Long-time Simulations with Complex Code Using Multiple Nodes of Intel Xeon Phi Knights Landing

Jonathan S. Graf[a], Matthias K. Gobbert[a,*], Samuel Khuvis[a]

[a] *Department of Mathematics and Statistics, University of Maryland, Baltimore County, 1000 Hilltop Circle, Baltimore, MD 21250, U.S.A.*

Abstract

Modern partial differential equation (PDE) models across scientific disciplines require sophisticated numerical methods resulting in complex codes as well as large numbers of simulations for analysis like parameter studies and uncertainty quantification. To evaluate the behavior of the model for sufficeintly long times, for instance, to compare to laboratory time scales, often requires long-time simulations with small time steps and high mesh resolutions. This motivates the need for very efficient numerical methods and the use of parallel computing on the most recent modern architectures. We use complex code resulting from a PDE model of calcium dynamics in a heart cell to analyze the performance of the recently released Intel Xeon Phi Knights Landing (KNL). The KNL is a second-generation many-integrated-core (MIC) processor released in 2016 with a theoretical peak performance of over 3 TFLOP/s of double-precision floating-point operations for which complex codes can be easily ported because of the x86 compatibility of each KNL core. We demonstrate the benefit of hybrid MPI+OpenMP code when implemented effectively and run efficiently on the KNL including on multiple KNL nodes. For multi-KNL runs for our sample code, it is shown to be optimal to use all cores of each KNL, one MPI process on every other tile, and only two of the maximum of four threads per core.

Keywords: Intel Xeon Phi; Knights Landing; MPI; OpenMP; Parabolic partial differential equations; Calcium Induced Calcium Release.
2000 MSC: 35K61 65M08 65Y05 68U20 92C35

1. INTRODUCTION

The size and structure of modern 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 power consumption and heat generation, 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. Typical state-of-the-art distributed-memory clusters contain two multi-core CPUs per node with, for instance, 8 to 16 cores. Recent developments in parallel computing architectures also include the use of graphics processing units (GPUs) as a massively parallel accelerator, with thousands of special purpose cores, in general purpose computing and many-integrated-core (MIC) architectures like the Intel Xeon Phi with more than 60 cores. Besides the larger number of computational cores in both GPU and Phi, the key difference to a CPU is each one's significant on-chip memory, on the order of several GB, which contributes significantly to their performance gain over CPUs. A difference between GPU and Phi is the x86 compatibility of each Xeon Phi core that makes porting of code from Intel CPUs to this architecture much more readily possible, typically by recompiling with the suggested addition of a compiler flag.

The recent emergence of the second-generation Intel Xeon Phi in 2016, codenamed Knights Landing (KNL), represents a significant improvement over the first-generation in 2012, codenamed Knights Corner (KNC). The KNL was announced in June 2014 [11] 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-

*Corresponding author Tel. +1 410 455 2404; fax +1 410 455 1066.
Email addresses: jongraf1@umbc.edu (Jonathan S. Graf), gobbert@umbc.edu (Matthias K. Gobbert), khsa1@umbc.edu (Samuel Khuvis)

point performance [24]. Already the first-generation Phi 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. On the High Performance Conjugate Gradient (HPCG) benchmark list (www.hpcg-benchmark.org) the same two clusters made the top ten on November 2016 HPCG list. The Oakforest-PACS cluster earned the #3 in this case, and the Cori cluster at NERSC again ranked #5. The same KNL model as in these machines is used in this work.

We use the cluster 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) [25]. At the time of this writing in early 2017, the Stampede-KNL cluster at TACC had 504 available KNL nodes; since Fall 2017, Stampede 2 has now over 4,200 KNL nodes available. This makes the results here applicable, timely, and useful for many researchers going forward.

As example of complex code, we use a system of coupled, non-linear, time-dependent advection-diffusion-reaction equations of the form

$$u_t^{(i)} - \nabla \cdot \left(D^{(i)} \nabla u^{(i)} \right) + \beta^{(i)} \cdot \left(\nabla u^{(i)} \right) + a^{(i)} u^{(i)} = q^{(i)}, \quad i = 1, \ldots, n_s, \tag{1.1}$$

with functions $u^{(i)} = u^{(i)}(\mathbf{x}, t)$, $i = 1, \ldots, n_s$, of space $\mathbf{x} \in \Omega \subset \mathbb{R}^3$ with a three-dimensional domain and time $0 \leq t \leq t_{\text{fin}}$ representing the concentrations of the n_s species. We use the problem of modeling calcium dynamics in a heart cell as motivating application, which combines the need for fine meshes of the three-dimensional domain with the need for sophisticated numerical methods due to the large number of point sources for the most crucial feature of the application [2, 3, 6, 8, 17, 22]. To address the need for long simulations times that match the scale of laboratory experiments, 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 in (1.1) can both be dealt with [9, 22]. The matrix-free implementation of the iterative methods as linear solver avoids storing any system matrix and enables simulations even for fine meshes to fit in the on-chip memory of the KNL.

Despite the optimal memory usage, very fine meshes will eventually require more memory than one KNL provides. In this case, pooling the memory from several KNLs across several nodes enables the solution of large problems as well as offers the chance for speeding up calculations. Thus the scalability and performance of the code using multiple KNLs is very important to be tested.

We demonstrate the feasibility of porting and tuning special purpose application code to a single KNL and demonstrate the performance of multiple KNL. We also 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 work demonstrates the scalability of the MPI and OpenMP implementations, investigates the balance of MPI processes versus OpenMP threads, and the choice of number of threads per core to use on the KNL, intended to provide experiences useful to researchers considering using the KNL. Finally, we show the scalability of the code using more than one KNL.

The remainder of this paper is organized as follows. Section 2 describes the key features of the second-generation KNL to emphasize the differences from the first-generation KNC. Section 3 introduces the motivating application problem of calcium dynamics in a heart cell. Section 4 describes the numerical method used and its implementation in a complex special purpose code. Section 5 presents performance studies with the code on the KNL. We study MPI only code scalability, assess two different MPI+OpenMP implementations, and demonstrate the scalability and potential benefit of using more than multiple KNLs. Section 6 summarizes our conclusions on the performance of the code on the KNL and discusses opportunities for future work.

2. INTEL XEON PHI KNIGHTS LANDING

Unlike the special purpose cores in a GPU, 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 an x86 CPU. This represents a very significant advantage to the programmer, as their code can be quickly run on the Phi, typically with 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 cores versus 2.7 GHz CPU cores. But Phi chips also include on-chip memory, like GPUs, on the order of multiple GB while CPUs have essentially no on-chip memory, namely only L3 cache on the order of 20 MB. The KNL node DDR4 memory is connected though 6 channels and has a total of 96 GB, with larger capacity also possible. The 16 GB MCDRAM (Multi-Channel DRAM) memory on board the KNL 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 [24]. Therefore, for bandwidth bound problems that fit in the GBs of on-chip memory of the Phi that do not require usage of the lower performance nodal memory, the Phi can outperform CPUs.

The second-generation of the Phi, codenamed Knights Landing (KNL), represents a very different design from the first-generation of the Phi, codenamed Knights Corner (KNC). The KNC must be configured as a co-processor to a CPU, like a GPU is a co-processor to a CPU. The KNL can serve as a standalone processor, without a CPU host.

The KNC can have up to 61 cores with 8 GB of GDDR on-chip memory connected with the cores through a bidirectional ring bus as shown in Figure 2.1 (a). The KNL can have up to 72 cores, with 2 cores on each of the tiles in the 2D mesh interconnect that provides high-bandwidth connections between tiles and controllers on the chip as shown in Figure 2.1 (b). The KNL on-chip MCDRAM is nearly 50% faster than the KNC on-chip GDDR5 memory.

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 [23].

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 [14]. Each core has a clock rate of 1.4 GHz. 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 interconnect between nodes on the Stampede-KNL cluster is a 100 Gb/sec Intel Omni-Path network.

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 [15] 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."

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.

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 capitalizes 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, the

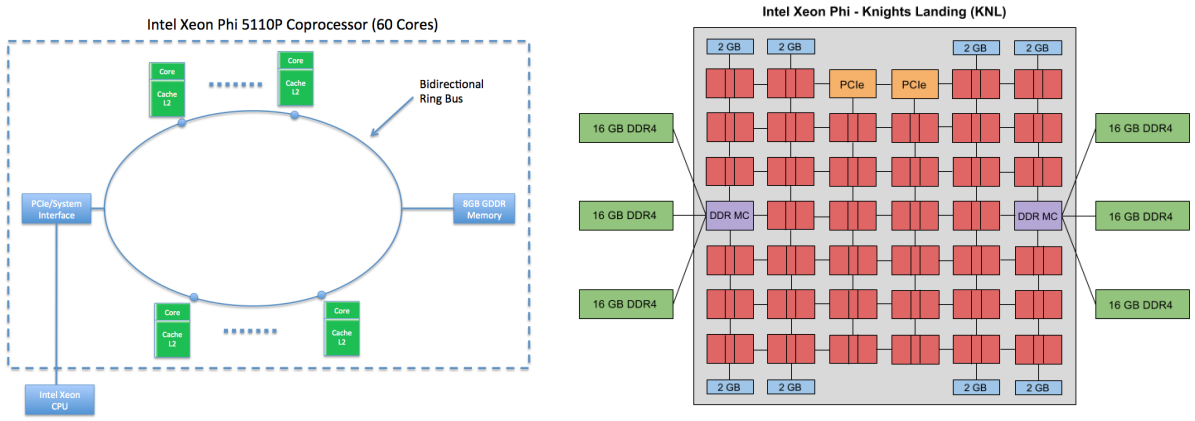


Figure 2.1: (a) Schematic of a KNC with 60 cores connected by a bi-directional ring bus to 8 GB on-chip memory. (b) Schematic of a KNL with 2 cores per tile, connected by a 2D mesh structure to 16 GB on-chip memory and 96 GB node memory. (Images courtesy of HPCF `hpcf.umbc.edu`.)

MCDRAM, or the MCDRAM and DDR4 once the MCDRAM fills up with run time flags that specify the type of memory to use.

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 [24] 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 memory 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.

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. For a simple understanding of the differences, consider that **All-to-All cluster mode** has no communication localized, **Quadrant cluster mode** has some communication localized, and **SNC-4 cluster mode** 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 [24]. 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 [24]. As a result, All-to-All cluster mode cache actions can have a higher latency than other mode. Quadrant cluster mode can have better latency than All-to-All, but SNC cluster mode should have the lowest latency among the three cluster modes.

3. APPLICATION PROBLEM

Our motivating problem of calcium dynamics in a heart cell is modeled by a system of time-dependent parabolic partial differential equations of the form (1.1), coupled by non-linear reaction and source terms in the right had side terms $q^{(i)}$. The full model described in [2] consists of 8 species. In this work, we focus on 6 species of the model, that is, $n_s = 6$ in (1.1). Table 3.1 lists the selected variables relevant to the simulation case used here. For a full description of the model with description of the physiological components, variables, parameter values, and coefficients, see [2] and the tables therein.

The study of this particular model is motivated by the overwhelming presence of heart disease as the leading cause of death in the United States according to the Centers for Disease Control and Prevention [4]. Models like the one we focus on seek to provide more in-depth understanding of the calcium dynamics that may yield new methods in the realm of drug therapy for the treatment of cardiac arrhythmias. In particular, it has been shown that the dysregulation of the interaction between the electrical, calcium, and mechanical systems is a precursor to cardiac arrhythmias [5, 20]. We use the model to perform simulations of the dynamics based on different physiological characteristics and parameters to complement the studies in a laboratory setting that require long time scales, often on the order of minutes. The most important components of the cell for this model are the calcium release units (CRUs) that are groups of individual calcium-sensitive ryanodine receptors. Calcium ions Ca^{2+} can be released, when the concentration of calcium is high enough, into the intracellular space known as the cytosol, from the primary calcium store in the cell, known as the sarcoplasmic reticulum (SR).

The domain in our model is a hexahedron, given by $\Omega = (-6.4, 6.4) \times (-6.4, 6.4) \times (-32.0, 32.0)$ in units of μm, to capture 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 Ω_s of all CRU locations of size $15 \times 15 \times 31 = 6{,}975$ CRUs throughout the interior of the cell. The numerical mesh required for the spatial discretization must be a least as fine as the CRU lattice, but in reality should be more refined in order to accurately capture the physiological behaviors. For physiological parameter studies we run simulations to final time of 1,000 ms at present, but longer simulation times are strongly desirable to approach the laboratory time scales of minutes eventually. To indicate the computational complexity of the model we discuss two key features:

(i) The key term in model is the way in which the release of calcium ions from the store in the SR to the cytosol is modeled by a term J_{CRU} included in the right-hand side $q^{(1)}$. The concentration of calcium ions in the cytosol is denoted by $c = u^{(1)}$ and the concentration of calcium ions in the SR is denoted by $s = u^{(4)}$ for short here. The effect of all CRUs is modeled as a superposition over all points $\hat{\mathbf{x}}$ of the CRU lattice Ω_s by

$$J_{CRU}(c, s, \mathbf{x}, t) \;=\; \sum_{\hat{\mathbf{x}} \in \Omega_s} \hat{\sigma}\, \tfrac{s-c}{s_0-c_0}\, \mathcal{O}(c, s, \mathbf{x}, t)\, \delta(\mathbf{x} - \hat{\mathbf{x}}) \tag{3.1}$$

with

$$\mathcal{O}(c, s, \mathbf{x}, t) \;=\; \begin{cases} 1 & \text{if } u_{rand} \le J_{prob}, \\ 0 & \text{if } u_{rand} > J_{prob}, \end{cases} \tag{3.2}$$

where

$$J_{prob}(c, s) = P_{max} \Big(\frac{c^{n_{prob}}}{K_{prob_c}^{n_{prob}} + c^{n_{prob}}} \Big) \Big(\frac{s^{n_{prob}}}{K_{prob_s}^{n_{prob}} + s^{n_{prob}}} \Big). \tag{3.3}$$

Table 3.1: Variables in the CICR model. Shorthand notation micromolar = μM = 10^{-6} mol/L.

Variable	Definition	Units
$\mathbf{x} = (x, y, z)$	spatial position	μm
t	time	ms
$u^{(1)}(\mathbf{x}, t)$	cytosol calcium concentration	μM
$u^{(2)}(\mathbf{x}, t)$	fluorescent dye concentration (cytosol buffer species)	μM
$u^{(3)}(\mathbf{x}, t)$	troponin concentration (cytosol buffer species)	μM
$u^{(4)}(\mathbf{x}, t)$	SR calcium concentration	μM
$u^{(5)}(\mathbf{x}, t)$	membrane potential (voltage)	mV
$u^{(6)}(\mathbf{x}, t)$	fraction of open potassium channels	1

Each CRU at its location $\hat{\mathbf{x}} \in \Omega_s$ is turned on and off by the gating function $\mathcal{O}$. This function lets calcium ions be released from the SR to the cytosol, when a uniform random number u_{rand} is less than an opening probability J_{prob}, which contains two fractions that model the effect of calcium concentration in the cytosol and SR, respectively, on the likelihood of a CRU opening. The first fraction is larger, if the concentration $c = u^{(1)}$ in the cytosol itself is larger; thus, this self-inducing mechanism gives rise to the term calcium induced calcium release (CICR). The second fraction in J_{prob} is larger, if the concentration $s = u^{(4)}$ in the SR is larger; this models the practical effect of "budgeting" the calcium SR stores such that when the stores are low, the CRU at that location becomes much less likely to open. Once a CRU opens, it stays open for 5 ms, after which it closes and is modeled not to re-open for 100 ms. During this time, other pump and leak terms in $q^{(1)}$ affect the regulation back to basal level $c = u^{(1)} = 0.1$ μM. The Dirac delta distribution $\delta(\mathbf{x} - \hat{\mathbf{x}})$ models each CRU as a point source for calcium release. The presence of the highly non-smooth Dirac delta distribution motivates the need for our special purpose code. The probabilistic model in the J_{CRU} term also demonstrates the need for uncertainty quantification in this model as was studied in [3].

(ii) The feedback coupling for the calcium signaling dynamics to the electrical excitation dynamics is achieved in the right-hand side term

$$q^{(5)} = \tau \frac{1}{C}\Big(I_{\text{app}} - g_L(V - V_L) - g_{Ca}\, m_\infty(V)\,(V - V_{Ca}) - g_K\, n\,(V - V_K) + \omega\,(J_{m_{pump}} - J_{m_{leak}})\Big), \quad (3.4)$$

with $V = u^{(5)}$ and $n = u^{(6)}$ for short. This term is an example of the need for many parameter studies with this model as in [1, 2]. In [2] the addition of the ω term in (3.4) for the feedback strength of the calcium signaling to the electrical excitation dynamics is studied with $\omega = 10, 30, 50$, and 100 μA ms/(μM cm^2) and compared to the case without feedback, parameterized by $\omega = 0$. We include that equation of the model explicitly here and show a specific case in the simulation here with $\omega = 10$. Each parameter study requires many simulations and the very long run times for each simulation make parameter studies very costly.

To demonstrate the capabilities of the code, we show some simulation results for the specific parameter set that is also used for the performance studies in this work. To give a flavor of results that can be obtained through the simulation, we show six different types of plots: CRU plots, isosurface plots, confocal images, a line scan, a voltage plot and a SR plot. Each of these plots displays different information in relation to the calcium dynamics within the system.

The CRU plots in Figure 3.1 show the open calcium release units, which are represented by blue dots in the domain.

The isosurface plots in Figure 3.2 show the concentration of Ca^{2+} in the cytosol, $c = u^{(1)}(\mathbf{x}, t)$. The color blue represents locations at which the Ca^{2+} concentration is 65 μM, indicating the presence of significantly more than basal level of calcium.

The confocal images in Figure 3.3 are a two-dimension view of the calcium concentration in the cytosol, $c = u^{(1)}(\mathbf{x}, t)$, consolidated to two dimensions through an integration along the y dimension. This is meant to replicate what scientists see in the laboratory experiments using florescent dye to bind to the calcium in the heart cell. The lighter shades of green indicate higher calcium concentrations, while the darker green shades indicate lower concentrations of calcium.

Figure 3.4 includes three other types of plots that each capture the full 1,000 ms simulation time in individual plots. Figure 3.4 (a) shows a line scan that is 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+} in the cytosol, $c = u^{(1)}(\mathbf{x}, t)$, are plotted on a two-dimensional domain versus time, and then overlaid upon each other producing the final image. Higher concentrations of Ca^{2+} are indicated by red, while lower concentrations are indicated by blue. Figure 3.4 (b) shows a SR plot that shows the concentration of Ca^{2+} in the SR, $s = u^{(4)}(\mathbf{x}, t)$ as function of time, at three characteristic points $\mathbf{x}$ at left, center, and right of the domain. Figure 3.4 (c) shows the voltage, $V = u^{(5)}(\mathbf{x}, t)$, at the center of the cell domain plotted versus time.

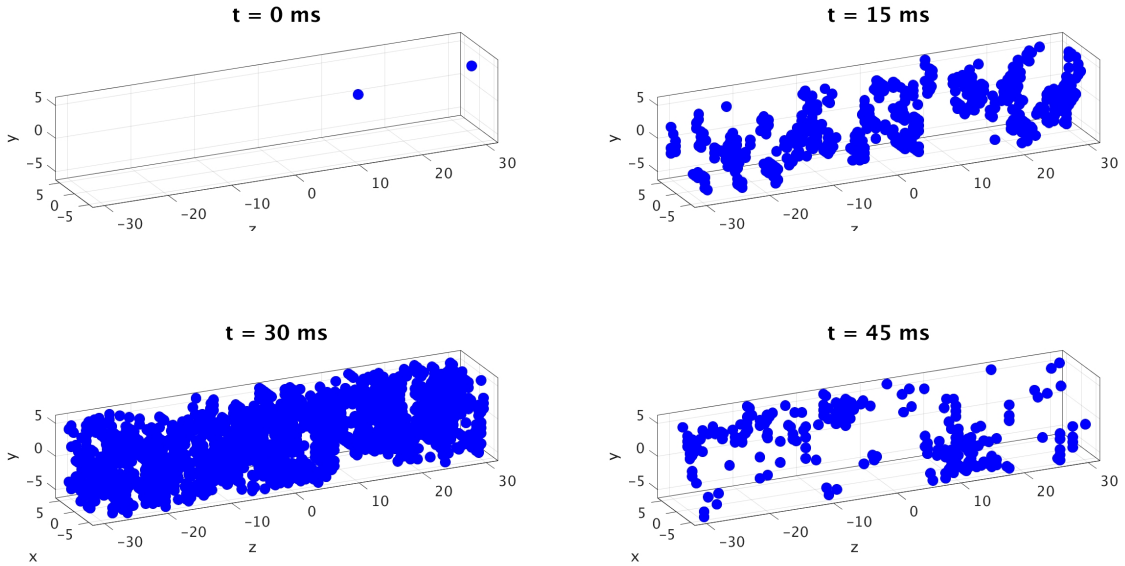


Figure 3.1: Snapshots of open CRUs at selected points in time showing almost zero open CRUs initially, through many open CRUs at $t = 30$ ms over the first 45 ms.

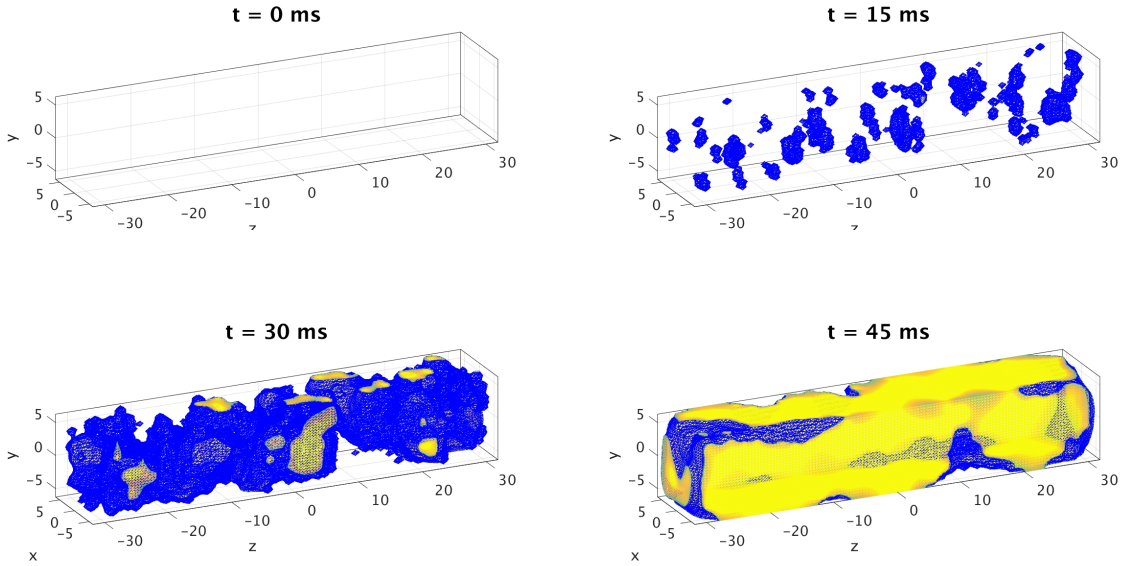


Figure 3.2: Snapshots of isosurface plots at selected points in time showing calcium concentration in the cytosol of the cell increasing dramatically over the first 50 ms of the simulation in accordance with the increasing number of open CRUs in the first 30 ms.

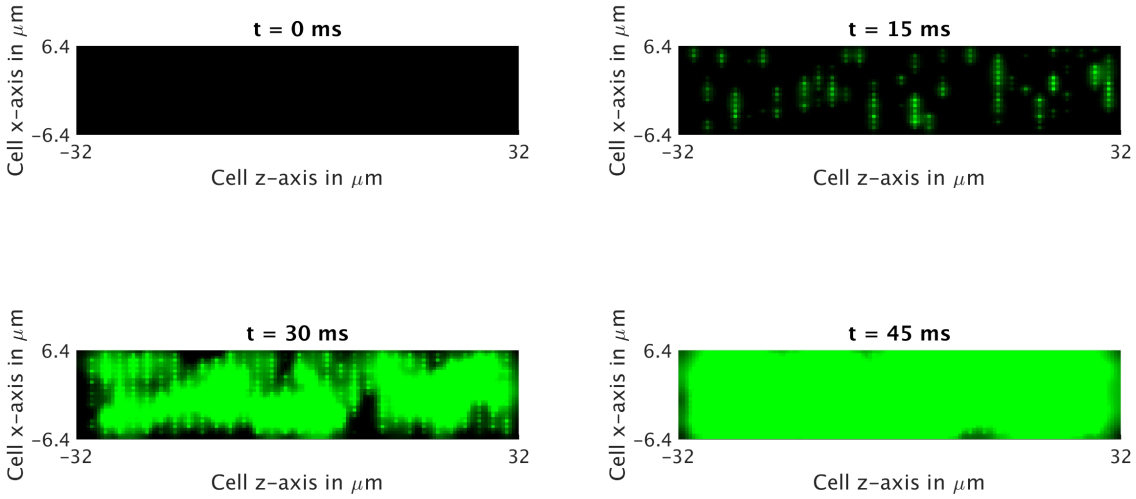
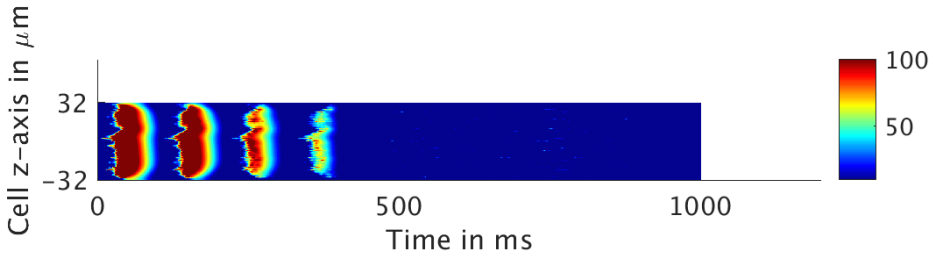
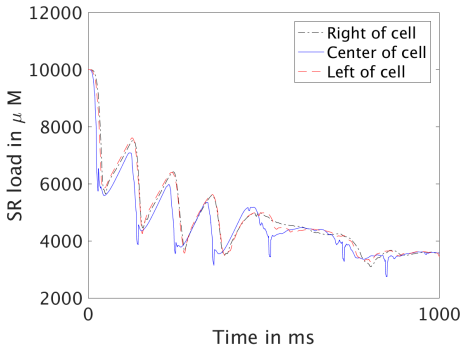


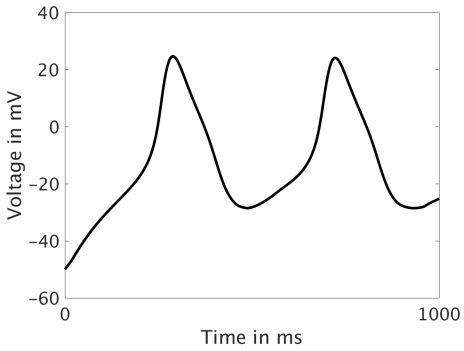
Figure 3.3: Snapshots of 2D confocal images at selected points in time showing calcium concentration in the cytosol of the cell increasing dramatically over the first 50 ms of the simulation in accordance with the increasing number of open CRUs in the first 30 ms.



(a) Line scan



(b) SR plot



(c) Voltage plot

Figure 3.4: (a) Line scan showing a 2D view of the calcium concentration in the cytosol. Calcium concentration in the cytosol increases rapidly to fill the cell is a repetitive manner four times before the SR calcium store is almost depleted of calcium. (b) SR plot showing the calcium concentration in the SR, with four sharp decreases corresponding with the significant increases of calcium in the cytosol as shown in (a). (c) Voltage at the center of the cell plotted against time with peaks around 280 and 720 ms. The third evacuation of calcium in the cell shown in the line scan (a) roughly corresponds with the first peak in voltage, but there is not enough calcium in the store to observe a visible release corresponding to the second peak in voltage. This choice of $\omega = 10$ shows a different behavior than the $\omega = 0$ case, demonstrating the feedback effect from the calcium signaling to the electrical excitation system.

4. NUMERICAL METHOD

We consider the system of time-dependent parabolic partial differential equations (PDEs) of the form (1.1) as test problem for parallel performance, since their implementation results in complex code with prototypical performance features. 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 at every time step. We use Newton's method as the non-linear solver, and at each Newton step we use the biconjugate gradient stabilized (BiCGSTAB) method as the linear solver. Complete details of the numerical method can be found in [9, 22].

Table 4.1 shows the number of degrees of freedom for different mesh sizes for this problem and gives the memory predictions for the 6 species 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 4.1, 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 4.1 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 be accommodated on a single KNL node.

By restricting our attention to mesh sizes 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 Section 3. The trade-off for the use of the finer meshes is the very long run times for each simulation. For this reason and to enable us to study finer meshes on the KNL with reasonable run times, we restrict our simulation final time for the parallel performance study to 10 ms only.

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. While `-O3` already includes vectorization of the code, the flag `-xMIC-AVX512` ensures that the optimal length of 512 bits is used for the vector instructions on the KNL. 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.

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.

Table 4.1: Sizing study 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) $n_{eq} = n_s N$, and the predicted memory usage in GB.

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

5. RESULTS

In this section, we present our study of the performance of the complex CICR code on the KNL. Given the significantly better performance using the high-performance on-chip MCDRAM as shown in [7, 18, 24], we focus our attention on all meshes in Table 4.1 that fit in the 16 GB of MCDRAM. Since we focus on runs that fit in the MCDRAM, we primarily use the Flat Quadrant KNL configuration using the MCDRAM only. We use our findings in [7, 18], to make initial decisions for our CICR runs on the KNL and test on a single KNL before using our results to guide our multiple KNL performance studies. The opportunity to significantly reduce runtimes through the use of multiple KNL nodes is very important for this and other models that require long simulation times, so the study of scalability and performance using multiple KNLs is very important in this case.

We also study different process and thread affinity types since it is another run time decision for the user. Intel provides some context and description for this in relation to the Phi architecture in [16]. The more general description of the thread affinity interface can be found in [12]. 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 but we focus on two popular choices with very different behaviors. With `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 different options provides some 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 [13]. 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` and `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. We can identify which hardware threads of the KNL are aligned with each KNL core by running the `lstopo` command on one of the KNL nodes. In particular we note 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, and so on.

We use the same physiological parameter set used in the simulations shown in Section 3. However, we use a final simulation time of 10 ms, rather than the 1,000 ms used in Section 3 or the longer simulation times required for laboratory scale experiments. We observed that there are approximately a factor of one hundred fewer time steps required for final time of 10 ms rather than 1,000 ms, thus the resulting run times are also approximately a factor of one hundred fewer in the 10 ms case.

The remainder of this section is organized as follows. Section 5.1 begins the study on a single KNL by testing the MPI only code for scalability and reporting memory usage observations. Section 5.2 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 5.3 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. Finally, Section 5.4 analyzes the performance of the code when pooling the resources of several KNL nodes. The results demonstrate the scalability and potential benefit of using more than one KNL with optimal run options in place.

5.1. MPI only code performance

We start with our existing MPI only code that was used in [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 5.2. As was done in [7], we present the memory observations for the code using different numbers of MPI processes. Recall for the Poisson problem in [7] 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 5.1 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 4.1 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 [7] 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 5.1 repeats the memory predictions from Table 4.1. From the second to the third column of Table 5.1 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. 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 5.2 on a single KNL in Cache Quadrant configuration rather than the Flat Quadrant configuration where we restricted our memory usage to the MCDRAM. The Cache Quadrant configuration is the primary configuration on the Stampede KNL cluster and performs comparably to Flat mode for problems that fit in the on chip memory as shown in [7].

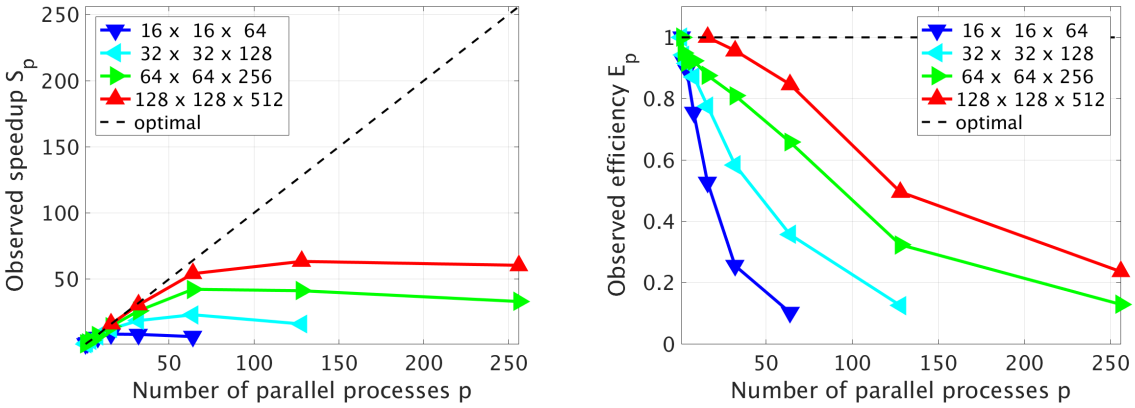
Table 5.2 presents a strong scalability study by number of MPI processes p. Each row lists the results for one problem size. Each column corresponds to the number of parallel processes p used in the run. Strong scalability is one key motivation for parallel computing: the run times for a problem of a 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 5.2 (b) shows the observed speedup S_p. Table 5.2 (c) shows the observed efficiency E_p.

In Table 5.2 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 5.1 (a) and (b) presents the customary graphical representations of speedup and efficiency, respectively, for code version 1 (MPI only) on a single KNL. Figure 5.1 (a) shows the speedup pattern in Table 5.2 (b) a bit more intuitively. The efficiency plotted in Figure 5.1 (b) is directly derived from the speedup, but the

Table 5.1: 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



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

Figure 5.1: Speedup (a) and Efficiency (b) plots for code version 1, MPI only, on one KNL using p MPI processes.

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 [10]. 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.

Table 5.2: 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 × 16 × 64	00:01:30	00:00:49	00:00:25	00:00:15	00:00:11	00:00:11	00:00:14	(**)	(**)
32 × 32 × 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 × 64 × 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 × 128 × 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 × 16 × 64	1.00	1.84	3.57	6.04	8.43	8.17	6.53	(**)	(**)
32 × 32 × 128	1.00	1.88	3.66	6.97	12.41	18.69	22.88	16.02	(**)
64 × 64 × 256	1.00	1.90	3.67	7.38	14.01	25.90	42.13	41.36	33.12
128 × 128 × 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 × 16 × 64	1.00	0.92	0.89	0.75	0.53	0.26	0.10	(**)	(**)
32 × 32 × 128	1.00	0.94	0.92	0.87	0.78	0.58	0.36	0.13	(**)
64 × 64 × 256	1.00	0.95	0.92	0.92	0.88	0.81	0.66	0.32	0.13
128 × 128 × 512	—	—	—	—	1.00	0.96	0.85	0.50	0.24

5.2. 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 5.3 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 5.3 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 5.3 (a) against Table 5.3 (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 [7].

We also test another important choice for running hybrid MPI+OpenMP code on the KNL in the distribution of threads to cores. Table 5.4 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 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 [7]. The choice of `KMP_AFFINITY=scatter` is also suggested from the results in [7] as well as the results in Table 5.3.

From Table 5.4 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 5.3 do not beat the best run times from version 1 of the code in Table 5.2 (a). However, Table 5.4 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 5.4 that using 1 or 2 threads per core gives the best performance. The optimal run times in Table 5.4 beat the best run times for version 1 of the code.

Table 5.3: 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`.

(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

5.3. 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 5.5 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 5.5 (a) `KMP_AFFINITY=compact` is used, while in Table 5.5 (b) `KMP_AFFINITY=scatter` is used. Next, Table 5.6 presents a performance study for different numbers of threads per core 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 5.7. Finally, Table 5.8 presents an OpenMP multi-threading strong scalability study on a single KNL.

In Table 5.5, we observe that neither MPI only nor OpenMP only parallelism performs as well as using MPI+OpenMP together. This matches our observation from Table 5.3 and again our conclusion from [2]. 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 5.5 (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, 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 5.3, the performance in Table 5.5 (a) with `KMP_AFFINITY=compact` is comparable to the performance with `KMP_AFFINITY=scatter` shown in Table 5.5 (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 5.3 to Table 5.5, 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

Table 5.4: 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

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 5.6 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 5.7 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 5.6 with Table 5.7, 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 5.4.

Table 5.8 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 to 8 processes or threads, the numbers in Table 5.8 and Table 5.2 are nearly identical.

Figure 5.2 (a) and (b) presents the customary graphical representations of speedup and efficiency, respectively, for code version 3 (MPI+OpenMP) on a single KNL. Figure 5.1 (a) shows the speedup pattern in Table 5.8 (b). The efficiency plotted in Figure 5.2 (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.

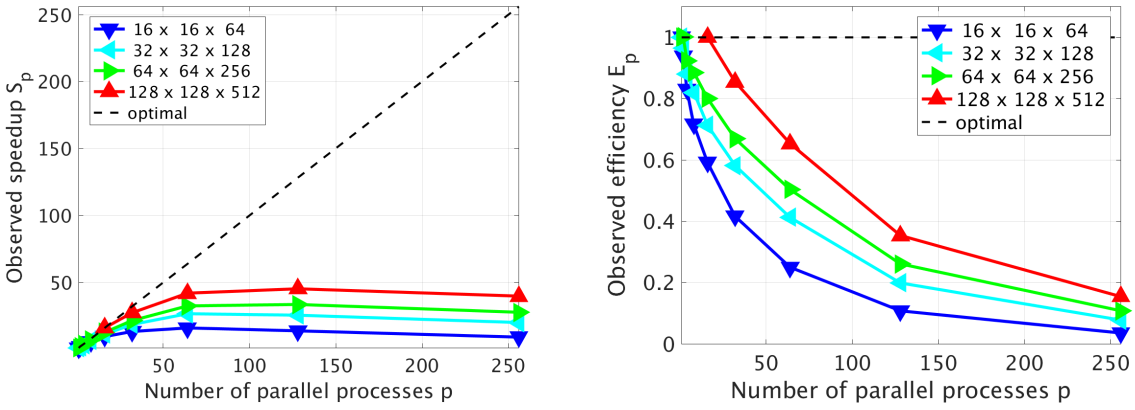
The scalability of version 1 of the code using MPI in Table 5.8 and Figure 5.2 is nearly identical to the scalability of version 3 of the code using OpenMP Table 5.2 and Figure 5.1 for small numbers of parallel process. For larger numbers of parallel process the MPI only code scales marginally better 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.

5.4. Multiple KNL

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 [19]. 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 [21] for the management of Intel Omni-Path Architecture (OPA)

Table 5.5: 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`.

(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



(a) Observed speedup S_p

(b) Observed efficiency E_p

Figure 5.2: Speedup (a) and Efficiency (b) plots for code version 3, MPI+OpenMP, on one KNL using p OpenMP threads.

traffic and improves scalability. It is also recommended in [21] 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 5.9 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 5.9 (a) we use only 8 MPI processes per node so that even the coarsest mesh can be run on 8 KNL

Table 5.6: 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 × 16 × 64	00:00:05	00:00:05	00:00:05	00:00:07	(**)	(**)
32 × 32 × 128	00:00:40	00:00:33	00:00:31	00:00:35	00:00:39	(**)
64 × 64 × 256	00:09:25	00:07:59	00:07:19	00:07:14	00:07:06	00:07:39
128 × 128 × 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 × 16 × 64	00:00:08	00:00:07	00:00:07	00:00:09	(**)	(**)
32 × 32 × 128	00:00:44	00:00:31	00:00:28	00:00:30	00:00:41	(**)
64 × 64 × 256	00:09:09	00:07:13	00:06:17	00:06:03	00:06:23	00:06:26
128 × 128 × 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 × 16 × 64	00:00:09	00:00:08	00:00:08	00:00:10	(**)	(**)
32 × 32 × 128	00:00:51	00:00:42	00:00:37	00:00:40	00:00:48	(**)
64 × 64 × 256	00:11:46	00:09:36	00:08:25	00:07:57	00:07:10	00:08:04
128 × 128 × 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 × 16 × 64	00:00:14	00:00:13	00:00:09	00:00:14	(**)	(**)
32 × 32 × 128	00:01:00	00:00:56	00:00:46	00:00:49	00:00:56	(**)
64 × 64 × 256	00:09:49	00:07:38	00:06:20	00:05:53	00:06:26	00:07:20
128 × 128 × 512	02:19:05	01:46:17	01:28:42	01:19:52	01:21:08	01:24:47

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 5.2 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 5.9 (b).

Table 5.9 (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 5.9 (a) with 8 MPI processes per node are hardly comparable to Table 5.9 (b) with 64 MPI processes per node. When using 8 MPI processes per node in Table 5.9 (a) it requires 8 KNL nodes to perform better than a single KNL with 64 MPI processes in Table 5.9 (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 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 5.9 (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 5.9 (a), but as was noted, the performance is significantly better.

Table 5.10 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 5.10 (a), we use 8 MPI processes per node and 16 OpenMP threads per process. In Table 5.10 (b), we use 16 MPI processes per node and 8 OpenMP threads per process. In Table 5.10 (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 5.7. Table 5.10 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 5.9.

We observe that the relative scalability and performance are not very different between these two choice of MPI process and OpenMP threads in Table 5.10 (a) and Table 5.10 (b), especially in the $128 \times 128 \times 512$ case. Table 5.10 (a) and Table 5.10 (b) also match the less than optimal run time performance in Table 5.9 (b) with MPI only parallelism. But, using 32 MPI processes with 4 OpenMP threads per core absolute run times in Table 5.10 (c) are better than Table 5.9 (b) with MPI only. This again demonstrates the need for hybrid code to achieve best performance.

Figure 5.3 presents a graphical representation of the hybrid MPI+OpenMP performance for the $128 \times$

Table 5.7: 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 5.8: 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.

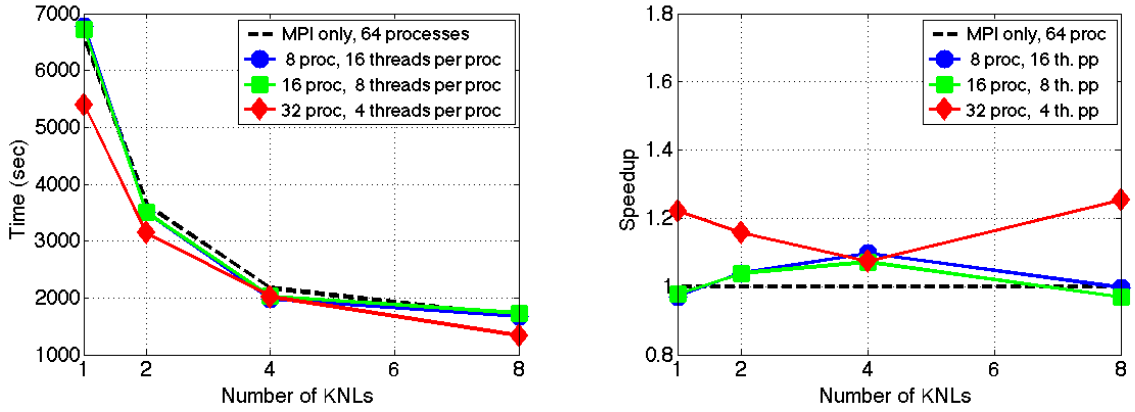
Code version 3, MPI+OpenMP – 1 KNL – Flat Quadrant Configuration – MCDRAM only									
(a) Wall clock time									
OpenMP threads p	1	2	4	8	16	32	64	128	256
$16 \times 16 \times 64$	00:01:29	00:00:47	00:00:27	00:00:15	00:00:09	00:00:07	00:00:06	(**)	(**)
$32 \times 32 \times 128$	00:18:54	00:09:47	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 5.9: 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

Table 5.10: 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 5.3: 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

128×512 mesh in Table 5.10 against the best MPI only performance from Table 5.9 (b). Figure 5.3 (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 5.10 (a), (b), and (c). Figure 5.3 (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 5.11 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 5.7 and Table 5.6, that show using 68 cores performs better than using only 64 cores. Table 5.6 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 5.10.

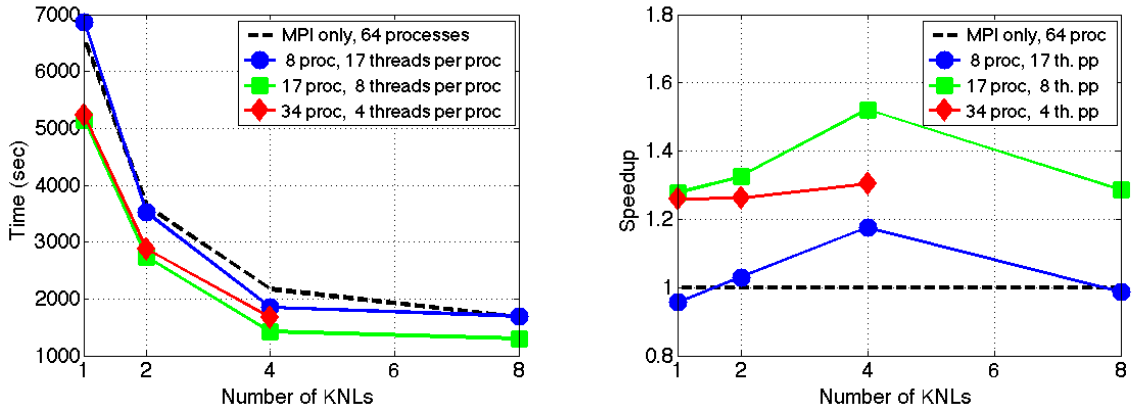
In the same manner as Table 5.10, Table 5.11 includes subtables with different combinations of MPI processes with OpenMP threads. In Table 5.11 (a), we use 8 MPI processes per node and 17 OpenMP threads per process. In Table 5.11 (b), we use 17 MPI processes per node and 8 OpenMP threads per process. In Table 5.11 (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 5.10 and Table 5.11 to determine the optimal choices for MPI processes and OpenMP threads when using multiple KNL. The best run times are in Table 5.11 (b), the 17 MPI processes per node with 8 threads per process case. Both Table 5.11 (c) and Table 5.10 (c) with 34 and 32 MPI processes per node with 4 threads, respectively, have comparable run times to Table 5.11 (b) that are not quite as good. Both Table 5.11 (a) and Table 5.10 (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 5.10 (b) with 16 MPI processes per node and 8 threads per process is comparable to Table 5.11 (a) and Table 5.10 (a) and is not optimal.

Figure 5.4 presents a graphical representation of the hybrid MPI+OpenMP performance in Table 5.11 against the best MPI only performance from Table 5.9 (b) in the same manner as Figure 5.3. Figure 5.4 (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 5.11 (a), (b), and (c). Figure 5.4 (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

Table 5.11: 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	(**)



(a) Wall clock times

(b) Speedup

Figure 5.4: 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

threads per processes, respectively, perform better than the MPI only case.

We can also compare Figures 5.3 and 5.4 to observe the speedup increase using 68 cores rather than 64 cores. The best speedup over the best MPI only run in Figure 5.3 (b) is just over 1.2, but in Figure 5.4 (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.

6. CONCLUSIONS AND OUTLOOK

We have demonstrated the potential for performance of complex codes on one as well as multiple KNLs using advantageous choices of the configurations modes based on the applications size and code characteristics. We used the CICR problem from [2] to show performance for a specific complex 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.

The CICR code requires more demanding and significant communication and is more computationally intensive than the Poisson problem from [7, 18]. 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. 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. The intermediate version 2 of the code clearly does not perform better than the original MPI-only version, which demonstrates that diligent work might be needed, such as in programming OpenMP pragmas in all needed places. We demonstrate here that the x86 compatibility of each Phi core is an advantage here, since it is possible to make incremental progress and to analyze performance, e.g., with VTune or other analyzing tools, at each step. Our final OpenMP implementation that implemented multi-threading also in the function with the non-linear reaction coupling terms demonstrated significantly better performance with fewer MPI processes 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. Since in reality, simulations using different mesh resolutions are often desired, e.g., a coarser one for large numbers of simulations for parameter studies or in uncertainty quantification vs. a finer mesh to produce high-resolution images, the flexibility of having a hybrid MPI+OpenMP code is useful.

We demonstrated scalability using multiple KNL with MPI only and our hybrid MPI+OpenMP code and observed the need to choose the optimal number and combination of MPI processes and OpenMP threads for best performance. The code scales well for the finest meshes for small numbers of KNLs and when using a small number of threads or cores. As observed on a single KNL, the best combinations of MPI processes and OpenMP threads using hybrid code on multiple KNLs outperformed using either MPI or OpenMP only, demonstrating the need for both in efficient production code. We observed that some choices of the distribution of MPI process to OpenMP threads result in poor performance relative to the best cases. In summary, for multi-KNL runs for our sample code, it is shown to be optimal to use all cores of each KNL, one MPI process on every other tile, and only two of the maximum of four threads per core. That is, the optimal combination of MPI processes and OpenMP threads per process uses a total of 17 MPI processes per node and 8 OpenMP threads per process. These specific results apply to this particular code, so researchers will have to experiment with their code. The result that it is benficial to use all cores, without leaving any idling, indicates to also investigate this possibility.

This work enables further study of the KNL performance using the full 8 species of the CICR model coupling both the mechanical contraction and electrical excitation systems to the calcium signaling system. With a firm understanding of KNL performance with the code, we can make use of 2 and 4 KNL nodes to run simulations efficiently to final times of 1,000 ms or more, rather than the 10 ms in the performance studies contained here. This is necessary for upcoming parameter studies of the additional components of the model and also since, for instance, laboratory time scales for this problem are on the order of minutes or only a modest multiple longer than achieved by 1,000 ms. We can explicitly test performance with different vectorization levels using the options of the Intel compiler and will continue to optimize OpenMP implementation choices.

Additionally, we have the opportunity to continue to improve the numerical methods in the code to take advantage of the features of the KNL.

ACKNOWLEDGMENTS

This work used the Extreme Science and Engineering Discovery Environment (XSEDE), which is supported by National Science Foundation grant number ACI–1053575 [25]. 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 paper. We also acknowledge 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). Jonathan Graf was supported by UMBC as HPCF RA.

REFERENCES

[1] Alexander, A. M., DeNardo, E. K., Frazier III, E., McCauley, M., Rojina, N., Coulibaly, Z., Peercy, B. E., Izu", L. T., 2015. "spontaneous calcium release in cardiac myocytes: Store overload and electrical dynamics". Spora: A Journal of Biomathematics 1, 36–48.
URL http://ir.library.illinoisstate.edu/spora/vol1/iss1/6

[2] Angeloff, K., Barajas, C. A., Middleton, A., Osia, U., Graf, J. S., Gobbert, M. K., Coulibaly, Z., 2016. Examining the effect of introducing a link from electrical excitation to calcium dynamics in a cardiomyocyte. Spora: A Journal of Biomathematics 2, 49–73.
URL http://ir.library.illinoisstate.edu/spora/vol2/iss1/6

[3] Brewster, M. W., Graf, J. S., Huang, X., Coulibaly, Z., Gobbert, M. K., Peercy, B. E., 2015. Calcium induced calcium release with stochastic uniform flux density in a heart cell. In: Mittal, S., Moon, I.-C., Syriani, E. (Eds.), Proceedings of the Conference on Summer Computer Simulation. SummerSim '15. Society for Computer Simulation International, pp. 1–6.

[4] Centers for Disease Control and Prevention, National Center for Health Statistics, 2015. Number of deaths for leading causes. https://www.cdc.gov/nchs/fastats/leading-causes-of-death.htm, page last reviewed March 17, 2017; accessed January 08, 2018.

[5] Gaeta, S. A., Krogh-Madsen, T., Christini, D. J., 2010. Feedback-control induced pattern formation in cardiac myocytes: a mathematical modeling study. J. Theor. Biol. 266 (3), 408–418.

[6] Gobbert, M. K., 2008. Long-time simulations on high resolution meshes to model calcium waves in a heart cell. SIAM J. Sci. Comput. 30 (6), 2922–2947.

[7] Graf, J. S., 2017. *Parallel Performance of Numerical Simulations for Applied Partial Differential Equation Models on the Intel Xeon Phi Knights Landing Processor*. Ph.D. Thesis, Department of Mathematics and Statistics, University of Maryland, Baltimore County.

[8] Hanhart, A. L., Gobbert, M. K., Izu, L. T., 2004. A memory-efficient finite element method for systems of reaction-diffusion equations with non-smooth forcing. J. Comput. Appl. Math. 169 (2), 431–458.

[9] Huang, X., 2015. *An MPI-CUDA Implementation of a Model for Calcium Induced Calcium Release in a Three-Dimensional Heart Cell on a Hybrid CPU/GPU Cluster*. Ph.D. Thesis, Department of Mathematics and Statistics, University of Maryland, Baltimore County.

[10] Huang, X., Gobbert, M. K., 2015. Parallel performance studies for a three-species application problem on the cluster maya. Tech. Rep. HPCF–2015–8, UMBC High Performance Computing Facility, University of Maryland, Baltimore County.
URL www.umbc.edu/hpcf

[11] Intel, June 23 2014. 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/, accessed January 08, 2018.

[12] Intel, 2015. Thread affinity interface. https://software.intel.com/en-us/node/522691#AFFINITY_TYPES, accessed January 08, 2018.

[13] Intel, 2016. Controlling thread allocation. https://software.intel.com/en-us/node/694293, accessed January 08, 2018.

[14] Intel, 2016. 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, accessed January 08, 2018.

[15] Intel, 2016. 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, accessed January 08, 2018.

[16] Intel, 2016. 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, accessed January 08, 2018.

[17] Izu, L. T., Means, S. A., Shadid, J. N., Chen-Izu, Y., Balke, C. W., 2006. Interplay of ryanodine receptor distribution and calcium dynamics. Biophys. J. 91, 95–112.

[18] Jabbie, I. A., Owen, G., Whiteley, B., 2017. Performance comparison of Intel Xeon Phi Knights Landing. SIAM Undergraduate Research Online (SIURO) 10.

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

[20] Qu, Z., Nivala, M., Weiss, J. N., 2013. Calcium alternans in cardiac myocytes: Order from disorder. J. Mol. Cell. Cardiol. 58, 100–109.

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

[22] Schäfer, J., Huang, X., Kopecz, S., Birken, P., Gobbert, M. K., Meister, A., 2015. A memory-efficient finite volume method for advection-diffusion-reaction systems with non-smooth sources. Numer. Methods Partial Differential Equations 31 (1), 143–167.

[23] Sodani, A., 2015. Intel Xeon Phi processor "Knights Landing" architectural overview. https://www.nersc.gov/assets/Uploads/KNL-ISC-2015-Workshop-Keynote.pdf, accessed January 08, 2018.

[24] Sodani, A., Gramunt, R., Corbal, J., Kim, H. S., Vinod, K., Chinthamani, S., Hutsell, S., Agarwal, R., Liu, Y. C., 2016. Knights Landing: Second-generation Intel Xeon Phi product. IEEE Micro 32 (2), 34–46.

[25] Towns, J., Cockerill, T., Dahan, M., Foster, I., Gaither, K., Grimshaw, A., Hazlewood, V., Lathrop, S., Lifka, D., Peterson, G. D., Roskies, R., Scott, J. R., Wilkins-Diehr, N., 2014. XSEDE: Accelerating scientific discovery. Comput. Sci. Eng. 16 (5), 62–74.