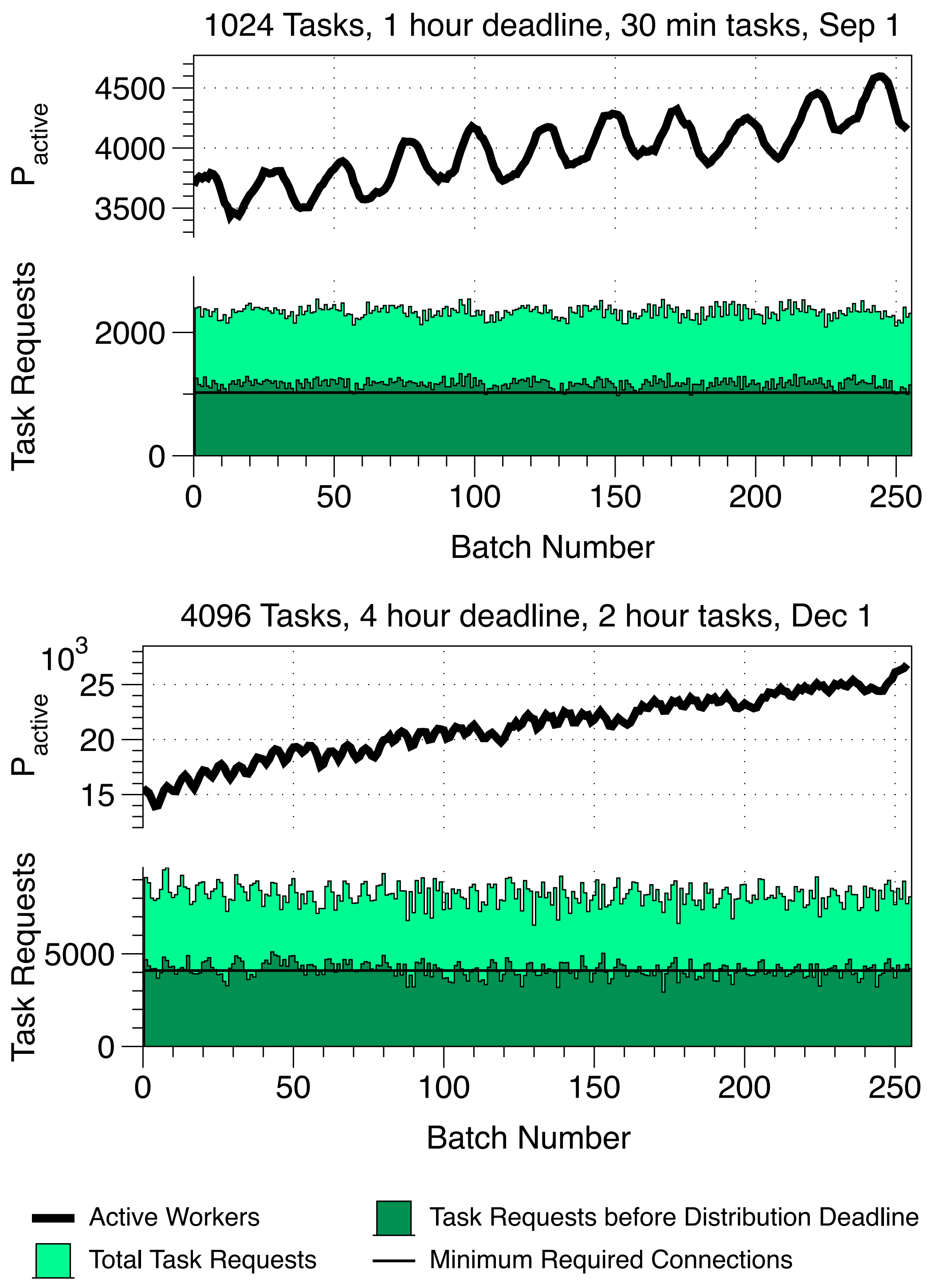


Figure 5.7: Two results from semi-communication reliable worker experiments.

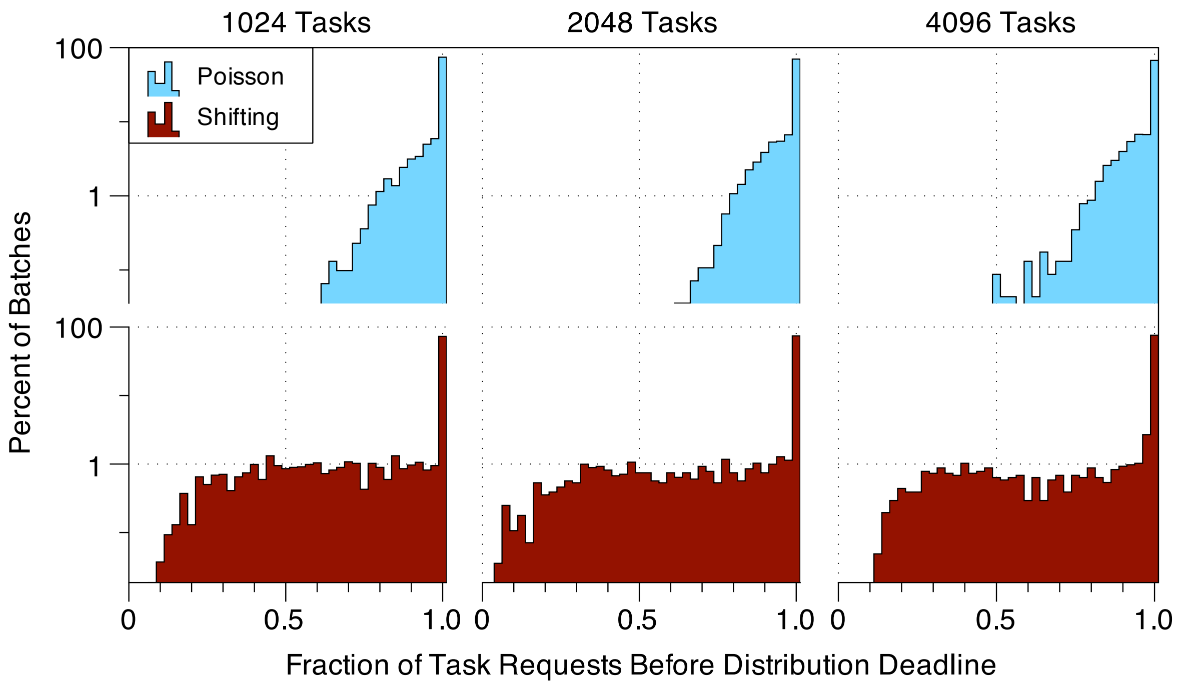


Figure 5.8: Fraction of tasks in a batch that were distributed before the deadline.

5.5.2 Semi-Reliable Workers

Finally we test the efficacy of Algorithm 6 in completing batch tasks before the deadline. The parameters for these experiments are the same as Section 5.5.1, shown in Table 5.2. In all these experiments, an average of two copies of each task is distributed to workers (more or less depending on the algorithm). For comparison, we also tested two alternate methods of computing the probability of success $Pr^{Success}$ in Algorithm 6.

In these experiments we tested three ways of computing $Pr^{Success}$. The first involves simple replication where each task is replicated twice and sent to an arbitrary worker, regardless of worker speed or past history. The second involves calculating $Pr^{Success}$ solely based on worker speed, using the equation described in Section 5.4.3. The final technique, described in Section 5.4.3 uses predictions based on past worker history in combination with worker speed to estimate $Pr^{Success}$.

The results of the experiments for Algorithm 6 are shown in Figure 5.9. This figure shows the percent of batches where a given fraction of task requests satisfied the deadline. The graphs going left to right represent different ratios of task computation time to deadline time. The leftmost graphs represent all experiments with the tighter deadline of task computation time C being half of deadline time D. The rightmost graphs represent experiments with a looser deadline of $C = D/8$. From top to bottom, the graphs show the results of performing Algorithm 6 using the simple method of task replication and distribution, task assignment based on worker speed, and task assignment based on worker speed with past history.

First we notice that longer deadlines relative to task size result in more tasks and batches being satisfied. This

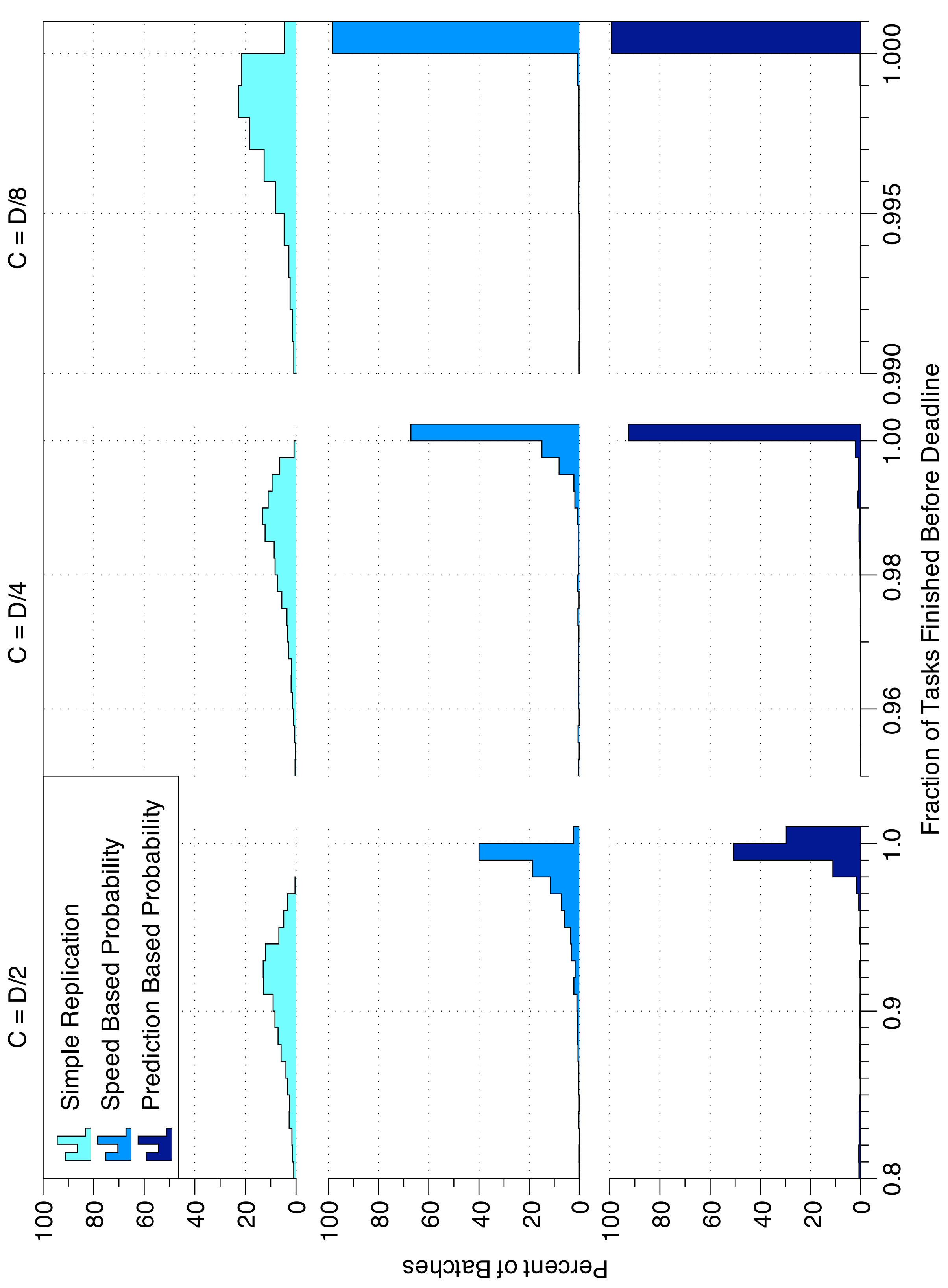


Figure 5.9: Task and batch satisfaction rates from the experiments.

makes sense because Pr^{Fail} is set to 0 when a task completes, meaning the remaining task requests in the batch will more likely be assigned tasks that would fail anyway. In contrast, with tighter deadlines the tasks are allocated to workers in a less efficient manner in order to meet the deadline.

There is a significant jump in batch satisfaction from using worker speed to estimate $Pr^{Success}$. This effect has been noted before in desktop grids [85]. The same study found no connection between worker speed and availability. This means that for any availability pattern a faster worker will have a greater chance of meeting a deadline simply because the task is more likely to finish before the worker goes into a long unavailable interval.

The bottom three graphs in Figure 5.9 show the effectiveness of the techniques proposed in this dissertation. For looser deadlines of $C = D/4$ and $C = D/8$, our techniques result in over 90% of batches being satisfied. For the tighter deadline of $C = D/2$, the number of satisfied batches significantly drops but is still higher than the alternate techniques. With tighter deadlines, worker unavailability from unpredictable causes such as user activity becomes more of an issue.

Based on these results, we feel Algorithm 6 provides a good way of performing low latency batches in a VC environment. Compared to simpler methods of managing communication unreliability such as the shifting reconnection algorithm, the Poisson method described in this dissertation provides a more accurate method of controlling workers. For managing computational unreliability, probabilistic assignment based on worker speed and past history proves more effective than arbitrary task assignment or using worker speed only.

5.6 Conclusions

In this chapter we proposed methods for performing low latency batches of tasks with deadlines in VC environments. To do so, we first proposed analysis based models for handling communication and computation unreliability in VC workers. These models were used to develop task distribution algorithms aimed at low latency batch VC, which were then validated using execution trace data from an actual VC environment.

Although the experiments showed the effectiveness of these algorithms, further work can be done to improve the algorithms, especially in regards to estimating task success on a given worker. Other techniques [84, 87] show promise in this regard, though implementations should avoid computationally intensive techniques when computing with deadlines.

To make practical use of these results we plan to study methods for integrating low-latency batch computing in existing VC systems such as BOINC. BOINC already supports client reconnection times and recording past availability states, so implementing low latency in BOINC is primarily a matter of allowing users to specify batch characteristics and quickly processes completed batches.

Chapter 6

Conclusions

6.1 Summary of Contributions

The field of biophysical modeling and simulation will expand in the coming years. Numerous software packages have been developed to deal with the modeling and simulation, though none of them handle parallel simulation of general biophysical models. In this dissertation, we detailed techniques for improving parallel simulations of biophysical models over a range of parallel computing platforms and scales. The contributions presented in this thesis can be summarized as follows:

Efficient parallelization of source based biophysical simulations. We explained how we used automated analysis of biophysical models to improve computational speed and minimize communication in the parallel simulation. This was combined with redundant computation of communication intensive function calculations to further reduce communication. Fundamentally, this demonstrates a method for distinguishing dependence types among computations (e.g. functions and states), then using this information to minimize dependencies between computing nodes. This has the effect of reducing communication and accelerating the simulation. This technique is potentially applicable in other fields when performing large scale simulations of irregular models.

Extendable program for efficient parallel simulation of large models. We introduced *insilico*Sim, an extendable parallel simulator for 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.

Heuristics for loosely synchronous biophysical simulations on volunteer computing platforms. In this dissertation we proposed methods for performing low latency batches of tasks with deadlines in VC environments. To do so, we first proposed analysis based models for handling communication and computation unreliability in VC workers. These models were used to develop task distribution algorithms aimed at low latency batch VC, which were then validated using execution trace data from an actual VC environment. These results are applicable to large biophysical computations, especially biomolecular dynamics simulations similar to Folding@home.

6.2 Future Work

There are several ways each of the areas in this dissertation could be extended and improved in future work:

Extend *insilico*Sim to additional input and output formats. Thanks to the interface based design of *insilico*Sim, it is relatively easy to extend it to additional formats. In particular, the CellML and FieldML formats offer a range of models suited for *insilico*Sim, and visualization libraries could be used to quickly examine the results of the simulation.

Implement *insilico*Sim interfaces for additional model types. The models used in this dissertation were composed of ODEs, functions and parameters. However, many biophysical phenomena can be best described by partial differential equations, finite element techniques, stochastic differential equations and agent based models. These are already or will soon be supported by ISML, and so their inclusion into *insilico*Sim is a necessary step towards a comprehensive simulation solution.

Improve performance of *insilico*Sim. Performance can likely be improved by ordering object calculations such that cache misses are minimized, and by overlapping computation and communication. These can be transparently achieved by the object manager described in 4.3.2.

Use more computationally intensive techniques to estimate worker availability. The techniques described in Chapter 5 are less computationally intensive than those described in other work [84, 87]. However, the other methods are potentially more accurate and could be used with the heuristics in this dissertation to improve overall performance.

Each of these areas represent a possible extension of the topics discussed in this dissertation. We believe the work in this dissertation will provide some initial steps towards improving the range and performance of large scale biophysical simulations to benefit the fields of biology and physiology.

Bibliography

[1] H.L Anderson. Metropolis, monte carlo and the maniac. *Los Alamos Science*, 14:96–108, 1986.

[2] Charles R Snyder, Charles A Snyder, and Chetan S Sankar. Use of information technologies in the process of building the boeing 777. *Journal of Information Technology Management (JITM)*, IX(III):31–42, 1998.

[3] András Varga and Rudolf Hornig. An overview of the omnet++ simulation environment. pages 1–10, 2008.

[4] Denis Noble. *The Music of Life: Biology beyond the Genome*. Jun 2006.

[5] Henry Markram. The blue brain project. *Nat Rev Neurosci*, 7(2):153–60, Feb 2006.

[6] Stefan M Larson, Christopher D Snow, Michael Shirts, and Vijay S Pande. Folding@home and genome@home: Using distributed computing to tackle previously intractable problems in computational biology. *Modern Methods in Computational Biology, Horizon Press*, 2003.

[7] Bojan Zagrovic and Vijay S Pande. Structural correspondence between the alpha-helix and the random-flight chain resolves how unfolded proteins can have native-like properties. *Nat Struct Biol*, 10(11):955–61, Nov 2003.

[8] Peter W Swaan and Sean Ekins. Reengineering the pharmaceutical industry by crash-testing molecules. *Drug Discov Today*, 10(17):1191–200, Sep 2005.

[9] Denis Noble. Modeling the heart–from genes to cells to the whole organ. *Science*, 295(5560):1678–1682, 2002.

[10] Anders Lansner. Associative memory models: from the cell-assembly theory to biophysically detailed cortex simulations. *Trends Neurosci*, 32(3):178–86, Mar 2009.

[11] M Hereld, R.L Stevens, W van Drongelen, and H.C Lee. Developing a petascale neural simulation. *Engineering in Medicine and Biology Society, 2004. IEMBS '04. 26th Annual International Conference of the IEEE*, 2:3999–4002, 2004.

[12] P J Hunter. The iups physiome project: a framework for computational physiology. *Prog Biophys Mol Biol*, 85(2-3):551–69, Jan 2004.

[13] J B Bassingthwaighte. Strategies for the physiome project. *Ann Biomed Eng*, 28(8):1043–58, Aug 2000.

[14] Taishin Nomura. Challenges of physiome projects. *IEEJ Trans. EIS*, 127(10):1491–1497, 2007.

[15] Andrew D McCulloch and Gary Huber. Integrative biological modelling in silico. *Novartis Found Symp*, 247:4–19; discussion 20–5, 84–90, 244–52, Jan 2002.

[16] Lena Strömbäck, David Hall, and Patrick Lambrix. A review of standards for data exchange within systems biology. *Proteomics*, 7(6):857–67, Mar 2007.

[17] 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.

[18] AA Cuellar, CM Lloyd, PF Nielsen, DP Bullivant, DP Nickerson, and PJ Hunter. An overview of cellml 1.1, a biological model description language. *Simul-T Soc Mod Sim*, 79(12):740–747, Jan 2003.

[19] R. S Rivlin and D. W Saunders. Large elastic deformations of isotropic materials. vii. experiments on the deformation of rubber. *Philosophical Transactions of the Royal Society of London. Series A*, 243:251, Apr 1951.

[20] Yoshiyuki Asai, Yasuyuki Suzuki, Yoshiyuki Kido, Hideki Oka, Eric Heien, Masao Nakanishi, Takahito Urai, Kenichi Hagihara, Yoshihisa Kurachi, and Taishin Nomura. Specifications of insilicoml 1.0: A multilevel biophysical model description language. *The journal of physiological sciences : JPS*, 58(7):447–58, Dec 2008.

[21] Berk Hess, Carsten Kutzner, David van der Spoel, and Erik Lindahl. Gromacs 4: Algorithms for highly efficient, load-balanced, and scalable molecular simulation. *J. Chem. Theory Comput.*, 4(3):435–447, Jan 2008.

[22] J Phillips, Gengbin Zheng, S Kumar, and L Kale. Namd: Biomolecular simulation on thousands of processors. *Supercomputing, ACM/IEEE 2002 Conference*, pages 36 – 36, Nov 2002.

[23] David Adalsteinsson, David McMillen, and Timothy C Elston. Biochemical network stochastic simulator (bionets): software for stochastic modeling of biochemical networks. *BMC Bioinformatics*, 5:24, Mar 2004.

[24] N Le Novère and T S Shimizu. Stochsim: modelling of stochastic biomolecular processes. *Bioinformatics*, 17(6):575–6, Jun 2001.

[25] Thierry Emonet, Charles M Macal, Michael J North, Charles E Wickersham, and Philippe Cluzel. Agentcell: a digital single-cell assay for bacterial chemotaxis. *Bioinformatics*, 21(11):2714–21, Jun 2005.

[26] Akira Funahashi, Yukiko Matsuoka, Akiya Jouraku, Mineo Morohashi, Norihiro Kikuchi, and Hiroaki Kitano. Celldesigner 3.5: A versatile modeling tool for biochemical networks. *P Ieee*, 96(8):1254–1265, Jan 2008.

[27] Bruce E Shapiro, Andre Levchenko, Elliot M Meyerowitz, Barbara J Wold, and Eric D Mjolsness. Cellerator: extending a computer algebra system to include biochemical arrows for signal transduction simulations. *Bioinformatics*, 19(5):677–8, Mar 2003.

[28] Masao Nagasaki, Atsushi Doi, Hiroshi Matsuno, and Satoru Miyano. Genomic object net: I. a platform for modelling and simulating biopathways. *Appl Bioinformatics*, 2(3):181–4, Jan 2003.

[29] Stefan Hoops, Sven Sahle, Ralph Gauges, Christine Lee, Jürgen Pahle, Natalia Simus, Mudita Singhal, Liang Xu, Pedro Mendes, and Ursula Kummer. Copasi–a complex pathway simulator. *Bioinformatics*, 22(24):3067–74, Dec 2006.

[30] I Goryanin, T C Hodgman, and E Selkov. Mathematical simulation and analysis of cellular metabolism and regulation. *Bioinformatics*, 15(9):749–58, Sep 1999.

[31] P Mendes. Biochemistry by numbers: simulation of biochemical pathways with gepasi 3. *Trends Biochem Sci*, 22(9):361–3, Sep 1997.

[32] H.M Sauro. Jarnac: a system for interactive metabolic analysis. *Animating the Cellular Map: Proc. Ninth Int'l Meeting on BioThermoKinetics*, pages 221–228, 2000.

[33] Marc Vass, Nicholas Allen, Clifford A Shaffer, Naren Ramakrishnan, Layne T Watson, and John J Tyson. the jigcell model builder and run manager. *Bioinformatics*, 20(18):3680–1, Dec 2004.

[34] M G Poolman. Scrumpy: metabolic modelling with python. *Systems biology*, 153(5):375–8, Sep 2006.

[35] Thomas D Garvey, Patrick Lincoln, Charles John Pedersen, David Martin, and Mark Johnson. Biospice: access to the most current computational tools for biologists. *OMICS*, 7(4):411–20, Dec 2003.

[36] Alan Garny, Denis Noble, Peter J Hunter, and Peter Kohl. Cellular open resource (cor): current status and future directions. *Philos Transact A Math Phys Eng Sci*, 367(1895):1885–905, May 2009.

[37] Trevor Cickovski, Chengbang Huang, Rajiv Chaturvedi, Tilmann Glimm, H Hentschel, Mark Alber, James Glazier, Stuart Newman, and Jesus Izaguirre. A framework for three-dimensional simulation of morphogenesis. *IEEE/ACM Transactions on Computational Biology and Bioinformatics (TCBB)*, 2(4), Oct 2005.

[38] Nobuyoshi Ishii, Martin Robert, Yoichi Nakayama, Akio Kanai, and Masaru Tomita. Toward large-scale modeling of the microbial cell for computer simulation. *J Biotechnol*, 113(1-3):281–94, Sep 2004.

[39] Samik Ghosh, Preetam Ghosh, Kalyan Basu, Sajal Das, and Simon Daefler. isimbiosys: A discrete event simulation platform for 'in silico' study of biological systems. *ANSS '06: Proceedings of the 39th annual Symposium on Simulation*, Apr 2006.

[40] Brett G Olivier, Johann M Rohwer, and Jan-Hendrik S Hofmeyr. Modelling cellular systems with pysces. *Bioinformatics*, 21(4):560–1, Feb 2005.

[41] Frank Bergmann and Herbert Sauro. Sbw - a modular framework for systems biology. *WSC '06: Proceedings of the 38th conference on Winter simulation*, Dec 2006.

[42] Nobuaki Sarai, Satoshi Matsuoka, and Akinori Noma. simbio: a java package for the development of detailed cell models. *Prog Biophys Mol Biol*, 90(1-3):360–77, Jan 2006.

[43] LM Loew and JC Schaff. The virtual cell: a software environment for computational cell biology, Jan 2001.

[44] Jacob Czech, Markus Dittrich, and Joel R Stiles. Rapid creation, monte carlo simulation, and visualization of realistic 3d cell models. *Methods Mol Biol*, 500:237–87, Jan 2009.

[45] R McFarlane and I Biktasheva. High performance computing for the simulation of cardiac electrophysiology. *Software Engineering Advances, 2008. ICSEA '08. The Third International Conference on*, pages 13 – 18, Oct 2008.

[46] Sergey Missan and Terence F McDonald. Cese: Cell electrophysiology simulation environment. *Appl Bioinformatics*, 4(2):155–6, Jan 2005.

[47] Semahat S Demir. Interactive cell modeling web-resource, icell, as a simulation-based teaching and learning tool to supplement electrophysiology education. *Ann Biomed Eng*, 34(7):1077–1087, Jan 2006.

[48] J L Puglisi and D M Bers. Labheart: an interactive computer model of rabbit ventricular myocyte ion channels and ca transport. *Am J Physiol, Cell Physiol*, 281(6):C2049–60, Dec 2001.

[49] Rajagopal Ananthanarayanan and Dharmendra Modha. Anatomy of a cortical simulator. *SC '07: Proceedings of the 2007 ACM/IEEE conference on Supercomputing*, Nov 2007.

[50] James M Bower and David Beeman. Constructing realistic neural simulations with genesis. *Methods Mol Biol*, 401:103–25, Jan 2007.

[51] Hans Plesser, Jochen Eppler, Abigail Morrison, Markus Diesmann, and Marc-Oliver Gewaltig. Efficient parallel simulation of large-scale neuronal networks on clusters of multiprocessor computers. *Euro-Par 2007 Parallel Processing*, pages 672–681, 2007.

[52] M L Hines and N T Carnevale. Neuron: a tool for neuroscientists. *The Neuroscientist : a review journal bringing neurobiology, neurology and psychiatry*, 7(2):123–35, Apr 2001.

[53] D Pecevski, T Natschläger, and K Schuch. Pcsim: A parallel simulation environment for neural circuits fully integrated with python. *Frontiers in neuroinformatics*, 3:11, Jan 2009.

[54] M Djurfeldt, M Lundqvist, C Johansson, M Rehn, O Ekeberg, and A Lansner. Brain-scale simulation of the neocortex on the ibm blue gene/l supercomputer. *Ibm J Res Dev*, 52(1-2):31–41, Jan 2008.

[55] Mark Pogson, Rod Smallwood, Eva Qwarnstrom, and Mike Holcombe. Formal agent-based modelling of intracellular chemical interactions. *Proceedings*, 85(1):37–45, Jul 2006.

[56] D. Nickerson, M. Nash, P. Nielsen, N. Smith, and P. Hunter. Computational multiscale modeling in the iups physiome project: modeling cardiac electromechanics. *IBM J. Res. Dev.*, 50(6):617–630, 2006.

[57] T Kawazu, M Nakanishi, Y Suzuki, S Odai, and T Nomura. A platform for in silico modeling of physiological systems. *Engineering in Medicine and Biology Society, 2007. EMBS 2007. 29th Annual International Conference of the IEEE*, pages 1394 – 1397, Jul 2007.

[58] GM Raymond, E Butterworth, and JB Bassingthwaighte. Jsim: Free software package for teaching phyiological modeling and research. *Experimental Biology*, 280:102, 2003.

[59] Daniel A Beard, Randall Britten, Mike T Cooling, Alan Garny, Matt D B Halstead, Peter J Hunter, James Lawson, Catherine M Lloyd, Justin Marsh, Andrew Miller, David P Nickerson, Poul M F Nielsen, Taishin Nomura, Shankar Subramanium, Sarala M Wimalaratne, and Tommy Yu. Cellml metadata standards, associated tools and repositories. *Philos Transact A Math Phys Eng Sci*, 367(1895):1845–67, May 2009.

[60] Bruce Damer. Evogrid website. 2009.

[61] Kenichi Asami and Tadashi Kitamura. The development of a physiological simulation system for the human circulatory system coupling macro and micro models. *IEICE TRANSACTIONS on Information and Systems*, 86(11):2499, 20031101.

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

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

[64] Yasuyuki Suzuki, Yoshiyuki Asai, Toshihiro Kawazu, Masao Nakanishi, Yoshiki Taniguchi, Eric Heien, Kenichi Hagihara, Yoshihisa Kurachi, and Taishin Nomura. A platform for in silico modeling of physiological systems ii. cellml compatibility and other extended capabilities. *Engineering in Medicine and Biology Society, 2008. EMBS 2008. 30th Annual International Conference of the IEEE*, pages 573–576, 2008.

[65] Alan Lloyd Hodgkin and Andrew Huxley. A quantitative description of membrane current and its application to conduction and excitation in nerve. *J Physiol (Lond)*, 117(4):500–44, Aug 1952.

[66] I A Rybak, J F Paton, and J S Schwaber. Modeling neural mechanisms for genesis of respiratory rhythm and pattern. i. models of respiratory neurons. *J Neurophysiol*, 77(4):1994–2006, Apr 1997.

[67] George Karypis and Vipin Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM J. Sci. Comput.*, 20(1):359–392, 1998.

[68] Ilya A Rybak, Katinka Stecina, Natalia A Shevtsova, and David A McCrea. Modelling spinal circuitry involved in locomotor pattern generation: insights from the effects of afferent stimulation. *J Physiol-London*, 577(2):641–658, Jan 2006.

[69] AX Xu and MR Guevara. Two forms of spiral-wave reentry in an ionic model of ischemic ventricular myocardium, Jan 1998.

[70] Matthias Jeschke, Roland Ewald, Alfred Park, Richard Fujimoto, and Adelinde Uhrmacher. A parallel and distributed discrete event approach for spatial cell-biological simulations. *ACM SIGMETRICS Performance Evaluation Review*, 35(4), Mar 2008.

[71] 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.

[72] 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.

[73] Alan Hindmarsh, Peter Brown, Keith Grant, Steven Lee, Radu Serban, Dan Shumaker, and Carol Woodward. Sundials: Suite of nonlinear and differential/algebraic equation solvers. *Transactions on Mathematical Software (TOMS)*, 31(3), Sep 2005.

[74] Eric Heien, Yoshiyuki Asai, Taishin Nomura, and Kenichi Hagihara. Optimization techniques for parallel biophysical simulations generated by insilicoide. *IPSJ Symposium Series*, 2009(2):1–8, Jan 2009.

[75] Ilya A Rybak, Natalia A Shevtsova, Myriam Lafreniere-Roula, and David 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.

[76] David Anderson, Jeff Cobb, Eric Korpela, Matt Lebofsky, and Dan Werthimer. Seti@home: an experiment in public-resource computing. *Communications of the ACM*, 45(11), Nov 2002.

[77] Leslie Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8), Aug 1990.

[78] Jennifer Schopf and Francine Berman. Stochastic scheduling. *Supercomputing '99: Proceedings of the 1999 ACM/IEEE conference on Supercomputing (CDROM*, Jan 1999.

[79] Krishnaveni Budati, Jason Sonnek, Abhishek Chandra, and Jon Weissman. Ridge: combining reliability and performance in open grid platforms. *HPDC '07: Proceedings of the 16th international symposium on High performance distributed computing*, Jun 2007.

[80] D Kondo, A.A Chien, and H Casanova. Resource management for rapid application turnaround on enterprise desktop grids. *Supercomputing, 2004. Proceedings of the ACM/IEEE SC2004 Conference*, pages 17–17, 2004.

[81] Brent Rood and Michael J Lewis. Scheduling on the grid via multi-state resource availability prediction. *Grid Computing, 2008 9th IEEE/ACM International Conference on*, pages 126 – 135, Jan 2008.

[82] D Kondo, M Taufer, C Brooks, H Casanova, and A Chien. Characterizing and evaluating desktop grids: an empirical study. *Parallel and Distributed Processing Symposium, 2004. Proceedings. 18th International*, page 26, Mar 2004.

[83] P Malecot, D Kondo, and G Fedak. Xtremlab: A system for characterizing internet desktop grids. *High Performance Distributed Computing, 2006 15th IEEE International Symposium on*, pages 357 – 358, Jan 2006.

[84] Derrick Kondo, Artur Andrzejak, and David P Anderson. On correlated availability in internet-distributed systems. *Grid Computing, 2008 9th IEEE/ACM International Conference on*, pages 276 – 283, Jan 2008.

[85] Derrick Kondo, Gilles Fedak, Franck Cappello, Andrew A Chien, and Henri Casanova. Characterizing resource availability in enterprise desktop grids. *Future Gener Comp Sy*, 23(7):888–903, Jan 2007.

[86] Daniel Nurmi, John Brevik, and Richard Wolski. Modeling machine availability in enterprise and wide-area distributed computing environments. pages 432–441, 2005.

[87] Artur Andrzejak, Derrick Kondo, and David P Anderson. Ensuring collective availability in volatile resource pools via forecasting. volume 5273, pages 149–161, 2008.

[88] EunJoung Byun, SungJin Choi, MaengSoon Baik, ChongSun Hwang, ChanYeol Park, and Soon Young Jung. Scheduling scheme based on dedication rate in volunteer computing environment. *Parallel and Distributed Computing, 2005. ISPDC 2005. The 4th International Symposium on*, pages 234–241, 2005.

[89] D Kondo, B Kindarji, G Fedak, and F Cappello. Towards soft real-time applications on enterprise desktop grids. *Cluster Computing and the Grid, 2006. CCGRID 06. Sixth IEEE International Symposium on*, 1:65–72, 2006.

[90] Derrick Kondo, Filipe Araujo, Patricio Domingues, and Luis Silva. Validating desktop grid results by comparing intermediate checkpoints. Technical Report TR-0059, Oct 2006.

[91] C Christensen, T Aina, and D Stainforth. The challenge of volunteer computing with lengthy climate model simulations. *e-Science and Grid Computing, 2005. First International Conference on*, pages 8 pp.–15, 2005.

[92] Trilce Estrada, Olac Fuentes, and Michela Taufer. A distributed evolutionary method to design scheduling policies for volunteer computing. *SIGMETRICS Performance Evaluation Review*, 36(3), Nov 2008.

[93] Y Murata, T Inaba, H Takizawa, and H Kobayashi. Implementation and evaluation of a distributed and cooperative load-balancing mechanism for dependable volunteer computing. *Dependable Systems and Networks With FTCS and DCC, 2008. DSN 2008. IEEE International Conference on*, pages 316 – 325, May 2008.

[94] Philippe Golle and Ilya Mironov. Uncheatable distributed computations. volume 2020, pages 425–440, 2001.

[95] J Sonnek, A Chandra, and J.B Weissman. Adaptive reputation-based scheduling on unreliable distributed infrastructures. *Parallel and Distributed Systems, IEEE Transactions on*, 18(11):1551–1564, 2007.

[96] D.P Anderson. Boinc: a system for public-resource computing and storage. *Grid Computing, 2004. Proceedings. Fifth IEEE/ACM International Workshop on*, pages 4– 10, 2004.

[97] D.P Anderson and G Fedak. The computational and storage potential of volunteer computing. *Cluster Computing and the Grid, 2006. CCGRID 06. Sixth IEEE International Symposium on*, 1:73– 80, 2006.

[98] E Heien, N Fujimoto, and K Hagihara. Computing low latency batches with unreliable workers in volunteer computing environments. *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on*, pages 1 – 8, Mar 2008.

[99] M A Stephens. Edf statistics for goodness of fit and some comparisons. *Journal of American Statistical Association*, 69(347):730–737, 1974. Analysis of the Anderson-Darling Test.

[100] Emanuel Parzen. Stochastic processes. page 324, Jan 1999.