APPROVAL SHEET

Title of Thesis: Parallel Performance of Numerical Simulations for Applied Partial Differential Equation Models on the Intel Xeon Phi Knights Landing Processor

Name of Candidate: Jonathan S. Graf
Doctor of Philosophy, 2017

Thesis and Abstract Approved: _____

Dr. Matthias K. Gobbert
Professor
Department of Mathematics and Statistics

Date Approved: _____

ABSTRACT

Title of Thesis: Parallel Performance of Numerical Simulations for Applied Partial Differential Equation Models on the Intel Xeon Phi Knights Landing Processor

Jonathan S. Graf, Doctor of Philosophy, 2017

Thesis directed by: Dr. Matthias K. Gobbert, Professor
Department of Mathematics and Statistics
University of Maryland, Baltimore County

Current high-performance computing clusters feature CPUs with 8 to 16 cores. The many-integrated-core (MIC) Intel Xeon Phi processors feature 60 or more cores on a single chip, with lower power consumption per core than CPUs. The Intel Xeon Phi Knights Landing (KNL) is the second-generation Xeon Phi processor released in 2016. It represents a significant improvement over the first-generation Knights Corner (KNC), since the KNL can serve as a standalone processor and has a 2D mesh interconnect on the chip to connect the cores to the 16 GB of high-performance memory on the chip. This architecture is very accessible to researchers who need only add a compiler flag to their code as a result of the x86 compatibility of each Xeon Phi core. But the different configurations available for the KNL add a layer of decisions for researchers on how to run their code. We use the Stampede cluster at the Texas Advanced Computing Center (TACC) for all hardware choices, since it is accessible to many researchers via an Extreme Science and Engineering Discovery Environment (XSEDE) allocation.

This work is inspired by the calcium induced calcium release (CICR) model of calcium dynamics in a three-dimensional heart cell. This application problem is modeled by a system of coupled, non-linear, time-dependent advection-diffusion-reaction partial differential equations. The model now includes eight species and connections between the electrical excitation, calcium signaling, and mechanical contraction systems. Parameter studies on modern CPUs examine the feedback strength from the

calcium signaling to the electrical excitation system and motivate the need for parameter studies on meshes that fit in the memory of a KNL.

The elliptic Poisson equation in two dimensions serves as a prototypical test problem, since the linear system solution by the conjugate gradient method mimics the computational kernels in many applications that use Krylov subspace methods. Our tests assess the configurations possible with the KNL and demonstrate the distinct advantage of the 16 GB of on-chip memory over the main memory of the node. For this problem, with localized communication and carefully managed memory efficiency, the performance of the main configuration choices are equivalent. We include a comparison to the first-generation KNC and modern CPU nodes currently available in Stampede and note the performance improvement.

Finally, we study the performance of the KNL when used for the CICR code. The CICR code requires more demanding and significant communication and is more computationally intensive than the Poisson problem. We demonstrate performance and scalability on a single KNL node with MPI only and hybrid MPI+OpenMP code. We also test scalability using multiple KNL and carefully consider the number and placement of OpenMP threads relative to the number of MPI processes used with hybrid MPI+OpenMP code. The scalability for multiple KNL nodes is good for both MPI and MPI+OpenMP code. The balance of OpenMP threads to MPI processes influences performance for this problem. Overall, the KNL demonstrates significant performance benefit when used appropriately on various application problems.

PARALLEL PERFORMANCE OF NUMERICAL SIMULATIONS FOR APPLIED PARTIAL DIFFERENTIAL EQUATION MODELS ON THE INTEL XEON PHI KNIGHTS LANDING PROCESSOR

by

Jonathan S. Graf

Thesis submitted to the Faculty of the Graduate School
of the University of Maryland in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
2017

© Copyright Jonathan S. Graf 2017

To my wife

ACKNOWLEDGMENTS

Many thanks to my advisor, Dr. Matthias K. Gobbert, for his consistent support, attention to detail, organization, and planning. I have learned many lessons from the intentional way in which Dr. Gobbert operates. Thanks to Dr. Bradford E. Peercy for his guidance and contributions on the calcium work throughout the course of my time at UMBC and for his insight in this work. Thanks to Dr. Marc Olano for his insight in hardware performance and careful review of this work. Also thanks to Dr. Meilin Yu and Dr. Bedřich Sousedík who have consistently offered their support and advice. Special thanks also to my other collaborators and friends. I would especially like to thank, Dr. Zana Coulibaly for his extension of the CICR model and his contributions to this work, Dr. Samuel Khuvis for his introduction of the KNL and his willingness to answer questions and provide guidance especially with the calcium code, OpenMP, and profiling with TAU, and Dr. Xuan Huang for introduction to the calcium code and insight with GPU coding. Also thanks to Team 5 of the REU in 2016, Kallista Angeloff, Carlos Barajas, Alexander D. Middleton, and Uchenna Osia, for their work with the calcium code, and to all of the other REU students and clients with whom I have worked. Finally, thanks to my family for all of the encouragement and support, especially, my wife for her deep understanding and commitment to the journey, my parents who are the rocks on which I stand, my brother who inspires me, and my sisters for their unwavering love.

I acknowledge financial support as Research Assistant for the High Performance Computing Facility (HPCF) at UMBC.

The hardware used in some of the computational studies is part of the UMBC High Performance Computing Facility (HPCF). The facility is supported by the U.S. National Science Foundation through the MRI program (grant nos. CNS–0821258 and CNS–1228778) and the SCREMS program (grant no. DMS–0821311), with additional

substantial support from the University of Maryland, Baltimore County (UMBC). See `hpcf.umbc.edu` for more information on HPCF and the projects using its resources.

This work also used the Extreme Science and Engineering Discovery Environment (XSEDE), which is supported by National Science Foundation grant number ACI–1053575. We acknowledge the Texas Advanced Computing Center (TACC) at The University of Texas at Austin for providing HPC resources that have contributed to the research results reported within this thesis.

We would also like to thank the Performance Research Laboratory, University of Oregon for access to the Grover KNL hardware through Dr. Samuel Khuvis and ParaTools, Inc..

TABLE OF CONTENTS

Page

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

1.1 Motivation

Heart disease is currently the leading cause of death in the United States, according to the Centers for Disease Control and Prevention [7]. Recent studies have shown that cardiac arrhythmias result from the disruption of the very tight coupling between the electrical, calcium, and mechanical properties of the heart [52]. Though devices like pacemakers have been shown to help reduce the death rate due to arrhythmia, they do not prevent onset arrhythmia. A more in-depth understanding of the calcium dynamics may yield new methods in the realm of drug therapy. To date, no medication has been developed that has been proven to be effective in more than isolated cases [45, 54]. It has been shown that the dysregulation of the interaction between the electrical, calcium, and mechanical systems is a precursor to cardiac arrhythmias [13, 41]. In particular, the disruption of the bi-directional coupling between calcium and electrical system can result in alternans (unwanted modulation of oscillation in the cell's calcium and electrical activity) [11, 52]. These alternans often precede arrhythmias.

Before we can better study the heart, we model the calcium induced calcium release (CICR) in an individual heart cell using the three components associated with the dynamics: electrical excitation, calcium signaling, and mechanical contraction. We represent the fully coupled electrical excitation, calcium signaling, and mechanical contraction components of the calcium and electrical dynamics by a system of eight time-dependent coupled partial differential equations (PDEs). The PDEs of the model are coupled, non-linear, advection-diffusion-reaction equations of the form

$$u_t^{(i)} - \nabla \cdot \left(D^{(i)}\, \nabla u^{(i)}\right) + \beta^{(i)} \cdot \left(\nabla u^{(i)}\right) + a^{(i)}\, u^{(i)} = q^{(i)}, \quad i = 1, \ldots, n_s, \tag{1.1.1}$$

with functions $u^{(i)} = u^{(i)}(\mathbf{x}, t)$, $i = 1, \ldots, n_s$, of space $\mathbf{x} \in \Omega \subset \mathbb{R}^3$ and time $0 \leq t \leq t_{\text{fin}}$ representing the concentrations of the n_s species. The solution of this PDE requires sophisticated numerical methods [6, 14, 18, 33, 46]. For reasonable simulation times, we take advantage of the power of parallel computing. For the numerical method we use a method of lines technique for which the spatial discretization results in a stiff system of ordinary differential equations (ODEs) that must be solved at each time step. We use the finite volume method as the spatial discretization so that advection and diffusion can both be present in (1.1.1) [22, 46]. We use both MPI and OpenMP parallelism in the implementation of this special purpose code [6].

Recent developments in parallel computing architectures includes the use of graphics processing units (GPUs) as a massively parallel accelerator in general purpose computing and many-integrated-core (MIC) architectures like the Intel Xeon Phi. Besides the larger number of computational cores in each, the key difference to a CPU is their significant on-chip memory, on the order of several GB. The architectures of GPU and Phi differ significantly from each other, and the use of GPUs requires significant code modifications. The x86 compatibility of each Xeon Phi core makes porting of code to this architecture as simple as the addition of a compiler flag, thus making it an excellent starting point. The recent emergence of the second-generation Intel Xeon Phi, codenamed Knights Landing (KNL), represents a significant improvement over the first-generation. The KNL was announced in June 2014 [24] and began shipping in July 2016. The KNL itself is like a 'massively parallel' supercomputer from the early 2000s with dozens of nodes connected by a Cartesian network, all in a single chip now with a theoretical peak performance of over 3 TFLOP/s of double-precision floating-point performance [49]. Already the first-generation Phi, codenamed Knights Corner (KNC), had an impact since its appearance in 2012, as exhibited by many of the highest-ranked clusters on the Top 500 list (`www.top500.org`) since then that use

the Phi, but the KNL has significant improvement in on-chip memory over the KNC. Two clusters using pre-production or early-production KNL chips achieved ranks #5 and #6 on the November 2016 Top 500 list. Entry #5 is the Cori cluster at NERSC (`www.nersc.gov`) in the USA with Cray XC40, Intel Xeon Phi 7250 68C 1.4GHz, and Aries interconnect. Entry #6 is the Oakforest-PACS cluster at the Joint Center for Advanced High Performance Computing in Japan with PRIMERGY CX1640 M1, Intel Xeon Phi 7250 68C 1.4GHz, and Intel Omni-Path network. The same KNL model as in these machines is used in this work.

At each time step in the solution of the PDE system for the CICR application, a linear system with a large, very sparse, highly structured system matrix must be solved. Methods from the family of Krylov subspace methods are standard for this purpose, since they permit a matrix-free implementation. That is, all matrix-vector products needed in the Krylov subspace method are provided directly without storing the matrix, allowing significant memory savings and the more efficient solution of larger problems. We use a stationary classical elliptic scalar test problem, the Poisson equation

$$-\Delta u = f \tag{1.1.2}$$

in the domain $\Omega = (0,1) \times (0,1) \subset \mathbb{R}^2$, that requires the same type linear system to mimic the computational kernel in the CICR model and many other simulations. Before running the full CICR application code on the KNL, we study performance on the KNL in detail using (1.1.2). In particular we assess performance advantages to the different memory and cluster mode configurations available on the KNL as well as the distribution of MPI processes and OpenMP threads over the architecture.

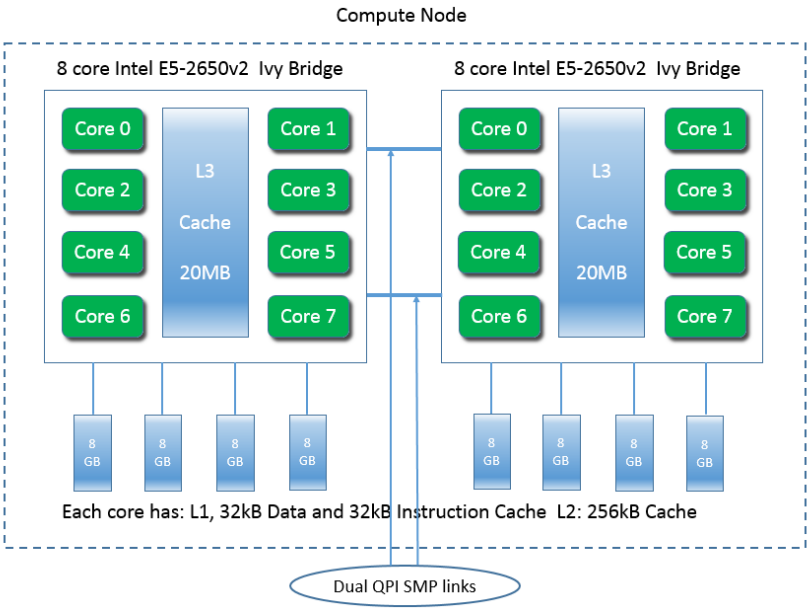


Figure 1.2.1 Schematic of a dual socket CPU node with two 8-core CPUs on the maya cluster. Source: `hpcf.umbc.edu`.

1.2 Intel Xeon Phi Knights Landing (KNL)

The size and structure of modern computing processors has developed significantly in recent years. As the rapid processing speed increases of a single chip stalled in the presence of the physical issues of heat and power consumption, a shift to multi-core architectures occurred. Today, CPUs in consumer devices are dual- or quad-core. The iPhone 7 features a quad-core processor, as do most mainstream laptops. State-of-the-art distributed-memory clusters contain multi-core CPUs with 8 to 16 cores. Figure 1.2.1 shows a schematic of a dual-socket CPU node with two 8-core CPUs on maya cluster in the High Performance Computing Facility (HPCF) at UMBC.

This trend also includes the emergence of general-purpose graphics processing units (GPGPUs). In this case, graphics processing units (GPUs), which originally were designed to handle computer graphics computations, are used for computations in applications that were handled by the CPU previously. The GPUs are set up as

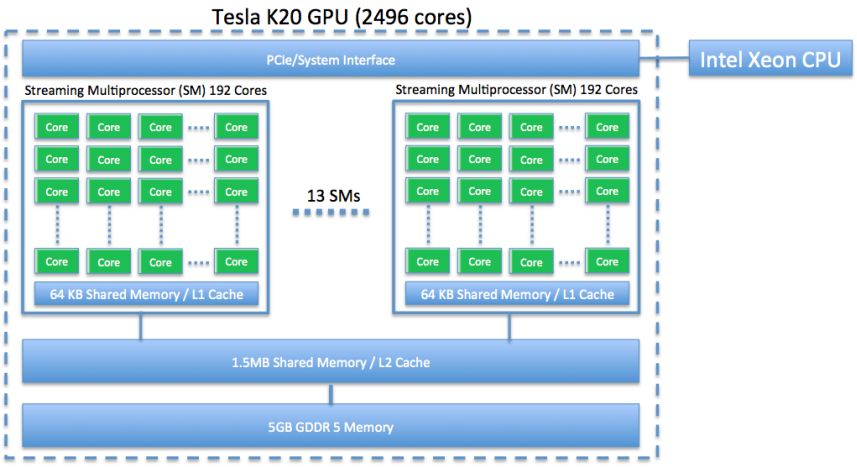


Figure 1.2.2 Schematic of a NVIDIA K20 GPU co-processor to a CPU on the maya cluster. Source: `hpcf.umbc.edu`.

a co-processor to a CPU and include a very large number of cores. One example is the NVIDIA K20 GPU, which has 5 GB of on-chip memory and features 2,496 computational cores that are distributed over 13 streaming multiprocessors (SMs) for a total of 196 cores on each SM. A significant challenge to GPGPU computing is the need to modify the existing code to run on the GPU. This requires a GPU programming language like Open Computing Language (OpenCL) or a framework like Compute Unified Device Architecture (CUDA). Figure 1.2.2 shows a schematic of a NVIDIA K20 GPU as a co-processor to the CPU on a hybrid node on maya cluster in the High Performance Computing Facility (HPCF) at UMBC.

Unlike GPUs, each Intel Xeon Phi core is an x86 compatible architecture which allows the user to run the same code on the Phi as they run on the CPU. This represents a very significant advantage to the programmer, as their code can be quickly run on the Phi with only the addition of a compiler flag. The Intel many-integrated-core (MIC) Xeon Phi processors feature more more than 60 cores. The cores in the Phi are slower than the cores in a modern CPU, for example, 1.4 GHz KNL

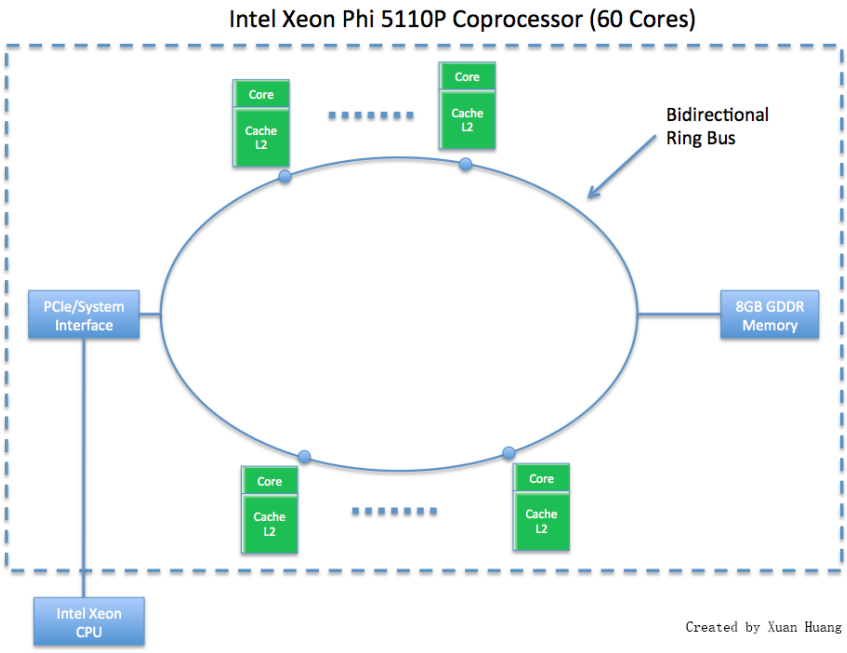


Figure 1.2.3 Intel schematic of a KNC with 60 cores connected by a bi-directional ring bus and 8GB on-chip memory. Source: `hpcf.umbc.edu`.

cores versus 2.7 GHz CPU cores. But Phi chips also include on-chip memory, like the GPU, while CPUs have essentially no on-chip memory. The first-generation of the Phi, codenamed Knights Corner (KNC), must be configured as a co-processor to a CPU, like a GPU is a co-processor to a CPU. The KNC includes, for example, 8 GB of GDDR on-chip memory connected with the cores through a bidirectional ring bus as shown in Figure 1.2.3.

The second-generation of the Phi, codenamed Knights Landing (KNL), represents a very different design from the KNC. The KNL can serve as a standalone processor, without a CPU host. Figure 1.2.4 shows a schematic of a KNL. The crucial improvements of the KNL are the 2D mesh interconnect providing high-bandwidth connections on the chip and to the high-performance 16 GB Multi-Channel DRAM (MCDRAM) memory on board the chip. The KNL can have up to 72 cores, with 2

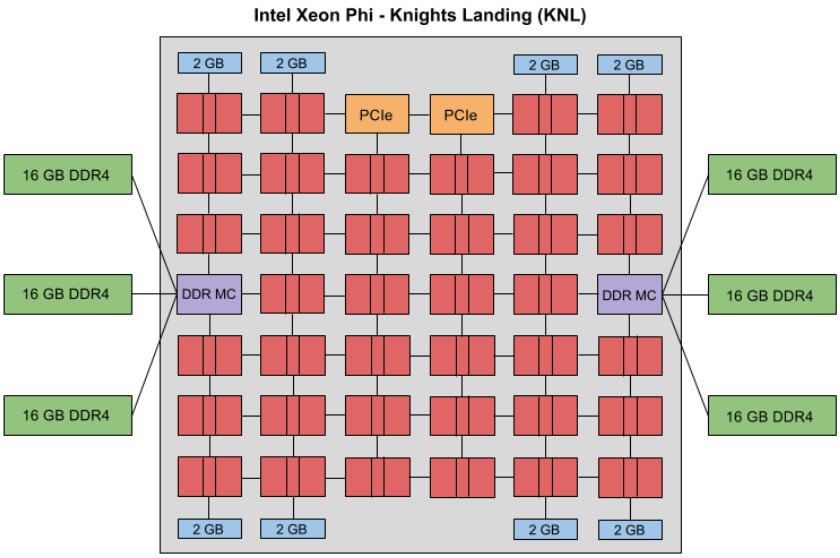


Figure 1.2.4 Schematic of a KNL with 2 cores per tile, connected in a 2D mesh structure.

cores on each of the tiles in the 2D mesh interconnect. The Phi can also access the DDR4 memory of the node, but MCDRAM is directly in the chip and is nominally 5x faster than the DDR4 memory [49]. The configuration of the KNL is versatile, but requires a choice at boot time on the configuration of the MCDRAM memory relative to the DDR memory of the node and choice of clustering mode for low level memory access localization. The Intel Developer Zone includes a tutorial on the High Bandwidth Memory on the KNL [28] and notes explicitly that "with the different memory modes by which the system can be booted, it becomes very challenging from a software perspective to understand the best mode suitable for an application." We focus our attention on the different KNL configurations and show performance results. For a complete description of the KNL, see Section 3.3.

1.3 Overview and Outline

The remaining chapters of this thesis detail a study of the CICR application, the performance of the KNL for the Poisson problem (1.1.2), and performance of the KNL for the parabolic CICR model (1.1.1) under the various possible configurations of the hardware.

Chapter 2 presents the expansion of the CICR model from a model with only calcium signaling and electrical excitation components to a complete 8-species model with electrical excitation, calcium signaling, and mechanical contraction components. We start to obtain an insight into the delicate electro-chemical balance at the scale of a cardiomyocyte with parameter studies of the feedback-feedforward linking between the calcium and electrical systems.

Chapter 3 studies the performance of the elliptic test problem (1.1.2) on the KNL. The details of the KNL hardware are presented with clear descriptions of the choices available for KNL configurations and the impact each choice may have on performance. Detailed performance comparisons are presented for the KNL configurations with clear run time instructions. Also, we explore the distribution of MPI processes and OpenMP threads.

Chapter 4 uses the developments and understanding of the KNL from Chapter 3 on the current CICR model from Chapter 2 to outline optimal simulation capabilities on the KNL. We study the performance of the application code on the KNL. This includes strong scalability studies and careful consideration of the number and placement of threads relative to the number of MPI processes used. In particular, we identify the ideal way to use the KNL for the application code, including using multiple KNL nodes.

Chapter 5 summarizes our conclusions and motivates future work on the KNL and with the calcium induced calcium release application problem.

CHAPTER 2

CALCIUM DYNAMICS IN A CARDIOMYOCYTE

This chapter studies the calcium induced calcium release (CICR) model of calcium dynamics in a cardiomyocyte. The content of this chapter is based on [2].

2.1 Introduction

Heart disease is currently the leading cause of death in the United States, according to the Centers for Disease Control and Prevention [7]. Recent studies have shown that cardiac arrhythmias result from the disruption of the very tight coupling between the electrical, calcium, and mechanical properties of the heart [52]. The current treatment for cardiac arrhythmia, mild or otherwise, is surgical implantation of a pacemaker to artificially stimulate the electrical patterns which would be otherwise naturally produced levels in a healthy heart. To date, no medication has been developed that been proven to be effective in more than a handful of cases [45, 54].

Though devices like pacemakers have shown to help reduce the death rate due to arrhythmia, they do not prevent onset arrhythmia. A more in-depth understanding of the calcium dynamics may yield new methods in the realm of drug therapy. In order to better study the heart, we examine individual cells and three components associated with the dynamics: electrical excitation, calcium signaling, and mechanical contraction. These three systems are coupled together as outlined in Figure 2.1.1 with calcium signaling being the central dynamic component between the electrical excitation and mechanical contraction. It has been shown that the dysregulation of the interaction between these three systems is a precursor to cardiac arrhythmias [13, 41]. In particular, the disruption of the bi-directional coupling between calcium and electrical system can result in alternans (unwanted modulation of oscillation in the cell's

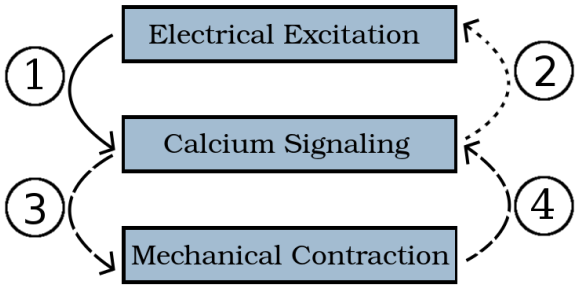


Figure 2.1.1 The calcium-mediated contractile rhythm of a given cardiomyocyte is a function of three coupled dynamics: electrical excitation, calcium signaling, and mechanical contraction. Their links are labeled ① to ④ for reference throughout the text.

calcium and electrical activity) [11,52]. These alternans often precede an arrhythmia. Our goal is to obtain an insight into the delicate electro-chemical balance at the scale of a cardiomyocyte by expanding upon a model of calcium induced calcium release to include the feedback-feedforward between calcium and electrical system.

The links shown in Figure 2.1.1 each represent a physiological component of the potential for *calcium induced calcium release (CICR)* inside the cardiac cell. The solid line for link ① refers to a link from the electrical to the calcium system, whose behavior was the focus of the simulations in [1], while the dotted line for link ② is the link from the calcium to the electrical system, whose effect is the focus of the present simulations. The dashed lines labeled ③ and ④ are the links between the calcium and the mechanical system, whose model is introduced here and in [1].

In normal conditions, the periodic nonarrhythmic contraction and relaxation of a cardiac myocyte is governed by periodic action potentials (cell's membrane depolarization and repolarization). In a regular functioning cardiac myocyte, the depolarization (increase in voltage) of the cell's membrane is due to an influx of sodium ions. The depolarization of the cell's membrane causes L-type calcium channels (LCC) to open

thus causing an influx of calcium ions (Ca^{2+}) into the cell. The influx of calcium ions inside the cell causes a localized increase in calcium concentration. Inside the heart cell, calcium ions are primarily stored in the sarcoplasmic reticulum (SR). The SR has calcium sensitive sites called calcium release units (CRUs - groups of individual calcium-sensitive ryanodine receptors) which, when the concentration of calcium is high enough, open to release calcium into the cytosol from the SR of the cell. The depolarization of the cell membrane also occurs when calcium ions are pumped out of the cell and sodium ions are pumped into the cell through the sodium-calcium exchanger. While these system processes are interacting, calcium is also binding and unbinding to immobile contractile proteins inside of the cytosol. Within a sarcomere (a contractile unit) calcium binds to troponin uncovering myosin muscle heads. Released Myosin binds to anchored actin filaments generating contraction. Repolarization of the membrane (decrease of the membrane voltage) occurs afterwards due to the activity of the potassium channels. The cell relaxation occurs when most of the calcium ions are pumped back into the SR through SR pumps. A closer analysis of these interactions is presented inside Section 2.2.

We represent the fully coupled electrical excitation, calcium signaling, and mechanical contraction components of the calcium and electrical dynamics by a system of eight time-dependent coupled partial differential equations (PDEs). The calcium signaling is described by five PDEs modeling concentrations of calcium ions and buffer species in the cytosol and SR. The electrical excitation, connected to the calcium system by links ① and ②, is represented by two PDEs: one representing voltage and one gating variable that modulated conductance through the K+ channel. The mechanical contraction, connected to the calcium system by links ③ and ④, is described by a final PDE representing the concentration of actively linked contractile proteins in the cytosol. A full description of the mathematical model is presented in Section 2.3.

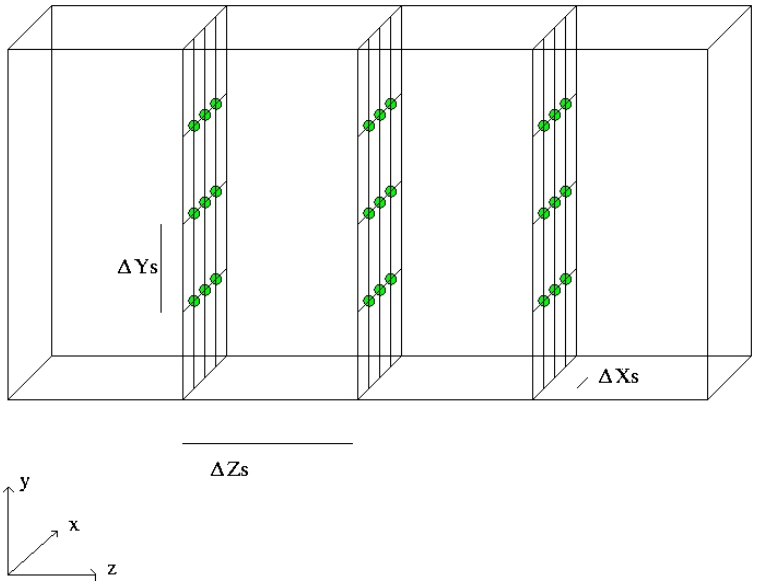


Figure 2.1.2 The CRU lattice with spacings Δx_s, Δy_s, Δz_s throughout the three-dimensional domain.

We use numerical methods to perform simulations of this system of time-dependent, coupled, parabolic partial differential equations The long time simulations required demand sophisticated numerical methods. We use a method of lines (MOL) approach and use the finite volume method (FVM) for the spatial discretization and the numerical differentiation formulas (NDFk) for time stepping. A memory efficient parallel implementation of this is done in C using MPI (Message Passing Interface, the most popular parallel communications library [40]) commands for parallel computing. A more detailed overview of the numerical methods be found in Section 2.4.

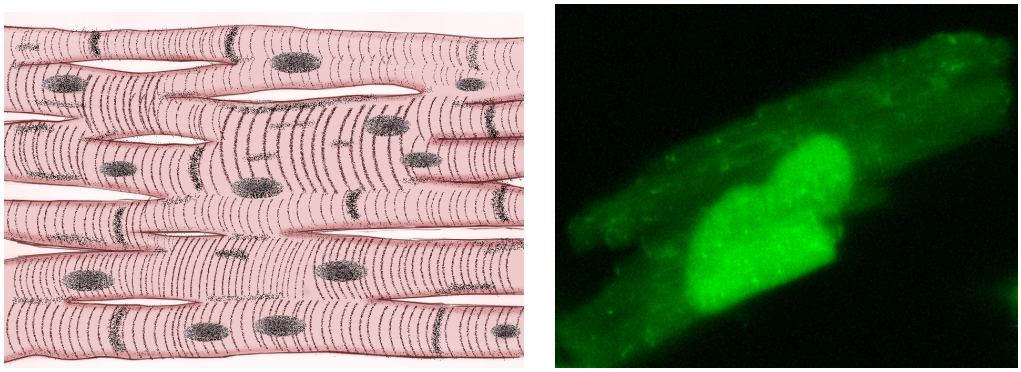
In Section 2.5, we present simulations of our model that examine back and forth interaction between the electrical excitation and calcium signaling systems. The simulations use as domain an elongated three-dimensional hexahedron that captures the key feature of a heart cell. Figure 2.1.2 shows the distances of the z-planes of CRUs being Δz_s apart, and the location of CRUs on each z-plane on a rectangular lattice with distances Δx_s and Δy_s. In Section 2.5.1, we show behavior with only the feed-

forward connection in which the electrical excitation impacts the calcium signaling, link ① in Figure 2.1.1. In Section 2.5.2 we analyze the impact of introducing the feedback connection enabling the calcium signaling to impact the electrical excitation in our model, that is, both link ① and link ② in Figure 2.1.1 are turned on. To do this, we modify the parameter that turns on and off the feedback connection, ω, and study the influence of different strengths of this feedback. This study examines the impact of this change on concentration of calcium in both the cytosol and SR We also show figures with the open CRUs at various time values and figures with the concentration of calcium in the cytosol at various time values.

Finally, Section 2.6 summarizes this work and presents opportunities for future work. Simulations indicate that the feedback and feedforward between electrical excitation and calcium signaling can influence the voltage in a physiologically realistic way. That is, the simulated action potential is qualitatively similar to an experimental range and duration. The interplay between the strengths of the feedback and feedforward links of electrical excitation and calcium signaling and impact of the mechanical contraction components can now be studied.

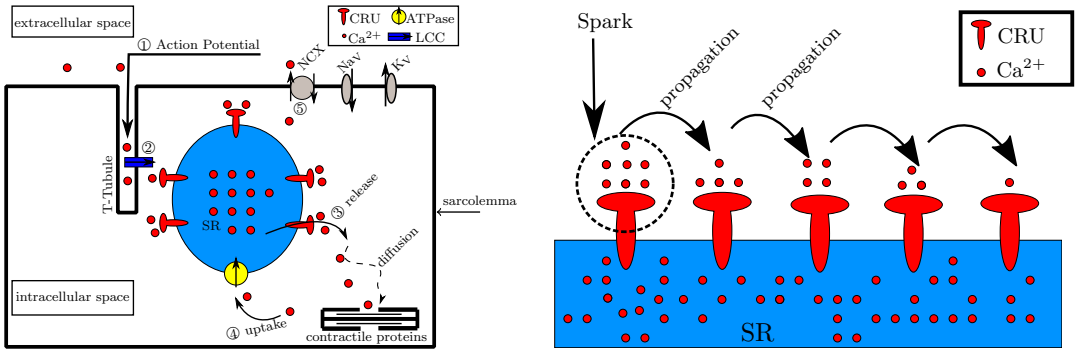
2.2 Background

The general structure of cardiomyocytes on the heart can be seen in Figure 2.2.1(a). The general shape of the cardiac cell is rectangular with several T-tubules along the side of the cell. The light pink areas of Figure 2.2.1(a) represent the cellular membrane of the cardiomyocyte. The muscle fibers run parallel to the contractile proteins of the cardiac cell. The dark splotches are the cell nuclei associated with their respective cell. Inside the cardiomyocyte is the sarcoplasmic reticulum (SR), a type of container that contains among other molecules, both calcium ions and calsequestrin (CSQ). The release of calcium from the SR into the cytosol occurs via calcium release



(a) Cardiac cell structure (b) Calcium filled cardiomyocyte

Figure 2.2.1 Heart Cell Structure [2]. (a) A conglomerate of cardiac cells. The muscle fibers of each heart cell contract in response to the change in shape of contractile proteins, in turn mediated by calcium levels in the cytosol. The thin striations mark the ends of each contractile unit, a sarcomere, within a cell. Darker striations are intercalated disks that join individual cardiac cells into muscle fibers. (b) An individual rabbit cardiomyocyte illuminated by a fluorescent dye. Experiment conducted by Dr. Kenneth Spitzer (University of Utah, personal communication with Dr. Bradford Peercy, June 2010).



(a) Cellular space (b) Calcium wave triggering

Figure 2.2.2 Interior of Cell Behavior Schematics. (a) The T-tubules enfold the L-type calcium channels (LCC) in the cell's plasma membrane. Periodic membrane depolarizations allow calcium to pass into the cytosol. (b) Calcium released from the calcium release units (CRUs) on the SR and begin the cascading calcium release that starts calcium wave propagation. [2]

units (CRUs) on the SR. Once the concentration is high enough, the CRUs will begin to open; this process is called *sparking*. Sparks are a few ryanodine receptors within the CRU opening and do not usually propagate to neighboring CRUs [8]. The fluorescent dye is mixed in the cytosol and used to make the calcium more visible during lab experiments; the dye diffuses through the cytosol interacting with and binding to calcium in the cytosol.

How these components combine to affect small strands of muscle fiber and cause cell pulsation is shown in Figure 2.2.2(a). The sodium-calcium electrical exchanger, labeled as NCX near the top of Figure 2.2.2(a), pushes a calcium ion out of the cell while bringing three sodium ions into cell. Calcium leaving the cell is part of a feedback mechanism, designated as link ② in Figure 2.1.1, by which the electrical properties of the heart are influenced by calcium concentration in the cytosol. When the concentration begins to change, it leads to a phenomenon in which regular de-

polarizations of the cell plasma membrane happen; the depolarization induces action potential, causing the L-type Calcium Channels (LCC) to open. This feedforward mechanism, represented by link ① in Figure 2.1.1, is another one-way coupling which allows for the electrical aspect of the cardiac cell to have an influence on the calcium concentration of the cytosol. These two methods result in a two-way coupling between the electrical excitation system and calcium signaling in the cell.

As the CRUs release more calcium into the cytosol, the spike in concentration can trigger neighboring CRUs to open as well possibly leading to the cascading effect depicted in Figure 2.2.2(b). This wave can propagate until the the end of the cell. When calcium begins to pour into the cytosol, the concentration begins to rise, triggering a wave event within the cell: calcium release [12]. This behavior is known as *calcium induced calcium release (CICR)*. As calcium diffuses through the cytosol, it reacts with other chemical species; among them, in this model, are the fluorescent dye fluoro-4 and tropomyosin contractile proteins.

The contraction and expansion of the cell's shape is a result of the actin and myosin contractile proteins and the troponin complex, made up of tropomyosin and troponin. These are depicted in the bottom right-hand corner of Figure 2.2.2(a) and attach to a sarcomere. When calcium binds to the troponin complex, the myosin heads are free to converge to the actin bridge; when the myosin heads come in contact with the bridge the striated muscle, the sarcomere, which is parallel to the tropomyosin, contracts. This myosin contraction is the physical process through which the cell expands and contracts; when these contractions are performed in unison with other cardiac cells, that section of the heart contracts. This process of calcium causing heart contractions presents the first coupling between the calcium and the contractile nature of a heart cell; these chemical interactions are a feedforward process and represented by link ③ in Figure 2.1.1. Once calcium is bound to the complex, the bridge-like structure

deforms, causing the rate at which calcium unbinds to decrease. Conceptually the change in the troponin complex causes the bridge to hang onto the calcium for longer. When the calcium is relinquished from the bridge, this increases the concentration of calcium in the cytosol as a feedback process, which is represented by link ④ in Figure 2.1.1.

2.3 Model

In this section, we present the equations of the complete model along with parameter tables and descriptions of how the equations represent the physiological components. The PDEs of the model are (2.3.1), (2.3.2) with $n_{sc} = 3$, (2.3.3), (2.3.4) with $n_{ss} = 1$, (2.3.12), and (2.3.13) yielding a total number of $n_s = 4 + n_{sc} + n_{ss} = 8$ PDEs. The eight species of the model are: calcium in the cytosol $c(\mathbf{x}, t)$, a florescent dye $b_1^{(c)}(\mathbf{x}, t)$, a contractile mechanism protein (troponin) $b_2^{(c)}(\mathbf{x}, t)$, a contractile actin-myosin cross-bridges buffer $b_3^{(c)}(\mathbf{x}, t)$, calcium in the SR $s(\mathbf{x}, t)$, calsequestrin in the SR $b_1^{(s)}(\mathbf{x}, t)$, voltage $V(\mathbf{x}, t)$, and a gating variable for the potassium channel $n(\mathbf{x}, t)$. We treat this problem as a bi-domain problem where at any point $\mathbf{x}$, both the cytosol and SR exist, while in reality they occupy separate domains.

Section 2.3.1 describes the calcium signaling portion of the model without the presence of electrical excitation or mechanical contraction that was originally introduced in [32, 34], extended in [33], and its numerics discussed with $n_{sc} = 2$ in [14, 18, 46, 47]. Section 2.3.2 introduces the electrical excitation that is connected to the calcium signaling in both the feedforward and feedback directions represented by link ① and link ② in Figure 2.1.1. Link ① from electrical system to the calcium dynamics was first established in [1]. Finally, Section 2.3.3 completes the links with the calcium signaling in the model by the addition of the mechanical contraction component that is also connected to the calcium signaling in both the feedback and feedforward di-

rections represented by links ③ and ④ in Figure 2.1.1. The effects of cell contraction are implemented via a pseudo-mechanical model which describes force in terms of the proportion of actively connected contractile proteins.

2.3.1 Calcium signaling

We start with a system of reaction diffusion PDEs

$$\begin{aligned}
\frac{\partial c}{\partial t} &= \nabla \cdot (D_c \nabla c) + \sum_{i=1}^{n_{sc}} R_i^{(c)} + (J_{CRU} + J_{leak} - J_{pump}) \qquad (2.3.1)\\
&\quad + \kappa J_{LCC} + J_{m_{leak}} - J_{m_{pump}},\\
\frac{\partial b_i^{(c)}}{\partial t} &= \nabla \cdot (D_{b_i^{(c)}} \nabla b_i^{(c)}) + R_i^{(c)}, \quad i = 1, \ldots, n_{sc}, \qquad (2.3.2)\\
\frac{\partial s}{\partial t} &= \nabla \cdot (D_s \nabla s) + \sum_{j=1}^{n_{ss}} R_j^{(s)} - \gamma(J_{CRU} + J_{leak} - J_{pump}), \qquad (2.3.3)\\
\frac{\partial b_j^{(s)}}{\partial t} &= \nabla \cdot (D_{b_j^{(s)}} \nabla b_j^{(s)}) + R_j^{(s)}, \quad j = 1, \ldots, n_{ss}, \qquad (2.3.4)
\end{aligned}$$

where $c(\mathbf{x}, t)$ and $s(\mathbf{x}, t)$ represent the concentrations of calcium in the cytosol and SR, respectively. $b_i^{(c)}(\mathbf{x}, t)$ and $b_j^{(s)}(\mathbf{x}, t)$ represent the concentration of each buffer species in the cytosol and SR, respectively. Table 2.3.1 collects the variables with their units as well as the values of parameters in the PDEs of the calcium system. $D_c, D_s, D_{b_i^{(c)}}$, and $D_{b_j^{(s)}}$ are diffusion matrices for Ca^{2+} in the cytosol, Ca^{2+} in the SR, and each buffer species in the cytosol and SR, respectively. While each buffer species programmatically possesses a diffusion matrix (following the template of (2.3.2) and (2.3.4)), not all species are mobile; hence the diffusion matrices for some species are zero matrices in Table 2.3.1.

The reaction terms $R_i^{(c)}$ and $R_j^{(s)}$ in (2.3.5) and (2.3.6) describe the reactions between calcium and the buffer species. They are the connections between (2.3.1) and (2.3.2), and between (2.3.3) and (2.3.4). More precisely,

$$R_i^{(c)} = -k^+_{b_i^{(c)}}\, c\, b_i^{(c)} + k^-_{b_i^{(c)}} \left(b_{i,total}^{(c)} - b_i^{(c)}\right), \quad i = 1, \ldots, n_{sc} - 1, \qquad (2.3.5)$$

Table 2.3.1 Variables and parameters for calcium signaling: PDEs.

Variable	Definition	Values/Units
$\mathbf{x}$	spatial position variable (x, y, z)	μm
t	time variable	seconds
$c(\mathbf{x}, t)$	cytosol calcium concentration	μM
$s(\mathbf{x}, t)$	SR calcium concentration	μM
n_{sc}	number of cytosol Ca^{2+} buffer species	2
n_{ss}	number of SR Ca^{2+} buffer species	1
${b_1^{(c)}}_{total}$	total amount of $b_1^{(c)}(\mathbf{x}, t)$, dye, in cytosol	50 μM
${b_2^{(c)}}_{total}$	total amount of $b_2^{(c)}(\mathbf{x}, t)$, troponin, in cytosol	123 μM
${b_1^{(s)}}_{total}$	total amount of $b_1^{(s)}(\mathbf{x}, t)$, calsequestrin, in SR	6000 μM
D_c	cytosolic calcium diffusion coefficient matrix	diag(0.15, 0.15, 0.3)
D_s	SR calcium diffusion coefficient matrix	diag(0.78, 0.78, 0.78) μm^2/ms
$D_{b_1^{(c)}}$	cytosol buffer diffusion coefficient matrix ($i = 1$, dye)	diag(0.01, 0.01, 0.02) μm^2/ms
$D_{b_2^{(c)}}$	cytosol buffer diffusion coefficient matrix ($i = 2$, troponin)	diag(0.00, 0.00, 0.00) μm^2/ms
$D_{b_1^{(s)}}$	SR buffer diffusion coefficient matrix ($i = 1$, calsequestrin)	diag(0.00, 0.00, 0.00) μm^2/ms
$R_i^{(c)}, R_j^{(s)}$	reactions of cytosol, SR Ca^{2+} with buffers	μM/ms
$k^+_{b_1^{(c)}}$	forward reaction coefficient for $b_1^{(c)}$, dye	80×10^{-3} μM/ms
$k^+_{b_2^{(c)}}$	forward reaction coefficient for $b_2^{(c)}$, troponin	100×10^{-3} μM/ms
$k^+_{b_1^{(s)}}$	forward reaction coefficient for $b_1^{(c)}$, calsequestrin	39×10^{-3} μM/ms
$k^-_{b_1^{(c)}}$	reverse reaction coefficient for $b_1^{(c)}$, dye	90×10^{-3} ms^{-1}
$k^-_{b_2^{(c)}}$	reverse reaction coefficient for $b_2^{(c)}$, troponin	100×10^{-3} ms^{-1}
$k^-_{b_1^{(s)}}$	reverse reaction coefficient for $b_1^{(s)}$, calsequestrin	78 ms^{-1}
γ	ratio of volume of cytosol to SR	14
c_0	basal cytosol calcium concentration	0.1 μM
J_{pump}	calcium transfer from cytosol to SR	μM/ms
J_{leak}	calcium leak from SR	μM/ms
V_{pump}	maximum pump rate	2 to 6 μM/ms
K_{pump}	pump sensitivity to Ca^{2+}	0.184 μM
n_{pump}	Hill coefficient for pump function	4.0
s_0	initial SR calcium concentration	1,000 to 10,000 μM
J_{CRU}	calcium flux from SR to cytosol via CRUs	μM/ms
$\mathcal{O}$	gating function for J_{CRU}	1
J_{prob}	probability of CRU opening	0 to 1
$\mathbf{x}$	three-dimensional vector for CRU location	μm
$\hat{\sigma}$	maximum rate of release	100 to 200 μM μm^3/ms
u_{rand}	uniformly distributed random variable	0 to 1
n_{prob}	Hill coefficient for probability function	1.6
P_{max}	maximum probability for release	0.3
K_{prob_c}	sensitivity of CRU to cytosol calcium	5 to 15 μM
K_{prob_s}	sensitivity of CRU to SR calcium	200 to 550 μM

model the reactions between cytosolic Ca^{2+} and each cytosolic buffer species, and

$$R_j^{(s)} \;=\; -\,k^{+}_{b_j^{(s)}}\, s\, b_j^{(s)} + k^{-}_{b_j^{(s)}} \left(b^{(s)}_{j,total} - b_j^{(s)}\right), \quad j = 1, \ldots, n_{ss}, \tag{2.3.6}$$

model the reactions between SR Ca^{2+} and each SR buffer species. The amounts of "free" calcium ions, $c(\mathbf{x}, t)$ and $s(\mathbf{x}, t)$ in (2.3.1) and (2.3.3), respectively, and of "free" buffer species in (2.3.2) and (2.3.4), respectively, are determined by these reactions: whatever has not been bound by a reaction is the concentration remaining. In the cytosol, two buffer species are considered: a fluorescent dye, $b_1^{(c)}(\mathbf{x}, t)$, and a contractile protein troponin, $b_2^{(c)}(\mathbf{x}, t)$. We will revisit the subject of troponin in our extension of this model to include the pseudo-mechanical dynamics of the cell. In the SR, a single buffer species is considered: calsequestrin, $b_1^{(s)}(\mathbf{x}, t)$, a calcium-binding protein which helps maintain the SR calcium reserves at a much higher concentration than the cytosol.

The flux terms J_{CRU}, J_{leak}, and J_{pump} in (2.3.1) describe the calcium induced release of Ca^{2+} into the cytosol from the SR, the continuous leak of Ca^{2+} into the cytosol from the SR, and the pumping of Ca^{2+} back into the SR from the cytosol. The terms $J_{LCC}, J_{m_{leak}}$, and $J_{m_{pump}}$ describe the fluxes of calcium into and out of the cell via the plasma membrane. The coupling between (2.3.1) and (2.3.3) is achieved by the three flux terms shared by both equations.

More precisely, $J_{LCC}, J_{m_{leak}}$, and $J_{m_{pump}}$ in (2.3.1) describe the fluxes of calcium into and out of the cell via the plasma membrane. J_{pump} replenishes the calcium stores in the SR; it increases SR calcium concentration by decreasing cytosol calcium concentration. J_{leak} is a continuous leakage of those SR calcium stores into the cytosol; it increases cytosol concentration by decreasing SR calcium concentration. The pump term

$$J_{pump}(c) \;=\; V_{pump} \left(\frac{c^{n_{pump}}}{K_{pump}^{n_{pump}} + c^{n_{pump}}} \right) \tag{2.3.7}$$

is thus a function of cytosol calcium $c(\mathbf{x}, t)$. The leak term J_{leak} is a constant defined by

$$J_{leak} \;=\; J_{pump}(c_0), \tag{2.3.8}$$

which balances $J_{pump}(c)$ at basal level $c_0 = 0.1\ \mu$M of cytosol calcium. The pump term J_{pump}, a function of cytosolic calcium $c(\mathbf{x}, t)$, consists of the maximum pump velocity V_{pump} multiplied against the relationship between $c(\mathbf{x}, t)$ and the pump sensitivity K_{pump}; the exponent n_{pump} refers to the Hill coefficient (quantifying the degree of co-operative binding) for the pump function. This has the practical effect of multiplying the maximum possible pump velocity against a number between 0 and 1, exclusive. J_{leak}, which continuously leaks calcium into the cytosol from the SR, is simply J_{pump} evaluated at the basal cytosolic calcium concentration $c_0 = 0.1\,\mu M$. As noted, J_{pump} balances J_{leak} in the absence of sparking. It can also balance J_{CRU} under conditions of active calcium release.

The term J_{CRU} in (2.3.1) is the Ca^{2+} flux into the cytosol from the SR via each individual point source at which a CRU has been assigned. The effect of all CRUs is modeled as a superposition such that

$$J_{CRU}(c, s, \mathbf{x}, t) \;=\; \sum_{\hat{\mathbf{x}} \in \Omega_s} \hat{\sigma}\, \tfrac{s-c}{s_0-c_0}\, \mathcal{O}(c, s)\, \delta(\mathbf{x} - \hat{\mathbf{x}}) \tag{2.3.9}$$

with

$$\mathcal{O}(c, s) \;=\; \begin{cases} 1 & \text{if } u_{rand} \leq J_{prob}, \\ 0 & \text{if } u_{rand} > J_{prob}, \end{cases} \tag{2.3.10}$$

where

$$J_{prob}(c, s) = P_{max} \Big(\frac{c^{n_{prob}}}{K_{prob_c}^{n_{prob}} + c^{n_{prob}}} \Big) \Big(\frac{s^{n_{prob}}}{K_{prob_s}^{n_{prob}} + s^{n_{prob}}} \Big). \tag{2.3.11}$$

Here, the effect of each CRU is modeled as a product of three terms: (i) Similarly to how in J_{pump} the maximum pump rate is scaled against the concentration of available

cytosol calcium, the maximum rate of Ca^{2+} release $\hat{\sigma}$ is scaled here against the ratios of the difference of calcium concentrations in the cytosol and in the SR. (ii) Following the same pattern a maximum value multiplied against some scaling proportion between 0 and 1 the gating function $\mathcal{O}$ has the practical effect of "budgeting" the calcium SR stores such that when the stores are low, the given CRU becomes much less likely to open; each CRU is assigned a uniformly distributed random value, which is compared to the single value returned by the CRU opening probability J_{prob} to determine whether or not the given CRU will open. (iii) The Dirac delta distribution $\delta(\mathbf{x} - \hat{\mathbf{x}})$ models each CRU as a point source for calcium release.

2.3.2 Electrical excitation

The membrane potential of the cell depends on both the cytosol calcium ion concentration and also on the fraction of open K^+ channels. [4,38]. While a complete description of the relationship between electrolytes and membrane potential is beyond the scope of this chapter, note the ω term in (2.3.12), an addition to our model which introduces a dependency on c to complete the coupling between the electrical and chemical systems. Table 2.3.2 collects the variables and parameters for electrical excitation.

The Ca^{2+} gating dynamics are much faster than the K^+ gating dynamics, so the calcium conductance can be approximated as m_∞ or instantaneously steady-state at all times; the potassium conductance requires a separate description in (2.3.13)

$$\begin{aligned} \frac{\partial V}{\partial t} &= \tau \frac{1}{C}\Big(I_{\text{app}} - g_L(V - V_L) - g_{Ca}\, m_\infty(V)\,(V - V_{Ca}) \\ &\qquad - g_K\, n\,(V - V_K) + \omega\,(J_{m_{pump}} - J_{m_{leak}})\Big), \end{aligned} \tag{2.3.12}$$

$$\frac{\partial n}{\partial t} = \tau\, \lambda_n(V)\,[n_\infty(V) - n]. \tag{2.3.13}$$

The connection between (2.3.1) and (2.3.12), link ① in Figure 2.1.1, the link from the electrical system to the calcium system, comes through J_{LCC}, the only calcium flux

term to involve voltage. Note the parameter κ, which is an external scaling factor for J_{LCC}, if the value of κ is set to 0, the connection, link ① in Figure 2.1.1, is effectively switched off and the calcium dynamics are then modeled as though voltage were not involved

$$J_{LCC} = \frac{S\, g_{Ca}\, m_{\infty}\, (V - V_{Ca})}{2F}. \tag{2.3.14}$$

The surface area, S, of the cell is included in light of the fact that J_{LCC} describes the influx of calcium through L-type calcium channels (LCCs), which are present in the enclosing plasma membrane of the cell: the surface area of the cell is the surface area of the membrane.

To incorporate the feed-forward link between the calcium and electrical systems we consider a simplified approach. A significant component in electrical handling with high cytosolic calcium is the sodium-calcium exchanger current (NCX) that generates an inward depolarizing current affecting the action potential while removing calcium from the cell. Since the pump mechanism is already available in the model, we tie the calcium efflux as $(J_{m_{pump}} - J_{m_{leak}})$ to a depolarizing current. This enables us to capture the net effect of depolarization from the NCX via a simple approach. A new term appears in (2.3.12) with $\omega\,(J_{m_{pump}} - J_{m_{leak}})$ where $\omega > 0$ represents the feedback strength of the effect. Modification of this approach to include the NCX directly or modifying J_{LCC} inactivation to be dependent on the calcium in the cell are future work.

The individual components of the calcium efflux term have the same form as the earlier J_{pump} and J_{leak} functions in (2.3.7) and (2.3.8), respectively. As J_{pump} described the removal of calcium from the cytosol and its transfer into SR stores,

$$J_{m_{pump}}(c) \;=\; V_{m_{pump}}\Big(\frac{c^{m_{n_{pump}}}}{K_{m_{pump}}^{m_{n_{pump}}} + c^{m_{n_{pump}}}}\Big) \tag{2.3.15}$$

describes the removal of calcium from the cytosol and its transfer to outside the cell

across the plasma-membrane. The leak term J_{leak} described a passive leak of calcium into the cytosol from the SR, while J_{CRU} described an abrupt, high-concentration (high relative to the leak) release of calcium into the cytosol from the SR. Similarly,

$$J_{m_{leak}} = J_{m_{pump}}(c_0) \tag{2.3.16}$$

describes a passive leak of calcium into the cytosol from outside the cell across the plasma membrane, while J_{LCC} describes a voltage-dependent influx of calcium release into the cytosol via the LCCs.

The model now connects the chemical system to the electrical system, link ② in Figure 2.1.1, via the inclusion of the current generated by calcium leaving the cell via $J_{m_{pump}}$ and $J_{m_{leak}}$, which directly affects the voltage. We collect and incorporate these as a single term, the calcium efflux ($J_{m_{pump}} - J_{m_{leak}}$), and introduce ω as a parameter for feedback strength in link ② in Figure 2.1.1, which is a scaling factor with the same essential function as κ in link ① in Figure 2.1.1 from (2.3.1): if it is set to 0, the only terms of (2.3.12) which depend on the cytosolic calcium concentration drop out, and the connection from calcium signaling to electrical excitation is severed.

2.3.3 Pseudo-mechanical contraction

We complete the proposed links of the model, ③ and ④ in Figure 2.1.1, by introducing feedback and feedforward terms for the contractile dynamics. We describe this as "pseudo-mechanical" because the domain itself is unchanged; in our model, the physical dimensions of the cell and the locations of the CRUs do not alter. We instead model the contraction via the proportion of contractile proteins which have bound to calcium and changed shape as a result, which generates the force required for cell contraction. The main effect of this in this model is on the ability for calcium to release from troponin during contraction. Table 2.3.3 collects the variables and parameters for pseudo-mechanical contraction.

Table 2.3.2 Variables and parameters for electrical excitation: gating functions and membrane potential.

Variable	Definition	Values/Units
$V(\mathbf{x},t)$	membrane potential (voltage)	mV
τ	scaling factor to fit action potential duration	0.1 μM μm^3/ms
V_L	equilibrium potential for leak conductance	-50 mV
V_{Ca}	equilibrium potential for Ca^{2+} conductance	100 mV
V_K	equilibrium potential for K^+ conductance	-70 mV
C	membrane capacitance	20 μF/cm^2
I_{app}	applied current	10 μA/cm^2
g_L	maximum/instantaneous conductance for leak	2 mmho/cm^2
g_{Ca}	max./instantaneous conductance for Ca^{2+}	4 mmho/cm^2
g_K	max./instantaneous conductance for K^+	8 mmho/cm^2
m_∞	fraction of open calcium channels at steady state	0 to 1
$n(\mathbf{x},t)$	fraction of open potassium channels	0 to 1
n_∞	fraction of open potassium channels at steady state	1
$\lambda_n(V)$	rate constant for opening of K^+ channels	s^{-1}
J_{LCC}	influx of calcium into cell via L-type calcium channels	μM/ms
S	surface area of the cell	3604.48 μm
F	Faraday constant	95484.56 C/mol
κ	scaling factor of J_{LCC}	0.01
ω	feedback strength (scaling factor) for Ca^{2+} efflux	μA ms/μM cm^2
$J_{m_{pump}}$	pump of calcium out from cell via L-type calcium channels	μM/ms
$J_{m_{leak}}$	leak of calcium out from cell via L-type calcium channels	μM/ms
$V_{m_{pump}}$	maximum pump rate	1 μM/ms
$m_{n_{pump}}$	membrane pump Hill coefficient	2
$K_{m_{pump}}$	membrane pump sensitivity	0.18

The contractile proteins in question, though considered as a single species, are the combination of actin and myosin when linked via cross-bridges. This linkage is made possible by Ca^{2+} binding to troponin, the cytosol buffer species $b_2^{(c)}(\mathbf{x},t)$: it is this binding that allows the actin-myosin cross-bridges to form. We therefore introduce a new cytosol species, $b_3^{(c)}(\mathbf{x},t)$, to describe these actin-myosin cross-bridges, and construct a third cytosol reaction term:

$$R_{b_3}^{(c)} \;=\; -\,k^{+}_{b_3^{(c)}} \left(\frac{b_{2,total}^{(c)} - b_2^{(c)}}{b_{2,total}^{(c)}} \right)^2 b_3^{(c)} + k^{-}_{b_3^{(c)}} \left(b_{3,total}^{(c)} - b_3^{(c)} \right). \tag{2.3.17}$$

Notice that this is not the same as the generic pattern for buffer species reaction terms from the initial model. There is no immediately clear dependence on cytosolic calcium $c(\mathbf{x},t)$. However, while $c(\mathbf{x},t)$ is not explicitly included, it is present in the proportion involving troponin, $b_2^{(c)}(\mathbf{x},t)$, which itself depends explicitly on cytosol calcium levels; $R_{b_3}^{(c)}$, like the other two reaction equations, does in fact depend on cytosol calcium concentration.

We modify the reaction equation for troponin as well. When troponin binds to Ca^{2+}, the protein as a whole, as noted, changes shape: not only does this allow actin-myosin cross-bridges to form, but it also traps the calcium in its connection to the troponin so that the disassociation rate decreases dramatically. To account for this, we add a shortening factor ε to describe how the separation of troponin and calcium has been physically, not chemically, impaired. Note, again, that $R_{b_2}^{(c)}$ remains a function of cytosol calcium concentration $c(\mathbf{x},t)$:

$$R_{b_2}^{(c)} \;=\; -\,k^{+}_{b_2^{(c)}}\, c\, b_2^{(c)} + k^{-}_{b_2^{(c)}} \left(b_{2,total}^{(c)} - b_2^{(c)} \right) \frac{1}{\varepsilon} \tag{2.3.18}$$

with

$$\varepsilon \;=\; \exp\left(F_{max}\, k_s \left(\frac{b_{3,total}^{(c)} - b_3^{(c)} - [XB]_0}{b_{3,total}^{(c)} - [XB]_0} \right) \right). \tag{2.3.19}$$

Table 2.3.3 Variables and parameters for mechanical contraction: new cytosol species reactions.

Variable	Definition	Values/Units
$b_3^{(c)}(\mathbf{x},t)$	inactive actin-myosin cross-bridges [X]	μM
$[XB]$	active (linked) actin-myosin cross-bridges	μM
$[XB]_0$	initial concentration of active cross-bridges	μM
$k^+_{b_3^{(c)}}$	forward reaction coefficient for $b_3^{(c)}(\mathbf{x},t)$, actin-myosin cross-bridges	0.04 ms^{-1}
$k^-_{b_3^{(c)}}$	reverse reaction coefficient for $b_3^{(c)}(\mathbf{x},t)$, actin-myosin cross-bridges	0.01 ms^{-1}
ε	shortening factor	0 to 1
k_s	stiffness of actin filament	0.025 N/m
F_{max}	maximum force generated by actin-myosin crossbridges	120 μN

This shortening factor ε links ③ and ④ in Figure 2.1.1. It refers back to the concentration of $b_3^{(c)}(\mathbf{x},t)$, the actin-myosin cross-bridges, and to the force that their linkage generates. It is scaled by the maximum possible contractile force F_{max}, the actin stiffness k_s, and the proportion of active to inactive actin-myosin cross-bridges. Like ω and κ, ε is our point of control over the linkage between systems: if the exponent is 0, the overall value simply turns to 1, and $R_{b_2}^{(c)}$ reverts to its earlier form (2.3.5).

The addition of these two reaction terms connects the last two components of our model. The calcium signaling is linked to the pseudo-mechanical contraction through the cross-bridge term, and the pseudo-mechanical contraction is in turn connected to the calcium signaling through the inclusion of the cytosol calcium concentration in the modified reaction equation for troponin. Thus all links ①, ②, ③, and ④ in Figure 2.1.1 are established, and thus the electrical excitation and mechanical contraction systems have complete links to the calcium signaling. While it would be possible to consider linking the mechanical contraction to the electrical excitation mechanisms through stretch activated channels we ignore those currently and consider their addition possible future work.

Table 2.4.1 Sizing study with $n_s = 6$ species using double precision arithmetic, listing the mesh resolution $N_x \times N_y \times N_z$, the number of control volumes $N = (N_x+1)(N_y+1)(N_z+1)$, the number of degrees of freedom (DOF) $n_{eq} = n_s N$, the number of time steps taken by the ODE solver, and the predicted and observed memory usage in GB for a one-process run.

Resolution	N	DOF n_{eq}	number of time steps	memory usage (GB) predicted	observed
$16 \times 16 \times 64$	18,785	112,710	32,047	0.01	0.25
$32 \times 32 \times 128$	140,481	842,886	43,473	0.11	0.64
$64 \times 64 \times 256$	1,085,825	6,514,950	64,843	0.83	2.17
$128 \times 128 \times 512$	8,536,833	51,220,998	160,798	6.49	8.44

2.4 Numerical Method

In order to do calculations for the CICR model, we need to solve a system of time-dependent parabolic partial differential equations (PDEs). These PDEs are coupled by several non-linear reaction and source terms. For the simulations in Section 2.5, we need six species, thus we have $n_s = 6$ coupled PDEs. The domain in our model is a hexahedron with isotropic CRU distribution as seen in Figure 2.1.2. Taking a method of lines (MOL) approach to spatially discretize this model, we use the finite volume method (FVM) as the spatial discretization, with $N = (N_x + 1)(N_y + 1)(N_z + 1)$ control volumes. Applying this to the case of the n_s PDEs results in a large system of ordinary differential equations (ODEs). A MOL discretization of a diffusion-reaction equations with second-order spatial derivatives results in a stiff ODE system. The time step size restrictions, due to the CFL condition, are considered too severe to allow for explicit time-stepping methods. This necessitates the use of a sophisticated ODE solver such as the family of numerical differentiation formulas (NDFk). Our stiff ODEs, which needs to use an implicit ODE method, require the solution of a non-linear system. We use Newton's Method as the non-linear solver, and at each

Newton step we use the biconjugate gradient stabilized method (BiCGSTAB) as the linear solver. Complete details of the numerical method can be found in [22, 46].

The implementation of this model is done in C using MPI to parallelize computations. Parallelization is accomplished through block-distribution all large arrays to all MPI processes. We split of the mesh in the z-direction with one subdomain on each of the parallel processes. MPI commands such as `MPI_Isend` and `MPI_Irecv`, which are non-blocking point-to-point communication commands, send messages between neighboring processes. The collective command `MPI_Allreduce` is used for the computation of scalar products and norms.

The spatial discretization of the application problem using the total number of species, $n_s = 6$, and the finite volume method with N control volumes results in a system of non-linear ordinary differential equations (ODEs) with $n_{eq} = n_s\, N$ degrees of freedom (DOF). Table 2.4.1 shows the number of degrees of freedom for different mesh sizes for this problem. Simulation times depend heavily on the number of time steps taken. For each of the mesh sizes, the total number of time steps is listed. Note that the number of time steps increases for finer meshes, but the increase in not as large at the increase in DOF. We are using a matrix-free method that minimizes memory usage by not storing any system matrix; the code with the NDFk method of orders $1 \leq k \leq 5$ requires then, including all auxiliary method vectors, the storage of only 17 arrays of significant size n_{eq}. Table 2.4.1 also shows predicted memory usage for the 6 species simulation on each mesh as well as observed total memory used. The predicted memory is a realistic underestimate of the total memory observation.

2.5 Results

In this work, we focus on the electrical excitation and calcium signaling link impacts, that is, the simulations presented here do not include the mechanical con-

Table 2.5.1 Selected parameter sets for our three base behavior test cases from studies in [1]. Common parameters across the three cases are $K_{prob_s} = 550$ μM, $V_{pump} = 2$ μM/ms, and SR diffusion $D_s = 0.78$ μm^2/ms.

Case	K_{prob_c}	SR load (s_0)	$\hat{\sigma}$
Case A: Sparking case	15 μM	2000 μM μm^3/ms	200 μM
Case B: Wave case	10 μM	5000 μM μm^3/ms	150 μM
Case C: Blowup case	5 μM	10000 μM μm^3/ms	200 μM

traction components of the model described above. Simulations necessary for this rely on six species: calcium in the cytosol $c(\mathbf{x}, t)$, a florescent dye $b_1^{(c)}(\mathbf{x}, t)$, troponin $b_2^{(c)}(\mathbf{x}, t)$, calcium in the SR $s(\mathbf{x}, t)$, voltage $V(\mathbf{x}, t)$, and the fraction of open K+ channels $n(\mathbf{x}, t)$. Thus, the $n_s = 6$ PDEs of the model are (2.3.1), (2.3.2) with $n_{sc} = 2$, (2.3.3), (2.3.12), and (2.3.13) as described in Section 2.3. We consider for this the domain $\Omega = (-6.4, 6.4) \times (-6.4, 6.4) \times (-32.0, 32.0)$ in Figure 2.1.2 that captures the key feature of the elongated shape of a heart cell. With the physiological constants $\Delta x_s = 0.8$, $\Delta y_s = 0.8$, and $\Delta z_s = 2.0$ for the CRU spacings, we have therefore a CRU lattice of $15 \times 15 \times 31 = 6{,}975$ CRUs throughout the interior of the cell.

When examining the calcium behavior in the cytosol this model has demonstrated three main behaviors: sparking, wave, and blowup. Sparking is the behavior in which only small, localized release of calcium occurs. That is, there is no propagation or build up of calcium in the cell, only sparks. For certain parameter values several sparks may organize and initiate a wave of calcium release. So called blowup occurs when the cell becomes flooded with calcium and does not recover to basal levels through the final time.

Studies in [1] established which of the calcium behaviors result under different parameter sets in relation to electrical excitation. We start by selecting a set of

parameters that resulted in each of the three primary behaviors, Table 2.5.1. Recall here that κ in (2.3.1) turns on the link ① in Figure 2.1.1 from the electrical excitation to the calcium signaling system.

To visualize the solutions, we present five different types of plots: CRU plots, isosurface plots, line scans, voltage plots and SR plots. Each of these plots displays different information in relation to the calcium dynamics within the system. CRU plots show the open calcium release units, which are represented by blue dots in the domain. Isosurface plots show the concentration of Ca^{2+} in the cytosol. The color blue represents locations at which the Ca^{2+} concentration is at least 65 μM, indicating the presence of more than trace amounts of calcium. Higher concentrations on the boundary are indicated by a yellow–red color palatte. Line-scans are produced by tracking the concentration of the cytosolic Ca^{2+} concentration along the center of the axis in the longitudinal direction of the cell at each millisecond. The concentrations of Ca^{2+} are plotted on a two-dimensional domain versus time, and then overlayed upon each other producing the final image. Higher concentrations of Ca^{2+} are indicated by red, while lower concentrations are indicated by blue. Voltage plots track the voltage at the center of the cell domain, and then is plotted versus time. SR plots show the concentration of Ca^{2+} in the SR as it relates to the right, center, and left of the cell.

In Section 2.5.1, we present results of one-way coupling between electrical excitation and calcium signaling with an enabled membrane pump. One-way coupling is achieved by only considering the forward connection between the electrical and calcium systems, link ① in Figure 2.1.1, via non-zero κ. In Section 2.5.2, we present results of a two-way coupling between the calcium and electrical systems, link ① and link ②. In the two-way coupling, we add in the feedback connection between the electrical and calcium systems. As described in Section 2.3, two-way coupling is mathematically achieved through the inclusion of a calcium efflux term, scaled by the

factor ω, referred to as a feedback strength coefficient. We present a parameter study of ω in which we examine how varying the value of ω impacts the behavior of our solutions.

2.5.1 Electrical Excitation to Calcium Signaling: One-Way Coupling

The first set of simulations run with the parameters set as in Table 2.5.1 use only the coupling from electrical excitation to calcium signaling. That is, $\kappa = 0.01$ and $\omega = 0$, so that the feedback from calcium signaling to electrical excitation is off. The change here from [1] is the value of $V_{m_{pump}}$ in (2.3.15). The pump of calcium leaving the cell was disabled by setting $V_{m_{pump}} = 0$ μM/ms in [1], but we introduce the effect of turning on this pump by $V_{m_{pump}} = 1$ μM/ms . The resulting behaviors, as observed, are not exactly of the basic three types. These observed behaviors become the base test for our parameter study in ω for the effect of turning on the feedback from calcium signaling to electrical excitation.

The sparking case presents the sparking behavior as before, with more responsiveness to the voltage. In the wave case as shown in Figure 4.1(a) of [1] the linescans show 'V' patterns as calcium propagates through the cytosol over time. With $V_{m_{pump}} = 1$ we see no such action as too much calcium is pumped out of the cell for propagation from calcium induced calcium release to occur. Instead, we see only a high concentration of calcium sparking around two instances in time, presented in the wave case.The behavior in the blowup case is characterized as blowup, but the nature of the blowup is different with $V_{m_{pump}} = 1$. Blowup cases were those in which the cell became flooded with calcium and maintained that high concentration of calcium until the final time. In our blowup case we observe the cell floods with calcium, which then leaks out of the cell and another flooding of calcium into the cytosol occurs. This pattern continues a few times over before the SR calcium store is so depleted that no

significant amount of calcium is released into the cell when a CRU opens.

Case A: Sparking Case ($\omega = 0$)

The first behavior we present is sparking under the parameters in Case A of Table 2.5.1. Figures 2.5.1 and 2.5.2 show CRU plots and isosurface plots at a sampling of times over the course of this simulation. At the specific time the CRU plot indicates each open CRU with a single blue dot. The isosurface plot shows the concentration of calcium in the cytosol. In this plot all of the observed concentrations of calcium are very low,

Figure 2.5.3 shows the line scan for this sparking behavior. The scale of this line scan is very sensitive to any release of calcium so that the sparking behavior can be observed, $0-5\,\mu M$. The scale will be much less sensitive to low calcium concentrations in the blowup case, $0-100\,\mu M$, so that only high levels of calcium are observed. Note there are two places in which more sparking occurs. The voltage plot shows that the spikes in the voltage correspond to these highest times of calcium release, which is a physiologically realistic behavior.

In Figure 2.5.1, the number of CRUs opening increases from 100 ms to 400 ms where at 500 ms the number of open CRUs decreases considerably. This corresponds direction with Figure 2.5.2 where the cytosolic calcium concentration is also at its greatest point in the same timeframe. However the calcium does not "buildup" as is seen at 500 ms by a much lower number open CRUs and a lower concentration of Ca^{2+}. The accompanying linescan in Figure 2.5.3 supports these observations as the cyan color, indicative of heavy sparking, can be seen in the 400 ms region but not in the 500 ms region. The concentration behavior repeats itself from 600 ms to 900 ms in Figure 2.5.1 (CRU) and Figure 2.5.2 (ISO). This repetition can also be seen in the linescan in Figure 2.5.3 during those timeframes.

CRU plots

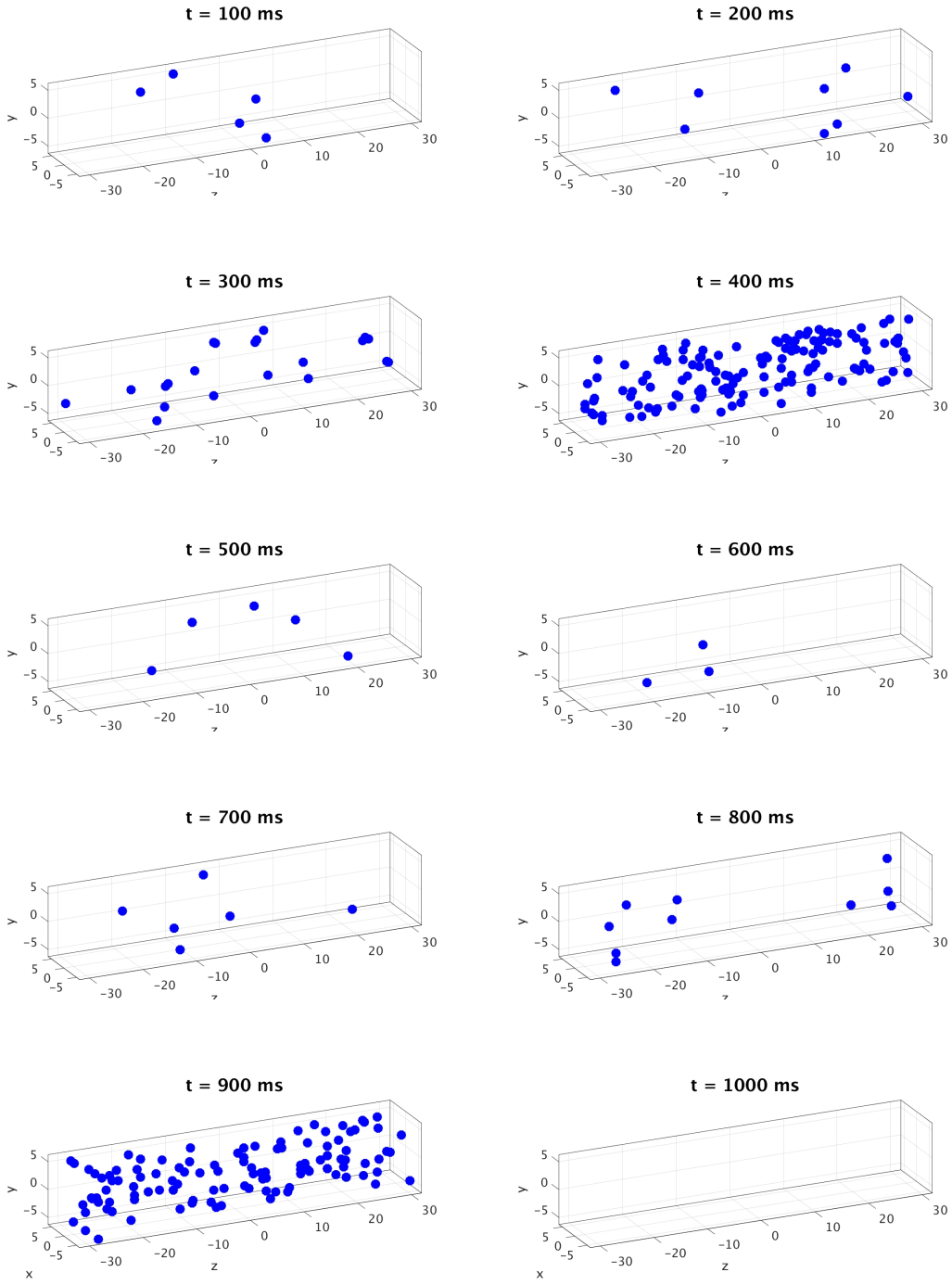


Figure 2.5.1 CRU plots for sparking case with $\omega = 0$ and other parameters from Case A of Table 2.5.1.

Isosurface plots

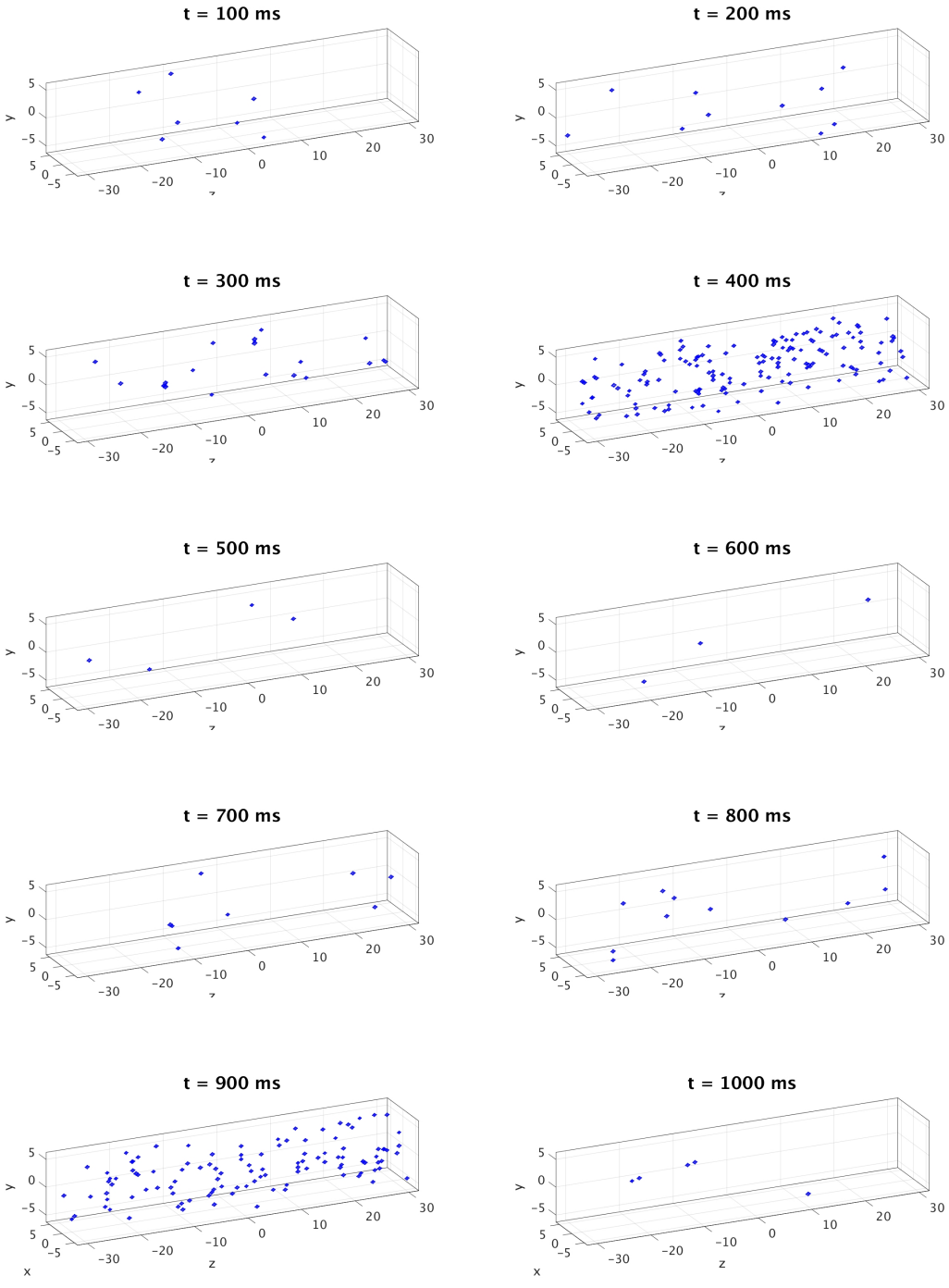


Figure 2.5.2 Isosurface plots for sparking case with $\omega = 0$ and other parameters from Case A of Table 2.5.1.

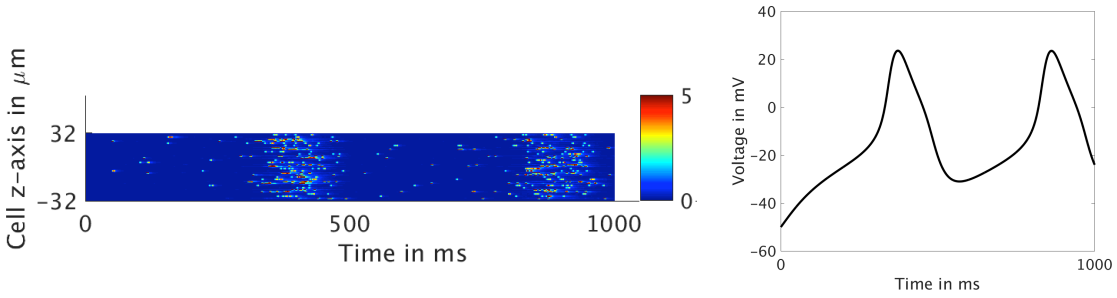


Figure 2.5.3 Line scan and voltage plot for sparking case with $\omega = 0$ and other parameters from Case A of Table 2.5.1.

Case B: Wave Case ($\omega = 0$)

The second behavior observed is extremely heavy sparking, rather than a calcium induced propagation of a wave through the cell under the parameters from Case B of Table 2.5.1. In fact, we observe what could be considered a very light wave of calcium in the line scan of Figure 2.5.6.

In Figures 2.5.4 and 2.5.5, we see similar behavior as in Figures 2.5.1 and 2.5.2, but with higher numbers of open CRUs and higher concentrations of calcium at the corresponding times. We do not see any very high concentrations of calcium, but we readily observe the high number of CRUs open around 400 ms and 900 ms and the higher concentrations of calcium at those times when compared to the sparking case. When looking at the linescan in Figure 2.5.6, it can be shown that the dark red regions correspond to the heavy sparking times that can be observed in Figures 2.5.4 and 2.5.5 with the ISO and the CRU plots. The scale of this line scan is the same as in the sparking case, very sensitive to any release of calcium so that the behavior can be observed, $0 - 5\,\mu M$. These clearly correspond with the spikes in voltage from the electrical excitation demonstrating a good connection between the systems.

CRU plots

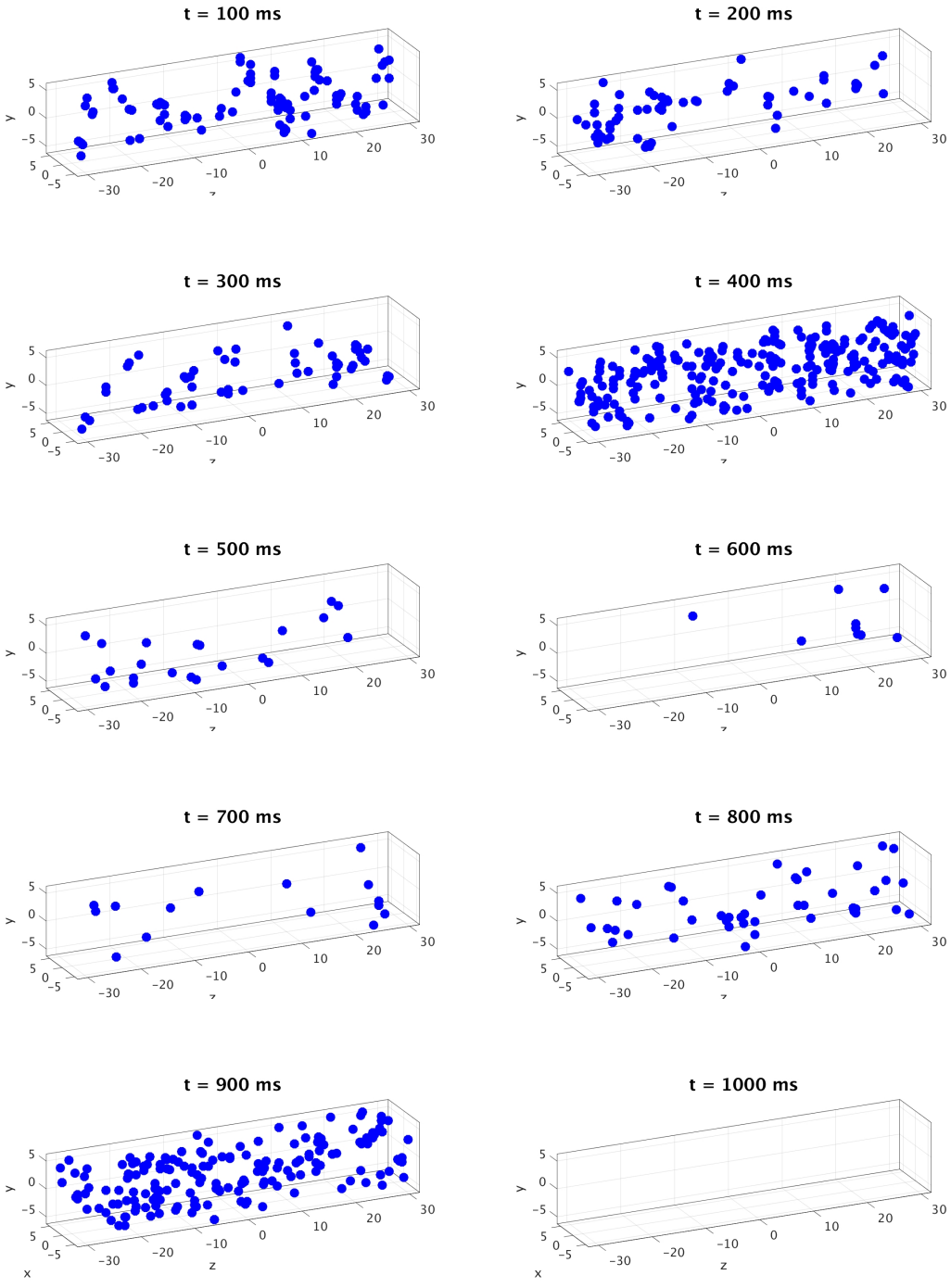


Figure 2.5.4 CRU plots for wave case with $\omega = 0$ and other parameters from Case B of Table 2.5.1.

Isosurface plots

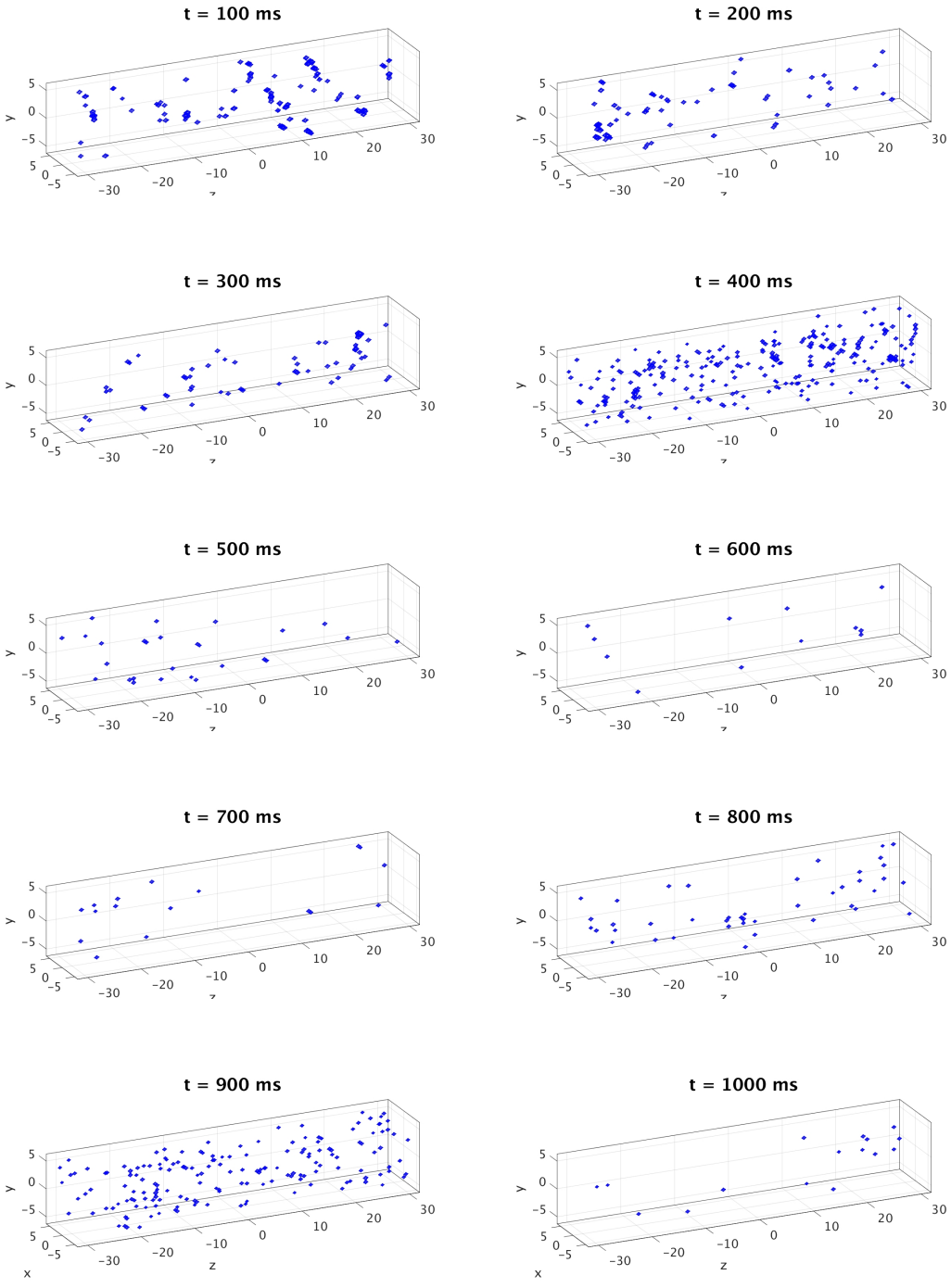


Figure 2.5.5 Isosurface plots for wave case with $\omega = 0$ and other parameters from Case B of Table 2.5.1.

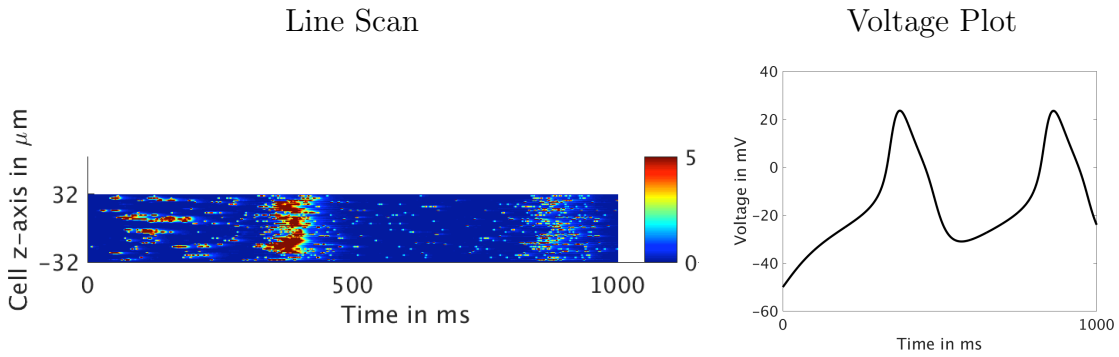


Figure 2.5.6 Line scan and voltage plot for the wave case with $\omega = 0$ and other parameters from Case B of Table 2.5.1.

Case C: Blowup Case ($\omega = 0$)

The final case presented here is the blowup case under the parameters in Case C of Table 2.5.1. To reiterate: a blowup case is where the concentration of cytosolic calcium is high early on, at $t = 50$ ms causing many CRUs to open as seen in Figure 2.5.7 at $t = 125$ ms. Then the behavior is repeated in the later images inside Figure 2.5.7. We consider a high level of calcium to occur when the average cytosolic calcium concentration at some point reaches above 5 μ M. The definition of a blowup that was provided does not seem to fit the behavior of this current case. While the concentration of cytosolic calcium is initially, and physiologically, too high it does not however stay high the entire simulation due to the enabling of the membrane pump. Instead the concentration is high for a few milliseconds but goes down to basal levels and then repeats in a periodic fashion. This behavior can be readily seen in the linescan in Figure 2.5.8, scaled from 0 to 100 μ M. It is even improper that, along with high concentrations, the spike in calcium is not in line with a spike in voltage. This is due to the spontaneous calcium release coupling with the 100 ms refactory period. The voltage should be introducing additional calcium into the system which should only increase the severity of the blowup but there does not appear to be any signs in

the linescan that this is happening. The explanation is that the concentration did not stay high for the entire simulation because release of calcium through the membrane with $V_{m_{pump}}$ turned on, was so severe that the SR was completely depleted of calcium leaving none left to flow into the the cytosol.

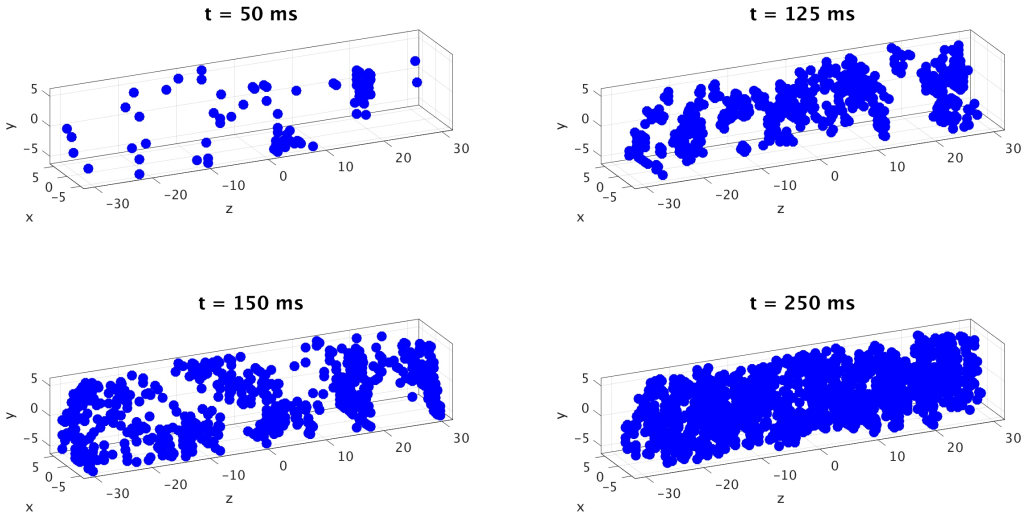
2.5.2 Electrical Excitation and Calcium Signaling: Two-Way Coupling

Now that we have demonstrated behaviors in the presence of the one-way coupling from electrical excitation to calcium signaling we consider cases in which two-way coupling is present. We turn on the coupling from calcium signaling to electrical excitation by setting $\omega > 0$. For each of the parameter sets that generated the behaviors in Section 2.5.1 we consider four cases of ω to test the feedback strength from calcium signaling and electrical excitation: $\omega = 10, 30, 50,$ and 100. In this study, we do not change the strength of the connection from electrical excitation to calcium signaling. This is certainly something that would be of interest in future study.

Case A: Sparking Case with $\omega > 0$

Under the same parameters as the sparking case in Section 2.5.1 , we present simulations with $\omega = 10, 30$ in Figure 2.5.9 and $\omega = 50, 100$ in Figure 2.5.10. We observed that for $\omega < 10$ the same general behavior as the case $\omega = 10$ and that of $\omega = 0$ from Figure 2.5.3. In the cases in which $\omega \leq 10$, we observe that the linescans and voltage plots are synchronized, the two peaks in voltage match up with the heavy concentration of sparking in the line scan around 400 ms and 900 ms. The SR plot of the concentration of calcium ions in the SR also shows the synchronization with this behavior as the SR calcium concentration dips at these times, appropriately, as the release of calcium ions from the SR into the cytosol is exactly the higher concentration

CRU plots



Isosurface plots

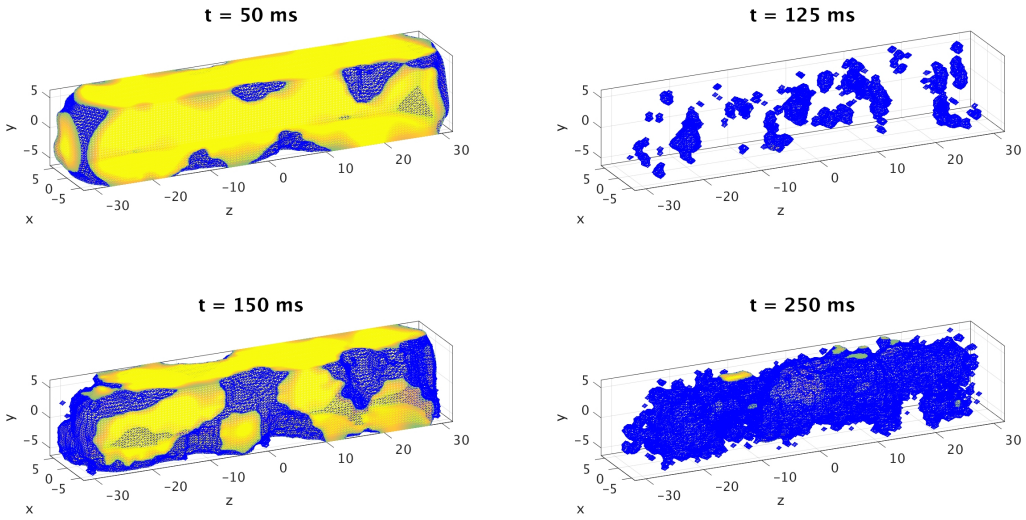


Figure 2.5.7 CRU plots and isosurface plots for blowup case with $\omega = 0$ and other parameters from Case C of Table 2.5.1.

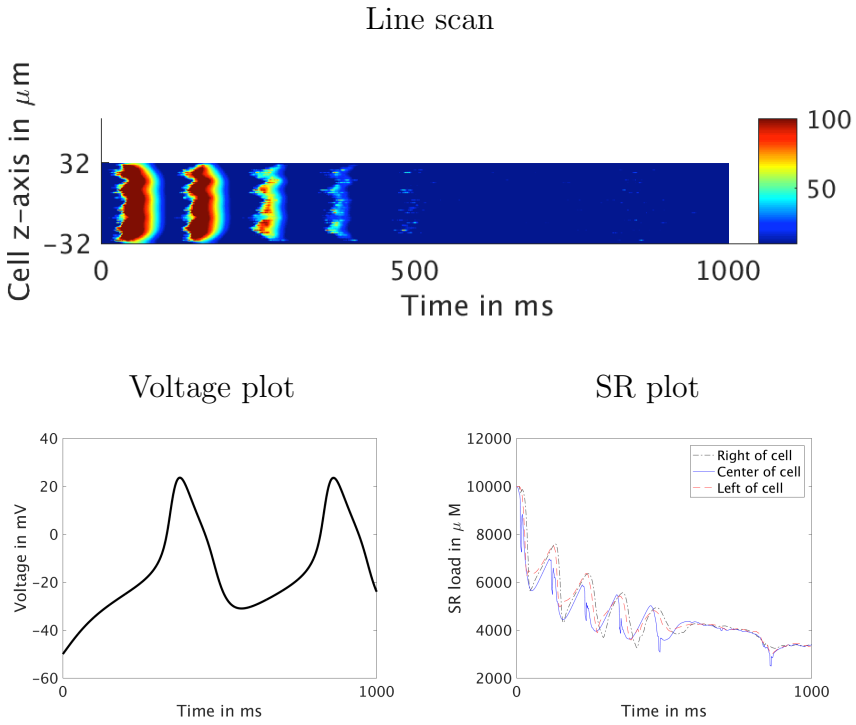


Figure 2.5.8 Line scan, voltage plot, and SR plot for the blowup case with $\omega = 0$ and other parameters from Case C of Table 2.5.1.

of calcium ions in the cytosol we see in the line scan.

For $\omega = 30$ we present CRU plots in Figure 2.5.11 and isosurface plots in Figure 2.5.12. This case is chosen as the voltage plot has a nicer attunement for the changed periodicity in voltage and the linescans for each case look relatively the same with nothing more than a light peppering of color to represent minor sparking.

As we make ω larger, we continue to observe differences in the behavior of the average voltage. In particular, for $\omega = 50$ we observe a third peak in the voltage, which is not matched by a corresponding dip in the SR or by a higher concentration of cytosol calcium sparking shown in the line scan. In this sparking case, we do not see the same depletion over time of the calcium store in the SR as we did with the wave and blowup cases. This indicates there is not a significant amount of calcium ions that are leaving the cell. For that reason, the sparking case is promising as a point of further study. Despite the clear differences in the voltage plots amongst all the ω values, the linescans remain more or less unchanged. It should be observed that $\omega = 50$ having more peaks than $\omega = 10$ would result in more calcium entering through the excited LCC but this is not observed in the linescans present in Figure 2.5.9 and Figure 2.5.10. We believe this can be that amount of calcium entering through the LCC is negligible when compared to the amount removed through the ω term and pumped into the cytosol by the SR and CRUs.

Case B: Wave Case with $\omega > 0$

Similar to what happened in the wave case of Section 2.5.1, the wave case here is also consistently heavy sparking when looking at all the values of ω. As expected there are several drastic changes in voltage periodicity as ω begins to become much larger. The baseline two peak seen when $\omega = 10$, in Figure 2.5.13, becomes less regular when $\omega = 100$ in Figure 2.5.14. Unlike in the most recent sparking case,

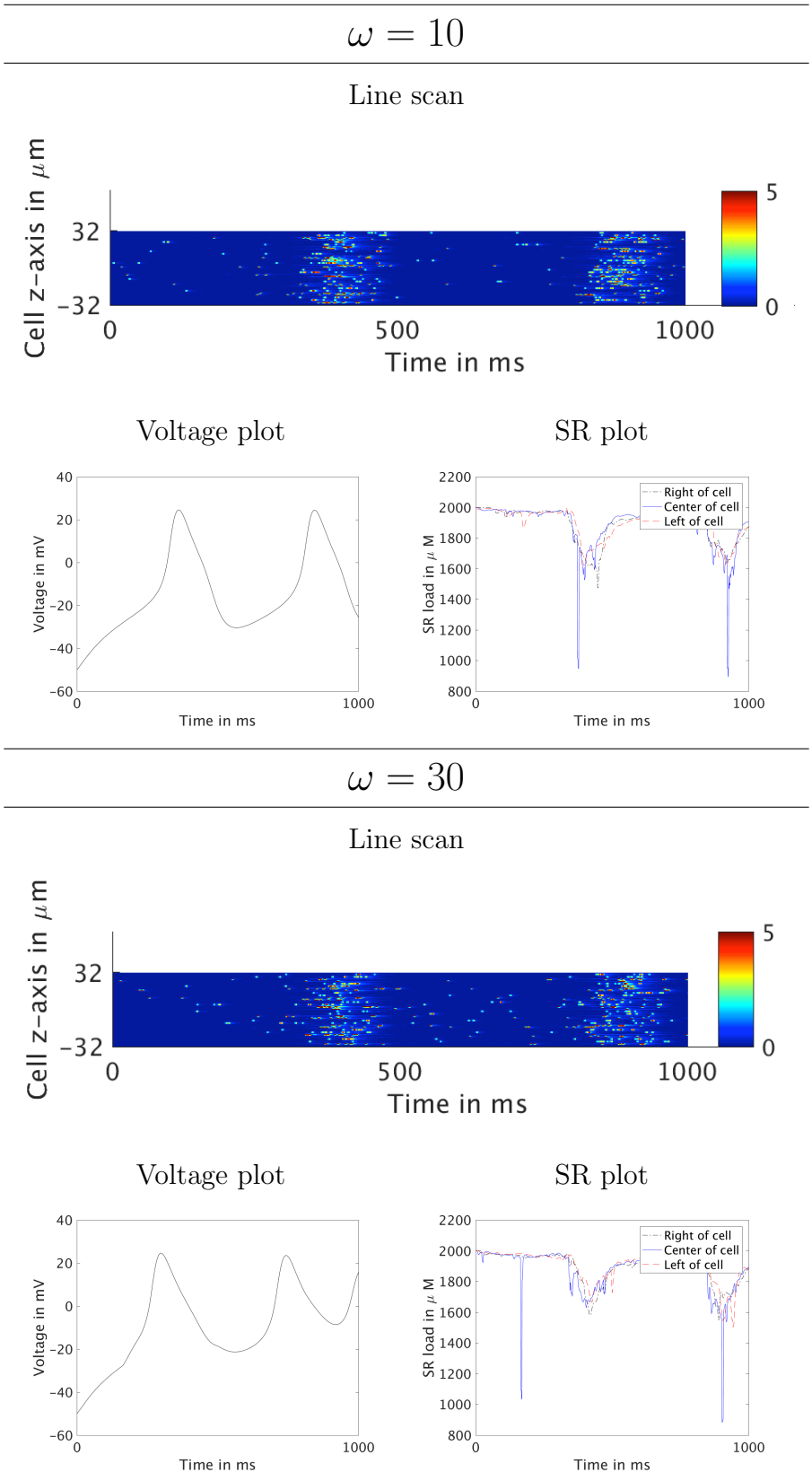
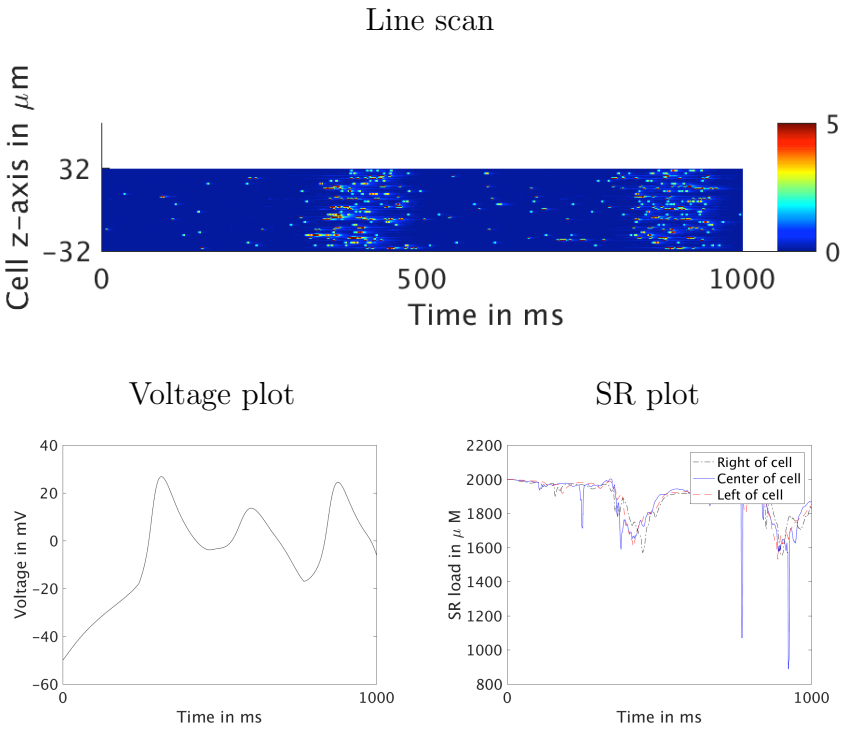


Figure 2.5.9 Line Scans, Voltage Plots, and SR Plots for a spark for $\omega = 10$ and 30 with other parameters from Case A of Table 2.5.1.

$\omega = 50$



$\omega = 100$

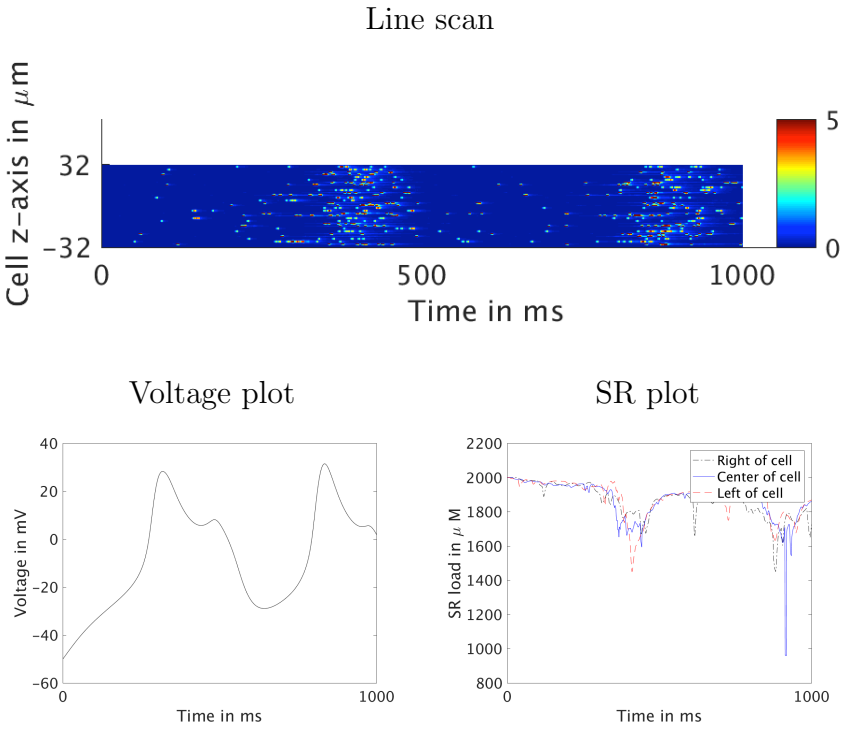


Figure 2.5.10 Line Scans, Voltage Plots, and SR Plots for a spark for $\omega = 50$ and 100 with other parameters from Case A of Table 2.5.1.

CRU plots

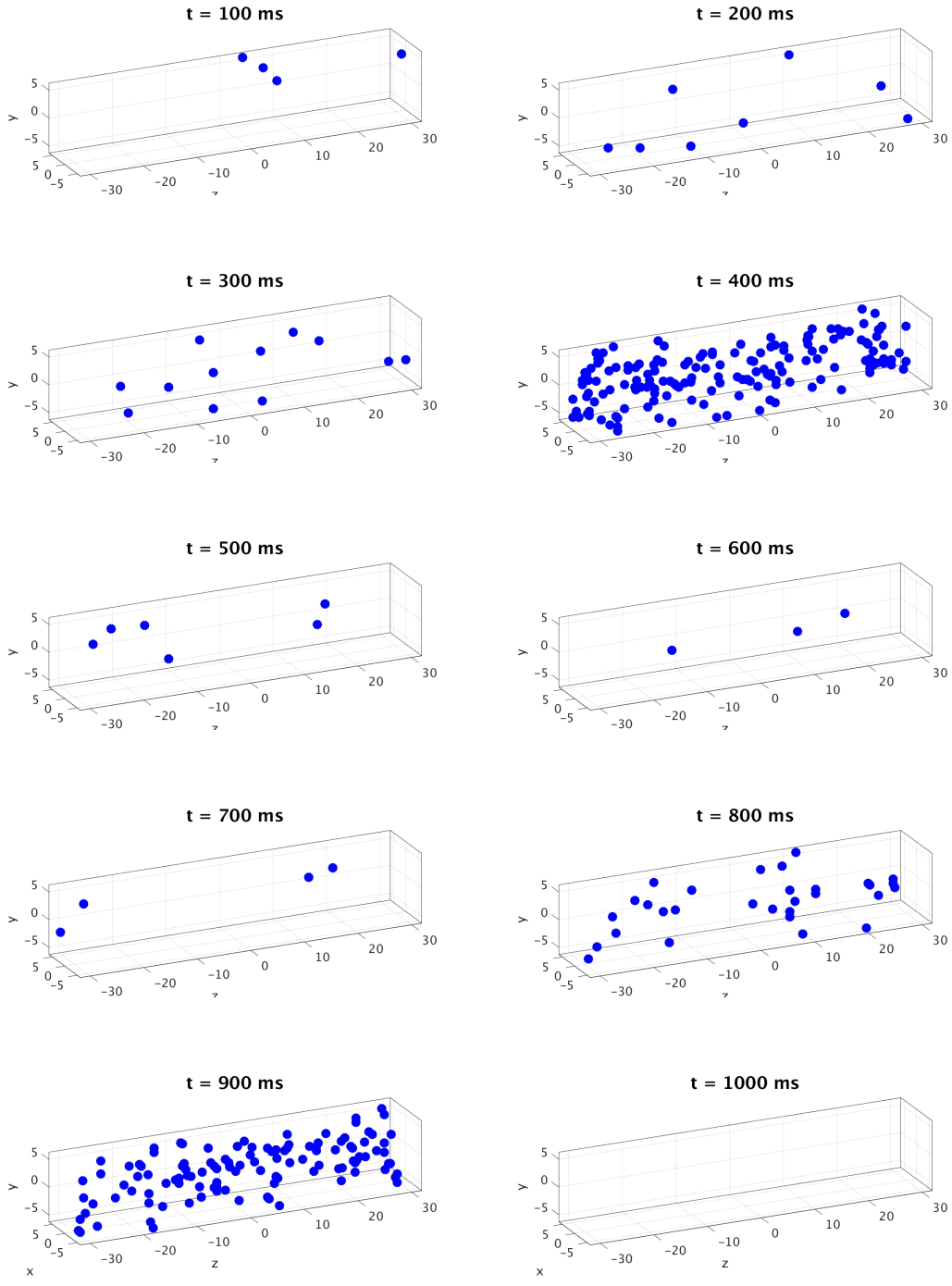


Figure 2.5.11 CRU plots for sparking case with $\omega = 30$ with other parameters from Case A of Table 2.5.1.

Isosurface plots

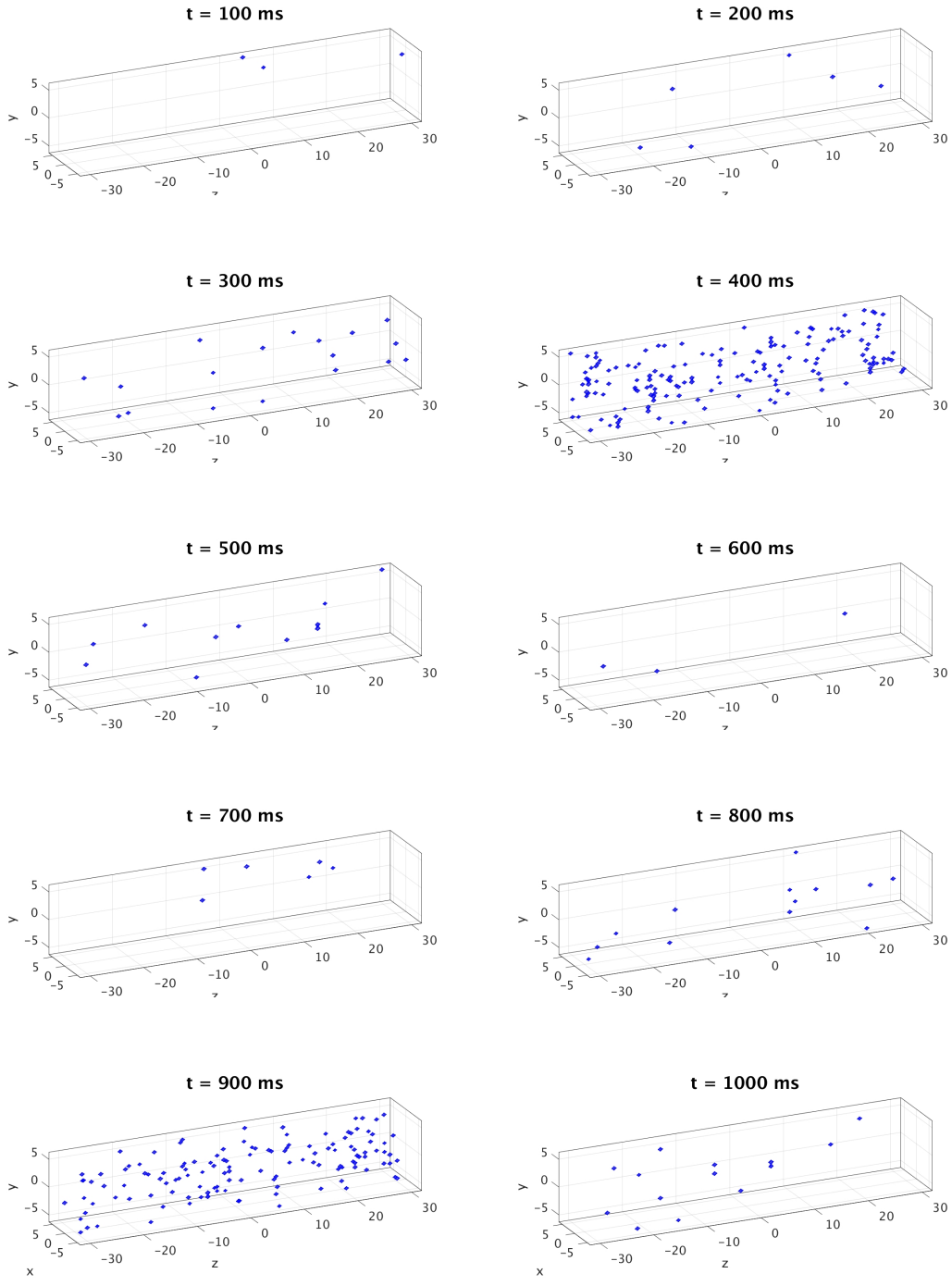


Figure 2.5.12 Isosurface plots for sparking case with $\omega = 30$ with other parameters from Case A of Table 2.5.1.

ω has a clear impact on the linescans present in Figure 2.5.13, and Figure 2.5.14 particularly when comparing extreme values like $\omega = 10$ and $\omega = 100$. This factor ten increase in ω causes heavy sparking to occur throughout the entire linescan of $\omega = 100$ whereas the linescan for $\omega = 10$ remains more similar to the wave case where $\omega = 0$. For $\omega = 10$, the concentration, though less frequent, is much greater when it occurs when compared to linescan of $\omega = 100$. The linescans in between demonstrate this metamorphosis in spark activity, gradually becoming more frequent but less concentrated.

Case C: Blowup Case with $\omega > 0$

Under the same parameters as the blowup case in Section 2.5.1, we present simulations with $\omega = 10, 30$ in Figure 2.5.15 and $\omega = 50, 100$ in Figure 2.5.16. While line scans appear similar, there is a clear distinction. As in Section 2.5.1, we experience a repeated blow up. As we increase ω, the repeated blowups continue for a longer amount of time. If we examine the accompanying voltage plots, we observe that as ω increases, so does the voltage entering the system. Beginning at $\omega = 50$, we observe a behavior known as early after-depolarization or EAD. EAD is a single depolarization that occurs after a previous depolarization, but earlier that expected in the regular frequency. An amalgamation of these behaviors in multiple cardiomyocytes could result in cardiac arrhythmia. SR plots show the amount of Ca^{2+} in the SR decreases significantly over the course of the simulation. We observe that as ω increases, the amount of Ca^{2+} that is being removed (or pumped out) increases.

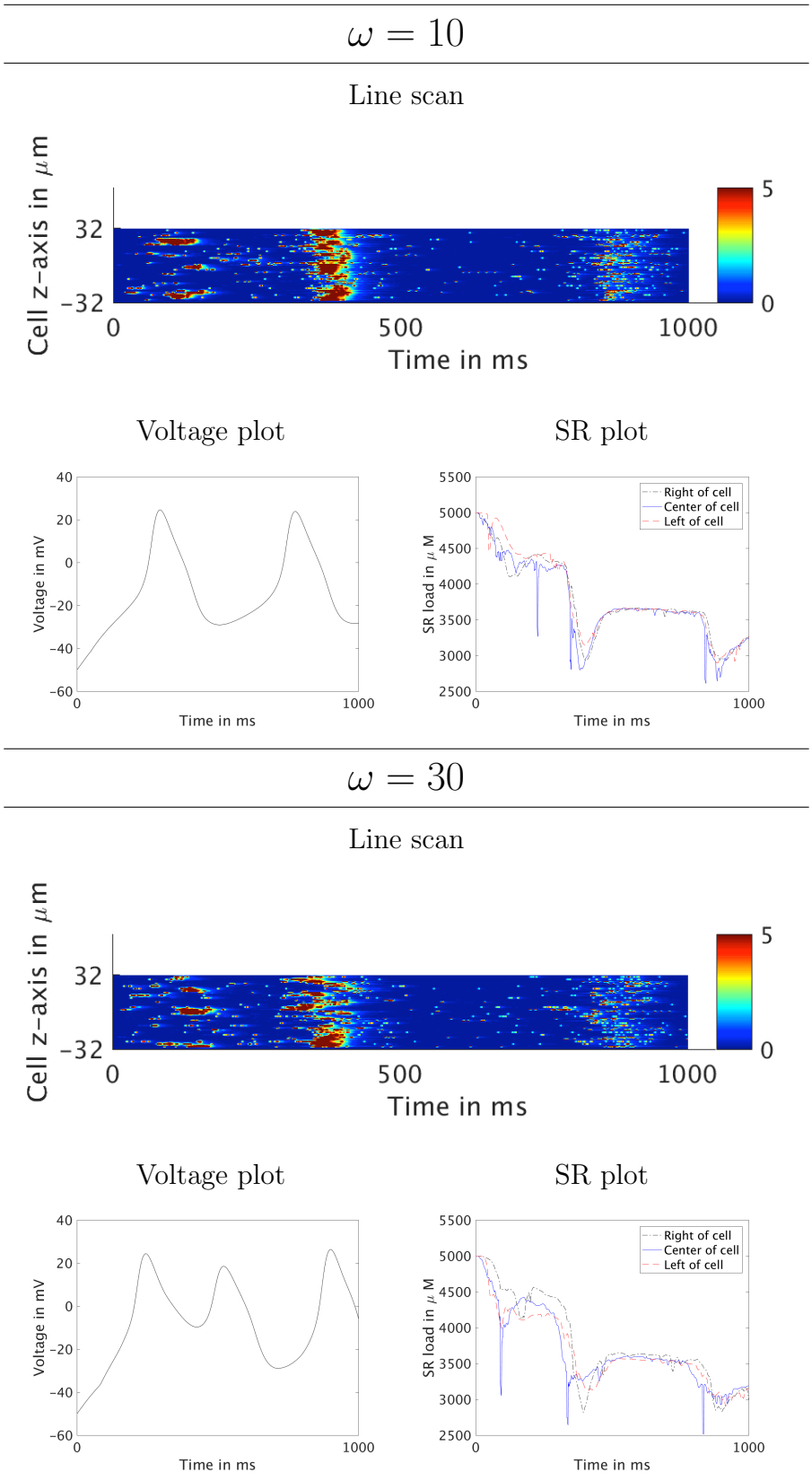


Figure 2.5.13 Line Scans, Voltage Plots, and SR Plots for a wave for $\omega = 10$ and 30 with other parameters from Case B of Table 2.5.1.

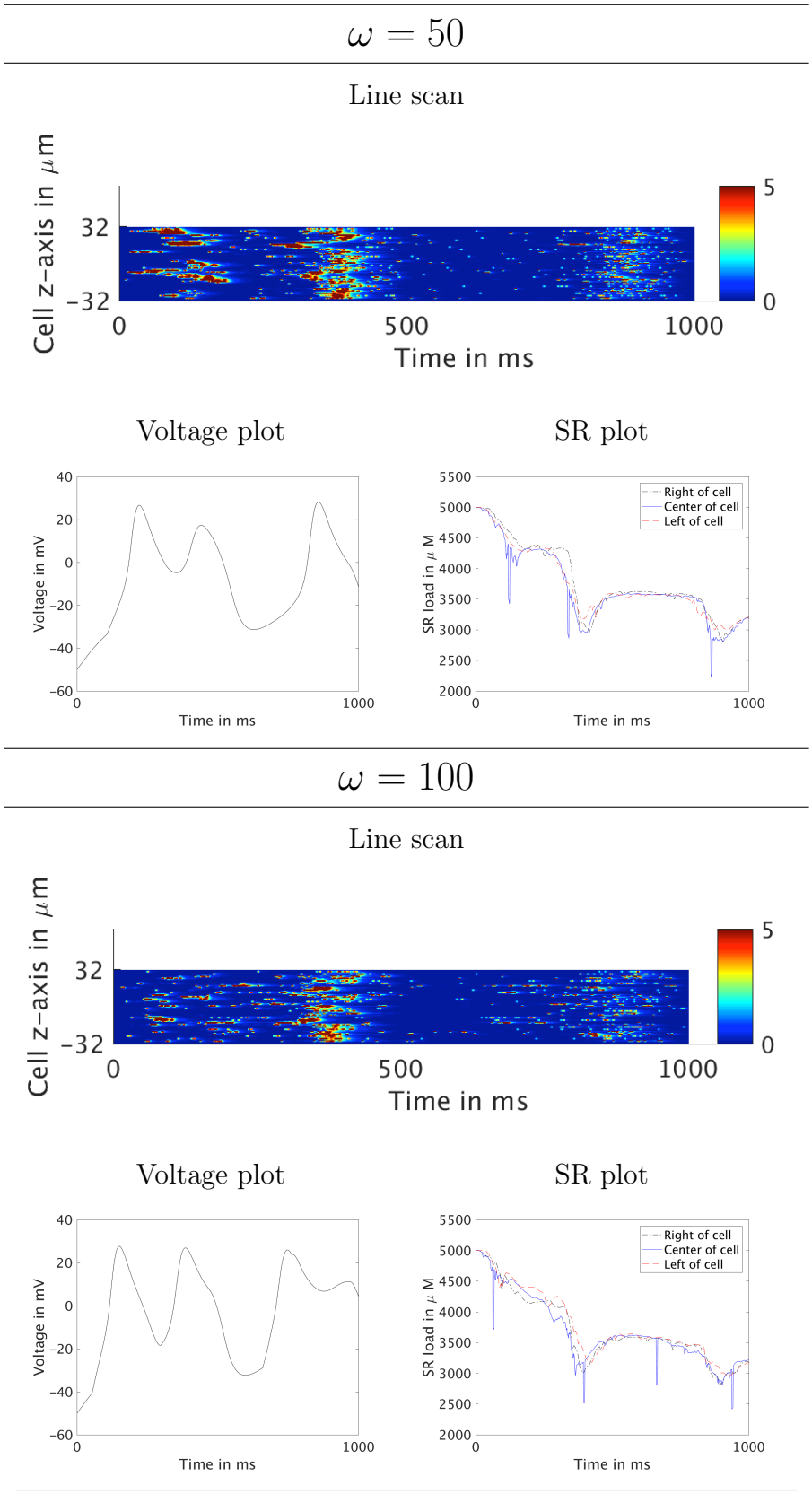


Figure 2.5.14 Line Scans, Voltage Plots, and SR Plots for a wave for $\omega = 50$ and 100 with other parameters from Case B of Table 2.5.1.

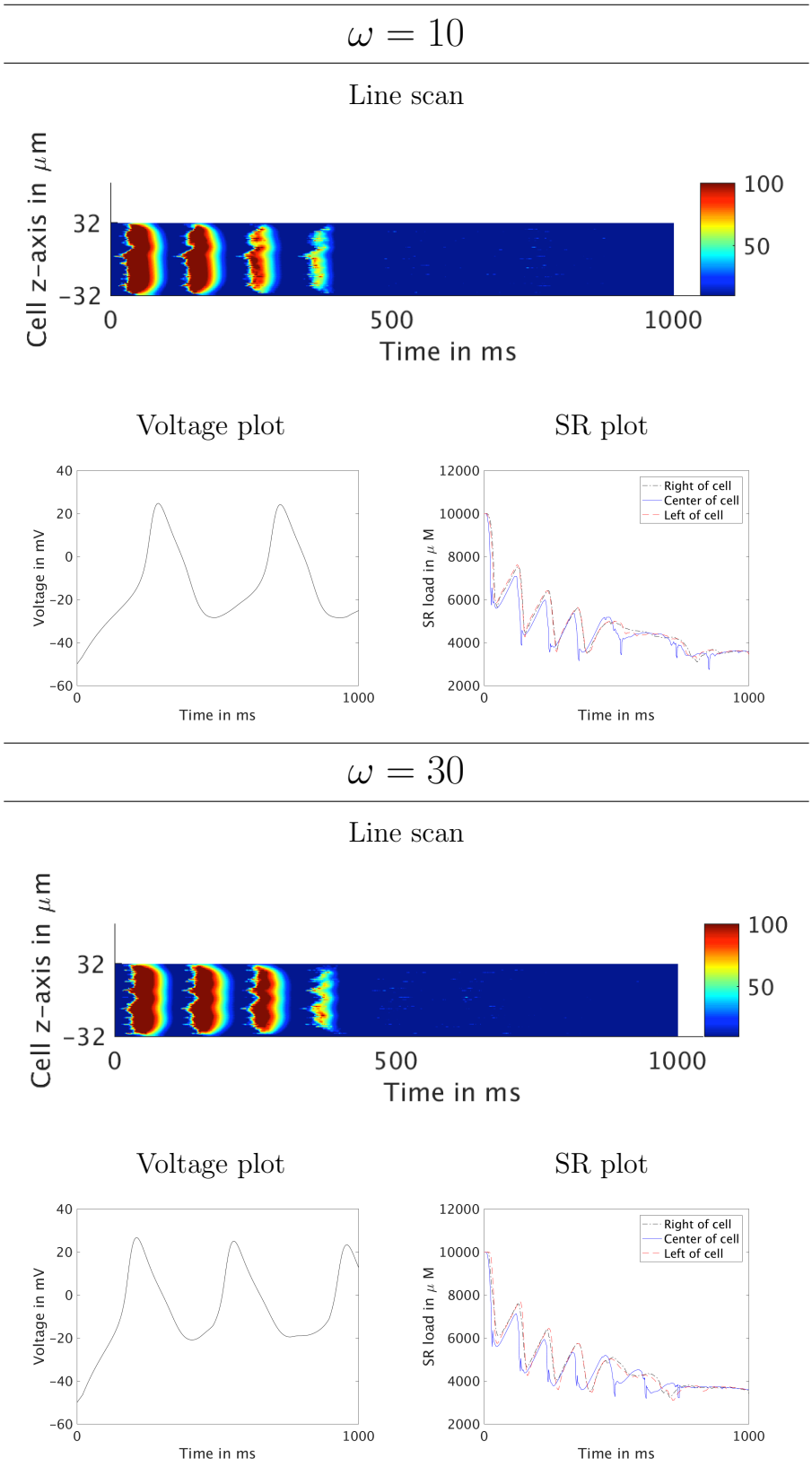


Figure 2.5.15 Line Scans, Voltage Plots, and SR Plots for blowup for $\omega = 10$ and 30 with other parameters from Case C of Table 2.5.1.

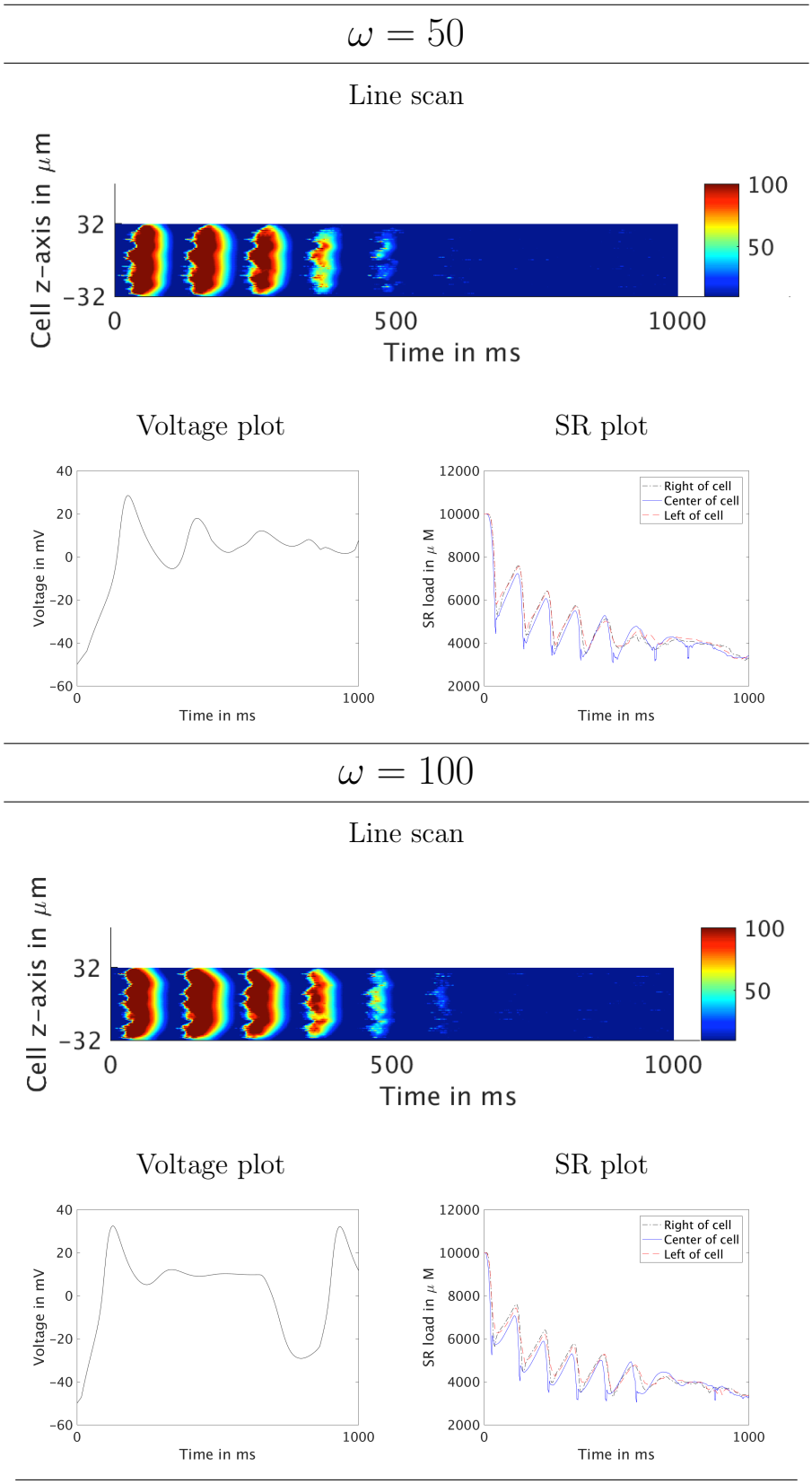


Figure 2.5.16 Line Scans, Voltage Plots, and SR Plots for blowup for $\omega = 50$ and 100 with other parameters from Case C of Table 2.5.1.

2.6 Conclusions

We formulated a complete model of the electrical excitation, calcium signaling, and mechanical contraction systems. The calcium and mechanical systems are linked via feedback and feedforward terms for the contractile dynamics model, using the proportion of actin-myosin cross-bridges which are actively linked and therefore generating contractile force at any given time. We introduced a new cytosol species to describe these actin-myosin cross-bridges and introduced a corresponding third cytosol reaction term. We also modified the reaction equation for troponin to more accurately describe the decreased calcium-troponin disassociation rate resulting from the protein's change in shape.

We introduced a calcium dependency into the voltage PDE, controlled by a scaling factor of feedback strength ω. The calcium efflux term $(J_{m_{pump}} - J_{m_{leak}})$ represents the only coupling between cytosol calcium levels and voltage in the full set of PDEs, which can be switched on or off by making the feedback strength parameter ω a positive value or zeroing it out to sever the connection from the calcium dynamic to the electrical dynamic. Our parameter study indicated that as the coupling strength between the calcium system and the electrical system increases, the duration of the action potential lengthens. The simulated action potential is qualitatively similar to an experimental range and duration. The range of action potential stays between -50 mV and $+20$ mV, while the duration of the action potential varies. Values for ω below 10 are too low to produce notably altered results. However, as we continued to scale ω up, we began to see a stronger and stronger influence of the current generated by calcium efflux through the membrane upon the electrical potential of the membrane itself. The formerly steady and unaffected periodicity of the voltage over time first sped up, then deteriorated altogether, as we continued to increase the value. Over strengthening the feedback connection from the current generated by calcium efflux

results in physiologically unrealistic behavior.

Future work should also consider continued development and refinement of the model. This could include investigating other options for the implementation of the feed-forward calcium to electrical system link. In particular, the addition of the NCX itself, or modifying J_{LCC} inactivation to be dependent on the calcium in the cell. Additionally, considering the coupling of the mechanical and electrical systems. One option would be to pursue the stretch activated channels that would couple contraction and voltage.

The formulation of the current model also opens many opportunities for further study. Simulations with 7 species that add the actin-myosin cross-bridges as a third cytosol buffer species thus enabling the feedback and feedforward links from calcium signaling to mechanical contraction. The addition of calsequestrin as a buffer species in the SR is also interesting. Since calsequestrin binds to calcium ions in the SR, it would have impacts across all three systems as less calcium ions would leave the SR. Examining the impact of calsequestrin on all three systems would require the use of 8 species. Less species could be used to examine calsequestrin as it impacts the electrical excitation effects without mechanical effects using 7 species or the calsequestrin impact with mechanical without electrical using only 6 species. We also suggest further study of the membrane pump term is important so that the addition of this term may not have such a significant impact on behaviors. Our introduction of ω has enabled a more direct representation of the relationship between cytosol calcium concentration and membrane potential. Thus we are also now able to study the interplay between the strengths of the feedback and feedforward links of electrical excitation and calcium signaling with κ and ω.

CHAPTER 3

USAGE STRATEGIES FOR THE INTEL XEON PHI

This chapter studies the performance of the KNL in different configuration modes on a classical elliptic test problem. The content of this chapter is contained in [15,35].

3.1 Introduction

The Intel Xeon Phi Knights Landing (KNL) is a recently released second-generation many-integrated-core (MIC) Xeon Phi processor with a theoretical peak performance of over 3 TFLOP/s of double precision floating-point performance [49]. The KNL was announced in June 2014 [24] and began shipping in July 2016. The Xeon Phi family is Intel's contributions to the use of many-core processors in parallel computing. The KNL itself is like a 'massively parallel' supercomputer from the early 2000s with dozens of nodes connected by a Cartesian network, all in a single chip now! Already the first-generation Phi Knights Corner (KNC) had an impact since its appearance in 2012, as exhibited by many of the highest-ranked clusters on the Top 500 list (`www.top500.org`) since then that use the Phi. Two clusters using pre-production or early-production KNL chips achieved ranks #5 and #6 on the November 2016 list. Entry #5 is the Cori cluster at NERSC (`www.nersc.gov`) in the USA with Cray XC40, Intel Xeon Phi 7250 68C 1.4GHz, and Aries interconnect. Entry #6 is the Oakforest-PACS cluster at the Joint Center for Advanced High Performance Computing in Japan with PRIMERGY CX1640 M1, Intel Xeon Phi 7250 68C 1.4GHz, and Intel Omni-Path network. The same KNL model as in these machines is installed on Stampede which is part of the Texas Advanced Computing Center (TACC) at the University of Texas at Austin (`www.tacc.utexas.edu`) [51]. The Stampede-KNL cluster at TACC at the time of this writing has 504 available KNL nodes, but announced on

December 13, 2016 from XSEDE User News that the upcoming Stampede 2 will reach a total size of 5,940 nodes after significant development planned for summer and fall of 2017. We concretely refer to Stampede in this work, since many researchers, e.g., U.S. based faculty, can apply for allocations through XSEDE (`www.xsede.org`).

The paper [49] by the chief designers of the KNL introduces the key features of the KNL. The most fundamental change with the second-generation Phi is its ability to serve as stand-alone processor, i.e., as a CPU. Since source code can in a first pass be ported to the Phi without change, but by simply compiling with an additional compile flag, there is potential for immediate impact on research as well as production codes. The crucial improvements for performance of the second-generation KNL Phi are the 2D mesh interconnect that provides high-bandwidth connections on the chip and the high-performance 16 GB MCDRAM memory on board the chip. The Phi can also access the DDR4 memory of the node, but MCDRAM is directly in the chip and is nominally 5x faster than DDR4 [49].

The KNL can be configured in one of three **memory modes** and one of three **cluster modes**. The three possible **memory modes** are Cache, Flat, and Hybrid mode. Each mode sets up the access of the MCDRAM and DDR4 differently, which can have significant impact on performance, especially for certain problem sizes. The user must make an appropriate choice for their simulation in order to achieve optimal performance. The three possible **cluster modes** are All-to-All, Quadrant, and Sub-NUMA 4 (SNC-4) mode. Each cluster mode handles cache level memory access differently. We also compare the distribution of MPI processes versus OpenMP threads and different thread to core control using different `KMP_AFFINITY` settings.

The Intel Developer Zone includes a tutorial on the High Bandwidth Memory on the KNL [28] and notes explicitly that "with the different memory modes by which the system can be booted, it becomes very challenging from a software perspective

to understand the best mode suitable for an application." We focus our attention on the different KNL configurations and show performance results. TACC provides documentation in the form of a user guide on their webpage [37] to instruct users on how to interact with the system. This documentation includes descriptions of the KNL memory modes, cluster modes, and the queues for each available KNL configuration. The Stampede technical report [43] provides specific recommendations for running on the Stampede KNL cluster. We believe, it is helpful to users to see the recommendations demonstrated with performance numbers explicitly. This work tests these recommendations and demonstrates performance results. We hope that this helps researchers choose the most suitable configuration for their needs with clear compile and run instructions for the Stampede KNL cluster.

We analyze the performance using a classical test problem, the Poisson equation with homogeneous Dirichlet boundary conditions

$$\begin{aligned} -\Delta u &= f \quad \text{in } \Omega, \\ u &= 0 \quad \text{on } \partial\Omega, \end{aligned} \tag{3.1.1}$$

on the domain $\Omega = (0,1) \times (0,1) \subset \mathbb{R}^2$. The equation is discretized by the finite difference method on a $N \times N$ mesh and the resulting system of N^2 linear equations solved by the conjugate gradient method. The numerical method is parallelized in C using hybrid MPI+OpenMP code. Section 3.2 provides more implementation details. We use a short self-written special-purpose code as test code, but since (3.1.1) is a very well-known standard example (e.g., [9, Section 6.3], [17, Subsection 9.1.1], [31, Chapter 12], and [53, Section 8.1]), researchers should be easily able to recreate the results.

The KNL has only recently begun to be available to the general public, and only few performance comparisons are available. One paper is [44] that approaches the question by running established benchmark codes (Mantevo suite [19], NAS Paral-

lel Benchmarks [3, 39, 55]) and sophisticated established research codes (WRF [48], LBS3D [42]) as test cases on the KNL. Another performance result of a formal benchmark is contained in the brief announcement [30] by Intel of results of the HPCG Benchmark that characterizes speedup of KNL over two 18-core CPUs (Intel E5-2697v4) as approximately 2x. The HPCG Benchmark [20] is essentially a more sophisticated implementation of a discretization of the 3-D version of (3.1.1) and designed to provide a modern reference for performance [10].

The remainder of the chapter is organized as follows. Section 3.2 describes the test problem and hybrid MPI+OpenMP code implementation. Section 3.3 describes the hardware used in our studies and details the differences in the KNL configuration options. Section 3.4 presents performance results for the different KNL configuration modes and includes baseline results for comparison to First-Generation Phi and CPU results. Finally, Section 3.6 summarizes our conclusions and suggests opportunities for future studies that would further explore features of the KNL.

3.2 Test Problem

We consider the classical elliptic test problem of the Poisson equation with homogeneous Dirichlet boundary conditions in (3.1.1), as used in many Numerical Linear Algebra textbooks as standard example, e.g., [9, Section 6.3], [17, Subsection 9.1.1], [31, Chapter 12], and [53, Section 8.1], and researchers should be easily able to recreate the results. Using $N + 2$ mesh points in each dimension, we construct a mesh with uniform mesh spacing $h = 1/(N + 1)$. Then approximate the second-order derivatives in the Laplace operator at the N^2 interior mesh points by centered difference approximations. Using these approximations together with the homogeneous boundary conditions (3.1.1) as determining conditions for the approximations u_{k_1,k_2} gives a system of N^2 linear equations for the finite difference approximations u_{k_1,k_2} at the

N^2 interior mesh points $k_i = 1, \ldots, N$, $i = 1, 2$.

For simplicity of coding, for reproducibility by others, and to provide a challenging benchmark for the hardware to be tested, we collect the N^2 unknown approximations u_{k_1,k_2} in a vector $u \in \mathbb{R}^{N^2}$ using the natural ordering of the mesh points with $k = k_1 + N(k_2 - 1)$ for $k_i = 1, \ldots, N$, $i = 1, 2$. We can state the problem as a system of linear equations in standard form $A\,u = b$ with a system matrix $A \in \mathbb{R}^{N^2 \times N^2}$ and a right-hand side vector $b \in \mathbb{R}^{N^2}$. The components of the right-hand side vector b are given by the product of h^2 multiplied by right-hand side function evaluations $f(x_{k_1}, x_{k_2})$ at the interior mesh points using the same ordering as the one used for u_{k_1,k_2}. The system matrix $A \in \mathbb{R}^{N^2 \times N^2}$ can be defined recursively as block tri-diagonal matrix with $N \times N$ blocks of size $N \times N$ each. Concretely, we have $A = \text{block-tridiag}(T, S, T) \in \mathbb{R}^{N^2 \times N^2}$ with the tri-diagonal matrix $S = \text{tridiag}(-1, 4, -1) \in \mathbb{R}^{N \times N}$ for the diagonal blocks of A and with $T = -I \in \mathbb{R}^{N \times N}$ denoting a negative identity matrix for the off-diagonal blocks of A; see, for instance, [36] for complete detail.

For fine meshes with large N, iterative methods such as the conjugate gradient method are appropriate for solving this linear system and guaranteed to converge, since the system matrix A is known to be symmetric positive definite [17]. We use a tolerance of 10^{-6} on the relative residual and run all reported cases to convergence. In a careful implementation, the conjugate gradient method requires in each iteration exactly two inner products between vectors, three vector updates, and one matrix-vector product involving the system matrix A. In fact, this matrix-vector product is the only way, in which A enters into the algorithm. Therefore, a so-called matrix-free implementation of the conjugate gradient method is possible that avoids setting up any matrix, if one provides a function that computes as its output the product vector $v = A\,u$ component-wise directly from the components of the input vector u by using the explicit knowledge of the values and positions of the non-zero components of A,

but without assembling A as a matrix.

Thus, without storing A, a careful, efficient, matrix-free implementation of the (unpreconditioned) conjugate gradient method only requires the storage of four vectors (commonly denoted as the solution vector x, the residual r, the search direction p, and an auxiliary vector q). In a parallel implementation of the conjugate gradient method, we split the domain in the last dimension for each parallel process. Each vector is split into as many blocks as parallel processes are available and one block is distributed to each process. each vector is split into as many blocks as parallel processes are available and one block distributed to each process. That is, each parallel process possesses its own block of each vector, and no vector is ever assembled in full on any process. By contrast, the vector updates in each iteration can be executed simultaneously on all processes on their local blocks, because they do not require any parallel communications. The MPI function `MPI_Allreduce` is necessary to compute the inner product and vector norm required in the algorithm. The matrix-vector product $v = A\,u$ also computes only the block of the vector v that is local to each process. But since the matrix A has non-zero off-diagonal elements, each local block needs values of u that are local to the two processes that hold the neighboring blocks of u. The commands `MPI_Isend` and `MPI_Irecv` are used as non-blocking communication commands between neighboring processes and to interleave calculations and communications in the matrix-vector product. In the hybrid MPI+OpenMP implementation, all expensive `for` loops with large trip counts are parallelized with OpenMP. This includes the loops in the computation of the matrix vector product, axpby operations, and computation of dot products. These `for` loops are parallelized with the OpenMP pragma `#pragma omp parallel for`, instructing each MPI process to divide the iterations of the loop between available threads and execute in parallel.

Table 3.2.1 presents a convergence study for the test problem (3.1.1). We use right-hand side function

$$f(x_1, x_2) = (-2\pi^2)\Big(\cos(2\pi x_1)\,\sin^2(\pi x_2) + \sin^2(\pi x_1)\,\cos(2\pi x_2)\Big), \tag{3.2.1}$$

for which the true analytical solution $u(x_1, x_2) = \sin^2(\pi x_1)\,\sin^2(\pi x_2)$ is known. Table 3.2.1 lists the mesh resolution $N \times N$, the number of degrees of freedom N^2 (DOF; i.e., the dimension of the linear system), the norm of the finite difference error $\|u - u_h\| \equiv \|u - u_h\|_{L^\infty(\Omega)}$, the ratio of consecutive errors $\|u - u_{2h}\|/\|u - u_h\|$, the number of conjugate gradient iterations `#iter`, and the predicted memory usage in GB. The norm of the finite difference error is calculated against the known true solution. Each of the progressively finer meshes is the result of doubling the mesh resolution N.

We observe that the norms of the errors in Table 3.2.1 decrease by a factor of approximately 4 as the mesh is refined by a factor 2. This confirms the second-order convergence of the finite difference method used, e.g., [5,31]. The optimal convergence order confirms that the linear solver tolerance is tight enough for a sufficiently accurate solution of the linear system. The number of iterations gives an estimate on the computational demand of the method. The memory usage predicted for the code with 4 vectors shows that cases up to 16,384 × 16,384 can fit in the memory of a KNL (but not in that of a KNC). We also note that the 65,536 × 65,536 problem size is estimated at 128 GB, larger than all available memory on the KNL node so it is omitted from this work.

3.3 Hardware

The second generation Intel Xeon Phi (x200 series) is codenamed Knights Landing (KNL). It was announced in June 2014 and began delivery in mid 2016. The most fundamental change with the second-generation Phi is its ability to serve as stand-

Table 3.2.1 Convergence study for the test problem (3.1.1) with iteration count and memory prediction.

$N \times N$	DOF	$\|u - u_h\|$	Ratio	`#iter`	predicted memory (GB)
1,024 × 1,024	1,048,576	3.1266e–06	—	1,581	< 0.100
2,048 × 2,048	4,194,304	7.8019e–07	4.01	3,192	0.125
4,096 × 4,096	16,777,216	1.9366e–07	4.03	6,452	0.500
8,192 × 8,192	67,108,864	4.7401e–08	4.09	13,033	2.000
16,384 × 16,384	268,435,456	1.1701e–08	4.05	26,316	8.000
32,768 × 32,768	1,073,741,824	3.0867e–09	3.79	53,141	32.000

alone processor, i.e., without a CPU as host in a node. The KNL also runs a full Linux-based OS. While the KNL can also function as a co-processor, similar to a KNC or GPU, we restrict our attention to the set up as a stand-alone processor. Also, for Stampede, the KNL as co-processor is not available at present.

The second-generation KNL Phi introduces a 2D mesh interconnect that provides high-bandwidth connections between tiles and controllers on the chip. Figure 1.2.4 shows a schematic of a KNL. The 16 GB MCDRAM memory on board the chip (Multi-Channel DRAM) is a new form of HMC (Hybrid Memory Cube) or stacked memory. The Phi can also access the DDR4 memory of the node, but MCDRAM is directly in the chip and is nominally 5x faster than DDR4 [49]. The MCDRAM is also nearly 50% faster than GDDR5 memory, which is the memory on board the first-generation Phi (KNC). The KNL node DDR4 memory is connected though 6 channels and has a total of 96 GB, with larger capacity also possible. The KNL on-chip memory is connected by a 2D mesh structure that allows for significantly more bandwidth than the first-generation Phi bi-directional ring bus. The KNL model we focus on has 68 cores, but up to 72 cores are possible.

It is also important to note that the KNL has double the number of Vector Processing Units (VPUs) from the first-generation model. The KNC had one VPU per

core while the KNL has two VPUs per core. Since each VPU is 512 bits wide, 8 double precision operations per cycle can be executed. Thus, on each KNL core 16 double precision operations can occur at the same time [50]. Section 3.3.1 discusses the KNL model used in more detail.

As baseline reference, we tested KNL performance against current Stampede cluster hardware. A majority of the compute nodes in the Stampede cluster are hybrid CPU/Phi nodes with one Phi KNC and two 8-core CPUs. That is, a first-generation Phi (KNC) as a co-processor in a compute node with two 8-core CPUs. Most of the compute nodes in that cluster have one Phi KNC, but some are configured with two Phi KNCs, alongside the two 8-core CPUs. More detail on this baseline hardware is given in Section 3.3.2.

3.3.1 Intel Xeon Phi Knights Landing (KNL)

We focus on the Intel Xeon Phi 7250 KNL compute nodes in the 508 node Stampede-KNL cluster. Each KNL runs CentOS 7 as a self-hosted node. This model has 68 cores with 4 hardware threads per core across 34 tiles with two cores each and L2 cache is shared by the two cores on each tile [27]. Each core has a clock rate of 1.4 GHz. Each of the KNL nodes includes 16 GB of MCDRAM and 96 GB of DDR4 for a total of 112 GB of memory. For a schematic diagram and a die photo of the KNL respectively see [49, Figure 1 (a) and Figure 2]. Memory is controlled by 2 DDR controllers on opposite sides of the chip, and 8 controllers for MCDRAM with two in each quadrant. The 2D mesh structure connects the tiles, and the controllers on the chip.

The KNL provides flexibility to the user in how to set up the different types of memory and localization of low level cache actions. This flexibility presents users with choices of what configuration of the KNL to use. The configuration of the KNL

is set at boot time so proper choice based on the application in advance of running tests is important. Stampede uses different queues for access to differently configured KNLs. The configuration of the KNL depends on two mode choices, the memory mode and the cluster mode. The memory mode determines how the MCDRAM and DDR4 RAM are set up. The cluster mode determines how the core level L1 cache and tile level L2 cache data transfers are localized on the chip.

KNL Memory Modes

The KNL has three possible **memory modes**: Cache, Flat, and Hybrid. Information on Stampede treatment of these modes comes from [37].

Cache memory mode: The 16 GB of high-performance MCDRAM is configured as a L3 cache and the 96 GB of DDR4 is set as RAM. The normal and development queues on the Stampede-KNL cluster are set up in Cache mode.

Flat memory mode: Separates the MCDRAM and DDR4 into distinct NUMA nodes. That is, the full 112 GB of RAM are available, but split into two parts. The default behavior is for memory allocations to use the DDR4, but the user may explicitly control the type of memory used. The Flat-Quadrant and Flat-All2All queues on the Stampede-KNL cluster are set up in Flat mode.

Hybrid memory mode: A mix of Cache and Flat mode in which either half or a quarter of the MCDRAM is set as a L3 cache as in Cache mode while the rest of the MCDRAM is set as a NUMA node alongside the DDR4 NUMA node as in flat mode. The cached MCDRAM in this mode serves all of the DDR4. Hybrid mode is not currently supported on any Stampede-KNL queue.

In Cache memory mode no software modifications are required, but there is higher latency for DDR access and L3 cache misses are limited by the DDR bandwidth. All

memory must go through the hierarchy, it is first transferred as DDR, then MCDRAM, then L2 cache, then to the KNL cores. There is also less total addressable memory in this mode, limited by the 96 GB size of the DDR4.

In Flat memory mode, the MCDRAM and DDR4 are separate addressable memory spaces, so the entire 112 GB is available. Using only the MCDRAM capitalized on the maximum bandwidth with lower latency than in Cache mode since the memory hierarchy does not include a L3 cache, but is limited to 16 GB. Using only the DDR4 fails to capitalize on the higher bandwidth of the MCDRAM and is limited to the 96 GB size of the DDR4. The `numactl` utility can be used to easily run Flat memory mode using either the DDR4 or the MCDRAM with run time flags that specify the type of memory to use. It is also possible using the `numactl` utility to specify the preferred option, indicating to use the MCDRAM if available, then the DDR4. This preferred option gives the user access to the full 112 GB and attempts to make use of both memory types appropriately. If the user wants to see maximum benefit in Flat memory mode software modifications are necessary and require decisions on what data should go where. In particular, careful management of the MCDRAM and tracking of its usage is important and can add significant complexity to the code. With this explicit control in Flat memory mode, the user can access all of the 112 GB of memory, exactly in the way they desire, within the size constraints. The `memkind` library (`memkind.github.io/memkind/`) is one option for users to manage memory in the code and control explicitly which parts of the code use which type of memory. The paper [49] includes discussion on their Flat MCDRAM software architecture including their `HBW_malloc` library in `memkind` to allocate critical memory in MCDRAM. For the reason that implementing the `memkind` library would require the user to modify the code, we focus on control using the `numactl` utility.

For a user very aware of the memory demands of their code a customized Hybrid mode setup, made from carefully choosing the amount of MCDRAM to use a L3 cache and managing access to the DDR4 and MCDRAM NUMA nodes, could be complicated, but beneficial. Since Hybrid mode is a combination of Cache and Flat memory modes and is not currently supported on Stampede, we omit further consideration at this time. We expect that the full treatment of Cache and Flat memory modes may help guide the sophisticated user in their Hybrid mode configuration choices.

KNL Cluster Modes

The KNL has three possible **cluster modes**: Quadrant, All-to-All, and SNC-4. The idea of the cluster mode is to lower latency and increase bandwidth by shortening the distance of protocol flows at the low level cache [49]. Information on Stampede treatment of these modes comes from [37].

All-to-All cluster mode: This mode distributes memory addresses uniformly. The Flat-All2All queue on the Stampede-KNL cluster uses All-to-All cluster mode.

Quadrant cluster mode: Tiles are grouped into four virtual quadrants and each tile manages MCDRAM addresses only in its quadrant. Thus communications are somewhat localized without explicit control by the programmer. This is Intel's recommended default [37]. The normal and development queues on the Stampede-KNL cluster use Quadrant cluster mode.

Sub-NUMA 4 (SNC-4) cluster mode: Splits the chip into four NUMA domain that function like a four socket processor. The communication is then restricted to a single NUMA domain, when possible. This requires explicit manual management by the programmer to get better performance. The Flat-SNC-4 queue on the Stampede-KNL cluster uses SNC-4 cluster mode.

Quadrant mode has a hemisphere variant in which the tiles are grouped into 2 virtual hemispheres rather than the four quadrants. Similarly, SNC-4 has a SNC-2 variant in which the chip is divided into 2 NUMA nodes rather than 4.

For a simple understanding of the differences, consider that All-to-All mode has no communication localized, Quadrant mode has some communication localized, and SNC-4 has all communication localized. To be precise, each cluster mode differs in the localization relationship between the tile, the distributed tag directory, and the memory [49]. All-to-All cluster mode has no localized relationship between the tile, the directory, and the memory. In Quadrant cluster mode, the four virtual quadrants provide localization between directory and memory. That is, a directory will only access memory in its own quadrant but a request from any tile can land on any directory. Sub-NUMA cluster mode further localizes the tile with directory and the memory. In SNC cluster mode, a request from a tile will access directory in its local cluster and the directory will access memory controllers also in that cluster [49]. As a result, All-to-All mode cache actions can have a higher latency than other mode. Quadrant mode can have better latency than All-to-All, but SNC cluster mode should have the lowest latency among the three cluster modes.

3.3.2 Baseline Stampede Hardware

One hybrid compute node contains, two 8-core Xeon E5 processors and 1 61-core Xeon Phi KNC (first-generation) co-processor. Each node has 32 GB per node of distributed memory and 8 GB memory on the KNC. Nodes are interconnected by InfiniBand with Mellanox network cards with FDR. Physically, the node is contained in a Dell C8220z double-wide sled in a 4 rack-unit chassis with 3 other sleds. Each node runs CentOS 6.3 with the `2.6.32 x86_64` Linux kernel. The section Intel Xeon Phi Knights Corner (KNC) provides more detail on the KNC and section CPU Hardware

briefly describes the CPU specifications.

Intel Xeon Phi Knights Corner (KNC)

The first generation Intel Xeon Phi (x100 series) is codenamed Knights Corner (KNC). It was announced in June 2011 and began delivery in the fourth quarter of 2012. While the second generation Phi is a standalone processor, the KNC is not. The KNC only has a Linux micro-OS and can only be run as a co-processor, thus the need for a host CPU paired with a Phi in a hybrid node. The KNC can have up to 61 total cores [23] distributed in a bi-directional ring bus structure with 8 GB of GDDR5 memory on the chip. The PCIe bus connects the KNC to the DDR3 memory of the node. As in the second generation model, each core can support up to 4 threads for a possible total of 244 threads on the KNC. The KNC on Stampede is a special production model, Xeon Phi SE10P, that has 61 cores each with a 1.1 GHz clock speed.

As each KNC is paired with a host CPU there are different run modes to handle the interaction of the processors. Formally, the configuration of the co-processor is fixed, but three different run modes are possible: native mode, symmetric mode, and offload mode. In offload mode, portions of the computation may be offloaded from the host CPU to the Phi. That is, the host CPU controls all of the MPI ranks. In native mode, outside of setup, everything is run on the Phi. No computational use of the host CPU and all MPI ranks exist only on the Phi. In symmetric mode, both the host CPU and Phi do computations and MPI ranks exist on both the CPU and the Phi. For a detailed study of the KNC for the test problem used here see [36].

CPU Hardware

For CPU only baseline performance we use a compute node on Stampede that contains two 8-core 2.7 GHz Intel E5-2680 Sandy Bridge CPUs for a total of 16 computational

cores per node. Each node includes 32 GB of DDR3 memory with 4-channel integrated controllers and a 64 KB L1 Cache per core.

3.4 Results

This section reports performance studies for the solution of the test problem using hybrid CPU+OpenMP code on a single Intel Xeon Phi KNL and on one CPU/KNC node on Stampede (using both the KNC and the CPUs only) as a baseline comparison. For the KNL, the hybrid MPI+OpenMP code in C was compiled for the KNL on the login node using the Intel compiler version 17.0.0 with flag for the KNL `-xMIC-AVX512` and the other flags `-O3 -std=c99 -Wall -mkl -qopenmp` (with OpenMP version 4.5) and Intel MPI version 17.0.0. Compiling the executable is independent of the configuration mode in which the executable is run. In the available configuration modes that require no code modifications by the user we examine the choice of using all 272 available threads, or using only 256 threads, and the distribution of MPI process to OpenMP threads. We compare these to the strategy of using only 1, 2, or 3 threads per core of the KNL.

We also study different process and thread affinity types. Intel provides some context and description for this in relation to the Phi architecture in [29]. The more general description of the thread affinity interface can be found in [25]. The `KMP_AFFINITY` environment variable is an Intel OpenMP runtime extension that pins OpenMP threads to hardware threads ("thread pinning"). The `KMP_AFFINITY` can take a number of different types. For `compact`, threads are packed close to each other, for `scatter` a round-robin threads to cores distribution is used, for `explicit` the proclist modifier is used to pin threads, for `none`, threads are not pinned, and for `balanced`, as in `scatter` a round-robin threads to cores distribution is used but OpenMP thread ids are kept consecutive. We restrict our attention in this work

to the common choices of `compact` and `scatter` for the KNL. To be specific, for `KMP_AFFINITY=compact` the current OpenMP thread is placed as close as possible to where the previous thread was placed. In the case of `KMP_AFFINITY=scatter`, we have the opposite behavior, threads are distributed as evenly as possible across the entire system. Comparing these two opposite actions will provide sufficient insight into the impact of the `KMP_AFFINITY` choice on the KNL. It is also possible to specify explicitly which hardware threads using `KMP_AFFINITY=proclist=[<id_list>],explicit`.

For Intel OpenMP it is also possible to use the runtime extension `KMP_HW_SUBSETS` to control the allocation of resources [26]. For example, `KMP_HW_SUBSETS=68c,4t` specifies using 68 cores and 4 threads per core. For more explicit control the user may use both environment variables, `KMP_HW_SUBSETS=2t KMP_AFFINITY=scatter`. On Stampede, `KMP_HW_SUBSETS` is the recommended choice in place of the deprecated `KMP_PLACE_THREADS`.

We are confident in the differences in the behavior of `KMP_AFFINITY=compact` and `KMP_AFFINITY=scatter` in our runs based on logs for each OpenMP thread that record their cpu id, MPI process id, and the node number. The behaviors are independent of the KNL configuration used. By running the `lstopo` command on one of the KNL nodes we can identify that core 0 of the KNL contains hardware threads 000, 068, 136, and 204, and is on the same tile with core 1, containing threads 001, 069, 137, and 205. For a concrete example of the differences in behavior, consider the case with 17 MPI processes and 16 threads per process (272 threads total with 4 threads per core). In the `KMP_AFFINITY=compact` case, the first four threads are placed on cpu ids 000, 068, 136, 204. The next four threads are placed on cpu ids 001, 069, 137, 205, continuing with 002, 070, 138, 206, and with 003, 071, 139, 207. In the `KMP_AFFINITY=scatter` case, the first four threads are placed on consecutive cpu id numbers, 000, 001, 002, 003. The next four are placed on cpu ids 068, 069, 070,

071, continuing with 136–139, then 204–207. After the first 16 threads are placed, we see the pattern repeat for the threads of the second MPI process, 004–007, 072–075, 140–143, 208–211. This confirms that `KMP_AFFINITY=compact` packs the threads close together and that scatter is spreading out the threads in a round-robin fashion.

We use the same logs to assess the assignment of threads to cores in the case of 256 threads. In the case of 272 threads, each hardware thread on the KNL is used. In the 256 threads cases, we observe that the 16 threads not used are distributed across all cores of the KNL. The distribution of the unassigned threads is different for `KMP_AFFINITY=compact` versus `KMP_AFFINITY=scatter` and depends on the number of MPI processes used, but is essentially an even distribution across the hardware. Notably, in both cases, no single core is left unused and each of the unassigned threads is taken from a different hardware core and tile. Notice that this test does not directly assesses the recommendation to leave entire cores idle. We also test using only 1, 2, or 3 threads per core.

Section 3.4.1 shows results with the KNL configured in Cache memory mode with Quadrant cluster mode, which we refer to as the Cache Quadrant configuration. Section 3.4.2 shows results with the KNL configured in Flat memory mode with Quadrant cluster mode, which we refer to as Flat Quadrant configuration. Section 3.4.3 shows results with the KNL configured in Flat memory mode with All-to-All cluster mode, which we refer to as Flat All-to-All configuration. Finally, as baseline comparison, Section 3.4.4 compares KNL performance against the first-generation Phi (KNC) as well as CPU hardware that is currently available on Stampede; however, this equipment is in the phasing out stages of the Stampede cluster. We use one hybrid compute node with two 8-core CPUs and one Intel Xeon Phi KNC. Results using the KNC run in native and symmetric modes are in Section 3.4.4 and CPU only results are in Section 3.4.4.

3.4.1 Cache Quadrant Configuration

In this section, we present results in Cache memory mode using Quadrant cluster mode. This is the default on Stampede, used in both the develop and normal queues. In the develop queue there is a max of 4 total nodes in the queue and a one job maximum per user. The max runtime for a job in this queue is 2 hours. In the normal queue there is a maximum of 80 nodes per job allocated for this queue and each user is restricted to a maximum of a 10 jobs in the queue. The max runtime for a job in this queue is 48 hours. In addition to standard options in our slurm script, we specify the number of nodes (1) using `#SBATCH -N 1` and the number of MPI tasks using `#SBATCH -n <num_MPIproc>`. We use `export KMP_AFFINITY` and `export OMP_NUM_THREADS` and run simply with `ibrun <executable>`.

Table 3.4.1 shows the results in the Cache Quadrant configuration using all 272 available threads on the KNL. Table 3.4.2 shows the results in this mode using only 256 threads. Table 3.4.3 contains the study of using only 1, 2, or 3 threads per core of the KNL.

In Table 3.4.1 (a), `KMP_AFFINITY=compact` is used, while in Table 3.4.1 (b), `KMP_AFFINITY=scatter` is used. Analogously, in Table 3.4.2 (a) and Table 3.4.2 (b) we utilize `KMP_AFFINITY=compact` and `KMP_AFFINITY=scatter`, respectively. The choice of `KMP_AFFINITY=compact` or `KMP_AFFINITY=scatter` shows essentially no impact on the run times. In some cases in both Table 3.4.1 and Table 3.4.2 we see `scatter` performing slightly better than `compact`. But there are also cases where `compact` performs better than `scatter`.

Comparing across each row of Table 3.4.1 we observe no significant difference in run time for particular combinations of MPI process and OpenMP threads. In fact, for the 16,384 case in both Table 3.4.1 (a) and Table 3.4.1 (b) we observe the best run time using 272 MPI processes with only 1 OpenMP thread per process. In particular,

we do not observe any disadvantage in terms of run time to using as many as 272 MPI processes. The same behavior is exhibited in Table 3.4.2 using up to 256 MPI processes. In this case however, the 256 MPI process runs are not the best, but are still competitive times, below the average run time across the row in both Table 3.4.2 (a) and Table 3.4.2 (b). It is also interesting to note that the multi-threading only case with only 1 MPI process was not consistently performing worse than using more MPI processes. The 1 MPI process case performed better compared to other entries in the row in the case of 272 threads rather than 256 threads. In the 32,768 case using multi-threading only, the run times of 15:17:07 and 15:09:12 in Table 3.4.1 are better than the run times of 15:41:13 and 16:01:08. The difference is significant in that large mesh size case, but for the smaller mesh sizes no significant difference in run times is present.

Examining the case of using all 272 threads in Table 3.4.1 compared to using only 256 threads in Table 3.4.2 we do not observe a very significant advantage to using only 256 threads. However, upon close inspection there does appear to be a slight benefit to using only 256 threads for up to the 16,384 case. We see this in, for example, the 16,384 $\times$ 16,384 case. Comparing the average run time across each row for 272 threads using `compact` is 00:43:20 with best time 00:39:11, and 272 threads using `scatter` is 00:40:50 with best time 00:39:02. Comparing the average run time across each row for 256 threads using `compact` is 00:39:48 with best time 00:38:15, and 256 threads using `scatter` is 00:39:42 with best time 00:38:14. This suggests that using only 256 threads performed better than using all 272 threads. The best run time in the 32,768 $\times$ 32,768 case in Table 3.4.2 using 256 threads was 14:58:33, better than the best 272 threads run, 15:09:12, in Table 3.4.1. However, In the 32,768 $\times$ 32,768 the average run time using all 272 threads is around 15.5 hours, but using only 256 threads was 15.75 hours. So, for the largest problem size, there are cases in which

using all 272 threads performs better that only 256 threads.

We also consider the explicit use of all 68 KNL cores, with 1, 2, 3, or all 4 threads on each core. To achieve this, we use `KMP_AFFINITY=scatter` with 68, 136, 204, and 272 total threads, respectively. It is not necessary, with these choices, to also use `KMP_HW_SUBSETS` to allocate the desired number of threads per core explicitly. In each case we test different distributions of the MPI processes and threads across the KNL. Recall that the first core of the KNL contains cpu ids 000, 068, 136, and 204, the second core with 001, 069, 137, 205 and so on for the rest of the KNL cores. With 68 threads, `KMP_AFFINITY=scatter` uses cpu ids 000–067, thus one thread per core. For 136 threads, `KMP_AFFINITY=scatter` uses cpu ids 000–135, thus using two threads per core. The grouping of the cpu ids to the MPI processes of course depends on the number of MPI processes. For one concrete example consider using 4 MPI processes, with 34 threads per process, for a total of 136 threads. We observe that the first MPI processes has cpu ids 000–016 and 068–084, which is the first and second thread on the first 17 cores. The second MPI process has cpu ids 017–033 and 85–101, the first and second threads on the second 17 cores. Then the third and fourth MPI processes fill out the first and second threads on the remaining 34 cores accordingly.

The results from testing this behavior are shown in Table 3.4.3, with Table 3.4.3 (a) using 1 thread per core, Table 3.4.3 (b) using 2 threads per core, and Table 3.4.3 (c) using 3 threads per core. Note that Table 3.4.3 (d) which uses 4 threads per core, repeats the threads structure in Table 3.4.1 (b). In Table 3.4.3 we restrict our attention to numbers of MPI processes that divide 68, the number of KNL cores, that is, 1, 2, 4, 17, 34, 68, which are fewer than the cases listed in Table 3.4.1. Then for the cases with more than 68 threads, we maintain our number of MPI processes and simply increase the number of treads per process. We leave off the largest problem size due to the long run times. In the 2048 case, we observe that using less than 4 threads per

Table 3.4.1 Observed wall clock times in units of HH:MM:SS on 1 KNL on Stampede using all 272 threads in Cache Quadrant Configuration, MCDRAM as cache with DDR4, with two settings of `KMP_AFFINITY`.

(a) KNL – Cache Quadrant Configuration – Compact

MPI proc	1	2	4	8	16	17	34	68	136	272
Threads/proc	272	134	68	34	17	16	8	4	2	1
1024 × 1024	00:00:02	00:00:02	00:00:02	00:00:02	00:00:01	00:00:02	00:00:02	00:00:02	00:00:02	00:00:02
2048 × 2048	00:00:08	00:00:10	00:00:10	00:00:11	00:00:10	00:00:10	00:00:10	00:00:10	00:00:10	00:00:11
4096 × 4096	00:00:40	00:00:41	00:00:41	00:00:42	00:00:42	00:00:41	00:00:41	00:00:41	00:00:45	00:00:48
8192 × 8192	00:05:12	00:05:12	00:05:06	00:05:06	00:05:06	00:05:09	00:05:05	00:05:11	00:05:09	00:05:04
16384 × 16384	00:42:48	00:42:47	00:41:47	00:39:32	00:46:21	00:42:29	00:44:29	00:49:37	00:44:18	00:39:11
32768 × 32768	15:17:07	15:11:42	15:42:44	15:38:11	15:35:35	15:59:01	15:43:46	15:46:32	15:59:36	16:21:57

(b) KNL – Cache Quadrant Configuration – Scatter

MPI proc	1	2	4	8	16	17	34	68	136	272
Threads/proc	272	136	68	34	17	16	8	4	2	1
1024 × 1024	00:00:02	00:00:02	00:00:02	00:00:02	00:00:02	00:00:02	00:00:01	00:00:02	00:00:02	00:00:04
2048 × 2048	00:00:10	00:00:10	00:00:10	00:00:10	00:00:10	00:00:10	00:00:10	00:00:10	00:00:10	00:00:14
4096 × 4096	00:00:42	00:00:41	00:00:42	00:00:41	00:00:41	00:00:41	00:00:42	00:00:42	00:00:45	00:01:10
8192 × 8192	00:05:04	00:05:08	00:05:06	00:05:07	00:05:08	00:05:03	00:05:11	00:05:06	00:05:04	00:05:04
16384 × 16384	00:39:49	00:39:07	00:39:14	00:39:13	00:39:23	00:44:30	00:39:58	00:45:36	00:42:25	00:39:02
32768 × 32768	15:09:12	15:27:00	15:11:30	15:35:55	15:46:24	15:17:15	15:39:50	15:42:58	16:06:05	16:24:07

Table 3.4.2 Observed wall clock times in units of HH:MM:SS on 1 KNL on Stampede using only 256 threads in Cache Quadrant Configuration, MCDRAM as cache with DDR4, with two settings of `KMP_AFFINITY`.

(a) KNL – Cache Quadrant Configuration – Compact

MPI proc	1	2	4	8	16	32	64	128	256
Threads/proc	256	128	64	32	16	8	4	2	1
1024×1024	00:00:01	00:00:02	00:00:01	00:00:01	00:00:01	00:00:01	00:00:01	00:00:01	00:00:01
2048×2048	00:00:07	00:00:06	00:00:06	00:00:09	00:00:05	00:00:05	00:00:06	00:00:06	00:00:06
4096×4096	00:00:41	00:00:45	00:00:37	00:00:37	00:00:37	00:00:37	00:00:37	00:00:37	00:00:40
8192×8192	00:05:20	00:05:11	00:05:03	00:05:26	00:05:04	00:05:03	00:04:56	00:04:54	00:05:01
16384×16384	00:40:26	00:39:02	00:39:03	00:39:19	00:42:07	00:38:15	00:41:53	00:39:02	00:39:08
32768×32768	15:41:13	15:39:07	15:30:31	15:39:20	15:40:48	15:33:41	15:46:59	16:01:35	16:03:48

(b) KNL – Cache Quadrant Configuration – Scatter

MPI proc	1	2	4	8	16	32	64	128	256
Threads/proc	256	128	64	32	16	8	4	2	1
1024×1024	00:00:01	00:00:01	00:00:01	00:00:01	00:00:01	00:00:01	00:00:01	00:00:01	00:00:01
2048×2048	00:00:07	00:00:06	00:00:06	00:00:06	00:00:05	00:00:05	00:00:06	00:00:06	00:00:06
4096×4096	00:00:41	00:00:39	00:00:38	00:00:38	00:00:38	00:00:37	00:00:37	00:00:37	00:00:39
8192×8192	00:05:27	00:05:16	00:05:18	00:05:08	00:05:09	00:05:13	00:05:04	00:04:50	00:05:01
16384×16384	00:40:24	00:39:34	00:39:52	00:40:10	00:42:30	00:38:43	00:38:32	00:38:14	00:39:16
32768×32768	16:01:08	15:20:38	15:40:15	15:53:03	15:18:33	14:58:33	15:52:00	15:36:31	15:51:53

Table 3.4.3 Observed wall clock times in units of HH:MM:SS on 1 KNL on Stampede using only 1,2,3 and 4 threads per core in Cache Quadrant Configuration, MCDRAM as cache with DDR4, with `KMP_AFFINITY=scatter`.

(a) KNL – Cache Quadrant Configuration – Scatter – 1 Thread per core						
MPI proc	1	2	4	17	34	68
Threads/proc	68	34	17	4	2	1
1024×1024	00:00:01	00:00:01	00:00:01	00:00:01	00:00:01	00:00:01
2048×2048	00:00:05	00:00:05	00:00:05	00:00:05	00:00:05	00:00:05
4096×4096	00:00:37	00:00:37	00:00:37	00:00:43	00:00:35	00:00:36
8192×8192	00:07:01	00:06:54	00:07:01	00:06:40	00:06:22	00:06:15
16384×16384	00:38:58	00:40:38	00:39:51	00:38:32	00:38:41	00:39:08
(b) KNL – Cache Quadrant Configuration – Scatter – 2 Threads per core						
MPI proc	1	2	4	17	34	68
Threads/proc	136	68	34	8	4	2
1024×1024	00:00:01	00:00:01	00:00:01	00:00:01	00:00:01	00:00:01
2048×2048	00:00:05	00:00:05	00:00:05	00:00:05	00:00:05	00:00:05
4096×4096	00:00:35	00:00:35	00:00:35	00:00:35	00:00:34	00:00:34
8192×8192	00:04:47	00:04:45	00:04:41	00:04:40	00:04:46	00:04:46
16384×16384	00:38:11	00:38:05	00:38:12	00:38:16	00:39:32	00:39:06
(c) KNL – Cache Quadrant Configuration – Scatter – 3 Threads per core						
MPI proc	1	2	4	17	34	68
Threads/proc	204	102	51	12	6	3
1024×1024	00:00:01	00:00:01	00:00:01	00:00:01	00:00:01	00:00:01
2048×2048	00:00:05	00:00:05	00:00:05	00:00:05	00:00:05	00:00:05
4096×4096	00:00:36	00:00:35	00:00:35	00:00:35	00:00:35	00:00:36
8192×8192	00:05:01	00:05:01	00:04:58	00:04:48	00:04:45	00:04:43
16384×16384	00:37:44	00:38:04	00:37:37	00:40:58	00:38:55	00:37:29
(d) KNL – Cache Quadrant Configuration – Scatter – 4 Threads per core						
MPI proc	1	2	4	17	34	68
Threads/proc	272	136	68	16	8	4
1024×1024	00:00:02	00:00:02	00:00:02	00:00:02	00:00:01	00:00:02
2048×2048	00:00:10	00:00:10	00:00:10	00:00:10	00:00:10	00:00:10
4096×4096	00:00:43	00:00:42	00:00:37	00:00:41	00:00:40	00:00:42
8192×8192	00:05:11	00:05:08	00:05:05	00:05:05	00:04:53	00:04:49
16384×16384	00:46:33	00:43:54	00:39:02	00:39:18	00:38:18	00:38:37

core was a factor of 2 better that using all 4 threads per core for this test code. More generally, It is clear from this comparison that using 2 or 3 threads per core performs better than using 1 thread per core and slightly better than 4 threads per core. Still, using all of the threads available on the hardware was not a significant disadvantage so for this work we continue with testing all of the hardware.

Before we consider the use of Flat memory mode configurations, it is important to make more precise memory observations so that runs fit in the chosen memory resource, most importantly the 16 GB of MCDRAM. In Table 3.4.4 we present the total

memory used for combinations of MPI process and OpenMP threads in Cache Quadrant mode runs. The memory usage is observed in the code by checking the `VmRSS` field in the the special file `/proc/self/status`. The first column of Table 3.4.4 (c) repeats the memory predictions from Table 3.2.1. The close agreement with the observed memory usage in the next column confirms that the implementation of the code, with the current MPI implementation, does not have any unexpected memory usage in the 1 MPI process case. In Table 3.4.4 (a) we show the memory impact of using multi-threading parallelism only for our two smallest problem sizes. We clearly see that using more OpenMP threads does not increase significantly to the overall memory usage. In Table 3.4.4 (b) we show the memory impact of using MPI parallelism only for our two smallest problem sizes. The increase in MPI processes comes at significant cost in terms of memory in the current MPI implementation. In both problem sizes, for less than 17 MPI processes the increase in memory for the additional MPI processes has only a small effect. But, when the number of MPI processes is doubled from 17 to 34, from 34 to 68, from 68 to 134, and from 134 to 272, the total memory required doubles as well. In both cases, code runs with 16 MPI processes use less than 1 GB, but using 272 MPI processes requires 14 GB. The Intel MPI implementation version 17.0.0, the default on the Stampede KNL cluster is used, and this amount of overhead for MPI processes seems very unreasonable.

Table 3.4.4 (c) is structured with combinations of MPI processes and OpenMP threads totaling 272, as in our performance tables like Table 3.4.1. The key observation in this table is the 16,384 $\times$ 16,384 case in which the just over 8 GB memory usage for small numbers of MPI processes significantly exceeds the 16 GB of MCDRAM when 272 MPI processes are used. In Cache memory mode, exceeding the 16 GB of MCDRAM is not a problem, as the DDR4 memory is used as needed. But, when this case in run in Flat memory mode using only MCDRAM, it is not possible

Table 3.4.4 Observed total memory usage in units of GB on 1 KNL on Stampede using all 272 threads in Cache Quadrant Configuration, MCDRAM as cache with DDR4, and `KMP_AFFINITY=compact`.

KNL – Cache Quadrant Configuration – Compact											
(a) Multi-threading only, up to 272 OpenMP threads											
MPI proc	Predicted	1	1	1	1	1	1	1	1	1	1
Threads/proc	(GB)	272	134	68	34	17	16	8	4	2	1
1024×1024	< 0.100	0.076	0.078	0.076	0.080	0.082	0.084	0.084	0.087	0.100	0.098
2048×2048	0.125	0.169	0.171	0.171	0.170	0.174	0.178	0.178	0.183	0.198	0.203
(b) MPI only, up to 272 MPI processes											
MPI proc	Predicted	1	2	4	8	16	17	34	68	136	272
Threads/proc	(GB)	1	1	1	1	1	1	1	1	1	1
1024×1024	< 0.100	0.077	0.111	0.191	0.352	0.682	0.722	1.408	2.805	6.263	13.964
2048×2048	0.125	0.171	0.207	0.283	0.446	0.772	0.809	1.505	2.909	6.332	14.058
(c) Combinations of MPI processes and OpenMP threads											
MPI proc	Predicted	1	2	4	8	16	17	34	68	136	272
Threads/proc	(GB)	272	134	68	34	17	16	8	4	2	1
1024×1024	< 0.100	0.10	0.15	0.25	0.41	0.73	0.78	1.44	2.82	6.27	13.97
2048×2048	0.125	0.20	0.25	0.34	0.51	0.83	0.88	1.55	2.92	6.37	14.08
4096×4096	0.500	0.57	0.62	0.72	0.91	1.24	1.28	1.95	3.31	6.74	14.50
8192×8192	2.000	2.08	2.12	2.22	2.43	2.76	2.82	3.51	4.96	8.52	16.52
16384×16384	8.000	8.08	8.12	8.23	8.44	8.82	8.85	9.51	10.95	14.55	22.55
32768×32768	32.000	32.07	32.14	32.25	32.47	32.84	32.89	33.53	34.99	38.64	46.78

to run with 272 or 256 MPI processes. It can also be noted that the 16 GB MCDRAM size is less than the $8{,}192 \times 8{,}192$ total observed memory of 16.52 GB, but this case can be run in the 16 GB of MCDRAM as the memory required at any particular time during the run is less than 16 GB. For this reason, significant memory overhead for many MPI processes, we understand the recommendation in [43] to use 64 MPI processes or fewer. Still, as we observed in Table 3.4.1 and Table 3.4.2, there is no significant difference in run time using 272 or 256 MPI processes. We continue to include the 272 and 256 threads runs, when sensible, in the performance tables that include Flat memory mode.

3.4.2 Flat Quadrant Configuration

In this section we present results in Flat memory mode using Quadrant cluster mode. On the Stampede-KNL cluster this configuration is the Flat-Quadrant queue. There is a maximum of 40 nodes per job allocated for this queue and each user is

restricted to a maximum of 5 jobs in the queue at any one time. The max runtime for a job in this queue is 48 hours. Again, in addition to standard options in our slurm script, we specify the number of nodes (1) and the number of MPI tasks. We use `export KMP_AFFINITY` and `export OMP_NUM_THREADS` and run with `ibrun numactl --<numactlflag> <executable>`. The `numactl` command is used to control the type of memory used in Flat memory mode. Its flag `--membind=1` forces the code to use the MCDRAM memory, while the flag `--membind=0` keeps everything in the the DDR4 memory. The user may also use the flag `--preferred=1` so that the MCDRAM is used when available, then the DDR4 if necessary. It is also possible to control memory more explicitly in the code using the memkind library. These results do not show performance with a memkind implementation, as this requires additional modifications to the code by the programmer.

Tables 3.4.5–3.4.7 show the results in Flat Quadrant mode using all 272 available threads on the KNL. Table 3.4.8–3.4.10 show the results in Flat Quadrant mode using only 256 threads. In each table, two subtables are presented, subtable (a) for `KMP_AFFINITY=compact` and subtable (b) for `KMP_AFFINITY=scatter`. In Tables 3.4.5 and 3.4.8 only the MCDRAM is used. In Tables 3.4.6 and 3.4.9 the MCDRAM is used when possible, then the DDR4 is used. In Tables 3.4.7 and 3.4.10 only the DDR4 is used.

In the 16,384 $\times$ 16,384 case using MCDRAM only with 272 or 256 MPI processes, the overhead for all of the MPI process is so significant that the runs expected to require close to 8 GB of memory use more than the 16 GB available in MCDRAM. The memory observations can be found in Table 3.4.4 for the Cache Quadrant KNL configuration. We use (*) in Tables 3.4.5–3.4.7 and Tables 3.4.8–3.4.10 to indicate that the job failed due to this memory issue. The 32,768 $\times$ 32,768 case will not fit in the 16 GB of MCDRAM, but we can run it in the DDR4 memory only in

Tables 3.4.7 and 3.4.10, and with the preferred option using both MCDRAM and DDR4 in Tables 3.4.6 and 3.4.9. In both cases, the run time is significant, approaching and in excess of 24 hours. We elect not to run all combinations of MPI process and OpenMP threads in this case. Instead we run only the 1 MPI process case and the case using 68 MPI processes (or 64 in the 256 threads case). We choose this case based on the 68 cores of the KNL.

The main observation in Table 3.4.1 and Table 3.4.2 is the significantly longer run times when using the DDR4 memory rather than the MCDRAM. MCDRAM only cases, Table 3.4.1 (a), (b) and Table 3.4.2 (a), (b) average around 39 minutes, while DDR4 only cases in Table 3.4.1 (e), (f) and Table 3.4.2 (e), (f) average around 3 hours and 12 minutes. Using those values, the MCDRAM is 4.92 times faster than the DDR4, nearly the 5x faster estimated in [49]. Since the `--preferred=1` option uses the MCDRAM when able, the run times are very comparable to the MCDRAM only cases.

The most interesting cases with the `--preferred=1` option are the cases not possible here using only the MCDRAM, the $16{,}384 \times 16{,}384$ run using 272 (or 256) threads, and the $32{,}768 \times 32{,}768$ runs. In the $32{,}768 \times 32{,}768$ we can compare against the MCDRAM in Cache memory mode in Tables 3.4.1 and 3.4.2 which average around 15.5 hours for 272 threads and 15.75 hours 256 threads. Using only the DDR4 is significantly slower, close to 26 hours, but using MCDRAM when available with DDR4 in the Flat Quadrant configuration was faster than using only DDR4, but considerably more inconsistent. Run times ranged from 19.5 hours to 24 hours in the 272 and 256 threads cases. In the $16{,}384 \times 16{,}384$ run using the `--preferred=1` option with 272 threads we observe run times of 01:58:05 and 01:45:28, which are significantly longer than the 256 threads case 01:18:16 and 01:21:50. All of these are faster than the over 3 hour run times using DDR4 only, but not very close to the 38 minute run times

using MCDRAM only.

We can also make comparisons to the Cache Quadrant configuration results in Table 3.4.1 and Table 3.4.2. In Table 3.4.1 (a) in which the Cache Quadrant configuration is used the average run time across the row in the 16,384 case is 43:20, slightly worse than the corresponding average in Table 3.4.5 (a) of 42:04, and Table 3.4.6 (a) of 39:44, in which the Flat Quadrant configuration using MCDRAM only and with the `--preferred=1` option, respectively, are used. We also observe in some cases Cache Quadrant performed slightly better than Flat Quadrant, like the cases using 1 MPI process. Although from these timings the Flat Quadrant configuration using MCDRAM appears slightly faster than the Cache Quadrant configuration when the problem fits entirely in the 16 GB of MCDRAM, we would need more testing to determine if this difference is in fact significant.

The choice of `KMP_AFFINITY=scatter` or `compact` in Flat Quadrant configuration shows little impact on the run times. In each Table 3.4.5–3.4.7 and Table 3.4.8–3.4.10, subtables (a) in each uses `KMP_AFFINITY=compact`, while subtables (b) in each uses `KMP_AFFINITY=scatter`. In some cases, `compact` outperforms `scatter`, for example, the 16,384 case with 256 threads, using DDR4 only, in Table 3.4.10. The average run time across the 16,384 case in Table 3.4.10 (a), using `compact`, is 03:10:47, but in Table 3.4.10 (b), using `scatter` is 03:12:08. But in other cases, `scatter` outperforms `compact`, for example, the 16,384 case with 256 threads, using MCDRAM only, in Table 3.4.5.

3.4.3 Flat All-to-All Configuration

In this section, we present results in Flat memory mode, using All-to-All cluster mode. The Flat-All2All queue on Stampede uses this configuration. In this queue there is a maximum of 2 nodes allocated per job and each user is restricted to a

Table 3.4.5 Observed wall clock times in units of HH:MM:SS on 1 KNL on Stampede using all 272 threads in Flat Quadrant Configuration, using MCDRAM only, with two settings of `KMP_AFFINITY`.

(a) KNL – Flat Quadrant Configuration – MCDRAM – Compact

MPI proc	1	2	4	8	16	17	34	68	136	272
Threads/proc	272	136	68	34	17	16	8	4	2	1
1024 × 1024	00:00:02	00:00:03	00:00:03	00:00:02	00:00:02	00:00:01	00:00:01	00:00:02	00:00:02	00:00:02
2048 × 2048	00:00:07	00:00:13	00:00:12	00:00:09	00:00:08	00:00:10	00:00:10	00:00:10	00:00:10	00:00:11
4096 × 4096	00:00:40	00:00:40	00:00:40	00:01:06	00:00:40	00:00:40	00:00:40	00:00:41	00:00:43	00:00:47
8192 × 8192	00:06:11	00:04:53	00:04:50	00:04:49	00:04:49	00:04:47	00:04:46	00:04:48	00:04:51	00:04:56
16384 × 16384	00:43:22	00:41:26	00:43:05	00:38:09	00:43:25	00:42:31	00:42:52	00:42:34	00:41:13	(*)
32768 × 32768	—– Memory requirement exceeds MCDRAM capacity —–									

(b) KNL – Flat Quadrant Configuration – MCDRAM – Scatter

MPI proc	1	2	4	8	16	17	34	68	136	272
Threads/proc	272	136	68	34	17	16	8	4	2	1
1024 × 1024	00:00:02	00:00:02	00:00:02	00:00:02	00:00:01	00:00:02	00:00:01	00:00:02	00:00:02	00:00:02
2048 × 2048	00:00:09	00:00:10	00:00:09	00:00:10	00:00:09	00:00:10	00:00:10	00:00:10	00:00:10	00:00:11
4096 × 4096	00:00:41	00:00:41	00:00:41	00:00:40	00:00:40	00:00:40	00:00:39	00:00:41	00:00:43	00:00:47
8192 × 8192	00:04:53	00:04:51	00:04:50	00:04:49	00:04:47	00:04:47	00:04:44	00:04:47	00:04:51	00:04:58
16384 × 16384	00:38:15	00:38:18	00:38:02	00:37:58	00:38:03	00:37:52	00:42:27	00:37:10	00:36:54	(*)
32768 × 32768	—– Memory requirement exceeds MCDRAM capacity —–									

Table 3.4.6 Observed wall clock times in units of HH:MM:SS on 1 KNL on Stampede using all 272 threads in Flat Quadrant Configuration, using MCDRAM and DDR4, with two settings of `KMP_AFFINITY`. ET indicates excessive time.

(a) KNL – Flat Quadrant Configuration – Preferred – Compact

MPI proc	1	2	4	8	16	17	34	68	136	272
Threads/proc	272	136	68	34	17	16	8	4	2	1
1024 × 1024	00:00:02	00:00:02	00:00:02	00:00:02	00:00:02	00:00:01	00:00:01	00:00:02	00:00:02	00:00:06
2048 × 2048	00:00:09	00:00:10	00:00:10	00:00:10	00:00:10	00:00:06	00:00:07	00:00:08	00:00:10	00:00:11
4096 × 4096	00:00:41	00:00:41	00:00:41	00:00:40	00:00:40	00:00:40	00:00:40	00:00:40	00:00:43	00:00:45
8192 × 8192	00:04:56	00:04:53	00:04:52	00:04:50	00:04:49	00:04:46	00:04:46	00:04:48	00:04:48	00:04:57
16384 × 16384	00:44:17	00:38:18	00:38:03	00:37:59	00:37:54	00:42:56	00:37:52	00:37:35	00:42:35	01:58:05
32768 × 32768	19:25:02	ET	ET	ET	ET	ET	ET	23:01:45	ET	ET

(b) KNL – Flat Quadrant Configuration – Preferred – Scatter

MPI proc	1	2	4	8	16	17	34	68	136	272
Threads/proc	272	136	68	34	17	16	8	4	2	1
1024 × 1024	00:00:02	00:00:02	00:00:02	00:00:02	00:00:02	00:00:02	00:00:01	00:00:01	00:00:02	00:00:02
2048 × 2048	00:00:09	00:00:10	00:00:10	00:00:10	00:00:10	00:00:09	00:00:09	00:00:10	00:00:10	00:00:14
4096 × 4096	00:00:37	00:00:37	00:00:40	00:00:40	00:00:40	00:00:39	00:00:39	00:00:51	00:00:43	00:01:09
8192 × 8192	00:04:54	00:04:52	00:04:50	00:04:51	00:04:46	00:04:46	00:04:49	00:04:48	00:04:45	00:04:57
16384 × 16384	00:38:24	00:43:16	00:38:07	00:40:33	00:38:04	00:37:52	00:37:49	00:37:50	00:37:42	01:45:28
32768 × 32768	20:42:51	ET	ET	ET	ET	ET	ET	23:48:52	ET	ET

Table 3.4.7 Observed wall clock times in units of HH:MM:SS on 1 KNL on Stampede using all 272 threads in Flat Quadrant Configuration, using DDR4 only, with two settings of `KMP_AFFINITY`. ET indicates excessive time.

(a) KNL – Flat Quadrant Configuration – DDR4 – Compact

MPI proc	1	2	4	8	16	17	34	68	136	272
Threads/proc	272	136	68	34	17	16	8	4	2	1
1024 × 1024	00:00:03	00:00:03	00:00:03	00:00:03	00:00:03	00:00:02	00:00:02	00:00:05	00:00:04	00:00:05
2048 × 2048	00:00:24	00:00:24	00:00:24	00:00:34	00:00:24	00:00:24	00:00:24	00:00:35	00:00:28	00:00:32
4096 × 4096	00:02:57	00:02:55	00:02:54	00:03:25	00:02:54	00:02:54	00:02:55	00:02:57	00:03:03	00:03:11
8192 × 8192	00:29:07	00:23:54	00:23:47	00:24:03	00:24:00	00:23:44	00:23:52	00:23:55	00:25:37	00:24:48
16384 × 16384	03:16:28	03:12:20	03:11:20	03:10:59	03:11:10	03:11:20	03:12:09	03:12:53	03:14:17	03:17:07
32768 × 32768	25:55:16	ET	ET	ET	ET	ET	ET	25:56:39	ET	ET

(b) KNL – Flat Quadrant Configuration – DDR4 – Scatter

MPI proc	1	2	4	8	16	17	34	68	136	272
Threads/proc	272	136	68	34	17	16	8	4	2	1
1024 × 1024	00:00:04	00:00:04	00:00:04	00:00:03	00:00:03	00:00:03	00:00:03	00:00:03	00:00:04	00:00:05
2048 × 2048	00:00:25	00:00:21	00:00:20	00:00:20	00:00:20	00:00:24	00:00:25	00:00:25	00:00:28	00:00:33
4096 × 4096	00:02:57	00:02:56	00:02:54	00:02:53	00:02:54	00:02:53	00:02:53	00:02:57	00:03:03	00:03:13
8192 × 8192	00:24:02	00:23:56	00:23:46	00:23:44	00:23:44	00:24:04	00:23:49	00:24:17	00:24:12	00:26:31
16384 × 16384	03:12:58	03:12:13	03:11:54	03:11:54	03:11:52	03:12:01	03:12:33	03:13:32	03:14:24	03:18:04
32768 × 32768	25:59:59	ET	ET	ET	ET	ET	ET	25:52:57	ET	ET

Table 3.4.8 Observed wall clock times in units of HH:MM:SS on 1 KNL on Stampede using only 256 threads in Flat Quadrant Configuration, using MCDRAM only, with two settings of `KMP_AFFINITY`.

(a) KNL – Flat Quadrant Configuration – MCDRAM – Compact

MPI proc	1	2	4	8	16	32	64	128	256
Threads/proc	256	128	64	32	16	8	4	2	1
1024×1024	00:00:01	00:00:01	00:00:01	00:00:01	00:00:01	00:00:01	00:00:01	00:00:01	00:00:01
2048×2048	00:00:07	00:00:06	00:00:06	00:00:05	00:00:05	00:00:05	00:00:05	00:00:05	00:00:06
4096×4096	00:00:40	00:00:38	00:00:37	00:00:37	00:00:37	00:00:36	00:00:36	00:00:36	00:00:39
8192×8192	00:04:58	00:04:53	00:04:51	00:04:54	00:04:54	00:04:45	00:04:44	00:04:45	00:04:56
16384×16384	00:37:46	00:37:54	00:38:06	00:38:13	00:38:51	00:37:40	00:37:31	00:37:09	(*)
32768×32768	—– Memory requirement exceeds MCDRAM capacity —–								

(b) KNL – Flat Quadrant Configuration – MCDRAM – Scatter

MPI proc	1	2	4	8	16	32	64	128	256
Threads/proc	256	128	64	32	16	8	4	2	1
1024×1024	00:00:01	00:00:01	00:00:01	00:00:01	00:00:01	00:00:01	00:00:01	00:00:01	00:00:01
2048×2048	00:00:07	00:00:06	00:00:06	00:00:05	00:00:05	00:00:05	00:00:05	00:00:06	00:00:06
4096×4096	00:00:41	00:00:39	00:00:38	00:00:37	00:00:36	00:00:36	00:00:36	00:00:37	00:00:39
8192×8192	00:05:04	00:04:55	00:04:56	00:04:53	00:04:56	00:04:47	00:04:43	00:04:43	00:04:56
16384×16384	00:38:43	00:38:34	00:38:37	00:38:08	00:38:31	00:37:43	00:37:26	00:37:34	(*)
32768×32768	—– Memory requirement exceeds MCDRAM capacity —–								

Table 3.4.9 Observed wall clock times in units of HH:MM:SS on 1 KNL on Stampede using only 256 threads in Flat Quadrant Configuration, using MCDRAM and DDR4, with two settings of `KMP_AFFINITY`. ET indicates excessive time.

(a) KNL – Flat Quadrant Configuration – Preferred – Compact

MPI proc	1	2	4	8	16	32	64	128	256
Threads/proc	256	128	64	32	16	8	4	2	1
1024 × 1024	00:00:01	00:00:01	00:00:01	00:00:01	00:00:01	00:00:01	00:00:01	00:00:01	00:00:01
2048 × 2048	00:00:07	00:00:06	00:00:05	00:00:05	00:00:05	00:00:05	00:00:05	00:00:05	00:00:06
4096 × 4096	00:00:40	00:00:39	00:00:36	00:00:37	00:00:37	00:00:35	00:00:36	00:00:37	00:00:38
8192 × 8192	00:04:56	00:04:56	00:04:49	00:04:48	00:04:55	00:04:43	00:04:41	00:04:44	00:04:53
16384 × 16384	00:38:18	00:37:28	00:37:42	00:38:09	00:38:38	00:37:06	00:37:13	00:37:39	01:18:16
32768 × 32768	20:38:12	ET	ET	ET	ET	ET	22:04:43	ET	ET

(b) KNL – Flat Quadrant Configuration – Preferred – Scatter

MPI proc	1	2	4	8	16	32	64	128	256
Threads/proc	256	128	64	32	16	8	4	2	1
1024 × 1024	00:00:01	00:00:01	00:00:01	00:00:01	00:00:01	00:00:01	00:00:01	00:00:01	00:00:01
2048 × 2048	00:00:07	00:00:06	00:00:05	00:00:05	00:00:05	00:00:05	00:00:05	00:00:05	00:00:06
4096 × 4096	00:00:40	00:00:39	00:00:38	00:00:37	00:00:37	00:00:35	00:00:36	00:00:37	00:00:38
8192 × 8192	00:05:04	00:04:56	00:04:54	00:04:55	00:04:54	00:04:43	00:04:41	00:04:44	00:04:53
16384 × 16384	00:38:37	00:38:21	00:38:07	00:38:16	00:38:34	00:37:14	00:37:16	00:37:38	01:21:50
32768 × 32768	19:48:12	ET	ET	ET	ET	ET	21:24:51	ET	ET

Table 3.4.10 Observed wall clock times in units of HH:MM:SS on 1 KNL on Stampede using only 256 threads in Flat Quadrant Configuration, using DDR4 only, with two settings of `KMP_AFFINITY`. ET indicates excessive time.

(a) KNL – Flat Quadrant Configuration – DDR4 – Compact

MPI proc	1	2	4	8	16	32	64	128	256
Threads/proc	256	128	64	32	16	8	4	2	1
1024 × 1024	00:00:01	00:00:01	00:00:01	00:00:01	00:00:01	00:00:01	00:00:02	00:00:02	00:00:03
2048 × 2048	00:00:21	00:00:21	00:00:20	00:00:20	00:00:20	00:00:20	00:00:21	00:00:22	00:00:24
4096 × 4096	00:02:51	00:02:49	00:02:49	00:02:48	00:02:49	00:02:50	00:02:52	00:02:56	00:03:06
8192 × 8192	00:23:27	00:23:25	00:23:19	00:23:19	00:23:20	00:23:29	00:23:36	00:23:50	00:24:11
16384 × 16384	03:10:03	03:09:55	03:09:24	03:09:34	03:09:34	03:10:09	03:11:03	03:12:03	03:14:58
32768 × 32768	25:38:37	ET	ET	ET	ET	ET	25:43:24	ET	ET

(b) KNL – Flat Quadrant Configuration – DDR4 – Scatter

MPI proc	1	2	4	8	16	32	64	128	256
Threads/proc	256	128	64	32	16	8	4	2	1
1024 × 1024	00:00:01	00:00:01	00:00:01	00:00:01	00:00:01	00:00:01	00:00:02	00:00:02	00:00:03
2048 × 2048	00:00:21	00:00:21	00:00:20	00:00:20	00:00:20	00:00:20	00:00:21	00:00:22	00:00:24
4096 × 4096	00:02:51	00:02:51	00:02:48	00:02:48	00:02:49	00:02:50	00:02:52	00:02:56	00:03:03
8192 × 8192	00:23:33	00:23:30	00:23:22	00:23:16	00:23:21	00:23:32	00:23:36	00:23:46	00:24:09
16384 × 16384	03:10:11	03:10:10	03:10:08	03:09:47	03:09:53	03:10:01	03:10:59	03:11:28	03:14:28
32768 × 32768	25:43:23	ET	ET	ET	ET	ET	25:41:33	ET	ET

maximum of a single job in the queue. We omit results using 256 threads in this case due to this 1 job per user maximum that makes completing a full performance study time consuming. The max runtime for a job in this queue is 12 hours, thus none of the 32,768 $\times$ 32,768 can be run in this case.

We run in the same manner as with Flat Quadrant configuration, except in a different queue. Thus the slurm scripts reflect only the difference of the Flat-All2All queue. Tables 3.4.11–3.4.13 show the results in this mode using 272 threads.

We can compare the Quadrant and All-to-All cluster modes using the Flat memory mode. Tables 3.4.5–3.4.7 use Quadrant cluster mode, while Tables 3.4.11–3.4.13 use All-to-All cluster mode. It is clear that when using the MCDRAM only in Tables 3.4.11 and 3.4.5, that Quadrant cluster mode performed slightly better than All-to-All. This behavior could be expected as the memory addresses are distributed evenly in the All-to-All cluster mode, and are somewhat localized in the Quadrant cluster mode. In Tables 3.4.12 and 3.4.13 and Tables 3.4.6 and 3.4.7 in which the `preferred` option, and the DDR4 only are used respectively, we do not observe the same run time advantage of Quadrant cluster mode. That is, the average run times across the rows of Table 3.4.6 and 3.4.7 are comparable to those of Tables 3.4.12 and 3.4.13. From this we cannot directly conclude that Quadrant cluster mode performs better that All-to-All cluster mode.

Overall the performance results in the Flat All-to-All configuration reflect those in the Flat Quadrant configuration. In particular, the choice of `KMP_AFFINITY` as `scatter` or `compact` has no observable impact on the run times. We again observe that using MCDRAM only is almost 5x faster than using DDR4 only. Using MCDRAM only, in the cases we can run, again showed the possibility for slightly better performance when configured with Flat memory mode rather than as L3 cache in Cache memory mode. The `preferred=1` option performed like using the MCDRAM

only when the case fit in the MCDRAM. For the 32,768 × 32,768 case we readily observe the benefit to using the MCDRAM with the DDR4 in Flat memory mode from the run times more than 10% faster than using DDR4 only.

3.4.4 Baseline Results

As a baseline comparison, we use the other hardware that is presently still available on Stampede. Specifically, we use a typical CPU/KNC hybrid node on Stampede, which has two 8-core CPUs and one KNC co-processor; there are also a limited number of nodes with two KNC on Stampede, see [36] for more results. We focus on a node with one KNC, since there are more of them and they motivate comparison of one KNC to one KNL. We again note that this hardware is currently in the phasing out stages. In Section 3.4.4, we present results using the first-generation KNC in both native and symmetric mode, and in Section 3.4.4 the results using the CPUs.

KNC Studies

Unlike the standalone KNL, the KNC is only a co-processor thus needs to be used in a hybrid node that also contains a CPU as host. Stampede uses a typical CPU/KNC hybrid node arrangement with two CPUs and one KNC in the node. In this arrangement, the KNC may be used by itself in native mode, with the CPU in symmetric mode, or in offload mode. In native mode, only a executable for the Phi is required as only the resources on the Phi are used. In symmetric mode, the Phi and CPU resources on the node are used simultaneously. To run in this mode requires a CPU executable and a Phi executable. In offload mode, the CPUs serve only as facilitators to MPI communication to other units so that all calculations take place on the Phi. Offload mode was demonstrated to be inefficient for this test problem in [36] based on the restriction to multi-threading only, not MPI parallelism, in offload regions on

Table 3.4.11 Observed wall clock times in units of HH:MM:SS on 1 KNL on Stampede using all 272 threads in Flat All-to-All Configuration, using MCDRAM only, with two settings of `KMP_AFFINITY`.

(a) KNL – Flat All-to-All Configuration – MCDRAM – Compact

MPI proc	1	2	4	8	16	17	34	68	136	272
Threads/proc	272	136	68	34	17	16	8	4	2	1
1024 × 1024	00:00:02	00:00:02	00:00:02	00:00:02	00:00:02	00:00:01	00:00:01	00:00:01	00:00:02	00:00:02
2048 × 2048	00:00:10	00:00:10	00:00:10	00:00:10	00:00:10	00:00:10	00:00:09	00:00:10	00:00:10	00:00:11
4096 × 4096	00:00:43	00:00:42	00:00:42	00:00:42	00:00:42	00:00:37	00:00:38	00:00:38	00:00:38	00:00:38
8192 × 8192	00:05:18	00:05:21	00:05:12	00:05:11	00:05:09	00:05:06	00:05:10	00:05:05	00:05:06	00:06:05
16384 × 16384	00:41:23	00:41:07	00:40:51	00:40:46	00:45:51	00:40:59	00:42:59	00:40:48	00:40:50	(*)
32768 × 32768	—– Memory requirement exceeds MCDRAM capacity —–									

(b) KNL – Flat All-to-All Configuration – MCDRAM – Scatter

MPI proc	1	2	4	8	16	17	34	68	136	272
Threads/proc	272	136	68	34	17	16	8	4	2	1
1024 × 1024	00:00:02	00:00:02	00:00:02	00:00:02	00:00:02	00:00:02	00:00:01	00:00:02	00:00:02	00:00:01
2048 × 2048	00:00:10	00:00:10	00:00:10	00:00:10	00:00:09	00:00:05	00:00:05	00:00:10	00:00:06	00:00:06
4096 × 4096	00:00:43	00:00:42	00:00:42	00:00:42	00:00:42	00:00:42	00:00:42	00:00:42	00:00:44	00:00:40
8192 × 8192	00:05:15	00:05:17	00:05:14	00:05:12	00:05:10	00:05:09	00:05:05	00:05:06	00:05:06	00:05:17
16384 × 16384	00:41:27	00:41:10	00:40:53	00:41:01	00:40:47	00:45:11	00:43:50	00:40:51	00:44:54	(*)
32768 × 32768	—– Memory requirement exceeds MCDRAM capacity —–									

Table 3.4.12 Observed wall clock times in units of HH:MM:SS on 1 KNL on Stampede using all 272 threads in Flat All-to-All Configuration, using MCDRAM and DDR4, with two settings of `KMP_AFFINITY`. ET indicates excessive time.

(a) KNL – Flat All-to-All Configuration – Preferred – Compact

MPI proc	1	2	4	8	16	17	34	68	136	272
Threads/proc	272	136	68	34	17	16	8	4	2	1
1024 × 1024	00:00:02	00:00:02	00:00:01	00:00:02	00:00:02	00:00:01	00:00:01	00:00:02	00:00:02	00:00:02
2048 × 2048	00:00:10	00:00:10	00:00:10	00:00:10	00:00:10	00:00:10	00:00:10	00:00:09	00:00:10	00:00:11
4096 × 4096	00:00:42	00:00:42	00:00:42	00:00:43	00:00:42	00:00:42	00:00:41	00:00:41	00:00:45	00:00:48
8192 × 8192	00:05:17	00:05:15	00:05:12	00:05:11	00:05:09	00:05:10	00:05:08	00:05:09	00:05:11	00:05:18
16384 × 16384	00:41:35	00:41:22	00:40:51	00:40:41	00:40:45	00:40:50	00:40:50	00:45:18	00:40:54	01:27:37
32768 × 32768	ET	ET	ET	ET	ET	ET	ET	ET	ET	ET

(b) KNL – Flat All-to-All Configuration – Preferred – Scatter

MPI proc	1	2	4	8	16	17	34	68	136	272
Threads/proc	272	136	68	34	17	16	8	4	2	1
1024 × 1024	00:00:02	00:00:02	00:00:02	00:00:02	00:00:02	00:00:02	00:00:01	00:00:01	00:00:02	00:00:02
2048 × 2048	00:00:08	00:00:08	00:00:10	00:00:10	00:00:10	00:00:10	00:00:10	00:00:10	00:00:11	00:00:11
4096 × 4096	00:00:42	00:00:42	00:00:42	00:00:42	00:00:40	00:00:42	00:00:42	00:00:43	00:00:45	00:00:49
8192 × 8192	00:05:18	00:05:16	00:05:13	00:05:11	00:05:10	00:05:10	00:05:10	00:05:10	00:05:12	00:05:18
16384 × 16384	00:42:07	00:41:01	00:45:46	00:45:15	00:45:37	00:45:20	00:40:50	00:45:28	00:40:50	01:24:08
32768 × 32768	ET	ET	ET	ET	ET	ET	ET	ET	ET	ET

Table 3.4.13 Observed wall clock times in units of HH:MM:SS on 1 KNL on Stampede using all 272 threads in Flat All-to-All Configuration, using DDR4 only, with two settings of `KMP_AFFINITY`. ET indicates excessive time.

(a) KNL – Flat All-to-All Configuration – DDR4 – Compact

MPI proc	1	2	4	8	16	17	34	68	136	272
Threads/proc	272	136	68	34	17	16	8	4	2	1
1024 × 1024	00:00:04	00:00:04	00:00:03	00:00:03	00:00:03	00:00:03	00:00:03	00:00:03	00:00:04	00:00:07
2048 × 2048	00:00:22	00:00:22	00:00:22	00:00:24	00:00:24	00:00:25	00:00:35	00:00:25	00:00:28	00:00:33
4096 × 4096	00:02:55	00:02:56	00:02:53	00:02:53	00:02:54	00:02:54	00:02:55	00:02:57	00:03:03	00:03:57
8192 × 8192	00:24:52	00:25:30	00:23:47	00:23:47	00:23:48	00:23:49	00:23:49	00:25:10	00:24:15	00:24:50
16384 × 16384	03:13:21	03:12:28	03:11:52	03:11:57	03:11:15	03:11:51	03:11:59	03:13:01	03:15:34	03:18:11
32768 × 32768	ET	ET	ET	ET	ET	ET	ET	ET	ET	ET

(b) KNL – Flat All-to-All Configuration – DDR4 – Scatter

MPI proc	1	2	4	8	16	17	34	68	136	272
Threads/proc	272	136	68	34	17	16	8	4	2	1
1024 × 1024	00:00:04	00:00:04	00:00:04	00:00:04	00:00:03	00:00:03	00:00:03	00:00:03	00:00:04	00:00:06
2048 × 2048	00:00:25	00:00:24	00:00:24	00:00:24	00:00:24	00:00:24	00:00:25	00:00:26	00:00:28	00:00:33
4096 × 4096	00:02:57	00:02:56	00:02:54	00:02:54	00:02:54	00:02:53	00:02:55	00:02:57	00:03:02	00:03:53
8192 × 8192	00:24:02	00:26:01	00:23:48	00:23:48	00:25:26	00:23:44	00:25:24	00:23:57	00:24:15	00:26:03
16384 × 16384	03:14:42	03:12:44	03:12:23	03:11:43	03:11:28	03:12:01	03:12:56	03:14:01	03:15:47	03:19:02
32768 × 32768	ET	ET	ET	ET	ET	ET	ET	ET	ET	ET

the Phi. As a result the performance in offload mode is roughly consistent with the result for 1 MPI process and 240 threads per process in Table 3.4.14. We restrict our attention in this work to KNC in native and symmetric mode as a comparison to KNL results. In Table 3.4.14 we present results for native mode. In Table 3.4.15 we present results for symmetric mode.

The hybrid MPI+OpenMP code in C was compiled for the KNC on Stampede using the Intel compiler version 15.0.2 with flags `-c99 -Wall -O3 -openmp -mmic` (with OpenMP version 4.0) and Intel MPI version 5.0.2. We use default settings in the run script except we use the normal-mic partition, `MIC_PPN` for the number of MPI processes, and `MIC_OMP_NUM_THREADS` for the number of OpenMP threads. The run command used for native mode is `ibrun.symm -m <mic_executable>`. The run command `ibrun.symm -c <cpu_executable> -m <mic_executable>` is used for symmetric mode. For symmetric mode we additionally specify `KMP_AFFINITY` and `OMP_NUM_THREADS` for the CPU.

We also note our use of `KMP_STACKSIZE` to control the maximum thread stacksize. In our case this is done with the line in the slurm script `export KMP_STACKSIZE=32m` which sets the maximum stacksize of each thread to 32 MB.

In these studies, we tested hybrid MPI+OpenMP code in native mode on 60 of the 61 KNC cores, leaving one core for the operating system. To do this, with a maximum of 4 threads per core, we set MPI processes times OpenMP threads always equal to 240. Our choice is based on studies in [36] that demonstrate strong scalability when using increasing numbers of cores on the KNC, leading to the conclusion that one should use all or nearly all available hardware on the KNC.

In Table 3.4.14 we show results for different `KMP_AFFINITY` in native mode on the KNC. In native mode, we are restricted to the 8 GB of GDDR5 memory on board the KNC. Examining the largest possible case, 8,192 $\times$ 8,192, the best run time is

21:28. Behavior for the different `KMP_MIC_AFFINITY` settings is most consistent in the `KMP_MIC_AFFINITY=compact` case.

In Table 3.4.15 we show results for different `KMP_AFFINITY` in symmetric mode on the KNC. We use CPU `KMP_AFFINITY` set to `compact` with `KMP_MIC_AFFINITY` set to `compact` or `scatter`. We run with a total of 16 MPI process and OpenMP threads on the CPU and a total of 240 on the KNC coprocessor. In doing this we maintain the number of MPI processes on the CPU and MIC. The `KMP_MIC_AFFINITY` setting does not show observable impact on the run time. In the 8,192 $\times$ 8,192, as in native mode, `compact` shows a bit more consistent results. Using the 8,192 $\times$ 8,192 case we observe that the best run time in native mode is 1.4 times slower than symmetric mode, 21:28 to 15:21. More importantly, larger problem sizes can be solved with symmetric mode. The 32,768 $\times$ 32,768 case is omitted due to excessively long run times.

When we compare the best results of the KNC against the best results of the KNL we observe that the KNL is significantly faster when the MCDRAM is used. In the 8,192 $\times$ 8,192 case the KNL run times under 5 minutes were 4.5 times faster than the KNC in native mode, and 3.3 times faster than the KNC in symmetric mode. But, without using the MCDRAM on the KNL run times were 23 minutes, longer than the KNC in native and symmetric modes. In the 16,384 $\times$ 16,384 case the KNL run times near 37 minutes were almost 3 times faster than the KNC symmetric mode result. Again without using the MCDRAM, the KNL was 1.8 times slower than the KNC in symmetric mode. This provides again a compelling case for making code memory-efficient, so that it fit into the MCDRAM of the KNL.

CPU Studies

This section presents results using only the CPUs on the node. The hybrid MPI+OpenMP code in C was compiled for CPUs on Stampede using the Intel com-

Table 3.4.14 Observed wall clock times in units of MM:SS on 1 KNC on Stampede using only 240 threads in native mode, GDDR5 on Phi, with two settings of `KMP_AFFINITY`.

(a) KNC – Stampede – Native Mode – Compact

MPI proc	1	2	4	8	15	16	30	60	120	240
Threads/proc	240	120	60	30	16	15	8	4	2	1
1024×1024	00:00:04	00:00:03	00:00:02	00:00:02	00:00:03	00:00:02	00:00:03	00:00:03	00:00:06	00:00:20
2048×2048	00:00:20	00:00:16	00:00:15	00:00:14	00:00:22	00:00:15	00:00:22	00:00:22	00:00:27	00:00:55
4096×4096	00:02:10	00:01:46	00:01:42	00:01:41	00:02:31	00:01:41	00:02:52	00:02:31	00:02:42	00:03:40
8192×8192	00:28:45	00:28:41	00:29:31	00:26:33	00:21:28	00:26:26	00:24:20	00:23:10	00:23:04	00:29:29
16384×16384	— Memory requirement exceeds KNC on chip capacity —									
32768×32768	— Memory requirement exceeds KNC on chip capacity —									

(b) KNC – Stampede – Native Mode – Scatter

MPI proc	1	2	4	8	15	16	30	60	120	240
Threads/proc	240	120	60	30	16	15	8	4	2	1
1024×1024	00:00:04	00:00:03	00:00:02	00:00:02	00:00:03	00:00:02	00:00:03	00:00:03	00:00:06	00:00:20
2048×2048	00:00:20	00:00:16	00:00:15	00:00:14	00:00:22	00:00:15	00:00:22	00:00:22	00:00:27	00:00:56
4096×4096	00:02:08	00:01:47	00:01:42	00:01:41	00:03:08	00:01:42	00:02:31	00:02:32	00:02:42	00:03:44
8192×8192	00:31:18	00:29:53	00:29:20	00:26:41	00:21:33	00:26:34	00:22:58	00:35:19	00:22:49	00:29:19
16384×16384	— Memory requirement exceeds KNC on chip capacity —									
32768×32768	— Memory requirement exceeds KNC on chip capacity —									

Table 3.4.15 Observed wall clock times in units of MM:SS on 1 KNC, 2 CPU in a single node on Stampede using only 240 threads in symmetric mode, GDDR5 on Phi, DDR3 on host, with two settings of `KMP_AFFINITY`. ET indicates excessive run time.

(a) KNC – Stampede – Symmetric Mode – Compact					
MPI proc	1	2	4	8	16
Threads/proc on CPU	16	8	4	2	1
Threads/proc on MIC	240	120	60	30	15
1024×1024	00:00:01	00:00:01	00:00:01	00:00:01	00:00:01
2048×2048	00:00:09	00:00:09	00:00:09	00:00:09	00:00:09
4096×4096	00:01:22	00:01:23	00:01:22	00:01:22	00:01:22
8192×8192	00:16:39	00:16:25	00:16:28	00:15:37	00:15:27
16384×16384	01:55:48	02:01:18	02:06:42	01:48:33	01:48:35
32768×32768	ET	ET	ET	ET	ET
(b) KNC – Stampede – Symmetric Mode – Scatter					
MPI proc	1	2	4	8	16
Threads/proc on CPU	16	8	4	2	1
Threads/proc on MIC	240	120	60	30	15
1024×1024	00:00:02	00:00:02	00:00:01	00:00:01	00:00:01
2048×2048	00:00:09	00:00:09	00:00:09	00:00:09	00:00:09
4096×4096	00:01:23	00:01:23	00:01:22	00:01:22	00:01:22
8192×8192	00:17:11	00:16:57	00:16:21	00:15:35	00:15:21
16384×16384	02:12:35	02:12:11	02:06:10	01:48:52	01:48:29
32768×32768	ET	ET	ET	ET	ET

piler version 15.0.2 with flags `-c99 -Wall -O3 -openmp` (with OpenMP version 4.0) and Intel MPI version 5.0.2. We use default settings in the run script and use the standard run command `ibrun`.

Table 3.4.16 reports the baseline performance results using only the two 8-core CPUs in one node presently available on Stampede, for comparison with the Intel Xeon Phi studies. In the studies using 1 CPU node, we tested hybrid MPI+OpenMP code such that we utilized all 16 cores on Stampede. These studies use the DDR3 RAM available on Stampede.

Results here show undesired treatment of the MPI process and threads distribution despite our explicit control of thread placement with `KMP_AFFINITY`. Using `KMP_AFFINITY=compact` we observe in the logs the correct distribution of threads and run times that match for up to the 8,192 case, and we observe approximately the same run time for all combinations of processes and threads. However in the 16,384

case we see improper placement for the 2 MPI processes, 8 threads per process in which all of the threads are places on the first two cores, leaving the other 14 cores idle. Similar behavior is also observed with 4 MPI processes and 4 threads per process in that case. Using `KMP_AFFINITY=scatter` thread placements are again flawed, with the best run time results occurring for MPI only parallelism. In both cases, the best run times occur using MPI parallelism only, so we use only these results to compare against KNC and KNL runs.

We compare back to Table 3.4.14 and observe that for most combinations of MPI processes and threads, the KNC run times using GDDR5 are comparable to the CPU run times using DDR3 on Stampede in Table 3.4.16. The KNL best times are only slightly faster, 21.5 minutes compared with 21.75 minutes. Making full use of the hybrid node with 1 KNL and two 8-core CPUs with the KNL in symmetric mode we are able to beat the CPU only again, but by a larger margin, 15.5 minutes compared to 21.75 minutes.

Now comparing back to the KNL, using MCDRAM in the 8,192 case with run times of 4.75 minutes, is more than 4.5x faster than the CPU only runs. In the 16,384 case using MCDRAM on the KNL run times were at best 37 minutes, which is more than 4.7x faster than the CPU only runs. Notably, when using only the DDR4 memory of the KNL node, with the 8,192 case run times of 23 minutes the CPU run times of less than 22 minutes were better. The same was true in the 16,384 case run times were over 3 hours, but the CPU only run times were less than 3 hours in the best cases. This emphasizes the obvious need to take advantage of the MCDRAM in KNL chip.

Table 3.4.16 Observed wall clock times in units of HH:MM:SS on one CPU node with two 8-core CPUs on Stampede using 16 threads, DDR3 memory on the node, with two settings of `KMP_AFFINITY`. ET indicates excessive run time.

(a) CPU only – DDR3 – Stampede – Compact					
MPI proc	1	2	4	8	16
Threads/proc	16	8	4	2	1
1024×1024	00:00:01	00:00:01	00:00:01	00:00:01	00:00:01
2048×2048	00:00:20	00:00:20	00:00:20	00:00:19	00:00:20
4096×4096	00:02:43	00:02:43	00:02:42	00:02:42	00:02:42
8192×8192	00:21:56	00:21:55	00:21:46	00:21:51	00:21:47
16384×16384	02:57:04	08:04:49	05:54:44	02:55:21	02:55:05
32768×32768	ET	ET	ET	ET	ET
(b) CPU only – DDR3 – Stampede – Scatter					
MPI proc	1	2	4	8	16
Threads/proc	16	8	4	2	1
1024×1024	00:00:12	00:00:06	00:00:04	00:00:03	00:00:01
2048×2048	00:01:53	00:00:58	00:00:41	00:00:41	00:00:20
4096×4096	00:16:01	00:07:27	00:05:31	00:05:25	00:02:43
8192×8192	00:22:32	01:00:16	00:44:48	00:43:37	00:21:49
16384×16384	02:56:45	08:01:16	05:56:53	05:51:16	02:55:07
32768×32768	ET	ET	ET	ET	ET

3.5 Flat All-to-All Configuration on Grover KNL

For our testing we had access to Grover, a KNL testing server, at the University of Oregon. Grover is a single node server with a Intel Xeon Phi 7250 pre-production model running as a standalone processor. The Grover KNL is the same model number as the Stampede-KNL cluster models.

The Grover KNL is configured in Flat All-to-All mode. The hybrid MPI+OpenMP code in C was compiled for the KNL on Grover using the Intel compiler version 16.0.4 with the flags `-xMIC-AVX512 -O3 -std=c99 -Wall -mkl -qopenmp` (with OpenMP version 4.0) and Intel MPI version 5.0.3. The code is run on Grover using the `numactl` command. As on Stampede, the flag `--membind=1` forces the code to use the MC-DRAM memory, while the flag `--membind=0` has it use the DDR4 memory.

Table 3.5.1 shows memory observations for different combinations of MPI processes and OpenMP threads on Grover. The behavior is significantly better than that of Stampede in terms of the amount of overhead per MPI processes.

Table 3.5.2 shows the performance results on the Grover KNL. These results show essentially the same behavior of those on Stampede. For full discussion of the Grover results see [35].

Table 3.5.1 Observed total memory usage in units of GB on the Grover KNL using all 272 threads in Flat All to All configuration, using MCDRAM only, and `KMP_AFFINITY=scatter`.

Grover – KNL – (Flat All to All Configuration)											
Combinations of MPI processes and OpenMP threads											
MPI proc	Predicted	1	2	4	8	16	17	34	68	136	272
Threads/proc	memory (GB)	272	134	68	34	17	16	8	4	2	1
1024×1024	< 0.100	0.09	0.10	0.12	0.16	0.28	0.28	0.51	1.01	2.14	4.38
2048×2048	0.125	0.18	0.19	0.22	0.28	0.37	0.38	0.62	1.09	2.26	4.47
4096×4096	0.500	0.56	0.57	0.59	0.66	0.77	0.77	1.00	1.48	2.64	4.86
8192×8192	2.000	2.06	2.07	2.10	2.17	2.30	2.31	2.60	3.13	4.40	6.88
16384×16384	8.000	8.06	8.07	8.10	8.17	8.31	8.31	8.59	9.16	10.47	12.97

Table 3.5.2 Observed wall clock times in units of MM:SS on the Grover KNL using all 272 threads in Flat All-to-All mode, MCDRAM or DDR4, with two settings of `KMP_AFFINITY`. ET indicates excessive time.

(a) KNL – Flat All-to-All Configuration – MCDRAM – Compact										
MPI proc	1	2	4	8	16	17	34	68	136	272
Threads/proc	272	136	68	34	17	16	8	4	2	1
1024×1024	00:01	00:01	00:01	00:01	00:01	00:01	00:01	00:01	00:01	00:02
2048×2048	00:06	00:06	00:06	00:06	00:06	00:06	00:06	00:08	00:07	00:08
4096×4096	00:41	00:41	00:41	00:41	00:41	00:41	00:41	00:40	00:42	00:46
8192×8192	05:15	05:14	05:13	05:13	05:12	05:11	05:13	05:13	05:18	05:30
16384×16384	43:46	41:14	41:09	41:11	40:55	40:59	40:55	40:52	41:32	41:38
(b) KNL – Flat All-to-All Configuration – MCDRAM – Scatter										
MPI proc	1	2	4	8	16	17	34	68	136	272
Threads/proc	272	136	68	34	17	16	8	4	2	1
1024×1024	00:01	00:01	00:01	00:01	00:01	00:01	00:01	00:01	00:01	00:02
2048×2048	00:06	00:06	00:06	00:06	00:07	00:06	00:06	00:07	00:07	00:08
4096×4096	00:41	00:41	00:41	00:40	00:41	00:41	00:40	00:40	00:42	00:46
8192×8192	05:15	05:16	05:15	05:14	05:12	05:11	05:12	05:14	05:17	05:30
16384×16384	41:33	41:15	41:37	41:50	40:58	40:59	40:51	41:09	41:01	41:43
(c) KNL – Flat All-to-All Configuration – DDR4 – Scatter										
MPI proc	1	2	4	8	16	17	34	68	136	272
Threads/proc	272	136	68	34	17	16	8	4	2	1
1024×1024	00:02	00:02	00:02	00:02	00:02	00:02	00:02	00:02	00:03	00:04
2048×2048	00:22	00:21	00:21	00:21	00:21	00:21	00:22	00:23	00:23	00:25
4096×4096	02:58	02:57	02:57	02:56	02:56	02:53	02:56	02:58	03:02	03:12
8192×8192	24:09	24:08	24:00	24:02	24:02	23:58	24:02	24:09	24:27	25:09
16384×16384	194:37	ET	ET	ET	ET	ET	ET	193:43	ET	ET
(d) KNL – Flat All-to-All Configuration – DDR4 – Compact										
MPI proc	1	2	4	8	16	17	34	68	136	272
Threads/proc	272	136	68	34	17	16	8	4	2	1
1024×1024	00:02	00:02	00:02	00:02	00:02	00:01	00:02	00:02	00:02	00:04
2048×2048	00:22	00:21	00:22	00:21	00:22	00:21	00:21	00:22	00:24	00:26
4096×4096	02:57	02:57	02:56	02:56	02:57	02:55	02:55	02:58	03:03	03:11
8192×8192	24:09	24:07	24:00	24:00	23:59	23:58	24:00	24:08	24:29	25:11
16384×16384	193:52	ET	ET	ET	ET	ET	ET	193:58	ET	ET

3.6 Conclusions and Outlook

The second-generation Intel Xeon Phi Knights Landing is a very promising new many-core processor, in particular because it can be used as standalone CPU, which can be useful for computationally intensive uses in individual research laboratories and not just supercomputing centers. However, the different memory and cluster modes available and the flexibility of their configuration options add layers of decision making for researchers running their code. In fact, most of these configurations needs to be chosen at boot time, so an individual researchers need to pick how to set up a KNL in their labs. We use a classical test problem that uses memory-bound code as a first test on how to run on the KNL, and in particular, what KNL configuration modes to use. While memory-bound, the communication needs among the MPI processes and OpenMP threads in the hybrid code are relatively light and highly structured. The next step will be to test a code with more complex communication requirements. Each of the available configurations of the KNL on the Stampede cluster are examined, with the exception of the SNC cluster modes that require code modifications by the user. We solve an assortment of different problem sizes including the largest possible mesh, a 32,768 $\times$ 32,768 mesh requiring 8 GB of memory, to provide stress on the memory access.

Hybrid code, with both MPI and OpenMP parallelism, is important to capturing the full potential of the KNL, even using a single KNL node, since neither MPI-only nor OpenMP-only runs were the best in most studies.

Using a single KNL node we did not see any consistent benefit to using only 256 threads (16 threads left free) versus all 272 threads (68 cores with 4 threads per cores) for this code. We observed that in some cases using 256 threads performed better than 272 threads, but this was not consistent across the board. It is recommended in [43] that only 1 or 2 threads per core be used, so we ran a small study for our code and

observed that 1 thread per core was not the best, and 2 and 3 threads per core were slightly better than 4 threads per core. Still, there was not a significant disadvantage, so we continued to test using all (272 threads) or almost all (256 threads) of the hardware in the KNL node in this work. In this sense, the demonstrations here are intended to stress the whole chip and put pressure on its 2D mesh structure for memory access.

Also suggested in [43] was using a maximum of one MPI task per core or even fewer. We demonstrate with memory monitoring that at least the current MPI implementation used on Stampede has a very significant memory overhead with each process. This may be the underlying reason for the suggestion in [43].

The differences in the cluster modes tested, Quadrant cluster mode and All-to-All cluster mode, were not very significant for this test code, but the better performance of Quadrant cluster mode could be observed in some cases.

Configuring the KNL in Cache memory mode, in which the high-performance MCDRAM is set as a L3 cache, gives the user excellent performance, without changes to their code. This is good performance from runs that require less than the 16 GB available of MCDRAM, and showed the best performance in our cases for the problems larger than 16 GB. We observed slightly better performance in Flat memory mode configurations of the KNL compares to Cache memory mode in the cases that required less than the 16 GB available of MCDRAM.

The on-chip high bandwidth memory MCDRAM performs almost 5x better that the DDR4 on the KNL. Examples of this are the configurations of the KNL with Flat memory mode: Using MCDRAM, the run time is about 40 minutes in the 16,384 case, but with DDR4 over 3 hours of run time is needed. This matches the estimated performance improvement of 5x based on the bandwidth difference [49].

We used the `--preferred=1` option of the `numactl` command to take advantage of both memory types in Flat memory mode. Performance that was significantly better than using DDR4 only confirms to us that the user could, with careful control, take full advantage of both memory types in a Flat memory mode configuration and see significant performance improvements. We would expect that the performance could be better than or at least competitive with that of Cache memory mode configurations, if executed efficiently. Since Cache memory mode does not require the user to manage the memory used explicitly in the code it is certainly easier. Recall that in Cache memory mode there is only 96 GB total memory available, whereas in Flat mode all 112 GB are available.

In Flat mode, using the MCDRAM if available and DDR4 otherwise (`preferred=1` setting with `numactl`) enables the use of all memory available. But, comparing to Cache mode results when more than 16 GB of memory are required we observed that 19.5 hour run time was not competitive with the 15.5 hour run time in Cache mode. This suggests that Cache mode is clearly preferred in cases in which the user wants to use all available 112 GB of memory on the KNL.

Based on these tests, Stampede users should be inclined to run using the most prominent KNL configuration on the Stampede system, the Cache Quadrant configuration. This matches the recommendation to use the Cache Quadrant configuration as default in [43]. The associated queues are the develop and the normal queues of the Stampede-KNL cluster. The normal queue is the largest on the cluster with the most accommodating limits per job of any of the queues, an 80 node maximum per job and 10 job maximum per user. The develop queue offers a nice alternative for short jobs less than 2 hours and code testing purposes in this desired KNL configuration. The combination of Cache memory mode and Quadrant cluster mode takes full advantage of the high-performance MCDRAM with the least amount of work from

the standpoint of the user.

If a specific code requires less than 16 GB of memory, the `Flat_Quadrant` queue is a good alternative and showed the ability perform with the Cache memory mode, when using explicitly the MCDRAM through `numactl`. That queue allows a 40 node maximum per job with a 5 job maximum per user. Our results do not demonstrate any significant advantage to the choice of `KMP_AFFINITY=compact` or `KMP_AFFINITY=scatter`. We used `KMP_AFFINITY=scatter` and the choice of number of threads as a way to restrict to particular number of threads per processes. This code does not require significant or very demanding communication in its implementation. Communication only takes place at the MPI level between 'neighboring' MPI processes. This could explain that lack of impact on the run time by the choice of `KMP_AFFINITY`.

In this work we restricted our attention to a single KNL, and more demands for communication and the network between nodes can be tested using more than one KNL. In some initial testing, preliminary observations indicate excellence performance using multiple KNL nodes, without performance drawbacks to using large numbers of MPI processes. In using more than one node, the recommendation in [43] to leave some cores free could be revisited.

We are in the process of running tests using more sophisticated real-world application code, in particular a system of coupled, non-linear, time-dependent PDEs that simulate the calcium ion dynamics within a heart cell [14, 22, 46]. This code requires more demanding and significant communication and is more computationally intensive than the test problem used here. This application code will put additional burden on the MPI communication and the balance with OpenMP threads.

An additional opportunity to tune the code for the KNL specifically could test the improvement from first-generation Phi to second-generation Phi in terms of the number of Vector Processing Units. Each core of the KNL has 2 VPUs as described in

Section 3.3.1 which doubled the number VPUs per code from the first-generation Phi. Results in [36, Table 1.6.2] show that for the KNC, with only 1 VPU per code, manual loop unrolling to match the VPU structure improved the performance by a factor of 2 for the three-dimensional version of our test problem. We are also in the process of investigating the effect of manual loop unrolling to improve the vectorization of the code for the KNL. Initial results show the potential for performance in the presence of manual loop unrolling to improve the vectorization of the code for the KNLs 2 VPUs per core. This work focused, by comparison, on the performance opportunities of the KNL that are achievable without changes in the code.

Testing against the baseline hardware of the current Stampede cluster, even though this equipment is being phased out, we confirmed the significant performance improvements of the KNL. For problems using less than 16 GB, in which the high-performance MCDRAM is used, the KNL was more than 4.5 times faster than the two 8-core CPUs on a Stampede node, 4.5 times faster than one KNC in native mode, and 3.3 times faster than one KNC in symmetric mode with the two 8-core CPUs. Notably, runs on the KNL that did not take advantage of the MCDRAM performed worse that the KNC and CPU results across the board. This further emphasizes the obvious need to take advantage of the MCDRAM memory on the KNL chip.

CHAPTER 4

CICR SIMULATION ON THE KNL

This chapter studies the performance of single and multiple KNL for the calcium application code. The content of this chapter is intended for [16].

4.1 Introduction

In this chapter, we study the performance of the calcium induced calcium reduce (CICR) application from Chapter 2 on the KNL. We use the results of Chapter 3 to guide our initial choice of configuration. We demonstrate the feasibility of porting and tuning special purpose application code to a single KNL and demonstrate the performance of multiple KNL. We demonstrate the need for implementing multi-threading in all time consuming portions of the code by showing results also for an intermediate version of OpenMP parallelization. Concretely, this chapter demonstrates the scalability of the MPI and OpenMP implementations, investigate the balance of MPI processes versus OpenMP threads, and the choice of number of threads per core to use on the KNL. Finally, we show the scalability of the code using more than one KNL.

Based on our findings in Chapter 3, we make initial decisions for our CICR runs on the KNL. Given the significantly better performance using the high-performance on-chip MCDRAM, we focus our attention to problem sizes that fit in the 16 GB of MCDRAM. Since we focus on runs that fit in the MCDRAM, we use the Flat Quadrant KNL configuration using the MCDRAM only. We do not explore the Flat All-to-All configuration in this chapter, as it did not perform better than Flat Quadrant in Chapter 3 and has the very restrictive queue limitation on Stampede of one job in the queue per user at any time.

By restricting our attention to mesh sizes for CICR that fit entirely into the on-chip memory of the KNL, we can accommodate two mesh refinements more than the $32 \times 32 \times 128$ mesh used in the studies of Chapter 2. The trade-off for the use of the finer meshes is the long run times for each simulation. For this reason, and the limitations of our XSEDE allocation, we run the performance studies in this chapter to a final time of 10 ms, rather than the full simulation time from Chapter 2 of 1000 ms. To capture our recently studied model components we choose parameter sets from Case C: Blowup Case with $\omega = 10$ in Section 2.5.2 for the performance studies in this chapter. In the CPU runs that produced the results in Chapter 3 we did not observe very significant difference in run time using different values for ω.

We concretely refer to Stampede in the Texas Advanced Computing Center (TACC) at the University of Texas at Austin (`www.tacc.utexas.edu`). in this work, since many researchers, e.g., U.S. based faculty, can apply for allocations through XSEDE (`www.xsede.org`) [51]. At the time of this writing, the Stampede-KNL cluster at TACC has 504 available KNL nodes, but the upcoming Stampede 2 cluster is in development and expected to be released in Fall 2017 will include almost 6,000 nodes.

The MPI code in C was compiled for the KNL on the Stampede-KNL cluster login node using the Intel compiler version 17.0.0 with flag for the KNL `-xMIC-AVX512` and the other flags `-O3 -std=c99 -Wall` and Intel MPI version 17.0.0. For the the hybrid MPI+OpenMP code we add the flag `-qopenmp` and note again the Intel compiler version 17.0.0, with OpenMP version 4.5, and Intel MPI version 17.0.0.

In this chapter, the notation (**) in results tables indicates that for the given case the level of parallelization required is too large for the given mesh. In particular, if there are more MPI processes than possible slices in the z-direction of the mesh, which constrains the parallelism in the implementation, the case is impossible. Then, for the mesh $16 \times 16 \times 64$, no more than 64 MPI processes can be used. Similarly, for

the $32 \times 32 \times 128$ mesh, 128 MPI processes is the maximum possible.

This chapter is organized as follows. Section 4.2 describes the numerical method implemented for the CICR application problem. Section 4.3 begins the study on a single KNL with testing the MPI only code for scalability and reporting memory usage observations. Section 4.4 introduces an initial MPI+OpenMP implementation that uses multi-threading for the most time consuming portion of the code and includes our first MPI versus OpenMP tests. Section 4.5 presents a second MPI+OpenMP implementation that performs significantly better than the first MPI+OpenMP implementation as a result of additional multi-threading around the reaction term computations in the code. We assess the balance of MPI process to OpenMP threads, the number of threads per core using all 68 or only 64 cores, and OpenMP multi-threading strong scalability on a single KNL. Section 4.6 demonstrates the scalability and potential benefit of using more than one KNL with optimal run options in place. Section 4.7 summarizes our conclusions on the performance of the CICR application code on the KNL and discusses opportunities for future work.

4.2 Numerical Method

In the CICR model, we solve a system of time-dependent parabolic partial differential equations (PDEs) of the form (1.1.1). These PDEs are coupled by several non-linear reaction and source terms in $q^{(i)}(u^{(1)}, \ldots, u^{(n_s)}, \mathbf{x}, t)$. Taking a method of lines (MOL) approach, we use the finite volume method (FVM) as the spatial discretization, with $N = (N_x + 1)(N_y + 1)(N_z + 1)$ control volumes. Applying the FVM to the n_s species PDEs results in a large system of ordinary differential equations (ODEs) with $n_{eq} = n_s N$ degrees of freedom (DOF). The resulting ODE system is stiff thus requires the use of implicit ODE methods. We make use of sophisticated time-stepping methods, in particular, the family of numerical differentiation formulas

(NDFk) that is both variable order and adaptive in time step size. Implicit ODE methods require the solution of the system of the n_{eq} non-linear equations. We use Newton's method as the non-linear solver, and at each Newton step we use the bi-conjugate gradient stabilized (BiCGSTAB) method as the linear solver. Complete details of the numerical method can be found in [22, 46].

In Table 4.2.1 we recall the sizing study for the application problem as in Table 2.4.1 which gives us memory predictions for different mesh sizes for the CICR problem. Table 4.2.1 and Table 2.4.1 both show shows the number of degrees of freedom for different mesh sizes for this problem and give the memory predictions for the 6 species CICR simulations. We are using a matrix-free method that minimizes memory usage by not storing any system matrix; the code with the NDFk method of orders $1 \leq k \leq 5$ requires then, including all auxiliary method vectors, the storage of only 17 arrays of significant size n_{eq}. For the simulations in Table 4.2.1, we observe that four of the mesh sizes presented, the finest being $128 \times 128 \times 512$, fit into the 16 GB of MCDRAM on the KNL. From Table 4.2.1 we note that the $256 \times 256 \times 1024$ mesh requires more than 50 GB and does not fit in the 16 GB of MCDRAM on the KNL, but can easily be accommodated on a single KNL node.

The $32 \times 32 \times 128$ mesh resolution was used in the studies in Chapter 2 and previous studies with the CICR model. Still, there could be cases in which the finer mesh like $128 \times 128 \times 512$ is advantageous to observe smaller scale physiological behavior. For this study of the KNL hardware, we prefer to solve the largest problems possible that fit in the MCDRAM of the KNL. Given the simulations fit in the MCDRAM of the KNL, we focus our attention to the Flat Quadrant KNL configuration where we can elect to run using only the MCDRAM.

In Table 4.2.1 and Table 2.4.1 the number of time steps are different. This reflects the 10 ms simulation time in the simulations of this chapter in contrast to Chapter 2

Table 4.2.1 Sizing study for CICR on a KNL with $n_s = 6$ species using double precision arithmetic, listing the mesh resolution $N_x \times N_y \times N_z$, the number of control volumes $N = (N_x + 1)(N_y + 1)(N_z + 1)$, the number of degrees of freedom (DOF) $n_{eq} = n_s N$, the number of time steps taken by the ODE solver, and the predicted memory usage in GB for a one-process run.

Resolution	N	DOF n_{eq}	number of time steps	memory usage predicted (GB)
$16 \times 16 \times 64$	18,785	112,710	273	0.01
$32 \times 32 \times 128$	140,481	842,886	497	0.11
$64 \times 64 \times 256$	1,085,825	6,514,950	841	0.83
$128 \times 128 \times 512$	8,536,833	51,220,998	1,469	6.49
$256 \times 256 \times 1024$	67,700,225	406,201,350	N/A	54.45

which used a 1000 ms simulation time. The 1000 ms run times require approximately 100 times more time steps than the 10 ms runs, so we expect the run times to be approximately 100 times shorter with a 10 ms simulation time.

4.3 MPI Only: Code Version 1

We start with our existing MPI only code that was use on Chapter 2 and refer to it as code version 1 in this work. To assess the existing code performance on a single KNL we present a strong scalability study of MPI processes in Table 4.3.2. As was done in Chapter 3, we present the memory observations for the code using different numbers of MPI processes. Recall for the Poisson problem in Chapter 3 we used the same Intel compiler and MPI implementation used here, since they are default on the Stampede-KNL cluster. The memory observations in Table 4.3.1 seek to verify that even with large number of MPI processes on a single KNL, runs fit in the high-performance memory resource, the 16 GB of MCDRAM. Though we expect from Table 4.2.1 that none of the four selected mesh sizes will require more than 16 GB of memory total, the significant memory overhead associated with MPI processes in

Table 4.3.1 Observed total memory usage for CICR in units of GB on 1 KNL in Stampede using 256 threads in Cache Quadrant configuration for code version 1, MPI only.

MPI proc	Predicted	$p = 1$	2	4	8	16	32	64	128	256
Threads/proc	(GB)	1	1	1	1	1	1	1	1	1
$16 \times 16 \times 64$	0.01	0.06	0.09	0.16	0.30	0.58	1.15	2.27	(**)	(**)
$32 \times 32 \times 128$	0.11	0.17	0.20	0.27	0.41	0.68	1.25	2.40	5.24	(**)
$64 \times 64 \times 256$	0.83	0.97	1.00	1.07	1.21	1.50	2.07	3.20	6.10	12.59
$128 \times 128 \times 512$	6.49	7.30	7.32	7.40	7.54	7.85	8.45	9.67	12.67	19.42

Chapter 3 make the observation worthwhile. The memory usage is observed in the code by checking the `VmRSS` field in the the special file `/proc/self/status`.

The second column of Table 4.3.1 repeats the memory predictions from Table 4.2.1. From the second to the third column of Table 4.3.1 we observe that the predicted memory usage is a reasonable underestimate of the total memory usage with the current MPI implementation, in the 1 MPI process case. As in Table 3.4.4 we observe the large overhead associated with the use of many MPI processes. The key observation is that in the $128 \times 128 \times 512$ mesh size case, in which the total memory observed is more than the 16 GB of MCDRAM. For this reason, we run the MPI strong scalability study in Table 4.3.2 on a single KNL in Cache Quadrant configuration rather than the Flat Quadrant configuration where we restrict our memory usage to the MCDRAM. This choice of Cache Quadrant configuration ensures that the code will execute, even if it requires more than 16 GB, on the high-performance on-chip memory.

Table 4.3.2 presents a strong scalability study by number of MPI processes p. Each row lists the results for one problem size. Each column corresponds to the number of parallel processes p used in the run. Strong scalability is one key motivation for parallel computing: the run times for a problem of a given, fixed size can be potentially dramatically reduced by spreading the work across a group of parallel processes. More precisely, the ideal behavior of code for a fixed problem size using p

parallel processes is that it be p times as fast. If $T_p(N)$ denotes the wall clock time for a problem of a fixed size parametrized by N using p processes, then the quantity $S_p = T_1(N)/T_p(N)$ measures the speedup of the code from 1 to p processes, whose optimal value is $S_p = p$. The efficiency $E_p = S_p/p$ characterizes in relative terms how close a run with p parallel processes is to this optimal value, for which $E_p = 1$. Table 4.3.2 (b) shows the observed speedup S_p. Table 4.3.2 (c) shows the observed efficiency E_p.

In Table 4.3.2 we observe that the code scales well, with near optimal halving of run time with the doubling of MPI processes from 2 to 4 in each mesh size. For example, with the $32 \times 32 \times 128$ mesh, using 2 MPI process the run time is 10:05, but with 4 MPI processes, the run time is nearly halved, 05:11. But, this good scaling slows down quickly. In the $16 \times 16 \times 64$ mesh case from 8 to 16 MPI process is only a 00:15 to 00:11 reduction in run time. Then there is no observed benefit from doubling the MPI processes again to 32, as the run time remains at 00:11 and there is a loss in performance in using 64 MPI processes. In the $32 \times 32 \times 128$ and $64 \times 64 \times 256$ mesh cases the 8 to 16 MPI processes jump still shows good scalability, but again more MPI processes on the single KNL loses its benefit after a point. For the $128 \times 128 \times 512$ case, since the run times with only a few processes are excessive, we start the study using 16 processes with confidence that the code scales well with less process already from the coarser meshes. For the speedup and efficiency calculations for the $128 \times 128 \times 512$ mesh we use the 16 process run as the base case. Overall, we focus on the finer meshes ans conclude that the code scales well for up to 64 processes.

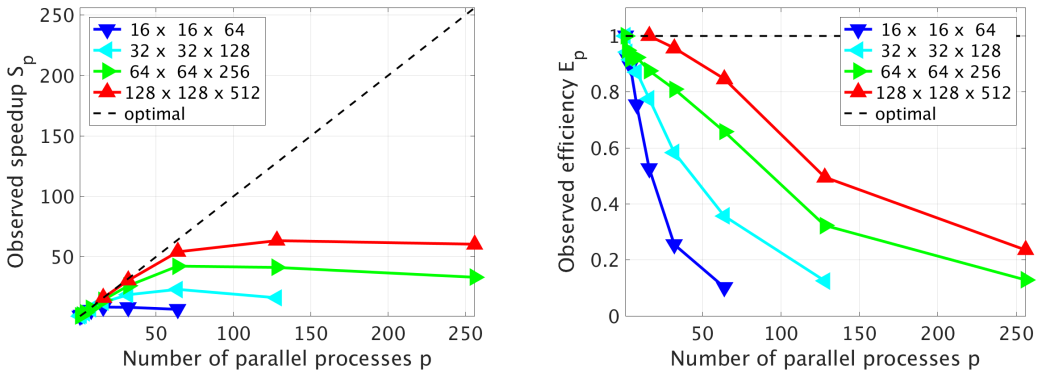
Figure 4.3.1 (a) and (b) presents the customary graphical representations of speedup and efficiency, respectively, for code version 1 (MPI only) on a single KNL. Figure 4.3.1 (a) shows the speedup pattern in Table 4.3.2 (b) a bit more intuitively. The efficiency plotted in Figure 4.3.1 (b) is directly derived from the speedup, but the

Table 4.3.2 CICR strong scalability study of MPI processes. Observed wall clock times in units of HH:MM:SS on 1 KNL in Cache Quadrant Configuration, using MPI parallelism only. For up to 64 processes one processes per core is used, then 2 processes per core (64 cores) for 128 processes, and 4 processes per core (64 cores) for 256 processes. ET indicates excessive time.

Code version 1, MPI only – 1 KNL – Cache Quadrant Configuration									
(a) Wall clock time									
MPI proc p	1	2	4	8	16	32	64	128	256
$16 \times 16 \times 64$	00:01:30	00:00:49	00:00:25	00:00:15	00:00:11	00:00:11	00:00:14	(**)	(**)
$32 \times 32 \times 128$	00:18:59	00:10:05	00:05:11	00:02:44	00:01:32	00:01:01	00:00:50	00:01:11	(**)
$64 \times 64 \times 256$	05:17:38	02:47:19	01:26:31	00:43:02	00:22:40	00:12:16	00:07:32	00:07:41	00:09:35
$128 \times 128 \times 512$	ET	ET	ET	ET	05:30:27	02:53:04	01:37:49	01:23:34	01:27:52
(b) Observed speedup S_p									
MPI proc p	1	2	4	8	16	32	64	128	256
$16 \times 16 \times 64$	1.00	1.84	3.57	6.04	8.43	8.17	6.53	(**)	(**)
$32 \times 32 \times 128$	1.00	1.88	3.66	6.97	12.41	18.69	22.88	16.02	(**)
$64 \times 64 \times 256$	1.00	1.90	3.67	7.38	14.01	25.90	42.13	41.36	33.12
$128 \times 128 \times 512$	—	—	—	—	16.00	30.60	54.14	63.36	60.26
(c) Observed efficiency E_p									
MPI proc p	1	2	4	8	16	32	64	128	256
$16 \times 16 \times 64$	1.00	0.92	0.89	0.75	0.53	0.26	0.10	(**)	(**)
$32 \times 32 \times 128$	1.00	0.94	0.92	0.87	0.78	0.58	0.36	0.13	(**)
$64 \times 64 \times 256$	1.00	0.95	0.92	0.92	0.88	0.81	0.66	0.32	0.13
$128 \times 128 \times 512$	—	—	—	—	1.00	0.96	0.85	0.50	0.24

plot is still useful because it details interesting features for small values of p that are hard to discern in the speedup plot. Here, we notice the consistency of most results for small numbers of MPI processes. We observe clearly that the finer mesh problem sizes perform better in this study.

The fundamental reason for the speedup and efficiency to trail off is that too little work is performed on each process. Due to the one-dimensional split in the z-direction into as many sub domains as parallel processes p, eventually only one or two x–y-planes of data are located on each process. This is not enough calculation work to justify the cost of communicating between the processes. In the 8 through 32 MPI process range the code performs slightly better in terms of scalability than the 3 species code on CPU nodes shown in [21]. This study can help recommend how many processes to use for a certain mesh size and indicated that 8 to 32 MPI processes seem to be a good initial choice on the KNL.



(a) Observed speedup S_p (b) Observed efficiency E_p

Figure 4.3.1 Speedup (a) and Efficiency (b) plots for code version 1, MPI only, on one KNL using p MPI processes.

4.4 Hybrid MPI+OpenMP: Code Version 2

We now consider an initial implementation of MPI+OpenMP code, which we refer to in this work as code version 2. This version represents an interim version in our development of the hybrid code. Performance studies here will demonstrate the need for a better OpenMP implementation. For the addition of OpenMP parallelism to the existing MPI code we began with adding `#pragma omp parallel for` around our sizable `for` loops in our utility routines. We intentionally added OpenMP loops for initialization of the large vectors in the code so that data would be set up in the way that it would be accessed in the multi-threaded loops. Additionally we added multi-threading around the main for loops in the most time consuming portion of the code, the matrix free matrix vector product.

We made use of profiling tools, specifically Intel's VTune Amplifier and TAU (Tuning and Analysis Utilities), to assess the performance of the implementations. TAU is a joint project between the University of Oregon Performance Research Lab, The LANL Advanced Computing 16 Laboratory, and The Research Center Jülich at

ZAM, Germany. More information about TAU is available at `www.cs.uoregon.edu/research/tau`.

Table 4.4.1 presents a performance study using 256 threads of a KNL with different combinations of MPI processes and OpenMP threads for the MPI+OpenMP code version 2. In sub table (a) `KMP_AFFINITY=compact` is used, while in sub table (b) `KMP_AFFINITY=scatter` is used.

In Table 4.4.1 we observe that for the MPI+OpenMP code version 2 using more MPI processes compares to OpenMP threads results in significantly better performance. The impact of the OpenMP parallelism is so muted that the timing from 1 to 2 to 4 MPI processes improve by almost a factor of 2 each time. This lead us to believe that a much better OpenMP implementation was still possible. The best run times for each mesh size though, do not use the maximum possible MPI process with minimal multi-threading, but rather use a balance of MPI and OpenMP parallelism. In the $32 \times 32 \times 128$ case the best run time uses 32 MPI processes with 8 threads per process, while in the $64 \times 64 \times 256$ case the best run time uses 64 MPI processes with 4 threads per process. This shows us already the importance of hybrid code, that is the use of both MPI and OpenMP parallelism on architectures like the KNL.

We also compare Table 4.4.1 (a) against Table 4.4.1 (b) to assess the performance of `KMP_AFFINITY=compact` versus `KMP_AFFINITY=scatter`. We observe that there is very little difference in performance, with a every so slight edge in favor of `KMP_AFFINITY=scatter`. This matches the result from Chapter 3.

We also test another important choice for running hybrid MPI+OpenMP code on the KNL in the distribution of threads to cores. Table 4.4.2 shows the wall clock times for varying combinations of MPI processes and OpenMP threads for the 6-species CICR code using all 68 KNL cores with 1, 2, 3 and 4 threads per core in Flat Quadrant Configuration and `KMP_AFFINITY=scatter`. In each case we maintain

Table 4.4.1 Observed wall clock times in units of HH:MM:SS for MPI+OpenMP code version 2 on 1 KNL on Stampede using 256 threads in Flat Quadrant Configuration, using MCDRAM only, with two settings of `KMP_AFFINITY`. ET indicates excessive time.

(a) KNL – Flat Quadrant Configuration – MCDRAM – Compact									
MPI proc	1	2	4	8	16	32	64	128	256
Threads/proc	256	128	64	32	16	8	4	2	1
$16 \times 16 \times 64$	00:00:35	00:00:22	00:00:15	00:00:12	00:00:12	00:00:15	(**)	(**)	(**)
$32 \times 32 \times 128$	00:05:03	00:03:21	00:01:58	00:01:23	00:01:03	00:01:01	00:01:12	(**)	(**)
$64 \times 64 \times 256$	00:44:28	00:27:48	00:18:39	00:14:17	00:11:30	00:09:58	00:09:48	00:10:39	00:12:24
(b) KNL – Flat Quadrant Configuration – MCDRAM – Scatter									
MPI proc	1	2	4	8	16	32	64	128	256
Threads/proc	256	128	64	32	16	8	4	2	1
$16 \times 16 \times 64$	00:00:33	00:00:21	00:00:15	00:00:12	00:00:13	00:00:16	(**)	(**)	(**)
$32 \times 32 \times 128$	00:05:00	00:03:17	00:01:59	00:01:21	00:01:04	00:01:01	00:01:11	(**)	(**)
$64 \times 64 \times 256$	00:45:05	00:28:00	00:18:35	00:13:57	00:11:20	00:09:46	00:09:36	00:10:38	00:12:28

the number of MPI processes and simply increase the number of threads per process to use more threads per core. We use only multiples of 68 for the number of MPI processes. We use all 68 cores and based on our results from Chapter 3. The choice of `KMP_AFFINITY=scatter` is also suggested from the results in Chapter 3 as well as the results in Table 4.4.1.

From Table 4.4.2 we observe that using only 1 or 2 threads per core performs better than using 3 or 4 threads per core. This result tells us that using the full 272 threads (68 cores with 4 threads per core) of the KNL is not advantageous to leaving significant amounts of the hardware free.

With this OpenMP implementation the best run times in Table 4.4.1 do not beat the best run times from version 1 of the code in Table 4.3.2 (a). However, Table 4.4.2 sheds light as to why not. Without making optimal choices for number of threads per core and the balance of MPI processes to OpenMP threads optimal performance is not possible. We see from Table 4.4.2 that using 1 or 2 threads per core gives the best performance. The optimal run times in Table 4.4.2 beat the best run times for version 1 of the code.

Table 4.4.2 Observed wall clock times in units of HH:MM:SS for MPI+OpenMP code version 2 on 1 KNL on Stampede using 68 cores with 1, 2, 3 and 4 threads per core in Flat Quadrant Configuration, using MCDRAM only.

(a) KNL – Flat Quadrant – 68 cores – 1 thread per core						
MPI proc	1	2	4	17	34	68
Threads/proc	68	34	17	4	2	1
$16 \times 16 \times 64$	00:00:17	00:00:12	00:00:08	00:00:07	(**)	(**)
$32 \times 32 \times 128$	00:03:18	00:01:52	00:01:09	00:00:42	00:00:41	00:00:58
$64 \times 64 \times 256$	00:40:45	00:24:08	00:15:10	00:08:45	00:07:33	00:07:56
(b) KNL – Flat Quadrant – 68 cores – 2 threads per core						
MPI proc	1	2	4	17	34	68
Threads/proc	136	68	34	8	4	2
$16 \times 16 \times 64$	00:00:23	00:00:14	00:00:10	00:00:10	(**)	(**)
$32 \times 32 \times 128$	00:04:04	00:02:16	00:01:20	00:00:43	00:00:49	(**)
$64 \times 64 \times 256$	00:41:32	00:24:40	00:15:33	00:08:23	00:07:26	00:06:48
(c) KNL – Flat Quadrant – 68 cores – 3 threads per core						
MPI proc	1	2	4	17	34	68
Threads/proc	204	102	51	12	6	3
$16 \times 16 \times 64$	00:00:30	00:00:19	00:00:13	00:00:11	(**)	(**)
$32 \times 32 \times 128$	00:04:37	00:02:55	00:01:44	00:00:54	00:00:53	(**)
$64 \times 64 \times 256$	00:44:53	00:27:52	00:18:29	00:11:02	00:08:27	00:08:28
(d) KNL – Flat Quadrant – 68 cores – 4 threads per core						
MPI proc	1	2	4	17	34	68
Threads/proc	272	136	68	16	8	4
$16 \times 16 \times 64$	00:00:36	00:00:24	00:00:18	00:00:16	(**)	(**)
$32 \times 32 \times 128$	00:05:05	00:03:21	00:02:03	00:01:05	00:01:02	(**)
$64 \times 64 \times 256$	00:43:50	00:26:00	00:16:44	00:09:32	00:12:51	00:07:51

4.5 Hybrid MPI+OpenMP: Code Version 3

We now consider our final MPI+OpenMP hybrid code implementation, which we refer to as code version 3 in this work. For this final version we add to code version 2 additional multi-threading as the most time consuming function of the code yet to use multi-threading directly. This is the function that includes the non-linear reactions in the CICR that couple the species of the model. It represents the final substantial portion of the CICR code that may benefit from multi-threading. The results of the performance tests in this section show that the OpenMP performance improves significantly over code version 2. Again we made use of Intel's VTune Amplifier and TAU to assess the performance of the implementations.

This section is organized as follows. Table 4.5.1 presents a performance study using 256 threads on a single KNL with different combinations of MPI processes and OpenMP threads for the MPI+OpenMP hybrid code version 3. In Table 4.5.1 (a) `KMP_AFFINITY=compact` is used, while in Table 4.5.1 (b) `KMP_AFFINITY=scatter` is used. Next, Table 4.5.2 presents a performance study for different numbers of threads per core study using all 68 cores on a single KNL. This is paired with a threads per core study using 64 cores on a single KNL, that is leaving 4 cores free, in Table 4.5.3. Finally, Table 4.5.4 presents an OpenMP multi-threading strong scalability study on a single KNL.

In Table 4.5.1, we observe that neither MPI only nor OpenMP only parallelism performs as well as using MPI+OpenMP together. This matches our observation from Table 4.4.1 and again our conclusion from Chapter 3. We observe this from the fact that the run times in the second column and right most column for each mesh are significantly longer than the best run time across the row for each mesh. For example, in Table 4.5.1 (a) with the $64 \times 64 \times 256$ mesh using 256 MPI processes with 1 thread per process runs for 13:28 and using 1 MPI process with 256 threads runs for 11:23,

Table 4.5.1 Observed wall clock times in units of HH:MM:SS for MPI+OpenMP code version 3 on 1 KNL on Stampede using 256 threads in Flat Quadrant Configuration, using MCDRAM only, with two settings of `KMP_AFFINITY`. ET indicates excessive time.

(a) KNL – Flat Quadrant Configuration – MCDRAM – Compact									
MPI proc	1	2	4	8	16	32	64	128	256
Threads/proc	256	128	64	32	16	8	4	2	1
$16 \times 16 \times 64$	00:00:10	00:00:09	00:00:09	00:00:09	00:00:10	00:00:14	(**)	(**)	(**)
$32 \times 32 \times 128$	00:00:56	00:00:45	00:00:41	00:00:40	00:00:42	00:00:50	00:01:38	(**)	(**)
$64 \times 64 \times 256$	00:11:23	00:09:08	00:08:01	00:07:36	00:07:35	00:08:14	00:09:19	00:11:00	00:13:28
(b) KNL – Flat Quadrant Configuration – MCDRAM – Scatter									
MPI proc	1	2	4	8	16	32	64	128	256
Threads/proc	256	128	64	32	16	8	4	2	1
$16 \times 16 \times 64$	00:00:10	00:00:09	00:00:09	00:00:09	00:00:11	00:00:14	(**)	(**)	(**)
$32 \times 32 \times 128$	00:00:55	00:00:46	00:00:41	00:00:41	00:00:44	00:00:51	00:01:07	(**)	(**)
$64 \times 64 \times 256$	00:11:29	00:09:10	00:08:06	00:07:38	00:07:32	00:07:55	00:08:46	00:10:21	00:12:30

while the best run time is 07:35 from 16 MPI process with 16 OpenMP threads per process. The use of 8 or 16 MPI processes and 32 or 16 threads per process performed well for all problem sizes.

As was true with version 2 of the MPI+OpenMP code in Table 4.4.1, the performance with `KMP_AFFINITY=compact` shown in Table 4.5.1 (a) is comparable to the performance with `KMP_AFFINITY=scatter` shown in Table 4.5.1 (b). The only observable difference is for 32 or more MPI processes in the finer mesh, where `scatter` outperforms `compact`. We continue to move forward with `KMP_AFFINITY=scatter` as our default choice.

By comparing Table 4.4.1 to Table 4.5.1, we observe the difference in performance between version 2 and version 3 of the MPI+OpenMP code. We observe that when using only a few MPI processes with large numbers of OpenMP threads version 3 significantly outperforms version 2. We also observe that the best run times for code version 3 are better than the best run times for code version 2. We also observe that version 3 is the only version that beats the best MPI only run times and uses a mix of MPI and OpenMP to do so. In the $32 \times 32 \times 128$ case the best MPI run time is 50 seconds, the best version 2 run time is just over 1 minute, and the best version 3

run time is 41 seconds. In the $64 \times 64 \times 256$ case the best MPI run time is 07:32, the best version 2 run time is 09:36, and the best version 3 run time is 07:32.

Table 4.5.2 shows the wall clock times for varying combinations of MPI processes and OpenMP threads for the 6-species CICR code using 68 KNL cores with 1, 2, 3 and 4 threads per core in Flat Quadrant Configuration and `KMP_AFFINITY=scatter`. In each case we maintain the number of MPI processes and simply increase the number of threads per process to use more threads per core. We use only multiples of 68 for the number of MPI processes. We add the finest possible mesh that fits in the 16 GB KNL on-chip memory $128 \times 128 \times 512$, for this optimal code implementation. For the coarsest two mesh sizes, using 1 or 2 threads per core is better than using 3 or 4 threads per core. But in the finer meshes, $64 \times 64 \times 256$ and $128 \times 128 \times 512$, the best run times take advantage of 4 threads per core. The 17 MPI process, and 4 OpenMP thread runs perform better for the finer mesh, than for the two coarser meshes. Overall, the coarser meshes benefit from more OpenMP threads while the finer meshes benefit from more MPI processes for this code.

Table 4.5.3 presents the threads per core study using 64 cores on a single KNL, that is leaving 4 cores free. The general observation that 1 or 2 threads per core is again clear, perhaps with a slight favoring of 2 threads per core for its better performance on the finer mesh. If we compare Table 4.5.2 with Table 4.5.3, we see that using all 68 cores appears to perform better than using only 64 cores on a single KNL. To see this take for example, the $64 \times 64 \times 256$ where the best run with 64 cores is 07:13, but with 68 cores it is 06:03. It is not just the best run times that are better with 68 cores, nearly all of the comparable MPI and OpenMP combinations for each number of threads per core are better with 68 cores rather than 64 on a single KNL. This justifies our use of 68 cores for the threads per core study with version 2 of the code in Table 4.4.2.

Table 4.5.2 Observed wall clock times in units of HH:MM:SS for MPI+OpenMP code version 3 on 1 KNL on Stampede using 68 cores with 1, 2, 3 and 4 threads per core in Flat Quadrant Configuration, using MCDRAM only.

(a) KNL – Flat Quadrant – MCDRAM only – 1 thread per core						
MPI proc	1	2	4	17	34	68
Threads/proc	68	34	17	4	2	1
$16 \times 16 \times 64$	00:00:05	00:00:05	00:00:05	00:00:07	(**)	(**)
$32 \times 32 \times 128$	00:00:40	00:00:33	00:00:31	00:00:35	00:00:39	(**)
$64 \times 64 \times 256$	00:09:25	00:07:59	00:07:19	00:07:14	00:07:06	00:07:39
$128 \times 128 \times 512$	02:36:12	02:12:32	01:45:21	01:43:05	01:41:38	01:40:45
(b) KNL – Flat Quadrant – MCDRAM only – 2 threads per core						
MPI proc	1	2	4	17	34	68
Threads/proc	136	68	34	8	4	2
$16 \times 16 \times 64$	00:00:08	00:00:07	00:00:07	00:00:09	(**)	(**)
$32 \times 32 \times 128$	00:00:44	00:00:31	00:00:28	00:00:30	00:00:41	(**)
$64 \times 64 \times 256$	00:09:09	00:07:13	00:06:17	00:06:03	00:06:23	00:06:26
$128 \times 128 \times 512$	02:11:12	01:57:17	01:41:50	01:24:39	01:26:17	01:23:29
(c) KNL – Flat Quadrant – MCDRAM only – 3 threads per core						
MPI proc	1	2	4	17	34	68
Threads/proc	204	102	51	12	6	3
$16 \times 16 \times 64$	00:00:09	00:00:08	00:00:08	00:00:10	(**)	(**)
$32 \times 32 \times 128$	00:00:51	00:00:42	00:00:37	00:00:40	00:00:48	(**)
$64 \times 64 \times 256$	00:11:46	00:09:36	00:08:25	00:07:57	00:07:10	00:08:04
$128 \times 128 \times 512$	02:29:06	01:57:22	01:39:58	01:32:09	01:33:29	01:28:09
(d) KNL – Flat Quadrant – MCDRAM only – 4 threads per core						
MPI proc	1	2	4	17	34	68
Threads/proc	272	136	68	16	8	4
$16 \times 16 \times 64$	00:00:14	00:00:13	00:00:09	00:00:14	(**)	(**)
$32 \times 32 \times 128$	00:01:00	00:00:56	00:00:46	00:00:49	00:00:56	(**)
$64 \times 64 \times 256$	00:09:49	00:07:38	00:06:20	00:05:53	00:06:26	00:07:20
$128 \times 128 \times 512$	02:19:05	01:46:17	01:28:42	01:19:52	01:21:08	01:24:47

Table 4.5.3 Observed wall clock times in units of HH:MM:SS for MPI+OpenMP code version 3 on 1 KNL on Stampede using 64 cores with 1, 2, 3 and 4 threads per core in Flat Quadrant Configuration, using MCDRAM only.

(a) KNL – Flat Quadrant – MCDRAM only – 1 thread per core							
MPI proc	1	2	4	8	16	32	64
Threads/proc	64	32	16	8	4	2	1
$16 \times 16 \times 64$	00:00:05	00:00:05	00:00:05	00:00:06	00:00:07	00:00:10	00:00:18
$32 \times 32 \times 128$	00:00:46	00:00:38	00:00:35	00:00:35	00:00:40	00:00:45	00:00:58
$64 \times 64 \times 256$	00:10:06	00:08:38	00:08:01	00:07:39	00:07:47	00:08:09	00:08:41
(b) KNL – Flat Quadrant – MCDRAM only – 2 threads per core							
MPI proc	1	2	4	8	16	32	64
Threads/proc	128	64	32	16	8	4	2
$16 \times 16 \times 64$	00:00:08	00:00:07	00:00:06	00:00:07	00:00:09	00:00:11	(**)
$32 \times 32 \times 128$	00:00:45	00:00:37	00:00:34	00:00:37	00:00:40	00:00:42	00:00:57
$64 \times 64 \times 256$	00:09:38	00:07:54	00:07:07	00:07:56	00:07:58	00:07:13	00:07:55
(c) KNL – Flat Quadrant – MCDRAM only – 3 threads per core							
MPI proc	1	2	4	8	16	32	64
Threads/proc	192	96	48	24	12	6	3
$16 \times 16 \times 64$	00:00:09	00:00:08	00:00:08	00:00:09	00:00:10	00:00:12	(**)
$32 \times 32 \times 128$	00:00:51	00:00:41	00:00:37	00:00:39	00:00:42	00:00:44	00:00:58
$64 \times 64 \times 256$	00:11:42	00:09:28	00:08:20	00:08:56	00:08:55	00:08:09	00:07:55
(d) KNL – Flat Quadrant – MCDRAM only – 4 threads per core							
MPI proc	1	2	4	8	16	32	64
Threads/proc	256	128	64	32	16	8	4
$16 \times 16 \times 64$	00:00:10	00:00:09	00:00:09	00:00:09	00:00:11	00:00:14	(**)
$32 \times 32 \times 128$	00:00:56	00:00:46	00:00:41	00:00:40	00:00:43	00:00:51	00:01:05
$64 \times 64 \times 256$	00:11:34	00:09:13	00:08:03	00:07:36	00:07:30	00:07:55	00:08:44

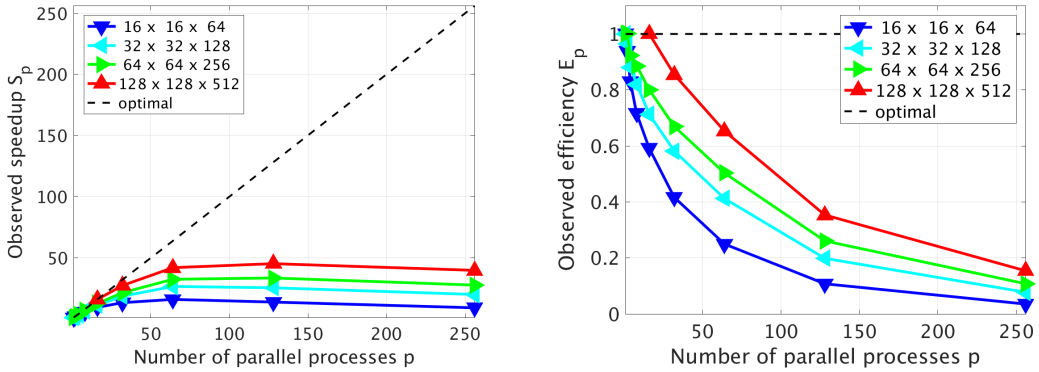
Table 4.5.4 CICR strong scalability study of OpenMP threads. Observed wall clock times in units of HH:MM:SS on 1 KNL node in Flat Quadrant Configuration. For up to 64 threads one thread per core is used, then 2 threads per core (64 cores) for 128 threads, and 4 threads per core (64 cores) for 256 threads with and `KMP_AFFINITY=scatter` in all cases. ET indicates excessive time.

Code version 3, MPI+OpenMP – 1 KNL – Flat Quadrant Configuration – MCDRAM only									
(a) Wall clock time									
OpenMP threads p	1	2	4	8	16	32	64	128	256
$16 \times 16 \times 64$	00:01:29	00:00:47	00:00:27	00:00:15	00:00:09	00:00:07	00:00:06	(**)	(**)
$32 \times 32 \times 128$	00:18:54	00:09:47	00:05:22	00:02:53	00:01:39	00:01:01	00:00:43	00:00:44	(**)
$64 \times 64 \times 256$	05:22:24	02:40:56	01:27:17	00:45:33	00:25:11	00:15:04	00:10:00	00:09:40	00:11:42
$128 \times 128 \times 512$	ET	ET	ET	ET	06:14:07	03:39:06	02:23:12	02:12:31	02:31:02
(b) Observed speedup S_p									
OpenMP threads p	1	2	4	8	16	32	64	128	256
$16 \times 16 \times 64$	1.00	1.88	3.32	5.73	9.47	13.32	15.99	13.80	9.10
$32 \times 32 \times 128$	1.00	1.93	3.52	6.55	11.40	18.59	26.44	25.51	19.83
$64 \times 64 \times 256$	1.00	2.00	3.69	7.08	12.80	21.41	32.26	33.35	27.57
$128 \times 128 \times 512$	—	—	—	—	16.00	27.32	41.80	45.17	39.63
(c) Observed efficiency E_p									
OpenMP threads p	1	2	4	8	16	32	64	128	256
$16 \times 16 \times 64$	1.00	0.94	0.83	0.72	0.59	0.42	0.25	0.11	0.04
$32 \times 32 \times 128$	1.00	0.97	0.88	0.82	0.71	0.58	0.41	0.20	0.08
$64 \times 64 \times 256$	1.00	1.00	0.92	0.88	0.80	0.67	0.50	0.26	0.11
$128 \times 128 \times 512$	—	—	—	—	1.00	0.85	0.65	0.35	0.15

Table 4.5.4 presents a OpenMP threads strong scalability study using our code version 3 OpenMP implementation with only 1 MPI process on a single KNL. For small numbers of parallel processes p (in this case OpenMP threads) the run times are approximately halved as p is doubled corresponding to near optimal scalability. We readily observe that the scalability with OpenMP parallelism mirrors, almost exactly the good MPI only scalability. In fact, for up the 8 processes or threads, the numbers in Table 4.5.4 and Table 4.3.2 are nearly identical.

Figure 4.5.1 (a) and (b) presents the customary graphical representations of speedup and efficiency, respectively, for code version 3 (MPI+OpenMP) on a single KNL. Figure 4.3.1 (a) shows the speedup pattern in Table 4.5.4 (b). The efficiency plotted in Figure 4.5.1 (a) is directly derived from the speedup, and shows the behavior of small values of p that are hard to discern in the speedup plot.

The scalability of version 1 of the code using MPI in Table 4.5.4 and Figure 4.5.1 is nearly identical to the scalability of version 3 of the code using OpenMP Table 4.3.2



(a) Observed speedup S_p (b) Observed efficiency E_p

Figure 4.5.1 Speedup (a) and Efficiency (b) plots for code version 3, MPI+OpenMP, on one KNL using p OpenMP threads.

and Figure 4.3.1 for small numbers of parallel process. For larger numbers of parallel process the MPI only code scales marginally better than the hybrid code using only OpenMP. We conclude that the OpenMP parallelism in the code version 3 implementation scales nearly as well as our original code version 1 with MPI only.

4.6 Multiple KNLs

We are also interested in the scalability of the code on multiple KNLs. On the Stampede-KNL cluster, as is typical currently, KNL nodes feature one KNL per node. The interconnect between nodes is a 100 Gb/s Intel Omni-Path network and uses a fat tree topology of eight core-switches and 320 leaf switches [37]. In our tests on a single KNL, we observed that using 68 cores performed better than using only 64 cores. However, We leave some cores free as is recommended in [43] for the management of OMB-Path Architecture (OPA) traffic and improves scalability. It is also recommended in [43] to use at least 2 MPI processes per KNL, which we do here.

Problems larger than 16 GB can be accommodated on a KNL node by making use of the 96 GB of DDR4 node memory. But better performance may be obtained by using more than one KNL to make use of more high-performance on-chip memory than is available on each KNL. This study is especially important for fine meshes, for which the run times on a single KNL are excessive. In this investigation we first test the MPI only code version 1 for scalability to multiple nodes then assess the MPI+OpenMP hybrid code version 3.

Table 4.6.1 shows the scalability of the original MPI only code on multiple KNLs. We use different numbers of MPI processes per node to demonstrate that this choice has an impact on run time and scalability. In Table 4.6.1 (a) we use only 8 MPI processes per node so that even the coarsest mesh can be run on 8 KNL nodes. We can thus assess the scalability across up to 8 KNLs for all four of our mesh sizes. Recall that (**) indicates that the run is not possible, given that the number of MPI process is greater than the number of slices in the z-mesh direction that constrains our MPI parallelism in the code. If we focus our attention on the finer meshes, Table 4.3.2 shows the MPI code scales well up to 64 MPI processes on a single KNL. This leads us to use 64 MPI processes per KNL in Table 4.6.1 (b).

Table 4.6.1 (a) uses 8 MPI processes per node to assess clearly the scalability of the MPI only code. We observe very good scalability through 8 KNL for all but our coarsest mesh, $16 \times 16 \times 64$ and observe a run time benefit to using more KNLs for all meshes. However, the absolute run times in Table 4.6.1 (a) with 8 MPI processes per node are hardly comparable to Table 4.6.1 (b) with 64 MPI processes per node. When using 8 MPI processes per node in Table 4.6.1 (a) it requires 8 KNL nodes to perform better than a single KNL with 64 MPI processes in Table 4.6.1 (b). This emphasizes the importance of choosing the optimal number of MPI processes on the KNL. Concretely, in the $64 \times 64 \times 256$ case, we observe a run time of 08:44 with

Table 4.6.1 CICR strong scalability study of MPI processes. Observed wall clock times in units of HH:MM:SS on multiple KNL node in Flat Quadrant Configuration. Different number of MPI processes per node are used in each subtable.

Code version 1, MPI only – Flat Quadrant, MCDRAM only				
(a) Wall clock time – 8 MPI processes per node				
Number of KNLs	1	2	4	8
$16 \times 16 \times 64$	00:00:16	00:00:11	00:00:08	00:00:07
$32 \times 32 \times 128$	00:02:54	00:01:38	00:00:56	00:00:35
$64 \times 64 \times 256$	00:43:43	00:22:32	00:12:06	00:07:17
$128 \times 128 \times 512$	11:00:53	05:39:04	02:51:11	01:31:58
(b) Wall clock time – 64 MPI processes per node				
Number of KNLs	1	2	4	8
$16 \times 16 \times 64$	00:00:19	(**)	(**)	(**)
$32 \times 32 \times 128$	00:01:07	00:00:56	(**)	(**)
$64 \times 64 \times 256$	00:08:44	00:05:47	00:03:56	(**)
$128 \times 128 \times 512$	01:49:33	01:00:34	00:36:14	00:27:55

1 KNL compared to 07:17 with 8 KNL, and 01:49:33 compared to 01:31:58 in the $128 \times 128 \times 512$ case. The large number of parallel processes required to use 64 MPI processes per node limits the cases we can observe in terms of scalability. In Table 4.6.1 (b) we observe that using 64 processes per KNL shows good scalability from 1 to 4 KNL nodes in the $64 \times 64 \times 256$ and $128 \times 128 \times 512$ mesh cases. The relative scalability is not quite as good as the 8 MPI processes per node case in Table 4.6.1 (a), but as was noted, the performance is significantly better.

Table 4.6.2 presents a multiple KNL scalability study of code version 3 using optimal choices for MPI processes and OpenMP threads from our studies on a single KNL. In Table 4.6.2 (a), we use 8 MPI processes per node and 16 OpenMP threads per process. In Table 4.6.2 (b), we use 16 MPI processes per node and 8 OpenMP threads per process. In Table 4.6.2 (c), we use 32 MPI processes per node and 4 OpenMP threads per process. In each of the choices we use 2 threads per core on 64 cores for a total of 128 threads based on our observations in Table 4.5.3. Table 4.6.2

assesses the balance of MPI processes to OpenMP with the hybrid MPI+OpenMP code, since the number of MPI processes was shown to have a very significant impact in Table 4.6.1.

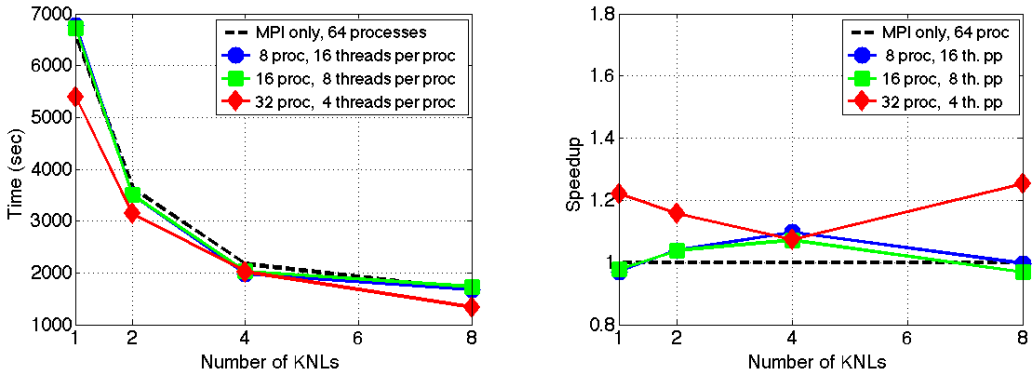
We observe that the relative scalability and performance are not very different between these two choice of MPI process and OpenMP threads in Table 4.6.2 (a) and Table 4.6.2 (b), especially in the $128 \times 128 \times 512$ case. Table 4.6.2 (a) and Table 4.6.2 (b) also match the less than optimal run time performance in Table 4.6.1 (b) with MPI only parallelism. But, using 32 MPI processes with 4 OpenMP threads per core absolute run times in Table 4.6.2 (c) are better than Table 4.6.1 (b) with MPI only. This again demonstrates the need for hybrid code to achieve best performance.

Figure 4.6.1 presents a graphical representation of the hybrid MPI+OpenMP performance for the $128 \times 128 \times 512$ mesh in Table 4.6.2 against the best MPI only performance from Table 4.6.1 (b). Figure 4.6.1 (a) shows the wall clock times in seconds for the MPI only code and each of the MPI processes to OpenMP threads choices in Tables 4.6.2 (a), (b), and (c). Figure 4.6.1 (b) presents the performance of the MPI+OpenMP code as speedup over the MPI only code runs. We confirm that the 8 and 16 MPI processes per node with 16 and 8 OpenMP threads, respectively, performance is comparable to the MPI only case but 32 processes per node with 4 threads per process consistently performs better than MPI only runs.

Table 4.6.3 tests the performance of multiple KNL without leaving any cores free. That is, we use all 68 cores on each KNL. We elected to test this based on our tests in Table 4.5.3 and Table 4.5.2, that show using 68 cores performs better than using only 64 cores. Table 4.5.2 also motivates our continued use of 2 threads per core on each of the 68 cores of the KNLs. The performance in terms of absolute run times using 68 cores is slightly better than all of the multi-KNL runs using 64 cores only in Table 4.6.2.

Table 4.6.2 CICR strong scalability study of multiple KNL nodes with hybrid MPI+OpenMP code version 3. Observed wall clock times in units of HH:MM:SS on multiple KNL nodes in Flat Quadrant Configuration. For each KNL 64 cores are used with 2 threads per core for a total of 128 threads and `KMP_AFFINITY=scatter` in all cases.

MPI+OpenMP – Flat Quadrant, MCDRAM only (2 threads per core, 64 cores)				
(a) 8 processes per node, 16 threads per process				
Number of KNLs	1	2	4	8
$16 \times 16 \times 64$	00:00:08	00:00:08	00:00:07	(**)
$32 \times 32 \times 128$	00:00:38	00:00:33	00:00:33	00:00:26
$64 \times 64 \times 256$	00:07:58	00:04:41	00:03:57	00:04:25
$128 \times 128 \times 512$	01:53:01	00:58:18	00:33:01	00:27:59
(b) 16 processes per node, 8 threads per process				
Number of KNLs	1	2	4	8
$16 \times 16 \times 64$	00:00:10	00:00:08	(**)	(**)
$32 \times 32 \times 128$	00:00:46	00:00:43	00:00:29	(**)
$64 \times 64 \times 256$	00:08:34	00:05:12	00:04:55	00:03:23
$128 \times 128 \times 512$	01:51:58	00:58:18	00:33:48	00:28:49
(c) 32 processes per node, 4 threads per process				
Number of KNLs	1	2	4	8
$16 \times 16 \times 64$	00:00:12	(**)	(**)	(**)
$32 \times 32 \times 128$	00:00:52	00:00:42	(**)	(**)
$64 \times 64 \times 256$	00:07:13	00:05:03	00:03:23	(**)
$128 \times 128 \times 512$	01:29:50	00:52:18	00:33:42	00:22:15



(a) Wall clock times (b) Speedup

Figure 4.6.1 Performance comparison of MPI only code versus hybrid MPI+OpenMP code version 3 using 64 KNL cores in the $128 \times 128 \times 512$ case. (a) Wall clock times in seconds for MPI only code and hybrid MPI+OpenMP code version 3 with different choices of MPI processes versus OpenMP threads. (b) Speedup of hybrid code over MPI only code

In the same manner as Table 4.6.2, Table 4.6.3 includes subtables with different combinations of MPI processes with OpenMP threads. In Table 4.6.3 (a), we use 8 MPI processes per node and 17 OpenMP threads per process. In Table 4.6.3 (b), we use 17 MPI processes per node and 8 OpenMP threads per process. In Table 4.6.3 (c), we use 34 MPI processes per node and 4 OpenMP threads per process. We observe that using 17 MPI processes with 8 threads per process performs better in terms of absolute run time than using 8 MPI processes with 17 threads per process and 34 MPI processes with 4 threads per process. Continuing to observe the $128 \times 128 \times 512$ case closely, the relative scalability from 1 to 2 and from 2 to 4 KNL is very good, but trails off significantly from 4 to 8 KNL.

We compare the absolute run times in Table 4.6.2 and Table 4.6.3 to determine the optimal choices for MPI processes and OpenMP threads when using multiple KNL. The best run times are in Table 4.6.3 (b), the 17 MPI processes per node with 8 threads per process case. Both Table 4.6.3 (c) and Table 4.6.2 (c) with 34 and 32 MPI processes per node with 4 threads, respectively, have comparable run times to Table 4.6.3 (b) that are not quite as good. Both Table 4.6.3 (a) and Table 4.6.2 (a) use 8 MPI processes per node with 16 and 17 threads, respectively, and do not perform nearly as well as the best case. The performance in Table 4.6.2 (b) with 16 MPI processes per node and 8 threads per process is comparable to Table 4.6.3 (a) and Table 4.6.2 (a) and is not optimal.

Overall, we observe that using only 64 cores did not show any significant benefit compared to using all 68 KNL cores for this code. In fact, the optimal combination of MPI processes and threads with multiple KNL nodes was 17 MPI processes per node with 8 OpenMP threads per process. In this case, all of the 68 cores of the KNL are used, with 1 MPI process on every other tile on the KNL. The 8 OpenMP threads per process use 2 threads on each of the 4 cores on the MPI process tile.

Table 4.6.3 CICR strong scalability study of multiple KNL nodes with hybrid MPI+OpenMP code version 3. Observed wall clock times in units of HH:MM:SS on multiple KNL nodes in Flat Quadrant Configuration. For each KNL 68 cores are used with 2 threads per core for a total of 136 threads and `KMP_AFFINITY=scatter` in all cases.

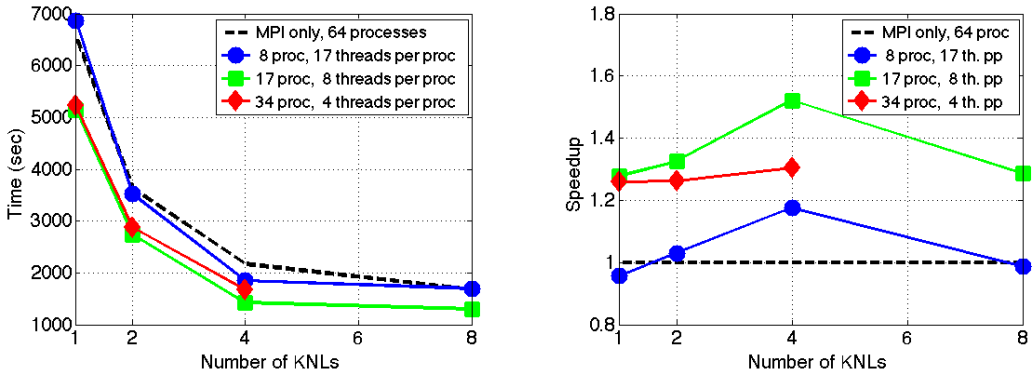
MPI+OpenMP – Flat Quadrant, MCDRAM only (2 threads per core, 68 cores)				
(a) 8 processes per node, 17 threads per process				
Number of KNLs	1	2	4	8
$16 \times 16 \times 64$	00:00:08	00:00:08	00:00:07	(**)
$32 \times 32 \times 128$	00:00:35	00:00:34	00:00:32	00:00:26
$64 \times 64 \times 256$	00:08:00	00:04:21	00:04:00	00:04:13
$128 \times 128 \times 512$	01:54:32	00:58:48	00:30:49	00:28:18
(b) 17 processes per node, 8 threads per process				
Number of KNLs	1	2	4	8
$16 \times 16 \times 64$	00:00:09	(**)	(**)	(**)
$32 \times 32 \times 128$	00:00:31	00:00:30	(**)	(**)
$64 \times 64 \times 256$	00:06:07	00:03:25	00:03:55	(**)
$128 \times 128 \times 512$	01:25:39	00:45:40	00:23:49	00:21:42
(c) 34 processes per node, 4 threads per process				
Number of KNLs	1	2	4	8
$16 \times 16 \times 64$	(**)	(**)	(**)	(**)
$32 \times 32 \times 128$	00:00:40	(**)	(**)	(**)
$64 \times 64 \times 256$	00:06:28	00:04:14	(**)	(**)
$128 \times 128 \times 512$	01:27:04	00:48:01	00:27:48	(**)

Figure 4.6.2 presents a graphical representation of the hybrid MPI+OpenMP performance in Table 4.6.3 against the best MPI only performance from Table 4.6.1 (b) in the same manner as Figure 4.6.1. Figure 4.6.2 (a) shows the wall clock times in seconds for the MPI only code and each of the MPI processes to OpenMP threads choices in Tables 4.6.3 (a), (b), and (c). Figure 4.6.2 (b) presents the performance of the MPI+OpenMP code as speedup over the MPI only code runs. We confirm that the 8 processes per node with 16 OpenMP threads per process performance is comparable to the MPI only case. Both 17 and 34 MPI processes with 8 and 4 threads per processes, respectively, perform better than the MPI only case.

We can also compare Figures 4.6.1 and 4.6.2 to observe the speedup increase using 68 cores rather than 64 cores. The best speedup over the best MPI only run in Figure 4.6.1 (b) is just over 1.2, but in Figure 4.6.2 (b) we observe that both the 17 MPI processes with 8 OpenMP threads per process case and the 34 processes with 4 threads per process case have better than 1.2 speedup for all cases. We confirm that the best performance results from 17 processes with 8 threads per process.

4.7 Conclusions

We have demonstrated the potential for performance of special purpose application codes on the KNL using advantageous choices of the configurations modes based on the applications size and code characteristics. We used the CICR problem from Chapter 2 to show performance for a specific application code. We discussed the addition of OpenMP parallelism to an existing MPI code and presented two different implementations with performance differences. This showed the need for hybrid MPI+OpenMP code implemented well for best performance. We also demonstrated the feasibility and benefit to using multiple KNL nodes.



(a) Wall clock times (b) Speedup

Figure 4.6.2 Performance comparison of MPI only code versus hybrid MPI+OpenMP code version 3 using 68 KNL cores in the $128 \times 128 \times 512$ case. (a) Wall clock times in seconds for MPI only code and hybrid MPI+OpenMP code version 3 with different choices of MPI processes versus OpenMP threads. (b) Speedup of hybrid code over MPI only code

The CICR code requires more demanding and significant communication and is more computationally intensive than the Poisson problem from Chapter 3. This application code put additional burden on the MPI communication and the balance with OpenMP threads. We observed this effect with two different MPI+OpenMP implementations. Careful memory observations showed the difference in memory overhead for MPI processes versus OpenMP threads as a benefit to OpenMP multi-threading parallelism. Memory observations confirmed that the careful memory management of this special purpose code enables us to study very fine meshes for physiological studies, like $128 \times 128 \times 512$, in the 16 GB of MCDRAM on the KNL. As a result, we primarily use the Flat Quadrant configuration and make use of only the MCDRAM. We tested the scalability of the MPI only code and the OpenMP scalability with hybrid MPI+OpenMP code on a single KNL and confirmed the benefit of using multiple KNL nodes.

With two different hybrid MPI+OpenMP code implementations, we considered the number and placement of OpenMP threads relative to the number of MPI processes used and assessed the optimal number of OpenMP threads per core on the KNL. The different implementations show different balances in performance with OpenMP threads versus MPI processes. Our final OpenMP implementation that implemented multi-threading in the function with the non-linear reaction coupling terms demonstrated significantly better performance with fewer MPI process and more multi-threading. We observed that optimal choices for OpenMP threads to MPI processes and threads per core depended on the mesh resolution of the problem. For coarser meshes, more OpenMP threads to MPI processes performed better, but for finer meshes, using more MPI processes relative to the number of threads performed better. For multi-KNL runs, the optimal combination of MPI processes and OpenMP threads per processes uses one MPI processes on every other tile and

2 threads per core across the tile, that is 17 MPI processes per node and 8 OpenMP threads per process.

This work enables further study of the KNL performance using the full 8 species of the model coupling both the mechanical contraction and electrical excitation systems to the calcium signaling system. Parameter studies with different physiological parameter sets will be important with the studies of these additional components of the model. With a firm understanding of the KNL performance with the CICR code, we can run simulations to final times of 1000 ms or more, rather than the 10 ms in the performance studies contained here. We can continue to optimize OpenMP implementation choices and explicitly test the vectorization options of the Intel compiler. Additionally, the $256 \times 256 \times 1024$ mesh case can be used to push a single KNL past the 16 GB MCDRAM, and compare against using multiple KNLs in which the memory demands distributed over 4 or more KNL would fit in the 16 GB MCDRAM of each KNL.

CHAPTER 5

CONCLUSIONS

In this chapter, we summarize our conclusions for the numerical solution of real-world application problems on the new second-generation Intel Xeon Phi Knights Landing (KNL). We studied the calcium induced calcium release (CICR) model as an application example. In order to run the CICR code faster, enabling more studies, we considered the new and exciting KNL. Before working with the full CICR application code on the KNL, we used the Poisson problem to perform a conscientious test of the hardware. Finally, we used our understanding of the KNL to study the performance of the CICR application code with hybrid MPI+OpenMP implementation on multiple KNL nodes.

The existing advection-diffusion-reaction PDE CICR model with three-dimensional cell domain was extended to a complete model with 8 species for the electrical excitation, calcium signaling, and mechanical contraction systems. This includes feedback and feedforward links with the calcium signaling from the electrical excitation and the mechanical contraction systems. Simulations demonstrated the evacuation of calcium from the cell via the membrane pump and the feedback strength parameter range that did not demonstrate a significant effect on the system behavior. The simulations used 6 species of the model, without the actin-myosin cross-bridge species for mechanical contraction and without calsequestrin buffer species that binds to calcium ions in the SR. We demonstrated the need for parameter studies through an investigation of the coupling strength between the calcium system and the electrical system and the impact of the membrane pump. These studies required significant use of the CPU hardware on maya, even with a relatively coarse mesh.

The Poisson problem on a two-dimensional unit square is our first test of the KNL since the solution mimics the computational kernel of many simulations including CICR. We provided timing results using the different KNL configurations for a single KNL on the Stampede cluster at TACC. Our observations confirmed the need for hybrid MPI+OpenMP code to achieve optimal performance. Our results did not demonstrate any significant advantage to the choice of `KMP_AFFINITY=compact` or `KMP_AFFINITY=scatter` for this code. We confirmed the importance of using MCDRAM, the new high-performance memory on board the KNL chip, which performed almost 5x faster than using only the DDR4 memory of the node. We tested problem sizes as large as possible on a KNL node, thus exceeding the MCDRAM. This confirms the usefulness of the Cache Quadrant configuration, with MCDRAM as L3 cache, in such cases. For problems that fit into the 16 GB of MCDRAM, the Flat memory mode showed ability to perform as good as the Cache memory mode, but further study would be necessary to determine if Flat memory mode performs better than Cache memory mode. We found that using less than the 272 threads (68 cores with 4 threads per core) available, either via 256 threads, or less than 4 threads per core, could show improved run time, but was not very significant for our code. The use of a large number of MPI processes was not optimal, due to the significant MPI memory overhead required, despite comparable run times with a large number of MPI processes. In these tests, the use of a single KNL, using the MCDRAM, was more than 4x faster than the two 8-core CPUs on a Stampede node, 4x times faster than one KNC in native mode, and 3x times faster than one KNC in symmetric mode with the two 8-core CPUs.

We analyzed how to use the KNL for the CICR application code on Stampede. The CICR code requires more demanding and significant communication and is more computationally intensive than the Poisson problem. This application code put addi-

tional burden on the MPI communication and the balance with OpenMP threads and we observed this effect with two different MPI+OpenMP implementations. From our study of the Poisson problem, we focused on mesh sizes for CICR that fit inside the 16 GB of MCDRAM. As a result, we primarily use the Flat Quadrant configuration and make use of only the MCDRAM. We demonstrated good scalability on a single KNL using the baseline code with MPI parallelism only and compared against the scalability with OpenMP parallelism. We confirmed that it is beneficial to use several KNL and can scale well. With different hybrid MPI+OpenMP code implementations, we considered the number and placement of OpenMP threads relative to the number of MPI processes used and assessed the optimal number of OpenMP threads per core on the KNL. We observed that optimal choices for OpenMP threads to MPI processes and threads per core depended on the mesh resolution of the problem. For 64 cores using 8 MPI process with 16 threads per process, for a total of 128 threads or 2 threads per core, was a good general recommendation. For multiple KNL, using 68 cores showed some benefit over using only 64 and using 17 MPI processes with 8 OpenMP processes per thread was optimal.

We have demonstrated the potential for performance improvements of real-world application codes on the second-generation Intel Xeon Phi Knights Landing. As a result, there are many opportunities for further study. With the availability of the KNL and the guidance provided here, more effective studies of the CICR phenomenon can be performed. This can include parameter studies on the interplay between the feedback and feedforward links of the electrical excitation and calcium signaling components. The membrane pump and the presence of calsequestrin as a buffer species in the SR should have interesting impacts in the presence of the new couplings that should be studied. Also, we can run simulations with 8-species that add the actin-myosin cross-bridges as a third cytosol buffer species thus enabling the

feedback and feedforward links from calcium signaling to mechanical contraction. Additional opportunities for refining the model include modifications to the approach of the feed-forward link from the calcium to electrical system through inclusion of the sodium-calcium exchanger directly or modifying J_{LCC} inactivation to be dependent on the calcium in the cell. The introduction of a feedback link from the mechanical contraction to the electrical excitation system through the addition of stretch activated channels is also possible. At the same time, immediate further considerations for the KNL include finer mesh testing with multiple KNL nodes and more directed investigation of vectorization of the code for the KNL.

BIBLIOGRAPHY

[1] Amanda M. Alexander, Erin K. DeNardo, Eric Frazier III, Michael McCauley, Nicholas Rojina, Zana Coulibaly, Bradford E. Peercy, and Leighton T. Izu. Spontaneous calcium release in cardiac myocytes: Store overload and electrical dynamics. *Spora: A Journal of Biomathematics*, vol. 1, no. 1, 2015.

[2] Kallista Angeloff, Carlos Barajas, Alexander D. Middleton, Uchenna Osia, Jonathan S. Graf, Matthias K. Gobbert, and Zana Coulibaly. Examining the effect of introducing a link from electrical excitation to calcium dynamics in a cardiomyocyte. *Spora: A Journal of Biomathematics*, vol. 2, 2016.

[3] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, D. Dagum, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, R. S. Schreiber, H. D. Simon, V. Venkatakrishnan, and S. K. Weeratunga. The NAS parallel benchmarks. *Int. J. Supercomputer Appl.*, vol. 5, no. 3, pp. 63–73, 1991.

[4] Tamas Banyasz, Balazs Horvath, Zhong Jian, Leighton T Izu, and Ye Chen-Izu. Profile of L-type Ca 2+ current and Na+/Ca 2+ exchange current during cardiac action potential in ventricular myocytes. *Heart Rhythm*, vol. 9, no. 1, pp. 134–142, 2012.

[5] Dietrich Braess. *Finite Elements*. Cambridge University Press, third edition, 2007.

[6] Matthew W. Brewster, Jonathan S. Graf, Xuan Huang, Zana Coulibaly, Matthias K. Gobbert, and Bradford E. Peercy. Calcium induced calcium release with stochastic uniform flux density in a heart cell. In Saurabh Mittal, Il-Chul Moon, and Eugene Syriani, editors, *Summer Computer Simulation Conference*

(SCSC 2015), vol. 47 of *Simulation Series*, pp. 488–493. Curran Associates, Inc., 2015.

[7] Centers for Disease Control and Prevention, National Center for Health Statistics. Number of deaths for leading causes, 2015. `https://www.cdc.gov/nchs/fastats/leading-causes-of-death.htm`, page last updated October 7, 2016; accessed December 22, 2016.

[8] H. Cheng, W. J. Lederer, and M. B. Cannell. Calcium sparks: elementary events underlying excitation-contraction coupling in heart muscle. *Science*, vol. 262, pp. 740–744, 1993.

[9] James W. Demmel. *Applied Numerical Linear Algebra*. SIAM, 1997.

[10] Jack Dongarra and Michael A. Heroux. Toward a new metric for ranking high performance computing systems. Technical Report SAND2013–4744, Sandia National Laboratories, June 2013. `https://software.sandia.gov/hpcg/doc/HPCG-Benchmark.pdf`, accessed on March 23, 2017.

[11] Joshua N. Edwards and Lothar A. Blatter. Cardiac alternans and intracellular calcium cycling. *Clin. Exp. Pharmacol. P.*, vol. 41, no. 7, pp. 524–532, 2014.

[12] L. E. Ford and R. J. Podolsky. Regenerative calcium release within muscle cells. *Science*, vol. 167, pp. 58–59, 1970.

[13] Stephen A. Gaeta, Trine Krogh-Madsen, and David J. Christini. Feedback-control induced pattern formation in cardiac myocytes: a mathematical modeling study. *J. Theor. Biol.*, vol. 266, no. 3, pp. 408–418, 2010.

[14] Matthias K. Gobbert. Long-time simulations on high resolution meshes to model calcium waves in a heart cell. *SIAM J. Sci. Comput.*, vol. 30, no. 6, pp. 2922–2947, 2008.

[15] Jonathan S. Graf and Matthias K. Gobbert. Effective usage strategies for the configuration modes of the Intel Xeon Phi Knights Landing. Submitted (2017).

[16] Jonathan S. Graf, Matthias K. Gobbert, and Samuel Khuvis. Long-time simulations with complex code using multiple nodes of Intel Xeon Phi Knights Landing. In preparation (2017).

[17] Anne Greenbaum. *Iterative Methods for Solving Linear Systems*, vol. 17 of *Frontiers in Applied Mathematics*. SIAM, 1997.

[18] Alexander L. Hanhart, Matthias K. Gobbert, and Leighton T. Izu. A memory-efficient finite element method for systems of reaction-diffusion equations with non-smooth forcing. *J. Comput. Appl. Math.*, vol. 169, no. 2, pp. 431–458, 2004.

[19] M. A. Heroux, D. W. Doerfler, P. S. Crozier, J. M. Willenbring, H. C. Edwards, A. Williams, M. Rajan, E. R. Keiter, H. K. Thornquist, and R. W. Numrich. Improving performance via mini-applications. Technical Report SAND2009–5574, Sandia National Laboratories, 2009.

[20] Michael A. Heroux, Jack Dongarra, and Piotr Luszczek. HPCG technical specification. Technical Report SAND2013–8752, Sandia National Laboratories, October 2013. `https://software.sandia.gov/hpcg/doc/HPCG-Specification.pdf`, accessed on March 23, 2017.

[21] Xuan Huang and Matthias K. Gobbert. Parallel performance studies for a three-species application problem on the cluster maya. Technical Report HPCF–2015–

8, UMBC High Performance Computing Facility, University of Maryland, Baltimore County, 2015.

[22] Xuan Huang, Matthias K. Gobbert, Bradford E. Peercy, Stefan Kopecz, Philipp Birken, and Andreas Meister. Order investigation of scalable memory-efficient finite volume methods for parabolic advection-diffusion-reaction equations with point sources. In preparation (2017).

[23] Intel Xeon Phi coprocessor block diagram. http://www.intel.com/content/www/us/en/processors/xeon/xeon-phi-coprocessor-block-diagram.html, 2012. Accessed March 23, 2017.

[24] Intel re-architects the fundamental building block for high-performance computing — Intel newsroom. https://newsroom.intel.com/news-releases/intel-re-architects-the-fundamental-building-block-for-high-performance-computing/, June 23 2014. Accessed March 23, 2017.

[25] Thread affinity interface. https://software.intel.com/en-us/node/522691#AFFINITY_TYPES, 2015. Accessed March 23, 2017.

[26] Controlling thread allocation. https://software.intel.com/en-us/node/694293, 2016. Accessed March 23, 2017.

[27] Intel Xeon Phi processor 7250 (16 GB, 1.40 GHz, 68 core) specifications. http://ark.intel.com/products/94035/Intel-Xeon-Phi-Processor-7250-16GB-1_40-GHz-68-core, 2016. Accessed March 23, 2017.

[28] MCDRAM (high bandwidth memory) on Knights Landing — analysis methods & tools. https://software.intel.com/en-us/articles/mcdram-high-bandwidth-memory-on-knights-landing-analysis-methods-tools, 2016. Accessed March 23, 2017.

[29] Process and thread affinity for Intel Xeon Phi processors. https://software.intel.com/en-us/articles/process-and-thread-affinity-for-intel-xeon-phi-processors-x200, 2016. Accessed March 23, 2017.

[30] Intel Measured Results. *High Performance Conjugate Gradients*. www.umbc.edu/~gobbert/papers/KNL_UMBC.PNG, April 2016.

[31] Arieh Iserles. *A First Course in the Numerical Analysis of Differential Equations*. Cambridge Texts in Applied Mathematics. Cambridge University Press, second edition, 2009.

[32] Leighton T. Izu, Joseph R. H. Mauban, C. William Balke, and W. Gil Wier. Large currents generate cardiac Ca^{2+} sparks. *Biophys. J.*, vol. 80, pp. 88–102, 2001.

[33] Leighton T. Izu, Shawn A. Means, John N. Shadid, Ye Chen-Izu, and C. William Balke. Interplay of ryanodine receptor distribution and calcium dynamics. *Biophys. J.*, vol. 91, pp. 95–112, 2006.

[34] Leighton T. Izu, W. Gil Wier, and C. William Balke. Evolution of cardiac calcium waves from stochastic calcium sparks. *Biophys. J.*, vol. 80, pp. 103–120, 2001.

[35] Ishmail A. Jabbie, George Owen, Benjamin Whiteley, Jonathan S. Graf, Matthias K. Gobbert, and Samuel Khuvis. Performance comparison of Intel Xeon Phi Knights Landing. Submitted (2017).

[36] Samuel Khuvis. *Porting and Tuning Numerical Kernels in Real-World Applications to Many-Core Intel Xeon Phi Accelerators*. Ph.D. Thesis, Department of Mathematics and Statistics, University of Maryland, Baltimore County, 2016.

[37] Stampede KNL cluster user guide. `https://portal.tacc.utexas.edu/user-guides/stampede#stampede-knl-clusterknl`, 2016.

[38] Catherine Morris and Harold Lecar. Voltage oscillations in the barnacle giant muscle fiber. *Biophys. J.*, vol. 35, no. 1, p. 193, 1981.

[39] NASA Advanced Supercomputing Division. NAS parallel benchmarks, 2016. `http://www.nas.nasa.gov/publications/npb.html`, accessed March 23, 2017.

[40] Peter S. Pacheco. *Parallel Programming with MPI*. Morgan Kaufmann, 1997.

[41] Zhilin Qu, Michael Nivala, and James N Weiss. Calcium alternans in cardiac myocytes: Order from disorder. *J. Mol. Cell. Cardiol.*, vol. 58, pp. 100–109, 2013.

[42] C. Rosales. Porting to the Intel Xeon Phi: Opportunities and challenges. In *Extreme Scaling Workshop (XSW 2013)*, pp. 1–7. IEEE, 2013.

[43] C. Rosales, D. James, A. Gómez-Iglesias, J. Cazes, L. Huang, H. Liu, S. Liu, and W. Barth. KNL utilization guidelines. Technical Report TR–16–03, Texas Advanced Computing Center, The University of Texas at Austin, 2016.

[44] Carlos Rosales, John Cazes, Kent Milfeld, Antonio Gómez-Iglesias, Lars Koesterke, Lei Huang, and Jerome Vienne. A comparative study of application performance and scalability on the Intel Knights Landing processor. In Michela Taufer, Bernd Mohr, and Julian M. Kunkel, editors, *High Performance Computing: ISC High Performance 2016 International Workshops, ExaComm, E-MuCoCoS, HPC-IODC, IXPUG, IWOPH, P^3MA, VHPC, WOPSSS, Frankfurt, Germany, June 19–23, 2016, Revised Selected Papers*, vol. 9945 of *Lecture Notes in Computer Science*, pp. 307–318. Springer-Verlag, 2016.

[45] J. Sanderson. The SWORD of Damocles. *Lancet*, vol. 348, no. 9019, pp. 2–3, 1996.

[46] Jonas Schäfer, Xuan Huang, Stefan Kopecz, Philipp Birken, Matthias K. Gobbert, and Andreas Meister. A memory-efficient finite volume method for advection-diffusion-reaction systems with non-smooth sources. *Numer. Methods Partial Differential Equations*, vol. 31, no. 1, pp. 143–167, 2015.

[47] Thomas I. Seidman, Matthias K. Gobbert, David W. Trott, and Martin Kružík. Finite element approximation for time-dependent diffusion with measure-valued source. *Numer. Math.*, vol. 122, no. 4, pp. 709–723, 2012.

[48] W. C. Skamarock, J. B. Klemp, J. Dudhia, D. O. Gill, M. Barker, K. G. Duda, X. Y. Huang, W. Wang, and J. G. Powers. A description of the advanced research WRF version 3. Technical report, National Center for Atmospheric Research, 2008.

[49] A. Sodani, R. Gramunt, J. Corbal, H. S. Kim, K. Vinod, S. Chinthamani, S. Hutsell, R. Agarwal, and Y. C. Liu. Knights Landing: Second-generation Intel Xeon Phi product. *IEEE Micro*, vol. 32, no. 2, pp. 34–46, 2016.

[50] Avinash Sodani. Intel Xeon Phi processor "Knights Landing" architectural overview. `https://www.nersc.gov/assets/Uploads/KNL-ISC-2015-Workshop-Keynote.pdf`, 2015. Accessed March 23, 2017.

[51] John Towns, Timothy Cockerill, Maytal Dahan, Ian Foster, Kelly Gaither, Andrew Grimshaw, Victor Hazlewood, Scott Lathrop, Dave Lifka, Gregory D. Peterson, Ralph Roskies, J. Ray Scott, and Nancy Wilkins-Diehr. XSEDE: Accelerating scientific discovery. *Comput. Sci. Eng.*, vol. 16, no. 5, pp. 62–74, 2014.

[52] Stefan Wagner, Lars S. Maier, and Donald M. Bers. Role of sodium and calcium dysregulation in tachyarrhythmias in sudden cardiac death. *Circ. Res.*, vol. 116, no. 12, pp. 1956–1970, 2015.

[53] David S. Watkins. *Fundamentals of Matrix Computations*. Wiley, third edition, 2010.

[54] J. N. Weiss, A. Garfinkel, H. S. Karagueuzian, P. S. Chen, and Q. Zhilin. Early afterdepolarizations and cardiac arrhythmias. *Heart Rhythm*, vol. 7, no. 12, pp. 1891–1899, 2010.

[55] F. C. Wong, R. P. Martin, R. H. Arpaci-Dusseau, and D. E. Culler. Architectural requirements and scalability of the NAS parallel benchmarks. In *Proceedings of the 1999 ACM/IEEE Conference on Supercomputing*, pp. 41–41, 1999.