HIGH PERFORMANCE NUMERICAL SIMULATIONS TO SUPPORT SYSTEM LEVEL DESIGN

By

Joshua D. Carl

Dissertation

Submitted to the Faculty of the

Graduate School of Vanderbilt University

in partial fulfillment of the requirements

for the degree of

DOCTOR OF PHILOSOPHY

in

Electrical Engineering

May, 2016

Nashville, Tennessee

Approved:

Gautam Biswas, Ph.D.

Sandeep Neema, Ph.D.

Gabor Karsai, Ph.D.

Xenofon D. Koutsoukos, Ph.D.

Tom Withrow, Ph.D.

To my wife, Karin, and to our kids.
Thank you for your never ending patience and support.
This degree is just as much yours as it is mine.

ACKNOWLEDGEMENTS

There are many people that influenced me in my pursuit of this degree, and they all have my thanks and gratitude.

I would like to thank my academic advisor, Gautam Biswas, for inviting me to Vanderbilt University. He has provided countless hours of advice, support, and guidance during my graduate career, and without him I would not be where I am.

I am thankful for the members of my Ph.D. committee, Sandeep Neema, Gabor Karsai, Xenofon Koutsoukos, and Tom Withrow, for providing me with feedback to improve my research and final dissertation.

I would also like to thank Sandeep Neema and Ted Bapty for financially supporting me at the beginning of this research under DARPA META contract FA8650-10-C-7082. The time on their project was a key contribution to determining my research direction.

My fellow graduate students, both current and former, helped me survive the occasionally arduous journey that is graduate school and include Daniel Mack, Chetan and Aditi Kulkarni, Hamed Khorasgani, Yi Dong, William Emfinger, and Pranav Kumar.

Zsolt Lattmann, Patrik Meijer, and James Klingler were always available to answer technical questions and graciously shared models they had created.

On a personal level I would like to thank my parents, David and Laureen, for putting the idea of earning a Ph.D. in my head, for raising me with the skills to accomplish it, and for their support of me and my family throughout, and especially at the end, of this process. My in-laws, George and Kathy, were also instrumental in helping me and my family across the finish line.

Our friends and the pastors at First Baptist Church Nashville, too many to list, were always encouraging and supportive to me and my family. Thank you for the fellowship.

My wife, Karin, and our kids stood by me through this process and were always understanding of the limitations that come with being a full time graduate student.

Finally, I need to thank my Lord and Savior Jesus Christ. Without his gifts of grace and protection this degree never would have happened.

Joshua D. Carl
Vanderbilt University
August 2015

TABLE OF CONTENTS

Page

LIST OF TABLES

LIST OF FIGURES

CHAPTER I

INTRODUCTION

Cyber-physical systems represent a class of complex engineered systems where functionality and behavior emerge through the interaction between the computation and physical domains; in addition, these interactions can occur, locally, within a system, or be distributed in a networked environment [48]. CPSs are also frequently designed to include human decision making as part of the control of the physical system. Current automobiles are a good example of CPSs. The components and subsystems in a modern automobile range from electrical controllers for the engine, transmission, stability control, air bag deployment, and anti-lock brakes; to mechanical systems such as transfer cases and differentials; to thermal systems that control component temperature; to hydraulic systems such as brakes; and sensors that measure variables such as wheel speed, steering angle, mass of air intake and engine temperature. Each of these systems interact with the other systems to create the final vehicle and produce its behavior. For example, the engine controller reads data from various sensors networked through the car, such as temperature sensors and a throttle position sensor on the drive-by-wire-throttle, to allow modern engines to meet fuel economy and emission standards [48] while providing the driver with the requested performance. The stability control computer can read wheel speed, lateral acceleration, yaw rate, and steering angle from sensors distributed across the vehicle to determine the likelihood of the vehicle rolling over or losing control. The stability control computer is then able to intervene through the anti-lock brakes or electric steering system to keep the vehicle from rolling over and under control [91] [90]. The human driver provides input to all of these systems by controlling the steering, drive-by-wire throttle, and brakes of the vehicle.

These cross-domain interactions are not limited to vehicles, but they extend to nearly all areas of modern system design. An aircraft has electrical controllers working to control

hydraulic systems that keep the plane level during flight with the sensors, controllers, and actuators distributed across the plane [106]. A power plant has thermal, hydraulic, mechanical, and electric systems working together to convert fuel to electricity [84]. Each of these systems have multiple layers of interaction between their physical and computational domains that govern the overall system behavior, and it is the interaction between the domains that allows the system to consistently produce this behavior.

Systems engineering methods play an important role in designing and anlyzing CPSs. In the traditional approach to systems engineering, design has followed a discipline by discipline approach, with individual components being created in isolation to meet specified design criteria. After the individual, compartmentalized, design phases, all of the components are brought together for integration testing to verify that the design criteria are met [101]. This method works well for simpler systems with few interacting components and physical domains.

With the increasing prevalence and complexity of present-day CPSs, the traditional systems engineering approach is proving to be detrimental to the overall design process. Several research papers, such as [48], [74], [64], [101], and [93], have discussed in detail the problems and challenges involved in designing and building CPSs. For CPSs, to analyze and understand the behaviors of the system requires methods by which the different component models can be composed and analyzed both as a full system and as individual components. One approach is to build all of the components in a virtual design environment, and use simulation to perform integration testing throughout the design process [64]. At the beginning of the design the different components can be represented by low fidelity models, possibly taken from a component library, and composed to form an initial design of the final system. As the design progresses the initial models can be replaced with more specific and detailed models, including the final component implementations. At all points in the design process the composed system can be simulated, providing a means to analyze how the integrated system performs [64]. Simulation, therefore, provides a very tight

design-test-redesign feedback loop for the designer. Any changes or tweaks to the model can be made quickly and simply in the virtual design environment and the test re-run to verify for correctness.

These ideas, solutions and approaches to designing CPSs have resulted from research performed as a part of DARPA's Adaptive Vehicle Make (AVM) project [1]. The research published thus far has primarily focused on improving the design and creation of the system models. However, simulating the models designed in the AVM process can lead to long simulation times and the entire AVM design process depends on fast simulation. Therefore, the primary focus of this thesis is to improve the development process of complex CPSs by reducing the simulation time for cross-disciplinary system and subsystem models. The focus of this work will specifically be on simulating the physical dynamics. However, the methods presented here are general, and can be applied to any large equation-based model of dynamic systems.

I.1 Problem Description

The goals of DARPA's adaptive vehicle make (AVM) project are to improve the design process and reduce the development time for next generation military vehicles [73]. A key component for reducing the development time is to employ formal methodologies that support systematic component-based design and simulation of large, complex systems. This simulation is an integral part of the design and analysis methodologies that have to be performed before the system can be fabricated and constructed.

The current models that are produced with the AVM tools have more than 10,000 equations and variables, and take a long time to simulate. As can be seen in Table 1 the simulation times for some vehicle models is measured in hours. Since simulation plays an important role in providing quick feedback to the designers on the correctness and performance of their design solutions, it hard to conceive of vehicle design approaches using these simulations. It is expected that the next generation of vehicles designed with such

Model Name	Test Bench	Equations	Simulation Time (seconds)	Wall Clock Time (hrs)
Tracked Vehicle Full Suspension	12” Half Round Bump 24kph	12,918	30	1.74
	18” Half Round Bump 8kph	20,559	30	2.47
	12” Half Round Bump 22kph	12,918	30	1.76
	18” Half Round Bump 8kph	12,918	30	2.55
	16” Half Round Bump 4kph	12,918	30	1.92

Table 1: Example model complexity and simulation time for AVM designed models.

tools will be even more complex and more detailed, and, therefore, require a much larger number of equations to describe the system behavior, which implies their simulation times will become even longer using the current tools.

Previously, we could rely on the development of faster processor speeds to speedup simulation runs. However, since the year 2005 processor clock speeds have largely leveled off (see Figure 1), and the increase in computing power for commercial chips has been achieved by adding processor cores rather than by increasing clock speed [41]. Modern CPUs in engineering workstations now have 4 to 8 physical cores, and each core may be able to execute two computation threads simultaneously [7][22]. This effectively gives the operating system 8 to 16 CPUs to use for allocating computational work. Exploiting the parallel processing power provided by multi-core architectures to improve the run time of a simulation requires algorithmic changes to the simulation; one has to develop parallel versions of the simulation algorithms to speed up the computation. However, developing these parallel simulation algorithms will require careful consideration of the physical CPU architecture to derive the best parallel performance.

It is worth noting that present day commercial Modelica simulation packages, such as

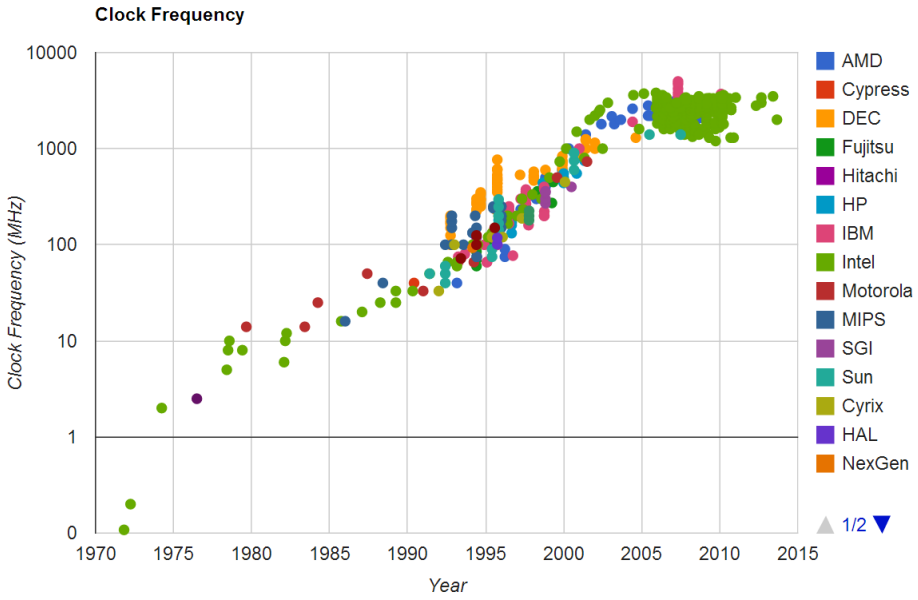


Figure 1: Processor clock frequency vs time [41].

MapleSim, SimulationX, and SystemModeler, at the time of this writing, fail to take advantage of the parallel computing capability provided by multi-core CPUs [12] [18]. Matlab [13] supports parallelization through parallel for-loops that use spawned Matlab computational engines, and while this is useful in many situations, it is not sufficient for efficient parallel processing of a set of model equations due to the tight synchronization required within a simulation time step. Matlab Simulink supports running multiple simulations of the same model in parallel, but it does not support parallelization within a simulation. Dymola only started supporting parallel simulation as a part of the 2015 FD01 release [27], and then only in a few specific situations. The lack of parallel processing support by commercial tools presents an opportunity to improve on the commercial state of the art and shows that there are open research opportunities in this area.

Modern CPUs have several layers of memory between the physical CPU registers and the main system memory collectively called the cache (the cache is discussed in more detail in Chapter V). The cache allows CPU core to keep data that it is currently working on readily available in a layer of cache that provides very quick access, and allows data that

is not needed to stay in a layer of cache farther away from the core [87]. The cache also allows the physical computation cores to communicate with each other. Mismanagement or ignoring the structure of the CPU cache in a multi threaded program can have a drastic negative impact on the program performance [87][83][78], and therefore, parallel software design on a multi-core CPU requires careful consideration of both how to partition the computational problem into independent pieces suitable for parallel simulation, and how those independent pieces interact with each other and the on chip memory.

The potential parallel processing power of modern multi-core CPUs, coupled with the potential performance pitfalls of the CPU cache, leads us to focus our research into parallel simulation algorithms that target a multi core CPU and use the CPU cache as a performance asset instead of a liability.

I.2 Thesis Goals

The goal of this thesis is to take steps toward facilitating the design process for cyber-physical systems by reducing the time it takes to simulate complex and large system models by developing parallel simulation algorithms for multi-core CPU architectures. A primary component of these algorithms is the incorporation of suitable memory management and the use of program constructs that take advantage of the CPU memory architecture. Our research contributions, therefore, focus on:

1. Developing parallel simulation algorithms that appropriately uses the multiple cores and the cache memory organization on a multi-core CPU, and

2. Running experimental studies that help us analyze the effectiveness of various multi-threading and memory management schemes for parallel simulations.

To evaluate our parallel simulation algorithms we will start from a very simple partitioning structure and memory management scheme. This simple algorithm will be evaluated

based on the experimental results from a set of models. The results of each round of experiments will guide the development of the next generation algorithm. This process will be repeated until we observe speedups that are on the order of the number of partitions used. This process is outlined in Chapter VI.

I.3 Thesis Outline

Chapter II introduces Modelica and Simulink as two different approaches to compositional modeling of physical systems. Chapter III presents a high level overview of the steps that need to be taken to translate a declarative model to a computational model. Chapter IV reviews some of the topics related to the simulation of a continuous system. This includes a review of the interaction between the system of equations and the solver, a review of the types of solvers available, and a review of previously published work on simulation parallelization. Chapter V describes the relevant aspects of the multi-core CPU architecture we will use to perform our simulation experiments. Chapter VI describes our parallel simulation algorithms and our experiment results. Finally, chapter VII presents our conclusions and future work.

CHAPTER II

COMPONENT-BASED MODELING

Component-based modeling describes the behavior of a system based on the behavior of composed individual components. Each of these components may in turn be created from multiple smaller components. This gives the final model a deep hierarchy of interacting components. Component-based modeling depends on two conditions to guarantee that the final model behaves as expected: compositionality and composability. Compositionality means that system level properties can be derived from local properties of the components. Composability means that the component properties do not change as a result of interactions with other components [101]. These two conditions allow component-based modeling to manage complexity and contain costs during the design process. The task of composing component models to construct systems models depends on the semantics of the component definitions and the interfaces that the component models expose to the rest of the system [94] [104]. Modeling the behavior of the components requires a separate modeling approach that accurately describes the physical behavior of the component [104].

This chapter reviews two different paradigms for modeling physical components: Modelica and Simulink. Modelica is a pure object-oriented equation based modeling language. Simulink describes the system equations structurally in causal form. Both of these methods supports the ideas of compositionality and composability. In the following sections, we briefly describe the two environments, and provide a description of their interface semantics. We also use an example component model of an electric motor and load to illustrate the features of the two environments. A flattened schematic of the motor model is shown in Figure 2. The model is broken into three components: the electric source, the motor, and the rotational load.

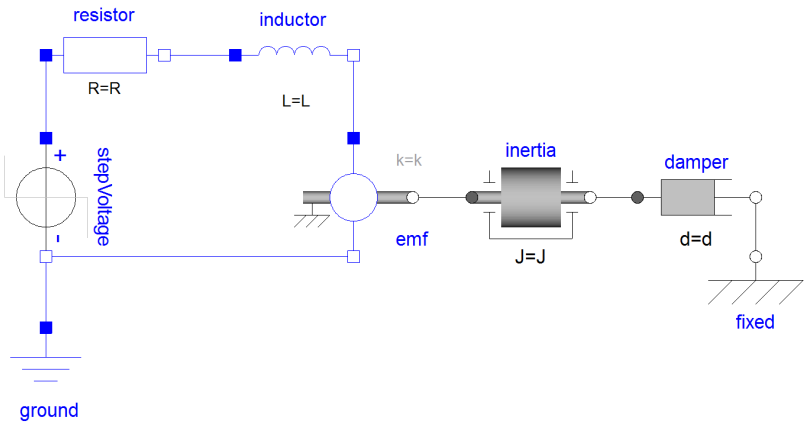


Figure 2: A flat model of a simple electric motor and load.

II.1 Modelica

Modelica is a programming language for describing the behavior of physical systems [14]. At its core, it is an acausal, object oriented, and equation based language for describing physical models using differential, algebraic, and discrete equations. The language supports hierarchy, model inheritance, and traditional programming language structures such as if-statements, case statements, loops, built-in primitive variable types, and parameters.

Hierarchical modeling is an intrinsic part of the Modelica language. The language supports model composition through model terminal variables that are exposed through a connector object. The connectors in Modelica are themselves primitive acausal models and are essentially simple container structures with a defined set of variables. When constructing a model, only connectors that have the same set of variables may be connected. This helps to ensure model correctness by avoiding crossover between physical domains. Since Modelica is an equation based language, it is also possible to connect models by linking variables from multiple models in an equation.

The acausal nature of Modelica benefits component based model construction, because large complex models can be built from smaller pieces without having to guarantee that input and output relations between the components match. The different components of

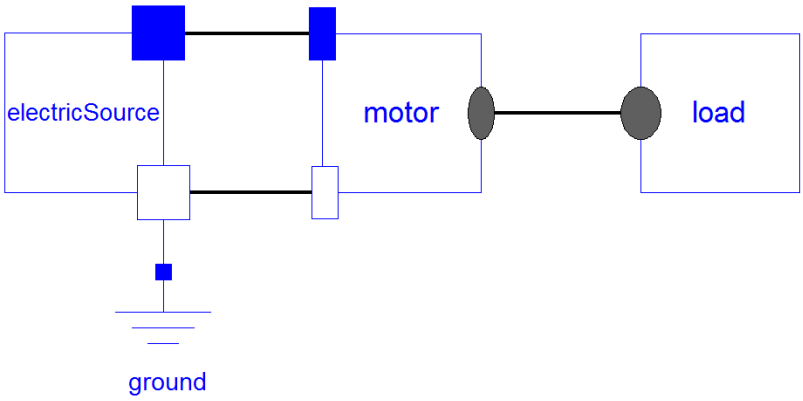


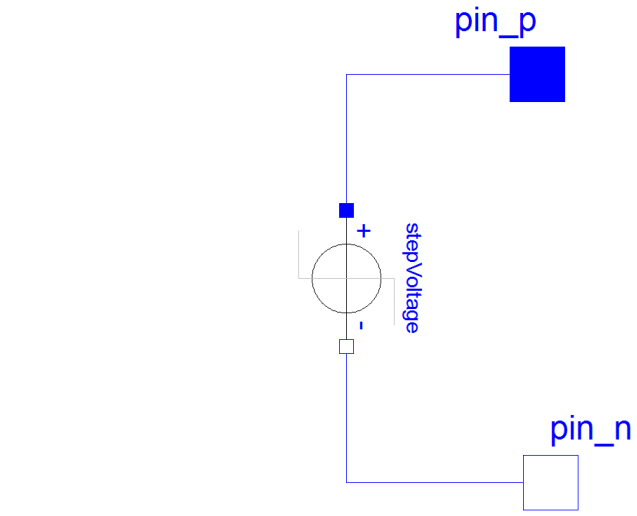
Figure 3: Hierarchical Modelica model of an electric motor.

a larger model are connected using the connectors, and the component behavior can be changed through model parameters.

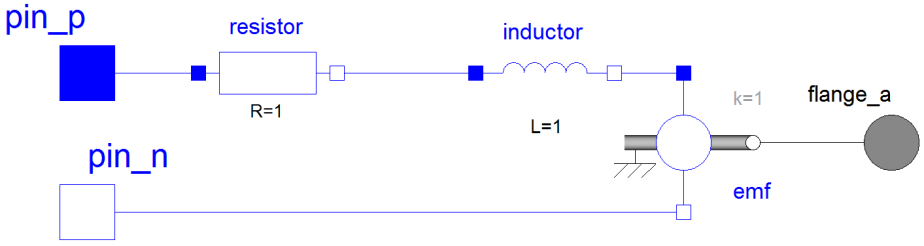
There are a number of different libraries available for Modelica that speed-up the model creation process. The most common library is the Modelica Standard Library (MSL). It has more than 1200 components [14] across multiple physical domains, with domain specific connectors for the components and is generally distributed with Modelica compilers. Other libraries include Power Systems, Bond Graphs, Buildings, Fuel Cells, and many others [14], including both free and commercial libraries.

An example hierarchical model of an electrical motor and load created using the Modelica Standard Library is shown in Figure 3. The electrical terminals are displayed as blue or white boxes. The mechanical rotational terminals are shown as gray or white circles. Only terminals of the same domain may be connected together. The different components are connected through acausal connectors. The equations describing the behavior of each of the detailed components are listed in the individual components.

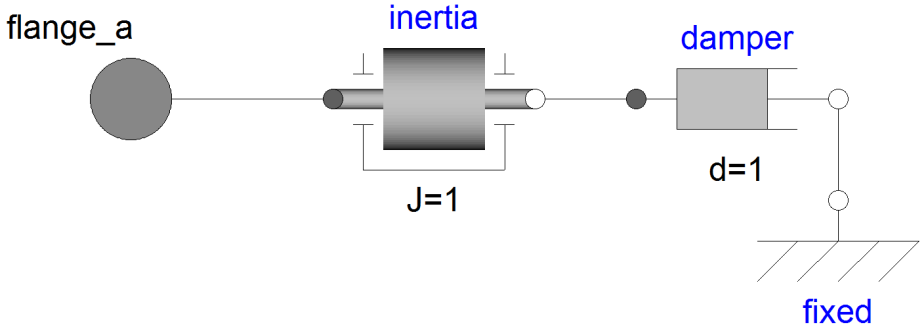
Transforming any declarative Modelica model into a simulation model generally involves two broad tasks and requires a dedicated model compiler [56]. The first task is to flatten the model hierarchy so that the model equations are at a single level. This is a very difficult task for complicated models due to the inheritance tree, and other language constructs such as replaceable models [56] [57]. The second task of a compiler is to transform



(A: Electric source)



(B: Simple motor)



(C: Rotational load)

Figure 4: Modelica component details.

the flattened equations into a form that can be simulated to generate behavior trajectory data. This second step will be covered in Chapter III.

There are a number of commercial and open source Modelica compilers and design environments. Commercial tools include Dymola [4] and MapleSim [12]. Open source tools include OpenModelica [15] and JModelica [10].

II.2 Simulink

Matlab Simulink from MathWorks [13] uses block diagrams to describe a causal model of the physics of the system. It is effectively a graphical programming language for creating ordinary differential equations (ODE) and represents a midpoint between the structural approach used in bond graphs [63] and the pure equation approach used in Modelica. The different blocks are connected using directed signals. Connecting the different blocks describes the mathematical behavior of the system. Simulink has a large library of standard functions and objects, it is extensible with user created blocks, and it has good support for integrating with external tools, such as hardware in the loop simulation, external C code, and others. As an example of Simulink's flexibility, the leading Modelica compiler, Dymola [4], integrates with Simulink to perform real time hardware-in-the-loop simulation [28].

Simulink models can be embedded inside other Simulink models. This allows for a natural hierarchy to be created that does not have an effect on the model simulation. The different components are connected together using directional links that point from the output of one component to the input of another. Information is exposed to higher levels using dedicated input and output blocks. Since basic Simulink models consist entirely of mathematical signals with no domain information, there is no automatic check on the ports of the components to make sure that the quantities being connected are from the same physical domain. It is entirely up to the modeler to guarantee that the proper systems are connected. The only requirement is that an output port connect to an input port.

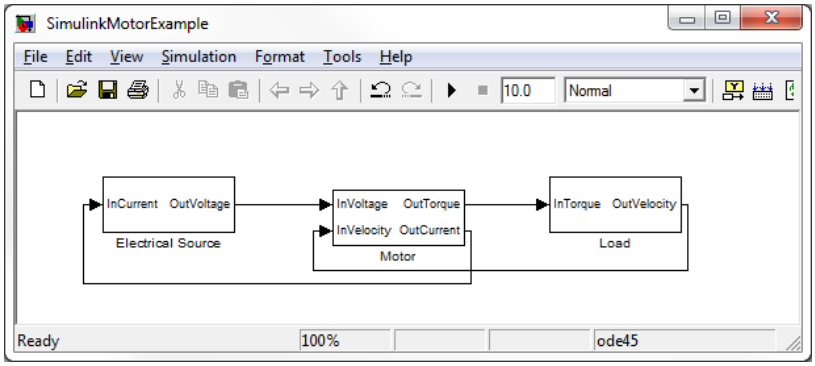
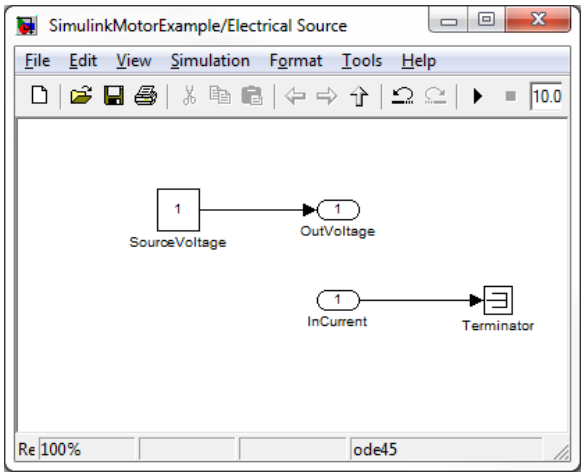


Figure 5: Simulink component model of an electric motor and load.

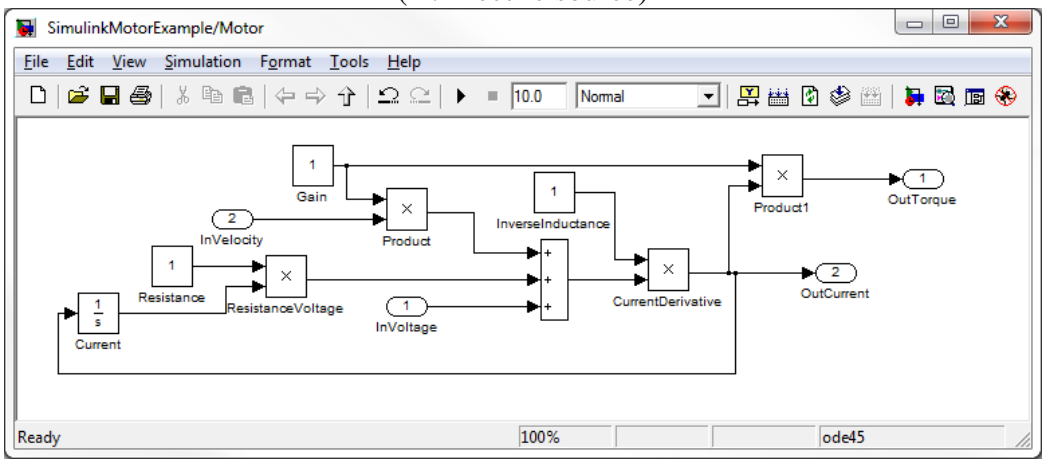
The library support for Simulink is very strong, with a number of libraries available from MathWorks. One of the key libraries for our interests is the Simscape library which adds support for acausal modeling, similar to Modelica and the MSL, provides an environment where domain specific connectors are respected [13], and provides a simple primitive component library. Beyond the Simscape library, Simulink can include a variety of other libraries for modeling control systems, neural networks, computer vision, digital signal processing, block diagrams, and others [13]. Some of the libraries are included with Simulink, and others require an extra license purchase.

A component model of the example electric motor and load in Simulink is shown in Figures 5 and 6. The Simulink representations of each of the components reduces to a graphical description of the mathematical equations that describe the behavior of the components. The interconnections between the components are through dedicated input and output ports.

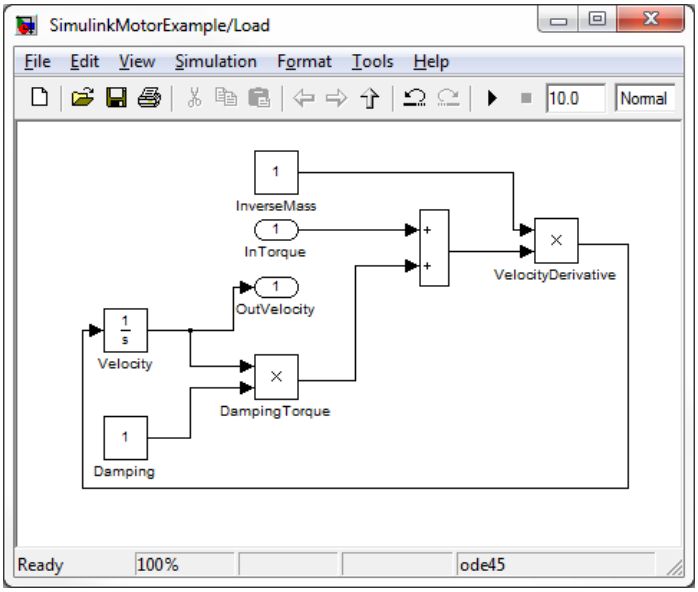
When the model is compiled for simulation, the Simulink compiler flattens the model structure and all hierarchy is removed [13]. After flattening, the model equations are transferred to a form suitable for simulation, and behavior trajectory is generated.



(A: Electric source)



(B: Simple motor)



(C: Rotational load)

Figure 6: Simulink component details.

II.3 Critical Review

Modelica is quickly becoming an industry standard tool for modeling complex physical systems [14], and Simulink is widely used to develop control systems where its causal structure works very well with the input/output formulations used for control systems [13]. The Modelica language is more flexible than Simulink, as there are very few built in limitations making it almost as expressive as general purpose programming languages, such as Java and C++. Simulink presents a graphical environment for describing a causal form of the system equations using a block diagram format. Since it is causal, it forces the task of causalizing the equations on the modeler, whereas Modelica leaves this task to the model compiler. The primitive component libraries found in Modelica and Simulink also allow for quick development of a description of a physical system. The Modelica Standard Library (MSL) [14] has more than 1200 components across multiple domains that can be connected through domain specific connectors. The SimScape library in Simulink, which includes first-principle components, SimDriveline, SimMechanics and others as sub-libraries, is similar to the MSL and Modelica in that it allows acausal modeling, something which is not allowed as a part of standard Simulink [13]. There are additional libraries for Simulink Some of the different properties of the design paradigms and the support provided by each paradigm is shown in Table 2.

For this thesis, our research will focus on improving the simulation of acausal equation based models of the behavior of physical systems. Modelica is the most suitable method for modeling physical system behavior, and therefore, the work in this thesis will primarily be performed using a sub-set of the Modelica language, but the general principles will apply to any equation-based language, including Simulink.

Feature	Modelica	Simulink
Acausal Modeling	Yes	Yes*
Domain Specific Connections	Yes	Yes*
Component Hierarchy	Yes	Yes
Component Library	Yes	Yes

*These are only available when using the SimScape library [13].

Table 2: Factors in designing complex systems

CHAPTER III

FROM DECLARATIVE TO COMPUTATIONAL MODELS: THE MODELICA APPROACH

Translating a Modelica model to an executable simulation model involves a number of steps. In general, the translation process is divided into two separate compilation processes: one executed by a frontend compiler and the second by a backend compiler [56]. The frontend compiler is responsible for reducing the hierarchical declarative model into a flattened set of equations. The backend compiler is responsible for translating the flattened set of equations into a computational form that is suitable for simulation. The algorithms in this chapter are discussed at a high level, and a more detailed description of many of these algorithms is presented in [33] and other sources.

Section III.1 of this chapter starts with basic graph definitions that are useful for representing the structure of the system equations. Section III.2 introduces an example that will be used to illustrate the operations of the algorithms discussed in the rest of this chapter. Section III.3 briefly discusses the approach for reducing the hierarchical declarative model to a flattened set of equations. This can be an involved task [56] because of the variety of model construction options available in the Modelica language. Section III.4 presents an overview of the initial steps in translating the flattened set of equations to a computational model for simulation. In Section III.5 a procedure for transforming the set of equations and variables into a partially ordered graph is described. The translation procedure applies a series of graph transformations to the model equations to create a computational model of the system. Section III.6 describes algebraic loops and how they affect the computational model. Finally, Section III.7 presents a critical summary of the methods presented in this chapter. Chapter IV goes into more details on the computational approaches to simulating a system model.

III.1 Definitions

The physical systems we will be working with are continuous dynamic systems, and are modeled using ordinary differential equations (ODE) or differential algebraic equations (DAE). ODE models are represented mathematically in the state equation form

$$\dot{\mathbf{x}}(t) = \mathbf{f}_x(t, \mathbf{x}(t), \mathbf{w}(t)). \tag{III.1}$$

One important point to note from equation III.1 is that the calculation of any value in $\dot{\mathbf{x}}(t)$ does not depend on any other values of $\dot{\mathbf{x}}(t)$ because no value of $\dot{\mathbf{x}}(t)$ appears as inputs to the function $\mathbf{f}_x$. This means that the calculation of the specific values in $\dot{\mathbf{x}}(t)$ can happen in any order.

DAEs are modeled in the form

$$\mathbf{F}_x(t, \mathbf{x}(t), \dot{\mathbf{x}}(t), \mathbf{w}(t)) = 0 \tag{III.2}$$

$$\mathbf{F}_x = \{F_1, \ldots, F_{n_x+n_w}\} : \mathbb{R}^{2n_x+n_w+1} \rightarrow \mathbb{R}^{n_x+n_w} \tag{III.3}$$

where $\mathbf{x}(t)$, the state variables of the system, are the set of differentiated variables, $\dot{\mathbf{x}}(t)$ is the set of derivatives of the state variables, $\mathbf{w}(t)$ is the set of algebraic variables, which includes any input variables, n_x is the number of state variables, and n_w is the number of algebraic variables [77]. The vectors $\mathbf{x}(t)$ and $\dot{\mathbf{x}}(t)$ have the same dimensions equal to n_x. The functions of $\mathbf{F}_x$ represent the equations describing the behavior of $\mathbf{w}(t)$ and $\dot{\mathbf{x}}(t)$. Solving the functions in $\mathbf{f}_x$ or $\mathbf{F}_x$ is called a function evaluation.

Definition 1 (Function Evaluation)**.** *A function evaluation is one evaluation of the vector function* ***f*** *from Equation III.1 or vector function* ***F*** *from Equation III.3.*

There is one equation for each variable that is not a state variable, that is, one equation for each element in the sets $\dot{\mathbf{x}}(t)$ and $\mathbf{w}(t)$, which makes the system just determined. At

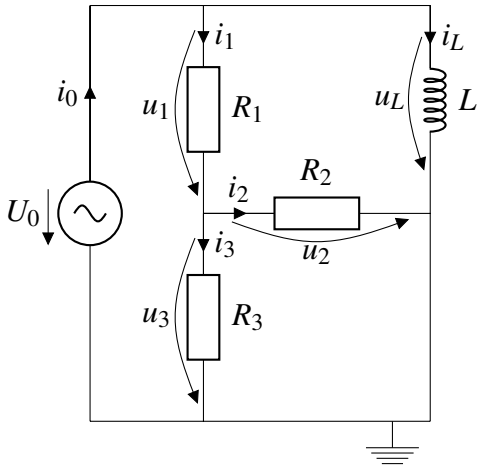


Figure 7: Circuit example

time point, t_i, the state variable derivatives, $\dot{\mathbf{x}}(t_i)$ are integrated using a numerical integration method to calculate the values of the state variables at time $\mathbf{x}(t_{i+1})$. This interaction between $\dot{\mathbf{x}}(t)$ and $\mathbf{x}(t)$ relate to the structure of a simulation, and is reviewed in Section IV.1. Integration methods are reviewed in Section IV.2.

III.2 Example RL Circuit

To illustrate the steps involved in translating a declarative model to a computational model, we will analyze the circuit shown in Figure 7 (this example circuit and equations are taken from [33]).

This circuit has 5 components: a voltage source, three resistors, and an inductor. There are 10 unknown quantities: the voltage drop and the current through each of the five components. Since there are 10 unknown quantities, there are 10 equations associated with the

circuit. Five of the equations are from the component behavior equations:

$$(1) \quad u_0 = f(t)$$

$$(2) \quad u_1 = R_1 i_1$$

$$(3) \quad u_2 = R_2 i_2$$

$$(4) \quad u_3 = R_3 i_3$$

$$(5) \quad u_L = L\frac{di_L}{dt}.$$

The voltage across the voltage source, u_0, is given by a forcing function $f(t)$, which is considered to be a known input to the system. The other equations in the system represent the connection equations between the components:

$$(6) \quad u_0 = u_1 + u_3$$

$$(7) \quad u_L = u_1 + u_2$$

$$(8) \quad u_3 = u_2$$

$$(9) \quad i_0 = i_1 + i_L$$

$$(10) \quad i_1 = i_2 + i_3$$

Equations 6-8 are derived from Kirchhoff's Voltage Law, and equations 9 and 10 are derived from Kirchoff's Current Law. The set of pure algebraic variables is $\mathbf{w}(t) = \{u_0, u_1, u_2, u_L, i_0, i_1, i_2, i_3\}$. The set of state variable derivatives is $\dot{\mathbf{x}}(t) = \{\frac{di_L}{dt}\}$, and the set of state variables is $\mathbf{x}(t) = \{i_L\}$. The equations 1-10 above correspond to the set $\mathbf{F}_x$. Also since the variable i_L is a state variable and its value is calculated using a numerical integration method and not directly from the model equations, it is treated as a constant in equations 1-10.

III.3 Model Flattening

As described in Section II.1, Modelica models can have a hierarchy of components, where each component is composed of multiple other components, and each of those sub-components can be composed of multiple components. This top level hierarichical model is called the declarative model. The hierarchy needs to be removed so that the system equations, $\mathbf{F}_x$, can be transformed into a form suitable for simulation. The process of removing the model hierarchy is called flattening, and is the first step of the compilation process.

Definition 2 (Declarative Model)**.** *The declarative model is a hierarchical model created using the Modelica modeling language in a commercial or open source modeling tool.*

Flattening can be a complex process due to the flexibility of the Modelica modeling language, and is usually handled with a separate compiler than the rest of the model compilation process [56]. The flattening compiler is sometimes called a frontend compiler. We do not know of any published literature on the flattening process employed by a Modelica compiler, and the following description is based on our understanding of the Modelica language and the task assigned to a flattening compiler. Some of the tasks the flattening compiler is responsible for is reducing model hierarchy, resolving model inheritance, and resolving any replaceable functions or models. As an example of what needs to happen during the flattening process, we will review the Modelica Standard Library components used to build the circuit from Figure 7.

In Modelica there can be an extensive inheritance tree for each model as Modelica allows multiple inheritance. Consider the resistor from the Modelica Standard Library (MSL). The inheritance diagram for the resistor is shown in Figure 8 and shows that the MSL resistor inherits from OnePort and ConditionalHeatPort. In Modelica the default inheritance is public, so all of the variables and components in OnePort and ConditionalHeatPort are available to be used by MSL resistor. This inheritance structure is a form of hierarchy that needs to be removed during model flattening.

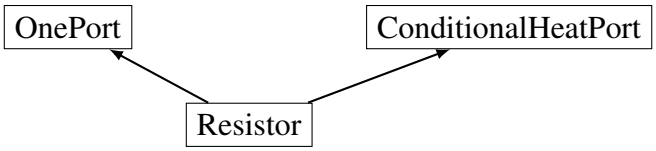


Figure 8: Inheritance tree for MSL resistor.

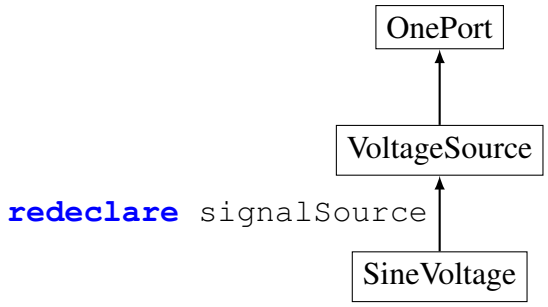


Figure 9: Inheritance tree for MSL SineVoltage.

Replaceable functions and models allow a model to be defined with one component, but when that model is compiled the replaceable component can be replaced with a different component. As an example, consider the voltage source from Figure 4. The inheritance tree is shown in Figure 9, and the model description code is shown in Figure 10. In the model description code the **`extends`** keyword indicates inheritance, and the **`redeclare`** keyword indicates that the `Modelica.Blocks.Sources.Sine` component should replace the model that defines the signalSource component in the parent model. The replaceable models are resolved at compile time so that the flattened set of equations represents what was described in the source model.

As a second example, consider the Modelica model in Figure 3. It includes three components: electric source, motor, and load. Each of those components has a number of subcomponents. In Figure 4 the electric motor contains a resistor, inductor, and emf components, and the necessary domain specific terminals. Each of the sub-components has equations, variables and other components that can describe its behavior. All of these layers of hierarchy will need to be removed during the flattening process.

In Modelica, variables within a component are referenced using dot notation, similar

```
model SineVoltage "Sine voltage source"
  parameter SI.Voltage V(start=1) "Amplitude of sine wave";
  parameter SI.Angle phase=0 "Phase of sine wave";
  parameter SI.Frequency freqHz(start=1) "Frequency of sine wave";
  extends Interfaces.VoltageSource(redeclare Modelica.Blocks.Sources.Sine
      signalSource(
      amplitude=V,
      freqHz=freqHz,
      phase=phase));
end SineVoltage;
```

Figure 10: Code defining the MSL sine voltage component.

to C++ and other programming languages, in the form component_name.variable_name. For example, the flattened equation describing the electrical behavior of the resistor from Figure 3 is:

```
motor.resistor.v=motor.resistor.R_actual*motor.inductor.i.
```

This naming structure is preserved in the flattened set of equations, and may provide a means for providing feedback to the system modeler.

Flattening the model involves resolving model inheritance, composition, and handling the replaceable functions and models. The output of the flattening compiler is a completely flat set of equations without any hierarchy, save for what is in the variable names, that is ready to be manipulated by the backend compiler and prepared for simulation.

III.4 Preprocessing: Initial Simplifications

After the model is flattened, the first steps in translating a declarative model to a computational simulation model relate to reducing the equation structure to its simplest possible form. This involves removing unnecessary equations and variables, combining equations and functions to avoid unnecessary computation, and substituting variables that have constant values with the value to reduce the number of variables. Other steps that are typically

executed to reduce the number of variables and simplify equations, such as scalarisation and function inlining, are presented in [95].

Any equations of the form $a = b$ and $b = c$ are simplified so that the variable a replaces all instances of b and c, and the original equations are removed from the system [56] [95]. In this example the number of equations and variables are reduced by 2 because all instances of variables b and c are replaced by variable a, and the equations equating a with b and c are no longer needed. In the circuit example equations above, equation 8 can be removed, and all instances of u_2 can be replaced by u_3, or vice versa, which reduces the number of equations and variables by 1. (Note that in the following descriptions equation 8 is not removed.) This reduction simplifies the causalization and simulation tasks, because there are fewer equations and variables to process.

Identical function calls are also simplified so that the function is called only once [81]. For example, there can be two equations such as $a = f(b)$ and $c = f(b)$. These equations would be modified so that the function $f(b)$ is called only once. This speeds up system computation by avoiding duplicate work.

Another simple step is to replace constant variables with their assigned value [81] [95]. This reduces the number of variables the system needs to track, which also leads to a faster simulation.

These simple steps strip out unnecessary equations and variables, and prepare the system $\mathbf{F}_x$ to be causalized and the system transformed into a computational model, which is a partial order in which the equations can be solved.

III.5 Transforming Equations to a Computational Model

After the model has been flattened, and the trivial equations stripped out, the model is ready for transformation to the computational model. The transformation involves applying a series of graph transformations to the equations. These steps will be listed here, and then expanded in the following sections.

Definition 3 (Computational Model)**.** *The computational model of a system is a mathematical description of the system describes the same behavior as the declarative model but is tailored for simulation.*

The first step in creating the computational model is to construct a bipartite graph of the equations, the functions in $\mathbf{F}_x$, and variables from $\dot{\mathbf{x}}(t)$ and $\mathbf{w}(t)$, and add a link from the variables to each equation in which they appear. Next the equations are causalized, that is, assigned to solve for one of the variables in the equation, using a maximum flow algorithm from network theory. The bipartite graph is then transformed into a directed graph by merging the causal links into a single node, and transforming the non-causal links into directed arcs. Any loops in the directed graph indicate systems of algebraic equations, and the individual nodes that make up the loop are collapsed into a single node. This transforms the directed graph into a directed acyclic graph (DAG), and the DAG represents the computational model that partially describes the order in which the equations must be solved.

III.5.1 Bipartite Graph Creation

Bipartite graphs effectively represent the variables present in each of the equations in the model. The first step in creating the system computational model is to create a bipartite graph that identifies the variables in $\mathbf{w}(t)$ and $\dot{\mathbf{x}}(t)$ that are present in each equation $\mathbf{F}_x$.

Definition 4 (Bipartite graph)**.** *A bipartite graph is an undirected graph, $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, with vertices $\mathcal{V}$ and edges $\mathcal{E}$ where the vertices can be divided into two disjoint sets, $\mathcal{V}_1$ that represents the set of equations in the model, $\boldsymbol{F}_x$, and $\mathcal{V}_2$ that represents the variables in sets $\boldsymbol{w}(t)$ and $\dot{\boldsymbol{x}}(t)$ such that $(u,v) \in \mathcal{E}$ implies that either $u \in \mathcal{V}_1$ and $v \in \mathcal{V}_2$ or $u \in \mathcal{V}_2$ and $v \in \mathcal{V}_1$ [39].*

The bipartite graph for the circuit in Figure 7 is shown in Figure 11. It is important to remember that because there are the same number of equations and variables in the model,

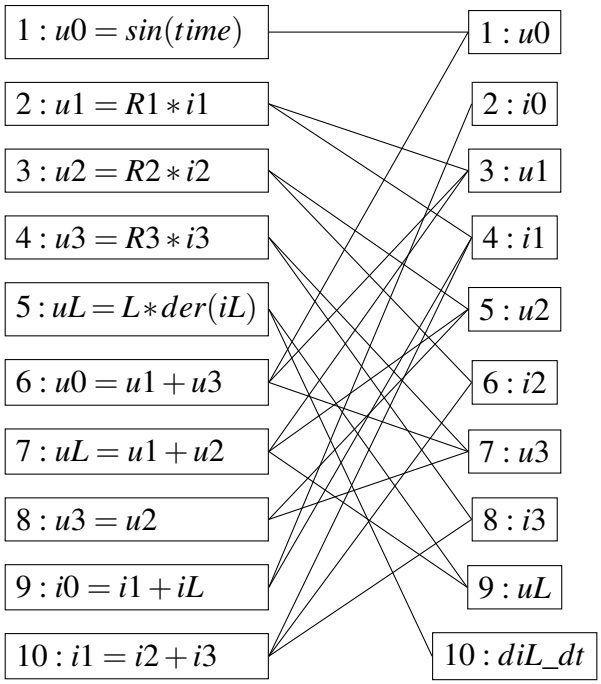


Figure 11: Bipartite graph for the circuit equations in Figure 7.

each set of vertices $\mathcal{V}_1$ and $\mathcal{V}_2$ has the same cardinality. There is a link between an equation and a variable if the variable appears in the equation.

III.5.2 Equation Causalization

After the bipartite graph is created, the equations can be causalized by using Ford/-Fulkerson's [51] or Edmonds/Karp's [44] maximum flow algorithm. Additional algorithms presented in [55], or the Augmenting Path algorithm from [77] may also be used. Causalizing a system of equations refers to assigning an equation to solve for each variable. It also allows us to define an order in which the system equations are to be solved, because any variables used in an equation must be calculated by a different equation before they can be used in this equation.

Definition 5 (Causal Equation)**.** *A causal equation is an equation that has n variables, where* $n-1$ *of the variables are assumed known, so that the value of the nth variable can be calculated.*

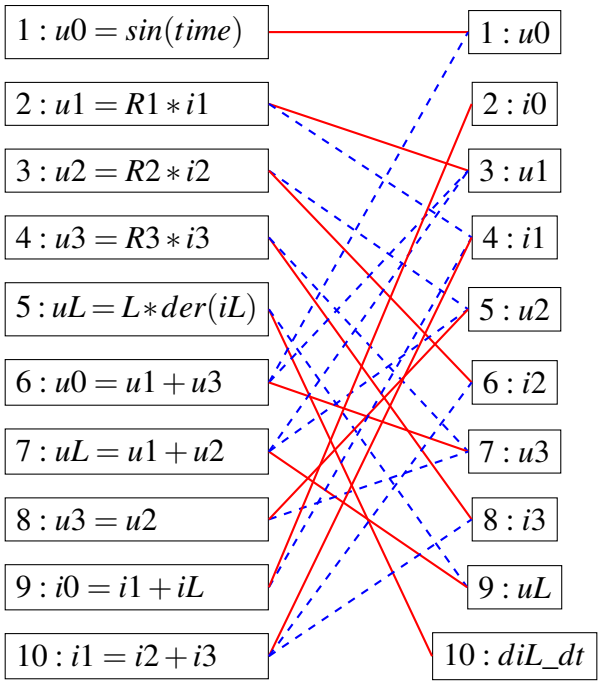


Figure 12: Causal equations

There is a one-to-one mapping of equations and variables, so that each equation solves for one and only one variable (see Section III.1) . The causalized bipartite graph for the circuit in Figure 7 is in Figure 12. The red solid lines are the causal equation/variable pairs and the blue dashed lines are the equations that use the variable, but do not solve for it.

III.5.3 Directed Graph and Directed Acyclic Graph Creation

Next, a directed graph is created from the bipartite graph by converting each non-matching edge into a directed edge pointing from the variable to the equation in which it is used (note that this is backwards from how the graph is constructed in [77] [32]). Then each equation-variable matching pair is collapsed into a single node. Collapsing these nodes leads to self loops on each of the vertices in the directed graph, these are removed. The directed graph describes the order in which the equations need to be solved to fulfill the requirements of the equation causalization.

Definition 6 (Directed Graph)**.** *A directed graph is a graph, $\mathscr{G}(\mathscr{V},\mathscr{E})$, where the vertices*

represent causal equations, and where the edges $\mathscr{E}$ are directed from one vertex v_1 to a second vertex v_2 and represent a precedence relationship between the vertices [77].

The directed graph showing the order in which the equations and variables need to be solved is in Figure 13.

In the directed graph there may be loops where, for example, node A is dependent on node B, and node B is dependent on node A. Figure 13 has a loop where equation 6 depends on equation 2, and equation 2 depends on equation 6 through equations 8, 4, 3, and 10. Each of these equations is reachable from each of the other equations by traversing the links in the graph. This indicates that the model has an algebraic loop.

Definition 7 (Algebraic Loop)**.** *An algebraic loop is a set of equations that need to be solved simultaneously.*

The loops in the directed graph are called strongly connected components.

Definition 8 (Strongly Connected Component)**.** *A Strongly Connected Component (SCC) is a subset of a graph where the vertices (equations) are mutually reachable, that is, every equation the subset can reach every other equation in the subset by traversing the directed arcs within the subset [39].*

Each SCC represents one or more equations that solve for one or more variables. A single equation is considered a SCC because each node in the directed graph has a self loop, allowing each node to reach itself (however, this self loop is ultimately removed). If a SCC has more than one equation, then it represents an algebraic loop. Since the algebraic loops require that all of the equations in the loop be solved simultaneously, not in sequence like a single equation, they need to be identified and treated as a unit. Tarjan's algorithm [102] is used to identify the SCCs. The SCCs representing algebraic loops are collapsed into a single vertex, removing any arcs between the equations of the SCC. These vertices in Figure 14 that represent the algebraic loop are collapsed into a single vertex, identified as

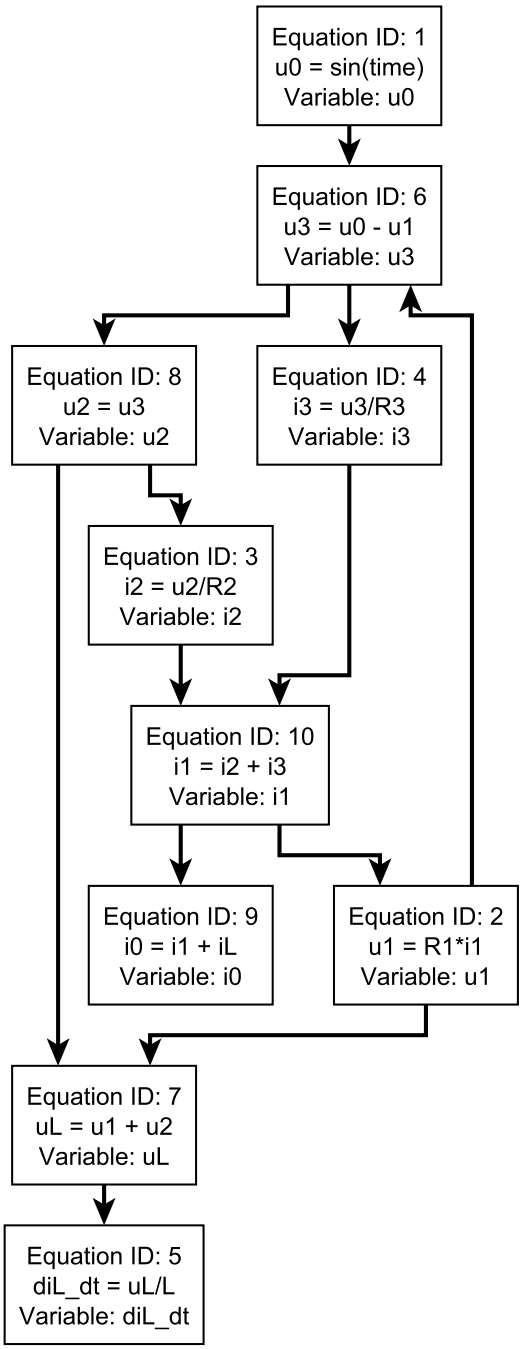


Figure 13: Directed graph of equations from Figure 7

block 4 in Figure 13. Collapsing the SCCs down to a single vertex transforms the directed graph into a directed acyclic graph.

Definition 9 (Directed Acyclic Graph)**.** *A Directed Acyclic Graph (DAG) is a directed graph where, during graph traversal, it is not possible to reach a vertex more than once.*

The term SCC is used to describe a node in the DAG that represents one or more equations. The term algebraic loop is used to specifically refer to multiple equations that must be solved simultaneously. The terms block and task refers to any node in the DAG, and we usually use the terms in the context of some computational work that needs to be performed. The DAG is used to describe the partial order in which the equations need to be solved, and is the final form representing the computational model of the system. Each vertex in the graph represents one equation, or a group of n equations, that need to be solved for one, or n variables, and the dependency relations between the vertices indicate the order in which the equations need to be solved.

These steps are also part of the Dulmage-Mendelsohn (DM) Decomposition [42]. However, the DM Decomposition is more general because it is able to identify overdetermined and underdetermined parts of the graph, the process outlined above assumed the graph is perfectly determined.

III.5.4 Convert to ODE Form

As will be described in Chapter VI, our simulations will be based on the set of ODEs that define the behavior of the model. To convert the model to ODE form, we simply begin to reduce the DAG levels by substituting the equation(s) in a parent node into the child node equation(s). In Figure 14, block 5 calculates the value of $u0$ from $sin(time)$. Block 4, the child node of block 5, uses the variable $u0$ in one of its equations. The reduction will simply substitute $sin(time)$ in for $u0$ in the correct equation in block 4, and then delete block 5 because it is no longer needed. A similar process can happen between block 2 and block 1 in Figure 14. Block 2 calculates uL from $u1 + u2$. Block 1 uses uL to calculate diL_dt. We

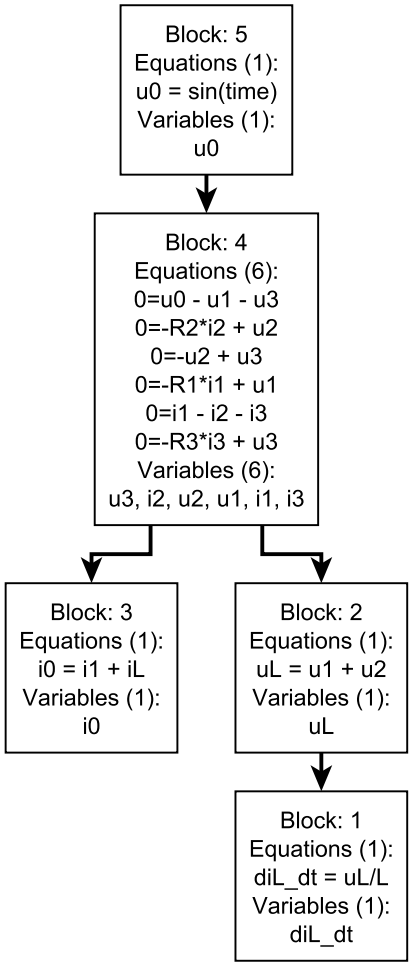


Figure 14: Directed acyclic graph of equations from Figure 7

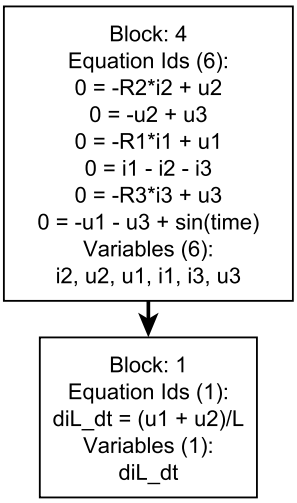


Figure 15: Final computational model of equations from Figure 7

can substitute $u1 + u2$ into the equation in block 1, remove block 2, and establish a parent-child relationship between block 4 and block 1. Also, since ODEs are strictly concerned with calculating the derivatives of state variables, we can remove block 3, because it plays no part in calculating the value of diL_dt. The algebraic loop will still remain in the set of state equations, and the ODE will need to be solved in 2 steps. The first step will solve the algebraic loop, and the second step will calculate the value of diL_dt. The final graph form of the ODE that will be used for the simulation is shown in Figure 15. If this example did not have an algebraic loop, then the final graph form would be a single layer of nodes, and each node would calculate the value of one state variable derivative.

III.5.5 Compilation Summary

In summary, the bipartite graph is used to causalize the system of equations and variables using one of a number of algorithms. After the system is causalized, the bipartite graph is transformed into a directed graph. Tarjan's algorithm is then used to find the strongly connected components, which are systems of equations that need to be solved simultaneously. The vertices in the SCCs are combined into a single vertex per SCC, this

transforms the directed graph into a DAG. The DAG of SCCs represents the computational model of the system, as it indicates the partial order in which the equations need to be solved. The next Section describes unique features of the model equations that require special processing during the creation of the computational model.

III.6 Algebraic Loops

Algebraic loops occur when a group of equations need to be solved simultaneously instead of sequentially. They are a result of the modeler applying an execution sequence to a physical phenomena that is not sequential. For example, two electrical resistors in series will cause an algebraic loop in the system of equations that represent the circuit behavior. Essentially, they are artifacts of an attempt to represent a non-causal system, i.e. the physical world, as a series of cause-and-effect relationships, i.e. equations [34]. These types of structures require extra attention during the creation of the system dependency graph, and there are a number of approaches to solving them. This section will cover the Newton Iteration approach, which is an iterative approach to solving algebraic loops. There are approaches to break algebraic loops that can reduce the number of equations in the loop that include equation tearing and the relaxation algorithm, detailed in [33] and [77]; however, we do not plan to implement these approaches, so they are not covered here.

III.6.1 Newton Iteration

The Newton Iteration is a zero-crossing algorithm [35]. A zero-crossing equation is created for the equation in question, and added to the system equations. Then the algorithm is applied iteratively until the variable in question is found to be within a specified tolerance. In the scalar case, when the equation to be iterated can be described as

$$f(x) = 0, \quad f : \mathbb{R} \to \mathbb{R}, \tag{III.4}$$

the Newton Iteration takes the form

$$x^{\ell+1} = x^{\ell} - f(x^{\ell})/\dot{f}(x^{\ell}). \tag{III.5}$$

This is essentially drawing tangents of the function $f(x)$ and determining where they cross the x-axis [66]. If the set of equations that represent the algebraic loop are described by

$$f(x) = 0, \quad f : \mathbb{R}^m \to \mathbb{R}^m, \tag{III.6}$$

then the Newton Iteration is

$$x^{\ell+1} = x^{\ell} - J^{-1}(x^{\ell}) f(x^{\ell}), \quad \ell = 0, 1, 2, \ldots \tag{III.7}$$

where $J(x) = (\partial f/\partial x)(x)$ is the Jacobian matrix of f with respect to x, and the ℓ superscript is the iteration number [66]. The iteration proceeds until two iterations are within a tolerance, that is $x^{\ell+1} - x^{\ell} < \varepsilon$. For linear systems this algorithm will converge in a single step [33] [77]. For non-linear algebraic loops the work to solve the algebraic loop is approximately $O(n^3)$ complexity where n is the number of unknowns [28] [45].

It is important to note that the Newton Iteration does not guarantee global convergence, only local convergence, and it is therefore very dependent on the initial guess of x^{ℓ}. If the iteration does not converge, then the simulation can either stop and display an error, or it can try a new guess and repeat the iteration with that new guess [66].

III.7 Summary

This chapter is primarily intended as a background chapter for translating a declarative Modelica model to a computational model. The first step is to simplify the model by eliminating the most simple equations. Then the declarative model is transformed into a

computational model by applying a series of graph transformations and equation substitutions. Once these algorithms are applied, the model equations are in a form that is ready for simulation. Also, briefly covered is a method for solving algebraic loops.

The DAG representation of the model equations is very appropriate to our task of creating a parallel simulation, as described in Chapter I. Working from the DAG form of the model we can partition the system based on the equations used to calculate the state variable derivatives, which will be at the bottom of the graph, and work up the graph to identify independent sub-components of the graph that can be solved in parallel. We can also parallelize the ODE form of the model, where each equation that calculates a state variable derivative, as described in Section III.1, is independent of every other equation that calculates a state variable derivative. This allows us to group the equations according to the needs of the computational architecture. The computational architecture is described in Chapter V, and our parallel simulation algorithms are described in Chapter VI.

CHAPTER IV

SIMULATION METHODOLOGY

After the computational model is derived a simulation algorithm is applied to generate system behaviors. This chapter presents an overview of basic simulation algorithms for continuous system equations. Section IV.1 presents relevant definitions and a high level overview of how a simulation is structured. Section IV.2 presents an overview of numerical integration methods relevant to our work. Section IV.3 presents Inline Integration, a numerical integration technique that we will use in our experiments in Chapter VI. Section IV.4 presents a series of methods that have been used to parallelize the system of DAE equations that describe the model's behavior, and presents a critical review of the methods. Finally, this chapter concludes with a summary of the material presented.

IV.1 Simulation Structure

The goal of a simulation is to generate time trajectory data for all of the variables in the dynamic system model. The high-level process is shown in Figure 16. The simulation is performed in discrete time, so the relevant variables are functions of the simulation step, $\mathbf{x}_n$, instead of time, $\mathbf{x}(t)$. Each value of n represents a specific point in time, t. The value of t at the next time step, $n+1$, is $t+h$.

In modern simulation software the model compiler converts the declarative DAE model to explicit ODE form [34] and creates the computational model described in the previous chapter. The ODE set of equations calculate the values of the derivatives of the state variables, $\dot{\mathbf{x}}_n$. After the state variable derivatives are calculated, the simulation software passes the calculated derivatives to the chosen solver.

Definition 10 (Numerical Integration Method)**.** *The numerical integration method integrates the derivative of a variable at time t to determine the value of the corresponding state variable at time $t+h$, where h is the simulation step size.*

Definition 11 (Step Size)**.** *The step size of a simulation is the distance between two individual time steps, and, depending on the solver, it may be kept fixed or dynamically adjusted during simulation. It is described by the variable h in definition 10.*

Definition 12 (Solver)**.** *The solver is a stand alone piece of software that implements a specific numerical integration method, and, potentially, step size control, order control, and any other tasks necessary to accurately complete a simulation [67].*

The solver uses a numerical integration method to integrate the derivatives to find the new values of the state variables at the next time step, $\mathbf{x}_{n+1}$ or in continuous time representation $\mathbf{x}(t+h)$. The values are checked to guarantee that the results are within the prescribed error tolerance. If they are not then the solver takes an appropriate action, usually halving the simulation step size, and tells the simulation code to re-evaluate the previous time step with the smaller step size. If the state variables are within the defined tolerance, the new state variables are passed to the computational model which then calculates the next values of the state variable derivatives. The solver can also take the step of making the simulation step size larger if a small step size is no longer needed. This process continues until the simulation stop time is reached.

The two aspects of the simulation, shown in Figure 16, the computational model and the solver are generally two separate pieces of software. This allows different integration algorithms to be paired with the same model by simply changing a setting in the overall modeling environment.

IV.2 Integration Method Overview

There are a variety of integration methods available and each is appropriate for integrating different kinds of systems and situations. The different methods are generally divided

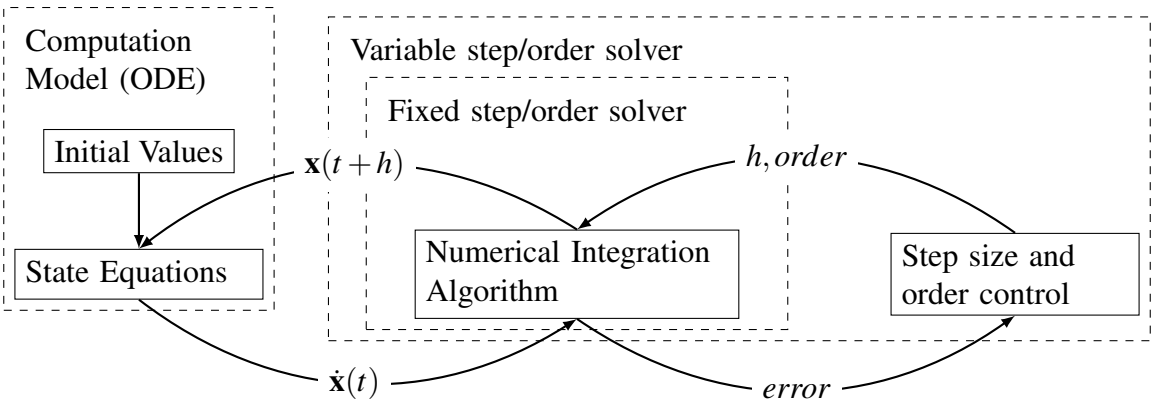


Figure 16: Simulation structure

into single-step and multi-step methods (the chapters of [36] and [68] are divided this way) and into explicit and implicit methods. The methods operate on a single state variable, so the following definitions are presented as scalars instead of as vectors.

One dividing line between integration methods is the number of state variable values from previous time steps that are used to calculate the next value of the state variable.

Definition 13 (Single-step Integration Method)**.** *Single-step integration methods calculate the next value of the state variable, x_{n+1}, as a function of the current value of the state variable, x_n and its derivative $\dot{x}_n$; they do not use values of x_n from previous time steps. That is:*

$$x_{n+1} = f(x_n, \dot{x}_n, h). \tag{IV.1}$$

Many single-step methods are also multi-stage methods, where the method calculates the values of x and $\dot{x}$ at micro-steps (stages) between t and $t+h$. The Runge-Kutta (RK) family of methods are all multi-stage methods and it is generally expected that a reference to a single-step method indicates a RK algorithm. RK methods can be described by:

$$x_{n+1} = x_n + h\sum_{i=1}^{s} b_i k_i \tag{IV.2}$$

where

$$k_i = f\left(t_n + c_i h, x_n + h\sum_{j=1}^{s} a_{ij}k_j\right), \quad i = 1,2,\ldots,s, \tag{IV.3}$$

where s is the number of stages in the method, and the parameters a, b, and c are specific to the integration method [69].

Definition 14 (Multi-step Integration Method)**.** *Multi-step integration methods calculate x_{n+1} from x_n, $\dot{x}_n$, and values of x from previous time steps. That is:*

$$x_{n+1} = f(x_n, \dot{x}_n, x_{n-1}, x_{n-2}, \ldots, x_{n-k-1}, h), \tag{IV.4}$$

where k is the order of the algorithm.

Linear multi-step methods assume that the time step, h, is constant across all of the previous time steps [70].

A second line dividing the algorithms is the separation between explicit and implicit methods. Example algorithms in each category are presented in Table 3.

Definition 15 (Explicit Integration Method)**.** *Explicit integration methods directly (explicitly) calculate the next value of the state variable, x_{n+1}, from the current value of the state variable, x_n using a single equation, or a series of equations applied in sequence; therefore, there are no algebraic loops.*

Usually explicit integration methods calculate x_{n+1} as a function of x_n and $\dot{x}_n$; that is:

$$x_{n+1} = f(x_n, \dot{x}_n, h). \tag{IV.5}$$

Definition 16 (Implicit Integration Method)**.** *Implicit integration methods calculate the next value of the state variable, x_{n+1}, using the derivative of the state variable computed at time step $n+1$, $\dot{x}_{n+1}$. Therefore, the integration method introduces an algebraic loop [67][35].*

	Explicit Method	Implicit Method
Single-Step	Forward Euler Runge-Kutta 4 (RK4) Dopri45	RadauIIA LobattoIIIA
Multi-Step	Adams-Bashforth	Backward Difference Formula

Table 3: Examples of different integration methods.

Implicit integration methods usually calculate x_{n+1} as a function of x_n and $\dot{x}_{n+1}$, that is:

$$x_{n+1} = f(x_n, \dot{x}_{n+1}, h). \tag{IV.6}$$

However, it is possible that one of the stages in a Runge-Kutta algorithm references itself, which also creates an algebraic loop and would therefore make the method implicit even though it does not reference $\dot{x}_{n+1}$. RK methods are explicit if they only reference previous stages of the method, and they are implicit if a stage references itself or future stages because this will create an algebraic loop.

The simplest integration method is the well-known explicit single-step Forward Euler (FE) method, which is defined as:

$$x_{n+1} = x_n + h \cdot \dot{x}_n, . \tag{IV.7}$$

The corresponding implicit single-step integration method is the Backward Euler (BE) method, which is defined as:

$$x_{n+1} = x_n + h \cdot \dot{x}_{n+1}. \tag{IV.8}$$

Because $\dot{x}_{n+1}$ is used to calculate x_{n+1}, this creates an algebraic loop that will need to be solved using a Newton Iteration (see Section III.6). It is a single-step method for the same reason as the Forward Euler method, because it does not use values of x from time steps before the current time step.

An example of a multi-stage explicit method is the explicit Runge-Kutta 4 method. The equations for the method are:

$$
\begin{aligned}
1^{st} \text{ stage: } k_1 &= f(x_n, t_n) \\
2^{nd} \text{ stage: } k_2 &= f(x_n + \frac{h}{2} \cdot k_1, t_n + \frac{h}{2}) \\
3^{rd} \text{ stage: } k_3 &= f(x_n + \frac{h}{2} \cdot k_2, t_n + \frac{h}{2}) \\
4^{th} \text{ stage: } k_4 &= f(x_n + h \cdot k_3, t_n + h) \\
\text{Final: } x_{n+1} &= x_n + \frac{h}{6} \cdot [k_1 + 2 \cdot k_2 + 2 \cdot k_3 + k_4]. \qquad \text{(IV.9)}
\end{aligned}
$$

These equations can be solved sequentially because each stage of the algorithm only uses information from the previous stages. To calculate the value of x_{n+1} the algorithm uses a weighted average of the derivatives calculated at each stage, and the weights assigned to each derivative are outlined in the final equation. There is no need to solve an algebraic loop to calculate the values of x_{n+1}, which indicates that this is an explicit algorithm. It only references current values of x_n, and does not rely on values from previous time steps, therefore the algorithm is a single-step algorithm. It is a multi-stage algorithm because there are 5 equations to be solved at different fractions of the time step h in order to arrive at the solution of x_{n+1}.

Another example of a multi-stage explicit method is the explicit Runge-Kutta-Fehlburg 4-5 method. The equations for the method are:

$$
\begin{aligned}
&1^{st} \text{ stage: } k_1 = f(x_n, t_n) \\
&2^{nd} \text{ stage: } k_2 = f\left(x_n + \frac{h}{4}\cdot k_1, t_n + \frac{h}{4}\right) \\
&3^{rd} \text{ stage: } k_3 = f\left(x_n + \frac{h\cdot 3}{32}\cdot k_1 + \frac{h\cdot 9}{32}, t_n + \frac{h\cdot 3}{8}\right) \\
&4^{th} \text{ stage: } k_4 = f\left(x_n + \frac{h\cdot 1932}{2197}\cdot k_1 - \frac{h\cdot 7200}{2197}\cdot k_2 + \frac{h\cdot 7296}{2197}\cdot k_3, t_n + \frac{h\cdot 12}{13}\right) \\
&5^{th} \text{ stage: } k_5 = f\left(x_n + \frac{h\cdot 439}{216}\cdot k_1 - 8\cdot h\cdot k_2 + \frac{h\cdot 3680}{513}\cdot k_3 - \frac{h\cdot 845}{4104}\cdot k_4, t_n + h\right) \\
&6^{th} \text{ stage: } k_6 = f\left(x_n - \frac{h\cdot 8}{27}\cdot k_1 + 2\cdot h\cdot k_2 - \frac{h\cdot 3544}{2565}\cdot k_3 - \frac{h\cdot 1859}{4104}\cdot k_4 - \frac{h\cdot 11}{40}, t_n + \frac{h}{2}\right) \\
&\text{Final 1: } x_{1\_n+1} = x_n + h\cdot\left[\frac{25}{216}\cdot k_1 + \frac{1408}{2565}\cdot k_3 + \frac{2197}{4104}\cdot k_4 - \frac{1}{5}\cdot k_5\right] \\
&\text{Final 2: } x_{2\_n+1} = x_n + h\cdot\left[\frac{16}{135}\cdot k_1 + \frac{6656}{12825}\cdot k_3 + \frac{28561}{56430}\cdot k_4 - \frac{9}{50}\cdot k_5 + \frac{2}{55}\cdot k_6\right].
\end{aligned}
\tag{IV.10}
$$

This method is really 2 separate RK methods that share their initial stages. One method is a 4^{th} order method that uses stages 1 through 5 and equation Final 1, and the second method is a 5^{th} order method that uses stages 1 through 6 and uses equation Final 2. This makes determining the error for the simulation trivial. The error for a time step is evaluated according to:

$$|x_{1\_n+1} - x_{2\_n+1}| < \varepsilon, \tag{IV.11}$$

if ε is above an error tolerance threshold then the simulation needs to choose a new step size and recalculate the time step. If ε is below the error tolerance threshold then the time step is accepted by the solver and the 5^{th} order value of x_{n+1} is propagated to the next time step. If ε is below both the error threshold and a separate step size threshold, then the solver may choose to make the time step greater.

A well-known linear multi-step implicit method is the Backward Difference Formula

(BDF) method. It forms the basis for the popular DASSL solver [89]. The BDF method is multi-step because it calculates the value at x_{n+1} based on values of x at previous time steps. There are a variety of BDF methods, with each method using more previous values of x. The different BDF methods can be represented by an α vector and a β matrix [37]:

$$\alpha = \begin{bmatrix} 1 & 2/3 & 6/11 & 12/25 & 60/137 \end{bmatrix}^T$$

$$\beta = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 4/3 & -1/3 & 0 & 0 & 0 \\ 18/11 & -9/11 & 2/11 & 0 & 0 \\ 48/25 & -36/25 & 16/25 & -3/25 & 0 \\ 300/137 & -300/137 & 200/137 & -75/137 & 12/137 \end{bmatrix} \tag{IV.12}$$

where the i^{th} row represents the ith-order BDF algorithm.

Definition 17 (Order)**.** *The order of the integration method identifies the number of higher derivatives included in the algorithm [35].*

The α vector scales $\dot{x}_n$, and the values of a row in the β matrix scale the current and previous values of x. As an example, the third-order BDF equation is

$$x_{n+1} = \frac{18}{11}x_n - \frac{9}{11}x_{n-1} + \frac{2}{11}x_{n-2} + \frac{6}{11} \cdot h \cdot f(x_{n+1}, t_n + h). \tag{IV.13}$$

Changing the order of the method can be performed by simply adding or dropping a previous value of x and adjusting the equation coefficients. Note from Equation IV.13 that calculating x_{n+1} requires $f(x_{n+1}, t_n + h)$, and that calculating $f(x_{n+1}, t_n + h)$ requires x_{n+1}. This circular dependence indicates the presence of an algebraic loop that needs to be solved at each time step, and therefore makes the method implicit. It is also interesting to note that

the first order BDF equation is the same as the BE method, making the BE method simultaneously a single step and a multi step method (it is also a first order Radau RK method [71]).

IV.2.1 Advantages and Disadvantages

Explicit and implicit, and single-step and multi-step integration methods have specific advantages and disadvantages that make them suitable for a particular type of problem. This sub-section highlights some of the differences between the integration methods by focusing on their ability to cope with changes in step size, variations in the order of the integration method, and stiff systems.

The requirement of linear multi-step methods that the time step is constant across all of the previous time steps adds difficulty when they are used as a part of a variable-step solver, because after the time step is changed the previous values of x need to be scaled to match the new time step size. Multi-step methods also have trouble at simulation startup and after discrete changes in the system state variables, because the previous values of the state variables the method relies on are not present at startup and not valid after a discrete change. This is known as the "startup problem" and potential solutions are covered in [37]. Single-step algorithms handle either of these situations without problem. However, because a single-step method is likely to be a multi-stage RK algorithm each time step will likely be more computationally expensive than a linear multi-step method.

The order of an integration method also has an effect on the simulation. A higher-order method is more accurate than a low-order method, but computationally more expensive; and a lower-order method is less accurate but less expensive computationally [37]. The computational cost of a higher-order method may be offset by being able to take larger step sizes. An advantage of multi-step algorithms is that they are easily able to change the order of the algorithm, to gain or lose accuracy, depending on the needs of the simulation at that

	Explicit Method	Implicit Method
Single Step	Fast computationally Easy step size change Easy discrete changes	Stiff systems Easy step size change Easy discrete changes
Multi Step	Fast computationally Simple order change	Stiff systems Simple order change

Table 4: Strengths of different integration methods.

time. The order of single-step algorithms is generally fixed, and changing the order of the method requires stopping the simulation and choosing a different algorithm.

Stiff systems are systems where the step size of the integration is forced below a value expected by the smoothness of the behavior trajectory when using an explicit integration method [71], and are very common in modern system design, especially when multiple physical domains are present in the same model [34]. Explicit methods are not very efficient at solving stiff systems because the stiff system requires the explicit algorithm to take many very small time steps [60]. Implicit methods are generally very good at solving stiff systems, because they are able to take much larger time steps.

The notion of stiffness, while a commonly used term and introduced in 1952 [40], lacks a precise mathematical definition [71] [38]. For our purposes in this thesis, accepting the above definition is sufficient as it highlights one of the key advantages of implicit integration methods over explicit methods, and the reader is encouraged to review [71], [60] and [38] for more information.

The different strengths and weaknesses of the integration methods are summarized in Table 4.

IV.3 Inline Integration

Inline integration was first introduced in [47], and added to a version of Dymola in [45]. It is described in some detail in [34]. Inline integration works by removing the separation between the model equations and the integration method. The integration equations

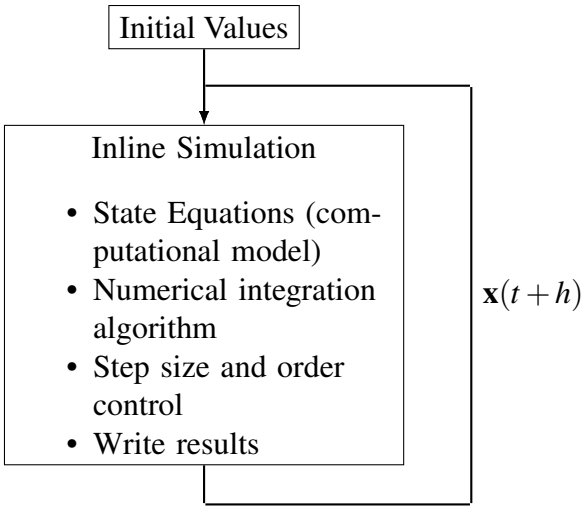


Figure 17: Inline integration simulation structure

are inserted ("inlined") directly into the model equations, and Figure 16 can be modified into Figure 17 when inline integration is used. This allows the system to be solved as a DAE system, and avoiding the need to convert the system to ODE form as Dymola and OpenModelica do when compiling the model. Solving the system as a DAE eliminates the differential equations inherent in the ODE form and the need to rely on an external solver, and instead the simulation only needs to solve a set of difference equations each time step. By itself, inline integration will not necessarily yield a reduction in simulation time, but it will provide a means for parallelizing the integration method with the model, and open up other opportunities for reduce the simulation time.

The advantages of inline integration are best illustrated with an example (this example from [34]). Consider the circuit shown in Figure 18. This example has a structural singularity because the capacitor is forced into derivative causality because it is in parallel with a voltage source. The structural singularity means that the system can be solved directly using an inlined implicit integration algorithm or it can be solved with an explicit algorithm if the system is reduced to index-1 or lower.

The equations with the inlined implicit Euler solver are shown in Figure 19. There are 12 equations and variables, including the integration equations. The DAG of the system

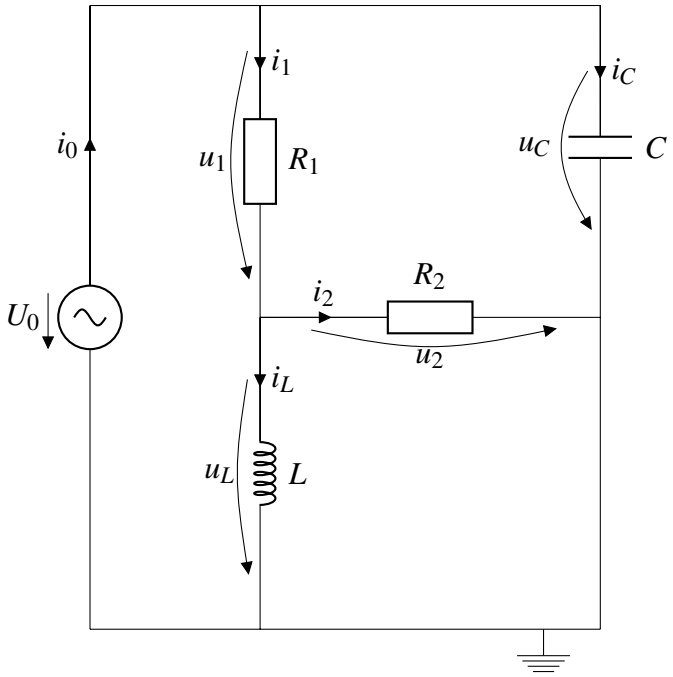


Figure 18: Inline integration example

is shown in Figure 20. With the inlined implicit equations there are 6 strongly connected components with one of the components representing an algebraic loop with 7 equations in it. The main drawback of not reducing the system to index-1 is that block 3 in Figure 20 (same as equation 12 in Figure 19), when rearranged into causal form is

$$duC\_dt = \frac{1}{h}(uC - pre(uC)), \tag{IV.14}$$

where $pre(uC)$ represents the voltage drop across the capacitor in the previous time step, has the step size of the simulation alone in the denominator of the equation. This is an undesirable situation when the system is integrated using a variable step-size solver because the step size can approach 0, which can make the system unstable. Reducing the system to index-1 before inlining the integration algorithm avoids this problem [34], as will be shown next.

To reduce the system to index-1 one of the integrators had to be thrown away, and 7 equations and variables added to the system. Adding the inline explicit equation adds

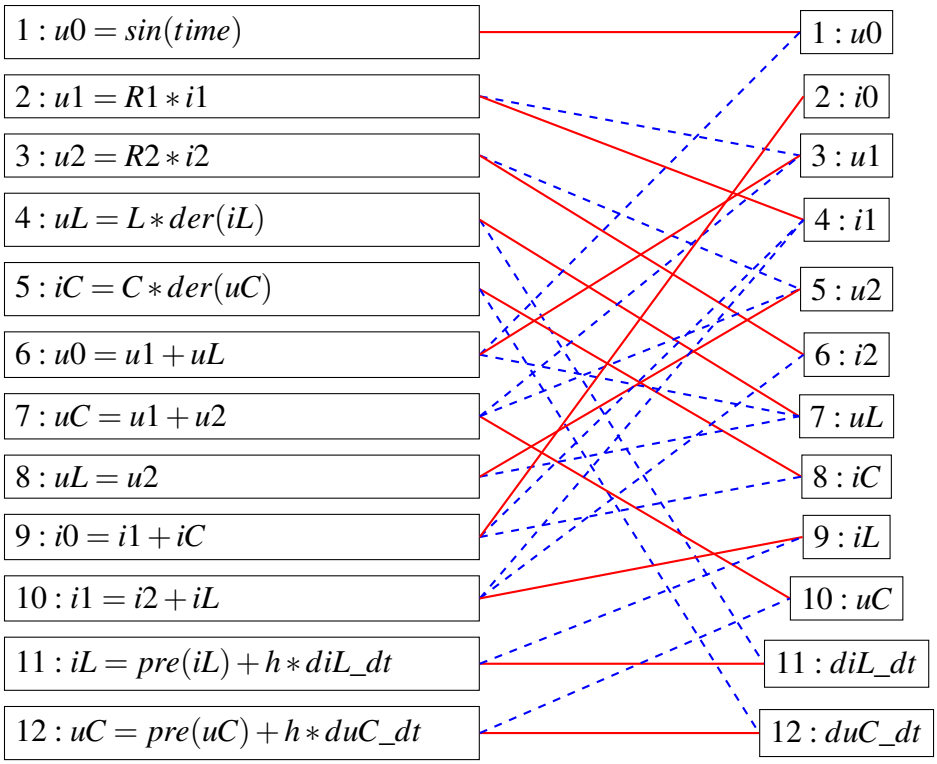


Figure 19: Causalized equations with an inlined implicit Euler solver.

another equation/variable pair to give the system a total of 18 equations and variables. The bipartite graph of the equations and variables is shown in Figure 21, and the corresponding DAG is shown in Figure 22. In addition to this graph having more equations to solve, it has 2 algebraic loops; one with 3 equations and one with 5 equations.

In this situation, inlining the implicit integration algorithm without reducing the system to index-1 has resulted in a simulation problem that has fewer variables and fewer algebraic loops than reducing the system to index-1 and using an explicit algorithm. However, requiring fewer variables came at the expense of creating a larger algebraic loop that needs to be solved at each time step, and creating a situation where the simulation step size, which can approach 0, is alone in the denominator of an equation. It remains to be seen if these two drawbacks are worth avoiding the work to reduce the system to index-1 before simulation.

With inline integration the integration methods are specific per state variable, and not specified for the entire system as is done traditionally. This means that inline integration

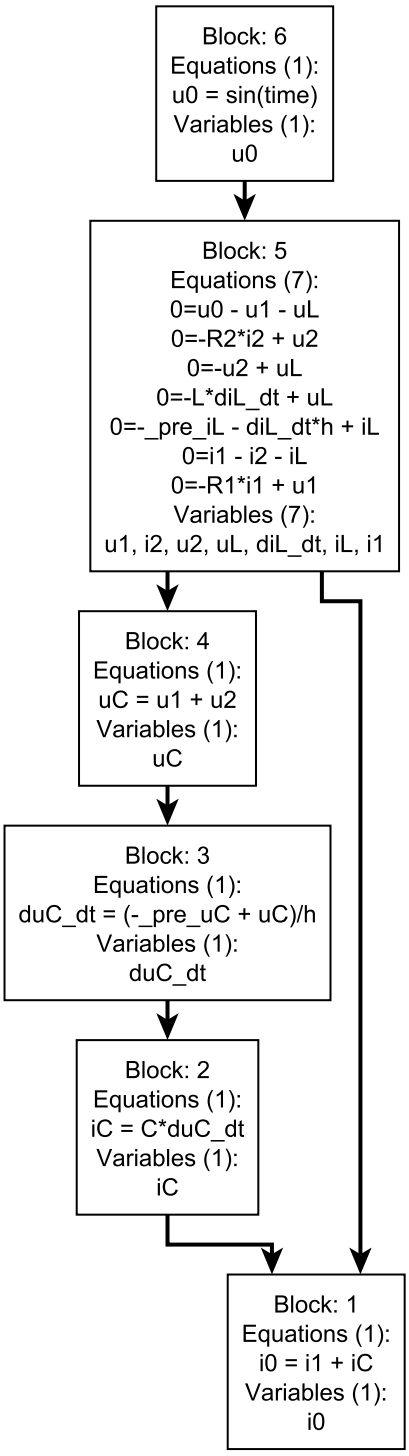


Figure 20: DAG with inlined implicit Euler solver.

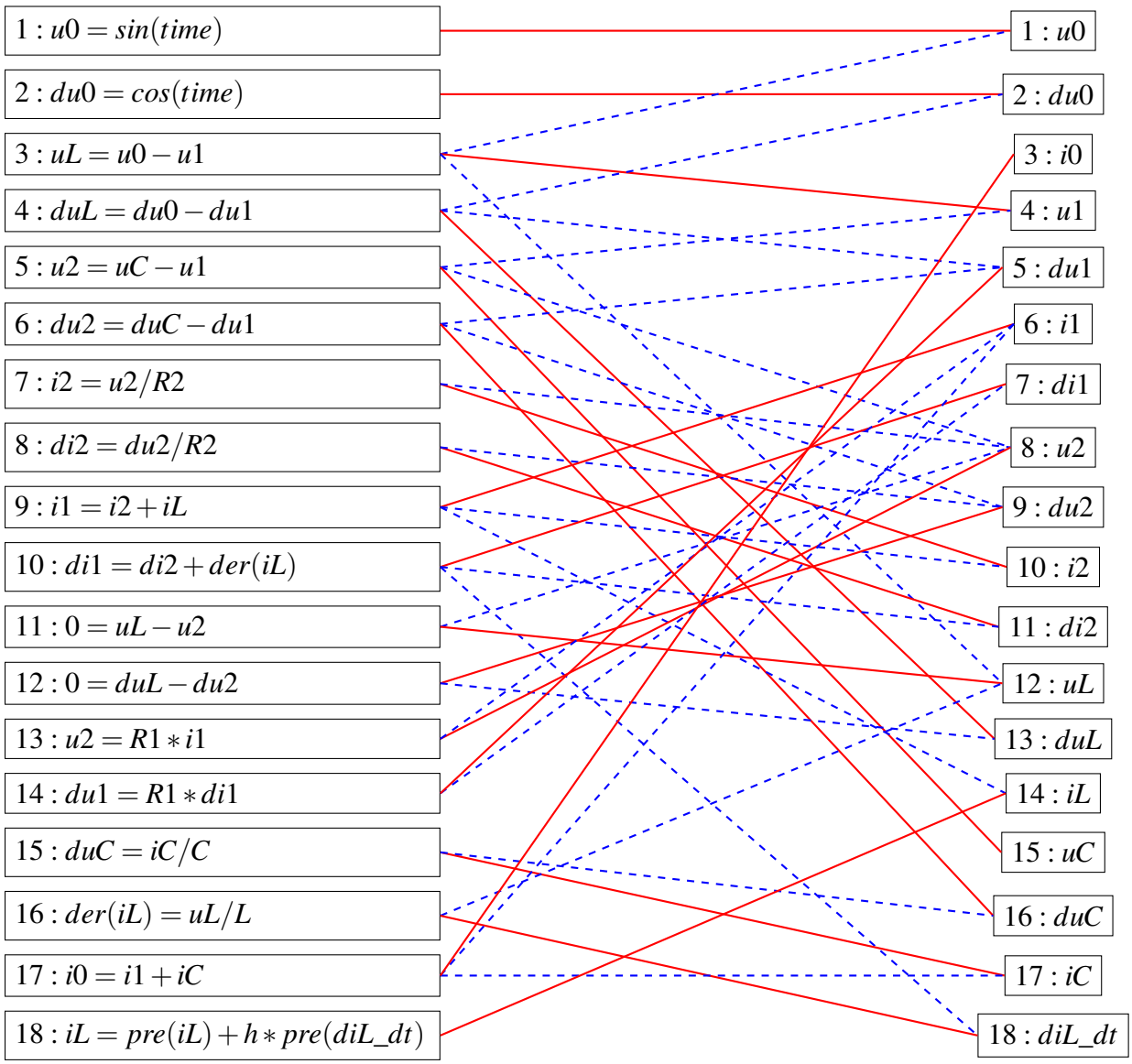


Figure 21: Causalized equations after converting Figure 18 to index 1.

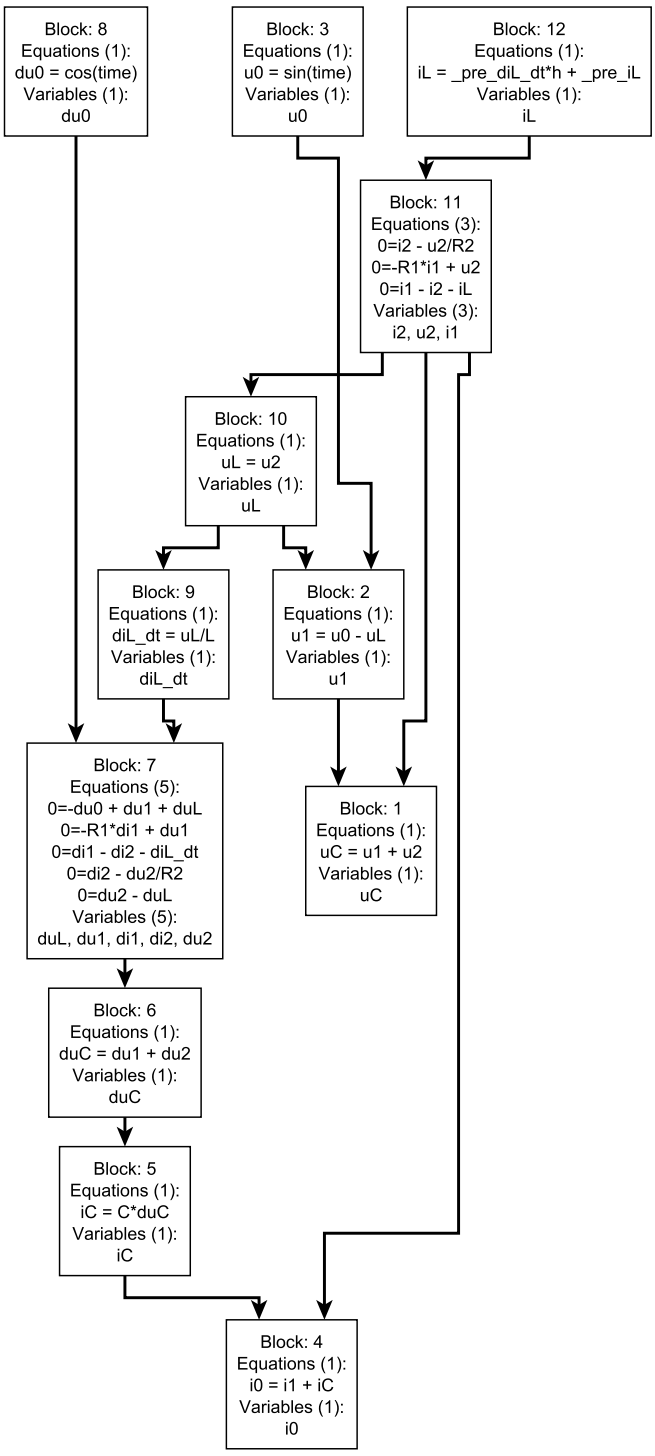


Figure 22: DAG with inlined explicit Euler solver after system is reduced to index 1.

provides a means for dynamically changing the integration algorithm per state variable. By dynamically changing the state variable we expect to reduce the simulation time for a model by always using the most appropriate algorithm for each state variable at each stage of the simulation.

Inline integration is not limited to simple algorithms such as implicit Euler equations, as it can be used with more complex methods such as implicit Runge-Kutta algorithms and the backward-differential formula algorithms. Inlining an implicit Runge-Kutta algorithm, such as the Radau algorithms, require that the entire set of model equations be duplicated for each stage of the algorithm [34]. It is an open question if inlining implicit RK algorithms are worth the extra cost of having the entire system repeated.

Dymola can use inline integration as a part of its real-time package [27]. When using inline integration Dymola limits the simulation to a fixed step size [28]. This is to guarantee timing requirements in a real time environment. Unfortunately, Dymola does not allow inline integration when simulating a model in a non-real-time environment. Despite this, inline integration does appear to produce good results as presented in [98]. The authors presented 3 test cases comparing inlined explicit Euler, inlined implicit Euler, and inlined implicit Runge-Kutta3 algorithms to a non-inlined explicit Euler algorithm. In every case, the inlined algorithm outperformed the non-inlined algorithm. In some sense, this is not a true example of the power behind inline integration, as in 2 of the cases the authors compared different solvers, and in one of those 2 cases the authors were simulating a stiff system, and the explicit Euler method is known to be very inefficient in simulating these types of problems. The most fair example compared a non-inlined explicit Euler integration method with an inlined explicit Euler integration method. In this case the inlined Euler integration method produced a relative speedup of 1.6.

IV.4 Parallelization of Modelica Model Simulation

Modelica is the primary modeling language of the AVM project, so we briefly discuss methods to parallelize the simulation of a Modelica model. There are two parallelization approaches used by the Modelica researchers: (1) automatic parallelization and (2) manual parallelization. Automatic parallelization refers to the model compiler and run-time environment parallelizing the computational model for the user. Manual parallelization provides the user with tools to divide the system declarative model into parts for parallelization [96].

IV.4.1 Automatic Parallelization

The team behind OpenModelica (OM) have researched several methods to parallelize the simulation of a Modelica model. Their algorithms are applied to a distributed memory parallel environment of multiple networked single core PCs. This parallel structure has significant communication time so much of their work has been dedicated to minimizing the communication between processors. Aronsson, in [30], proposes ordering the equations in a similar way as we describe in Section III.5, but he takes it a step further by breaking each of the model equations down into the individual arithmetic operations and treating each operation as a task in a dependency graph. Then the different tasks are merged or duplicated to produce a dependency graph that is easily parallelized. In some cases the tasks are merged or duplicated in such a way that no communication is needed between the processors; however, the result of this approach is that each duplicated task results in duplicate computational work. When no communication is required between the processors, there is no schedule needed because each processor simply completes the tasks assigned to it, and there is never a decision to complete one task over another. The integration method was not parallelized, so once all of the equations were solved for a time step, the solved variables were sent back to the main CPU running the solver to perform the integration. The OM group presented some later work with the same equation parallelization scheme,

but also included a parallel solver [75]. In [76] they presented an algorithm to sort the tasks assigned to the various processors to minimize communication delays.

The OpenModelica team has also examined using a graphics processing unit (GPU) to solve the model equations [82] [100]. Their approach is similar to what was presented in [30], where the equations are broken down into arithmetic operations and then merged or duplicated to produce parallel tasks. They use a critical path scheduling algorithm to create the schedule, and a simple task estimation scheme to estimate the computation costs of the tasks. They assign each simple operation a cost of 1, any special functions are given a cost of 4, and all communication costs are given a value of 100. As evidenced from their communication cost estimate, minimizing the communication time between processing units is, again, very important to ensuring good performance. However, using a GPU to simulate a Modelica model is only suitable for certain Modelica models that fit a GPU's particular strengths of large data sets, high parallelism, and minimal dependency between data elements. In general most Modelica models do not meet this standard due to the limited parallelism and strong dependency between data elements [100].

Casella [32] proposed an algorithm for the parallel simulation of a Modelica model that uses a similar computation model to what we described in Section III.5, where the equations in the model are arranged, according to their dependencies, into a DAG. The algorithm starts by grouping the tasks that have no predecessors into a set S_1. These tasks are then removed from the graph, and the new nodes that have no predecessors are grouped into a set S_2. This process continues until all of the nodes in the graph are assigned into one of the sets of S_i. The sets are scheduled in order, with the tasks in each set scheduled in a random order. The simulation must wait until all of the tasks in each set are complete before beginning to process the next set. This slows down the overall processing as some tasks from a later set S_{j+1} will likely be able to start if there are idle processors waiting for the tasks from the current set S_j to finish. Casella does make two important assumptions, that are not necessarily reflected in reality, that will improve the performance of his algorithm

[31]. The first assumption is that the number of tasks in each set is much greater than the number of processor cores, and the second is that the tasks have approximately equal processing time. Together these ensure that it is not likely that processors will be standing idle, which will improve the performance of the proposed algorithm. In early testing we have found that neither of these assumptions are true: only for very large systems will the number of tasks be much larger than the number of processor cores, and tasks will likely have vastly different processing times due to algebraic loops requiring much more computation time than single equations.

Walther, et. al. [103] also presented an approach to parallelizing a Modelica model using a DAG computational model, similar to what we described in Section III.5. Their scheduling approach required estimates for the computation times of the tasks and the communication times between tasks. The communication costs were based on measured communications between processors, and then applied to the system based on how much data needed to be transported between tasks. The computation times for each task were measured by running the model serially. The measured task times were then applied to the dependency graph for scheduling. In order to help reduce the communication costs between the tasks, they merge tasks in certain situations. They use a fixed schedule, where each task is assigned to a processor prior to run time. They did a comparative analysis of three parallel scheduling algorithms on 3 different Modelica models from the Modelica Standard Library. The algorithms were tested by modifying OpenModelica to parallelize the simulation. The three algorithms tested were the level scheduling (essentially the approach suggested by Casella), a simple list schedule of their own design, and the Modified Critical Path algorithm [105]. Their approach showed a relative speedup between 4 and 6 for 6 processors, but the data in the paper is not presented in a form where the exact speedup can be determined.

In [46] Elmqvist, Mattsson and Olsson presented a refinement of the Casella approach to creating a parallel schedule. The authors used heuristic rules to divide the system of

equations into several layers. Within each layer a number of "sections" are created, and the specific SCCs are assigned to a section such that the SCCs within a section are to be executed in sequence, but each section within a layer can be executed in parallel. The authors try to load balance the sections by estimating the evaluation cost of each SCC by estimating the number of operations needed to solve each SCC. This means that the scheduling assignment is static during the simulation. They achieved speed up factors of 2.1 to 3.4 on a variety of system models, where the speedup factor is defined as the serial execution time divided by the parallel execution time. One of the driving forces of this approach is to reduce the overhead costs of setting up and starting the various threads. This seems to indicate that the authors are spawning new execution threads as the simulation progresses. It is worth investigating if reusing execution threads can further speed up their approach. The different models were simulated using a modified version of Dymola. Their approach was released in the commercial Dymola product as a part of the Dymola 2015 FD01 release [27], which makes Dymola one of the few commercial simulation tools to support parallelization.

IV.4.2 Manual Parallelization

The OpenModelica team have also investigated transmission line modeling (TLM) [80] [96] [97]. TLM is based on wave propagation theory, and tries to break a system into components by replacing capacitive elements with transmission line elements. Adding the transmission line elements effectively adds a delay to the system corresponding to the propagation delay of a physical phenomena. With the transmission line elements in place, the simulation step size can be set so that it is smaller than the delay from the transmission line components. This allows the system to be broken into independent components at the transmission lines. These components can be simulated on separate processors, with data transferring between the components after each time step. Despite the fact that this method is based on first principles physics, it has multiple difficulties. The first difficulty

is that this approach is very dependent on the modeler making good decisions as to where transmission line elements will be most effective and appropriate. The second problem is that the propagation delays are likely to be very small, which limits the step size available to the simulation solver, which in turn limits the benefit that can be gained by dividing the model into independent pieces.

In [59] the author presents an extension for explicitly stating parallelism in the model to the algorithm subset of the Modelica language. The extension is called PARModelica, which is short for Parallel Modelica. This approach is intended to allow the modeler to parallelize commonly used functions such as matrix transposition or multiplication. The drawback is that the language extensions only apply to algorithm part of the Modelica language, and not to the rest of the Modelica language that is used to describe the mathematical behavior of the model. This limitation severely restricts the usefulness of this approach. Also, because this is a manual parallelization method, its implementation will be limited to those users with a strong background in parallel programming.

IV.4.3 Critical Review

In our parallelization algorithms we will avoid manual parallelization methods because we do not want to task a design engineer with having to parallelize their system model. Manual parallelization will require a level of expertise and an understanding of the computational architecture on which the model will be simulated that a design engineer is not likely to have. Therefore, our parallel algorithms will support automatic parallelization based on the structure of the equations in the system. We will also focus on reducing our models to ODE and state-equation form, because this will help reduce the amount of communication that needs to happen within a time step, which will avoid the communication problems presented in [30], [75], [76], [82], [100] and [103]. Reducing the model to ODE and state equation form also makes partitioning the model easier, because, as described in Section III.1, there are no dependencies between the state equations meaning they can

be partitioned into whatever form best suits our simulation needs. This avoids the graph rewrite rules that were a key contribution of [30], the task merging rules of [103], and the heuristic rules to divide the system of equations described in [46]. Reducing the computational model to state equation form also has the advantage of making the model suitable to simulating on a GPU, and we leave this task to future work, described in Section VII.3. One of the primary problems with the method presented in [46] is that they are create threads throughout the simulation process. This adds overhead to the simulation and takes time away from computation. Our parallel algorithms will maintain one set of threads for the duration of the simulation, as a means to reduce overhead.

IV.5 Chapter Summary

In this chapter we presented problems pertaining to the simulation of a system of equations. We first presented key definitions and a high level overview of traditional simulation in Section IV.1. We also presented a high level comparison of the different kinds of integration methods that are available in Section IV.2. They can be broken down into single-step methods, that only reference the most recent time step, and multi-step algorithms that look to multiple previous time steps. The methods can be further divided into explicit and implicit methods, where explicit methods can directly calculate the next value of the integrated variable, and implicit methods must calculate the next value through an algebraic loop. Section IV.3 presented Inline Integration, a numerical integration technique that we will use in Chapter VI. Finally, Section IV.4 gave an overview of published methods that have been used to parallelize the simulation of a Modelica model, and presented a critical review of those methods.

CHAPTER V

SHARED MEMORY PARALLEL PROCESSING ARCHITECTURE

The primary method we are going to pursue to increase the performance of equation based model simulation is parallelization. This chapter discusses various parallel processing architectures and hardware technologies that have been used in creating parallel processing systems. Section V.1 discusses the traditional parallel architectures categorized according to Flynn's taxonomy [50], and highlights the Multiple Instruction, Multiple Data (MIMD) architecture and Graphics Processing Units (GPU), which are a variation of the Single Instruction, Multiple Data (SIMD) architecture. Section V.2 discusses the hardware architecture of a modern multi-core CPU and the corresponding memory structure for the CPU. This section also presents experimental data that is used to describe the behavior of the memory and cache configuration on the processors. Finally, Section V.3 presents a summary of the topics covered in this chapter.

V.1 Parallel Processing Architectures and Languages

Traditionally computing architectures have been described in terms of four qualitative categories according to Flynn's taxonomy [50][92]:

1. **Single Instruction, Single Data (SISD):** This architecture is a conventional sequential computer with a single processing element that has access to a single program data storage.
2. **Multiple Instruction, Single Data (MISD):** In this architecture there are multiple processing elements that have access to a single global data memory. Each processing element obtains the same data element from memory, and performs its own instruction on the data. This is a very restrictive architecture, and no commercial parallel computer of this type has been built [92].

3. **Single Instruction, Multiple Data (SIMD):** In this architecture, multiple processing elements execute the same instruction on their own data element. Applications with a high degree of parallelism, such as multimedia and computer graphics, can be efficiently computed using a SIMD architecture [92]. Modern tools for parallel processing large amounts of database information, such as Hadoop [6], are closely related to the SIMD architecture.
4. **Multiple Instruction, Multiple Data (MIMD):** In this architecture, there are multiple processing elements and each element accesses and performs operations on its own data. The data may be accessed from local memory on the processor, from from shared memory, i.e., memory shared by multiple processor units.

V.1.1 MIMD Architecture

MIMD architectures underly the processors in modern engineering workstations, such as the Intel Core processors [7] and AMD FX processors [22]. We are targeting improved performance of simulation on engineering workstations so the MIMD architecture will be our focus in this research.

There are two primary broad categories of MIMD parallel processing architectures: (1) shared memory architectures and (2) distributed memory architectures [92]. (Much of this summary is derived from [92].) Shared memory architectures have a global shared memory that can be accessed by all of the processing units; an example shared memory architecture is illustrated in Figure 23. Data is passed between processing units by one unit writing data to the shared memory and a second unit then reading the data to use in further processing. Synchronizing the reading and writing to the global shared memory is a primary means of characterizing the behavior of the program. Modern engineering workstations are examples of shared memory machines, where each of the computing cores on the processor have equal priority to access the on-chip memory. The on-chip memory is called the cache and is described in Section V.2.2. Distributed memory architectures have a

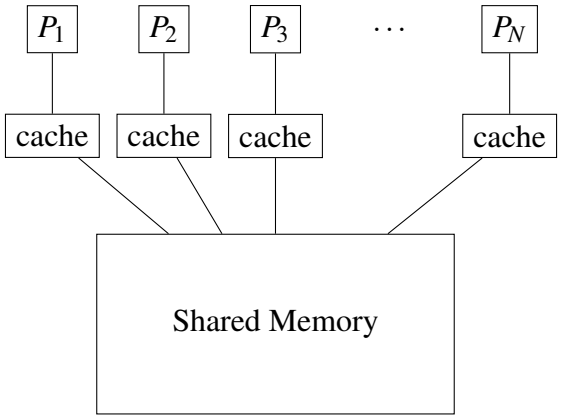


Figure 23: Example shared memory MIMD architecture.

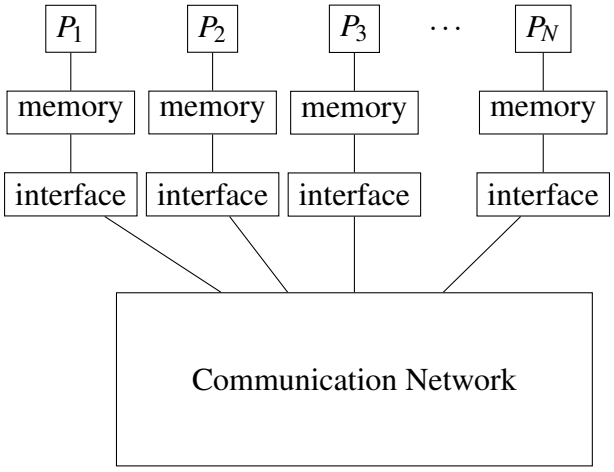


Figure 24: Example distributed memory MIMD architecture.

private, or local, memory for each processor, and no global synchronization of the memory is needed. An example of a distributed memory architecture is in Figure 24. Information is exchanged between processors by sending the data through a connection network (message passing) from one processor to another. This data exchange is what provides the program synchronization. A workstation cluster communicating through a network is an example of a distributed memory parallel architecture.

V.1.2 Graphic Processing Units

Graphic Processing Units (GPUs) are becoming common in scientific research [79][86], and are highly parallel, extremely high performance, floating point-calculators. GPUs mix elements of SIMD and MIMD architectures; however in our research we focus strictly MIMD parallel architectures and will leave applying parallel simulation methods to a GPU to future research. Even though GPUs are not our focus, they are rising in popularity in the field of scientific computing, and therefore they will be briefly discussed here.

Early GPUs were released in the late 1990s [23], and were designed exclusively for rendering computer graphics. Researchers soon became interested in leveraging the processing power of a GPU to help solve scientific problems, and the CUDA programming language was developed to help support using a GPU for non-graphics computing [23][79].

GPU architectures use many processor cores that are organized into multithreaded multiprocessors (Figure 25). Each streaming multiprocessor (SM) has several streaming processors (SP) that synchronize memory only at synchronization points [86]. The SPs are highly multithreaded, and each manages the state of concurrent threads in hardware [86]. The entire GPU contains several of the SMs, the number of SMs depends on the desired performance of the GPU, and the SMs share memory through an interconnection network. This arrangement of parallel processors within parallel processors makes GPUs very powerful floating-point calculators, and their use is spreading through the scientific community [86].

V.2 Multi-Core CPU

Multi-core CPUs, such as Intel's Core line of processors [7] and AMD's FX processors [22], are the backbone of modern engineering workstations and are an example of MIMD parallel architecture. At a conceptual level, the CPU can be divided into two parts: the computational processor, and the memory. Our primary focus will be on the processor

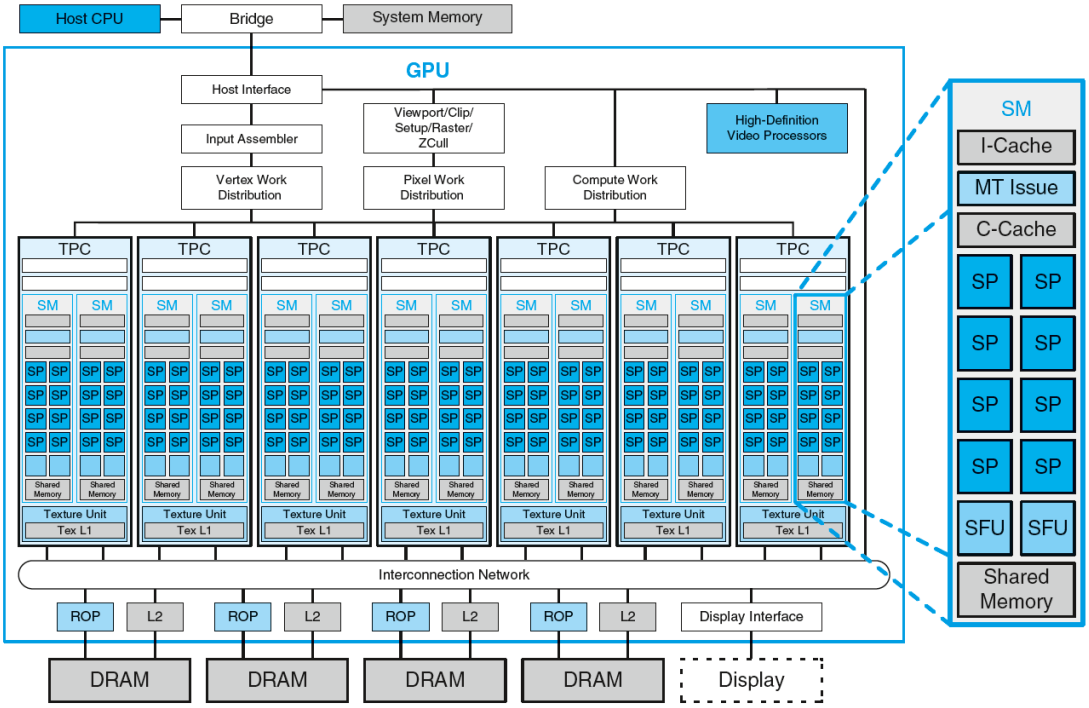


Figure 25: Example GPU architecture[86].

memory, which is covered in Section V.2.2, but we will first summarize the important aspects of the processor as it relates to the memory system.

V.2.1 Processor

Current high end chips have four to six processor cores [7][22], and each processor core can handle two computational threads through hardware multithreading.

Definition 18 (Thread)**.** *A single program execution stream that includes the program counter, the register state and the stack [85]. Multiple threads can share an address space, meaning threads can access memory used by other threads. Due to this shared address space, a processor can switch between threads without invoking the operating system.*

Definition 19 (Hardware Multithreading)**.** *Increasing utilization of a processor by switching to another thread when one thread is stalled due to causes such as waiting for data from memory, or a no operation instruction[85].*

Definition 20 (Process)**.** *A higher level computation unit, above threads. A process includes one or more threads, the address space, and the operating system state. Changing between processes requires invoking the operating system. [85]*

CPU hardware that implements hardware multithreading is designed such that each hardware thread is presented to the Operating System (OS) as an individual CPU, which gives the OS 8 to 12 processors (double the number of cores) to use for computational work. In our work, we treat these multithreaded processors as individual processors, just like the OS, so that a computer with 4 cores that supports 8 threads (2 threads per core) is treated as having 8 unique cores.

V.2.2 Cache

The chip has a memory hierarchy, called a cache, which is the memory closest to each processor core. The cache closest to each core is called the L1 cache, and, in a 3 level hierarchy such as Intel's Core architecture [7], the cache farthest away from the core is called L3. Beyond the L3 cache is the main system memory (RAM). An example of the Intel Core i7 cache structure is in Figure 26. In the Intel i7 example in Figure 26 the L1 cache is split into an instruction cache and a data cache, and each L1 and each L2 cache are usable by only 1 core, while the L3 cache is shared across all cores. This L3 cache allows the individual cores to exchange data, and the L1 cache is used to keep frequently used data readily available, as cache access times are longer the farther away the level of cache is from the core. Also, the cache levels that are farther away from a processor are going to be larger than the levels closer to the processor. On the CPU we use to perform our experiments in Chapter VI each L1 cache is 32KB, each L2 cache is 256KB, and the L3 cache is 8MB [7].

In most cases, the cache is implemented as a hierarchy, where data cannot be in the L1

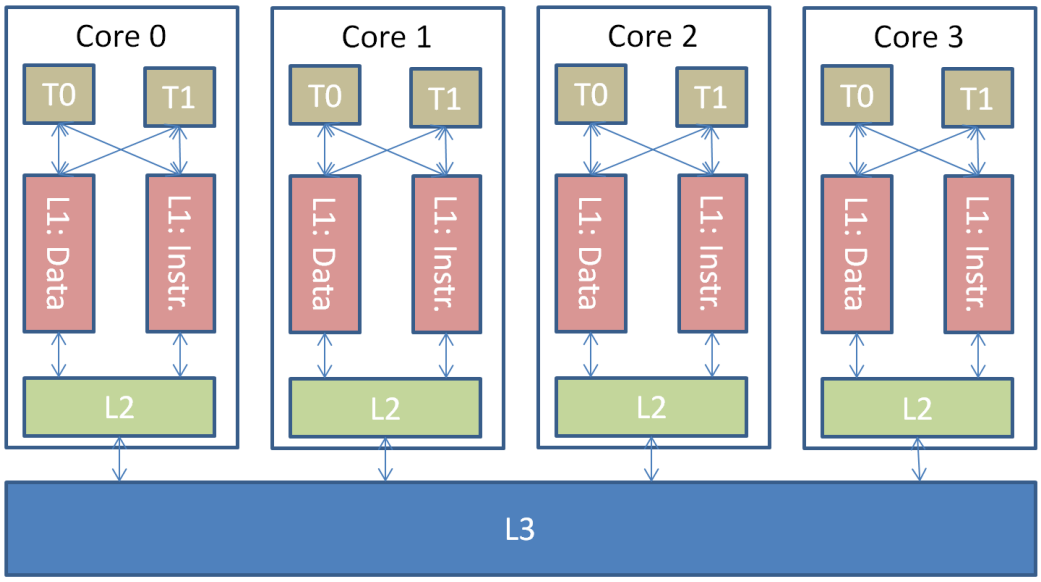


Figure 26: An example of the memory structure of an Intel i7 processor [7][78].

cache without it also being in L2 and L3 [88]. More formally:

$$\text{if } x = \text{some data} \tag{V.1}$$

$$x \in L1 \implies x \in L2 \implies x \in L3. \tag{V.2}$$

In order for processors to communicate or exchange data, the data needs to be in the layer(s) of cache that are shared across cores. Usually, as shown in Figure 26 this is the cache that is the farthest from the processor, and therefore the slowest layer of cache. As an example, since the threads can only directly access their L1 cache, to send data from core 0 to core 2, core 0 will first have to write its data to the L1 cache. Then core 2 will request the data through a load or get instruction. The memory controller on the chip will then move the data from the L1 cache on core 0, to the L2 cache on that core, and then to the global shared L3 cache. After it is in the L3 cache the data will be moved into the Core 2 L2 cache, and finally into the L1 cache on core 0 where the data can finally be used.

The cache is organized into cache lines, or blocks.

Definition 21 (Cache Line)**.** *The minimum unit of information that can be present or not present in a cache [88].*

Definition 22 (Cache Line Read)**.** *The process of pulling a cache line from a cache far away from a CPU core to a cache close to the CPU core.*

In most modern desktop processors the cache line is 64 bytes. This means that moving 4 bytes of data (typically, the size of an integer in C/C++) to the L1 cache will move the entire 64 byte cache line that contains the requested 4 bytes of data. Multiple threads in different cores accessing data in the same cache line can lead to a problem known as cache line sharing.

Definition 23 (Cache Line Sharing)**.** *When two unrelated variables are located in the same cache line are being written to by threads on separate cores, the full line is exchanged between the two cores even though the cores are accessing different variables [***?** *].*

Cache line sharing can have a dramatic effect on the performance of software execution, and should be avoided [85][83]. One method to avoid cache line sharing is to use cache aligned data structures.

Definition 24 (Cache Aligned Data)**.** *Cache Aligned Data is data that has a memory address that is an even divisor of the cache line size. That is, in most modern architectures, cache aligned data has memory address* `modulo` $64 = 0$.

Cache line sharing will be discussed in more detail in Section V.2.2.1.

The cache access times, also called a cache line read, are an important factor in understanding processor performance. To illustrate this, we can construct a simple example program that creates a very large array of data, larger than the size of the L3 cache. We then measure the time it takes to perform a simple calculation on every K^{th} element of the array, where K is a variable parameter defined by $K \in 2^x | x \in \mathbb{Z}$. The pseudocode of the algorithm used for the experiment is shown in Algorithm 1 (credit [83]). To conduct the

experiment we vary the parameter K from $1 = 2^1$ up to a threshold, in this case $1024 = 2^{10}$, repeat the experiment 20 times for each value of K, and then average the time to complete the computational work for each value of K. Intuition indicates that since the program is performing half of the computation work with each increase in K, that the time it takes to complete the experiment will also be reduced by half for each increase in K.

Algorithm 1 Algorithm for an experiment showing the effect of cache line reads when compared to computation time.

```
size = 64 * 1024 * 1024
arr = new[size]
for k = 1; k <= 1024; k* = 2 do
    timeStart = timestamp
    for i = 0; i < size; i+ = k do
        arr[i]* = 3
    end for
    timeStop = timestamp
    timeDuration = timeStop − timeStart
    print k, timeDuration
end for
```

This experiment was performed on a laptop running Windows 7 x64, with an Intel i7-2620 CPU @ 2.7 GHz, and 8GB RAM. The code was written in C++, and *arr* from Algorithm 1 is an array of integers. In C++ on this hardware, integers have a size of 4 bytes, which means that 16 integers can be stored in the processor's 64 byte cache line. Also, in C++ arrays are stored in contiguous memory, so we know that as we move through the array the program will regularly and predictively need to pull data from a new cache line. After the experiment was run 20 times, the time durations are averaged, and the results are shown in Figure 27. What these results show is that reading the cache lines dominates the program run time, not the computational work. For $2 \leq K \leq 16$ the program run time is nearly constant. Since 16 integers fit in a cache line, all values of $K \leq 16$ require reading the same number of cache lines. For $K \geq 32$ the processing time drops off exponentially because the memory system has to read half the number of cache lines, and perform half

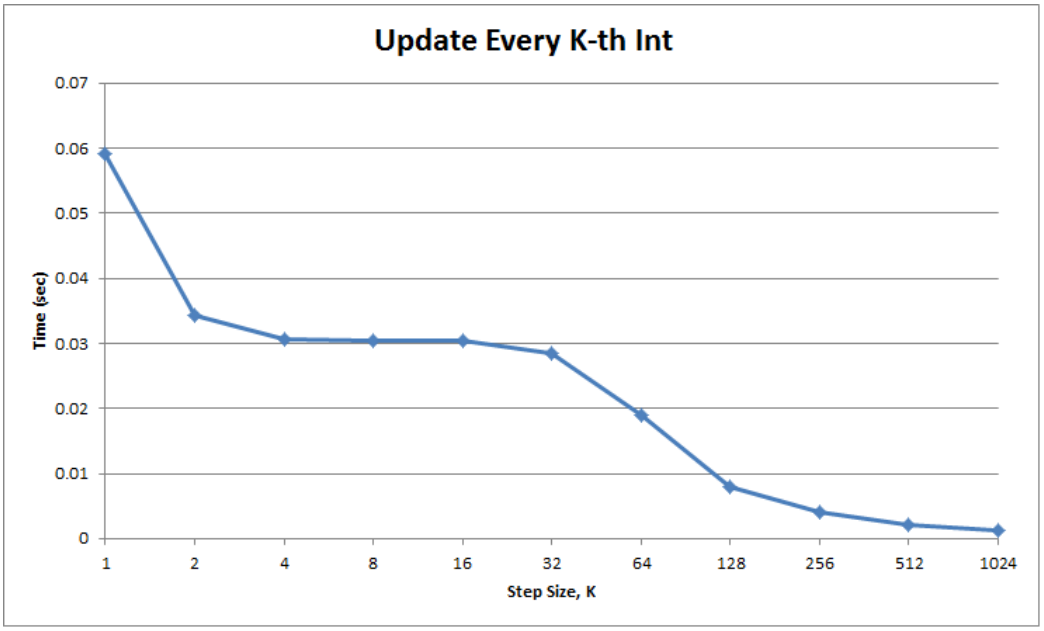


Figure 27: Results from an experiment comparing cache line read time to computation time.

the computational work, for each increase in K. There is a drop in time between $K = 1$ and $K = 2$, and this indicates that for this interval the computational work of touching every element in the array dominates the cache read times.

The sizes of the cache levels can be found in the system documentation, but the access times between the various levels of cache and the system main memory can be used to determine the cache size at each level of the cache experimentally [83]. To perform this experiment we modify Algorithm 1 such that instead of modifying the number of cache lines read, we modify the array size from 1kB up to 64MB, and always read the same number of cache lines regardless of the size of the array. The algorithm will only perform a simple calculation of the data in the cache line, which will force the cache access times to dominate the experiment run time. The pseudocode for this algorithm is in Algorithm 2. With this experiment, we expect to observe the time to read all of the cache lines to increase as the array we are iterating over spills over from one level of cache to the next, but we also

expect to observe, based on the previous experiment, that the time to read all the cache lines to be the same if different array lengths can fit inside the same level of cache.

Algorithm 2 Algorithm for an experiment showing the differences in cache line reads between layers of cache.

```
cacheLines = 64 * 1024 * 1024
arr = arrpointer
initArrSize = 1024 {1kB}
maxArrSize = pow(2,26) {64MB}
varsInCacheLine = cacheLineSize/sizeof(arrtype)
for arrSize = initArrSize; arrSize ≤ maxArrSize; arrSize *= 2 do
    elementCount = arrSize/sizeof(arrtype)
    arr = new[elementCount]
    timeStart = timestamp
    for i = 0; i < cacheLines; i++ do {Touch a large number of cache lines in each array,
    loop back to the beginning of the array and start over if we reach the end.}
        arr[(i * varsInCacheLine) modulo elementCount]++
    end for
    timeStop = timestamp
    timeDuration = timeStop − timeStart
    print arrSize, timeDuration
    delete arr
end for
```

This experiment was run on the same laptop as the previous experiment. The processor in that laptop has L1 caches of 32kB, L2 caches of 256kB, and an L3 cache of 4MB. We implemented Algorithm 2 in C++ using integers as the data type for *arr*, and then ran the algorithm 20 times and averaged the results of those 20 runs. The results are in Figure 28. With this experiment, it is clear when the array grows too large for its current level of cache and must be stored across multiple cache levels. There is a distinct jump in processing time when the array grows from 32kB to 64kB, from 256kB to 512kB, and from 4MB to 8MB. There is an unexpected jump in processing time between array 2MB and 4MB array sizes, since the L3 cache is 4MB we only expect a jump when the array grows larger than 4MB. This extra jump is likely due to data being in the L3 cache, such as operating system data,

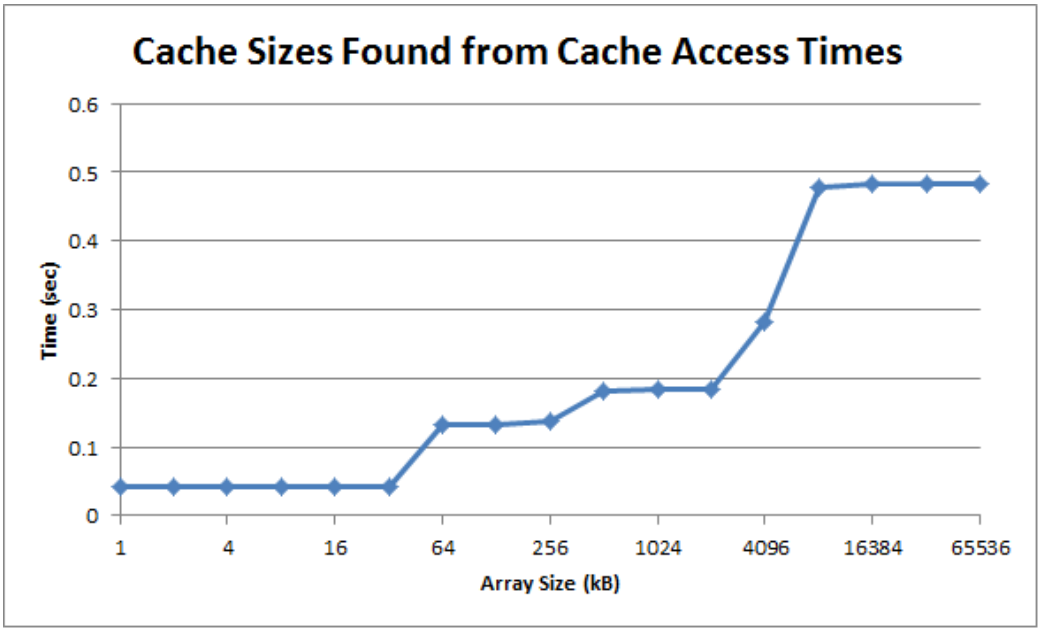


Figure 28: Results from an experiment that found the cache sizes from their access times.

that we have no control over from the perspective of a simple C++ program. It is also clear when the array size remains within a layer of cache, due to the leveling off of execution time at 4 different places in Figure 28.

V.2.2.1 Cache Coherence

A multi-core CPU with a cache accessible by multiple cores, such as the architecture in Figure 26, introduces difficulties managing the cache, called the cache coherence problem.

Definition 25 (Cache Coherence)**.** *A cache is coherent if any read of a data item returns the most recently written value of that data item [85].*

As an example, consider the situation described by Table 5. At time step 0, the cache for CPU A and CPU B is empty, and memory location X has a value of 0. At time step 1, CPU A reads location X, and the value of X is loaded into CPU A's cache. At time step 2, CPU B reads location X, and the value of X is loaded into CPU B's cache. At time step 3, CPU A stores 1 into X. This updates CPU A's cache, and memory location X. Because

Time Step	Event	Cache CPU A	Cache CPU B	Location X
0				0
1	CPU A reads X	0		0
2	CPU B reads X	0	0	0
3	CPU A writes 1 into X	1	0	1
4	CPU B adds 2 to X	1	2	2

Table 5: Cache coherence problem for a single memory location (X), and two processors (A and B) [85].

this processor does not have a coherent cache, CPU B's cache is not updated with the new value from CPU A. At time step 4, CPU B adds 2 to its value of X. Since CPU B's cache was not updated in time step 3, after the add it has a value of 2, instead of the correct value of 3. CPU B also writes its new value to location X, and now every copy of location X, CPU A cache, CPU B cache, and location X, all have an incorrect value of X. The way to avoid this problem is to enforce cache coherence and write serialization.

Definition 26 (Write Serialization)**.** *Two writes to the same memory location by any two processors are seen in the same order by all processors [85].*

Cache coherence will force the CPU B cache to update with the write from CPU A before it performs the add in time step 4. Write serialization will cover a situation where 4 processors, A, B, C, and D, all have location X in their cache, and during the same clock cycle both A and B write to location X. Write serialization will force the last (most recent) write to X, CPU A or B, to be propagated to all other CPUs that have location X in their cache. Without write serialization it is undetermined which write will be considered most recent by each processor's cache; CPU C may consider the write from CPU A to be the most recent, and CPU D may consider the write from CPU B to be the most recent. Having different values for the same memory location removes all of the advantages presented by the shared memory architecture.

The example in Table 5 also illustrates cache hits and cache misses. There is a cache miss at time steps 1 and 2, because, at the time of the read, the value of location X is not

Time Step	Processor Activity	Bus Activity	Cache CPU A	Cache CPU B	Location X
0					0
1	CPU A reads X	Cache miss for X	0		0
2	CPU B reads X	Cache miss for X	0	0	0
3	CPU A writes 1 into X	Invalidation for X	1		1
4	CPU B reads X	Cache miss for X	1	1	1

Table 6: Example of cache snooping and write invalidation protocols [85].

currently in the cache of CPUs A and B. Time steps 3 and 4 have cache hits, because the value of location X is in the cache of the respective CPUs when the CPUs take an action on the memory location.

Definition 27 (Cache Miss)**.** *A request for data from the cache that cannot be filled because the data is not present in the cache [88].*

Definition 28 (Cache Hit)**.** *A request for data from the cache that can be filled because the data is present in the cache [88].*

Cache coherence and write serialization are known problems, and they are solved with a hardware memory controller. Typically this controller will implement a snooping protocol that monitors the memory bus in the processor to determine if a cache has the most recent copy of the data [85]. The snooping protocol will invalidate all copies in cache of memory location X after a write to any one of the copies of X. An example of snooping and write invalidate protocols is in Table 6. In this example, the behavior is the same as in Table 5 until time step 3, when A writes 1 into X. At at that moment, because this hypothetical processor implements snooping and write invalidation protocols, the value of X in CPU B's cache is invalidated, and CPU B must read location X again (time step 4) before it can take any action on X. At the end of time step 4, CPU A cache, CPU B cache and location X will all have the same value of X.

The combination of cache lines that can contain many unrelated variables, and the need

for cache coherence, can lead to cache line sharing. Cache line sharing is a problem when two threads on separate cores are writing to unrelated variables on the same cache line. The cache line memory structure allows multiple unrelated variables to be stored in the same cache line and multi-core CPUs can share cache lines across multiple threads. The cache coherence policy will force the cache line to be updated whenever any variable in that cache line is updated. As an example, if thread 1 is using variable A and thread 2 is using variable B, but both variables are on the same cache line, then whenever A is updated by thread 1 it will force thread 2 to update the cache line, even though thread 2 is not using variable A. The same is true in reverse. If thread 2 updates variable B, then the cache coherence policy will force thread 1 will update the cache line even though thread 1 is not using variable B. Cache line sharing is a critical problem because it forces extra cache line reads, which stall the calling thread while the thread waits for the updated data to be retrieved.

This cache line sharing can have a dramatic impact on performance, which can again be demonstrated through a simple experiment (modified from [83]). We create a large array of data, and a varying number of threads, from 1 up to n, where n is equal to 2 times the number of processors recognized by the operating system. We assign each thread a location in the array on which to perform some work. Since arrays are stored in contiguous memory, if we assign threads 1 through n indexes in array from 0 through $n-1$, the threads will all be working with data on the same cache line. If we assign each thread a location in the array index of 0 through $(n-1)*$ variables in a cache line then each thread will be working with data on separate cache lines. The expectation is that the execution times when the threads are working on the same cache line will be much longer than the execution times when the threads are working on separate cache lines.

This experiment was performed on a machine running Ubuntu 15.04 with an Intel i7-880 CPU clocked at 3.08GHz and 8GB of RAM. This CPU has 4 physical cores that each support 2 hardware threads, the same architecture as Figure 26. The experiment was run 20 times and the computation times averaged. The results of the experiment are shown

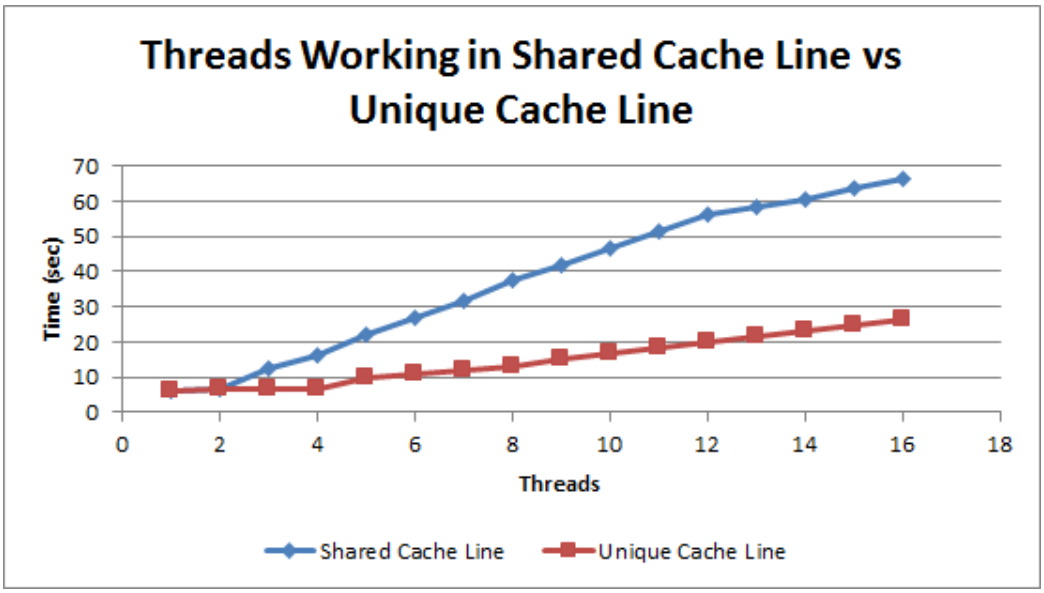


Figure 29: Results of an experiment comparing with multiple threads working on data that exists in the same cache line to multiple threads working on data that exists in separate cache lines.

in Figure 29. The results show that as the number of threads increases the computation time increases as well, nearly linearly in terms of the number of threads. The results also show very little increase in the computation time when using 1 through 4 threads. This is expected as the threads are working on independent cache lines and do not interfere with each other. The large jump in computation time above 4 threads is likely due to hardware limitations of Intel's hardware multithreading implementation (marketed as Hyperthreading by Intel [7]). There may appear to be 8 unique CPUs to the operating system, but the physical hardware cannot accurately mimic 8 independent cores when there are only 4 physical cores to work with. When using 4 threads, the method using independent cache lines is 2.9 times faster than the implementation using a shared cache line.

As this example shows, sharing cache lines between threads can lead to significant performance problems. Programmers need to consider the processor cache, and avoid cache line sharing, when designing multithreaded applications to ensure program performance.

V.3 Summary

This chapter reviewed Flynn's taxonomy for parallel architectures. Modern engineering workstations are an implementation of the Multiple Instruction, Multiple Data (MIMD) architecture because they contain CPUs that have multiple independent processing cores that perform their own instructions. Further, modern engineering workstation CPUs share system memory. GPUs were also briefly discussed, and they are a variation of a Single Instruction, Multiple Data (SIMD) architecture, with small pieces of their architecture being MIMD. GPUs are being used more frequently in scientific research, but our focus is on improving the performance of simulations on engineering workstations where high-performance GPUs are not very common.

This chapter also reviewed the architecture of Multi-Core CPUs, with the review split into a brief review of the important parts of the processor for our purposes, and a longer review of the processor cache. The cache review covered topics such as the cache hierarchy, and cache lines. We also presented examples showing the need for a strong cache coherence policy in the chip design, and one method of implementing that coherence policy through a snooping protocol. We presented experimental results (based on algorithms presented in [83]) that showed the effects cache line reads can have on program performance, the difference in access times of the different layers of cache, and the supremely negative impact cache line sharing can have on multithreaded program performance.

CHAPTER VI

PARALLEL SIMULATION OF ORDINARY DIFFERENTIAL EQUATIONS

The goal of this work is to reduce the simulation run time of a system whose dynamic model is made up of a set of ordinary differential equations (ODEs) (ODEs are defined in Equation III.1):

$$\dot{\mathbf{x}}(t) = \mathbf{f}_x(t, \mathbf{x}(t), \mathbf{w}(t)), \tag{VI.1}$$

where $\mathbf{x}(t)$ is a vector of the state variables of the system, with a corresponding derivative vector $\dot{\mathbf{x}}(t)$. $\mathbf{w}(t)$ is the set of algebraic variables, which includes the input variables. n_x is the number of state variables, and n_w is the number of algebraic variables [77]. The vectors $\mathbf{x}(t)$ and $\dot{\mathbf{x}}(t)$ have the same dimensions equal to n_x. The functional form $\mathbf{f}_x$ specifies the values of the derivatives of each state variable, which when integrated provides the evolution of the system behavior, $\mathbf{x}_t$ over time. The continuous time ODE set, $\mathbf{f}_x(t, \mathbf{x}(t), \mathbf{w}(t))$, can be expressed in discrete-time form by plugging Equation VI.1 into a numerical integration method, the Forward Euler method from Equation IV.7 in this case [35]:

$$\mathbf{x}_{n+1} = \mathbf{x}_n + h \cdot \mathbf{f}_n(t_n, \mathbf{x}_n, \mathbf{w}_n) \tag{VI.2}$$

All of the parameters have the same meaning as the previous equation, except they are in terms of the current simulation step n instead of continuous time. It is possible that the discrete time function, $\mathbf{f}_n$ will be very different from the continuous time function $\mathbf{f}_x$. However, if the time interval between time steps n and $n+1$, represented by h, is small, the first order approximation above is quite accurate.

We will measure the simulation run time using the wall-clock time of the simulation.

Definition 29 (Wall-Clock Time)**.** *The wall-clock time of a simulation is the time it takes*

from an external user's perspective for the overall simulation to be completed, i.e. the time for a simulation to complete as measured by a clock on a wall.

We use the wall-clock time so that we can study the reduction in simulation time using parallel algorithms from the user's point of view. In a parallel program it is also possible to measure time as the aggregate of the busy time of each CPU, but this is not useful from an end user perspective. It is a better measure of the performance from a hardware use perspective. Our target parallel architecture is a modern multi-core shared-memory CPU, the standard processor in modern engineering workstations.

We measure the effectiveness of a parallel simulation as the relative speedup of the wall clock time for parallel simulation compared to a serial simulation. The equation to calculate the relative speedup is:

$$\text{rel. speedup} = \frac{t_{serial}}{t_{parallel}}. \tag{VI.3}$$

A value of greater than 1 implies that the parallel simulation provides a speedup, and a value of less than 1 indicates that the parallel code runs slower than the serial code. The simulation times, t_{serial} and $t_{parallel}$ are the measured wall-clock times for the two simulations.

One important point to note from equation VI.1 is that the calculation of any value in $\dot{\mathbf{x}}(t)$ does not depend on any other values of $\dot{\mathbf{x}}(t)$, because no value of $\dot{\mathbf{x}}(t)$ appears as inputs to the function $\mathbf{f}_x$. This means that the calculation of the specific values in $\dot{\mathbf{x}}(t)$ can happen in any order. This fact allows us to divide the functions represented by $\mathbf{f}_x$ into subsets, where each subset contains one or more of the equations to compute the derivatives, and is assigned to one of the execution threads on a multi-core CPU. These threads execute in parallel on the CPU, but at the end of each simulation time step the threads will have to synchronize their data to preserve the correctness of the simulation result. This approach

allows us to parallelize the computation of each time step in the simulation, and this should reduce the wall clock-time of the simulation.

As described in Chapter I our target research areas center on:

1. Developing parallel simulation algorithms that appropriately uses the multiple cores and the cache memory organization on a multi-core CPU, and

2. Running experimental studies that help us analyze the effectiveness of various multi-threading and memory management schemes for parallel simulations.

A parallel simulation algorithm that appropriately uses multiple cores and the CPU cache has conflicting goals, even though using multiple cores effectively and managing the cache both focus on the physical hardware of the CPU. An ideal program architecture from the standpoint of a multi-core CPU will involve a limited number of parallel computation threads, with very little sharing of data between the threads. An ideal program architecture from the standpoint of the CPU cache might be to divide the program into very small pieces so that the data being worked on by each thread fits entirely into the size of one cache line. These two ideal architectures are often in conflict with each other, and finding the right balance between the two is the key aspect of our research.

Partitioning the system of equations, such that the cache can be used effectively to minimize the communication between the execution threads was a key factor that drove the design of the simulation algorithms presented in this chapter. We focus on (1) Minimizing the communication between execution threads, and (2) Utilizing the fastest communication methods for exchanges that have to take place. We leverage the shared-memory architecture in modern multi-core CPUs so that communication between threads can be handled in hardware as a part of the processor's cache (parallel architectures and processor cache are covered in Chapter V). However, we design our algorithms so that they share only a minimum amount of data, as sharing data through the cache across threads also causes computation delays, as we demonstrated in Section V.2.2.

The characteristics of the Operating System (OS) also play a role in our algorithm development. The OS allocates the computation threads to physical CPU cores, so the size of the partitions for the model, which directly determines the number of computation threads created is an important consideration, especially if we want to avoid too much thread swapping in the CPU cores. ODE simulation requires tight synchronization across all of the computation threads at the end of every time step in the simulation, which influences the way each thread is programmed.

Partitioning any computational problem into independent components is the first step to solving the problem in parallel [52]. However, we leave the problem of finding an optimal model partitioning to future work, and will instead rely on simple heuristic partitioning schemes to match the multi-core processor architecture for our experiments. This has been a non-trivial problem, but our work represents good progress in that direction. The results derived here can influence the algorithms we develop in the future for optimal partitioning. In this work, we investigate the number of parallel threads that produce optimal run-time performance.

The simulation programming language used for this study is C/C++, compiled and run on Linux. C/C++ simplifies the programming task and generates very efficient execution code. Further, C/C++ also has low-level memory management functions that allow cache-aligned data structures to be created. We target Linux as a simulation environment because it provides more control over the created threads, and generally faster execution than Windows.

This chapter outlines the experimental studies we ran for empirically analyzing the simulation time of large ODE models. Section VI.1 describes the models that we have used to perform our experiments. Section VI.2 describes our simulation algorithms and the execution of code flows we developed to test the parallelization. For each algorithm, we also identified the overhead introduced, and looked for ways to improve program performance. Section VI.3 describes the mathematical basis for partitioning a set of ODEs into subsets

suitable for parallelization. Section VI.4 presents our base serial simulation to which we compared our parallel simulations. Section VI.5 describes our parallel simulation algorithms. Section VI.6 describes techniques that we used to improve the run time of our C++ code. Section VI.7 describes our experimental results and a discussion of those results. Finally, Section VI.8 summarizes this chapter.

VI.1 Models

We used large sets of cascaded RLC circuits and a simplified version of NASA's Water Recovery System (WRS) [43] to evaluate our parallel simulation algorithms. The RLC models are described in Sections VI.1.1 and VI.1.2, and the WRS is described in Section VI.1.3.

VI.1.1 RLC Models

The RLC circuit models were implemented to exploit of Modelica's hierarchical nature (see Chapter II), which allows for efficient construction of models from components. The base component is shown in Figure 30. The inductor and capacitor implementations are from the Modelica Standard Library (MSL) [14]. The resistor is a modified version of the MSL resistor model. The thermal components were removed from the model because they contain Modelica `if` statements, which can lead to hybrid behavior. The terminals (ports) associated with the component are also specified in the MSL. This set of base components were used to create a larger component with six connected base components as shown in Figure 31. The MSL electrical terminals were not altered. This larger component was used to create two of the models we used for our experiments. We created a model with 288 state variables that used 24 of the components shown in Figure 32, and we created a model with 804 state variables that used 67 components as shown in Figure 33 [72].

A third model that effectively connected 500 of the base components in series to create a model with 1000 state variables (see Figure 30). This serial configuration gives the model

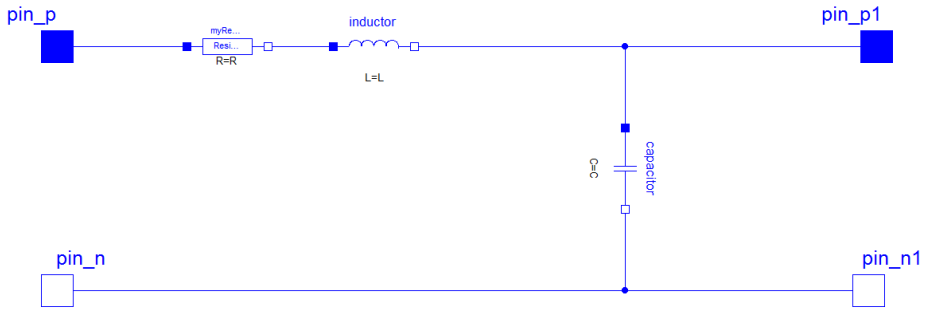


Figure 30: Base RLC component model with 2 state variables.

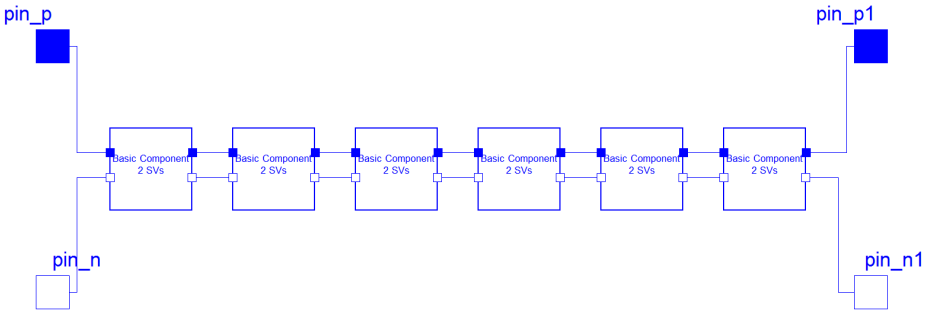


Figure 31: RLC component model with 12 state variables.

a much more regular structure than the models in Figures 32 and 33. This model requires much less shared data between the hardware threads, which should speed up simulation execution time. To construct the regular structure model we created a component with 25 state variables, shown in Figure 34, that duplicated the base RLC component in Figure 30 with 2 state variables 12 times and added an extra resistor and capacitor. We then duplicated that model 10 times to create a new component with 250 state variables, shown in Figure 35. This 250 state variable component was used to create the regular structure model shown in Figure 36.

VI.1.2 RLC Models with Algebraic Loops

Algebraic loops are very common in modern complex models (Section III.6). To evaluate the performance of parallelizing the simulation of a model with algebraic loops, we

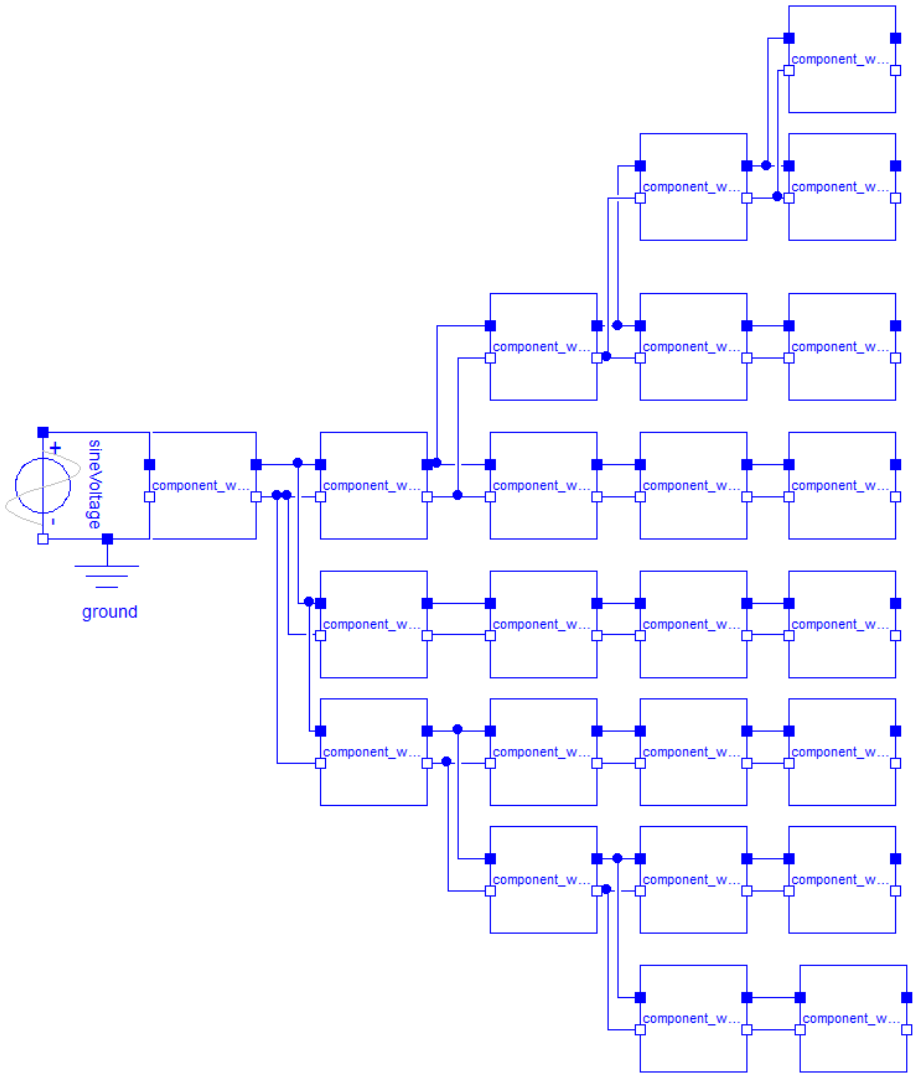


Figure 32: Complex RLC model with 288 state variables.

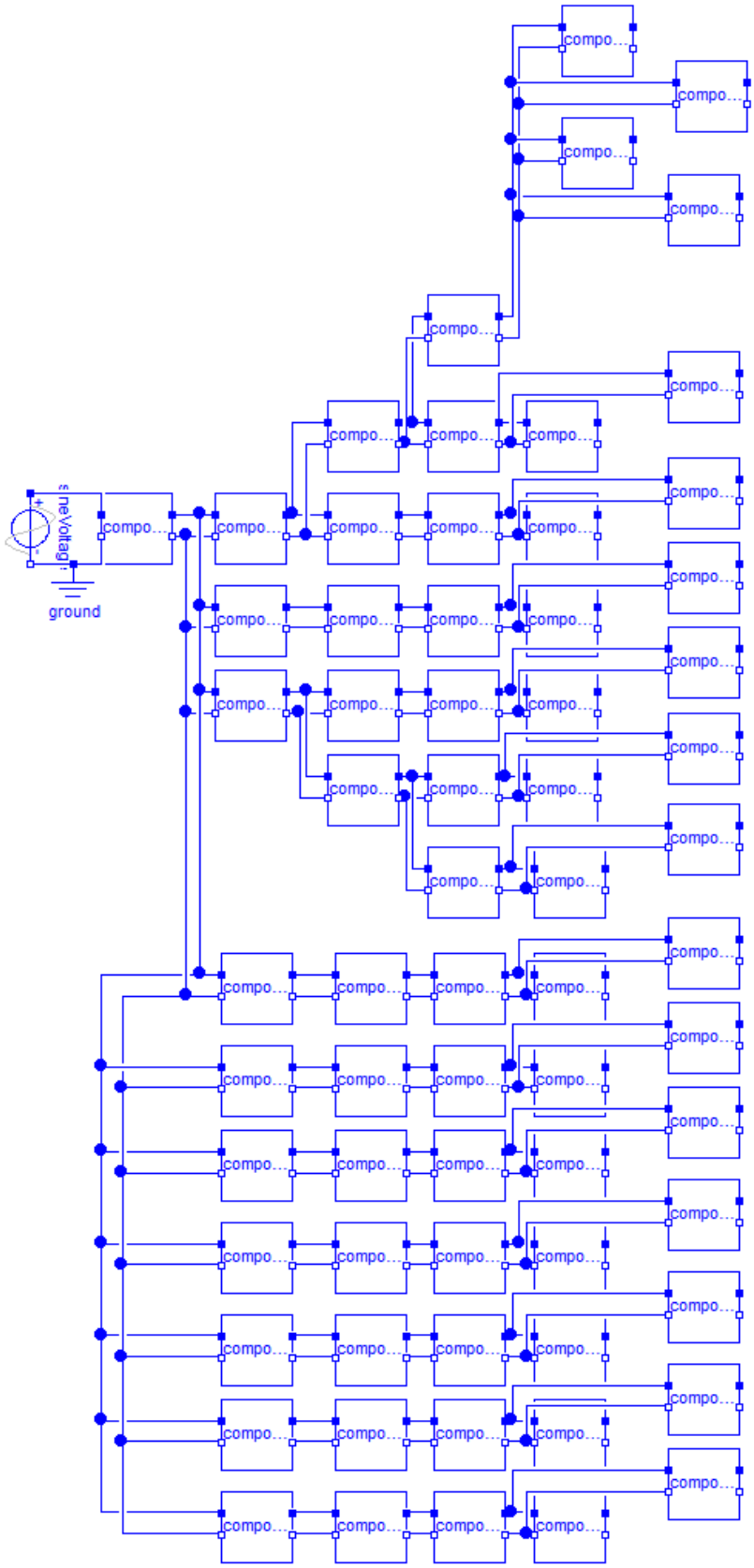


Figure 33: Complex RLC model with 804 state variables.

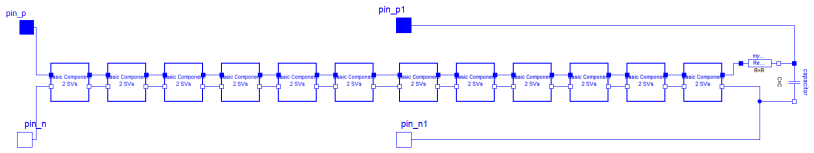


Figure 34: RLC component model with 25 state variables.

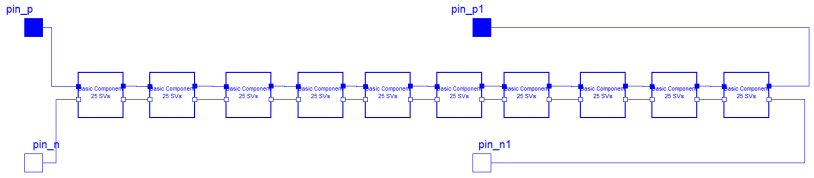


Figure 35: RLC component model with 250 state variables.

modified the models in Figures 32, 33, and 36 to include components that introduced algebraic loops. We followed the same approach to building the models with algebraic loops as we did for the models described above: we used Modelica's hierarchical structure to build complex components from the base components. We created 2 new base components, one with a linear algebraic loop and one with a non-linear algebraic loop. They are shown in Figures 37 and 38, respectively. The algebraic loop is introduced due to the series resistors ([27] also has an example of series resistors causing algebraic loops), and, as a result, in these circuit fragments there is no way to uniquely determine the voltage drop across each resistor. Therefore, the voltage drop across the resistors must be calculated simultaneously because the voltage across each resistor depends on the voltage across every other resistor. The non-linear loop is due to one of the resistance values in the loop being a non-linear function of the voltage across another resistor in the loop.

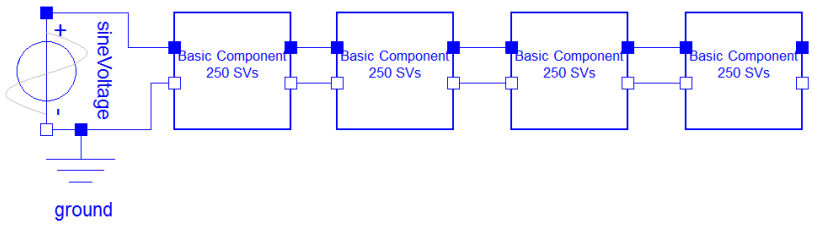


Figure 36: RLC with regular structure and1000 state variables.

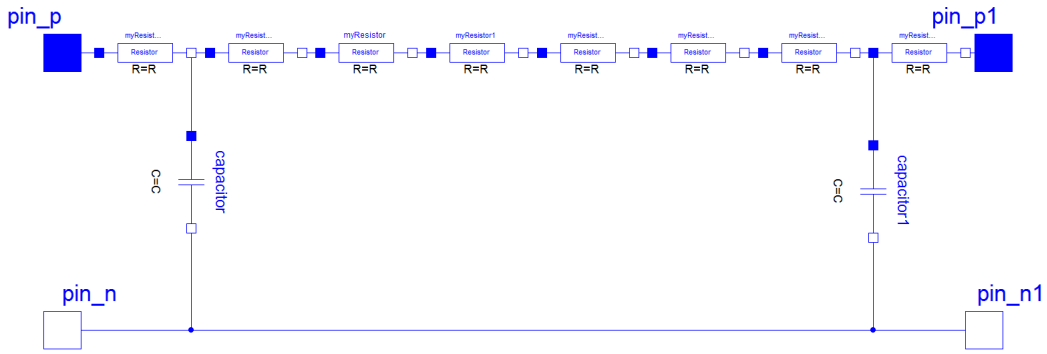


Figure 37: Base RLC component model with 2 state variables and a linear loop.

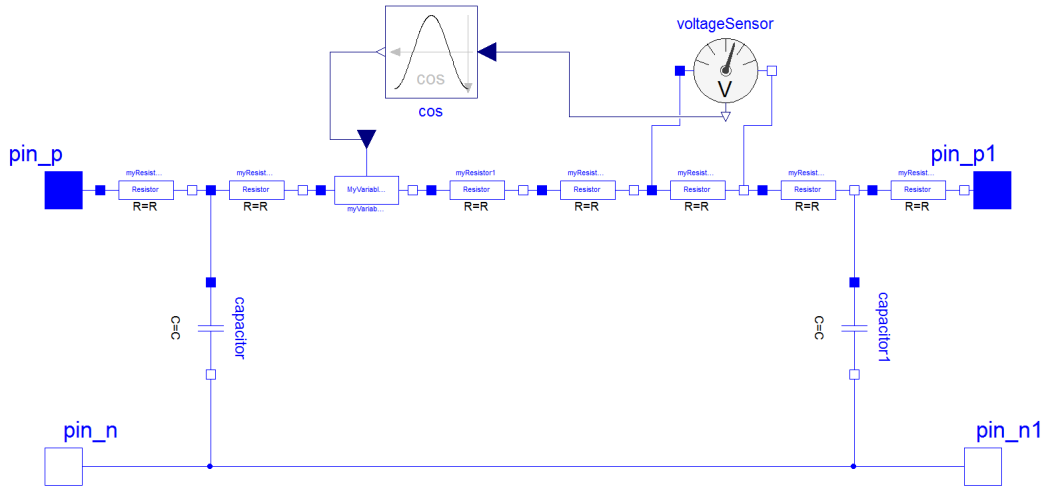


Figure 38: Base RLC component model with 2 state variables and a non-linear loop.

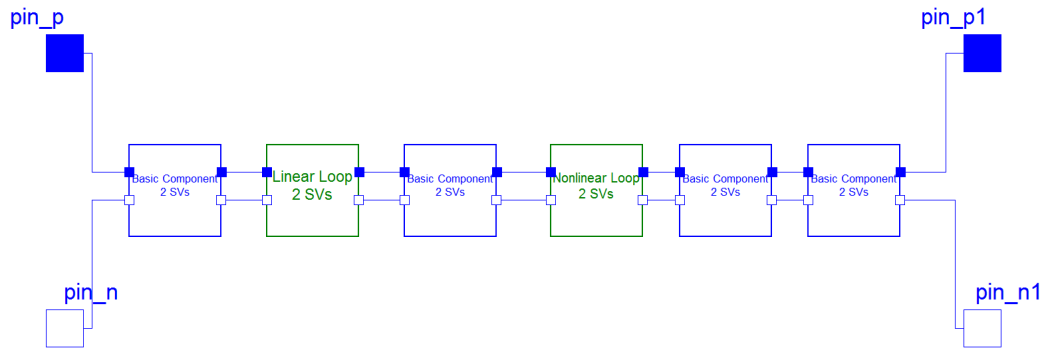


Figure 39: RLC component model with 12 state variables and 2 algebraic loop components.

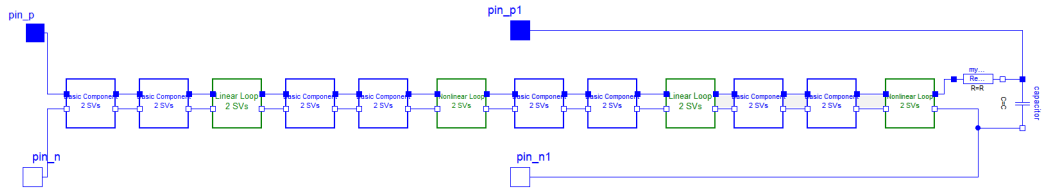


Figure 40: RLC component model with 25 state variables and 4 algebraic loop components.

To create components that can be used to build large test circuits we replaced some of the components in Figure 31 with the components from Figures 37 and 38, that introduced loops, into all of the previously described components to create the components shown in Figures 39, 40, and 41. In these figures, the boxes in green are components with algebraic loops and the boxes in blue are components without loops. We used the basic loop components in Figures 39, 40, and 41 to create the models with loops that we used in our experiments by replacing some of non-loop components with loop components. These new models are shown in Figures 42, 43, and 44. The loop components we added were split approximately evenly between the linear loop components and non-linear loop components.

We used two sets of parameter values in our experiments. The first set of values set all parameters equal to 1, and the resulting electrical circuits have slow time constants. A parameter value of 1 for all an electrical components of a circuit is not realistic for most

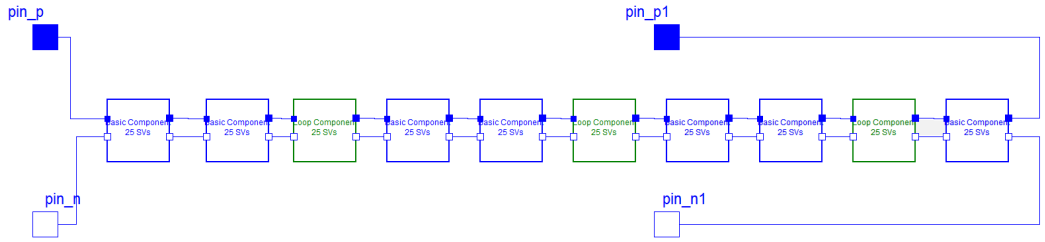


Figure 41: RLC component model with 250 state variables and 3 algebraic loop components.

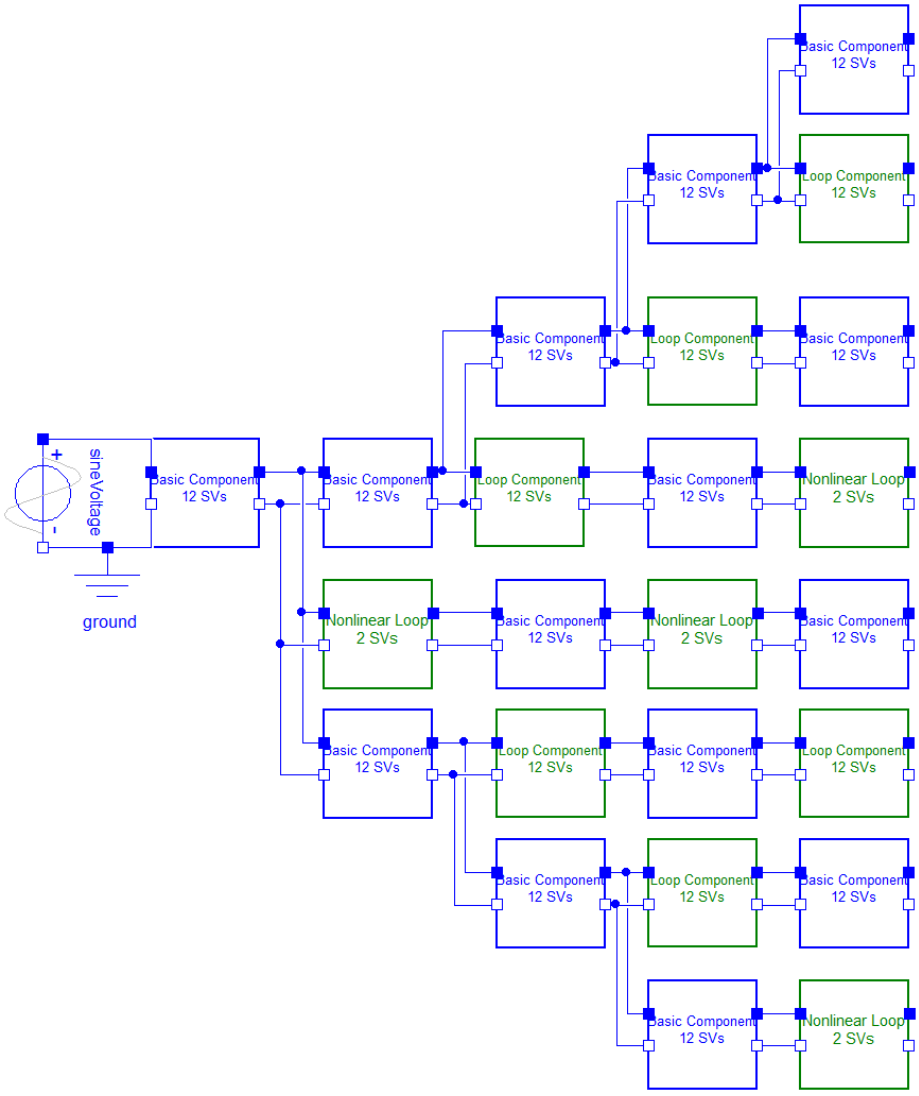


Figure 42: Complex RLC model with 288 state variables and 19 algebraic loops (9 linear and 10 non-linear).

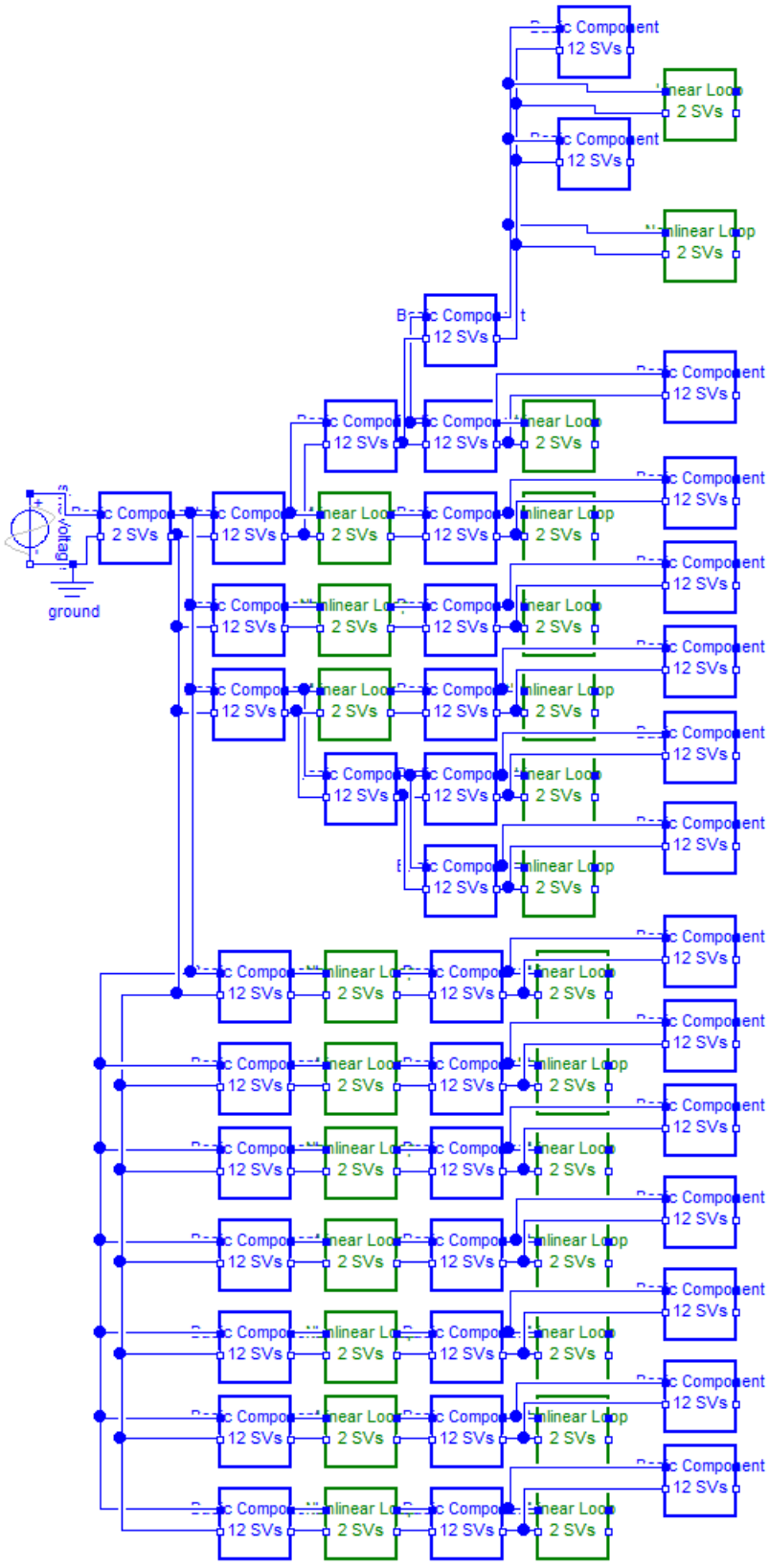


Figure 43: Complex RLC model with 804 state variables and 26 algebraic loops (14 linear and 12 non-linear).

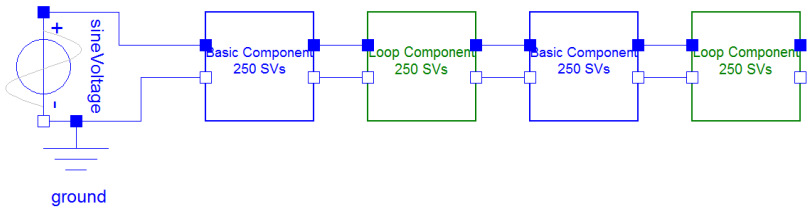


Figure 44: RLC regular structure model with 1000 state variables and 30 algebraic loops (18 linear and 12 non-linear).

applications, but it was useful for an initial evaluation of a parallel algorithm. We also used more realistic parameter values of 15Ω for all resistors, 15mH for all inductors, and 250μF for all capacitors. The circuits using these parameters had much faster time constants. In our experiments the complex models, Figures 32, 33, 42, and 44, were tested with both sets of parameter values. The regular structure models, Figures 36 and 44, were only tested with the fast parameter values in order to ensure a measurable change in state variables farthest from the voltage source.

VI.1.3 Water Recovery System

We tested NASA's Water Recovery System (WRS) [43] as a real world model. The WRS recycles waste water into potable water, and was designed to support long duration space missions. It is composed of four major sub-systems: the Biological Water Processor (BWP), the Reverse Osmosis system (RO), the Air Evaporation System (AES), and the Post Processing System (PPS). An overall diagram of the WRS is shown in Figure 45. The WRS has several modes of operation, but we limit ourselves to a single operating mode for each sub-system to avoid the complexities of simulating hybrid behavior. We are also primarily interested in simulating the hydraulic, mechanical, and thermal physical domains, and do not focus on modeling the water quality.

The Biological Water Processor (BWP), Figure 46, draws waste-water from a storage tank and removes organic compounds. A pump pulls water from the waste-water tank and sends it into the recycling loop. The water in the loop passes through two different reactors,

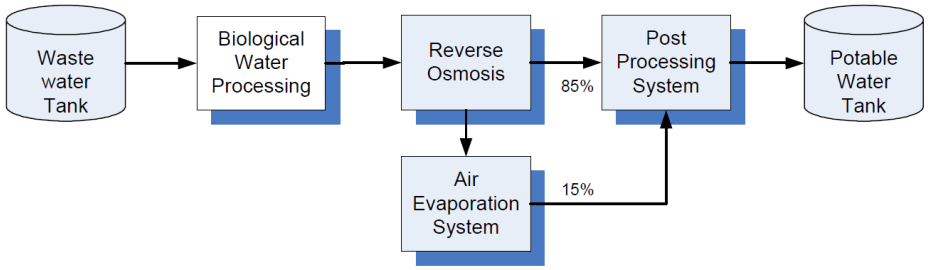


Figure 45: Block diagram of the Water Recovery System [43].

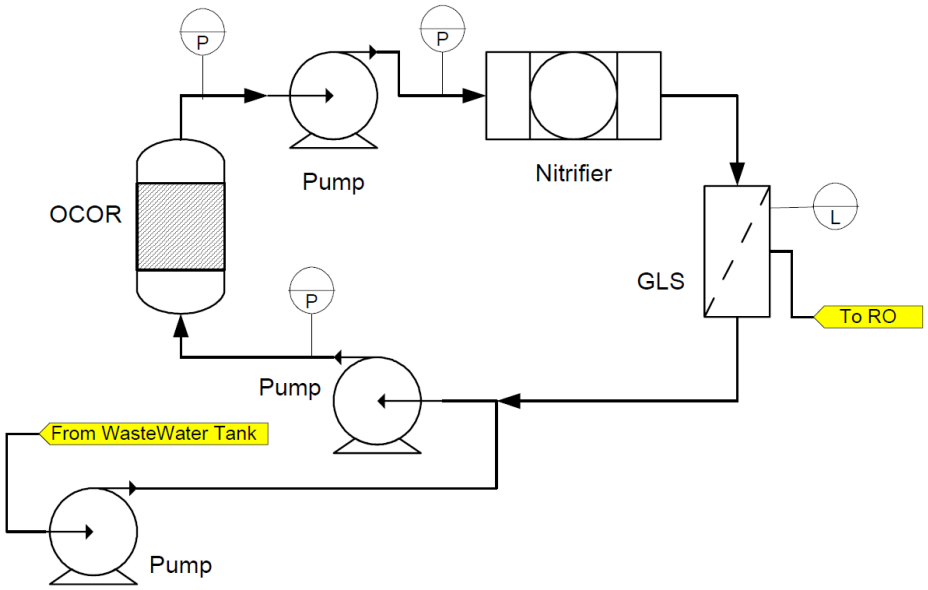


Figure 46: Diagram of the Biological Waste Processor sub-system of the WRS [43].

the Organic Carbon Oxidation Reactor (OCOR) and the nitrifier. Output from the reactors is sent to the Gas Liquid Separator (GLS). Output from the GLS is split into two feeds. One feed goes to the RO system for further processing, and the other feed is sent back through the recycling loop.

The Reverse Osmosis (RO) system, Figure 47, removes inorganic compounds through a filter. The RO system has 3 operating modes that correspond to the setting of a multi-position valve; however, in our work we limit the model to the first mode of operation. In this first mode of operation effluent from the BWP is combined with the water already in the system in a coiled section of pipe that acts like a resevior. A pump cycles the water into

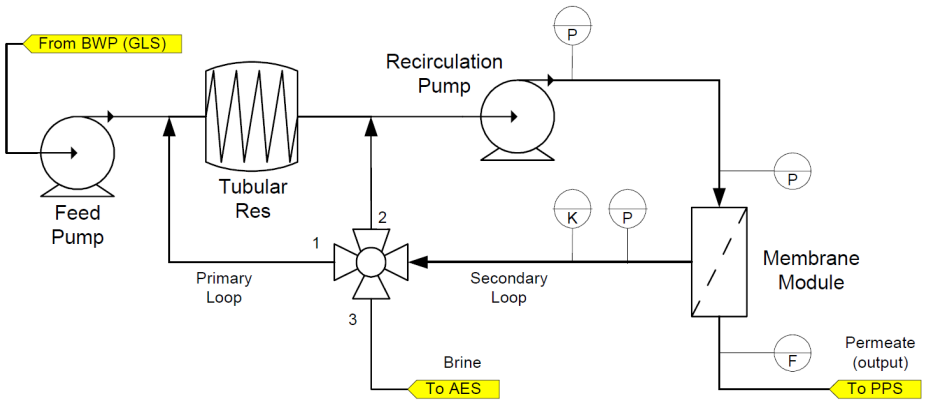


Figure 47: Diagram of the Reverse Osmosis sub-system of the WRS [43].

the membrane module. The membrane module has two output feeds. Clean water is sent to the Post Processing System and the water that was not cleaned is cycled back through the system, causing the water to progressively become more dirty. Eventually the membrane efficiency decreases and the system switches to the second mode. In this mode the loop is shortened so that the reservior is avoided, and to maintain pressure the recirculation pump speed is increased. Eventually, the filter is no longer effective and the system purged (mode 3) by sending remaining dirty water to the Air Evaporation System to be cleaned. About 85% of the water in the RO is sufficiently cleaned by the membrane to be sent to the PPS. The remaining 15% that cannot be cleaned through a filter and is sent to the AES.

The Air Evaporation System (AES), Figure 48, operates as a cycle of multiple heat exchange processes. A reservoir collects the brine from the RO system, where it is absorbed by a wick and evaporated by blowing hot air over the wick. The evaporated water is condensed by passing it through a heat exchanger and collected before it is sent to the PPS.

The Post Processing System (PPS) treats the water through a multi-step process. The hydraulic and mechanical domains of the system are very simple because it has no explicit internal storage capacity or mechanical pumps to drive the water; therefore, no diagram is included to describe the system. The PPS removes any remaining trace contaminants to bring the water quality to potable level.

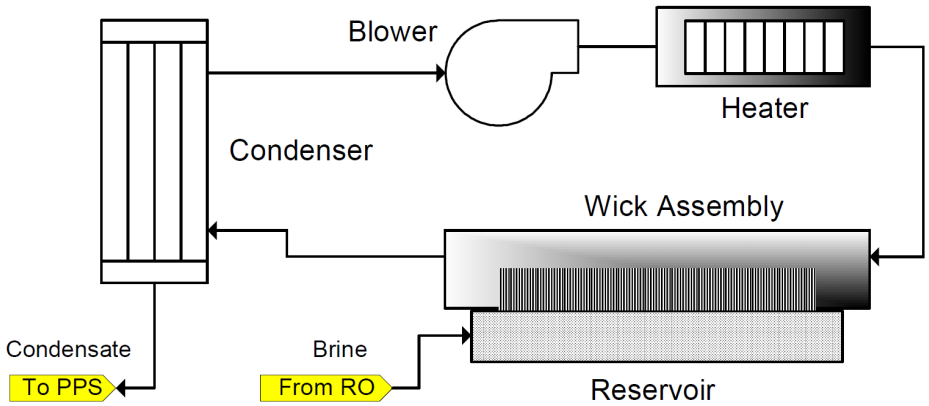


Figure 48: Diagram of the Air Evaporation System of the WRS [43].

We modeled the WRS system as a flat bond graph [63], that is, without a system component hierarchy, in the Generic Modeling Environment (GME) [65][5]. We then used an interpreter to convert the bond graph to a set of equations in Modelica syntax that can be simulated by Dymola. The system has 24 state variables and 4 algebraic loops.

VI.2 Executing the Models

The state equations can be integrated independently and in any order for each time step (Section III.5.4). This gives us freedom, within a time step, to group the state equations in ways that achieve the best run time performance. At the end of a time step, the updated state variable values will need to be synchronized across the state equations of the system, to allow the different execution threads to acquire the updated state variable values before starting the calculations for the next time step.

We implemented two types of integrators for the parallel simulation algorithms: (1) a fixed-step Euler integrator, and (2): a variable-step Runge-Kutta integrator. We discuss our implementations for the two integration schemes next.

VI.2.1 Fixed Step Integration

We used an Inline Forward Euler (IFE) integration method [35] (Equation IV.7) for all of our fixed step simulations. Each step of a Forward Euler (FE) integrator in discrete time is described as:

$$\mathbf{x}_{n+1} = \mathbf{x}_n + h \cdot \dot{\mathbf{x}}_n, \tag{VI.4}$$

where h is the step size. Inlining the integration equation allows our simulation to calculate $\mathbf{x}_{n+1}$, in a single equation instead of two equations one calculating the derivative of the state variable, $\dot{\mathbf{x}}_n$, and then a second calculating the value of the state variable at $\mathbf{x}_{n+1}$, as is traditionally done in simulation (see Section IV.1). In this case, inline integration works by taking the equation for ODEs, Equation VI.1, which solves for $\dot{\mathbf{x}}(n)$ and substituting that equation into the equation for FE integration, Equation VI.4, i.e.,

$$\mathbf{x}_{n+1} = \mathbf{x}_n + h \cdot \mathbf{f}_x(t_n, \mathbf{x}_n, \mathbf{w}_n). \tag{VI.5}$$

We simplify the definition of this equation to be:

$$\mathbf{x}_{n+1} = \mathbf{f}_{IFE}(h, \mathbf{x}_n, \mathbf{f}_x(t_n, \mathbf{x}_n, \mathbf{w}_n)) \tag{VI.6}$$

During simulation using the IFE method, for each time step the simulation calculates the values for $\mathbf{x}_{n+1}$ during time step n. After all of the values for $\mathbf{x}_{n+1}$ are calculated the simulation copies $\mathbf{x}_{n+1}$ into $\mathbf{x}_n$:

$$\mathbf{x}_n = \mathbf{x}_{n+1}. \tag{VI.7}$$

and then advances the simulation time:

$$t_n = t_n + h, \tag{VI.8}$$

where t is the simulation time and h is the time step. Then the simulation loops and repeats the process until the simulation reaches the simulation end time. In our fixed step simulations the step size of the simulation is passed as a parameter to the simulation at run-time.

The simulation algorithm using IFE integration is described in Algorithm 3.

Algorithm 3 Algorithm describing simulation using Inline Forward Euler integration.

1: **while** $t_n < StopSimulationTime$ **do**
2: $\quad \mathbf{x}_{n+1} = \mathbf{f}_{IFE}(h, \mathbf{x}_n, \mathbf{f}_x(t_n, \mathbf{x}_n, \mathbf{w}_n))$
3: $\quad \mathbf{x}_n = \mathbf{x}_{n+1}$
4: $\quad t_n = t_n + h$
5: **end while**

VI.2.2 Variable Step Integration

We used an explicit Runge-Kutta-Fehlberg 4,5 (RKF4,5) solver ([49] and described in Equation IV.10) from the GNU Scientific Library (GSL) [25] for our variable step simulations. The GSL implements the RKF4,5 as a stand-alone solver that follows a traditional approach to simulation, as described in Section IV.1 and shown in Figure 16. To use the solver we supply a function for calculating the derivatives of the state variables, Equation III.1, and a function for calculating the system Jacobian matrix. The solver determines the time step size, performs the integration, and maintains the synchronization between time steps. Since this is a variable step solver, we supply an output interval that indicates how frequently we want to receive updates to the system state.

The simulation algorithm using the RKF4,5 solver is described in Algorithm 4.

VI.2.3 Pre-processing: Code Flow

This section describes, at a high-level, the steps required pre-process a differential equation based state-space model to generate the C++ simulation code. A flowchart illustrating

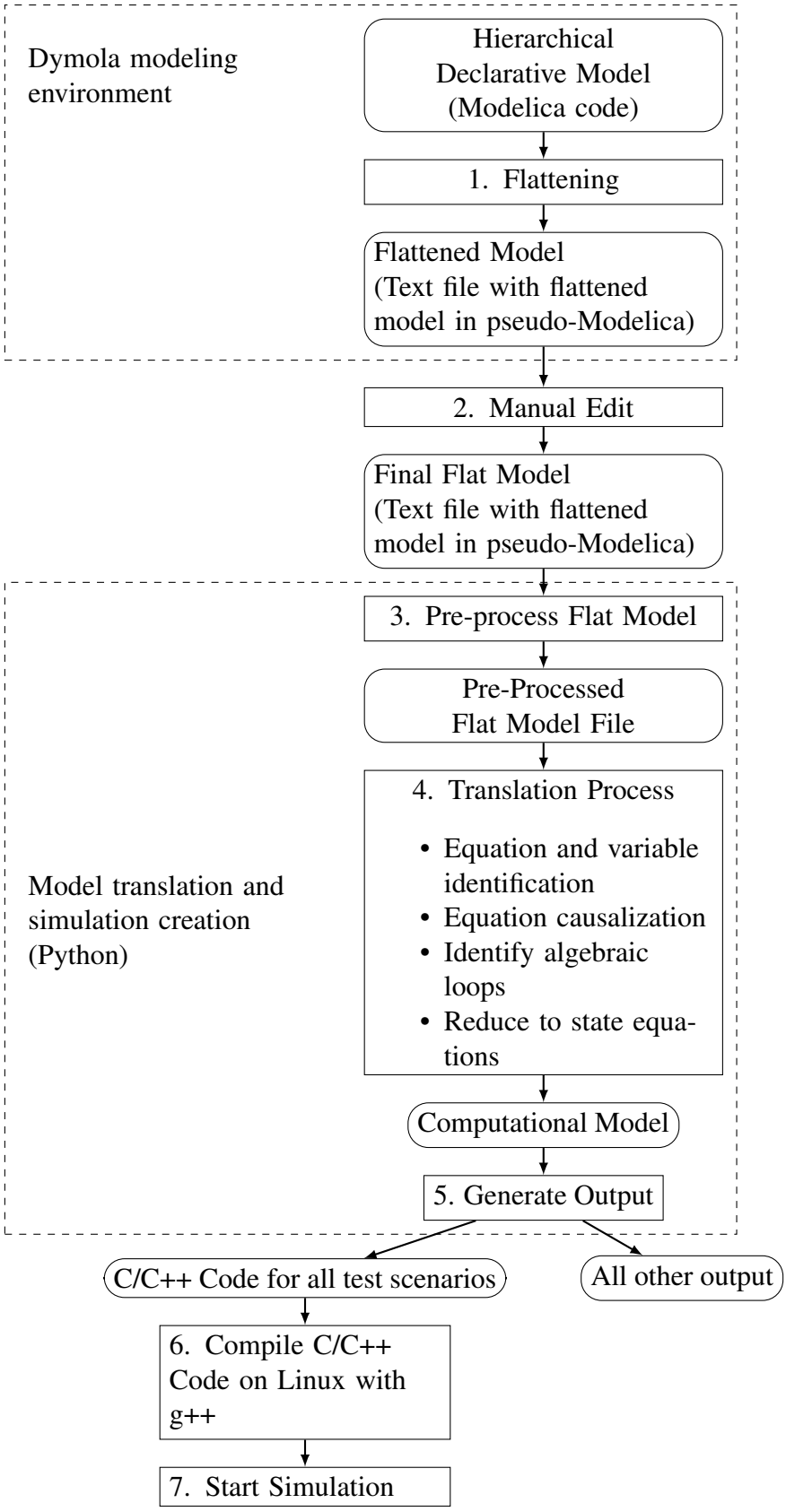


Figure 49: High-level flowchart of the model progression from initial declarative model to compiled simulation.

Algorithm 4 Algorithm describing simulation using a variable step RKF4,5 solver.

1: $current\_time = 0$
2: $output\_time = output\_interval$
3: **while** $current\_time < StopSimulationTime$ **do**
4: $\quad RKF45\_Integrate(\mathbf{f}_x, \text{from} = current\_time, \text{to} = output\_time, \text{RKF45 output} = \mathbf{x}_n)$
5: $\quad current\_time = output\_time$
6: $\quad output\_time = output\_time + output\_interval$
7: **end while**

the process is shown in Figure 49. In the chart, the rectangular boxes with sharp corners represent an action, and rectangular boxes with rounded corners represent outputs.

The very top dashed box includes all steps that are computed using Dymola, a commercial simulation tool. The rest of the algorithm and the accompanying C++ code was developed as a part of this thesis work. This part of the algorithm is included in the Generate Output box. The algorithms for translating the model in the model translation and simulation creation box are all known and defined in literature, and described in Section III.5.

The declarative hierarchical model was created using the Modelica modeling language in Dymola [4]. The Dymola software is used to flatten the model (box 1 in Figure 49). The flattening process was described in Section III.3. The result is a flattened model in Modelica-like syntax [27], and output as a text file. This text file is then manually edited (box 2) to remove any Modelica language constructs that do not affect the model behavior and are not supported by the model translation and simulation creation code. The models described in Section VI.1 use a sinusoidal voltage driver that introduces calls to external functions and an if-statement. The external function is a call to a C-function that is used to calculate the sine value. This is not needed because we directly call the sine function from the C++ math library. An if-statement is used to delay the start of the sine wave generation through an offset parameter. The logic for the if statement in the sine voltage driver is shown in Algorithm 5. We do not use the offset in our experiments, so we remove the if statement in the flat Modelica code. After the manual edits are complete, the flattened

model is ready to be read and compiled by the model translation and simulation creation code.

Algorithm 5 Algorithm describing the logic of the if statement in the sine voltage component of the MSL.

```
1: if simulation_time < offset then
2:     output_voltage = default
3: else
4:     output_voltage = sin(...)
5: end if
```

The model translation and simulation creation code from Figure 49 is written in Python [16] and is used to convert the model from a flat file in pseudo-Modelica, to C++ code ready for simulation. The translation code we have written can only process simple equations, described using Modelica syntax, because we wanted to focus on parallel simulation of ODEs and not on implementing a new Modelica language parser. There are many elements of the Modelica language, such as tables, if statements, when statements, arrays, and algorithm sections, that our code does not support in an effort to maintain that focus.

The Python model translation code first reads the flat file model and processes it, box 3 in Figure 49, to create an intermediate flat file format that is compatible with SymPy [21], the Python library we use for symbolic equation manipulation. Specifically this involves removing the dot-operator from the variable names in the flat Modelica-like code and replacing it with a different character; in our case we use an underscore ("_"). In Modelica, and in the Modelica-like syntax, the dot-operator is used to identify the model hierarchy for each variable. In the flattened model the dots in the variable names are irrelevant. In Python, like in C/C++, the dot-operator is used to access member variables and functions in a class object, so the dot-operator needs to be removed so that the symbolic math library does not try to access member variables and functions that do not exist.

After the pre-processed file is created, it is read by the translation process, box 4 in

Figure 49, that implements the process described in Section III.5 (more details on the procedure are described in that section). First the equations and variables are processed and a mapping is created that identifies the variables used in each equation (Section III.5.1). Then the equations are causalized to identify which equation solves for each variable (Section III.5.2). After this the equations are sorted to determine the partial order in which they need to be solved, which creates a directed graph, and any algebraic loops in the model are identified (Section III.5.3), which creates a directed acyclic graph. After the algebraic loops are identified, the model is reduced to a set of ODEs (Section III.5.4). The ODE form of the model, Equation VI.1, represents the computational model.

The computational model is used to generate all of the model output, box 5 of Figure 49. The generated files contain all of the C++ code for compiling and running the parallel approaches described in Sections VI.5. We use g++ to compile the C++ code, box 6, and we run our simulations using a Linux Bash script, box 7.

VI.2.4 Overhead in Running Parallel Simulations

In any parallel program the parallelization adds overhead that is not present in a serial execution of a program. The number of computations per time step of the simulation is the same for sequential and parallel algorithms. However, the overhead generated by parallelization can limit the effectiveness of the parallel implementation. This overhead typically takes the form of scheduling overhead and communication overhead. The created threads need to be assigned, by the operating system, to a CPU core during the thread runtime. It is the responsibility of the OS to ensure that all threads are given enough time on a CPU core to complete their work. There is also overhead due to communication between the created threads. This communication overhead does not occur in a single threaded program, but it is necessary in a parallel program synchronization. The run time of a parallel program can

be described at a high level as:

$$\text{wall clock time} = \text{computation time} + \text{overhead}. \tag{VI.9}$$

The challenge in designing parallel software is to minimize the overhead component of Equation VI.9. This will generally involve two components: (1) the overhead required to manage shared memory between the different execution threads, and (2) the amount of swapping that needs to be performed when there are more threads than available processor cores. These two components are not independent of each other, and we describe a number of algorithms that trade off these two parameters.

VI.2.5 Experiment Configuration

Unless otherwise noted, all experiments were run on an Intel i7-880 desktop PC clocked to 3.08GHz with 8GB of memory running Ubuntu 15.04. The generated C++ code was compiled using g++ version 4.9.2 [24]. All experiments were run 20 times on the models presented in Section VI.1 for parameter values that created slow and fast behavioral time constants. The experiments were run without writing state variable time trajectory data to disk in order to guarantee more consistent simulation times. To calculate the relative speedup for a particular algorithm and thread count, we calculate the average of the wall clock time for the serial simulation. Using this average we then calculate the relative speedup for each algorithm and thread count. The relative speedups are averaged for a thread count, and the standard deviation is calculated from the relative speedups.

VI.3 Mathematical Description of Model Partitioning

A key step in a parallel simulation is to partition the computational model into parts, such that each part can be executed on a separate processor. Therefore, we take the dynamic system model described in ODE form (Equation VI.1), and divide the system into ℓ sets,

where ℓ is the number of threads being used. In this work, we create the partition for the equations, such that about an equal number of state variable calculations are assigned to each thread [1].

$$\dot{\mathbf{x}}_n = \dot{\mathbf{x}}_{n\_0} + \dot{\mathbf{x}}_{n\_1} + \cdots + \dot{\mathbf{x}}_{n\_\ell} \tag{VI.10}$$

$$\mathbf{x}_n = \mathbf{x}_{n\_0} + \mathbf{x}_{n\_1} + \cdots + \mathbf{x}_{n\_\ell} \tag{VI.11}$$

$$\mathbf{x}_{n+1} = \mathbf{x}_{(n+1)\_0} + \mathbf{x}_{(n+1)\_1} + \cdots + \mathbf{x}_{(n+1)\_\ell} \tag{VI.12}$$

$$\mathbf{f}(t, \mathbf{x}_n, \mathbf{w}_n) = \mathbf{f}_0(t, \mathbf{x}_{n\_0}, \ldots, \mathbf{x}_{n\_\ell}, \mathbf{w}_n) + \mathbf{f}_1(t, \mathbf{x}_{n\_0}, \ldots, \mathbf{x}_{n\_\ell} + \mathbf{w}_n), \ldots, \mathbf{f}_\ell(t, \mathbf{x}_{n\_0}, \ldots, \mathbf{x}_{n\_\ell}, \mathbf{w}_n). \tag{VI.13}$$

Differences in the cardinality of the sets accounts for remainders in integer division of n_x/ℓ. Subsets $1, \ldots, (n_x \text{ modulo } \ell)$, will have a cardinality of $n_x/\ell + 1$, and subsets $(n_x \text{ modulo } \ell + 1), \ldots, \ell$ will have a cardinality of n_x/ℓ.

The value of ℓ is dependent on the simulation algorithm. For example, if $\ell = n_x$, there is one state variable per subset in each partition. For all of the other tests ℓ is going to have a value of m_{test}, where m_{test} represents the number of threads being used for a test and ranges from 1 to m, where m is the number of parallel threads on the CPU.

As discussed earlier, this partitioning approach allows us to complete the calculations within each time step in parallel, but it will require a synchronization phase at the end of each time step to guarantee correct results (Section VI.2). The partitioning, and the memory structure that supports the chosen partition, will differ for each of our parallel simulation algorithms, and the specific differences are highlighted in Section VI.5, which describes the parallel simulation algorithms and the results of the experimental runs with those algorithms.

[1] This partitioning may not work very well when the structure of the differential equations is not homogeneous.

VI.4 Base Test Case

The notion of speedup for a parallel simulation algorithm is always defined in terms of the runtime in comparison with a sequential algorithm that runs in a single thread on a processor. We focus on comparing our parallel simulation algorithms to the fastest and most basic serial simulation algorithm we created. Comparing a parallel implementation to a fast serial implementation of the same problem was advocated by [54], as a means to prove that the parallel implementation provides a practical benefit, and is simply not an academic exercise. The type of serial simulation, either fixed step or variable step, is matched to the type of parallel simulation we are using, so fixed step parallel simulations are compared to a fixed step serial simulation, and variable step parallel simulations are compared to a variable step serial simulation.

VI.5 Progression of Parallel Algorithms

This section presents the set of parallel simulation algorithms we developed in a progression using an agglomeration strategy. The different simulation algorithms varied on how many threads were created, and how the variables in the sets $\mathbf{x}_{n+1}$ and $\mathbf{x}_n$ were divided into blocks of memory corresponding to the created threads. We also paid special attention to minimizing memory sharing across threads, in an effort to maximize the performance of the CPU cache, and by extension the simulation speedup. The differences in memory sharing is a primary differentiator between our simulation algorithms, and a primary driver in reworking the thread assignments.

The set of threads used in each experiment is:

$$\mathbf{T} = \{T_0, T_1, T_2, \ldots, T_\ell, T_{main}\}, \tag{VI.14}$$

where $\mathbf{T}$ is the set of all threads in the experiment, T_0 through T_ℓ are the threads created for parallel execution of the particular simulation algorithm, and T_{main} is the primary thread

that spawns the child threads and controls the advance of the simulation time steps. The value of ℓ typically ranges between 0 and $m-1$, where m is the number of CPUs available to the operating system (on a processor that implements hardware multithreading that number of CPUs available to the OS is going to be double the number of cores on the processor, see Section V.2). As an extreme we also developed an algorithm where ℓ was set to the number of state equations (n_x) of the dynamic system model. We also define:

$$\mathbf{T}_{spawn} \subset \mathbf{T} = \{T_0, T_1, T_2, \ldots, T_\ell\} \tag{VI.15}$$

to identify the threads that were created by the main thread, T_{main}, for the simulation. Each algorithm creates one or more memory blocks that will be assigned to the threads. We describe the size of the memory blocks as:

$$M[X], \tag{VI.16}$$

where M represents a block of memory, and X is the size of that memory. We will also use the symbols $\rightarrow$, $\leftarrow$, and $\leftrightarrow$ to describe if a thread writes to a memory block, reads from a memory block, or reads and writes to a memory block, respectively. Each of the threads in $\mathbf{T}_{spawn}$ only communicates its status to T_{main} and does not have to communicate with any other thread.

Another factor that drove the implementation of our algorithms was to enable fast communication and simple synchronization between hardware threads, i.e. $\mathbf{T}_{spawn}$ and T_{main}. Synchronization is handled using shared variables. Simple spin locks are used at synchronization points to pause threads [29]. In a spin lock, a thread continually polls a variable waiting for it to change value, and provides for fast communication to handle synchronization between threads because all communication is handled in hardware through the processor's cache. As soon as the assigned variable is updated in the waiting thread's cache, a thread can continue its computation. An alternative to spin locks are semaphore or

mutex locking, but these locking schemes allow the operating system to move the waiting thread off of the core and replace it with a separate waiting thread. Re-starting the original thread will require not only monitoring the semaphore or mutex in question, but also requiring the OS to move the thread back onto the core. Whereas this may not be significantly impact computational efficiency for some applications, it can significantly deter the execution time of the parallel simulation algorithms, which require synchronization once or twice a time step. Since the simulation runs complete many time steps (up to 50,000 in some experiments) the extra work by the OS to put the thread back into active processing causes significant negative impact on the simulation. Spin locks, because the thread stays active, do not suffer from this problem.

This section presents three different simulation algorithms (Section VI.5.1, VI.5.2, and VI.5.3) and presents our investigations into optimizing software for speed (Section VI.6).

VI.5.1 Parallel Algorithm Type 1: One Thread Per State Equation, Full Shared Memory

Our first parallel algorithm creates a separate thread for each individual function in $\mathbf{f}_{IFE}$ (i.e., state equation + inline code for performing an integration step). Therefore:

$$\mathbf{T}_{spawn} = \{T_0, T_1, \ldots, T_{n_x}\}.$$

Even for moderately sized models, this results in many more threads than cores available on a typical multi-core CPU (typically there are 4 to 8 cores available [7][22]). This algorithm creates one shared memory block across all threads defined by:

$$M[2 \cdot n_x],$$

where M represents the block of memory, n_x represents the number of state variables in the system, and the size of the memory block in terms of the number of variables in the block

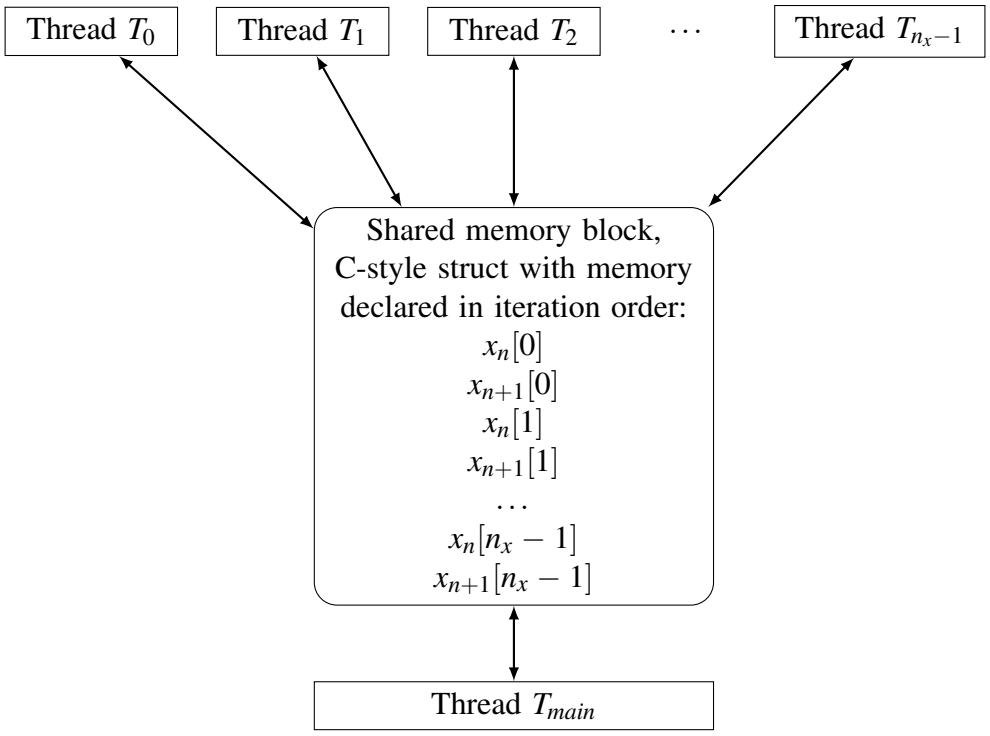


Figure 50: Figure describing shared memory implementation when creating one thread per state equation.

is $2 \cdot n_x$. The memory block is twice the number of state variables because there are two variables for each state variable: one to store the current time step value of the state variable ($\mathbf{x}_n$), and one to store the next time step value ($\mathbf{x}_{n+1}$). Each of the threads has read/write access to the single shared memory block:

$$\mathbf{T} \leftrightarrow M. \tag{VI.17}$$

The memory management approach used is shown in Figure 50.

This algorithm only changes line 2 of Algorithm 3, so that line 2 is solved in parallel by the threads in $\mathbf{T}_{spawn}$, but lines 3 and 4 are solved serially by T_{main}. This means that there is a back-and-forth flow between the threads in $\mathbf{T}_{spawn}$ and thread T_{main}. The threads in $\mathbf{T}_{spawn}$ will solve their subsets of $\mathbf{f}_{IFE}$, while thread T_{main} is in a spin lock waiting for them to complete. When the threads in $\mathbf{T}_{spawn}$ complete their calculations, they spin lock until

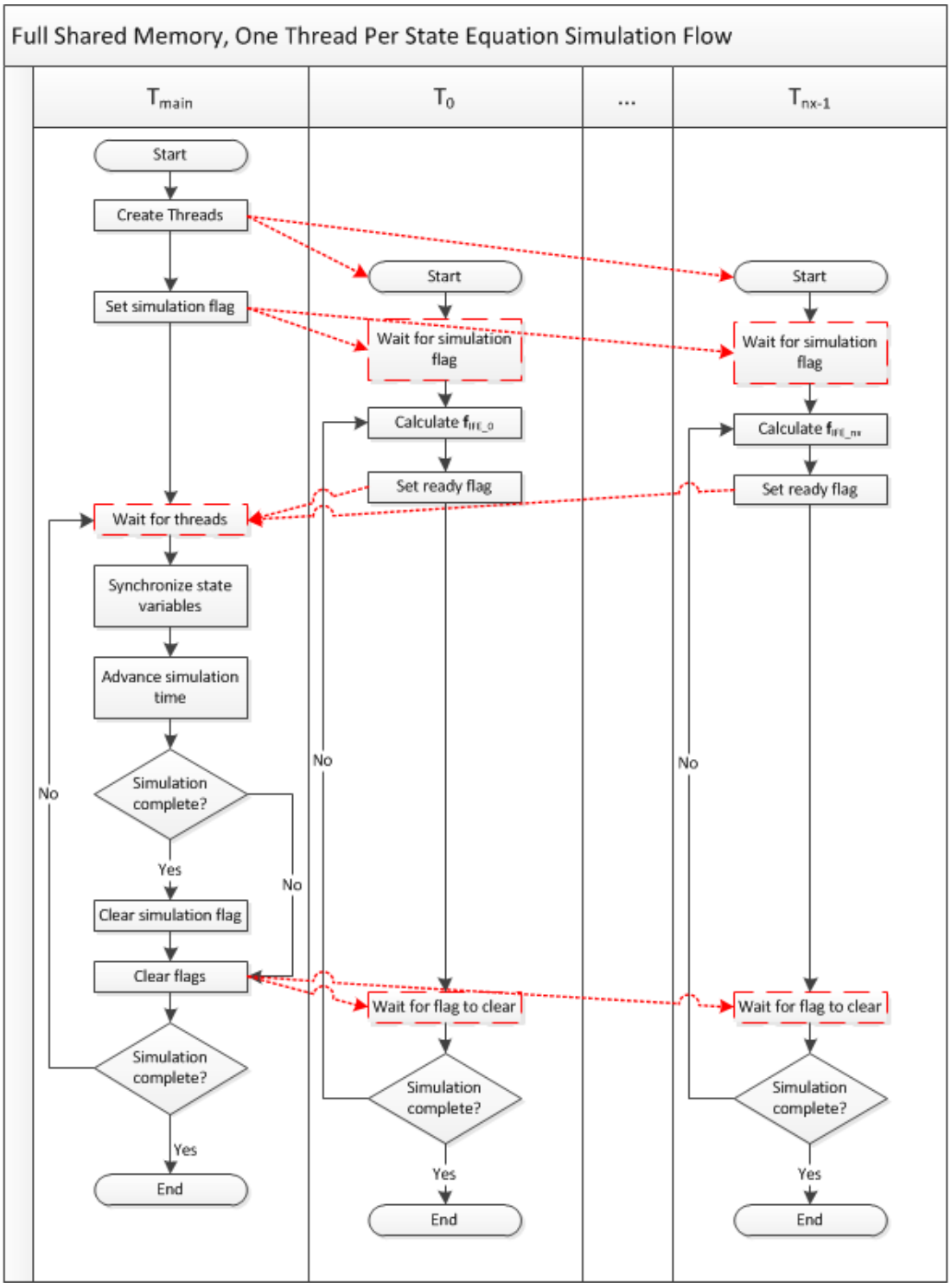


Figure 51: Back-and-forth program flow between T_{main} and T_{spawn}. The red dashed lines indicate communication between threads, and the boxes with red dashes are wait states.

Thread Count	OS Controlled, Rel Speedup	Affinity, Rel Speedup
2	N/A	2.9×10^{-4}
3	N/A	5.7×10^{-4}
4	N/A	8.5×10^{-4}
5	N/A	1.1×10^{-3}
6	N/A	1.4×10^{-3}
7	N/A	1.7×10^{-3}
8	1.69×10^{-3}	2.0×10^{-3}

Table 7: Test results showing the relative speedup for the complex RLC model with 288 state variables and slow time constants.

T_{main} is done merging, i.e. implementing Equation VI.7. This back-and-forth flow is shown in Figure 51. In the figure the red dashed lines indicate communication between threads, and the boxes with red dashes are wait states. The back and forth can be seen because after each communication, both from T_{main} to threads in $\mathbf{T}_{spawn}$ and from $\mathbf{T}_{spawn}$ to T_{main}, the thread sending the message immediately enters a wait state to wait for a message from the thread that received the message.

We also tested a second version of this algorithm that set the thread affinity for each of the created threads, such that the threads were evenly distributed between the processor cores. The expectation with setting the thread affinity is that it would offload the work of dynamically scheduling the threads from the OS and fix the schedule, thus reducing the simulation time. The results of simulating a model by creating one thread per state variable are shown in the next section in Table 7.

Definition 30 (Thread Affinity)**.** *The thread affinity identifies on which processor a thread is allowed to run.*

VI.5.1.1 Experimental Runs to Evaluate Speedup

Unfortunately, this algorithm produced very poor results in preliminary experiments so we quickly eliminated it from further testing. Table 7 presents simulation relative speed up using this simulation approach. The standard deviation values were calculated, but not

included in the above table because they are on the order of 10^{-5} to 10^{-7}. The model used in this experiment is the model shown in Figure 32 whose parameter values imply slow time constants for the simulation. The model was simulated for 5 seconds of simulation time over 10 repetitions, on an 8 core virtual machine running Linux on a cloud computing cluster hosted at our research institute. The relative speed up was significantly less than 1, indicating that this parallel approach was much slower than the serial simulation. Setting the thread affinity only slightly improved the simulation time compared to letting the OS handle the scheduling when using 5 spawned threads and 1 main thread, but the relative speedup was still orders of magnitude below 1.

There are several factors that caused this poor performance. The primary factor that slowed down the simulation is that the small amount of computation work assigned to each thread, just a single equation, was not large enough to overcome thread creation, scheduling, and communication overhead. In order for the parallel processing to be effective the overhead caused by managing threads has to be lower than the computation time per time step within each thread, as described in Section VI.2.4. In this approach the computation assigned to each thread is a single function from the set $\mathbf{f}_{IFE}$, but the overhead needed to calculate that function involves:

1. Communicating with thread T_{main} to ensure synchronization,
2. Pushing a C++ function onto the program stack and then popping it off,
3. Storing data to shared memory where every thread will be writing to the same cache line as 3 other threads,
4. Moving the thread onto a CPU core for processing, then moving it back off to make room for another thread.

These sources of overhead require a significant amount of time to perform, and because they are repeated for each thread at each time step of the simulation. They were repeated 144000 times during the simulation (5 seconds of simulation time with a time step of 0.01

and 288 state variables), which makes their contribution to the simulation run time significant. A related cause of poor performance is implied in number 4 above. By creating more threads then there are processor cores, the operating system needs to continually move threads onto and off of the CPU cores to ensure that all threads complete their tasks. This takes time, that again might be insignificant if it only needed to happen once, but because it needs to happen repeatedly within each time step this is a significant drain on simulation performance.

The outcome of these early experiments was to drop this approach of creating one thread for each state variable. It was simply too inefficient to be worth pursing further because we were not assigning enough work to our created threads to allow them to overcome overhead and we were creating more threads than could be efficiently handled by the OS. Taken together, these two facts point to the need for an agglomeration strategy [52]. Agglomeration will allow us to group the equations in $\mathbf{f}_{IFE}$ so that we create m total threads. This will prevent us from overwhelming the operating system with too many threads and will allow for easy assignment of threads to processing cores. Agglomeration will also allow us to assign more computational work to each processor core, which will allow the work in a thread to overcome the scheduling overhead. The next section details that agglomeration approach.

VI.5.2 Parallel Algorithm Type 2: Agglomerated Full Shared Memory

Based on the results in the previous section we needed to agglomerate our state variable calculations so that we could create fewer than n_x threads, where n_x is the number of state variables in the model. We targeted creating at most m total threads, where m is the number of CPUs available to the operating system. This avoids the problem of having to move threads onto and off of a CPU core, therefore avoiding a primary source of overhead in the previous parallel algorithm. Algorithm 2 partitions $\mathbf{f}_{IFE}$ into subsets with each subset

executing in parallel on a separate thread. Since each thread contains multiple state equations, each thread requires larger amounts of computational work, and this creates a better balance between the computational work and overhead required for scheduling and communication. We developed two different agglomeration strategies that we call (1) simple agglomeration and (2) smart agglomeration.

Algorithm Full Shared Memory Simple Agglomeration, uses a simple agglomeration scheme where the equations in $\mathbf{f}_{IFE}$ are partitioned according to the order they were specified in the pseudo-Modelica input file shown in Figure 49. This means that the first $n_x/(m-1)$ equations were assigned to T_0, the second set of $n_x/(m-1)$ equations were assigned to T_1, and so on until all of the equations in $\mathbf{f}_{IFE}$ were assigned to the threads in $\mathbf{T}_{spawn}$. There may be variations in the sizes of the subsets as described in Section VI.3.

Algorithm Full Shared Memory Smart Agglomeration, uses a smart agglomeration scheme that groups the component ODE equations, such that the equations that have a large number of dependencies (used a large number of state variables to calculate a particular value of $\mathbf{x}_{n+1}$) were grouped together in the same thread in $\mathbf{T}_{spawn}$. The idea behind smart agglomeration is to limit the amount of data communication between threads, which reduces the overhead, and, therefore, should reduce the simulation time.

In these experimental runs $|\mathbf{T}_{spawn}|$ is $m-1$ threads. When combined with the main thread T_{main}, the total number of threads used by a simulation was m. The division of work between the threads in $\mathbf{T}_{spawn}$ and T_{main} is also the same between this agglomerated approach and what was described in Section VI.5.1. The threads in $\mathbf{T}_{spawn}$ are responsible for calculating their subset of $\mathbf{f}_{IFE}$ (line 2 of Algorithm 3), and thread T_{main} is responsible for merging $\mathbf{x}_{n+1}$ into $\mathbf{x}_n$ (Equation VI.7 and line 3 of Algorithm 3) and advancing the simulation time (Equation VI.8 and line 4 of Algorithm 3). This results in a back and forth program flow between T_{main} and $\mathbf{T}_{spawn}$, where the threads of $\mathbf{T}_{spawn}$ perform their calculations while T_{main} waits, then the threads of $\mathbf{T}_{spawn}$ wait while T_{main} performs

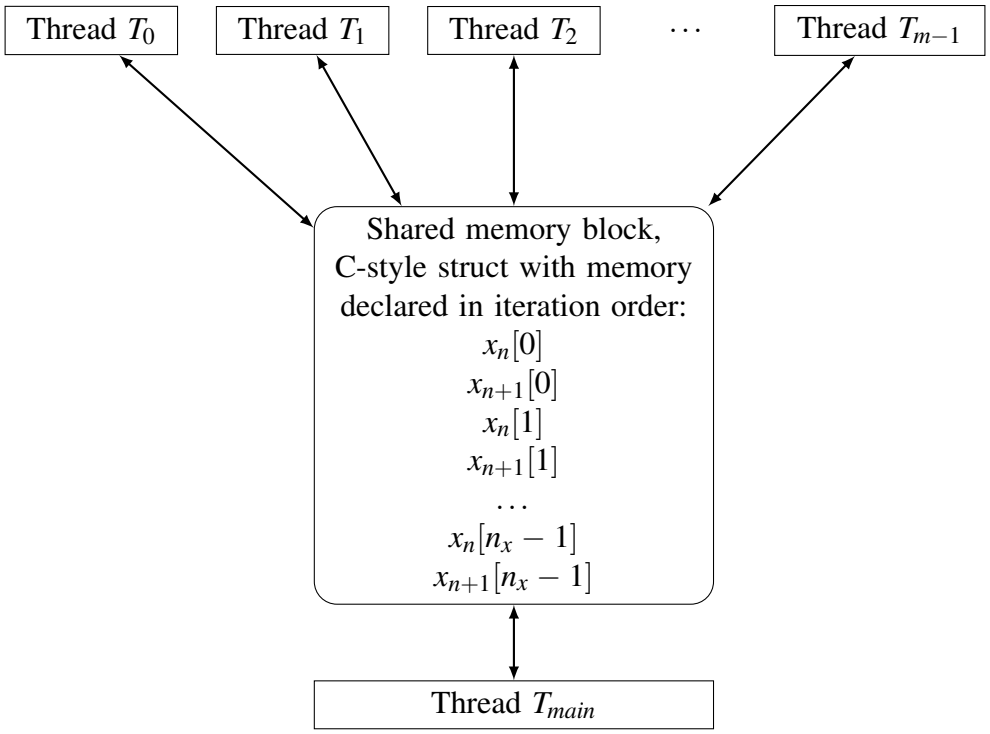


Figure 52: Figure describing shared memory implementation when using an agglomeration approach and 1 thread per CPU core.

the synchronization. The figure used to describe the program flow in Section VI.5.1, Figure 51, also describes the flow for this agglomerated shared memory approach, except that the final thread in the figure should be labeled T_{m-1}. These experiments used a shared memory block where every thread read from, and wrote to, the same shared memory block (Equation VI.17), as shown in Figure 52.

VI.5.2.1 Experimental Runs to Evaluate Speedup

The average relative speedup for these two algorithms for the small complex RLC model, Figure 32, is listed in Table 8, the average relative speedup for the large complex RLC model, Figure 33, is listed in Table 9, and the average relative speedup for the regular RLC model, Figure 36, is in Table 10. Figure 53 presents the relative speedups across all models. The standard deviation in the speedups was very small, in most cases it had a magnitude of 10^{-3}, so we did not include it in the tables.

	Slow Time Constants		Fast Time Constants	
Total Threads	Simple Agglom	Smart Agglom	Simple Agglom	Smart Agglom
2	0.10	0.11	0.14	0.13
3	0.18	0.19	0.25	0.24
4	0.24	0.25	0.32	0.32
5	0.24	0.24	0.32	0.30
6	0.27	0.25	0.35	0.32
7	0.28	0.26	0.37	0.33
8	0.29	0.27	0.37	0.34

Table 8: Relative speedup for Full Shared Memory approach using simple and smart agglomeration on the complex RLC model with 288 state variables.

	Slow Time Constants		Fast Time Constants	
Total Threads	Simple Agglom	Smart Agglom	Simple Agglom	Smart Agglom
2	0.14	0.13	0.19	0.17
3	0.26	0.25	0.35	0.32
4	0.37	0.35	0.50	0.48
5	0.37	0.35	0.47	0.45
6	0.42	0.41	0.55	0.52
7	0.46	0.44	0.59	0.56
8	0.49	0.44	0.64	0.57

Table 9: Relative speedup for Full Shared Memory approaches using simple and smart agglomeration on the complex RLC model with 804 state variables.

	Fast Time Constants	
Threads	Simple Agglom	Smart Agglom
2	0.24	0.22
3	0.40	0.38
4	0.56	0.54
5	0.55	0.54
6	0.61	0.74
7	0.65	0.65
8	0.68	0.92

Table 10: Relative speedup for Full Shared Memory approaches using simple and smart agglomeration on the regular RLC model with 1000 state variables.

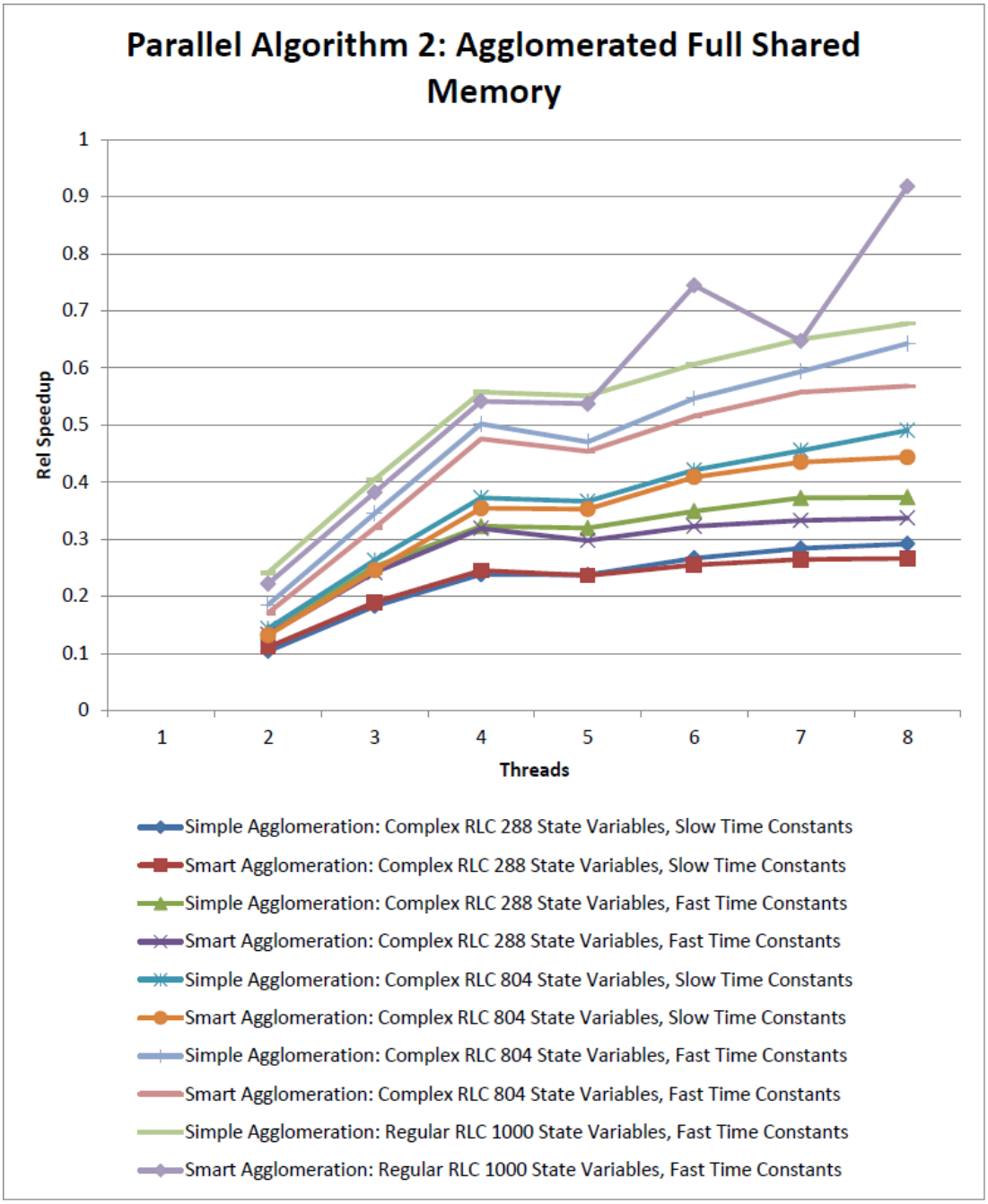


Figure 53: Figure showing the relative speedups across the models for the Full Shared Memory algorithms.

An initial analysis of the results presented in Tables 8, 9, and 10 shows little difference between the simple and smart agglomeration approaches. Comparing these tables to the data in Table 7, we see that there is a large improvement in the relative speedup in the agglomerated approach compared to the non-agglomerated approach. Unfortunately, the agglomerated approach speedup is still less than 1. The best speedup the agglomerated approach produced was 0.92 on the model with 1000 state variables using the smart agglomeration approach. For the complex models the best speedup of 0.64 was achieved on the model with 804 state variables with fast time constants and 8 threads. This implies the speedup may have improved on processors with a larger number of cores (e.g., GPU's). However, the more important lesson was to find better ways of partitioning the equations and allocating memory to improve overall speedup.

One reason for the lack of speedup in this experiment is possibly due to cache line sharing across the threads. Since every thread reads and writes to the same memory block, there are multiple threads that need to write to the same cache line. Tables 11, 12, and 13 present an analysis of shared cache lines when the models use the simple agglomeration approach. The data block is not guaranteed to be aligned to a cache line, so we analyzed each model twice, once when the memory block is aligned to a cache line and once when it is aligned 16 byes off of a cache line (the size of 2 `doubles` in our architecture). In general, when the data is aligned to a cache line boundary there is less cache line sharing than when the data is not aligned to a cache line boundary. However, there is still significant cache line sharing, which, as discussed in Section V.2.2.1, can lead to significant overhead during simulation. In the next algorithm, we will attempt to remove cache line sharing.

We did not analyze cache line sharing for the smart agglomeration approach, but that approach likely has greater problems with cache line sharing than the simple agglomeration approach. The reason for this is that in the simple agglomeration approach the functions in $\mathbf{f}_{IFE}$ were partitioned to the threads in $\mathbf{T}_{spawn}$ in the same order as the variables in the block of memory were declared, allowing for a natural, orderly, division of which pieces

Threads in T_{spawn}	Data Aligned to Cache Line	Data 16 Bytes Off a Cache Line
2	No sharing	(T_0,T_1), (T_1,T_2)
3	No sharing	(T_0,T_1), (T_1,T_2), (T_3,T_3)
4	No sharing	(T_0,T_1), (T_1,T_2), (T_2,T_3), (T_3,T_4)
5	(T_0,T_1), (T_2,T_3), (T_3,T_4)	(T_0,T_1), (T_1,T_2), (T_2,T_3), (T_4,T_5)
6	No sharing	(T_0,T_1), (T_1,T_2), (T_2,T_3), (T_3,T_4), (T_4,T_5), (T_5,T_6)
7	(T_0,T_1), (T_1,T_2), (T_3,T_4), (T_4,T_5), (T_5,T_6)	(T_0,T_1), (T_2,T_3), (T_3,T_4), (T_4,T_5), (T_6,T_7)

Table 11: Complex RLC model with 288 state variables and simple agglomeration: Cache line sharing between threads when the data block is allocated on a cache line boundary, and when it is allocated 16 bytes (the size of 2 `doubles`) off of a cache line boundary. The pairs of threads in parenthesis indicate both threads write to the same cache line.

Threads in T_{spawn}	Data Aligned to Cache Line	Data 16 Bytes Off a Cache Line
2	(T_0,T_1)	(T_0,T_1), (T_1,T_2)
3	No sharing	(T_0,T_1), (T_1,T_2), (T_2,T_3)
4	(T_0,T_1), (T_1,T_2), (T_2,T_3)	(T_0,T_1), (T_1,T_2), (T_3,T_4)
5	(T_0,T_1), (T_1,T_2), (T_2,T_3)	(T_0,T_1), (T_1,T_2), (T_3,T_4), (T_4,T_5)
6	(T_0,T_1), (T_2,T_3), (T_4,T_5)	(T_0,T_1), (T_1,T_2), (T_2,T_3), (T_3,T_4), (T_4,T_5), (T_5,T_6)
7	(T_0,T_1), (T_1,T_2), (T_2,T_3), (T_4,T_5), (T_5,T_6)	(T_1,T_2), (T_2,T_3), (T_3,T_4), (T_5,T_6), (T_6,T_7)

Table 12: Complex RLC model with 804 state variables and simple agglomeration: Cache line sharing between threads when the data block is allocated on a cache line boundary, and when it is allocated 16 bytes (the size of 2 `doubles`) off of a cache line boundary. The pairs of threads in parenthesis indicate both threads write to the same cache line.

Threads in T_{spawn}	Data Aligned to Cache Line	Data 16 Bytes Off a Cache Line
2	No sharing	(T_0, T_1), (T_1, T_2)
3	(T_0, T_1), (T_1, T_2)	(T_0, T_1), (T_2, T_3)
4	(T_0, T_1), (T_2, T_3)	(T_0, T_1), (T_1, T_2), (T_2, T_3), (T_3, T_4)
5	No sharing	(T_0, T_1), (T_1, T_2), (T_2, T_3), (T_3, T_4), (T_4, T_5)
6	(T_0, T_1), (T_1, T_2), (T_2, T_3), (T_4, T_5)	(T_1, T_2), (T_2, T_3), (T_3, T_4), (T_4, T_5), (T_5, T_6)
7	(T_0, T_1), (T_1, T_2), (T_2, T_3), (T_4, T_5), (T_5, T_6)	(T_1, T_2), (T_2, T_3), (T_3, T_4), (T_5, T_6), (T_6, T_7)

Table 13: Complex RLC model with 1000 state variables and simple agglomeration: Cache line sharing between threads when the data block is allocated on a cache line boundary, and when it is allocated 16 bytes (the size of 2 `doubles`) off of a cache line boundary. The pairs of threads in parenthesis indicate both threads write to the same cache line.

of the memory block were written to by each thread. The only possible cache line sharing that can happen is with each thread's neighbor threads. With the smart agglomeration approach, we changed the order in which the functions $\mathbf{f}_{IFE}$ were assigned to the threads in $\mathbf{T}_{spawn}$, but we did not change the order in which the variables those threads wrote to and read from, $\mathbf{x}_{n+1}$ and $\mathbf{x}_n$ respectively, were declared in the block of memory. This means that the orderly division of which threads wrote to which pieces of the memory block of the simple agglomeration approach is not applicable, and that each thread was much more likely to share cache lines with multiple other threads. With the smart agglomeration approach it is possible that each thread of T_{spawn} has to write to write to a cache line shared with 3 other threads (cache lines are 64 bytes, `doubles` are 8 bytes, meaning there are 8 `doubles` per cache line; each state variable has 2 `doubles` in the memory block declared sequentially meaning that there are 4 state variables per cache line). The smart agglomeration partitioning also means that the threads in $\mathbf{T}_{spawn}$ may have to write to more cache lines than the threads in the simple agglomeration approach. Writing to more cache lines can cause a problem because in order for the cache line to be written to, it first needs to read the cache line [87], and, as shown in Section V.2.2, reading a large number of

cache lines can take a significant amount of time where the processor is prevented from performing useful work. These two factors, the possibility of more cache line sharing and more cache line reads for each thread in $\mathbf{T}_{spawn}$, mean that the lack of benefit provided by the smart agglomeration approach is expected, and, from a certain perspective, it is even surprising that it generally kept up with the simple agglomeration approach.

These observations point to the need to eliminate cache line sharing between the threads. An algorithm that eliminates cache line sharing is presented in the next section, and Section VI.7 presents a deeper analysis of these results.

VI.5.3 Parallel Algorithms Type 3: Partial Partitioned Memory

The problems with cache line sharing in the previous approach, clearly indicates a need to eliminate the sharing in order to improve performance. We note that the division of labor between T_{main} and $\mathbf{T}_{spawn}$ leads to a division of the reads and writes to memory. The threads in $\mathbf{T}_{spawn}$ always read the values from $\mathbf{x}_n$, perform the required computations, and then write the values of $\mathbf{x}_{n+1}$ for the next simulation time step. Thread T_{main} always reads the values from $\mathbf{x}_{n+1}$, and writes to $\mathbf{x}_n$ as a part of the synchronization step of Algorithm 3. At no point during the simulation are T_{main} and $\mathbf{T}_{spawn}$ writing to the same variables or writing at the same time. This insight led us to design a memory structure in an effort to make these read and write processes more efficient in the simulation algorithm.

The new memory structure reduces cache line sharing by creating individual data structures aligned to a cache line boundary for each thread in $\mathbf{T}_{spawn}$ (Section V.2.2), to which the threads write their calculated values of $\mathbf{x}_{n+1}$. In this case the cache alignment is accomplished through the C function `aligned_alloc` that is supported by our C++ compiler (see Section VI.2.5). The `aligned_alloc` function was created as a part of the C11 standard [2], and takes 2 parameters: `alignment` and `size`. The function dynamically creates an array of size `size` that will be assigned a memory address that is an even multiple of `alignment`. If the `alignment` parameter is given a value of 64, the size of a

cache line in our CPU architecture, then the array will be aligned to a cache line. When every thread in $\mathbf{T}_{spawn}$ is assigned an array that is aligned to a unique cache line, each individual thread will avoid sharing cache lines with other threads, and this should improve overall run time performance.

This parallel simulation algorithm, Partial Partitioned Memory, is identical to the simple agglomeration approach presented above in Section VI.5.2, except that it partitions $\mathbf{x}_{n+1}$ into $m-1$ independent memory blocks that match the subsets of $\mathbf{f}_{IFE}$. This avoids a single large shared memory block across all threads, and the created memory blocks can be represented as:

$$M_0\left[\frac{n_x}{m-1}\right], M_1\left[\frac{n_x}{m-1}\right], \ldots, M_{m-2}\left[\frac{n_x}{m-1}\right], \tag{VI.18}$$

where each memory block M_x is aligned to a cache line using the `aligned_alloc` function, and each block is writable by only one of the threads in $\mathbf{T}_{spawn}$. There is also a shared memory block

$$M_{main}[n_x] \tag{VI.19}$$

that contains the values of $\mathbf{x}_n$ and is readable by all of the threads in $\mathbf{T}_{spawn}$, but is only writable by thread T_{main}. Since thread T_{main} has the same role as algorithms 1 and 2 (Sections VI.5.1 and VI.5.2), and it is responsible for synchronizing $\mathbf{x}_{n+1}$ and $\mathbf{x}_n$, it needs to write to M_{main} and read from memory blocks $M_0, \ldots, M_{m-2}$. The relationships between the threads and the memory blocks are described in Figure 54.

This algorithm's simulation flow is the same as in the two previous algorithms in terms of the back-and-forth flow between T_{main} and $\mathbf{T}_{spawn}$. The program flow is described in Figure 51 except that the final thread is T_{m-2} instead of T_{n_x-1}.

One drawback to partitioning $\mathbf{x}_{n+1}$ into independent blocks is that because thread T_{main} is responsible for merging $\mathbf{x}_{n+1}$ into $\mathbf{x}_n$, according to Equation VI.7, having the values of $\mathbf{x}_{n+1}$ spread across at least m cache lines will require T_{main} to perform more cache line reads

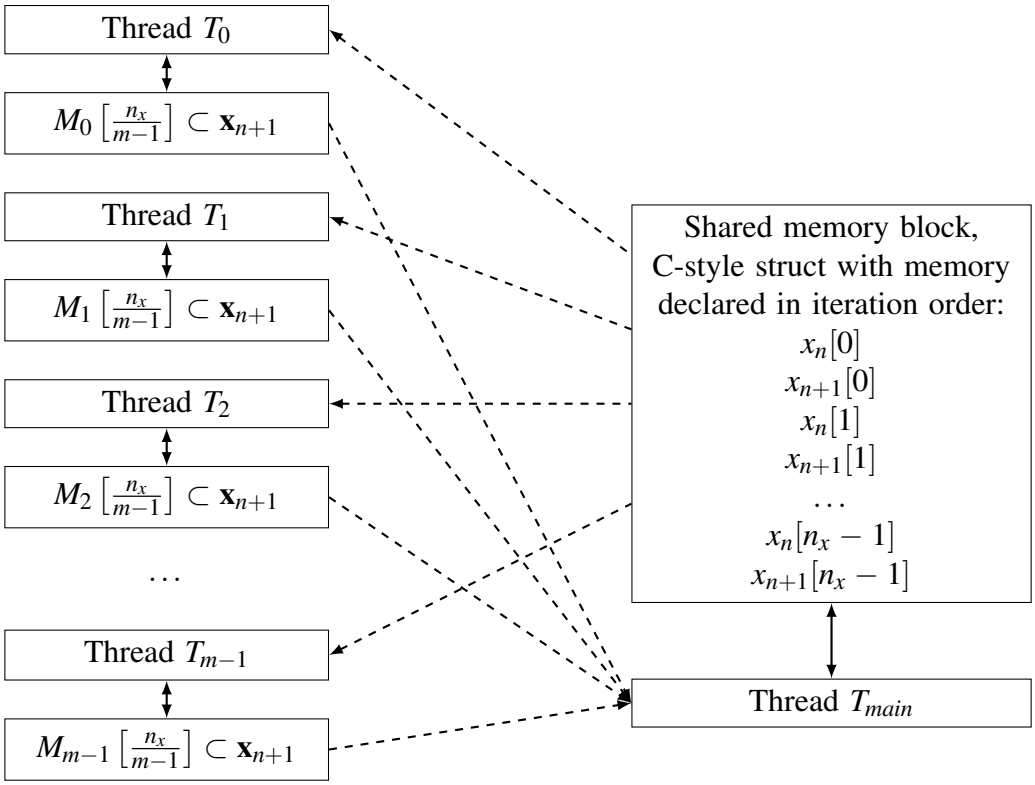


Figure 54: Figure describing partial partitioned memory implementation when using agglomeration and 1 thread per CPU core. Note: The $\mathbf{x}_{n+1}$ values are present in this data structure but are not used. $\mathbf{x}_{n+1}$ values were not removed from this struct in order to re-use data structures from previous tests.

to complete the synchronization than the simulation algorithms presented in Section VI.5.2. At most there are m extra cache line reads per time step to merge $\mathbf{x}_{n+1}$ into $\mathbf{x}_n$. These extra cache line reads cause the synchronization part of the simulation to take longer than in the previous approaches, but it is expected that avoiding the cache line sharing of the previous approach will provide greater time savings, and, therefore, a better speedup than the previous algorithms.

VI.5.3.1 Experimental Runs to Evaluate Speedup

The average relative speedups for this experiment are shown in Tables 14, 15, and 16. The standard deviation in the speedups was very small, in most cases it had a magnitude of 10^{-3}, so we did not include it in the tables. Here, again, we note that the relative speedups

Threads	Slow Time Constants	Fast Time Constants
2	0.10	0.14
3	0.16	0.21
4	0.20	0.27
5	0.19	0.24
6	0.19	0.24
7	0.18	0.25
8	0.18	0.24

Table 14: Relative speedups for the Partial Partitioned Memory approach on the complex RLC model with 288 state variables.

Threads	Slow Time Constants	Fast Time Constants
2	0.15	0.18
3	0.24	0.31
4	0.31	0.41
5	0.30	0.38
6	0.29	0.40
7	0.29	0.39
8	0.30	0.40

Table 15: Relative speedups for the Partial Partitioned Memory approach on the complex RLC model with 804 state variables.

Threads	Fast Time Constants
2	0.24
3	0.38
4	0.49
5	0.45
6	0.47
7	0.47
8	0.47

Table 16: Relative speedups for the Partial Partitioned Memory approach on the regular RLC model with 1000 state variables.

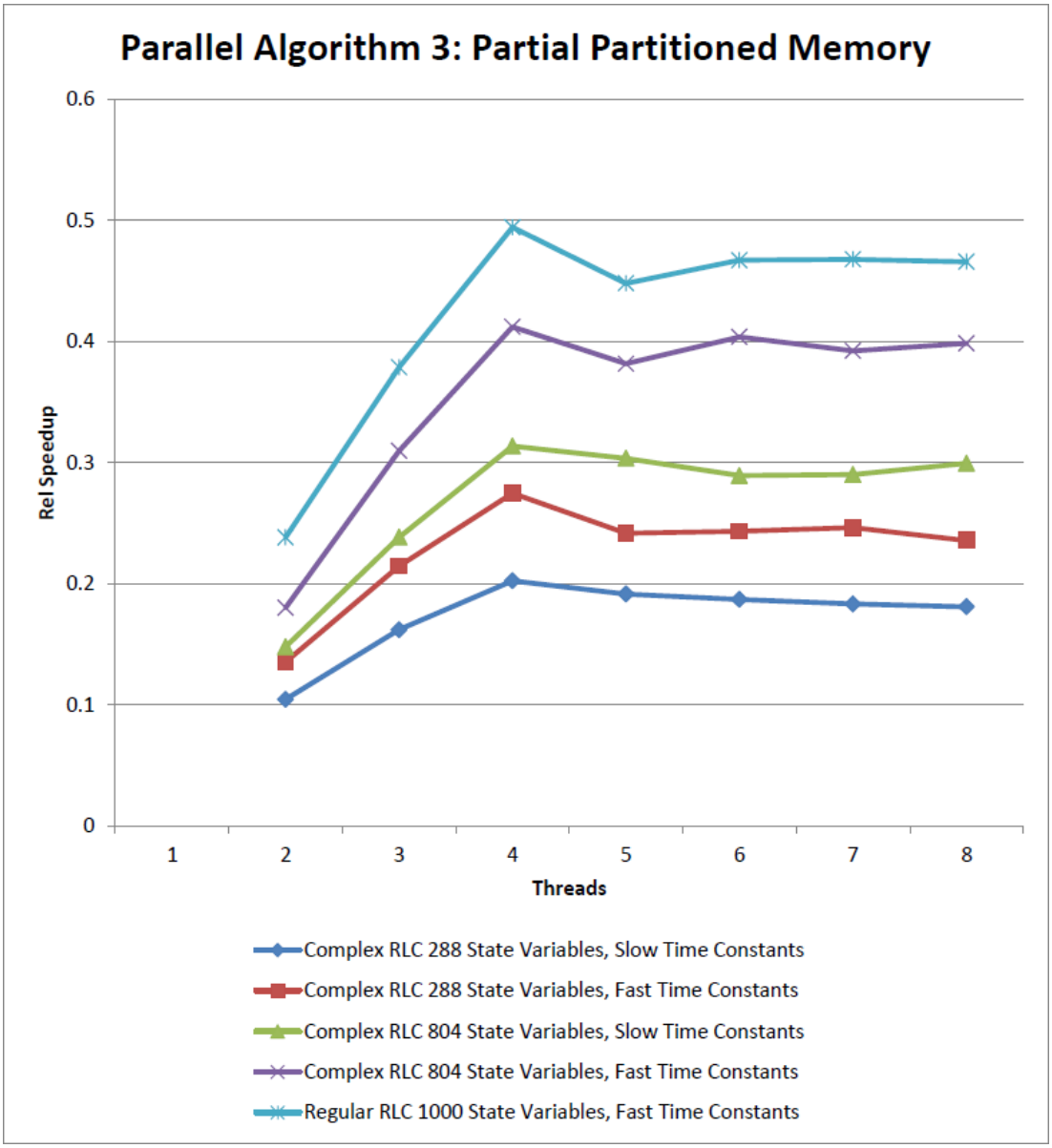


Figure 55: Figure showing the relative speedups across the models for the Partial Partitioned Memory algorithm.

are not competitive with the serial simulation. Comparing these results to the full shared memory results in Tables 8, 9, and 10 show that this partial shared memory approach is generally slower than the two full shared memory approaches. Figure 55 presents the relative speedups across all models.

The likely reason for the poorer performance than the algorithms detailed in Section VI.5.2 is the splitting up of $\mathbf{x}_{n+1}$ into subsets aligned to separate cache lines greatly increased the amount of memory reads and writes T_{main} needed to perform for the synchronization. Another possible reason for the poor performance is that the suspected cache line sharing from the previous algorithms was not causing a performance degradation, so eliminating the cache line sharing did not show a performance benefit. In the simple agglomeration approach of Section VI.5.2 the threads that had to write to the same cache line were writing to that cache line at different parts of the time step due to the way the equations in $\mathbf{f}_{IFE}$ were ordered. For example, in Table 12 when using total 8 threads, the threads T_0 and T_1 both needed to write to the same cache line. However, T_1 wrote to the shared cache line at the beginning of the time step because the data on the shared cache line was the first in its list to process, and T_0 wrote to the shared cache line at the end of the time step because the data on the shared cache line was the last in its list to process. It is likely that by the time T_0 wrote to the shared cache line, that T_1 had already completed with its writing, and no cache sharing conflict occurred.

Another cause of the poor performance is the back-and-forth nature of the simulation where T_{main} and the threads in $\mathbf{T}_{spawn}$ alternate between working and waiting for the other thread(s) to finish working. This leads to much wasted time in the simulation, and that wasted time has become a bottleneck for performance. By not assigning T_{main} a set of functions in $\mathbf{f}_{IFE}$ to solve, and by forcing all synchronization to take place in T_{main}, we were not using our processing resources to their full potential. The threads T_{main} and $\mathbf{T}_{spawn}$ alternates between working and not working, and time spent not working is time wasted.

A final reason for the poor performance is that the cache line reads thread T_{main} is

forced to perform for the synchronization is dominating the computation time to perform the synchronization. This situation is described in Figure 27 in Section V.2.2. The combined effects of these factors result in the algorithms taking an order of magnitude more time than the serial simulation. Getting better speedup with the parallel simulation will require modifying our simulation approach so we better use our processing resources and their memory. The next section present a simulation algorithm that allowed us to have a full parallel simulation and to provide a speedup beyond the serial simulation case.

VI.5.4 Parallel Algorithms Type 4: Full Partitioned Memory, Fixed Step

The fixed-step simulations that gave us the best performance used fully distributed memory. This means that each thread has its own cache aligned block of memory that it writes to which includes both $\mathbf{x}_{n+1}$ and $\mathbf{x}_n$. We tested two different versions of the algorithm, Full Partitioned Memory Minimum Sharing and Full Partitioned Memory Simple Agglomeration.

These two algorithms expand the roles of the threads in $\mathbf{T}_{spawn}$ and the thread T_{main}. The threads in $\mathbf{T}_{spawn}$ now perform their own synchronization step when given a signal by T_{main}. Thread T_{main}, instead of remaining idle when computations are being completed by threads in $\mathbf{T}_{spawn}$, solves some of the functions in $\mathbf{f}_{IFE}$. The program flow describing these two partitioned memory algorithms is described in Figure 56. Compared to the previous program flow diagram in Figure 51, the back and forth flow between threads is eliminated. Since each of the threads are responsible for their own synchronization according to Equation VI.7, the synchronization and advancing simulation time steps, on lines 3 and 4 of Algorithm 3, are performed in parallel. The threads in $\mathbf{T}_{spawn}$ pause before synchronizing to make sure all threads have completed calculating their values of $\mathbf{x}_{n+1}$. They proceed to the synchronization phase of the simulation on a signal from T_{main}. The threads again wait after synchronization for a signal from T_{main} before they begin to calculate their assigned equations from $\mathbf{f}_{IFE}$. This second synchronization point guarantees that there are no race

conditions between the threads in $\mathbf{T}$ during synchronization. Even though each thread calculates its own simulation time variable, only thread T_{main} can issue a stop signal to the threads in $\mathbf{T}_{spawn}$ when the simulation is complete.

VI.5.4.1 Full Partitioned Memory Minimum Sharing

The first full distributed memory approach, Full Partitioned Memory Minimum Sharing, created memory blocks:

$$M_0[2\cdot(n_x/m)], M_1[2\cdot(n_x/m)], \ldots, M_{m-2}[2\cdot(n_x/m)], M_{shared}[2\cdot(n_x/m)]. \tag{VI.20}$$

Each memory block is created on its own cache line to avoid cache line sharing. Also, it is assigned a subset of variables from $\mathbf{x}_{n+1}$ and $\mathbf{x}_n$, and then assigned to a thread in $\mathbf{T}_{spawn}$. The block of memory M_{shared} is assigned to T_{main}. This is shown in Figure 57. The memory blocks are structured so that each thread in $\mathbf{T}_{spawn}$ only needs the variables in its memory block and the variables in M_{shared} to solve its subset of $\mathbf{f}_{IFE}$. Therefore, each thread in $\mathbf{T}_{spawn}$ only writes its assigned $\mathbf{x}_{n+1}$ values to its own block of memory, and only reads from M_{shared}. Thread T_{main} has control of M_{shared}, and the variables stored in M_{shared} are the variables that are needed in more than one thread. To calculate its values of $\mathbf{x}_{n+1}$, T_{main} has read access to all of the other memory blocks. This memory structure is called Minimum Sharing, because each of the threads in $\mathbf{T}_{spawn}$ only share memory with thread T_{main}.

Aligning the data structures to a cache line boundary is accomplished through the `alignas` C++ keyword introduced in the C++11 standard [3]. The `alignas` keyword is used in the declaration of a variable or other object, and takes a parameter that describes the memory alignment requirement for the variable being declared. As an example the code `alignas(64) double var1;` will create a new `double` called `var1`, and the memory address of `var1` will be an even multiple of 64. Since our processor uses a 64

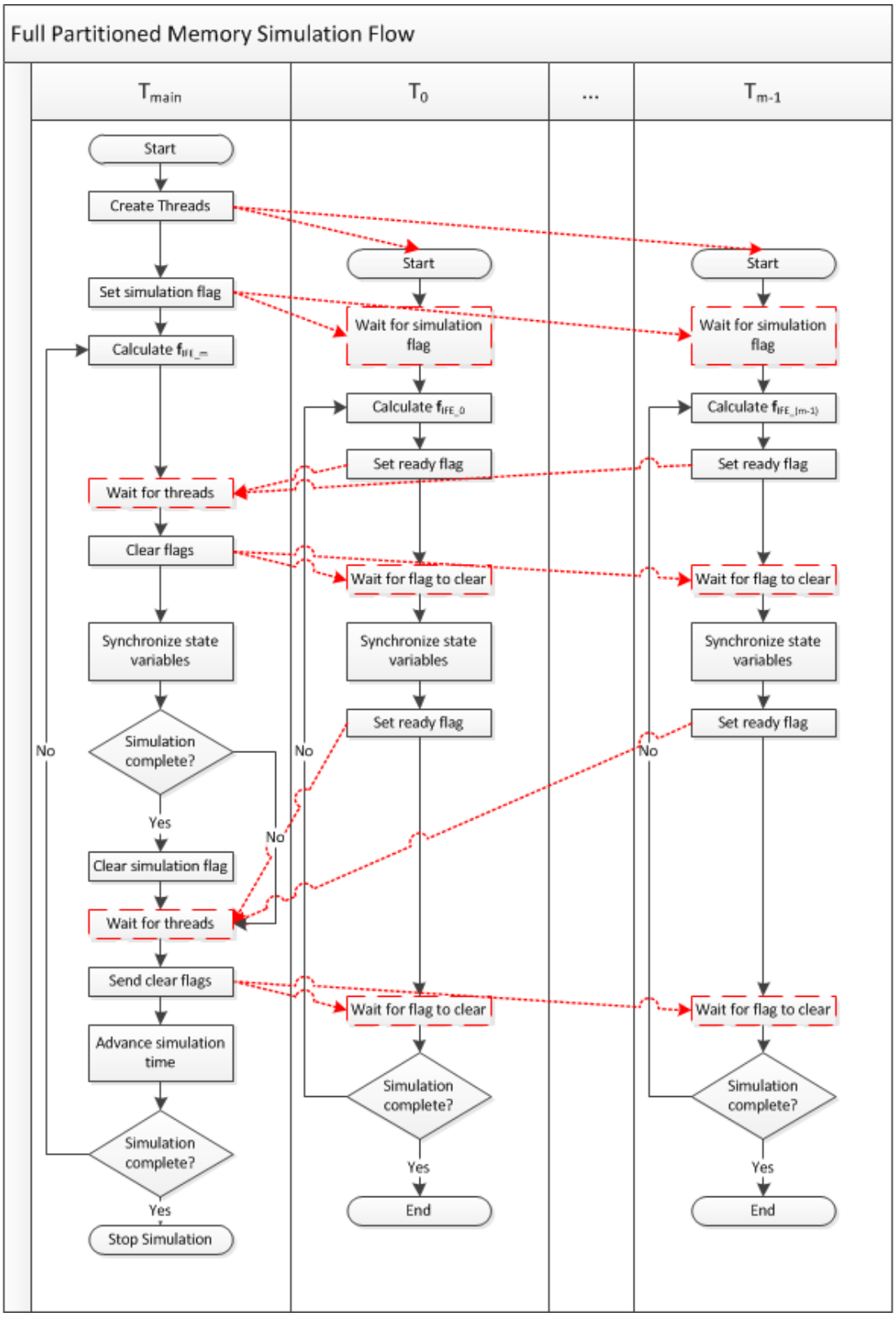


Figure 56: Full parallel simulation program flow. The red dashed lines indicate communication between threads, and the boxes with red dashes are wait states.

byte cache line, using `alignas(64)` when a variable is declared will align that variable to a cache line. The keyword applies when creating arrays and structs as well, and will provide a means to create an individual memory blocks for each thread in $\mathbf{T}$ that is aligned to a cache line. If each thread only writes to one memory block, and all memory blocks are aligned to separate cache lines, then there will be no cache line sharing between threads.

This memory model is very similar to the memory model in Partial Partitioned Memory. However, the advantage of this memory model over Partial Partitioned Memory is that the shared memory block M_{shared} is much smaller than the shared memory block, M_{main}, in the Partial Partitioned Memory approach. Another advantage is that the memory blocks assigned to the threads in $\mathbf{T}_{spawn}$ have values from both $\mathbf{x}_n$ and $\mathbf{x}_{n+1}$, instead of only having variables from $\mathbf{x}_{n+1}$. In the Partial Partitioned Memory the threads in $\mathbf{T}_{spawn}$ have to look to the shared memory block, and to their local block, to calculate a value for $\mathbf{x}_{n+1}$. With the full partitioned memory approach the threads in $\mathbf{T}_{spawn}$ need to primarily look to their local memory block to calculate $\mathbf{x}_{n+1}$, and only need to look to M_{shared} for a very few values needed to calculate their assigned values of $\mathbf{x}_{n+1}$. This will limit the number of cache line reads the threads in $\mathbf{T}_{spawn}$ will have to perform to calculate $\mathbf{x}_{n+1}$. Unfortunately, this approach has the opposite effect on T_{main}. Because T_{main} needs to access all of the memory blocks $M_0, \ldots, M_{m-2}$, it will require more cache line reads to perform its calculations than the threads in $\mathbf{T}_{spawn}$.

VI.5.4.2 Full Partitioned Memory Simple Agglomeration

The second full distributed memory approach, Full Partitioned Memory Simple Agglomeration, partitions the variables according to the simple partitioning scheme outlined in Section VI.5.2. It has the same program flow as Full Partitioned Memory Minimum Sharing, shown in Figure 56, where thread T_{main} solves a subset of $\mathbf{f}_{IFE}$ but still controls when to terminate the simulation, and the threads in $\mathbf{T}_{spawn}$ synchronize their assigned variables.

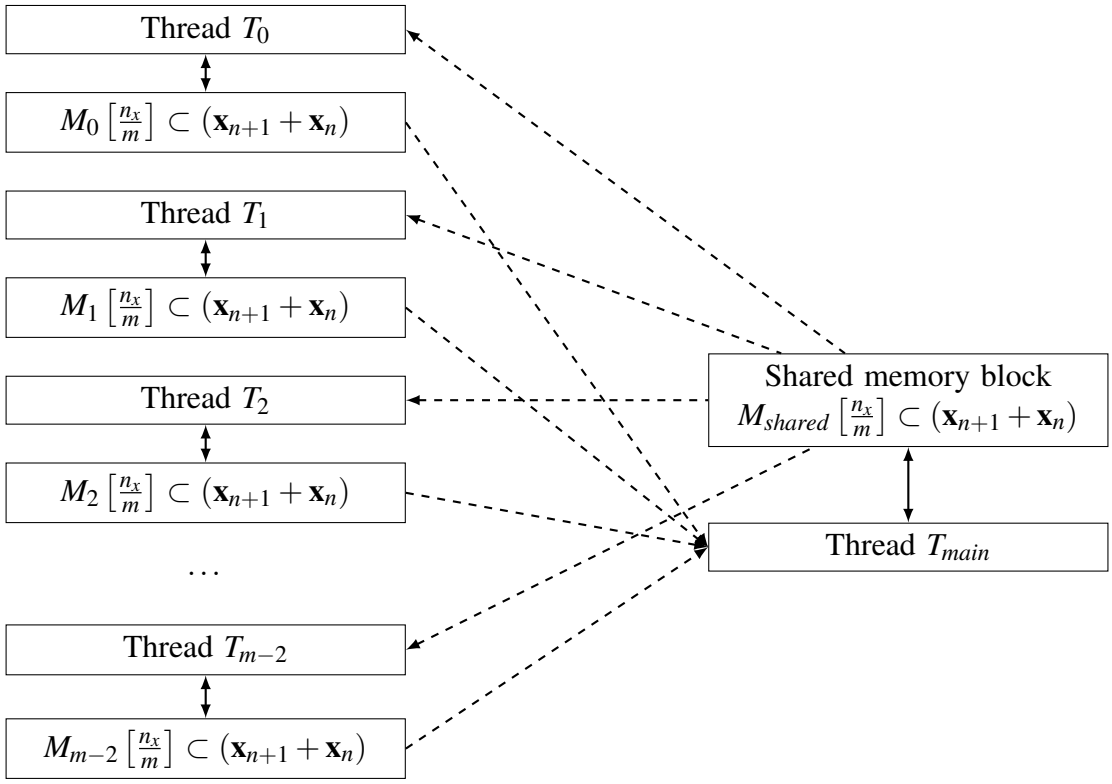


Figure 57: Figure describing Full Partitioned Memory Minimum Sharing memory structure using 1 thread per CPU core. The dashed lines indicate a read-only relationship.

The memory approach used in Simple Agglomeration is a little different from Minimum Sharing. Each thread in **T** has its own local memory block to which it reads and writes, and that represents the subset of values in $\mathbf{x}_{n+1}$ for which the thread is responsible. There also is a series of shared memory blocks $M_{shared0}, M_{shared1}, \ldots, M_{sharedm-1}$, where each block is a subset of $\mathbf{x}_n$. All of the threads in **T** can read from all of the shared blocks, but only one thread writes to each shared block. The portion of M_{shared} that each thread writes to aligned to a cache line boundary, so there are no cache line sharing problems. The memory model of Full Partitioned Memory Simple Agglomeration is:

$$M_0[(n_x/m)], M_1[(n_x/m)], \ldots, M_{m-1}[(n_x/m)], M_{shared0}[n_x/m], M_{shared1}[n_x/m], \ldots, M_{sharedm-1}[n_x/m]. \quad \text{(VI.21)}$$

The relationships between the threads and the different memory blocks is shown in Figure 58. With this approach, each of the threads in **T** needs read access to M_{shared}, but only writes to its own local block and its own portion of M_{shared}. This memory design relies on the fact that simply because a thread has access to a piece of memory does not mean that it will read from that memory. The threads in **T** are designed so that they only read the data they need to calculate their values of $\mathbf{x}_{n+1}$ from the shared memory. This design prevents unnecessary cache line reads, and therefore keeps memory accesses to a minimum.

VI.5.4.3 Experimental Runs to Evaluate Speedup

The relative speedups compared to the serial case are shown in Tables 17, 18, and 19. These tables include the standard deviations because several of the standard deviations are within 10% of the relative speed up value. Figure 59 presents the relative speedups across all models.

From Table 17 we can see that the model with 288 state variables did not produce a speedup compared to the serial case. Since the other models did produce a speedup,

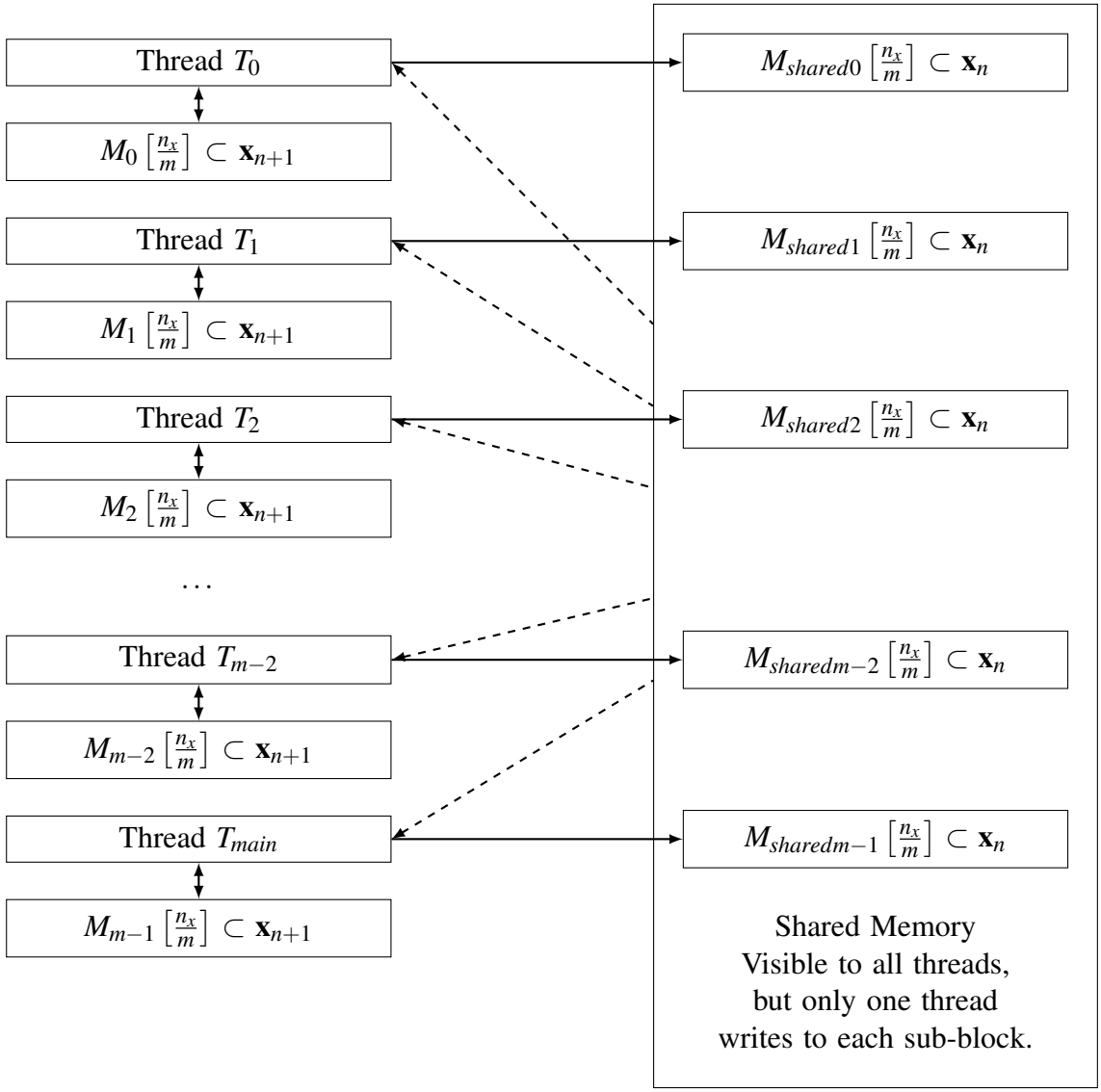


Figure 58: Figure describing Full Partitioned Memory Simple Agglomeration memory structure using 1 thread per CPU core. The dashed lines indicate a read-only relationship.

	Slow Time Constants			
	Minimum Sharing		Simple Agglomeration	
Total Threads	Rel Speedup	Std Dev	Rel Speedup	Std Dev
1	1.01	4.30×10^{-02}	0.46	5.51×10^{-03}
2	0.50	1.37×10^{-02}	0.47	8.02×10^{-03}
3	0.52	3.17×10^{-02}	0.50	1.44×10^{-02}
4	0.52	9.81×10^{-03}	0.56	1.82×10^{-02}
5	0.45	1.91×10^{-02}	0.49	2.35×10^{-02}
6	0.45	2.35×10^{-02}	0.51	2.74×10^{-02}
7	0.43	1.26×10^{-02}	0.50	4.02×10^{-02}
8	0.39	8.28×10^{-03}	0.46	1.27×10^{-02}
	Fast Time Constants			
	Minimum Sharing		Simple Agglomeration	
Total Threads	Rel Speedup	Std Dev	Rel Speedup	Std Dev
1	1.01	1.15×10^{-02}	0.43	1.17×10^{-02}
2	0.62	8.85×10^{-03}	0.50	7.07×10^{-03}
3	0.66	1.50×10^{-02}	0.55	1.70×10^{-02}
4	0.65	1.15×10^{-02}	0.61	2.23×10^{-02}
5	0.58	1.83×10^{-02}	0.53	1.65×10^{-02}
6	0.59	2.18×10^{-02}	0.56	1.64×10^{-02}
7	0.55	1.35×10^{-02}	0.56	1.71×10^{-02}
8	0.53	2.18×10^{-02}	0.51	1.23×10^{-02}

Table 17: Relative speedup and standard deviations for Full Partitioned Memory approaches versions 1 and 2 on the complex RLC model with 288 state variables.

	Slow Time Constants			
	Minimum Sharing		Simple Agglomeration	
Total Threads	Rel Speedup	Std Dev	Rel Speedup	Std Dev
1	1.03	4.92×10^{-03}	0.52	2.53×10^{-03}
2	0.80	1.13×10^{-02}	0.65	6.41×10^{-03}
3	1.02	9.76×10^{-03}	0.81	8.95×10^{-03}
4	1.17	1.26×10^{-02}	1.02	1.67×10^{-02}
5	1.02	4.53×10^{-02}	0.86	1.36×10^{-02}
6	1.12	2.64×10^{-02}	0.98	2.93×10^{-02}
7	1.11	4.41×10^{-02}	1.04	2.52×10^{-02}
8	1.10	2.36×10^{-02}	1.05	1.85×10^{-02}
	Fast Time Constants			
	Minimum Sharing		Simple Agglomeration	
Total Threads	Rel Speedup	Std Dev	Rel Speedup	Std Dev
1	1.00	5.87×10^{-03}	0.59	3.74×10^{-03}
2	0.92	9.98×10^{-03}	0.78	9.88×10^{-03}
3	1.21	1.06×10^{-02}	0.97	1.36×10^{-02}
4	1.44	1.80×10^{-02}	1.26	3.06×10^{-02}
5	1.24	6.27×10^{-02}	1.05	3.56×10^{-02}
6	1.34	2.85×10^{-02}	1.19	2.39×10^{-02}
7	1.37	3.03×10^{-02}	1.27	2.38×10^{-02}
8	1.39	2.41×10^{-02}	1.29	3.50×10^{-02}

Table 18: Relative speedup and standard deviations for Full Partitioned Memory approaches versions 1 and 2 on the complex RLC model with 804 state variables.

	Fast Time Constants			
	Minimum Sharing		Simple Agglomeration	
Total Threads	Rel Speedup	Std Dev	Rel Speedup	Std Dev
1	1.00	4.04×10^{-03}	0.67	2.52×10^{-03}
2	1.56	1.03×10^{-02}	0.89	2.82×10^{-03}
3	2.00	3.61×10^{-02}	0.99	5.21×10^{-03}
4	2.53	4.38×10^{-02}	1.16	4.40×10^{-03}
5	2.12	0.22	1.06	2.80×10^{-02}
6	2.29	0.25	1.09	2.53×10^{-02}
7	2.10	0.56	1.11	4.33×10^{-02}
8	1.49	0.51	1.12	8.44×10^{-03}

Table 19: Relative speedup and standard deviations for Full Partitioned Memory approaches versions 1 and 2 on the regular RLC model with 1000 state variables.

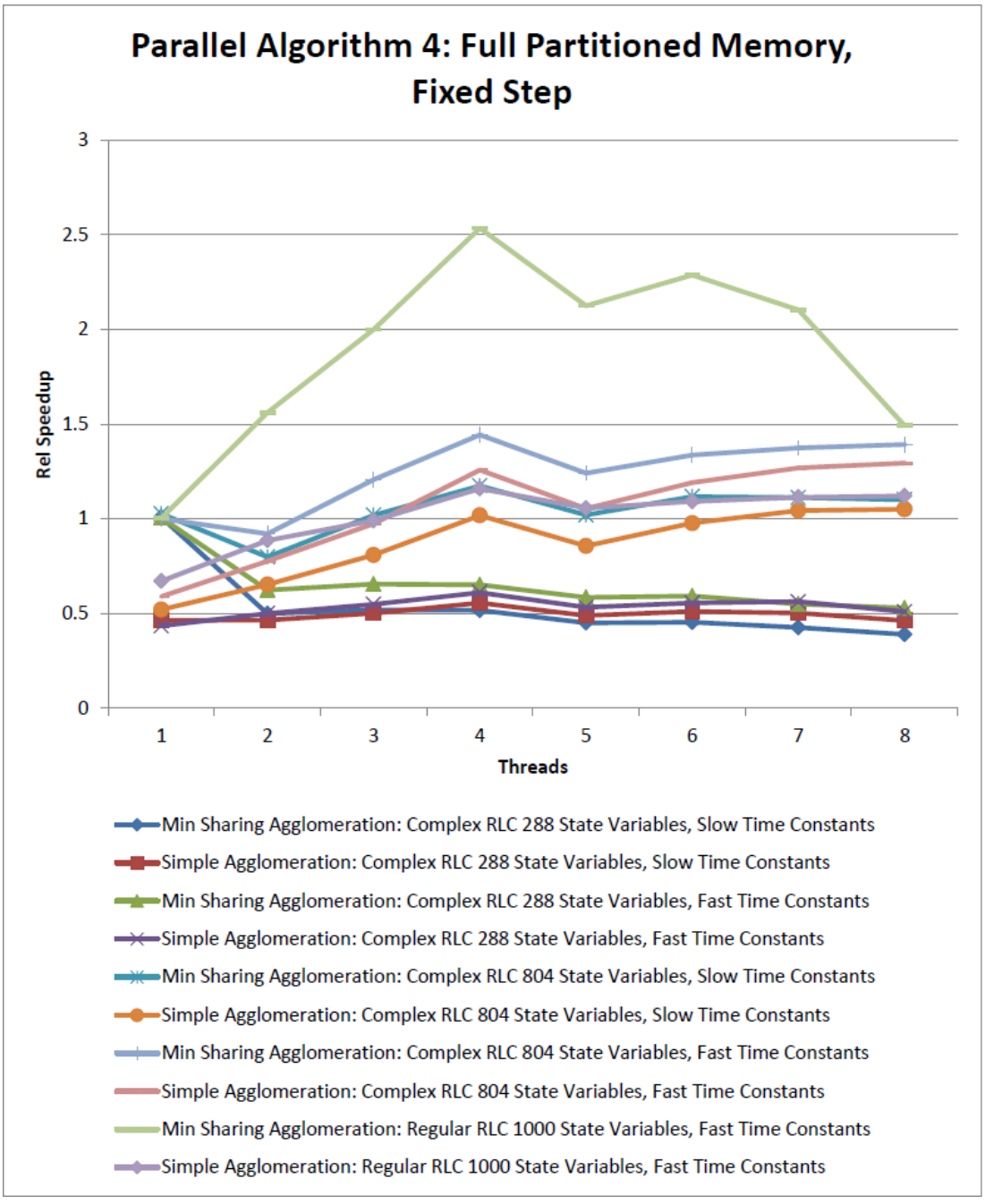


Figure 59: Figure showing the relative speedups across the models for the Full Partitioned Memory algorithms.

the lack of speedup on the smaller model is likely due to the model not having enough computational work to overcome the overhead associated with parallelization.

Table 18 shows the relative speedup when simulating the complex RLC model with 804 state variables (Figure 33). For most thread values Minimum Sharing matches the serial simulation, and in a few cases, such as when the model has fast time constants and is simulated using 4 threads, is able to surpass the serial simulation performance and provide a speedup of up to 1.44. Simple Agglomeration provided a small speedup when using the model with fast time constants, but was only able to match the serial simulation when simulating the model with slow time constants.

We note that Minimum Sharing and Simple Agglomeration have very different wall clock times when they are run only using one thread. This is an unexpected result because the memory differences between the two partitioning approaches should not come into play when only using one thread. The likely reason for the difference in single threaded simulation time is differences in implementation. Version 1 uses C-style structs for sharing data, and version 2 uses C-style arrays. Indexing into an array requires pointer arithmetic, which takes extra time, that is not present when using structs.

We also note that both of the Full Partitioned Approaches perform better on the complex RLC model with 804 state variables using fast time constants than slow time constants. A possible explanation is due to the equations in $\mathbf{f}_{IFE}$ for the fast parameters are more complicated than the equations for the slow parameters. This extra complexity, essentially just the presence of parameter values scaling the state variable values, means that the processor has more computational work to solve each equation and therefore the ratio of cache line reads to computational work goes down, and the cache line reads have less of an opportunity to dominate the run time. This analysis seems tenuous, but since these methods use fixed step integration, the change in model behavior has no effect on the integration (because the step size does not change as a result of system dynamics), and the only real difference

between the fast and slow parameter values is the presence of the parameter value terms in the integration equations.

In Figure 59, and in the above tables, we also see that the speedup drops between threads 4 and 5. This is due to the hardware multithreading built into the processor. At 5 threads one of the CPU cores needs to run two threads instead of just one, and, because the cache is allocated to a CPU core, not to a thread, the threads must fight for cache resources. It is also possible that the OS moves the extra thread from core to core to try and balance the workload assigned to each core. But this ends up hurting performance because each time the thread is moved it must populate the cache on the new core, and is, therefore, not able to realize the benefits of using a cache.

We also see in Figure 59 that in some instances there is a performance drop between using one thread and two threads. We are unsure what causes this.

We performed a mean-squared error analysis on our simulation results for both Minimum Sharing and Simple Agglomeration, and those results are shown in Tables 20 through 24. The mean square error calculations compared a single threaded fixed step simulation, using the italicised time step in the table, to a parallel simulation using 8 threads and the time steps listed in the table. Using a step size one order of magnitude smaller than the serial simulation provided error calculations on the order of 10^{-5} at the largest, to essentially 0 (on the order of 10^{-287}) at the smallest. This shows that our baseline simulation used an appropriate time step to generate correct results. If the baseline time step was too large, then the mean square error would be much larger when compared to a simulation using a smaller time step. The small error also shows that no accuracy is lost in during the parallel simulation.

Comparing our baseline simulation to a parallel simulation with a larger step size shows more interesting results. For the complex RLC model with 288 state variables and slow time constants, shown in Table 20, a step size of one order of magnitude larger than the baseline simulation provided small error values, with the largest error values on the order

of 10^{-3}. The complex RLC model with 804 state variables and slow time constants had very large errors when a time step one order of magnitude greater than the baseline serial simulation was used, shown in Table 22. The largest error in this case is on the order of 10^{37}. Using a time step 1.5 orders of magnitude larger than the baseline simulation resulted in mean square errors large enough that they are labeled infinite (abbreviated as "inf" in the tables).

The models with fast time constants required a much smaller change in time step compared to the models with slow time constants before the simulations became unstable, Tables 21, 23, and 24. In most cases, a change of less than an order of magnitude in the time step resulted in large errors. These models have faster dynamics, so it is expected that the requirements on the simulation time step are more strict than for the models with slow time constants.

These error analysis results show that when using a parallel simulation a small change in the simulation time step can move a simulation from small errors to large errors. It also shows that, because the parallel simulations had very small error when using the same time step size as the base simulation, that the process of parallelizing a simulation does not negatively affect the accuracy of the simulation.

VI.5.5 Parallel Algorithms Type 5: Full Partitioned Memory, Variable Step Simulation

All of our previous simulations used a fixed step simulation; however, a variable step simulation is likely to provide better simulation performance. Our final parallel algorithm is to test a parallel version of a variable step solver, using a full partitioned memory approach, to determine if further speedups can be found.

We use a Runge-Kutta-Fehlberg4,5 solver from the GNU Scientific Library [25] as our variable step solver. It is implemented using a traditional simulation method, as shown in Figure 16 and described in Section IV.1. To use the solver the user provides functions to

	Minimum Sharing Partitioning Mean-Square Error at Step Size			
Variable	0.001	*0.01*	0.1	0.5
1	1.02×10^{-5}	6.20×10^{-16}	1.06×10^{-3}	inf
2	1.33×10^{-8}	4.30×10^{-17}	2.69×10^{-6}	inf
3	1.10×10^{-8}	3.88×10^{-18}	1.44×10^{-6}	inf
4	1.71×10^{-9}	1.29×10^{-18}	2.18×10^{-7}	inf
285	1.76×10^{-19}	3.21×10^{-19}	6.86×10^{-16}	inf
286	7.32×10^{-16}	3.24×10^{-16}	6.70×10^{-12}	inf
287	4.73×10^{-20}	8.41×10^{-20}	1.73×10^{-16}	inf
288	7.30×10^{-16}	3.23×10^{-16}	6.68×10^{-12}	inf

	Simple Agglomeration Partitioning Mean-Square Error at Step Size			
Variable	0.001	*0.01*	0.1	0.5
1	1.02×10^{-5}	0.0	1.06×10^{-3}	inf
2	1.33×10^{-8}	0.0	2.69×10^{-6}	inf
3	1.10×10^{-8}	0.0	1.44×10^{-6}	inf
4	1.71×10^{-9}	0.0	2.18×10^{-7}	inf
285	1.76×10^{-19}	0.0	6.70×10^{-16}	inf
286	7.32×10^{-16}	0.0	8.01×10^{-12}	inf
287	4.73×10^{-20}	0.0	1.68×10^{-16}	inf
288	7.30×10^{-16}	0.0	7.99×10^{-12}	inf

Table 20: Mean square error calculations for 8 state variables of the model with 288 state variables and slow time constants using 8 threads. The italicised colum header indicates the time step that was used for the relative speedup experiments and as a baseline for calculating the MSE.

	Minimum Sharing Partitioning Mean-Square Error at Step Size			
Variable	0.00001	*0.0001*	0.0005	0.0008
1	2.12×10^{-11}	2.49×10^{-15}	4.22×10^{-10}	inf
2	1.17×10^{-7}	7.23×10^{-13}	2.31×10^{-6}	inf
3	1.79×10^{-11}	2.85×10^{-15}	3.57×10^{-10}	inf
4	8.25×10^{-8}	2.69×10^{-15}	1.63×10^{-6}	inf
285	6.43×10^{-16}	1.03×10^{-16}	5.39×10^{-14}	inf
286	4.27×10^{-11}	1.02×10^{-10}	1.20×10^{-8}	inf
287	1.66×10^{-16}	2.98×10^{-17}	2.83×10^{-14}	inf
288	4.39×10^{-11}	1.03×10^{-10}	1.20×10^{-8}	inf

	Simple Agglomeration Partitioning Mean-Square Error at Step Size			
Variable	0.00001	*0.0001*	0.0005	0.0008
1	2.12×10^{-11}	0.0	4.22×10^{-10}	inf
2	1.17×10^{-7}	0.0	2.31×10^{-6}	inf
3	1.79×10^{-11}	0.0	3.56×10^{-10}	inf
4	8.25×10^{-8}	0.0	1.63×10^{-6}	inf
285	6.33×10^{-16}	0.0	1.25×10^{-14}	inf
286	3.75×10^{-11}	0.0	7.44×10^{-10}	inf
287	1.60×10^{-16}	0.0	3.17×10^{-15}	inf
288	3.86×10^{-11}	0.0	7.67×10^{-10}	inf

Table 21: Mean square error calculations for 8 state variables of the model with 288 state variables and fast time constants using 8 threads. The italicised colum header indicates the time step that was used for the relative speedup experiments and as a baseline for calculating the MSE.

	Minimum Sharing Partitioning Mean-Square Error at Step Size			
Variable	0.001	*0.01*	0.1	0.5
1	1.02×10^{-5}	6.46×10^{-16}	1.13×10^{34}	inf
2	1.33×10^{-8}	5.20×10^{-17}	1.44×10^{35}	inf
3	1.10×10^{-8}	8.51×10^{-18}	1.39×10^{36}	inf
4	1.71×10^{-9}	4.66×10^{-18}	1.47×10^{37}	inf
801	5.68×10^{-21}	7.31×10^{-23}	5.19×10^{11}	inf
802	5.93×10^{-17}	1.48×10^{-18}	6.62×10^{15}	inf
803	1.51×10^{-21}	1.83×10^{-23}	1.30×10^{11}	inf
804	5.90×10^{-17}	1.47×10^{-18}	6.61×10^{15}	inf

	Simple Agglomeration Partitioning Mean-Square Error at Step Size			
Variable	0.001	*0.01*	0.1	0.5
1	1.02×10^{-5}	0.0	1.81×10^{19}	inf
2	1.33×10^{-8}	0.0	2.10×10^{20}	inf
3	1.10×10^{-8}	0.0	2.24×10^{21}	inf
4	1.71×10^{-9}	0.0	2.14×10^{22}	inf
801	5.75×10^{-21}	0.0	1.32×10^{-13}	inf
802	6.10×10^{-17}	0.0	9.96×10^{-13}	inf
803	1.53×10^{-21}	0.0	3.04×10^{-14}	inf
804	6.07×10^{-17}	0.0	9.02×10^{-13}	inf

Table 22: Mean square error calculations for 8 state variables of the model with 804 state variables and slow time constants using 8 threads. The italicised colum header indicates the time step that was used for the relative speedup experiments and as a baseline for calculating the MSE.

	Minimum Sharing Partitioning Mean-Square Error at Step Size			
Variable	0.00001	*0.0001*	0.0002	0.0005
1	2.17×10^{-11}	0.0	2.90×10^{-11}	inf
2	1.13×10^{-7}	0.0	1.40×10^{-7}	inf
3	1.96×10^{-11}	0.0	2.64×10^{-11}	inf
4	7.45×10^{-8}	0.0	9.42×10^{-8}	inf
801	1.19×10^{-17}	0.0	1.85×10^{-15}	inf
802	1.31×10^{-12}	0.0	3.79×10^{-8}	inf
803	3.01×10^{-18}	0.0	2.02×10^{-14}	inf
804	1.33×10^{-12}	0.0	3.79×10^{-8}	inf

	Simple Agglomeration Partitioning Mean-Square Error at Step Size			
Variable	0.00001	*0.0001*	0.0002	0.0005
1	2.17×10^{-11}	0.0	2.67×10^{-11}	inf
2	1.13×10^{-7}	0.0	1.39×10^{-7}	inf
3	1.96×10^{-11}	0.0	2.41×10^{-11}	inf
4	7.45×10^{-8}	0.0	9.20×10^{-8}	inf
801	1.19×10^{-17}	0.0	1.47×10^{-17}	inf
802	6.93×10^{-13}	0.0	8.56×10^{-13}	inf
803	3.01×10^{-18}	0.0	3.72×10^{-18}	inf
804	7.18×10^{-13}	0.0	8.88×10^{-13}	inf

Table 23: Mean square error calculations for 8 state variables of the model with 804 state variables and fast time constants using 8 threads. The italicised colum header indicates the time step that was used for the relative speedup experiments and as a baseline for calculating the MSE.

Variable	Minimum Sharing Partitioning Mean-Square Error at Step Size			
	0.000001	*0.00001*	0.00005	0.0001
1	7.16×10^{-11}	0.0	3.08×10^{-11}	inf
2	6.78×10^{-9}	0.0	3.37×10^{-9}	inf
3	1.22×10^{-10}	0.0	6.05×10^{-11}	inf
4	6.88×10^{-9}	0.0	2.73×10^{-9}	inf
801	3.05×10^{-140}	3.12×10^{-138}	1.08×10^{-181}	inf
802	5.87×10^{-141}	6.02×10^{-139}	3.94×10^{-181}	inf
803	3.91×10^{-142}	4.00×10^{-140}	1.86×10^{-182}	inf
804	9.12×10^{-143}	9.34×10^{-141}	1.71×10^{-181}	inf

Variable	Simple Agglomeration Partitioning Mean-Square Error at Step Size			
	0.000001	*0.00001*	0.00005	0.0001
1	7.16×10^{-11}	0.0	3.35×10^{305}	inf
2	6.78×10^{-9}	0.0	1.12×10^{305}	inf
3	1.22×10^{-10}	0.0	6.28×10^{303}	inf
4	6.88×10^{-9}	0.0	4.35×10^{304}	inf
997	2.46×10^{-9}	0.0	4.41×10^{-212}	inf
998	7.91×10^{-284}	0.0	2.24×10^{-217}	inf
999	1.57×10^{-285}	0.0	7.42×10^{-217}	inf
1000	5.21×10^{-287}	0.0	1.48×10^{-222}	inf

Table 24: Mean square error calculations for 8 state variables of the model with 1000 state variables and fast time constants using 8 threads. The italicised colum header indicates the time step that was used for the relative speedup experiments and as a baseline for calculating the MSE.

calculate the system state variable derivatives, $\dot{\mathbf{x}}_n$, and the system Jacobian matrix. The user also provides the solver with an output interval detailing how frequently the user wants to receive updates on the state variable derivatives. Controlling the simulation time step is left up to the solver (see Section VI.2.2).

To create a partitioned variable step solver we divide the model into ℓ independent pieces and assign a solver to each independent piece of the model. Each solver sets its own step size, and synchronization of the state variables occurs at at pre-defined output points. To partition a model into ℓ independent pieces we modify the model partitioning in Equation VI.13 to:

$$\begin{aligned}
\dot{\mathbf{x}}_{n_0} &= \mathbf{f}_0\left(t_n, \mathbf{x}_{n_0}, \mathbf{w}_n, \mathbf{x}_{n_1_prev}, \ldots, \mathbf{x}_{n_(\ell-1)_prev}\right) \\
\dot{\mathbf{x}}_{n_1} &= \mathbf{f}_1\left(t_n, \mathbf{x}_{n_1}, \mathbf{w}_n, \mathbf{x}_{n_0_prev}, \mathbf{x}_{n_2_prev}, \ldots, \mathbf{x}_{n_(\ell-1)_prev}\right) \\
&\ldots \\
\dot{\mathbf{x}}_{n_\ell-1} &= \mathbf{f}_{\ell-1}\left(t_n, \mathbf{x}_{n_(\ell-1)}, \mathbf{w}_n, \mathbf{x}_{n_0_prev}, \ldots, \mathbf{x}_{n_(\ell-2)_prev}\right),
\end{aligned} \tag{VI.22}$$

where the $\mathbf{x}_{n\_x\_prev}$ values are the state variable values calculated by the other solvers at the previous synchronization point. The primary difference between Equation VI.13 and the above Equation VI.22, is in that in Equation VI.22 the functions of $\mathbf{f}$ are required to use values of the state variables, $\mathbf{x}_n$, at the last synchronization point to calculate the state variable derivative values, $\dot{\mathbf{x}}_n$. In Equation VI.13 the functions $\mathbf{f}$ use the current values of the state variables to calculate the state variable derivative values. This partitioned system, where each RKF4,5 solver independently sets its own time step values, is a form of Multi-Rate Integration, which is investigated in [58] and [62].

The $\mathbf{x}_{n\_x\_prev}$ values are stored in a shared memory that is readable by all threads, but only one thread will write to a block of *prev* values to avoid cache line sharing problems.

The synchronization points will update the *prev* values according to:

$$\begin{aligned}
\mathbf{x}_{n_0_prev} &= \mathbf{x}_{n_0} \\
\mathbf{x}_{n_1_prev} &= \mathbf{x}_{n_1} \\
&\cdots \\
\mathbf{x}_{n_(\ell-1)_prev} &= \mathbf{x}_{n_(\ell-1)}.
\end{aligned} \tag{VI.23}$$

Each independent RKF4,5 solver is responsible for integrating its portion of state variable derivative values:

$$\begin{aligned}
RKF4,5_0(\dot{\mathbf{x}}_{n_0}) &\rightarrow \mathbf{x}_{n_0} \\
RKF4,5_1(\dot{\mathbf{x}}_{n_1}) &\rightarrow \mathbf{x}_{n_1} \\
&\cdots \\
RKF4,5_{\ell-1}(\dot{\mathbf{x}}_{n_(\ell-1)}) &\rightarrow \mathbf{x}_{n_(\ell-1)}.
\end{aligned} \tag{VI.24}$$

Due to the fact that the solvers are independent, the solvers will be forced to integrate their set of $\dot{\mathbf{x}}_n$ values without a full view of the system state. This can lead to errors in the integration and incorrect simulation results because the solvers are designed to integrate an entire system, not simply a small part of it (however, we did not have a problem with accuracy in our simulations). Another source of errors is that the there will need to be an individual Jacobian function calculated for each solver based on the subset of $\mathbf{f}$ for which the solver is responsible. There will be state variables used in the equations of the subset of $\mathbf{f}$ that are not a part of a solver's assigned subset of $\mathbf{x}_n$. Therefore, when the Jacobian function is calculated, these extra variables will be treated as constants instead of as system state variables. This will make the Jacobian matrix incomplete because it will not include all of the information that it would if the system were not partitioned. As an example of

this, consider this system of state equations:

$$\begin{bmatrix} \dot{x}_1 \\ \dot{x}_2 \\ \dot{x}_3 \\ \dot{x}_4 \end{bmatrix} = \begin{bmatrix} a_1 & b_1 & c_1 & 0 \\ 0 & b_2 & c_2 & d_2 \\ a_3 & 0 & c_3 & d_3 \\ a_4 & b_4 & 0 & d_4 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix}. \tag{VI.25}$$

Since this system is so simple, the Jacobian matrix of this system matches the system matrix above:

$$J = \begin{bmatrix} a_1 & b_1 & c_1 & 0 \\ 0 & b_2 & c_2 & d_2 \\ a_3 & 0 & c_3 & d_3 \\ a_4 & b_4 & 0 & d_4 \end{bmatrix}. \tag{VI.26}$$

However, if this system is divided into two independent systems, as we do for the partitioned RKF4,5 parallel approach, the equations for the first system become:

$$\begin{bmatrix} \dot{x}_1 \\ \dot{x}_2 \end{bmatrix} = \begin{bmatrix} a_1 & b_1 & c_1 & 0 \\ 0 & b_2 & c_2 & d_2 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_{3\_prev} \\ x_{4\_prev} \end{bmatrix} \tag{VI.27}$$

$$J = \begin{bmatrix} a_1 & b_1 \\ 0 & b_2 \end{bmatrix}, \tag{VI.28}$$

and the equations for the second system become:

$$\begin{bmatrix} \dot{x}_3 \\ \dot{x}_4 \end{bmatrix} = \begin{bmatrix} a_3 & 0 & c_3 & d_3 \\ a_4 & b_4 & 0 & d_4 \end{bmatrix} \begin{bmatrix} x_{1\_prev} \\ x_{2\_prev} \\ x_3 \\ x_4 \end{bmatrix} \tag{VI.29}$$

$$F = \begin{bmatrix} c_3 & d_3 \\ 0 & d_4 \end{bmatrix}. \tag{VI.30}$$

where the *prev* values are the state variable values from the previous synchronization point. When comparing the full system description in Equation VI.25 to the two partial descriptions in Equations VI.27 and VI.29 we can see that the full description is able to use current values of the state variables to calculate the state variable derivatives, while the partitioned models have to use outdated state variable values to calculate their state variable derivatives. Also notice that the two Jacobian matrices for the partitioned systems, Equations VI.28 and VI.30, are much smaller and do not contain all of the data present in the full matrix, Equation VI.26. Specifically, the Jacobian matrix in Equation VI.28 is missing the partial derivatives with respect to x_3 and x_4, and the Jacobian matrix in Equation VI.30 is missing the partial derivatives with respect to x_1 and x_2.

Taken together, the two factors of out of date data and an incomplete Jacobian matrix can lead to significant problems for this simulation approach. The only option we have to control this potential problem is to reduce the synchronization time interval, because a faster synchronization time, like a smaller step size, will help to reduce errors in simulation. Producing an accurate simulation data will require balancing between setting the synchronization interval small enough that the errors in the approximations are small, but not setting the synchronization interval so small that there is no performance benefit.

The memory structure of the simulation is shown in Figure 60. The structure is similar to the full partitioned memory version 2, but there are two local memory blocks for each thread: one for $\mathbf{x}_n$ and one for $\dot{\mathbf{x}}_n$. The threads read and write to their local memory blocks at every function evaluation. The threads only update the shared memory block at the synchronization points, but can read from it at any time.

The program flow for the partitioned RKF4,5 simulation is somewhat more complex than the previous program flows and is shown in Figure 61. The synchronization points, where we update the shared memory with the most recent values of $\mathbf{x}_n$, only happen when

Total Threads	Slow Time Constants		Fast Time Constants	
	Rel Speedup	Std Dev	Rel Speedup	Std Dev
1	1.00	1.25×10^{-02}	1.00	4.70×10^{-03}
2	1.75	3.78×10^{-02}	1.76	5.60×10^{-03}
3	2.17	0.23	2.30	2.64×10^{-03}
4	2.81	1.25×10^{-02}	2.84	1.11×10^{-02}
5	2.41	3.84×10^{-02}	2.17	1.88×10^{-02}
6	2.68	6.53×10^{-02}	2.41	1.97×10^{-02}
7	2.60	0.39	2.64	2.55×10^{-02}
8	2.53	0.52	2.81	2.13×10^{-02}

Table 25: Relative speedups and standard deviations for the Partitioned RKF4,5 approach on the complex RLC model with 288 state variables.

Total Threads	Slow Time Constants		Fast Time Constants	
	Rel Speedup	Std Dev	Rel Speedup	Std Dev
1	1.00	1.00×10^{-02}	1.00	9.23×10^{-03}
2	1.81	2.75×10^{-02}	1.93	1.22×10^{-02}
3	2.43	6.65×10^{-03}	2.72	7.89×10^{-03}
4	3.14	1.61×10^{-02}	3.56	1.27×10^{-02}
5	2.46	2.47×10^{-02}	2.12	0.15
6	2.87	1.06×10^{-02}	2.01	0.39
7	1.44	0.20	0.32	4.15×10^{-02}
8	1.51	1.07×10^{-02}	0.33	1.89×10^{-03}

Table 26: Relative speedup and standard deviations for the Partitioned RKF4,5 approach on the complex RLC model with 804 state variables.

the RKF4,5 solvers pause to provide output. We do not synchronize after every time step like we do for the fixed step algorithms detailed above. In this program flow, like the full partitioned memory approaches described above, all of the threads in $\mathbf{T}$ are responsible for integrating a portion of $\dot{\mathbf{x}}_n$, and they all synchronize their assigned variables.

VI.5.5.1 Experimental Runs to Evaluate Speedup

This approach generally performed very well. The relative speedups for the models in Figures 32, 33, and 36 are shown in Tables 25, 26, and 27, respectively. Figure 62 presents

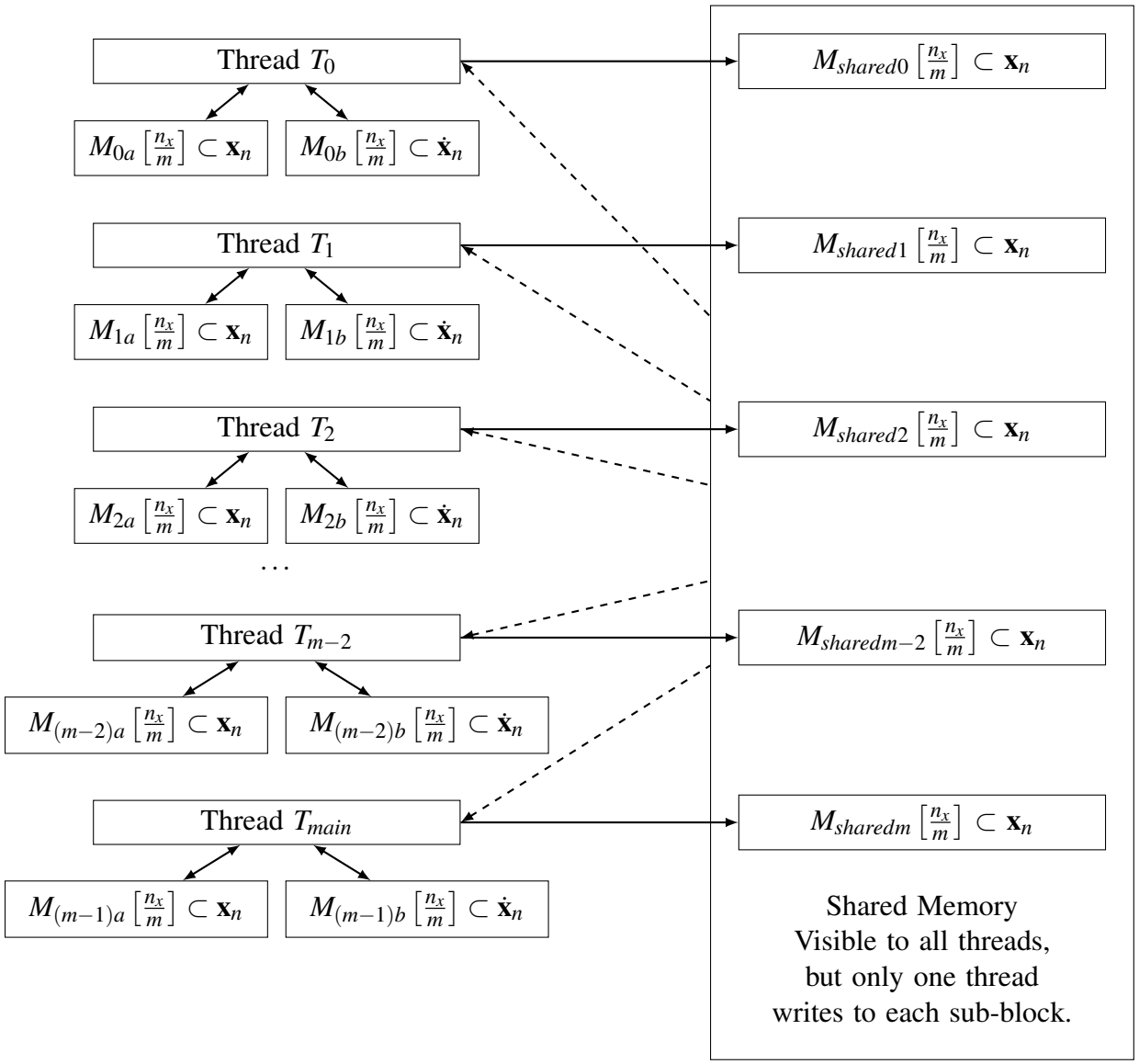


Figure 60: Figure describing full partitioned memory implementation of a partitioned RKF4,5 simulation when using agglomeration and 1 thread/RKF4,5 solver per CPU core. The dashed lines indicate a read-only relationship.

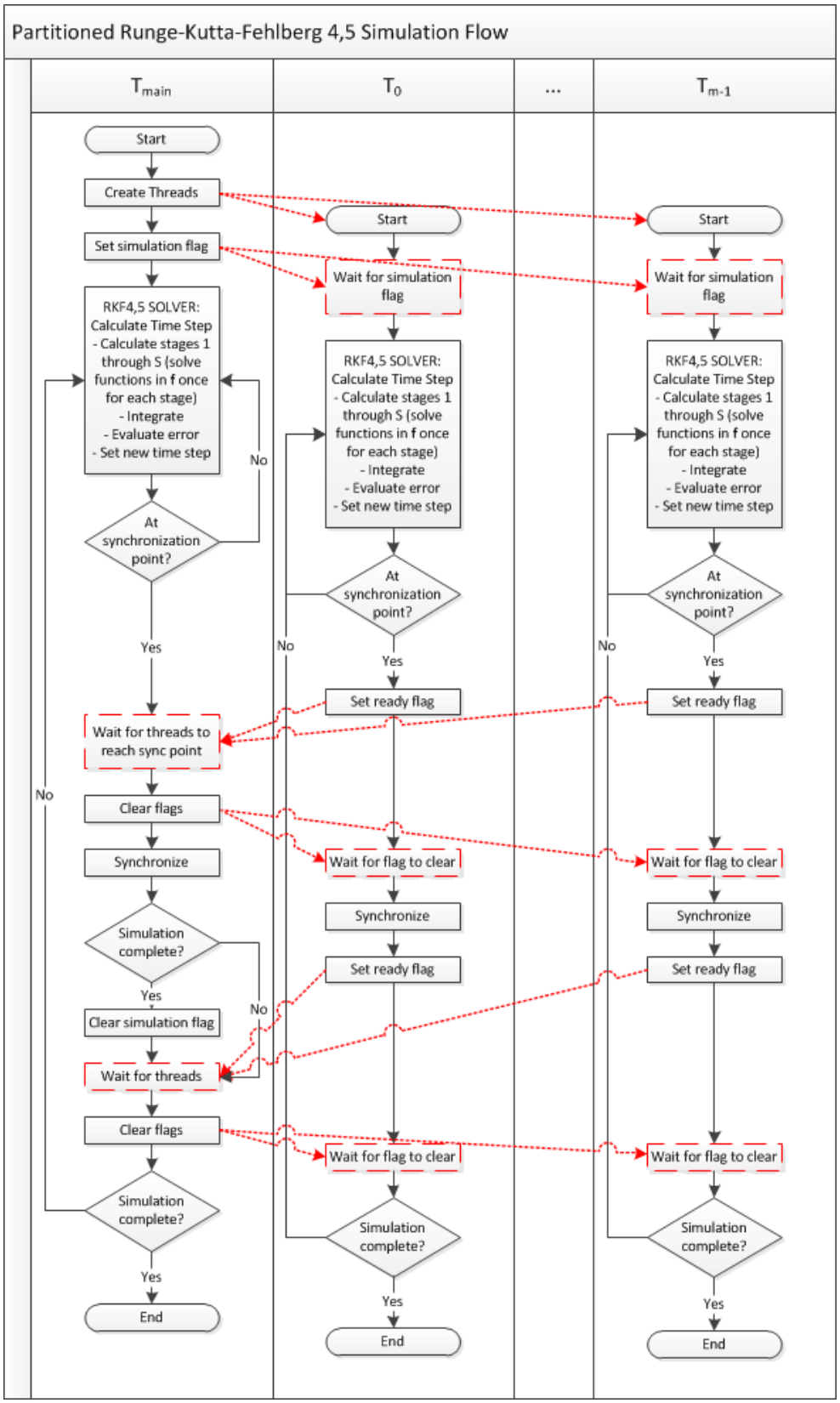


Figure 61: Partitioned RKF4,5 parallel simulation program flow. The red dashed lines indicate communication between threads, and the boxes with red dashes are wait states.

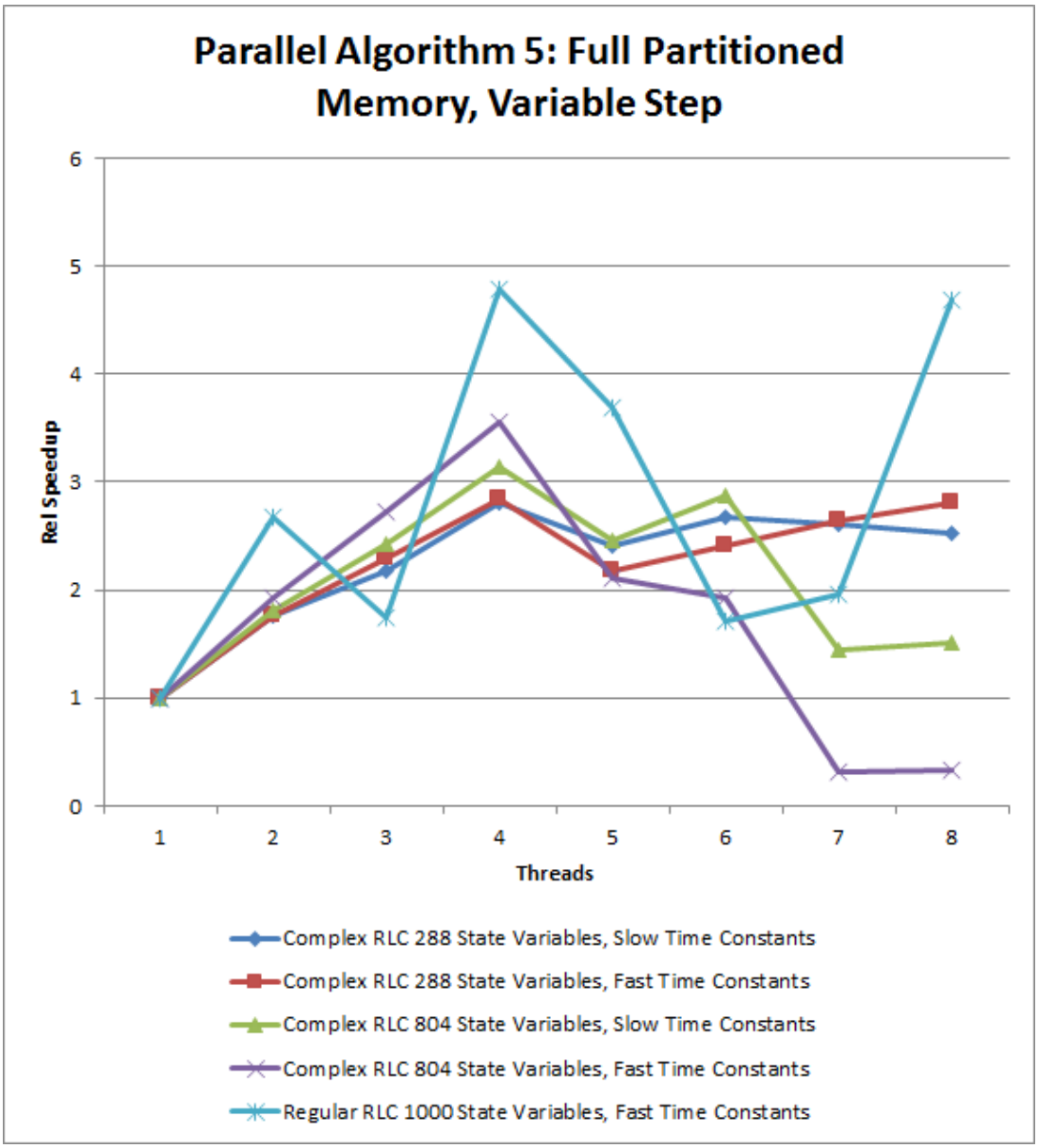


Figure 62: Figure showing the relative speedups across the models for the variable step Full Partitioned Memory algorithms.

Total Threads	Fast Time Constants	
	Rel Speedup	Std Dev
1	1.00	2.84×10^{-02}
2	2.76	0.35
3	1.80	0.24
4	5.23	0.86
5	3.95	0.61
6	1.77	0.23
7	2.07	0.31
8	5.15	0.86

Table 27: Relative speedups and standard deviations for the Partitioned RKF4,5 approach on the regular RLC model with 1000 state variables.

the relative speedups across all models. The relative speedup numbers were calculated by comparing the simulation times to a single threaded, serial, variable step simulation.

These results show that for the complex RLC model with 288 state variables the partitioned RKF4,5 method was able to match the serial simulation or provide a speedup when using 2 through 8 threads. The models with fast and slow time constants had similar performance, with the best speedup of approximately 2.8 coming when 4 threads were used.

For the complex RLC model with 804 state variables and slow time constants the partitioned RKF4,5 method was able to provide a speedup when using 2 through 8 threads. When simulating the model with using fast time constants the algorithm was able to provide a speedup when using 2 through 6 threads. When using 7 and 8 threads, the model with fast time constants did not provide a speedup.

The regular RLC model with 1000 state variables provided the best speedup so far. When using 2 threads it produced a speedup of 2.7 compared to the serial variable step simulation. A speedup of greater than the number of threads is vary rare, described as superlinear speedup [8][53], but it is possible in this case because of how the system state equations are partitioned across the threads. When using 4 threads the method again achieved superliner speedup of 4.8. Overall, this approach provided a speedup when using 2 through 8 threads.

The RKF4,5 approach on the regular RLC model with 1000 state variables also has unexpected behavior losses when using 3, 6, and 7 threads. The reasons for the loss of performance when using these threads is unclear. However, 3, 6, and 7 are all uneven divisors of 1000. While we do not see an effect due to unequal partitioning on the other models, it is possible that it appears here due to the regular structure of the model. The problem is also potentially due to cache line sharing between the solvers that only manifests itself when using a thread count that is an unequal divisor of the number of the state variables. We are unable to control where the solvers are allocated in memory and it is possible that the allocation function does not assign the solver memory block to a cache line boundary.

We performed a Mean Squared Error (MSE) analysis for all 5 RLC models at a variety of synchronization intervals, and those results are shown in Tables 28 through 32. We compared our parallel simulations to the same serial simulations used in the MSE analysis for Algorithm 4. In all cases the simulations that used a synchronization interval equal to the interval used in the relative speedup tests produced simulation data that was very accurate, with the largest mean square error on the order of 10^{-5}; which occured on the first state variable of the complex RLC model with 288 state variables. Also, for all cases the simulations with a synchronization interval that is smaller than the baseline test produced very accurate data, again with the largest error on the order of 10^{-5}. This indicates that the synchronization interval used in the relative speedup tests generated correct simulation data.

For the complex RLC models, except for the model with 288 state variables and slow time constants, when we tried to increase the synchronization interval above what was used in the relative speedup tests, the simulation generated very large MSE estimates with a very small change in in the synchronization interval. In most cases the change in the synchronization interval was less than half an order of magnitude. This is typical behavior for a simulation as shown in Cellier and Kofman [35].

We were not able to force the simulation into instability for the regular RLC model

Variable	Variable Step Simulation Mean-Square Error at Synchronization Interval			
	0.001	*0.1*	0.5	1.0
1	1.25×10^{-5}	1.25×10^{-5}	1.38×10^{-9}	2.64×10^{-3}
2	1.63×10^{-8}	1.63×10^{-8}	3.32×10^{-9}	8.05×10^{-3}
3	1.36×10^{-8}	1.35×10^{-8}	9.72×10^{-9}	7.41×10^{-3}
4	2.11×10^{-9}	2.11×10^{-9}	2.81×10^{-9}	1.73×10^{-2}
285	2.12×10^{-19}	2.03×10^{-18}	4.94×10^{-17}	3.11×10^{-13}
286	7.37×10^{-16}	6.85×10^{-16}	1.23×10^{-14}	3.56×10^{-13}
287	5.70×10^{-20}	5.13×10^{-19}	1.25×10^{-17}	1.38×10^{-13}
288	7.35×10^{-16}	6.84×10^{-16}	1.23×10^{-14}	1.89×10^{-13}

Table 28: Mean square error calculations for 8 state variables of the model with 288 state variables and slow time constants using 8 threads. The italicised colum header indicates the synchronization interval that was used for the relative speedup experiments. The baseline simulation used to derive these error calculations was a fixed step simulation that used a timestep of 0.01 seconds.

with 1000 state variables (see Table 32). Even with a synchronization interval of 5 seconds the simulation still produced small error estimates. This likely because there is little interaction and dependence between the partitions, so each partition does not need frequently information from the other partitions in order to meet its accuracy requirements.

As with parallel Algorithm 4, these error analysis results show that when using a parallel simulation a small change in the simulation synchronization interval can move a simulation from small errors to large errors. It also shows that, because the parallel simulations had very small error when using the same synchronization interval as the base simulation, that the process of parallelizing a simulation does not negatively affect the accuracy of the simulation.

VI.5.6 Parallel Algorithms Type 6: Algebraic Loop Simulation

Algebraic loops are very common in modern complex models, and an example of how to calculate the computational model of a system with an algebraic loop is described in

Variable	Variable Step Simulation Mean-Square Error at Synchronization Interval			
	0.00001	*0.001*	0.002	0.005
1	2.63×10^{-11}	1.05×10^{-10}	3.14×10^{-10}	inf
2	1.44×10^{-7}	9.88×10^{-8}	8.77×10^{-8}	inf
3	2.24×10^{-11}	1.15×10^{-10}	3.38×10^{-10}	inf
4	1.01×10^{-7}	4.92×10^{-8}	1.25×10^{-7}	inf
285	7.99×10^{-16}	1.51×10^{-14}	5.74×10^{-14}	inf
286	4.53×10^{-11}	1.24×10^{-9}	5.27×10^{-9}	inf
287	2.02×10^{-16}	3.77×10^{-15}	1.44×10^{-14}	inf
288	4.67×10^{-11}	1.24×10^{-9}	5.28×10^{-9}	inf

Table 29: Mean square error calculations for 8 state variables of the model with 288 state variables and fast time constants using 8 threads. The italicised colum header indicates the synchronization interval that was used for the relative speedup experiments. The baseline simulation used to derive these error calculations was a fixed step simulation that used a timestep of 0.0001 seconds.

Variable	Variable Step Simulation Mean-Square Error at Synchronization Interval			
	0.01	*0.1*	0.2	0.5
1	1.25×10^{-5}	1.25×10^{-5}	3.05×10^{113}	inf
2	1.63×10^{-8}	1.63×10^{-8}	3.54×10^{115}	inf
3	1.36×10^{-8}	1.36×10^{-8}	1.91×10^{115}	inf
4	2.11×10^{-9}	2.11×10^{-9}	2.00×10^{117}	inf
801	6.61×10^{-21}	2.06×10^{-19}	1.32×10^{99}	inf
802	6.92×10^{-17}	5.10×10^{-16}	2.97×10^{99}	inf
803	1.75×10^{-21}	5.19×10^{-20}	2.38×10^{98}	inf
804	6.89×10^{-17}	5.05×10^{-16}	1.18×10^{99}	inf

Table 30: Mean square error calculations for 8 state variables of the model with 804 state variables and slow time constants using 8 threads. The italicised colum header indicates the synchronization interval that was used for the relative speedup experiments. The baseline simulation used to derive these error calculations was a fixed step simulation that used a timestep of 0.01 seconds.

Variable	Variable Step Simulation Mean-Square Error at Synchronization Interval			
	0.00005	*0.0004*	0.0005	0.001
1	2.76×10^{-11}	3.16×10^{-11}	3.27×10^{305}	inf
2	1.37×10^{-7}	1.30×10^{-7}	3.58×10^{305}	inf
3	2.54×10^{-11}	3.36×10^{-11}	3.33×10^{305}	inf
4	8.86×10^{-8}	8.96×10^{-8}	2.14×10^{305}	inf
801	9.65×10^{-18}	2.34×10^{-17}	3.20×10^{305}	inf
802	9.56×10^{-13}	3.52×10^{-11}	inf	inf
803	2.45×10^{-18}	5.90×10^{-18}	2.38×10^{305}	inf
804	9.86×10^{-13}	3.52×10^{-11}	inf	inf

Table 31: Mean square error calculations for 8 state variables of the model with 804 state variables and fast time constants using 8 threads. The italicised colum header indicates the synchronization interval that was used for the relative speedup experiments. The baseline simulation used to derive these error calculations was a fixed step simulation that used a timestep of 0.0001 seconds.

Variable	Variable Step Simulation Mean-Square Error at Synchronization Interval			
	0.00001	*0.0001*	0.001	5.0
1	8.31×10^{-11}	8.31×10^{-11}	8.31×10^{-11}	1.32×10^{-10}
2	7.65×10^{-9}	7.65×10^{-9}	7.64×10^{-9}	9.84×10^{-8}
3	1.38×10^{-10}	1.38×10^{-10}	1.38×10^{-10}	1.38×10^{-10}
4	8.11×10^{-9}	8.11×10^{-9}	8.11×10^{-9}	9.44×10^{-8}
997	2.87×10^{-282}	1.55×10^{-282}	1.41×10^{-25}	1.67×10^{-276}
998	9.21×10^{-284}	4.99×10^{-284}	4.24×10^{-24}	5.29×10^{-278}
999	1.82×10^{-285}	9.91×10^{-286}	3.46×10^{-26}	1.03×10^{-279}
1000	6.07×10^{-287}	3.30×10^{-287}	1.85×10^{-21}	3.39×10^{-281}

Table 32: Mean square error calculations for 8 state variables of the model with 1000 state variables and fast time constants using 8 threads. The italicised colum header indicates the synchronization interval that was used for the relative speedup experiments. The baseline simulation used to derive these error calculations was a fixed step simulation that used a timestep of 0.00001 seconds.

Section III.5. Algebraic loops are typically solved using the Newton Iteration method (detailed in Section III.6). For non-linear algebraic loops the number of iterations is unknown, because it depends on the convergence time of the Newton Iteration, which is dependent on the initial guess provided to the algorithm. It is reported that nonlinear loops have a computational complexity of $O(n^3)$ [28] [45], so our approach to parallelize the simulation of a model that contains algebraic loops will focus on solving the algebraic loops in parallel, and then perform the numerical integration serially. We use KINSOL from the SUNDIALS solver library [20] from Lawrence Livermore National Laboratory [11] as our algebraic loop solver. We use the same fixed step (Forward Euler, Equation IV.7) and variable step (RKF4,5) numerical integration methods that we used in the above tests for experiments on models with algebraic loops.

The models have a number of properties that allow us to simplify the simulation algorithms. The first property is that the variables calculated by the loops are from the set $\mathbf{w}_n$ from Equation VI.1. All of the loops need to be solved before the system state variables are calculated, because the equations in $\mathbf{f}_x$ may depend on the values of $\mathbf{w}_n$ calculated in the loops in order to calculate the state variable derivatives, $\dot{\mathbf{x}}_n$. Another property is that all of the loops are independent of each other, and therefore can be solved in any order for each time step. In addition, each thread is assigned approximately the same number of loops to solve for each time step. Once all the loops are solved, the state variable derivatives are calculated and then integrated to end the simulation time step. The program flow is shown in Figure 63. We applied this method to the fixed step and variable step simulation algorithms for our experiments.

The memory management scheme employed is shown in Figure 64. Each thread is assigned a group of algebraic loops to solve, $0, \ldots, pT$, where pT is the number of loops assigned to a thread. There are m groups of loops created, where m is the number of threads being used. Each loop has its own dedicated memory block that has the same number of variables as there are equations in the loop (the sizes of the loop-specific memory blocks

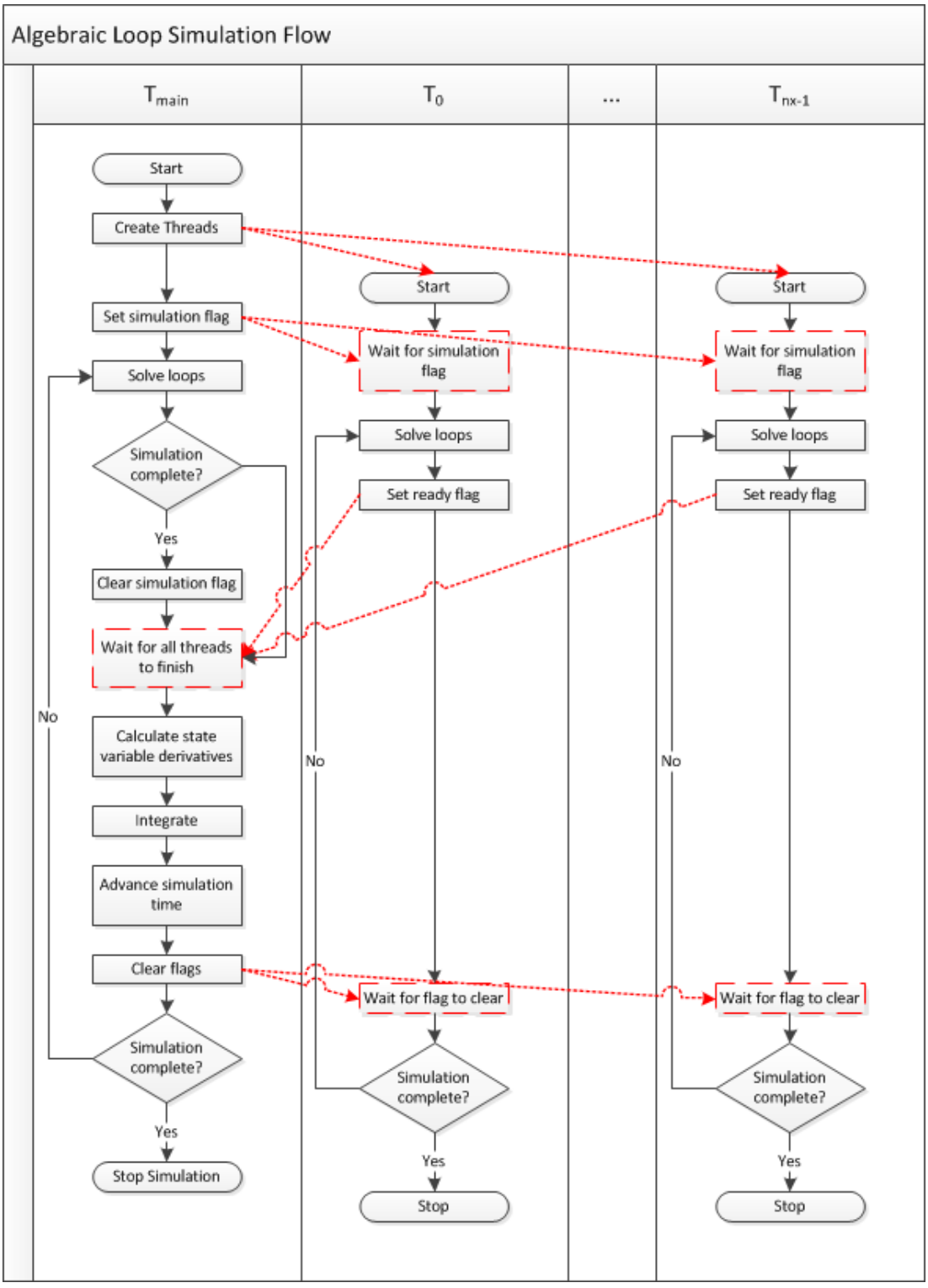


Figure 63: Algebraic loop simulation program flow. The red dashed lines indicate communication between threads, and the boxes with red dashes are wait states.

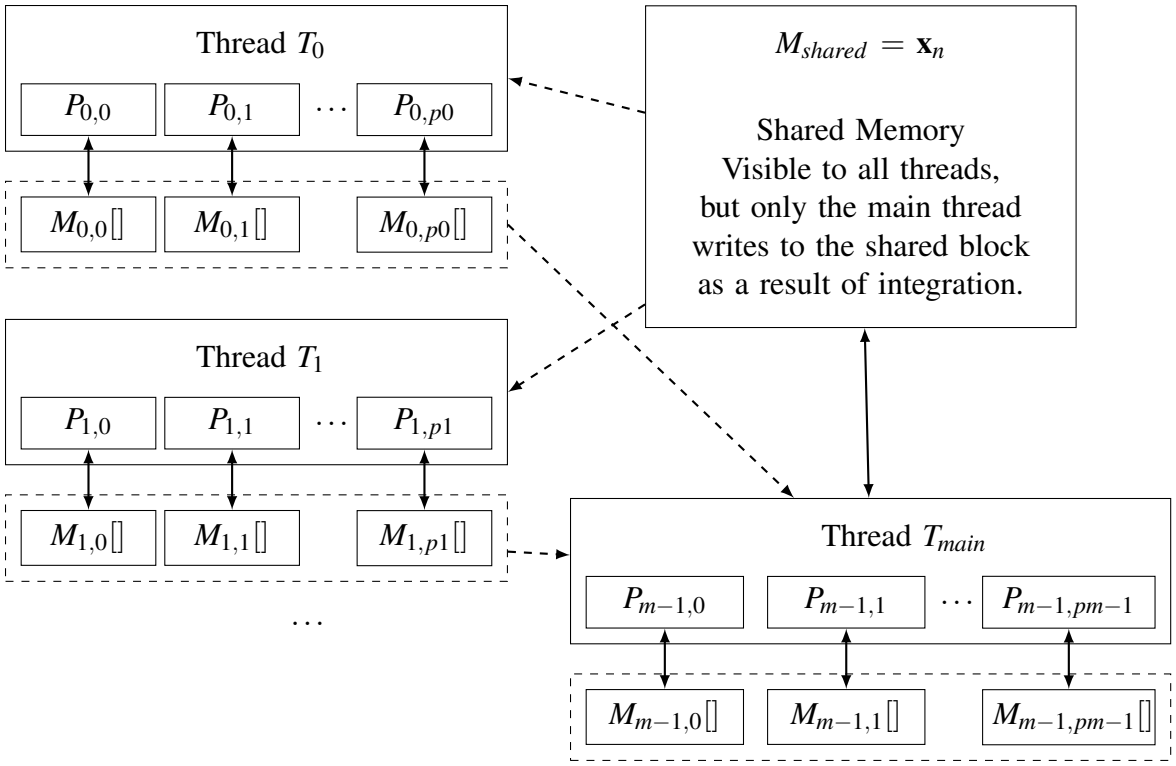


Figure 64: Figure describing memory implementation parallel loop simulation. The dashed lines indicate a read-only relationship.

was omitted in the figure to save space). Solving the loops will require data from the state variables, $\mathbf{x}_n$, so each thread needs read access to the main shared memory so it can pass that data to the loop solvers. The main thread is assigned a subset of algebraic loops to solve, and is also responsible for integrating the state variable derivatives (see Figure 63 above). Some of the calculations for the state variable derivatives may depend on the variables calculated from the algebraic loops, which means that the main thread needs read access to each loop specific memory block.

When developing this approach for parallelizing the solving of algebraic loops, we elected to not parallelize the variable step solver, because parallelizing it would likely add much more computational work to the simulation, and likely end up slowing down the simulation instead of making it faster. The reason for the potential slowdown is due to the fact

that solving the algebraic loops is embedded the function $\mathbf{f}_x$ from Equation VI.1. When partitioning the model as described in Section VI.5.5 in Equation VI.22, some of the loops in $\mathbf{f}_x$ will be duplicated and assigned to multiple partitions because of dependencies between the loops and the functions to calculate the state variable derivatives, $\dot{\mathbf{x}}_n$. As an example of the possible extra work potentially involved in parallelizing the integration of the RKF4,5 method, consider the computational work to integrate across a timestep for a system with 4 state variables. The RKF4,5 solver is a 6-stage method [38], meaning there are 6 function evaluations to integrate across a time step. When integrating a model that has 4 algebraic loops, to integrate the model across one time step will require solving the 4 loops 6 times, for a total of 24 loop evaluations. If we partition the systen into 4 components, with one state varible in each component as described in Section VI.5.5, in the worst case integrating each partition will require solving each of the 4 loops. Therefore, integrating the entire model across 1 time step (even though the time step sizes may be different for each partition) will require 4 loops $\times 4$ partitions $\times 6$ stages $= 96$ loop evaluations. Since loops are so computationally expensive to solve, adding the extra loop evaluations above the serial case will likely increase the amount of time required to solve the system. This is only a worst case analysis, however, and a real model will not likely suffer from this problem. We leave partitioning the integration of a system with algebraic loops for future work.

VI.5.6.1 Experimental Runs to Evaluate Speedup

The relative speedups and standard deviations of the models with algebraic loops are shown in Tables 33, 34, and 35. Figures 65 and 66 present the speedups graphically. To calculate these speedups, the fixed step parallel simulations were compared to a fixed step serial simulation, and the variable step parallel simulations were compared to a variable step serial simulation. Solving the loops in parallel generally had very good performance, and every test, except for the single threaded experiments which served as a control, showed a speedup.

	Slow Time Constants			
	Fixed Step		Variable Step	
Total Threads	Rel Speedup	Std Dev	Rel Speedup	Std Dev
1	1.00	2.26×10^{-03}	1.00	9.97×10^{-03}
2	1.59	3.65×10^{-03}	1.81	2.94×10^{-02}
3	2.06	1.74×10^{-03}	2.33	6.65×10^{-03}
4	2.32	5.53×10^{-02}	3.06	7.34×10^{-03}
5	1.99	7.80×10^{-02}	2.63	0.24
6	1.99	7.10×10^{-02}	2.43	9.92×10^{-02}
7	1.70	6.32×10^{-02}	2.46	7.42×10^{-02}
8	1.67	1.95×10^{-02}	2.40	1.94×10^{-02}

	Fast Time Constants			
	Fixed Step		Variable Step	
Total Threads	Rel Speedup	Std Dev	Rel Speedup	Std Dev
1	1.00	1.85×10^{-03}	1.00	3.49×10^{-03}
2	1.85	2.38×10^{-03}	1.76	9.48×10^{-03}
3	2.50	1.30×10^{-03}	2.36	5.09×10^{-03}
4	3.32	4.47×10^{-03}	3.20	9.42×10^{-03}
5	2.69	8.58×10^{-03}	2.49	0.16
6	2.95	7.91×10^{-02}	2.62	4.99×10^{-02}
7	3.32	1.10×10^{-02}	3.04	7.62×10^{-02}
8	3.31	9.94×10^{-03}	2.97	3.69×10^{-02}

Table 33: Relative speedup and standard deviation for Parallel Algebraic Loop approach using fixed step and variable step integration methods on the complex RLC model with loops and 288 state variables.

	Slow Time Constants			
	Fixed Step		Variable Step	
Total Threads	Rel Speedup	Std Dev	Rel Speedup	Std Dev
1	1.00	9.70×10^{-03}	1.00	1.19×10^{-02}
2	1.46	1.65×10^{-02}	1.72	2.08×10^{-02}
3	1.95	2.65×10^{-03}	2.42	9.84×10^{-03}
4	2.38	3.47×10^{-03}	3.14	1.31×10^{-02}
5	1.83	0.11	2.56	5.04×10^{-02}
6	2.14	0.13	2.93	0.17
7	2.10	8.79×10^{-02}	2.91	3.50×10^{-02}
8	2.11	7.71×10^{-03}	3.11	5.02×10^{-02}
	Fast Time Constants			
	Fixed Step		Variable Step	
Total Threads	Rel Speedup	Std Dev	Rel Speedup	Std Dev
1	1.00	3.81×10^{-03}	1.00	2.07×10^{-03}
2	1.92	7.05×10^{-03}	1.88	8.40×10^{-03}
3	2.52	2.19×10^{-03}	2.11	1.46×10^{-03}
4	3.33	3.37×10^{-02}	2.88	0.26
5	2.72	3.04×10^{-02}	2.42	0.11
6	2.94	3.80×10^{-02}	2.53	0.15
7	3.46	1.18×10^{-02}	2.67	0.22
8	3.49	2.02×10^{-02}	2.43	3.85×10^{-02}

Table 34: Relative speedup and standard deviation for Parallel Algebraic Loop approach using fixed step and variable step integration methods on the complex RLC model with loops and 804 state variables.

	Fast Time Constants			
	Fixed Step		Variable Step	
Total Threads	Rel Speedup	Std Dev	Rel Speedup	Std Dev
1	1.00	6.04×10^{-03}	1.00	5.67×10^{-03}
2	1.81	2.38×10^{-02}	1.66	8.96×10^{-03}
3	2.30	3.53×10^{-02}	2.01	1.58×10^{-02}
4	2.63	8.26×10^{-03}	2.46	2.86×10^{-02}
5	2.26	1.27×10^{-02}	2.23	0.15
6	2.47	2.82×10^{-02}	1.95	0.13
7	2.48	2.23×10^{-02}	1.84	9.95×10^{-02}
8	2.54	2.92×10^{-02}	1.79	9.81×10^{-03}

Table 35: Relative speedup and standard deviation for Parallel Algebraic Loop approach using fixed step and variable step integration methods on the regular RLC model with loops and 1000 state variables.

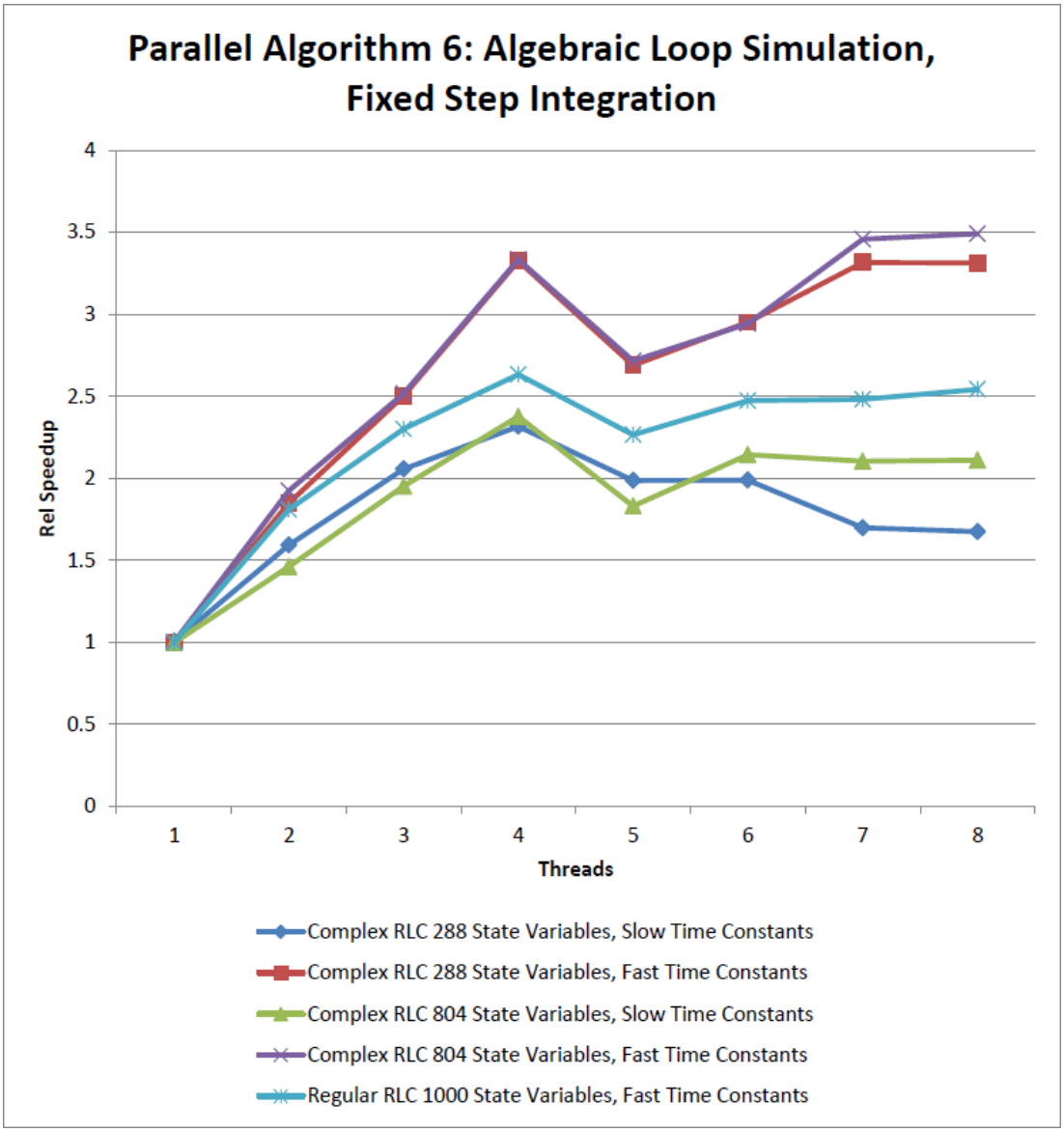


Figure 65: Figure showing the relative speedups across the models with algebraic loops using fixed step numerical integration.

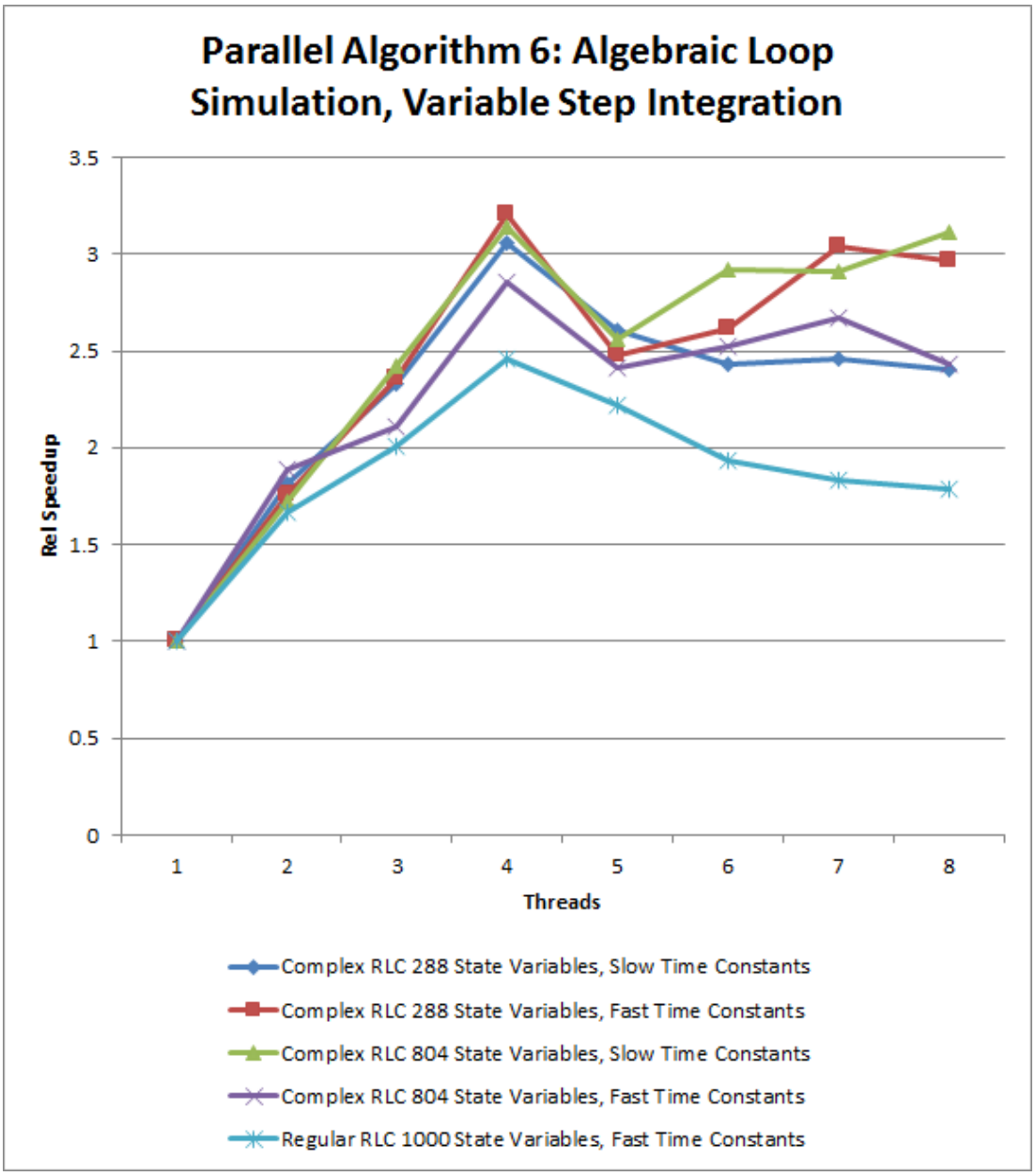


Figure 66: Figure showing the relative speedups across the models with algebraic loops using variable step numerical integration.

Total Threads	Fixed Step	Variable Step
1	1.00	1.00
2	1.23	1.26
3	1.23	1.27
4	1.23	1.25

Table 36: Relative speedup for the Water Recovery System.

The variable step simulation performed slightly better than the fixed step simulation on the models with slow time constants. On these models the variable step simulation achieved a maximum speedup of slightly more than 3 when using 4 threads, while the fixed step simulation only achieved a speedup of about 2.4 when using 4 threads.

The fixed step and variable step simulations had near identical performance on the complex RLC model with 288 state variables and fast time constants; when using 4 threads the fixed step simulation acheiving a maximum speedup of 3.3 and the variable step simulation achieving a maximum speedup of 3.2. The results between the fixed step and variable step simulations are likely similar because the processing of the algebraic loops is dominating the simulation run time, and therefore there is no advantage to using a fixed step or variable step solver.

The complex RLC model with 804 state variables and fast time constants provided interesting results because the fixed step simulation performed better than the variable step simulation. The fixed step simulation achieved a speedup of 3.5 when using 8 threads (it achieved a speedup of 3.33 when using 4 threads), while the variable step simulation achieved a speedup of only 2.88. The fixed step simulation also performed better than the varible step simulation on the regular RLC model with 1000 state variables, though the differences were not as pronounced with the fixed step solver achieving a speedup of 2.63, and the variable step solver achieving a speedup of 2.46. These are the only two instances where this happened, and is likely due to, again, the processing of the algebraic loops dominating the computation time, which allows the computational efficiency of the fixed step simulation to provide a speedup over the variable step simulation.

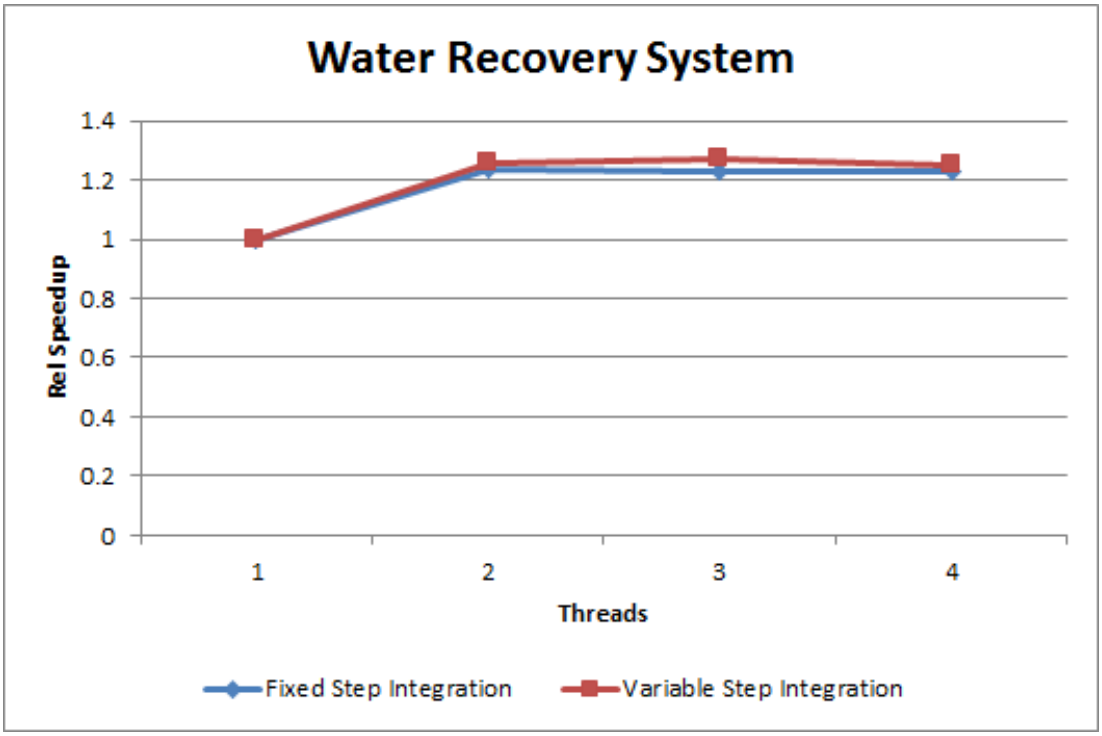


Figure 67: Figure showing the relative speedups for the Water Recovery System.

VI.5.7 Case Study: Water Recovery System Simulation

This section describes the simulation results for the WRS system simulations. The WRS has only 4 algebraic loops, so we were only able to partition the model across 4 threads. The relative speedups for the simulation are shown in Table 36 and in Figure 67. The fixed step simulation provided a speedup of 1.23 and the variable step simulation provided a speedup of 1.11. The speedup is less than we saw in the previous section. The reason for the smaller speedup values is due to the different sizes of the algebraic loops. In the previous section the algebraic loops were all approximately the same size. However, in the WRS the size of the loops varied greatly with 21, 12, 17, and 56 equations in each loop. In this case the large loop dominated the simulation time, because in the worst case its computational complexity is $O(n^3) = O(56^3) = O(175616)$, while the worst case complexity for the next smallest loops is $O(n^3) = O(21^3) = O(9261)$. The large loop has a complexity value of almost 19 times that of the next smaller loop.

A future improvement to our partitioning of algebraic loops is to consider the sizes of the loops when we assign them to a partition. This will allow for an even load balancing between the threads, which will allow for greater speedups in models with algebraic loops.

VI.6 Optimizing Implementation for Speed

We were interested in optimizing the runtime performance of our simulations so we investigated programs we could use to characterize our simulations to find functions and code that were slowing down the simulation. The two pieces of software we looked at were Intel VTune Amplifier XE 2015 [9] and Valgrind [26]. The primary benefit Valgrind has over Intel's software in that it is free, and is released under a GNU General Public License, while Intel's software can cost many thousands of dollars for a commercial license. However, Intel's software is free for students, and we were able to take advantage of this to obtain our copy. The Valgrind software is in some ways not as advanced as Intel's software because it was not able to profile our parallel simulations because it blocked our threads from communicating, therefore causing our simulations to block. Intel's software did not interfere with our thread communication, so we used it to profile our experiments. It is unfortunate that Valgrind was not able to characterize our simulations because there is a portion of the program dedicated to simulating a processor's cache, and this would have been useful in determining the optimal program architecture to take advantage of the cache.

The changes derived from the VTune Amplifier analysis can be broadly described as moving performance critical code that is run a large number of times from C++ classes to C, and there were 3 primary changes we made to our simulation code to reduce the simulation time. One change was to move to using C-style arrays from using C++ `stl::vectors` for items that are accessed frequently. The VTune Amplifier identified the function `operator[]` as taking a large amount of time in our simulations. In C++ the ability to overload operators is a great advantage of the language, but even though the `operator[]`

function of a `stl::vector` may be able to directly index into the `stl::vector`'s underlying array, in order to get to that direct index the code first needs to push the `operator[]` function onto the stack. This extra function push onto the stack is avoided when using C-style arrays in place of `stl::vectors`.

Another change we made as a result of VTune Amplifier analysis is to move away from `TBB::<atomic>` data types. These are atomic variables that are a part of Intel's Threading Building Blocks parallel library [8]. Atomic variables allow multiple threads to write to, and read from, them safely without the need for explicit locking constructs such as mutexes. Due to their thread-safe nature we were sharing these variables across threads as a communication device, but were able to switch to using simple integers to accomplish the same task and reduce time.

Finally, a last change that we made as a result of our VTune Amplifier analysis was a simple good programming habit; which is to store the size of a vector or list as a variable instead of repeatedly calling vector or list's `.size()` function.

Together these changes allowed us to reduce the simulation time of the RLC circuit in Figure 33 by approximately a third from approximately 0.312 seconds to approximately 0.220 seconds for a 500 second simulation with a step size of 0.01 seconds and using 7 spawned threads and one main thread. These lessons were applied while testing the Full and Partial Shared Memory approaches listed above in Sections VI.5.2 and VI.5.3, and applied to all subsequent parallel algorithms.

VI.7 Results Summary and Discussion

This section summarizes our parallel algorithms, and experiment results, and discusses the conclusions we are able to draw from those results. As a review, Table 37 presents the key differences between the different parallel simulation algorithms.

Algorithm	Memory Structure	$\lvert\mathbf{T}_{spawn}\rvert$	Role of $\mathbf{T}_{spawn}$	Role of T_{main}	Agglomeration
Type 1	Full Shared	n_x	Calculate $\mathbf{f}_{IFE}$	Merge $\mathbf{x}_{n+1}$ into $\mathbf{x}_n$	None
Type 2	Full Shared	$m-1$	Calculate $\mathbf{f}_{IFE}$	Merge $\mathbf{x}_{n+1}$ into $\mathbf{x}_n$	Simple and Smart
Type 3	Partial Partitioned	$m-1$	Calculate $\mathbf{f}_{IFE}$	Merge $\mathbf{x}_{n+1}$ into $\mathbf{x}_n$	Simple
Type 4	Full Partitioned	$m-1$	Calculate $\mathbf{f}_{IFE}$ and Merge	Calculate $\mathbf{f}_{IFE}$ and Merge	Simple and Minimum Sharing
Type 5	Full Partitioned	$m-1$	Calculate $\dot{\mathbf{x}}_n$ and Merge	Calculate $\dot{\mathbf{x}}_n$ and Merge	Simple
Type 6	Full Partitioned	$m-1$	Solve algebraic loops	Solve algebraic loops and integrate	Simple

	Best Performance		
Algorithm	Rel Speedup	Threads	Model
Type 1	2.0×10^{-3}	8	Complex RLC 288 State Variables, Slow Time Constants
Type 2	0.92	8	regular RLC model
Type 3	0.49	4	regular RLC model
Type 4	2.53	4	regular RLC model
Type 5	5.23	4	regular RLC model
Type 6	3.33	4	Complex RLC 804 State Variables, Fast Time Constants

Table 37: Summary of parallel algorithms.

VI.7.1 Results Review

Table 37 presents a summary of the best relative speedups produced by each parallel algorithm. Our algorithms show a steady improvement, except for parallel algorithm type 3, from algorithm 1 through algorithm 5. In algorithm 1 our best speedup was on the order of 10^{-3}, which means our parallel implementation was significantly slower than the serial case. In this algorithm we were creating more threads than the CPU and the operating system were able to efficiently handle and most of the processing time of the simulation was spent on the overhead of switching between the threads instead of on advancing the simulation.

In algorithm 2 we reduced the number of threads so that the maximum number of threads used for the simulation was equal to the number of CPU cores available on our processor. This method did not provide a speedup, but it was able to match the serial simulation time. The lack of speedup for algorithm 2 was caused by not considering the CPU cache in our experiments, and the different threads had to wait for an update to their cache lines instead of completing the simulation.

In algorithm 3 we attempted to avoid the problems of cache line sharing by creating a memory structure that would prevent cache conflicts between the threads. Unfortunately this algorithm performed worse than algorithm 2 due to our memory structure significantly increasing the amount of work T_{main} needed to complete each time step. This extra work by T_{main} prevented algorithm 3 from produing a speedup.

In algorithms 4 and 5 we further enhanced our memory structure so that the problems seen in algorithm 3 were solved, and we reduced the workload of T_{main} and expanded the role of the threads in $\mathbf{T}_{spawn}$ so that all of the work of the simulation was spread across all threads. Algorithms 4 and 5 use the same memory structure and the same responsibilities of each thread, but algorithm 4 uses a fixed step numerical integration method, and algorithm 5 uses a variable step numerical integration algorithm. These two algorithms provided

good speedups compared to the serial simulation case. The best speedup of algorithm 4 was 2.53, the best speedup of algorithm 5 was 5.23, and both on the regular RLC model.

Algorithm 6 only parallelized the algebraic loops, it did not parallelize the simulation as a whole as the previous algorithms. However, in developing algorithm 6 we applied the lessons learned in algorithms 1 through 5. These lessons include: limiting the number of threads created, avoiding cache line sharing, and evenly distributing the processing across all threads. Algorithm 6 also provided a good speedup compared to a serial simulation, and the largest speedup was 3.33 on the complex RLC model with 804 state variables and fast time constants. Further conclusions are discussed in Section VI.7.3.

VI.7.2 Comparison to Dymola

We also compared our simulation runtimes to the runtimes generated by Dymola [4]. We used a `*.mos` script (a `*.mos` script is a script language specific to Modelica) to launch Dymola in no window mode, and to simulate the models in Section VI.1 using Dymola's DASSL solver, and then using Dymola's Parallel Euler solver. Dymola is one of the few commercial software packages that is able to parallelize its simulation (see [27] and Chapter I). The DASSL solver only supports using a single thread, and the Parallel Euler solver supports using multiple threads. We also used the script to instruct Dymola to not produce simulation output, to improve the consistency of the simulation times by not requiring Dymola to interact with the disk. We ran our Dymola simulations in Windows 7, using the same computer as our previous experiments. Each model was simulated 20 times, and the simulation times averaged. This average was used as the serial value in the relative speedup equation in VI.3. We compared all of the models listed in Section VI.1 to Dymola, using Algorithm 4, Minimum Sharing partitioning, Algorithm 5, and Algorithm 6, where appropriate, for the algorithms and models. We used this subset of our parallel algorithms because these algorithms provided the best speedups. We also limited our comparison to only 4 threads, as nearly every test in Section VI.5 produced its best performance when

using 4 threads. The relative speedups are in Tables 38, 39, 40, and 41, and in Figures 68, 69, 70, and 71.

Our results show a speedup over Dymola's default DASSL solver when simulating:

1. The complex RLC model with 804 state variables and fast time constants using Algorithm 4,

2. The complex RLC model with 804 state variables and slow time constants using Algorithm 5,

3. The complex RLC model with 288 state variables and slow time constants using Algorithm 6,

4. The complex RLC model with 804 state variables and slow time constants using Algorithm 6.

The best speedup we obtained with respect to Dymola's DASSL solver is 3.4, achieved using Algorithm 6 and a variable step solver to simulate the complex RLC model with 804 state variables and slow time constants. The few number of algorithms and models that provided a speedup shows the strength of Dymola's default solver. It is able to take much larger time steps than the fixed step and variable step solvers that we used in our experiments, which in turn allowed it to complete the simulation faster using a single thread than our algorithms were using multiple threads. The models and algorithms that showed the biggest speedup were the models with slow time constants and algebraic loops simulated using a variable step solver. These models performed well because the algebraic loops dominated the processing of the simulation of the DASSL solver. Our approach parallelized the algebraic loops which allowed Algorithm 6 to provide a speedup. On the models with fast time constants and algebraic loops, DASSL's step size advantage allowed it to complete the simulation significantly faster than our parallel approach.

We were also able to provide a speedup over Dymola's Parallel Euler solver when simulating:

1. Every model without algebraic loops using both fixed step and variable step numerical integration,

2. The regular RLC model with 1000 state variables and fast time constants using fixed step integration, and

3. Every model with algebraic loops using variable step numerical integration.

Nearly every algorithm and model in this case study was able to provide a speedup above Dymola's Parallel Euler solver; the exception is Algorithm 6 using fixed step integration. The best speedup above the Parallel Euler solver is 26, achieved using Algorithm 5 to simulate the regular RLC model with 1000 state variables. Using variable step integration we are able to provide a speedup for all of the tested models compared to the Parallel Euler solver. Besides the advantages of variable step simulation compared to fixed step simulation, one of the primary reasons for our speedup compared to Dymola's Parallel Euler solver is due to a difference in simulation approach. For our simulation we reduce the entire set of equations down to a set of state equations. Dymola does not reduce its simulation models to state equation form; it keeps all of the intermediate equations. By reducing the models down to state equations, we are able to dramatically reduce the number of calculations required to complete one time step compared to Dymola. This reduction in computational load together with our parallelization approach account for the large relative speedup that we observed.

Our algebraic loop models using fixed step integration did not perform well against Dymola's Parallel Euler solver, and only one model, regular RLC with 1000 state variables, provided a speedup. The reason for the poor performance is due to Dymola's ability to reduce the size of algebraic loops. By reducing the size of the loops Dymola significantly cuts down on its processing time per time step, and our parallelization method is not able to match the benefit gained by breaking the loops.

We would like to extend our work to apply Algorithm 6 to a solver that implements

a BDF numerical integration method (Equation IV.12). The BDF numerical integration method is used by Dymola's DASSL solver [27], and we expect that by applying our variable step parallelization algorithm to a BDF-based solver that we will be able to provide further speedups above Dymola's DASSL solver.

VI.7.3 Conclusions

This research provided a number of interesting and practical conclusions about parallel simulation. These will be detailed in the following sub-sections.

VI.7.3.1 Conclusion 1: Model Size

The first conclusion that we are able to draw from these results is that there is a model size threshold below which it is difficult to draw a benefit from parallelization. We are parallelizing within a time step, so the amount of time taken per time step is the real barrier we are trying to beat with parallelization. Our complex RLC model with 288 state variables when using parameters that created slow time constants had a time per time step of about 500ns; when using the parameters that created the fast time constants the time per time step is about 700ns. On this model we only saw a speedup when using variable step integration; the best performance for the fixed step integration was 0.56 when using the slow parameters, and 0.61 when using the fast parameters. For the large complex model the time per time step for the serial simulation is about 2μs for the slow parameters and about 3μs for the fast parameters, and the full partitioned memory method was able to produce a small speedup of about 1.2 for the slow parameters and of about 1.4 for the fast parameters. A better measurement is CPU clock cycles. On the CPU we used for our experiments, clocked to 3GHz, 700ns equates to approximately 2100 clock cycles on the CPU, while 3μs is approximately 9000 clock cycles. Since the large model produced a speedup and the small model did not, we can determine that the minimum model size above which parallelization is practical is going to be just under 800 state variables, or about 9000 clock cycles, and

Algorithm	Model	Relative Speedup			
		Thread 1	Thread 2	Thread 3	Thread 4
Algorithm 4	Complex RLC 288 State Variables, Slow Time Constants	0.24	0.12	0.12	0.12
	Complex RLC 288 State Variables, Fast Time Constants	0.003	0.002	0.002	0.002
	Complex RLC 804 State Variables, Slow Time Constants	0.58	0.45	0.58	0.66
	Complex RLC 804 State Variables, Fast Time Constants	0.78	0.72	0.95	1.13
	Regular RLC 1000 State Variables, Fast Time Constants	0.15	0.23	0.29	0.37
Algorithm 5	Complex RLC 288 State Variables, Slow Time Constants	0.11	0.19	0.24	0.31
	Complex RLC 288 State Variables, Fast Time Constants	0.002	0.003	0.004	0.005
	Complex RLC 804 State Variables, Slow Time Constants	0.40	0.73	0.98	1.27
	Complex RLC 804 State Variables, Fast Time Constants	0.27	0.52	0.73	0.95
	Regular RLC 1000 State Variables, Fast Time Constants	0.12	0.32	0.21	0.58

Table 38: Relative speedups of Algorithm 4, Minimum Sharing, and Algorithm 5 compared to Dymola's DASSL solver.

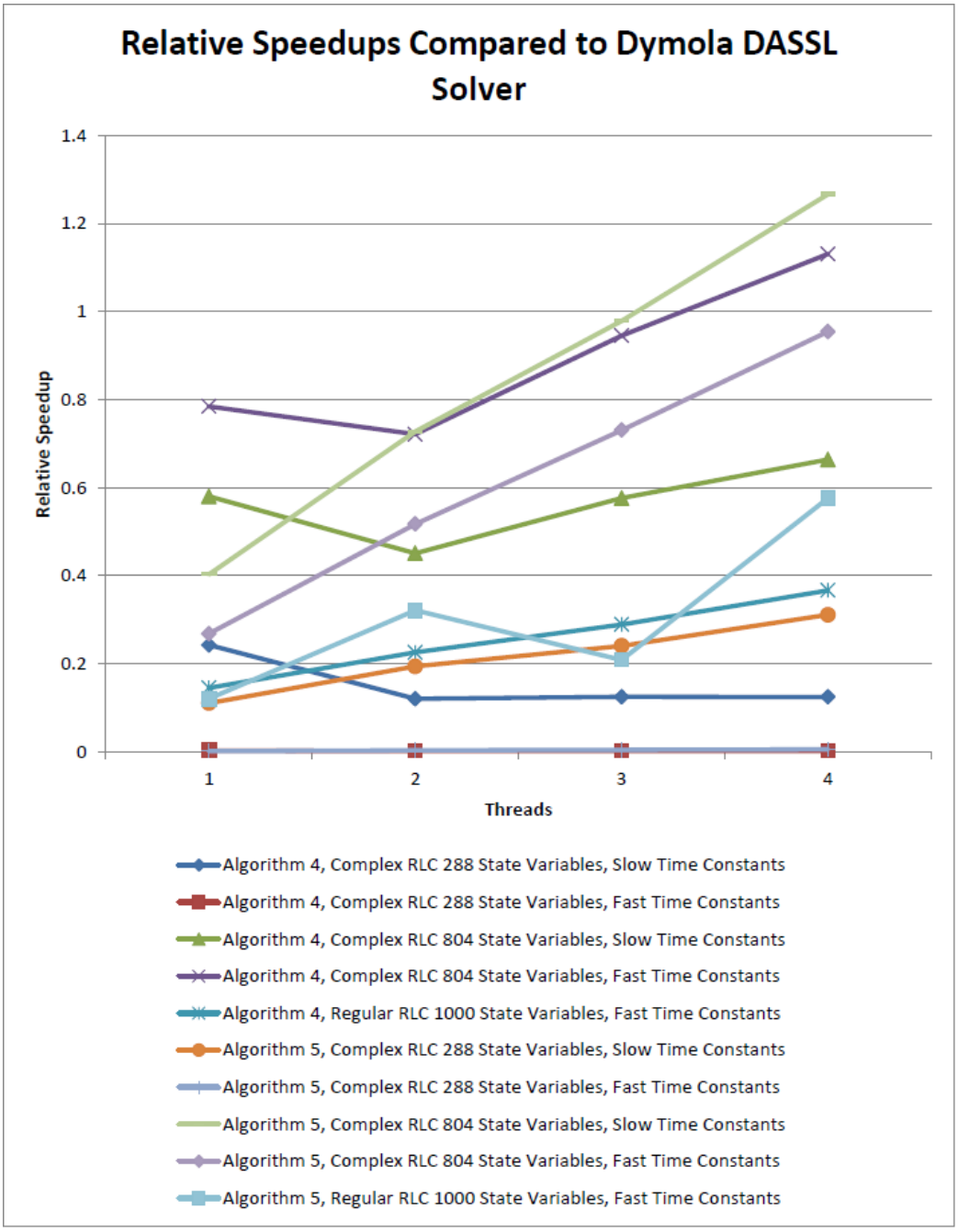


Figure 68: Relative speedups of Algorithm 4, Minimum Sharing, and Algorithm 5 compared to Dymola's DASSL solver.

Algorithm	Model	Relative Speedup			
		Thread 1	Thread 2	Thread 3	Thread 4
Algorithm 4	Complex RLC 288 State Variables, Slow Time Constants	19.25	9.54	9.90	9.89
	Complex RLC 288 State Variables, Fast Time Constants	13.74	8.51	8.95	8.90
	Complex RLC 804 State Variables, Slow Time Constants	11.43	8.89	11.36	13.08
	Complex RLC 804 State Variables, Fast Time Constants	8.07	7.42	9.72	11.63
	Regular RLC 1000 State Variables, Fast Time Constants	6.56	10.20	13.07	16.56
Algorithm 5	Complex RLC 288 State Variables, Slow Time Constants	8.78	15.39	19.09	24.69
	Complex RLC 288 State Variables, Fast Time Constants	8.71	15.37	20.02	24.77
	Complex RLC 804 State Variables, Slow Time Constants	7.94	14.34	17.28	24.95
	Complex RLC 804 State Variables, Fast Time Constants	2.76	5.32	7.51	9.81
	Regular RLC 1000 State Variables, Fast Time Constants	5.44	14.52	9.44	26.03

Table 39: Relative speedups of Algorithm 4, Minimum Sharing, and Algorithm 5 compared to Dymola's Parallel Euler solver.

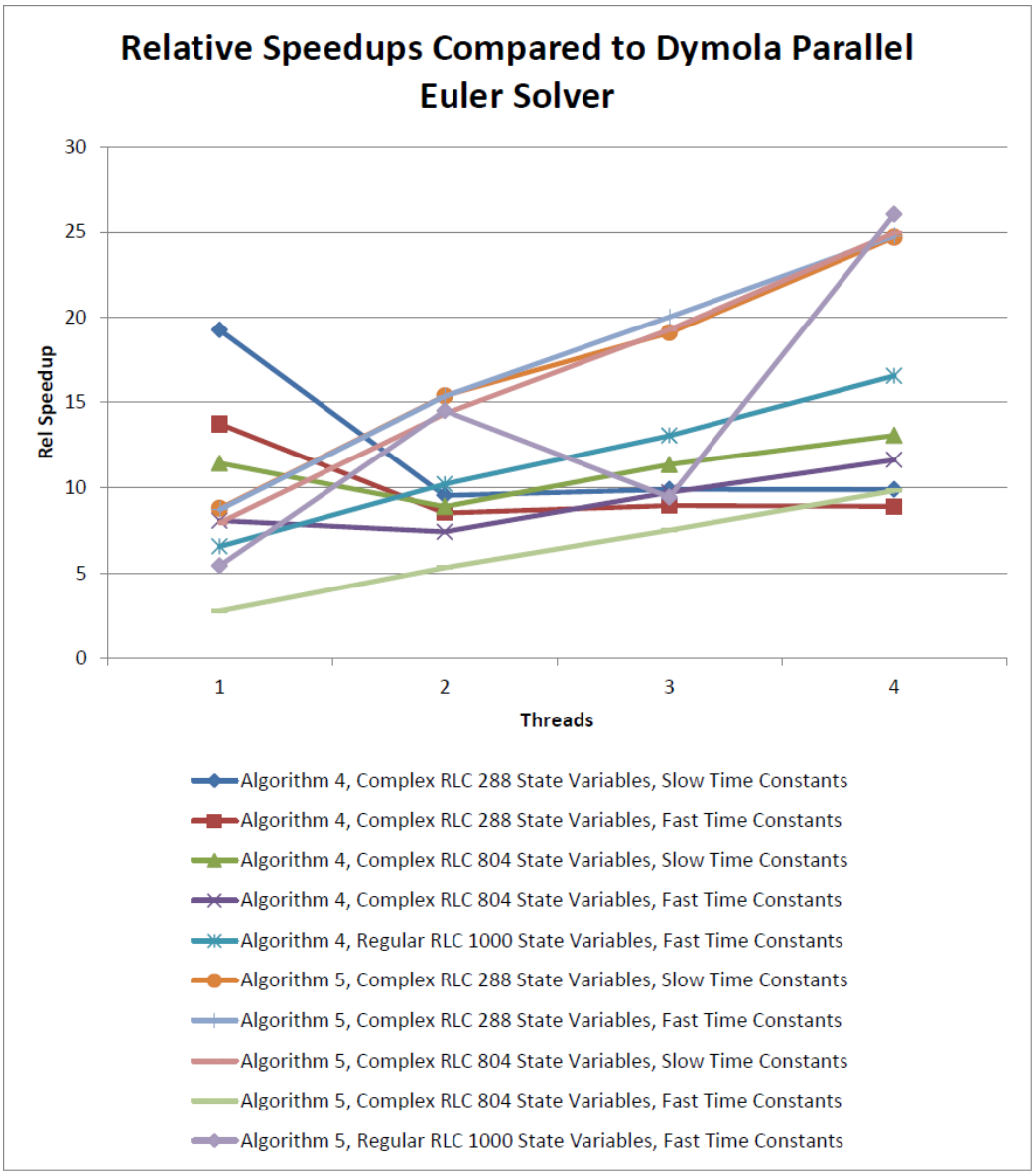


Figure 69: Relative speedups of Algorithm 4, Minimum Sharing, and Algorithm 5 compared to Dymola's Parallel Euler solver.

Algorithm	Model	Relative Speedup			
		Thread 1	Thread 2	Thread 3	Thread 4
Fixed Step	Complex RLC 288 State Variables, Slow Time Constants	0.16	0.56	0.33	0.37
	Complex RLC 288 State Variables, Fast Time Constants	0.001	0.001	0.002	0.003
	Complex RLC 804 State Variables, Slow Time Constants	0.42	0.61	0.82	0.99
	Complex RLC 804 State Variables, Fast Time Constants	0.001	0.002	0.003	0.004
	Regular RLC 1000 State Variables, Fast Time Constants	0.04	0.07	0.09	0.10
Variable Step	Complex RLC 288 State Variables, Slow Time Constants	0.68	1.23	1.58	2.06
	Complex RLC 288 State Variables, Fast Time Constants	0.04	0.06	0.09	0.12
	Complex RLC 804 State Variables, Slow Time Constants	1.11	1.91	2.68	3.48
	Complex RLC 804 State Variables, Fast Time Constants	0.03	0.06	0.07	0.09
	Regular RLC 1000 State Variables, Fast Time Constants	0.07	0.11	0.14	0.17

Table 40: Relative speedups of Algorithm 6, using variable step and fixed step solvers, compared to Dymola's DASSL solver.

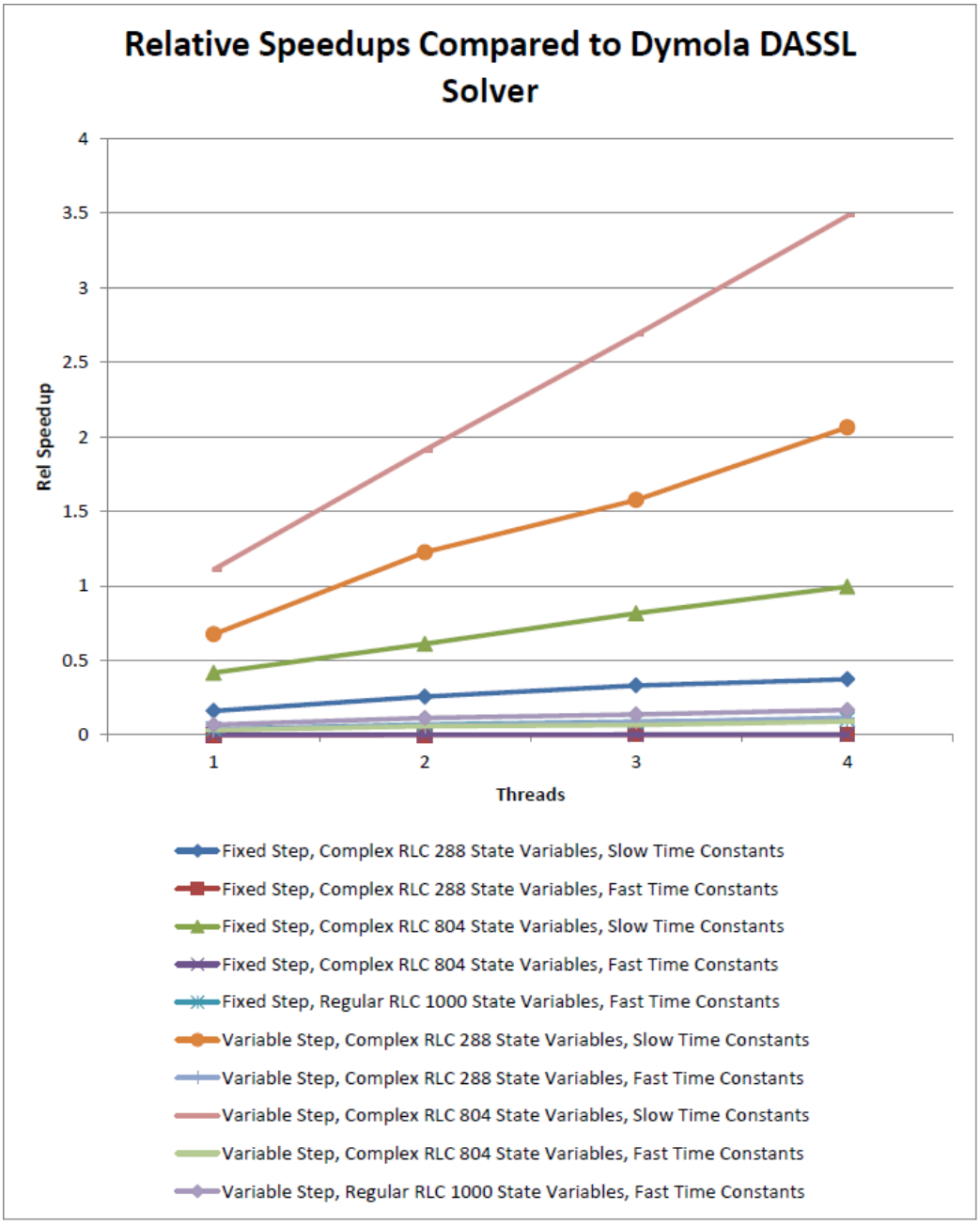


Figure 70: Relative speedups of Algorithm 6, using variable step and fixed step solvers, compared to Dymola's DASSL solver.

Algorithm	Model	Relative Speedup			
		Thread 1	Thread 2	Thread 3	Thread 4
Fixed Step	Complex RLC 288 State Variables, Slow Time Constants	0.11	0.18	0.23	0.26
	Complex RLC 288 State Variables, Fast Time Constants	0.03	0.06	0.08	0.10
	Complex RLC 804 State Variables, Slow Time Constants	0.16	0.23	0.31	0.37
	Complex RLC 804 State Variables, Fast Time Constants	0.03	0.06	0.08	0.11
	Regular RLC 1000 State Variables, Fast Time Constants	1.64	2.96	3.77	4.32
Variable Step	Complex RLC 288 State Variables, Slow Time Constants	0.47	0.86	1.10	1.44
	Complex RLC 288 State Variables, Fast Time Constants	1.48	2.60	3.48	4.73
	Complex RLC 804 State Variables, Slow Time Constants	0.42	0.72	1.01	1.30
	Complex RLC 804 State Variables, Fast Time Constants	0.84	1.58	1.77	2.40
	Regular RLC 1000 State Variables, Fast Time Constants	2.87	4.77	5.76	7.05

Table 41: Relative speedups of Algorithm 6, using variable step and fixed step solvers, compared to Dymola's Parallel Euler solver.

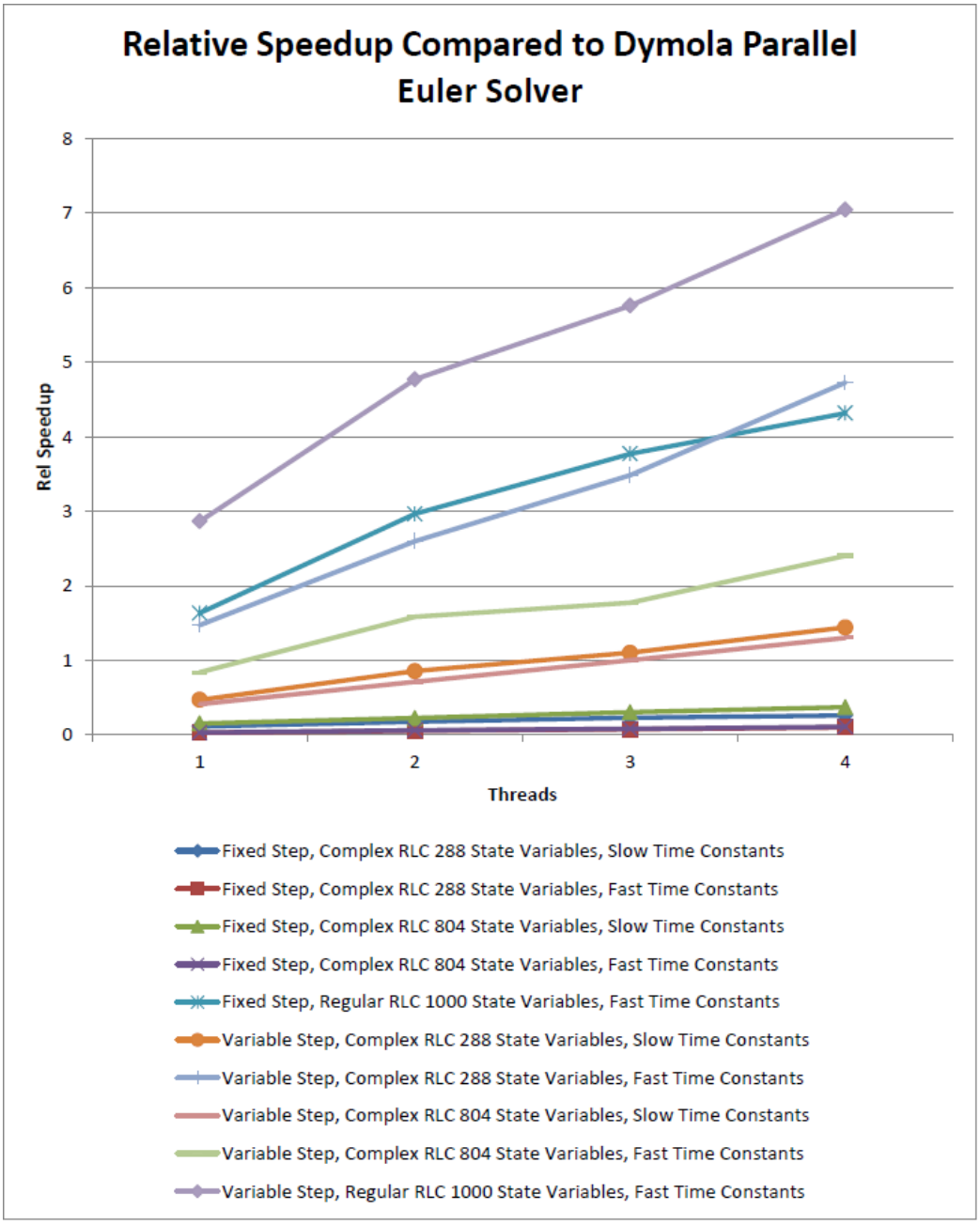


Figure 71: Relative speedups of Algorithm 6, using variable step and fixed step solvers, compared to Dymola's Parallel Euler solver.

the serial time of the computation being parallelized, the time for one time step in our case, needs to be on the order of microseconds.

VI.7.3.2 Conclusion 2: Limit Threads to Number of Cores on the CPU

In Section V.2.2 we presented an experiment that shows that cache line sharing can have a significant impact on a program's run time (Figure 29). Another feature that can be drawn from that experiment is that hardware multithreading does not produce the same benefit as having separate physical CPU cores because the time to complete the experiment begins to increase when we start using 5 threads. We are using a 4 core CPU where each core is multithreaded and can support 2 threads. When using 4 cores the OS is able to partition one thread per CPU core. However, when using 5 threads one of the cores is going to have to run 2 threads simultaneously, which degrades performance. We see this in our experiments in Section VI.5, where there is a drop off in performance in moving from 4 threads to 5 threads. For most experiments the best performance was using 4 threads. The conclusion that we can draw from this data is that for simulation purposes hardware multithreading (see Definition 19), which is the technology used to support more than 1 thread on a single CPU core, produces worse performance than using the same number of threads as there are cores on the CPU, and it should be avoided. A possible reason for the lack of performance of hardware multithreading is due to both threads on the core having to share an L1 cache, and therefore the amount of cache available for each thread is reduced by half compared to the single threaded case. A second reason is that simulation is very processor intensive activity, and the hardware multithreading logic built into the CPU is not able to adaquately partition the computational work of 2 threads onto the single processor built into the CPU core.

VI.7.3.3 Conclusion 3: Design Memory Management to Support the Needs of the CPU Cache

Another conclusion that we can draw from Section V.2.2 and from our experiment results in Section VI.5 is that minimizing cache line reads and avoiding cache line sharing is crucial to parallel program performance. These two factors are what prevented the parallel simulation methods described in Sections VI.5.2 and VI.5.3 from providing very good performance.

In the full shared memory approaches there was both cache line sharing between the threads of $\mathbf{T}_{spawn}$ and too many cache line reads for thread T_{main}. The problem with the cache line reads of T_{main} is that T_{main} was simply merging the variables of $\mathbf{x}_{n+1}$ into the variables of $\mathbf{x}_n$. It was not performing any calculation work on the variables, and therefore as demonstrated in Chapter V in Figure 27, the time to perform the cache line reads dominated the time to perform the merge. The benefit of assigning each thread in $\mathbf{T}$ to perform its own merge step, as was done in the full partitioned methods, Sections VI.5.4 and VI.5.5, is that the variables to perform the merge are already in the cache for the thread and there is no need to read a large number of cache lines to perform the merge as required by the full shared memory and partial partitioned memory methods.

Another factor in the good performance of these methods is that the memory block that each thread in $\mathbf{T}$ wrote to did not share cache lines with any other block. We were able to guarantee this through the use of the `alignas` keyword, introduced as a part of the C++11 language standard. Each memory block used the `alignas` keyword to align the block of memory to a cache line boundary, by doing this no part of the memory block was on the same cache line as another block.

VI.7.3.4 Conclusion 4: Division of Labor

Another important aspect to designing parallel simulation algorithms is to ensure that the computational work of a time step is divided evenly across the computational threads.

This was a second major problem with the parallel simulation methods described in Sections VI.5.2 and VI.5.3. These methods assigned the computational work of solving and integrating the equations in $\mathbf{f}_{IFE}$ to the threads in $\mathbf{T}_{spawn}$ while the thread T_{main} was only responsible for merging $\mathbf{x}_{n+1}$ into $\mathbf{x}_n$ and for advancing the simulation time. Not only did this setup contribute to problems of too many cache line reads with too little computational work in T_{main} but it also meant that only thread T_{main} or threads $\mathbf{T}_{spawn}$ were active at any given time. There was a distinct back and forth flow between T_{main} and $\mathbf{T}_{spawn}$. This is not a good use of computational resources, as the time the threads spend waiting is time wasted. The full partitioned methods, Sections VI.5.4 and VI.5.5, and the parallel loop method, Section VI.5.6, solve this problem by including T_{main} in the computation of $\mathbf{f}_{IFE}$ and by including $\mathbf{T}_{spawn}$ in the merge process. By expanding the roles of both sets of threads we were able to avoid wasted time during the simulation.

VI.7.3.5 Conclusion 5: Software Design

Simulations for a long simulation time will likely require a large number of time steps. For our experiments, some models were were run for 5 million time steps. At that number of iterations small inefficiencies in the simulation code, that would have been ignored or undetectable had they only been run once, can become a source of significant lost time. Section VI.6 described the features of our simulation code that we discovered were slowing down our simulation. In a program such as a simulation where the same piece of code is repeated many times, every line of code in that program needs to be closely examined to make sure it is as efficient as possible.

VI.8 Summary

This chapter presented details and results using a case study approach to study the effectiveness of 6 different parallel algorithms that can be used to parallelize the simulation of a system of Ordinary Differential Equations. Section VI.1 described the models we used

to evaluate the performance of our parallel algorithms. We used two complex RLC models, one with 288 state variables and one with 804 state variables, and one regular RLC model with 1000 state variables. The complex RLC models were tested with two sets of parameter values, a slow set and a fast set, and the regular RLC model was tested with only the fast set. We also modified these models so that they contained algebraic loops, and tested them on a parallel approach designed for models with algebraic loops. Section VI.2 described fixed step and variable step integration algorithms, and the pre-processing code we developed to evaluate our parallel simulation algorithms. Section VI.3 described the mathematical basis for partitioning a system of ODEs into independent subsets. Section VI.4 describes our base serial test case to which we compared our non-algebraic loop parallel algorithms. Section VI.5 described our parallel simulation algorithms, and a summary of each is presented below and in Table 37. Section VI.6 presents our investigations into optimizing software for high performance, and Section VI.7 presented our results.

The first parallel approach, described and analyzed in Section VI.5.1, created one computation thread for each state equation in the model. This approach did not perform well as it involved too much overhead per thread relative to the amount of computational work assigned to each thread. The second approach, described in Section VI.5.2, agglomerated the state variables into subsets in an effort reduce the overhead created by using 1 thread per state variable. This approach performed better than the first approach, but it still did not produce a speedup over the serial case. The primary cause of the lack of speedup was attributed to cache line sharing between the different threads. The third approach, described in Section VI.5.3, tried to avoid this cache line sharing by having each thread in $\mathbf{T}_{spawn}$ write to its own cache aligned data structure. Unfortunately, this approach also did not provide a speedup above the serial case, and the likely reason is that by having each thread in $\mathbf{T}_{spawn}$ write to its own memory block, we added work to thread T_{main} when it was time for T_{main} to perform the synchronization.

The fourth approach, described and analyzed in Section VI.5.4, presented two full partitioned memory approaches that used a fixed step integration, and it adjusted the roles of the threads in $\mathbf{T}_{spawn}$ and T_{main}. In the full partitioned memory approach each thread was assigned a subset of the equations in $\mathbf{f}_{IFE}$ to solve, and each thread performed the synchronization step on the state variables for which it was responsible. This distributed the work of performing the simulation evenly across all threads. Each thread was also assigned a cache aligned block of memory to write to, which avoided cache line sharing problems. The fixed step full partitioned memory approach provided a speedup of 1.44 when using 4 threads on the complex RLC model with 804 state variables, and a speedup of 2.5 when using 4 threads on the regular RLC model with 1000 state variables.

The fifth parallel approach, described and analyzed in Section VI.5.5, used the memory partitioning ideas of approach 4, and applied them to a variable step solver instead of a fixed step solver. The variable step integration provided better speedups than the fixed step approach, with speedups of 2.2 when using 2 threads, and 3.95 when using 4 threads on the regular RLC model with 1000 state variables. On the complex model with 804 state variables it provided a speedup of 2.3 when using 4 threads. This approach provided a greater speedup than the previous approach because each partition of the system was able to dictate its own step size allowing the simulation to progress more efficiently than requiring each partition to use the same time step.

The final parallel approach, described and analyzed in Section VI.5.6, applied parallel simulation to models with algebraic loops. In this approach only the algebraic loops were parallelized, and the numerical integration was kept single threaded. This method also provided good performance. It produced a speedup of 3.32 when using 4 threads on the complex RLC model with 288 state variables, and a speedup of 3.49 when using 8 threads on the complex RLC model with 804 state variables. This method provided a speeup because the algebraic loops were dominating the run-time of the simulation, so by parallelizing the

algebraic loops we were able to reduce their computation time and therefore to reduce the overall time of the simulation.

This chapter also looked at a model of the Water Recovery System at NASA Ames. The WRS has 4 algebraic loops and 24 state variables. Our parallel simulation of the WRS produced a speedup of 1.23 for fixed step simulation, and 1.11 for variable step simulation. The performance of this model was held back because one of the loops in the model was very large, more than double the size of the other loops, and the large loop so dominated the simulation run-time that parallelizing the other loops did not provide an advantage.

The next chapter will present an overall summary of our work, our results and lessons learned, and discuss the future work we have planned.

CHAPTER VII

DISCUSSIONS, CONCLUSIONS, AND FUTURE WORK

This chapter presents a discussion and conclusions based on the material presented in the previous chapters, and outlines our future work. Section VII.1 presents a brief discussion of the results and lessons learned presented in the previous chapter. Sections VII.2 presents a summary of our conclusions based on this work. Section VII.3 described the next steps and future work for this research. Section VII.4 briefly outlines what we would like to see in a next generation parallel CPU architecture based on what we learned performing this research.

VII.1 Discussion

Multi-core CPU architectures provide unique challenges that are not present in other parallel architectures. In a multi-core CPU, the processor cache and the number of CPU cores, need to be considered when designing the parallel program. The needs of both often conflict, and finding the balance between them is the key aspect of our research. We are able to balance these two needs by developing parallel algorithms that meet the following criteria:

- Create the same, or fewer, threads as there are CPUs available to the OS,
- Partition the memory structure of the parallel algorithm so that each thread only writes to one memory block, and
- Evenly distribute the work of the simulation across all threads.

Creating the same number of threads as there are CPUs available to the OS avoids the OS having to swap threads onto and off of a core. Assigning a unique memory block to each thread avoids problems of cache line sharing (Section V.2.2). Distributing the work evenly

across all threads avoids unnecessary cache line reads by a controlling thread, and keeps all of the processors used for the simulation busy.

Another key facet of our algorithms is to avoid the communication problems that the Modelica parallel simulation approaches presented in Section IV.4 tried to solve. Those methods parallelize a large DAG of equations describing the model behavior, which results in increased communication between the computational threads during each time step. We avoid this problem by reducing the ODE model to state equations, which eliminates dependencies between equations that adds complications to the parallelization algorithm. We also maintain our computational threads for the duration of the simulation, which avoids another source of overhead presented in [46].

The parallel simulation algorithms that gave the best performance are Full Partitioned Memory, Fixed Step algorithm (Section VI.5.4) and Full Partitioned Memory, Variable Step algorithm (Section VI.5.5). These methods use an individual block of memory for each thread, and all of the threads participated in all of the simulation computations. The Full Partitioned Memory, Fixed Step produced a maximum speedup of 2.53, and the Full Partitioned Memory, Variable Step produced a maximum speedup of 5.23 on our Intel i7-880 CPU with 4 cores that each support 2 threads through hardware multithreading. The Algebraic Loop Simulation (Section VI.5.6) also used a separate block of memory for each thread, and all threads contributed to solving the algebraic loops in the models. However, the numerical integration step for these models was not parallelized. The Algebraic Loop simulation produced a maximum speedup of 3.33.

When run on an appropriate architecture, such as what we describe in Section VII.4, and on large models, we expect these algorithms to scale up and to continue to provide larger speedups as more CPU cores are added to the simulation. To achieve further speedups the additional CPU cores need to be physical CPU cores, and not cores added as a result of hardware multithreading, because in our experiments we did not see a benefit when hardware multithreading was used. The models also need to be large enough such that the

computation involved in each time step must be sufficient to offset the additional overhead required to maintain the synchronization between threads at the end of every time step. GPUs seem to be a near ideal architecture for parallel simulation of ODEs, and applying our parallel simulation algorithms to a GPU is a direction we would like to pursue in future work.

VII.2 Conclusions

In this work, we presented a series of parallel simulation algorithms for ODEs that are specifically designed to accommodate the features of a modern multi-core CPU. Specifically, the algorithms consider the multiple CPU cores, the processor cache, and their interactions to derive maximum speedup. The final three of these algorithms, Full Partitioned Memory Fixed Step (Section VI.5.4), Full Partitioned Memory Variable Step (Section VI.5.5), and Algebraic Loop Simulation (Section VI.5.6), produced good speedup when compared to out serial test case. We also developed a series of experimental studies that allowed us to analyze the effectiveness of various multi-threading and memory management schemes for parallel simulations. Since these algorithms are based on an ODE representation of the model behavior, they can be applied equally across models that cover multiple physical domains.

In the process of systematically developing these algorithms we derived a set of recommendations that apply to parallel simulation (Section VI.7.3) on modern multi-core processors. These conclusions include recommendations on the size of the model to be parallelized, the number of threads to use for the simulation, the memory management scheme to use, how to divide the computational work between the threads, and software optimizations to implement.

An important limitation of this work is that the models to be simulated must be reduced to ODE form. This results in a loss of the time trajectory data of the algebraic variables in the system, but the trajectory data for the state variables is maintained. The time trajectory

data of these algebraic variables is typically preserved in traditional Modelica simulation [27]; however, by reducing the model to ODE form we save computation time during the simulation. Another limitation is that our parallel algorithms are specifically designed for the parallel architecture of a multicore CPU. If these algorithms are applied, unaltered, to a different parallel architecture, such as a SIMD architecture or a to a GPU, the algorithms will likely not perform as well as they did in our experiments. A third limitation is that we only address a small subset of the Modelica language; more complicated Modelica models that include discrete mode transitions and conditional behavior cannot be parallelized with these algorithms.

For future work, we would like to develop methods for formally solving the optimization problem of the model equations, expand our parallelization algorithms to support an additional numerical integration method, apply our algorithms to a General Purpose GPU, parallelize the integration of our algebraic loop simulations, and apply intelligent load balancing when assigning algebraic loops to threads. The next section discusses some of these approaches in more detail as future work.

VII.3 Future Work

There are a number of promising directions to pursue to further extend the results obtained in this research. These directions include: formally optimizing the partitioning of the model equations, parallelizing an additional variable step numerical integration method, extending our algorithms to a General Purpose GPU, further parallelize the simulation of our algebraic loop simulations, and to apply intelligent load balancing when assigning algebraic loops to parallel threads.

The first direction of research we would like to pursue in our future work is to optimize the partitioning of a model to minimize the data sharing that needs to happen across parallel

threads. The optimization function can be defined as

$$\min(K)|K = K_{1\_2} + K_{1\_3} + \cdots + K_{1\_\ell} + K_{2\_1} + K_{2\_3} + \cdots + K_{2\_\ell} + K_{\ell\_1} + K_{\ell\_2} + K_{\ell\_(\ell-1)}, \tag{VII.1}$$

where K_{1_2} are the state variables that are used by thread 1 but solved by thread 2, K_{2_1} are the state variables that are used by thread 2 but solved by thread 1, and so on. The optimization problem is to partition the state variables such that the sum of the individual values of K is as small as possible, as this represents the smallest amount of sharing that needs to happen between the partitions, and therefore will lead to the smallest amount of cache line reads per time step for each individual thread.

The optimization formulation in Equation VII.1 is only a first step in optimizing the parallel partitioning. It does not take into account that the cores on a processor communicate through cache lines. In the processor architecture we are using 8 `doubles` fit into one cache line, and this means that it is just as much work for a thread to read 1 variable from a different thread as it is to read 8 cache aligned variables from a different thread. Not taking this into account is partially responsible for the poor performance of the partial partitioned memory algorithm described in Section VI.5.3. In order to truly optimize the partitioning of a model across multiple threads, the optimization needs to consider communication through cache lines, and it needs to dictate how the memory is created in each thread. Without the ability to dictate how memory is created in each thread, the optimization process may assign one thread to read 8 `doubles` from a different thread, and, in order for that read to be efficient, the thread supplying those variables must align all of them so that all 8 variables fit on one cache line.

The second direction of research we would like to pursue is to add support for a variable step integration method that supports Backward Differential Formula (BDF) integration, described in Equation IV.12, such as the IDA solver from the SUNDIALS [20] solver library. BDF integration is used by Dymola's default solver, DASSL, and the integration

method is very efficient and stable, and was able to out perform our parallel methods. By applying our parallel methods, especially the methods described in Section VI.5.5, to a variable step BDF solver we would be able to compare directly against Dymola's default to see if our parallel algorithms are able to provide a speedup against a very efficient serial application.

A third area of research we would like to pursue in the future is to apply our parallel methods to a General Purpose GPU, Section V.1.2. A GPGPU, is designed to support a very large number of parallel threads, very efficient floating point calculations, and has very natural thread synchronization built into the processor [79]. These design decisions built into modern GPGPUs seem tailor made to support the needs of parallel simulation, and we would like to apply our parallel simulation methods to such purpose built hardware.

A fourth area of research we would like to pursue is to parallelize the integration step of our algebraic loop simulations. We designed the current algorithm based on a worst case assumption that every state equation is dependent on the output from every algebraic loop. In a real physical model only a small portion of the state variables are going to be dependent on the output of the algebraic loops. Therefore, we can apply a smart partitioning algorithm that partitions the system such that the algebraic loops and their dependent state variables are grouped together. This will avoid the problem of duplicating work by needing to solve all loops in each state variable partition described in Section VI.5.6.

A fifth area of research is to apply load balancing to our partitioning of the algebraic loops. There can be a drastic difference in the amount of time it takes to solve algebraic loops, due to solving the loops having a computational complexity of $O(n^3)$, where n is the number of equations in the loop. If the size of the loop is considered when assigning the loops to threads, then the simulations will more likely be able to avoid the problem of a large loop dominating the simulation run time.

VII.4 Future Processor Architecture

Based on our conclusions above, and the fact that the processor architecture played such a large role in the performance of our simulations, a possible research direction is to theoretically design a processor architecture to fit the precise needs of parallel simulation. Our new architecture would simplify the architecture of modern multicore CPUs. First it would remove hardware multithreading and instead create as many physical cores as possible. In our experiments hardware multithreading did not provide a computational benefit, and the simulation performance always dropped when the CPU had to start using hardware multithreading. However, we did see performance increase when adding threads up to the number of physical cores on the CPU. Based on these observations we would remove hardware multithreading and instead add a multitude of physical CPU cores.

A second feature of our theoretical processor architecture would be to eliminate the cache hierarchy, and, as much as possible, remove the cache partitioning between threads. The cache hierarchy makes it difficult to share memory and data between threads. In current processor architectures, in order for a write to a variable by one thread to be seen by a second thread, that variable must be written to the L1 cache of the writing core, to the L2 cache, to the shared L3 cache, then back down to the L2 and L1 caches of the reading core. Moving data between layers of cache takes time that could otherwise have been spent in useful computation. A completely flat cache structure that is shared across all cores with each core able to see the entire on-chip memory space, essentially a CPU with only a single L1 cache equally accessible by all cores, would eliminate this problem and provide for much faster on-chip memory. The processor cache hierarchy was originally created to give the illusion of a single large block of memory at a reasonable cost [88], by including fast and expensive memory as the L1 cache, and slower, less expensive, memory as the L2 and L3 caches. Implementing a single shared L1 cache across all cores would likely increase the price of a processor, but the performance benefit would be significant.

In our research we did not investigate the power consumption of the CPU, but according to [61] it takes approximately 4200 picojoules to move 64 bits from DRAM to the CPU registers, and 60% of that energy usage is consumed by the on-chip cache hierarchy. Contrast that power usage to the fact that it takes approximately 100 picojoules to perform a double precision floating point operation. Moving the data consumes more than 40 times the amount of energy than simply operating on that data. By removing the processor cache hierarchy we would save a significant amount of energy, in addition to making data exchange between cores more simple.

Finally, our theoretical CPU architecture we would eliminate the cache line as the means of communication between CPU cores. It would not be necessary in a completely flat on-chip memory structure, and using cache lines requires each CPU core to do more work than is needed to pull a single variable down into the L1 cache.

Many of these ideas are not unique to simulation, and others in the CPU hardware industry are noting the lack of efficiency working with a cache memory hierarchy and the limitations imposed by hardware multithreading. At the time of this writing a young company called Rex Computing [17] is currently designing a new CPU architecture, called Neo, that nearly perfectly aligns with our proposed theoretical architecture. Their architecture is still in development, so full details have not been released, but highlights can be seen on their website and at these sources: [99], [61], [19]. There is no cache hierarchy, and each core has its own local cache that is accessible by every other core on the CPU. Their design promises to be very fast and very energy efficient. If their design becomes a commercial reality, it would make an ideal architecture on which to base a high performance commercial parallel simulation tool.

APPENDIX A

LIST OF PUBLICATIONS

Our research has lead to the following conference publications.

A.1 Refereed Conference Publications

C-1 **Joshua D. Carl**, Gautam Biswas, Sandeep Neema, and Ted Bapty. "An Approach to Parallelizing the Simulation of Complicated Modelica Models." SCS Summer Simulation Multi-Conference. 2014.

C-2 **Joshua D. Carl**, Zsolt Lattmann, and Gautam Biswas. "Modeling and Simulation Semantics for Building Large-Scale Multi-Domain Embedded Systems." 27th European Conference on Modelling and Simulation (ECMS 2013), Ålesund, Norway. 2013.

C-3 **Joshua D. Carl**, Daniel L. C. Mack, Ashraf Tantawy, Gautam Biswas, and Xenofon Koutsoukos. "Fault Isolation for Spacecraft Systems: An Application to a Power Distribution Testbed." 8th IFAC SafeProcess: Fault Detection, Supervision and Safety of Technical Processes. Vol. 8. No. 1. 2012.

C-4 **Joshua D. Carl**, Ashraf Tantawy, Gautam Biswas, and Xenofon Koutsoukos. "Detection and estimation of multiple fault profiles using generalized likelihood ratio tests: A case study." 16th IFAC Sysid (2012).

REFERENCES

[1] Adaptive vehicle make. http://www.darpa.mil/Our_Work/TTO/Programs/Adaptive_Vehicle_Make__%28AVM%29.aspx.

[2] C - approved standards. http://www.open-std.org/jtc1/sc22/wg14/www/standards.

[3] The c++ programming language. https://isocpp.org/.

[4] Dymola, dynasim ab. http://www.dymola.com.

[5] Generic modeling environment. http://www.isis.vanderbilt.edu/Projects/gme/.

[6] Hadoop. http://hadoop.apache.org/.

[7] Intel. http://www.intel.com.

[8] Intel threading building blocks. https://www.threadingbuildingblocks.org/.

[9] Intel vtune amplifier xe 2015. https://software.intel.com/en-us/intel-vtune-amplifier-xe.

[10] Jmodelica. http://www.jmodelica.org/.

[11] Lawrence livermore national laboratory. https://www.llnl.gov/.

[12] Maplesim, maplesoft. http://www.maplesoft.com/products/maplesim/.

[13] Matlab, the mathworks, inc. http://www.mathworks.com/.

[14] Modelica. https://www.modelica.org/.

[15] Openmodelica. http://www.openmodelica.org.

[16] Python programming language. https://www.python.org.

[17] Rex computing. http://www.rexcomputing.com/index.html.

[18] Simulationx. http://www.simulationx.com/.

[19] Slashdot discussion. http://news.slashdot.org/story/15/07/22/222210/19-year-olds-supercomputer-chip-startup-gets-darpa-contract-funding.

[20] Sundials. http://computation.llnl.gov/casc/sundials/main.html.

[21] Sympy. http://sympy.org/en/index.html.

[22] AMD. http://www.amd.com.

[23] CUDA. http://www.nvidia.com/object/cuda_home_new.html.

[24] GCC, the gnu compiler collection. https://gcc.gnu.org/.

[25] GNU scientific library. http://www.gnu.org/software/gsl/.

[26] Valgrind. http://valgrind.org/.

[27] D. S. AB. Dymola user manual volume 1. Technical report, Dassault Systems AB.

[28] D. S. AB. Dymola user manual volume 2. Technical report, Dassault Systems AB.

[29] T. Anderson. The performance of spin lock alternatives for shared-money multiprocessors. *Parallel and Distributed Systems, IEEE Transactions on*, 1(1):6–16, Jan 1990.

[30] P. Aronsson. PhD thesis.

[31] J. D. Carl, G. Biswas, S. Neema, and T. Bapty. An approach to parallelizing the simulation of complicated modelica models. In *2014 Summer Simulation Multi-Conference*, Monterey, California, July 2014.

[32] F. Casella. A strategy for parallel simulation of declarative object-oriented models of generalized physical networks. In H. Nilsson, editor, *Proceedings of the 5th International Workshop on Equation-Based Object-Oriented Modeling Languages and Tools*. Linköping University Electronic Press, April 2013.

[33] F. E. Cellier and E. Kofman. *Continuous system simulation*. Springer, 2006. Chapter 7, Pages 253-318.

[34] F. E. Cellier and E. Kofman. *Continuous system simulation*. Springer, 2006. Chapter 8, Pages 319-396.

[35] F. E. Cellier and E. Kofman. *Continuous system simulation*. Springer, 2006. Chapter 2, Pages 25-56.

[36] F. E. Cellier and E. Kofman. *Continuous system simulation*. Springer, 2006.

[37] F. E. Cellier and E. Kofman. *Continuous system simulation*. Springer, 2006. Chapter 4, Pages 117-164.

[38] F. E. Cellier and E. Kofman. *Continuous system simulation*. Springer, 2006. Chapter 3, Pages 57-116.

[39] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*.

McGraw-Hill Book Company, 1998.

[40] C. F. Curtiss and J. O. Hirschfelder. Integration of stiff equations. *Proceedings of the National Academy of Sciences of the United States of America*, 38:235–243.

[41] A. Danowitz, K. Kelley, J. Mao, J. P. Stevenson, and M. Horowitz. Cpu db: Recording microprocessor history. *Commun. ACM*, 55(4):55–63, April 2012.

[42] A. L. Dulmage and N. S. Mendelsohn. Two algorithms for bipartite graphs. *Journal of the Society for Industrial and Applied Mathematics*, 11(1):pp. 183–194, 1963.

[43] G. B. D. K. E.-J. Manders, S. E. Bell. Multi-scale modeling of advanced life support systems. In *Proc. of the 35th Intl. Conf. on Environmental Systems*, Rome, Italy, July 2005.

[44] J. Edmonds and R. M. Karp. Theoretical improvements in algorithmic efficiency for network flow problems. *J. ACM*, 19(2):248–264, Apr. 1972.

[45] H. Elmqvist, S. E. Mattsson, and H. Olsson. New methods for hardware-in-the-loop simulation of stiff models. 2002.

[46] H. Elmqvist, S. E. Mattsson, and H. Olsson. Parallel model execution on many cores. In *Proceedings of the 10th International Modelica Conference*, Lund, Sweden, March 2014. Modelica.org.

[47] H. Elmqvist, M. Otter, and F. E. Cellier. Inline integration: A new mixed symbolic/numeric approach for solving differential-algebraic equation systems. In *In Proceedings of ESM 95, European Simulation Multiconference*, 1995.

[48] B. H. K. et al. Cyber-physical systems executive summary. Technical report, CPS Steering Group, March 2008.

[49] E. Fehlberg. Classical fifth-, sixth-, seventh-, and eighth-order runge-kutta formulas. Technical Report TR R-287, NASA Johnson Space Center, Houston, Texas, 1968.

[50] M. Flynn. Some computer organizations and their effectiveness. *Computers, IEEE Transactions on*, C-21(9):948–960, Sept 1972.

[51] D. R. Ford and D. R. Fulkerson. *Flows in Networks*. Princeton University Press, Princeton, NJ, USA, 2010.

[52] I. Foster. *Designing and Building Parallel Programs: Concepts and Tools for Parallel Software Engineering*. Addison-Wesley, 1995. Chapter 2, Pages 27-82.

[53] I. Foster. *Designing and Building Parallel Programs: Concepts and Tools for Parallel Software Engineering*. Addison-Wesley, 1995.

[54] D. G. M. Frank McSherry, Michael Isard. Scalability! but at what COST? In *15th Workshop on Hot Topics in Operating Systems (HotOS XV)*, 2015.

[55] J. Frenkel, G. Kunze, and P. Fritzson. Survey of appropriate matching algorithms for large scale systems of differential algebraic equations. In *Modelica 2012, the 9th International Modelica Conference*, 2012.

[56] J. Frenkel, G. Kunze, P. Fritzson, M. Sjólund, A. Pop, and W. Braun. Towards a modular and accessible Modelica compiler backend. In C. Claub, editor, *Proceedings of the 8th International Modelica Conference*. Linköping University Electronic Press, March 2011.

[57] P. Fritzson. *Principles of object-oriented modeling and simulation with Modelica 2.1*. John Wiley & Sons, 2010.

[58] C. Gear and D. Wells. Multirate linear multistep methods. *BIT Numerical Mathematics*, 24(4):484–502, 1984.

[59] M. Gebremedhin. Parmodelica : Extending the algorithmic subset of modelica with explicit parallel language constructs for multi-core simulation. Master's thesis, Linköping University, Department of Computer and Information Science, 2011.

[60] E. Hairer and G. Wanner. *Solving Ordinary Differential Equations II: Stiff and Differential-Algebraic Problems*. 1991.

[61] N. Hemsoth. Supercomputer chip startup scores funding, darpa contract. http://www.theplatform.net/2015/07/22/supercomputer-chip-startup-scores-funding-darpa-contract/.

[62] R. M. Howe. Real-time multirate synchronous simulation with single and multiple processors, 1998.

[63] D. C. Karnopp, D. L. Margolis, and R. C. Rosenberg. *System Dynamics*. John Wiley & Sons, Inc., Hoboken, New Jersey, 4 edition, 2006.

[64] G. Karsai and J. Sztipanovits. Model-integrated development of cyber-physical systems. In U. Brinkschulte, T. Givargis, and S. Russo, editors, *Software Technologies for Embedded and Ubiquitous Systems*, volume 5287 of *Lecture Notes in Computer Science*, pages 46–54. Springer Berlin Heidelberg, 2008.

[65] G. Karsai, J. Sztipanovits, A. Ledeczi, and T. Bapty. Model-integrated development of embedded software. *Proceedings of the IEEE*, 91(1):145–164, January 2003.

[66] J. D. Lambert. *Numerical methods for ordinary differential systems: the initial value problem*. John Wiley & Sons, Inc., 1991. Chapter 1, Pages 1-20.

[67] J. D. Lambert. *Numerical methods for ordinary differential systems: the initial value*

problem. John Wiley & Sons, Inc., 1991. Chapter 2, Pages 21-44.

[68] J. D. Lambert. *Numerical methods for ordinary differential systems: the initial value problem*. John Wiley & Sons, Inc., 1991.

[69] J. D. Lambert. *Numerical methods for ordinary differential systems: the initial value problem*. John Wiley & Sons, Inc., 1991. Chapter 5, Pages 149-212.

[70] J. D. Lambert. *Numerical methods for ordinary differential systems: the initial value problem*. John Wiley & Sons, Inc., 1991. Chapter 3, Pages 45-102.

[71] J. D. Lambert. *Numerical methods for ordinary differential systems: the initial value problem*. John Wiley & Sons, Inc., 1991. Chapter 6, Pages 213-260.

[72] Z. Lattmann. RLC models, 2014. Personal files.

[73] Z. Lattmann, A. Nagel, J. Scott, K. Smyth, J. Ceisel, C. vanBuskirk, J. Porter, S. Neema, T. Bapty, D. Mavris, and J. Szipanovits. Towards automated evaluation of vehicle dynamics in system-level designs. In *Proc. ASME International Design Engineering Technical Conf. & Computers and Information in Engineering Conf. (IDETC/CIE 2012)*, Chicago, IL, USA, 08/2012 2012.

[74] E. Lee. Cyber physical systems: Design challenges. In *Object Oriented Real-Time Distributed Computing (ISORC), 2008 11th IEEE International Symposium on*, pages 363–369, May 2008.

[75] H. Lundvall and P. Fritzson. Automatic parallelization of object oriented models across method and system. In *Proceedings of the 6th Eurosim Congress*, 2007.

[76] H. Lundvall, K. Stavåker, P. Fritzson, and C. Kessler. Automatic parallelization of simulation code for equation-based models with software pipelining and measurements on three platforms. *SIGARCH Comput. Archit. News*, 36(5):46–55, 2009.

[77] P. Meijer. Tearing systems of differential algebraic equations, December 2011.

[78] S. Meyers. CPU caches and why you care, April 2011. http://www.aristeia.com/TalkNotes/ACCU2011_CPUCaches.pdf.

[79] J. Nickolls, I. Buck, M. Garland, and K. Skadron. Scalable parallel programming. *ACM Queue*, March 2008.

[80] K. Nyström and P. Fritzson. Parallel simulation with transmission lines in modelica. In *Proceedings of the 5th International Modelica Conference (Modelica'2006)*, 2006.

[81] L. Ochel. Openmodelica: back end flowchart. Documentation in OM source code SVN., September 2013.

[82] P. Ostlund, K. Stavaker, and P. Fritzson. Parallel simulation of equation-based models on CUDA-enabled GPUS. In *Proceedings of the 9th Workshop on Parallel/High-Performance Object-Oriented Scientific Computing*, POOSC '10, pages 5:1–5:6, New York, NY, USA, 2010. ACM.

[83] I. Ostrovsky. Gallery of processor cache effects. Blog, January 2010. http://igoro.com/archive/gallery-of-processor-cache-effects/.

[84] D. Park, Y. Shin, and H. Kim. Control strategy for the ultra-super critical coal-firing thermal power plant. In *SICE-ICASE, 2006. International Joint Conference*, pages 1719–1721, Oct 2006.

[85] D. Patterson and J. Hennessy. *Computer Organization and Design: The Hardware/-Software Interface*. Morgan Kaufmann, fifth edition, 2014. Chapter 6, Pages 500-575.

[86] D. Patterson and J. Hennessy. *Computer Organization and Design: The Hardware/-Software Interface*. Morgan Kaufmann, fifth edition, 2014. Appendix C.

[87] D. Patterson and J. Hennessy. *Computer Organization and Design: The Hardware/-Software Interface*. Morgan Kaufmann, fifth edition, 2014.

[88] D. Patterson and J. Hennessy. *Computer Organization and Design: The Hardware/-Software Interface*. Morgan Kaufmann, fifth edition, 2014. Chapter 5, Pages 372-499.

[89] L. R. Petzold. A description of DASSL: A differential/algebraic system solver. In *Proc. IMACS World Congress*, 1982.

[90] R. Rajamani. *Vehicle Dynamics and Control*. Springer, second edition, 2012.

[91] R. Rajamani and D. Piyabongkarn. New paradigms for the integration of yaw stability and rollover prevention functions in vehicle stability control. *Intelligent Transportation Systems, IEEE Transactions on*, 14(1):249–261, March 2013.

[92] T. Rauber and G. Rünger. *Parallel Programming for Multicore and Cluster Systems*. Springer, 2013.

[93] A. Sangiovanni-Vincentelli. Quo vadis, SLD? reasoning about the trends and challenges of system level design. *Proceedings of the IEEE*, 95(3):467–506, March 2007.

[94] G. Simko, T. Levendovszky, M. Maroti, and J. Sztipanovits. Towards a theory for cyber-physical systems modeling. In *Proceedings of the 3rd Workshop on Design, Modeling and Evaluation of Cyber Physical Systems (CyPhy)*, 2013.

[95] M. Sjölund. Tools for understanding, debugging, and simulation performance improvement of equation-based models, 2013.

[96] M. Sjölund, R. Braun, P. Fritzson, and P. Krus. Towards efficient distributed simulation in modelica using transmission line modeling. In *EOOLT*, pages 71–80. Citeseer, 2010.

[97] M. Sjölund, M. Gebremedhin, and P. Fritzson. Parallelizing equation-based models for simulation on multi-core platforms by utilizing model structure. In A. Darte, editor, *Proceedings of the 17th Workshop on Compilers for Parallel Computing*, July 2013.

[98] S. Soejima and T. Matsuba. Application of mixed mode integration and new implicit inline integration at toyota, 2002.

[99] T. Sohmers. Neo and neo64, 2015. http://rexcomputing.com/REX_OCPSummit2015.pdf.

[100] K. Stavåker. Contributions to parallel simulation of equation-based models on graphics processing units, 2011.

[101] J. Sztipanovits, X. Koutsoukos, G. Karsai, N. Kottenstette, P. Antsaklis, V. Gupta, B. Goodwine, J. Baras, and S. Wang. Toward a science of cyber-physical system integration. *Proceedings of the IEEE*, 100(1):29–44, January 2012.

[102] R. Tarjan. Depth-first search and linear graph algorithms. *SIAM Journal on Computing*, 1(2):146–160, 1972.

[103] M. Walther, V. Waurich, and C. S. D.-I. I. Gubsch. Equation based parallelization of modelica models. In *Proceedings of the 10th International Modelica Conference*, Lund, Sweden, March 2014. Modelica.org.

[104] J. Willems. The behavioral approach to open and interconnected systems. *Control Systems, IEEE*, 27(6):46–99, Dec 2007.

[105] M.-Y. Wu and D. Gajski. Hypertool: a programming aid for message-passing systems. *Parallel and Distributed Systems, IEEE Transactions on*, 1(3):330–343, Jul 1990.

[106] Y. C. Yeh. Design considerations in boeing 777 fly-by-wire computers. In *High-Assurance Systems Engineering Symposium, 1998. Proceedings. Third IEEE International*, pages 64–72, Nov 1998.