APPROVAL SHEET

Title of Thesis: Parallel Performance Studies for a Linear Parabolic Test Problem Using the Intel Xeon Phi

Name of Candidate: Ryan D. Day
Master of Science, 2016

Thesis and Abstract Approved: _____
Dr. Matthias K. Gobbert
Professor
Department of Mathematics and Statistics

Date Approved: _____

CURRICULUM VITAE

Name: Ryan David Day.
Degree and Date to be Conferred: Master of Science, 2016.

Collegiate Institutions Attended:

- Gettysburg College, 2010–2014, B.A. Mathematics and Religious Studies May 2014.
- University of Maryland, Baltimore County, 2014–2016, M.S. Applied Mathematics May 2016.

Teaching Experience:

- Teaching Assistant, Department of Mathematics and Statistics, UMBC, Fall 2014 – Spring 2015

Presentations:

- On the maximum size of an h-fold span of an m-element subset of an abelian group. Given at 23rd annual St. Joseph's University Research Symposium, St. Joseph's University, Philadelphia, PA, Spring 2012.
- On the size of $\nu_{\pm}(\mathbb{Z}_n, m, [0, s])$ for all n, m, and s. Given at Gettysburg College Mathematics Research Symposium, Gettysburg College, Spring 2012.
- On the maximum size of a restricted $[0, s]$-fold sumset. Given at Gettysburg College Mathematics Research Symposium, Gettysburg College, Spring 2013.

ABSTRACT

Title of Thesis: Parallel Performance Studies for a Linear Parabolic Test Problem Using the Intel Xeon Phi

Ryan D. Day, Master of Science, 2016

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

The performance of parallel computer code depends on several factors including the system hardware, the numerical algorithm chosen, and how the algorithm is implemented. We consider parallel performance of a parabolic test problem on the CPUs of one and multiple nodes and using the Intel Xeon Phi in native and symmetric mode, with MPI only and with hybrid MPI+OpenMP programming models.

We report the performance of a classical parabolic test problem whose structure is representative of kernels of real-world application codes. This test problem is the linear heat equation with homogeneous Dirichlet boundary conditions in two spatial dimensions on the unit square, which can be approximated using backward Euler for the time derivative and centered finite difference approximation for the spatial derivatives in the diffusion term. The implementation of the conjugate gradient method for the iterative solution of this system at each time step has the potential to perform well up to many parallel processes. This test problem lies in complexity between linear stationary elliptic and non-linear transient parabolic problems. Analyzing its performance based on excellent results for the former problems will give guidance on the potential for good performance on the latter ones.

We report parallel performance studies for the 2013 portion of the maya cluster in the UMBC High Performance Computing Facility and the Stampede cluster in the Texas Advanced Computing Center. We conduct parallel performance studies with MPI and OpenMP on the CPUs only as well as using CPUs in combination with

Intel Xeon Phi. The results show good performance using MPI on CPUs for up to 32 compute nodes. The results show code with a high degree of parallelism is required to take advantage of the many cores of the Phi and to achieve better performance than on CPUs and that for code with a sufficiently high degree of parallelism using both CPUs and Phis jointly on a hybrid node results in the best performance. The results show that code with smaller mesh resolutions is compute-bound and code with larger mesh resolutions is memory-bound.

PARALLEL PERFORMANCE STUDIES FOR A LINEAR PARABOLIC TEST PROBLEM USING THE INTEL XEON PHI

by

Ryan D. Day

Thesis submitted to the Faculty of the Graduate School
of the University of Maryland in partial fulfillment
of the requirements for the degree of
Master of Science
2016

To my parents, David and Susan Day

ACKNOWLEDGMENTS

Many thanks to my advisor, Dr. Matthias K. Gobbert, for his encouragement, advice, patience, and trust. I would also like to thank Jonathan Graf and Samuel Khuvis for their advice and assistance throughout my research.

The hardware used in the computational studies is part of the UMBC High Performance Computing Facility (HPCF). The facility is supported by the U.S. National Science Foundation through the MRI program (grant nos. CNS–0821258 and CNS–1228778) and the SCREMS program (grant no. DMS–0821311), with additional substantial support from the University of Maryland, Baltimore County (UMBC). See `hpcf.umbc.edu` for more information on HPCF and the projects using its resources.

This work used the Extreme Science and Engineering Discovery Environment (XSEDE), which is supported by National Science Foundation grant number ACI–1053575. I 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. See `www.tacc.utexas.edu` for information on TACC and its resources.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

1.1 Motivation

From the beginning of the twenty-first century there has been a shift in focus in modern CPUs from developing faster single cores to packaging more cores into a single CPU. As a result of this, parallel computing techniques must be developed to take full advantage of modern CPUs. Modern CPUs are typically packaged with 2 to 8 cores. Each of the compute nodes on the 2013 portion of the maya cluster in the UMBC High Performance Computing Facility (HPCF) and the Stampede cluster in the Texas Advanced Computing Center (TACC) contain 2 CPUs with 8 cores on each CPU. Programming techniques such as MPI and OpenMP allow for the utilization of all 16 computational cores in one node simultaneously. One concern for many-core CPUs is power consumption and heat dissipation. For large clusters of CPUs, these issues must be taken into consideration.

One solution to this issue is the use of coprocessors in a hybrid node to supplement the work of a CPU. Coprocessors generally have many more cores and threads than a multi-core CPU and use power more efficiently, although they usually have less memory than a modern CPU. A typical arrangement for a hybrid node is to have 2 CPUs each connected to a coprocessor. The most commonly used coprocessor used is the massively-parallel general-purpose Graphics Processing Unit (GPGPU).

Intel developed the many-core Intel Xeon Phi as an alternative to the GPGPU. The Intel Xeon Phi contains approximately 60 cores, with each core capable of 4 threads. There are far fewer threads on the Phi compared to the GPU, but each core of the Phi is x86 compatible and capable of running its own instruction stream, while

the GPU is not x86 compatible and operates in SIMD blocks [4]. This allows the user to run code on the Phi using MPI and OpenMP. The first Intel Xeon Phi was released to the public in 2013. Thus, it is still a relatively new technology that is not yet well understood for real-world applications.

The performance of the Intel Phi in real-world applications was studied by Khuvis [8]. In particular, his work shows that excellent performance can be obtained using the Intel Phi for a linear stationary elliptic test problem. The linear stationary elliptic test problem has been studied extensively on both the Phi and CPUs. Performance for an elliptic test problem has been studied extensively on the maya cluster. In addition to studies done for the Phi by Khuvis, studies on CPUs have been done in mutliple technical reports [10] [9]. These show excellent performance for the elliptic test problem on both the CPUs and the Phis. We extend these results to a linear parabolic test problem.

We consider the heat equation in two spatial dimensions on the unit square domain $\Omega = (0,1) \times (0,1) \subset \mathbb{R}^2$,

$$\begin{aligned} u_t - \Delta u &= f \quad \text{in } \Omega \quad \text{for } t > 0, \\ u &= 0 \quad \text{on } \partial\Omega \quad \text{for } t > 0, \\ u &= 0 \quad \text{in } \overline{\Omega} \quad \text{at } t = 0, \end{aligned} \tag{1.1.1}$$

with

$$\begin{aligned} f(x,y,t) = \frac{2t}{4} \mathrm{e}^{-t^2/4} \sin^2(\pi x) \sin^2(\pi y) \\ - 2\pi^2 (1 - \mathrm{e}^{-t^2/4}) [\cos(2\pi x) \sin^2(\pi y) + \sin^2(\pi x) \cos(2\pi y)]. \end{aligned} \tag{1.1.2}$$

This problem admits the known true solution

$$u(x,y,t) = (1 - e^{-t^2/4}) \sin^2(\pi x) \sin^2(\pi y). \tag{1.1.3}$$

This test problem lies in complexity between the linear stationary elliptic test problem and the non-linear transient parabolic problem of modeling calcium induced

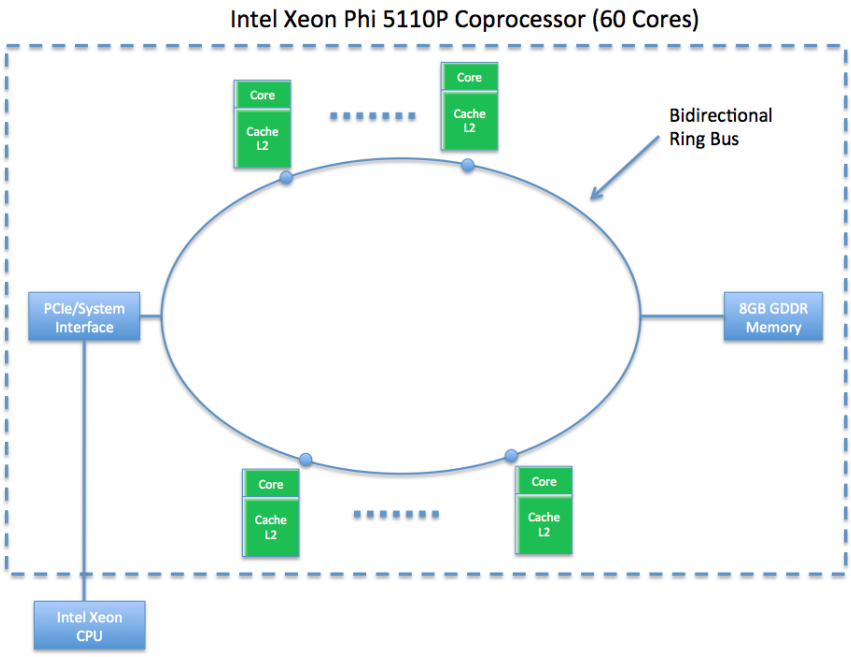


Figure 1.2.1 Schematic of Intel Phi 5110P. Figure credit HPCF.

calcium release (CICR) in a heart cell studied by Khuvis [8]. Analyzing its performance based on excellent results for the elliptic test problem will give guidance on the potential for good performance on the more complex CICR problem and help to determine where the problem for good performance of the complex codes lies. On the one hand, the linear stationary elliptic test problem consists of the Poisson equation, which is a scalar (i.e., single-species) linear partial differential equation. For this problem, results indicated excellent performance. On the other hand, the CICR problem is a multi-species problem, modeled mathematically by a system of transient non-linear partial differential equations. Its parallelization is a lot more difficult, and the work done by Khuvis did not show good performance, yet [8]. Thus, studying the performance of the scalar linear transient test problem (1.1.1) will help understand where performance bottlenecks are in the CICR problem.

1.2 Overview of Intel Xeon Phi Computing

The Intel Xeon Phi is a coprocessor developed by Intel which uses the Intel Many Integrated Core, or MIC, architecture. The Phi became available to the public in 2013. Figure 1.2.1 is a schematic of the Phi 5110P coprocessor available on maya connected to a CPU. Each Phi 5110P on maya has 60 cores core with dedicated L2 cache. These cores are connected to each other and 8 GB of main memory through a bidirectional ring bus. The Phi is then connected to its CPU by the PCIe interface.

Each core is capable of four threads for a total of 240 threads, or logical cores, on each Phi 5110P. Each core is x86 compatible and is capable of running its own instruction stream. Code developed for use with the Phi can use MPI and OpenMP frameworks due to the x86 compatibility.

There are three modes of running programs on the Intel Phi. These are offloading, native, and symmetric mode. In offload mode, the program is run on the CPU and segments of the code are offloaded to the Phi. The benefit of this mode is that multiple CPUs and Phis may be used in parallel. The downside to this mode is that communication between a CPU and its Phi is very expensive and that Phis can only communicate via the CPUs. In native mode, the program runs exclusively on the Phi. This is done by adding the `-mmic` flag to the compiler to create an executable that can be run directly on the Phi. One of the benefits of this mode is that it can be run using existing MPI or OpenMP CPU code, although this code may still need to be modified to obtain the best parallel performance. This mode limits the user to the use of Phis and no CPUs. Lastly, in symmetric mode the program is run on the CPU and on the Phi concurrently. This difference between this mode and native mode is that it allows for the use of all resources on a hybrid node. Unlike offload mode, this mode does not require modification of existing MPI code. The key to good performance in symmetric mode is proper load balancing between the CPUs and the

Phis which is accomplished by the choice of MPI processes and OpenMP threads in hybrid MPI+OpenMP code [8].

1.3 Computational Environment

The work in this thesis was completed using the UMBC High Performance Computing Facility and the Texas Advanced Computing Center.

The UMBC High Performance Computing Facility (HPCF) is the community-based, interdisciplinary core facility for scientific computing and research on parallel algorithms at UMBC. Started in 2008 by more than 20 researchers from ten academic departments and research centers from all three colleges, it is supported by faculty contributions, federal grants, and the UMBC administration. The facility is open to UMBC researchers at no charge. Researchers can contribute funding for long-term priority access. System administration is provided by the UMBC Division of Information Technology, and users have access to consulting support provided by dedicated full-time graduate assistants. See `hpcf.umbc.edu` for more information on HPCF and the projects using its resources.

The current machine in HPCF is the distributed-memory cluster maya with over 300 nodes. The newest components of the cluster are the 72 nodes with two eight-core 2.6 GHz Intel E5-2650v2 Ivy Bridge CPUs and 64 GB memory that include 19 hybrid nodes with two high-end NVIDIA K20 GPUs (graphics processing units) designed for scientific computing and 19 hybrid nodes with two cutting-edge 60-core Intel Xeon Phi 5110P accelerators. These new nodes are connected along with the 84 nodes with two quad-core 2.6 GHz Intel Nehalem X5550 CPUs and 24 GB memory by a high-speed quad-data rate (QDR) InfiniBand network for research on parallel algorithms. The remaining 168 nodes with two quad-core 2.8 GHz Intel Nehalem X5560 CPUs and 24 GB memory are designed for fastest number crunching and connected by a

dual-data rate (DDR) InfiniBand network. All nodes are connected via InfiniBand to a central storage of more than 750 TB.

The Texas Advanced Computing Center (TACC) at the University of Texas at Austin is one of the leading centers in computational excellence in the United States. TACC designs and operates some of the world's most powerful computing resources. The center's mission is to enable discoveries that advance science and society through the application of advanced computing technologies. TACC's resources are being used in over 200 research projects from over 100 institutions exploring over 50 fields of science at any time. See `www.tacc.utexas.edu` for more information on TACC and the projects using its resources. Access to TACC was obtained through the Extreme Science and Engineering Discovery Environment (XSEDE) [12].

Some work in this thesis was completed on TACC's Stampede system. Stampede is a 10 PFLOP/s (**P**eta **FL**oating Point **OP**erations per second) distributed memory cluster with over 6400 nodes. The majority of these 6400 nodes are compute nodes with two eight-core 2.7 GHz Intel E5-2680 Sandy Bridge CPUs and 32 GB memory. Most of these nodes are configured with one 61-core Intel Phi SE10P accelerator but a small number of nodes are configured with two Intel Phi SE10P accelerators. There are an additional 16 large-memory nodes with 32 cores per node and 1 TB of memory for data-intense applications. There are also 128 visualization nodes available with one NVIDIA K20 GPU on each node with 5 GB of GDDR5 memory. Each compute node contains a local 250 GB disk. Nodes are connected by a fourteen-data rate (FDR) InfiniBand network from Mellanox in a 2-level fat-tree topology.

1.4 Software Environments

The Intel C compiler version 15.0 is used on both Stampede and maya with compiler options `-c99 -Wall -O3` for MPI only code and additionally `-openmp` for the

MPI+OpenMP code. Intel MPI version 5.0.2 is used on Stampede and Intel MPI version 5.0.3 is used on maya. We use the Intel compiler and MPI because support for other versions is limited on the Intel Phi. The compiler flag `-mmic` is used to create executables for the Intel Phi. SLURM version 14.11 was used on both systems for resource management.

1.5 Summary and Outline

Chapter 2 is devoted to properties of the heat equation as well as the true solution of the heat equation (1.1.1) with the right hand side function (1.1.2).

Chapter 3 introduces the numerical methods for both the elliptic test problem and the linear parabolic test problem (1.1.1). Both methods are introduced, because the method for the time-dependent parabolic test problem is a generalization of the method for the elliptic test problem. We also explain the parallel implementation of these methods. This chapter also contains a sequence of careful convergence studies to ensure the numerical correctness and reliability of the parameter choices used in the performance studies.

In Chapter 4, the results of our parallel performance studies will be discussed. The performance of MPI only code and hybrid MPI+OpenMP code on the CPUs and the Intel Phis will be presented. We study performance on CPUs only, the Intel Phi in native mode, and the CPUs and Intel Phis in symmetric mode. One fundamental conclusion we make is that code with a high degree of parallelism is required to take advantage of the many cores of the Phi and to achieve better performance than on CPUs. Table 1.5.1 here provides a summary of the most important performance results of Chapter 4 for the case of a large mesh resolution using a 2048×2048 mesh, denoted by $N = 2048$. For a sufficiently large problem size such as for the mesh resolution in Table 1.5.1, the Intel Phi in native mode outperforms both CPUs in

one node, and using both CPUs and Intel Phis in symmetric mode further reduces the runtime. One observation we make is that on Stampede hybrid MPI+OpenMP code is key to good performance on the Intel Phi in native mode, while on maya and on the CPUs there is no difference in performance between MPI only code and MPI+OpenMP code.

Table 1.5.1 Summary of parabolic test problem results for $N = 2048$ on 1 hybrid node from Table 4.6.1.

Stampede					
CPUs	Phis	Mode	CPU prog. model	Phi prog. model	Runtime
2	0	CPU only	MPI only	NA	37:58:11
0	1	Native	NA	MPI+OpenMP	29:14:16
0	2	Native	NA	MPI+OpenMP	17:16:39
2	2	Symmetric	MPI only	MPI only	16:31:15
maya					
CPUs	Phis	Mode	CPU prog. model	Phi prog. model	Runtime
2	0	CPU only	MPI only	NA	36:19:55
0	1	Native	NA	MPI+OpenMP	31:28:06

Chapter 5 summarizes our conclusions on the performance of CPUs and the Intel Xeon Phi.

CHAPTER 2

MODEL PROBLEM

This chapter explains the background of the heat equation with Dirichlet boundary conditions in two spatial dimensions. Section 2.1 discusses properties of the heat equation and several properties of our test problem, (1.1.1). Section 2.2 discusses the true solution of our test problem and examines a plot of the true solution at time $t = 1, 2, \ldots, 6$.

2.1 Properties of the Heat Equation

The heat equation, also known as the diffusion equation, is typically used to describe the evolution in time of the density u of some quantity such as heat or chemical concentration. If our domain Ω is a smooth region in $\mathbb{R}^2$, then the rate of change of the total quantity within Ω equals the negative of the net flux through the boundary, $\partial\Omega$. For the homogeneous version of the heat equation, this is equivalent to the statement $u_t = -\operatorname{div} F$, where F is the flux density. The boundary conditions in (1.1.1) are $u = 0$ on $\partial\Omega$ for $t \geq 0$. This is known as homogeneous Dirichlet boundary conditions. This can be physically interpreted as saying that there is a constant temperature of 0 on our boundary for all times t.

There are several important properties of the heat equation that can be used in the discussion of its solution. The first of these properties is the uniqueness of classical solutions to the heat equation on bounded domains. Since the boundary and initial conditions are smooth, the solution exists and is unique [3]. In fact, our test problem is known to admit the true solution $u(x, y, t)$ in 1.1.3 that is infinitely often differentiable.

To test the numerical method and its implementation, we consider the parabolic

test problem (1.1.1) with right-hand side function (1.1.2) over the unit square $\Omega = (0,1) \times (0,1)$. The solution $u(x,y,t)$ in (1.1.3) to this test problem has several properties worth mentioning. The first is that we have

$$\min_{(x,y)\in\Omega} u(x,y,t) = 0. \tag{2.1.1}$$

The next is that

$$\max_{(x,y)\in\Omega} u(x,y,t) = 1 - e^{-t^2/4}. \tag{2.1.2}$$

The solution tends to the steady-state quickly with

$$u_{SS}(x,y) = \sin^2(\pi x)\sin^2(\pi y) \quad \text{as } t \to \infty, \tag{2.1.3}$$

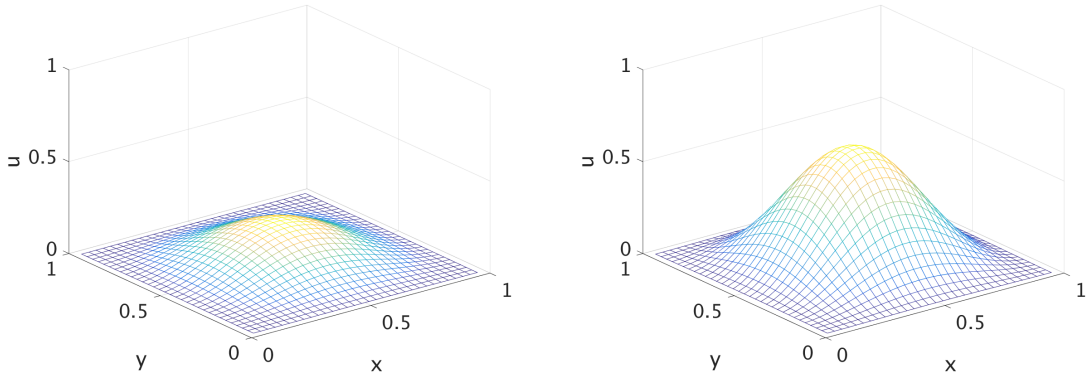
since $1 - e^{-t^2/4} \to 1$ rapidly as $t \to \infty$. So the solution to our test problem is non-negative at all times t for all points $(x,y) \in \overline{\Omega}$. The solution is 0 at points $(x,y) \in \overline{\Omega}$ at $t = 0$, and is 0 on $\partial\Omega$ for $t > 0$. For $t > 0$, the solution for all $(x,y) \in \Omega$ is positive. The solution changes very rapidly in space for times $1 \leq t \leq 4$, since $1 - e^{-t^2/4}$ changes rapidly, but does not change much for larger values of t. The solutions tends to steady-state, (2.1.3), very quickly in time, with the solution barely changing for values of t larger than 6. For fixed times t, the maximum (2.1.2) occurs at the center of the spatial domain, $(x,y) = (1/2, 1/2)$.

2.2 True Solution of the Model Problem

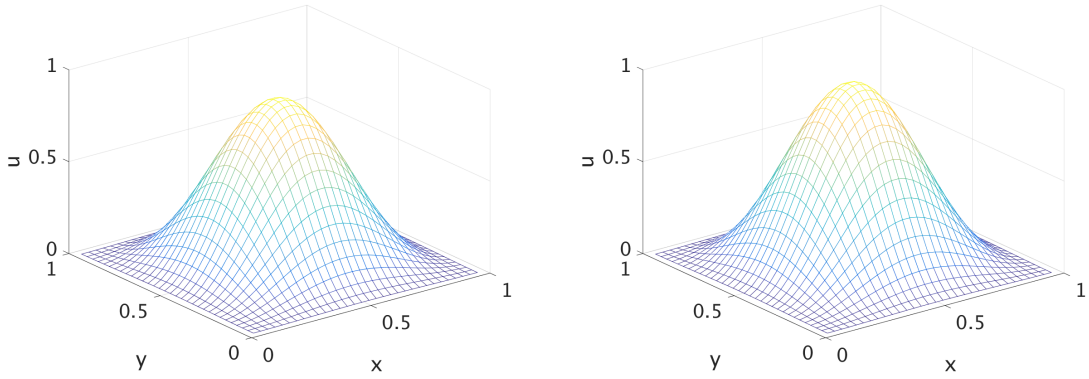
In this section we discuss the solution of the test problem, and examine plots of the true solution at times $t = 1, 2, \ldots, 6$. Figure 2.2.1 plots the true solution at times $t = 1, 2, \ldots, 6$. Examining the plots of the true solution, we notice that u converges to this steady state quickly in time, as the solution for fixed (x,y) barely changes for $t \geq 5$. This confirms our observation in Section 2.1 that the solution of (1.1.1) tends to steady-state quickly. If we were to plot the solution for times larger than 6, the

solution would appear nearly identical to the solution at $t = 6$. However, if we examine the plots of the solution for $1 \leq t \leq 4$, we see that the solution changes rapidly in time. This confirms another observation we made in the previous section that the solution changes very rapidly when $1 \leq t \leq 4$. We can also see from Figure 2.2.1 that the solution is positive for all $(x, y) \in \Omega$, and the solution is 0 on $\partial\Omega$, and that for fixed t, the maximum of u occurs at the center of the domain, $(x, y) = (1/2, 1/2)$.

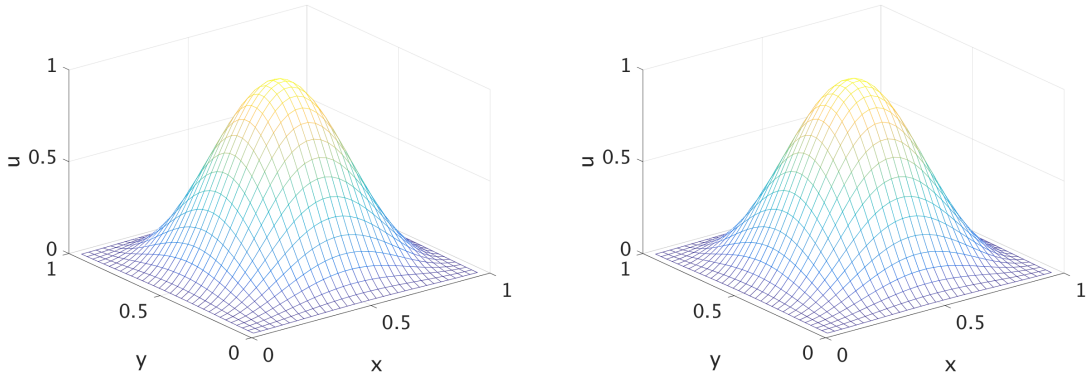
Each of the properties of the true solution that we discussed make this an excellent test problem for our numerical method. The true solution to this problem is known, which allows us to verify the convergence of our numerical method. Further, as we have discussed, the solution to this problem changes fairly rapidly with respect to time when $1 \leq t \leq 4$, but changes more and more slowly as t increases past this point.



(a) $t = 1$ (b) $t = 2$



(c) $t = 3$ (d) $t = 4$



(e) $t = 5$ (f) $t = 6$

Figure 2.2.1 Mesh plots of the true solution (1.1.3) at times $t = 1, 2, \ldots, 6$ using mesh resolution $N = 32$.

CHAPTER 3

NUMERICAL METHOD AND ITS IMPLEMENTATION

This chapter explains the numerical method used to obtain a solution to the heat equation for our test problem. We first discuss a numerical method for obtaining a solution to the Poisson equation in Section 3.1, because this method is closely related to the numerical method we use to solve the heat equation. In Section 3.2, we build upon the numerical method implemented in Section 3.1 by developing a numerical solution for our test problem. In Section 3.3, we discuss convergence theory for the numerical methods used to solve the Poisson equation in Section 3.1 and the heat equation in Section 3.2, and we conduct convergence studies using Matlab. In Section 3.4, we discuss how the numerical methods are implemented in C code, and in Section 3.5 we report the results of a convergence study for our test problem for the linear heat equation using the serial and parallel C code.

3.1 Finite Difference Discretization of the Poisson Equation

We first present a numerical solution of the Poisson equation with homogeneous Dirichlet boundary conditions,

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

on the unit square domain $\Omega = (0,1) \times (0,1) \subset \mathbb{R}^2$. The Laplacian Δu is defined as $\Delta u = \frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2}$. Using the finite difference method to approximate Δu produces a system of linear equations that can be solved using the conjugate gradient method, an iterative method for solving a system of linear equations. As we will see, our system matrix is symmetric positive definite, and therefore the conjugate gradient method is guaranteed to converge for this problem.

In order to utilize the finite difference method, we need to discretize the closure of the domain $\overline{\Omega} = [0,1] \times [0,1]$. This can be done by constructing a mesh using $N+2$ uniformly spaced mesh points in each dimension, where N is our mesh resolution. Then we have uniform mesh spacing $h = 1/(N+1)$, with each mesh point evenly spaced over $\overline{\Omega}$. In order to solve the Poisson problem using finite differences, each of the derivatives in the Laplacian operator Δu is replaced by a numerical differentiation approximation. This approximation is applied to all interior mesh points (x_i, y_j) on Ω, with $x_i = hi$, $i = 0, 1, \ldots, N, N+1$ and $y_j = hj$, $j = 0, 1, \ldots, N, N+1$. We denote this set of discrete mesh points by Ω_h. Then denote the approximation to the solution at each of the mesh points $(x_i, y_j) \in \Omega_h$ by $u_{i,j} \approx u(x_i, y_j)$. We can approximate the Laplacian at each interior mesh point using a centered finite difference approximation,

$$\frac{\partial^2 u(x_i, y_j)}{\partial x^2} + \frac{\partial^2 u(x_i, y_j)}{\partial y^2} \approx \frac{u_{i-1,j} - 2u_{i,j} + u_{i+1,j}}{h^2} + \frac{u_{i,j-1} - 2u_{i,j} + u_{i,j+1}}{h^2}. \quad (3.1.2)$$

Using this approximation along with the boundary conditions in (3.1.1), we get a system of N^2 linear equations that give us a finite difference approximation for u at each of the interior mesh points.

We can collect the unknown approximation (3.1.2) at each interior mesh point into a vector $u \in \mathbb{R}^{N^2}$ with components $u_k = u_{ij}$ with $k = i + N(j-1)$ for $i, j = 1, \ldots, N$. and state the problem as $Ku = b$, where $K \in \mathbb{R}^{N^2 \times N^2}$ is a system matrix and $b \in \mathbb{R}^{N^2}$ is the vector of values of the right-hand side function f in (3.1.1) at the interior mesh points in Ω_h, giving us $b_k = f(x_i, y_j)$, with $k = i + N(j-1)$ for $i, j = 1, \ldots, N$. The system matrix K can be constructed using the coefficients in (3.1.2). Using this

approximation, we have

$$K = \frac{1}{h^2}\begin{bmatrix} S & T & & & \\ T & S & T & & \\ & \ddots & \ddots & \ddots & \\ & & T & S & T \\ & & & T & S \end{bmatrix} \tag{3.1.3}$$

where $S \in \mathbb{R}^{N\times N}$ is a tri-diagonal matrix, $S = \text{tridiag}(-1, 4, -1)$, and $T = -I \in \mathbb{R}^{N\times N}$. As previously mentioned, K is known to be symmetric positive definite, and therefore the conjugate gradient method is guaranteed to converge for this problem [13]. For more discussion of this problem, including parallel performance studies, see [10].

3.2 Full Discretization of the Heat Equation

In this section we present a numerical method for the classical parabolic test problem of the linear heat equation with homogeneous Dirichlet boundary conditions in two spatial dimensions, (1.1.1), over the unit square $\Omega = (0, 1) \times (0, 1) \in \mathbb{R}^2$, with right-hand side function (1.1.2). Our numerical method for solving this problem is closely related to the method discussed in Section 3.1. We use a full finite difference discretization. With this method, both space and time are discretized simultaneously.

We discretize the closure of the domain $\overline{\Omega} = [0, 1] \times [0, 1]$ just as we did for the Poisson problem, using uniform mesh spacing $h = 1/(N+1)$ for (x_i, y_j), with $x_i = hi$, $i = 0, 1, \ldots, N, N + 1$ and $y_j = hj$, $j = 0, 1, \ldots, N, N + 1$. We denote this set of discrete mesh points by Ω_h. We discretize the time domain using a fixed time step Δt by defining $t_n = n\ \Delta t$ for $n = 0, 1, \ldots, N_t$, such that $N_t\ \Delta t = t_{\text{fin}}$, where t_{fin} is the final time we are solving the heat equation up to. We denote the approximation of the solution at an interior mesh point $(x_i, y_j) \in \Omega_h$ at time t_n by $u_{ij}^n \approx u(x_i, y_j, t_n)$.

We use backwards Euler to approximate the derivative with respect to time and a centered finite difference approximation for the spatial derivatives. Entering these approximations into the PDE in (1.1.1) at time t_{n+1} results in a finite difference approximation for u_{ij}^{n+1},

$$\frac{u_{ij}^{n+1} - u_{ij}^{n}}{\Delta t} + \frac{-u_{i,j-1}^{n+1} - u_{i-1,j}^{n+1} + 4u_{ij}^{n+1} - u_{i+1,j}^{n+1} - u_{i,j+1}^{n+1}}{h^2} = f_{ij}^{n+1}, \tag{3.2.1}$$

where we use short-hand notation $f_{ij}^{n+1} = f(x_i, y_j, t_{n+1})$. Multiplying both sides by Δt and moving u_{ij}^{n} to the right-hand side, we get

$$u_{ij}^{n+1} + \Delta t\, \frac{-u_{i,j-1}^{n+1} - u_{i-1,j}^{n+1} + 4u_{ij}^{n+1} - u_{i+1,j}^{n+1} - u_{i,j+1}^{n+1}}{h^2} = u_{ij}^{n} + \Delta t\, f_{ij}^{n+1} \tag{3.2.2}$$

at all interior mesh points. We can collect the N^2 unknown approximations u_{ij}^{n+1} into a vector $u^{n+1} \in \mathbb{R}^{N^2}$ with components $u_k^{n+1} = u_{ij}^{n+1}$ with $k = i + N(j-1)$ for $i, j = 1, \ldots, N$. We can add the components of the solution u_{ij}^{n} from the previous time step t_n to $\Delta t\, f_{ij}^{n+1}$ and collect these values into a vector $b^n \in \mathbb{R}^{N^2}$ with components $b_k^n = u_{ij}^{n} + \Delta t\, f_{ij}^{n+1}$ with $k = i + N(j-1)$ for $i, j = 1, \ldots, N$. Organizing these equations into matrix-vector form, we obtain

$$u^{n+1} + \Delta t\, K u^{n+1} = b^n \quad \text{for } n = 0, 1, \ldots, N_t - 1, \tag{3.2.3}$$

where K is identical to (3.1.3), the system matrix for the Poisson problem in Section 3.1 Then at each time step we can arrange these terms into a system of linear equations of the form $A\, u^{n+1} = b^n$, where $A = I + \Delta t\, K$. We finally write the method for u^{n+1} at each time step by solving the system of linear equations,

$$A\, u^{n+1} = b^n \text{ for } n = 0, 1, \ldots, N_t - 1. \tag{3.2.4}$$

Along with K, also $A = I + \Delta t\, K$ is symmetric positive definite, and therefore the conjugate gradient method is guaranteed to converge and is appropriate to solve the linear system at each time step. Given that an implementation for the Poisson

problem in Section 3.1 is available, this approach is very simple to implement by extending the available implementation to the heat equation. The convergence of the iterative conjugate gradient method is also expected to be excellent, since the solution u^n at the previous time step is available as excellent initial guess for the conjugate gradient method to compute u^{n+1}.

3.3 Convergence Studies for the Numerical Method in Matlab

The finite difference method for the Poisson equation with Dirichlet boundary conditions, (3.1.1), is second-order convergent [2, Chapter I], as summarized in the following theorem.

Theorem 3.3.1 *Consider the Poisson equation with Dirichlet boundary conditions* $-\Delta u = f$. *For* $u \in C^4(\overline{\Omega})$, *the finite difference approximation* u_h *given by (3.1.2) converges to the solution* u *of* $-\Delta u = f$ *and satisfies*

$$\|u(\cdot,\cdot,t) - u_h(\cdot,\cdot,t)\|_{L^\infty(\Omega)} \leq C\, h^2 \quad \text{as } h \to 0,$$

where h *denotes the grid size.*

This convergence behavior can be seen in the convergence studies in [10].

For our test problem, since we use a centered finite difference approximation for the spatial derivatives in the linear heat equation, we can expect the spatial errors to be on the order of h^2. Similar to Theorem 3.3.1, we have

$$\|u(\cdot,\cdot,t) - u_h(\cdot,\cdot,t)\|_{L^\infty(\Omega)} \leq C\, h^2 \quad \text{as } h \to 0$$

for all times t. We use backwards Euler to approximate the derivative with respect to time. The time discretization by implicit Euler has error on the order of Δt [1, 7]. In order to obtain proper convergence for our test problem, we need balance the errors from time and space. Combining the spatial and time discretization errors, we expect the total error for our method to be on the order of $\Delta t + h^2$.

The goal of this work is to conduct parallel performance studies in different programming environments and hardwares. We wish to conduct these using a computationally challenging test problem, solved by a convergent numerical method. We mean by this a test problem, whose numerical parameters are chosen such that the problem is a significant challenge to solve in reasonable amount of time in serial and whose numerical solution is confirmed to converge to the true solution in accordance with the numerical theory. In order to obtain proper convergence for our test problem it is necessary to test several time step sizes Δt. In order to determine a proper time step, we first solve the system of linear equations (3.2.4) at each discrete time step using Gaussian elimination. Once a sufficiently small time step is determined, we can then test convergence using the conjugate gradient method to solve (3.2.4) at each time step with several different tolerances. This will allow us to determine a sufficiently tight tolerance for the conjugate gradient method to maintain proper spatial convergence.

To obtain convergence for this method, a sufficiently small time step must be selected so the error incurred using a discrete time step does not dominate the spatial error. To test the convergence of the finite difference method and to determine an appropriate time step for the method, we first implement this method in Matlab, using Gaussian elimination to solve $A\,u^{n+1} = b^n$ at each discrete time step. We were able to test this method using Gaussian elimination for mesh resolutions $N = 7, 15, 31, \ldots, 255$, with discrete time steps $\Delta t = 10^{-2}, 10^{-3}, 10^{-4}$. Finer mesh resolutions and smaller time steps are not able to be solved by Matlab, as it takes too long to compute Gaussian elimination at each time step. We use this choice of mesh resolutions because $h = 1/(N+1)$. So with these choices of N, each time we increase N, h is halved. Since we used a centered finite difference approximation for the spatial derivatives, we expect our method to be second-order convergent in space.

Table 3.3.1 Convergence study in Matlab for the test problem using Gaussian elimination with $\Delta t = 10^{-2}, 10^{-3}$, and 10^{-4}.

(a) Time step $\Delta t = 10^{-2}$

N	$t = 1$ (Ratio)	$t = 2$ (Ratio)	$t = 3$ (Ratio)	$t = 4$ (Ratio)	$t = 5$ (Ratio)	$t = 6$ (Ratio)
7	1.097e-02	3.268e-02	4.704e-02	5.196e-02	5.291e-02	5.302e-02
15	2.714e-03 (4.04)	7.959e-03 (4.11)	1.146e-02 (4.10)	1.268e-02 (4.10)	1.292e-02 (4.10)	1.295e-02 (4.10)
31	7.073e-04 (3.84)	1.952e-03 (4.08)	2.821e-03 (4.06)	3.141e-03 (4.03)	3.210e-03 (4.03)	3.218e-03 (4.02)
63	2.090e-04 (3.38)	4.619e-04 (4.23)	6.761e-04 (4.17)	7.740e-04 (4.06)	7.994e-04 (4.02)	8.032e-04 (4.01)
127	8.465e-05 (2.47)	8.980e-05 (5.14)	1.408e-04 (4.80)	1.832e-04 (4.22)	1.980e-04 (4.04)	2.006e-04 (4.02)
255	5.356e-05 (1.58)	7.343e-06 (12.23)	7.091e-06 (19.86)	3.560e-05 (5.15)	4.765e-05 (4.15)	4.996e-05 (4.07)

(b) Time step $\Delta t = 10^{-3}$

N	$t = 1$ (Ratio)	$t = 2$ (Ratio)	$t = 3$ (Ratio)	$t = 4$ (Ratio)	$t = 5$ (Ratio)	$t = 6$ (Ratio)
7	1.092e-02	3.271e-02	4.707e-02	5.197e-02	5.291e-02	5.302e-02
15	2.674e-03 (4.08)	7.990e-03 (4.09)	1.149e-02 (4.10)	1.269e-02 (4.09)	1.292e-02 (4.09)	1.294e-02 (4.09)
31	6.682e-04 (4.00)	1.983e-03 (4.03)	2.854e-03 (4.03)	3.153e-03 (4.02)	3.211e-03 (4.02)	3.218e-03 (4.02)
63	1.700e-04 (3.93)	4.926e-04 (4.03)	7.098e-04 (4.02)	7.862e-04 (4.01)	8.016e-04 (4.01)	8.034e-04 (4.01)
127	4.571e-05 (3.72)	1.205e-04 (4.09)	1.745e-04 (4.07)	1.954e-04 (4.02)	2.001e-04 (4.01)	2.007e-04 (4.00)
255	1.464e-05 (3.12)	2.755e-05 (4.37)	4.083e-05 (4.28)	4.784e-05 (4.09)	4.985e-05 (4.02)	5.017e-05 (4.00)

(c) Time step $\Delta t = 10^{-4}$

N	$t = 1$ (Ratio)	$t = 2$ (Ratio)	$t = 3$ (Ratio)	$t = 4$ (Ratio)	$t = 5$ (Ratio)	$t = 6$ (Ratio)
7	1.092e-02	3.271e-02	4.708e-02	5.197e-02	5.292e-02	5.302e-02
15	2.670e-03 (4.09)	7.993e-03 (4.09)	1.150e-02 (4.09)	1.269e-02 (4.09)	1.292e-02 (4.09)	1.295e-02 (4.09)
31	6.643e-04 (4.02)	1.987e-03 (4.02)	2.858e-03 (4.02)	3.155e-03 (4.02)	3.212e-03 (4.02)	3.219e-03 (4.02)
63	1.661e-04 (4.00)	4.957e-04 (4.01)	7.132e-04 (4.01)	7.875e-04 (4.01)	8.018e-04 (4.01)	8.035e-04 (4.01)
127	4.185e-05 (3.97)	1.236e-04 (4.01)	1.779e-04 (4.01)	1.965e-04 (4.00)	2.004e-04 (4.00)	2.008e-04 (4.00)
255	1.078e-05 (3.88)	3.065e-05 (4.03)	4.420e-05 (4.03)	4.907e-05 (4.01)	5.007e-05 (4.00)	5.019e-05 (4.00)

Thus, each time we halve h we expect to see the errors decrease by a factor of about 4, if we choose sufficiently small Δt. We present the findings in Table 3.3.1. In each of the subtables, the first column lists the mesh resolutions N and each of the other columns lists the L^∞-norm of the error between the true solution u and the numerical solution u_h, $\|u(\cdot,\cdot,t) - u_h(\cdot,\cdot,t)\|_{L^\infty(\Omega)}$ at the corresponding mesh resolution N. At time t, the ratio of consecutive errors with respect to N is listed in parentheses next to each of these errors.

As mentioned earlier in this section, we expect the total error for this method to be on the order of $\Delta t + h^2$. We can observe from Table 3.3.1 that this is indeed the case. For the mesh resolutions $N = 63, 127$ and 255, $h^2 \approx 2.520 \times 10^{-4}$, 6.20×10^{-5}, and 1.538×10^{-5}, respectively. Observing the errors for $\Delta t = 10^{-2}, 10^{-3}$, and 10^{-4} we see that the errors at every time in Table 3.3.1 compares well to order of magnitude of the theoretically predicted error.

As we can see from these Table 3.3.1, $\Delta t = 10^{-4}$ is a sufficiently small time step to obtain the theoretically predicted convergence rate of the finite difference method. As we can see from Table 3.3.1 (c), the norms of the finite difference errors decrease by a factor of about 4 each time h is refined by a factor of 2. This confirms that the finite difference method is second-order convergent, as predicted by the numerical theory, and that $\Delta t = 10^{-4}$ is a sufficiently small time step to obtain spatial convergence for our choice of mesh resolutions in Table 3.3.1.

Next, we implement this method in Matlab using the conjugate gradient method to solve $Au^{n+1} = b^n$ at each time step. This method is implemented using the conjugate gradient method as the linear solver with several different choices of tolerances to ensure that the tolerance is tight enough to maintain second-order convergence of our method. The system matrix A is symmetric positive definite, so the conjugate gradient method is guaranteed to converge for this method if we choose an appropriate time step Δt and a sufficiently tight tolerance. This implementation uses Matlab's `pcg.m` function to implement the conjugate gradient method with tolerances $10^{-4}, 10^{-5}, \ldots, 10^{-10}$. For each tolerance choice, we use mesh resolutions $N = 7, 15, 31, \ldots, 255$ and $\Delta t = 10^{-4}$, as we confirmed that this is an appropriately small time step using Gaussian elimination. The results for tolerances $10^{-6}, 10^{-8}$, and 10^{-10} are in Table 3.3.2. In each of the subtables of Table 3.3.2, the first column lists the mesh resolutions N, and each of the other columns lists the L^∞-norm of the error between the true solution u and the numerical solution u_h, $\|u - u_h\|_{L^\infty(\Omega)}$ at the corresponding mesh resolution N. The ratio of consecutive errors with respect to N is listed in parentheses next to each of these errors. As we can see from the results, the tolerance must be tightened to 10^{-8} in order to obtain the same quality of convergence that we saw for Gaussian elimination in Table 3.3.1 (c). While further decreasing the tolerance to 10^{-10} does not improve our results for these mesh resolutions and results

Table 3.3.2 Convergence study in Matlab for the test problem using the conjugate gradient method with $\Delta t = 10^{-4}$ and $\tau_{\text{relres}}^{(\text{lin})} = 10^{-6}, 10^{-8}$, and 10^{-10}.

(a) L^∞-norm of the Errors, $\|u - u_h\|_{L^\infty(\Omega)}$ for tolerance $\tau_{\text{relres}}^{(\text{lin})} = 10^{-6}$

N	$t = 1$ (Ratio)	$t = 2$ (Ratio)	$t = 3$ (Ratio)	$t = 4$ (Ratio)	$t = 5$ (Ratio)	$t = 6$ (Ratio)
7	1.093e-02	3.272e-02	4.708e-02	5.197e-02	5.292e-02	5.300e-02
15	2.674e-03 (4.09)	7.992e-03 (4.09)	1.150e-02 (4.10)	1.269e-02 (4.10)	1.292e-02 (4.10)	1.290e-02 (4.11)
31	6.653e-04 (4.02)	1.986e-03 (4.02)	2.855e-03 (4.03)	3.153e-03 (4.03)	3.199e-03 (4.04)	3.113e-03 (4.14)
63	1.662e-04 (4.00)	4.933e-04 (4.03)	7.115e-04 (4.01)	7.850e-04 (4.02)	7.424e-04 (4.31)	5.670e-04 (5.49)
127	4.088e-05 (4.07)	1.222e-04 (4.04)	1.755e-04 (4.05)	1.857e-04 (4.23)	4.361e-05 (17.03)	1.938e-04 (2.93)
255	9.941e-06 (4.11)	2.883e-05 (4.24)	3.709e-05 (4.73)	7.921e-06 (23.44)	7.104e-04 (0.06)	8.692e-04 (0.22)

(b) L^∞-norm of the Errors, $\|u - u_h\|_{L^\infty(\Omega)}$ for tolerance $\tau_{\text{relres}}^{(\text{lin})} = 10^{-8}$

N	$t = 1$ (Ratio)	$t = 2$ (Ratio)	$t = 3$ (Ratio)	$t = 4$ (Ratio)	$t = 5$ (Ratio)	$t = 6$ (Ratio)
7	1.092e-02	3.272e-02	4.708e-02	5.197e-02	5.292e-02	5.302e-02
15	2.671e-03 (4.09)	7.993e-03 (4.09)	1.150e-02 (4.09)	1.269e-02 (4.10)	1.292e-02 (4.10)	1.295e-02 (4.10)
31	6.643e-04 (4.02)	1.987e-03 (4.02)	2.858e-03 (4.02)	3.155e-03 (4.02)	3.212e-03 (4.02)	3.219e-03 (4.02)
63	1.662e-04 (4.00)	4.957e-04 (4.01)	7.132e-04 (4.01)	7.874e-04 (4.01)	8.018e-04 (4.01)	8.034e-04 (4.01)
127	4.185e-05 (3.97)	1.236e-04 (4.01)	1.780e-04 (4.01)	1.967e-04 (4.00)	2.003e-04 (4.00)	2.008e-04 (4.00)
255	1.079e-05 (3.88)	3.065e-05 (4.03)	4.420e-05 (4.03)	4.905e-05 (4.01)	5.004e-05 (4.00)	5.014e-05 (4.00)

(c) L^∞-norm of the Errors, $\|u - u_h\|_{L^\infty(\Omega)}$ for tolerance $\tau_{\text{relres}}^{(\text{lin})} = 10^{-10}$

N	$t = 1$ (Ratio)	$t = 2$ (Ratio)	$t = 3$ (Ratio)	$t = 4$ (Ratio)	$t = 5$ (Ratio)	$t = 6$ (Ratio)
7	1.092e-02	3.272e-02	4.708e-02	5.197e-02	5.292e-02	5.302e-02
15	2.671e-03 (4.09)	7.993e-03 (4.09)	1.150e-02 (4.09)	1.269e-02 (4.10)	1.292e-02 (4.10)	1.295e-02 (4.10)
31	6.643e-04 (4.02)	1.987e-03 (4.02)	2.858e-03 (4.02)	3.155e-03 (4.02)	3.212e-03 (4.02)	3.219e-03 (4.02)
63	1.662e-04 (4.00)	4.957e-04 (4.01)	7.132e-04 (4.01)	7.875e-04 (4.01)	8.018e-04 (4.01)	8.035e-04 (4.01)
127	4.185e-05 (3.97)	1.236e-04 (4.01)	1.780e-04 (4.01)	1.967e-04 (4.00)	2.004e-04 (4.00)	2.008e-04 (4.00)
255	1.078e-05 (3.88)	3.065e-05 (4.03)	4.421e-05 (4.03)	4.907e-05 (4.01)	5.007e-05 (4.00)	5.019e-05 (4.00)

(d) Iteration Counts for tolerance $\tau_{\text{relres}}^{(\text{lin})} = 10^{-6}$

N	$t = 1$	$t = 2$	$t = 3$	$t = 4$	$t = 5$	$t = 6$
7	3	2	2	2	1	0
15	3	2	2	1	1	1
31	3	2	1	2	1	0
63	3	2	2	1	2	0
127	2	2	3	2	2	0
255	4	3	3	3	2	0

(e) Iteration Counts for tolerance $\tau_{\text{relres}}^{(\text{lin})} = 10^{-8}$

N	$t = 1$	$t = 2$	$t = 3$	$t = 4$	$t = 5$	$t = 6$
7	2	2	2	1	1	1
15	2	2	2	1	1	1
31	3	2	2	1	1	1
63	4	4	3	2	1	1
127	9	7	6	4	3	2
255	19	15	12	10	4	5

(f) Iteration Counts for tolerance $\tau_{\text{relres}}^{(\text{lin})} = 10^{-10}$

N	$t = 1$	$t = 2$	$t = 3$	$t = 4$	$t = 5$	$t = 6$
7	3	3	3	2	2	1
15	4	3	3	3	2	1
31	5	4	4	3	2	1
63	8	7	6	4	4	2
127	16	14	12	10	7	4
255	34	30	26	21	14	9

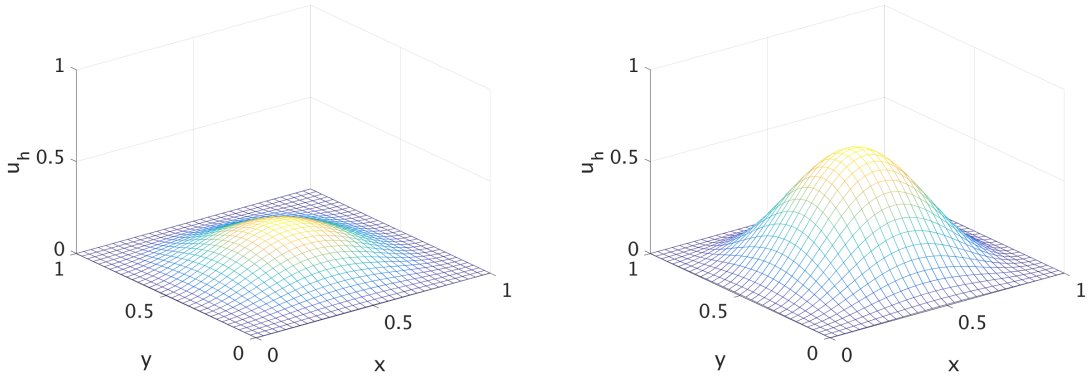
in a larger iteration count for each mesh resolution, it is necessary to decrease the tolerance to 10^{-10} in order to obtain the proper quality of convergence for finer mesh resolutions when we implement this method in C.

We can observe the effectiveness of this method by plotting the numerical solution in Figure 3.3.1 and the error $\|u - u_h\|_{L^\infty(\Omega)}$ in Figure 3.3.2 for mesh resolution $N = 32$, time step $\Delta t = 10^{-4}$, and tolerance $\tau_{\text{relres}}^{(\text{lin})} = 10^{-8}$ at times $t = 1, \ldots, 6$. As we can see from Figure 3.3.1, the shape of the numerical solution is very close to that of the true solution in Figure 2.2.1. Figure 3.3.2 confirms this and shows that the error is greatest at the center of the domain where u reaches its maximum. As we can see from Table 3.3.2 (b), the maximum error increases slightly as time increases. We also see that the error compares well to the order of magnitude $\Delta t + h^2 \approx 1.08 \times 10^{-3}$ of the theoretically predicted error.

3.4 Implementation of the Numerical Method Using C

We have an implementation of the CG method from Section 3.1. Matlab stores the system matrix in sparse form. For our implementation in C, we take advantage of the fact that the CG method only requires the matrix-vector product of the system matrix with a vector. Thus, similar to the Khuvis' implementation [10], we can compute $A\,u$ without ever assembling A. We can achieve this by providing a function that computes $v = A\,u$ without ever assembling A. This function computes v component-wise directly from the component vector u using the values of the non-zero components of A. Thus, we can implement the conjugate gradient method without storing the matrix A.

In the parallel implementation of this method, we parallelize the function Au that computes $v = Au$. In the parallel implementation of this function, each vector is split into as many blocks as there are parallel processes available, and one block

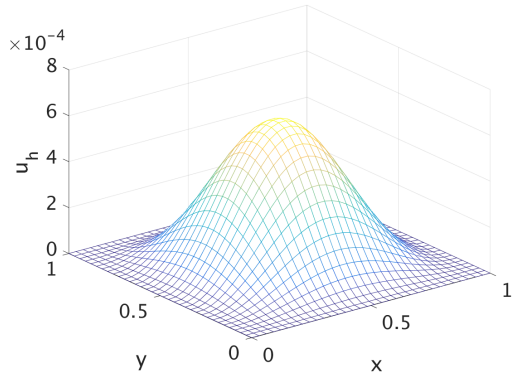


(a) $t = 1$ (b) $t = 2$

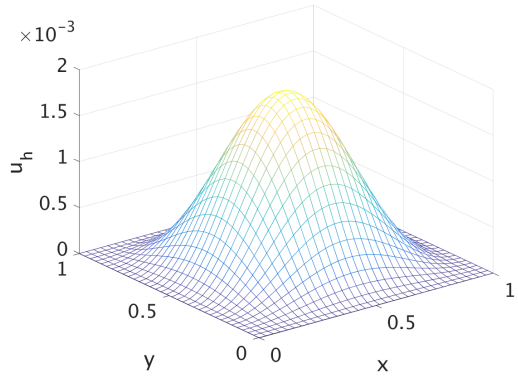
(c) $t = 3$ (d) $t = 4$

(e) $t = 5$ (f) $t = 6$

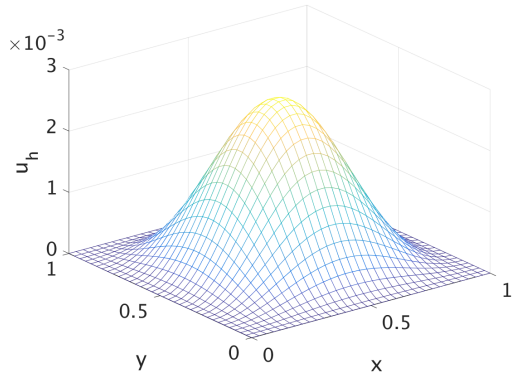
Figure 3.3.1 Mesh plots of the numerical solution for the test problem using the conjugate gradient method at times $t = 1, 2, \ldots, 6$ for mesh resolution $N = 32$ and $\Delta t = 10^{-4}$.



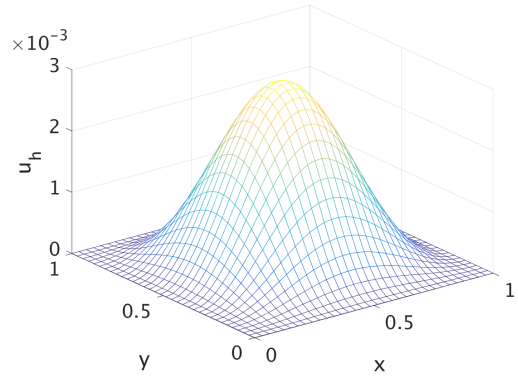
(a) $t = 1$



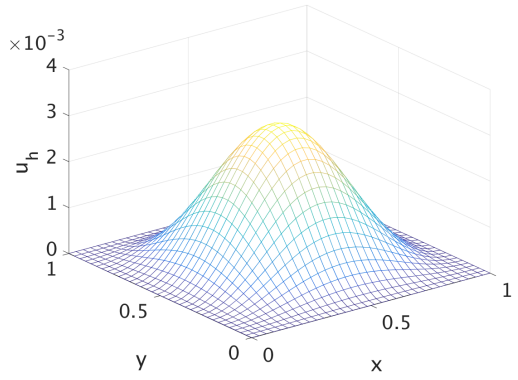
(b) $t = 2$



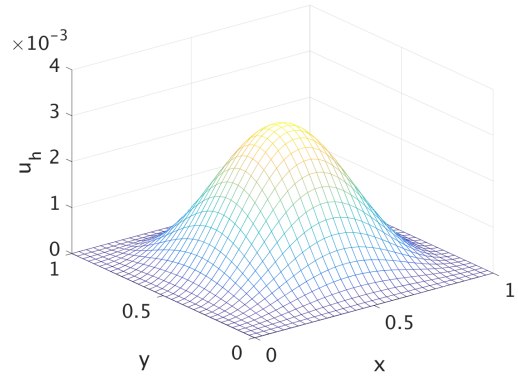
(c) $t = 3$



(d) $t = 4$



(e) $t = 5$



(f) $t = 6$

Figure 3.3.2 Mesh plots of the numerical error for the test problem using the conjugate gradient method at times $t = 1, 2, \ldots, 6$ for mesh resolution $N = 32$ and $\Delta t = 10^{-4}$. Notice the scales on the vertical axes.

Table 3.5.1 Convergence study in serial C code for the test problem using the conjugate gradient method with $\Delta t = 10^{-4}$ and $\tau_{\text{relres}}^{(\text{lin})} = 10^{-10}$.

(a) L^∞-norm of the Errors, $\|u - u_h\|_{L^\infty(\Omega)}$

N	$t = 1$ (Ratio)	$t = 2$ (Ratio)	$t = 3$ (Ratio)	$t = 4$ (Ratio)	$t = 5$ (Ratio)	$t = 6$ (Ratio)
7	1.092e-02	3.272e-02	4.708e-02	5.197e-02	5.292e-02	5.302e-02
15	2.671e-03 (4.09)	7.993e-03 (4.09)	1.150e-02 (4.09)	1.269e-02 (4.10)	1.292e-02 (4.10)	1.295e-02 (4.10)
31	6.643e-04 (4.02)	1.987e-03 (4.02)	2.858e-03 (4.02)	3.155e-03 (4.02)	3.212e-03 (4.02)	3.219e-03 (4.02)
63	1.662e-04 (4.00)	4.957e-04 (4.01)	7.132e-04 (4.01)	7.874e-04 (4.01)	8.018e-04 (4.01)	8.034e-04 (4.01)
127	4.185e-05 (3.97)	1.236e-04 (4.01)	1.780e-04 (4.01)	1.967e-04 (4.00)	2.003e-04 (4.00)	2.008e-04 (4.00)
255	1.079e-05 (3.88)	3.065e-05 (4.03)	4.420e-05 (4.03)	4.905e-05 (4.01)	5.004e-05 (4.00)	5.014e-05 (4.00)

(b) Iteration Counts

N	$t = 1$	$t = 2$	$t = 3$	$t = 4$	$t = 5$	$t = 6$
7	3	3	3	2	2	1
15	4	3	3	3	2	1
31	5	4	4	3	2	1
63	8	7	6	5	4	2
127	16	14	12	10	7	4
255	34	30	26	21	14	9

of each vector is given to each process. With this implementation, the vectors u and v never need to be fully assembled on any one process. Instead the blocks are assembled and stored on the process using them. Using this implementation, the local portions of v can be computed simultaneously on all processes. Since the matrix A contains non-zero off-diagonal elements, each process will require elements of u contained on neighboring blocks to compute v. Thus, there is a need for point-to-point communications between neighboring processes. Most of the components of v are not computed using data contained on neighboring processes. Thus, it is most efficient for each process to compute the components of the v that only require data local to the process while communicating with neighboring blocks to obtain the data necessary to compute the remaining components of v. This can be implemented using non-blocking MPI communication commands `MPI_Isend` and `MPI_Irecv` [11].

3.5 Convergence Study for the Test Problem using C

To test the implementation of the numerical method in C, we run convergence studies for our test problem. First in Tables 3.5.1 (a) and (b), we present the results for mesh resolutions $N = 7, 15, \ldots, 255$ with time step $\Delta t = 10^{-4}$ and $\tau_{\text{relres}}^{(\text{lin})} = 10^{-10}$ in

Table 3.5.2 Convergence study in parallel C code for the test problem using the conjugate gradient method with $\Delta t = 10^{-4}$ and $\tau_{\text{relres}}^{(\text{lin})} = 10^{-10}$.

(a) L^∞-norm of the Errors, $\|u - u_h\|_{L^\infty(\Omega)}$

N	$t = 1$ (Ratio)	$t = 2$ (Ratio)	$t = 3$ (Ratio)	$t = 4$ (Ratio)	$t = 5$ (Ratio)	$t = 6$ (Ratio)
32	6.217e-04	1.859e-03	2.675e-03	2.953e-03	3.006e-03	3.012e-03
64	1.609e-04 (3.86)	4.800e-04 (3.87)	6.906e-04 (3.87)	7.625e-04 (3.87)	7.764e-04 (3.87)	7.780e-04 (3.87)
128	4.120e-05 (3.91)	1.217e-04 (3.94)	1.752e-04 (3.94)	1.936e-04 (3.94)	1.972e-04 (3.94)	1.976e-04 (3.94)
256	1.070e-05 (3.85)	3.040e-05 (4.00)	4.386e-05 (3.99)	4.868e-05 (3.98)	4.968e-05 (3.97)	4.980e-05 (3.97)
512	3.008e-06 (3.55)	7.374e-06 (4.12)	1.073e-05 (4.09)	1.212e-05 (4.01)	1.245e-05 (3.99)	1.250e-05 (3.98)
1024	1.075e-06 (2.80)	1.590e-06 (4.64)	2.407e-06 (4.46)	2.934e-06 (4.13)	3.100e-06 (4.02)	3.129e-06 (3.99)
2048	5.907e-07 (1.82)	1.406e-07 (11.31)	3.217e-07 (7.48)	6.327e-07 (4.64)	7.577e-07 (4.09)	7.811e-07 (4.01)

(b) Iteration Counts

N	$t = 1$	$t = 2$	$t = 3$	$t = 4$	$t = 5$	$t = 6$
32	5	4	4	3	2	1
64	8	7	6	5	4	2
128	16	15	12	10	7	4
256	34	30	26	21	15	9
512	72	64	56	45	31	20
1024	149	134	117	95	65	42
2048	309	278	244	199	137	82

order to compare the results with those we obtained using Matlab in Tables 3.3.2 (c) and (f), respectively. For each of these results, the code was run in serial using 1 node and 1 process per node. We observe that the the errors, $\|u - u_h\|_{L^\infty(\Omega)}$, in this table are identical for at least the first several decimal places to those obtained in Table 3.5.1. We observe that for each mesh resolution N, the iterations taken for the conjugate gradient method to converge at each time are the same as those taken in Table 3.3.2, except for $N = 63$ at time $t = 4$, where the iteration count is 5 for our method in C and 4 for our method in Matlab. The round-off errors in Matlab and C result in a slightly different iterate count due to the small differences they cause in each iteration of the conjugate gradient method. In some very rare cases, these differences can lead to some difference in the quantity in the stopping test that can cause exactly one more or fewer iteration to be needed for the conjugate gradient method to converge at a fixed time step. Thus, we can conclude that the code for our test problem in Matlab and C give identical results up to round-off.

Next we present convergence studies for the mesh resolutions used in the parallel performance studies in Chapter 4. We first present a convergence study using $\Delta t =$

10^{-4} and $\tau_{\text{relres}}^{(\text{lin})} = 10^{-10}$ at mesh resolutions $N = 32, 64, \ldots, 2048$ in Table 3.5.2. This is the same tolerance and time step used in Table 3.5.1. We change our choice of mesh resolutions at this point because our parallel C code only accepts mesh resolutions that are divisible by the total number of parallel processes used. The parallel performance studies run in Chapter 4 are run using parallel processes that are increasing powers of 2, so our choice of mesh resolutions must be divisible by powers of 2 as well. This choice of N still results in h decreasing by close to a factor of 2 each time we increase our mesh resolution, so we would still expect the ratio of consecutive errors to be close to 4.

As we can see in Table 3.5.2, the time step $\Delta t = 10^{-4}$ is not small enough to obtain the proper rate of spatial convergence for the finest mesh resolutions. We observe that at mesh resolutions $N = 1024$ and $N = 2048$, the norms of the finite difference errors decrease at an erratic rate at times $t = 1, 2, 3, 4$. They should decrease by a factor of about 4 each time the mesh resolution is refined by a factor of 2. However, at $t = 1$ the norms decrease at a much slower rate, and at times $t = 2, 3, 4$, the norms decrease at a faster rate, which indicates that it may be necessary to refine Δt further so that the error with respect to time does not dominate the spatial error. In order to obtain the theoretically predicted rate of convergence, we decrease the time step to 10^{-5}. The results for this choice of Δt are presented in Table 3.5.3. We can see from the results in Table 3.5.3 (a) that this is a sufficiently small time step to obtain the theoretically predicted convergence results. We can also observe from comparing Table 3.5.2 (b) and Table 3.5.3 (b) that with a much smaller time step, the number of iterations it takes for the conjugate gradient method to converge at each time step is much smaller. The conjugate gradient method converges much more quickly at each time step because the initial guess for the conjugate gradient method is much better.

Table 3.5.3 Convergence study in parallel C code for the test problem using the conjugate gradient method with $\Delta t = 10^{-5}$ and $\tau_{\text{relres}}^{(\text{lin})} = 10^{-10}$.

(a) L^∞ Norm of the Errors

N	$t = 1$ (Ratio)	$t = 2$ (Ratio)	$t = 3$ (Ratio)	$t = 4$ (Ratio)	$t = 5$ (Ratio)	$t = 6$ (Ratio)
32	6.213e-04	1.860e-03	2.675e-03	2.953e-03	3.006e-03	3.012e-03
64	1.609e-04 (3.86)	4.803e-04 (3.87)	6.910e-04 (3.87)	7.627e-04 (3.87)	7.765e-04 (3.87)	7.780e-04 (3.87)
128	4.081e-05 (3.94)	1.220e-04 (3.94)	1.755e-04 (3.94)	1.937e-04 (3.93)	1.972e-04 (3.94)	1.976e-04 (3.94)
256	1.032e-05 (3.95)	3.071e-05 (3.97)	4.419e-05 (3.97)	4.880e-05 (3.97)	4.970e-05 (3.97)	4.980e-05 (3.97)
512	2.621e-06 (3.94)	7.682e-06 (4.00)	1.106e-05 (3.99)	1.224e-05 (3.99)	1.247e-05 (3.99)	1.250e-05 (3.98)
1024	6.887e-07 (3.81)	1.898e-06 (4.05)	2.743e-06 (4.03)	3.054e-06 (4.01)	3.120e-06 (4.00)	3.130e-06 (3.99)
2048	2.045e-07 (3.37)	4.492e-07 (4.23)	6.579e-07 (4.12)	7.537e-07 (4.05)	7.768e-07 (4.02)	7.827e-07 (4.00)

(b) Iteration Counts

N	$t = 1$	$t = 2$	$t = 3$	$t = 4$	$t = 5$	$t = 6$
32	2	2	2	1	1	1
64	2	2	2	1	1	1
128	3	2	2	2	1	1
256	5	4	3	2	2	1
512	10	8	7	5	3	3
1024	22	17	15	11	4	7
2048	47	36	31	26	10	9

CHAPTER 4

PARALLEL PERFORMANCE STUDIES

In this chapter we present the results of our parallel performance studies for the solution of the parabolic test problem on the 2013 portion of the maya cluster and on the Stampede cluster. Section 4.1 contains parallel performance studies for the test problem on CPUs using MPI only code. Section 4.2 contains parallel performance studies on CPUs using hybrid MPI+OpenMP code. Section 4.3 contains parallel performance studies on 1 hybrid CPU/Phi node in native mode using MPI only code. Section 4.4 contains parallel performance studies on 1 hybrid CPU/Phi node in native mode using hybrid MPI+OpenMP code. Section 4.5 contains parallel performance studies on 1 hybrid CPU/Phi in symmetric mode. Section 4.6 provides a summary of all relevant results for the clusters maya and Stampede.

We omit parallel performance studies for the performance of the test problem on a CPU and Intel Phi in offload mode because this mode was shown have poor performance by Khuvis for the 2D elliptic test problem, (3.1.1) [8]. The poor performance in this mode was due to the cost of communication required inside of the conjugate gradient method. All versions of the code in offload mode performed worse than the CPU only code run on 16 cores. Since the 2D parabolic test problem (1.1.1) requires the same amount of communication inside of the conjugate gradient method, we expect similarly poor performance for our test problem.

Each section except for Section 4.5 contains performance studies on both the maya and Stampede clusters. We omit studies on CPUs and Intel Phis in symmetric mode on the maya cluster because symmetric mode is currently not working on maya. For each of the other modes studied we present extensive studies done on the maya cluster and reach a conclusion about each of the different modes tested. We then

use the Stampede cluster to confirm the conclusions reached using the maya cluster and extend our results to CPUs and Intel Phis in symmetric mode. Each of the studies presented for the Stampede cluster are shown with the goal of comparing the effectiveness of each of the different modes for the test problem on a hybrid node that contains 2 CPUs and 2 Intel Phis.

The 2013 portion of the maya cluster has computes nodes that are made up of two eight-core 2.6 GHz Intel E5-2650v2 Ivy Bridge CPUs and have 64 GB memory. The two CPUs in each node are connected by two QPI (quick path interconnect) links. The nodes in this portion of the cluster are connected by a quad-data rate InfiniBand interconnect. The majority of the compute nodes on Stampede have two eight-core 2.6 GHz Intel E5-2680 Sandy Bridge CPUs with 32 GB memory. Nodes on Stampede are connected by a fourteen-data rate (FDR) InfiniBand network from Mellanox in a 2-level fat-tree topology.

Each of the hybrid nodes used for runs with the Phi on maya have 2 CPUs each connected to an an Intel Phi 5110P. On Stampede some hybrid nodes contain 2 CPUs and only 1 Intel SE10P. These nodes were used to conduct runs on Stampede using only 1 Intel Phi. Other nodes on Stampede contain 2 CPUs each connected to an Intel SE10P. These nodes were used to conduct runs on Stampede using 2 Intel Phis. Each Intel Phi has 8 GB of onboard memory. The Intel Phi on maya is capable of 4 threads on each of the 60 cores on the 5110P for a total of 240 threads. The Intel Phi on Stampede is capable of 4 threads on each of the 61 cores on the SE10P for a total of 244 threads.

Numerical experiments for the test problem (1.1.1) were conducted using progressively finer spatial meshes, $N = 128, 256, 512, 1024, 2048$. For each of these resolutions, the time step Δt is 10^{-5} and the tolerance of the conjugate gradient method is 10^{-10}. For all runs, the test problem was solved up to time $t = 10$.

The memory usage of this code is estimated by observing that our code requires 5 double precision vectors of length N^2 (and several vectors of much smaller size) to run. Since each double-precision number requires 8 bytes of storage, we can estimate the amount of memory needed in gigabytes by taking $8 \cdot 5N^2/1024^3$. The largest mesh resolution used in this study is $N = 2048$, so we do not expect the total memory used to be much more than 156 megabytes for small numbers of parallel processes. The memory usage is slightly higher than the predicted usage because of the presence of several other smaller double precision vectors, However the total memory usage is well under 1 GB for all runs, regardless of the number of parallel processes used. Therefore, memory is not a limiting factor for any of our performance studies.

4.1 Parallel Performance Studies on CPUs Using MPI Only Code

In this section we will describe the results of performance studies using on the 2013 portion of the maya cluster and on the Stampede cluster using CPUs and MPI only code. All of the results were obtained using the default Intel compiler and Intel MPI. Each run was submitted using a SLURM submission script that uses the `srun` command to initiate a job on maya and `ibrun` on Stampede. The SLURM submission script specifies the number of nodes and the number of processes per node for the job using the `nodes` and `ntasks-per-node` options respectively. For all results in the performance studies, the entire node is dedicated to the job, with the remaining cores idling while the job runs. This is achieved using the `exclusive` flag in the SLURM submission script.

Numerical experiments were conducted on the 2013 portion of maya using all possible combinations of nodes from 1 to 32 by powers of two and processes per node from 1 to 16 by powers of two, except for cases where the total number of processes equals or exceeds the mesh resolution, and the serial case for $N = 2048$ which has

an excessive runtime. Numerical experiments were conducted on Stampede for mesh resolutions $N = 128, 256$ and 512 using 1 node with 1, 2, 4, 8, and 16 processes per node, and for mesh resolutions $N = 1024$ and 2048 using 1 node and 16 processes per node. We conduct these experiments in order to compare the results to those obtained for different modes on 1 hybrid node with 2 CPUs and 2 Intel Phis on the Stampede cluster.

In Subsection 4.1.1 we will present a summary of the results on maya using CPUs only on multiple nodes up to 32 nodes. In Subsection 4.1.2 we will present a summary of our results on maya and Stampede using 1 hybrid node.

4.1.1 Summary of Performance on CPUs Only Using Multiple Nodes on maya

We will first discuss the performance studies for the 2013 portion of maya. Several different performance studies were conducted for `I_MPI_FABRICS` with 3 different settings, which is an environment variable from the Intel MPI Library that controls the particular network fabric to be used for communication [5]. The default setting on the maya 2013 cluster is `I_MPI_FABRICS=shm:ofa`. The `shm` option is set to use shared memory for intra-node communication. The option `ofa` is used to set the fabric for inter-node communication. This is the network fabric provided by the Open Fabrics Alliance (OFA). Another setting that was tested in performance studies is `I_MPI_FABRICS=shm:tmi`, where `tmi` sets network fabrics with tag matching capabilities through the Tag Matching Interface (TMI) for inter-node communication [6]. Each of these options can be set by executing the command `export I_MPI_FABRICS=shm:ofa` or `export I_MPI_FABRICS=shm:tmi` from the command line session on the head node before submitting a run. The last option that was tested was to unset `I_MPI_FABRICS` using the command `unset I_MPI_FABRICS` in the SLURM submis-

sion file.

We found that setting `I_MPI_FABRICS=shm:ofa` results in the same parallel performance as unsetting `I_MPI_FABRICS`. While there was no difference in runtimes between the two settings, we found that unsetting `I_MPI_FABRICS` will occasionally result in nodes on the maya cluster that have Intel Phi coprocessors becoming stuck in the completing stage. Therefore, when choosing between these two settings for CPU only runs, it is best to avoid unsetting `I_MPI_FABRICS` if nodes on the maya cluster with Intel Phi coprocessors may be used.

We found that setting `I_MPI_FABRICS=shm:tmi` produces the best results for CPU only runs. Using this fabric improves the runtimes for all mesh resolutions and combinations of nodes and processes per node on the maya cluster. The improvements become more pronounced as the number of nodes and processes per node increases. This option can only be used for runs on maya using nodes that do not have Intel Phi coprocessors. For all of the performance studies on the maya cluster in this section, the `--exclude=nodes` flag was added to our SLURM submission script, where `nodes` is the list of nodes on the maya 2013 cluster that have Intel Phis.

Table 4.1.1 contains the observed wall clock times for mesh resolutions N =128, 256, 512, 1024 and 2048 for CPU only runs using MPI only code on the 2013 portion of maya. It is arranged into 5 different subtables corresponding to each of the mesh resolutions the test problem was run for. In each subtable, the rows are arranged by processes per node, and the columns are arranged by amount of nodes used. Each entry in the subtable lists the wall clock time (total time to execute the code) in HH:MM:SS (hours:minutes:seconds) format. There are dashes (`--`) in the Table in cases where the number of parallel processes is smaller than slices in y-direction. In these cases, the code would not have any points in y-direction on an MPI process.

We will discuss the mesh resolution $N = 512$ in detail as an example. The first

row of this subtable presents the timing results for $N = 512$ using 1 process per node up to 32 nodes. The first column of this row contains the runtime for the serial case, as it uses only 1 process per node and 1 node. For this case, the serial runtime took 3 hours, 31 minutes, and 5 seconds. Moving over one column from 1 to 2 nodes we can see that the runtime is nearly halved. Similarly, moving down 1 row from 1 process per node to 2 processes per nodes, we can see that the runtime is nearly halved in this case as well. Moving down the first column, we can see that the runtime is nearly halved as we double the amount of processes on each node up to 16 processes per node. The runtime is nearly halved again as we go from 1 node with 16 processes per node to 2 nodes with 16 processes per node. We observe that the runtime is not halved as we increase the number of parallel processes further, but there is still a significant improvement in runtime as the the number of processes increases.

Tables 4.1.1 (a) through (d) exhibit largely analogous behavior. In each of these subtables, the runtime is approximately halved as the number of parallel processes doubles for up to a certain number of parallel processes. There is still a significant improvement in runtime in most cases as we further increase the number of parallel processes, but not the optimal halving of runtimes that occurred for smaller numbers of parallel processes. We also note that in these cases it is slightly more efficient to use as few nodes as possible. For example, we can see that in each of these cases the runtime using 1 node and 16 processes per node is slightly faster than cases using the equivalent number of parallel processes spread across more nodes. This is because for simulations with small memory usage like our test problem intra-node communication is slightly faster than communication between nodes.

The results in Table 4.1.1 (e) for $N = 2048$ exhibit different behavior than the results for other mesh resolutions. We observe that while the runtimes improve in this case as we move from 4 to 8 processes per node and from 8 to 16 processes

Table 4.1.1 Wall clock time in HH:MM:SS on the 2013 portion of the maya cluster using CPUs only with MPI only code for the parabolic test problem arranged by nodes and processes per node.

(a) Mesh resolution $N = 128$						
	1 node	2 nodes	4 nodes	8 nodes	16 nodes	32 nodes
1 process per node	00:09:29	00:05:09	00:02:55	00:01:54	00:01:31	00:01:27
2 processes per node	00:05:05	00:02:52	00:01:52	00:01:25	00:01:21	00:01:30
4 processes per node	00:02:41	00:01:42	00:01:14	00:01:09	00:01:16	—
8 processes per node	00:01:34	00:01:10	00:01:00	00:01:06	—	—
16 processes per node	00:01:01	00:00:55	00:00:58	—	—	—
(b) Mesh resolution $N = 256$						
	1 node	2 nodes	4 nodes	8 nodes	16 nodes	32 nodes
1 process per node	00:42:49	00:22:27	00:11:49	00:07:30	00:04:11	00:03:09
2 processes per node	00:22:05	00:11:40	00:08:11	00:04:16	00:02:56	00:02:44
4 processes per node	00:11:34	00:06:28	00:03:49	00:02:37	00:02:18	00:02:26
8 processes per node	00:06:18	00:03:44	00:02:32	00:02:02	00:02:04	—
16 processes per node	00:03:32	00:02:22	00:01:53	00:01:51	—	—
(c) Mesh resolution $N = 512$						
	1 node	2 nodes	4 nodes	8 nodes	16 nodes	32 nodes
1 process per node	03:31:05	01:48:01	00:55:49	00:29:59	00:17:31	00:11:12
2 processes per node	01:47:36	00:56:01	00:29:48	00:16:39	00:11:43	00:08:08
4 processes per node	00:56:24	00:29:58	00:17:06	00:10:32	00:06:48	00:05:42
8 processes per node	00:30:21	00:16:31	00:13:40	00:06:24	00:05:09	00:04:52
16 processes per node	00:16:25	00:09:43	00:08:11	00:04:46	00:04:30	—
(d) Mesh resolution $N = 1024$						
	1 node	2 nodes	4 nodes	8 nodes	16 nodes	32 nodes
1 process per node	24:59:09	10:47:08	05:09:06	02:42:32	01:30:14	00:51:21
2 processes per node	10:07:35	05:15:24	02:43:23	01:28:51	00:50:37	00:31:44
4 processes per node	05:16:24	02:48:41	01:28:01	00:49:04	00:29:59	00:22:05
8 processes per node	02:55:43	01:33:20	00:49:14	00:28:50	00:20:12	00:16:07
16 processes per node	01:32:36	00:49:47	00:29:05	00:25:56	00:17:03	00:14:16
(e) Mesh resolution $N = 2048$						
	1 node	2 nodes	4 nodes	8 nodes	16 nodes	32 nodes
1 process per node	216:39:07	105:00:57	44:56:16	17:31:51	09:10:28	04:59:14
2 processes per node	104:43:22	44:43:46	17:10:06	09:07:43	04:56:48	02:48:38
4 processes per node	58:38:03	24:50:14	09:04:05	04:57:30	02:42:31	01:45:15
8 processes per node	41:10:56	16:54:53	05:31:54	02:42:05	01:34:38	01:01:10
16 processes per node	36:19:55	13:37:19	02:58:23	01:38:52	01:02:54	00:52:14

per node, the halving of runtimes that we saw for smaller mesh resolutions does not occur. For example, as we move from 1 node and 8 processes per node to 1 node and 16 processes per node, we see that the runtime only improves from approximately 41 hours to approximately 36 hours. However, the runtime using 2 nodes and 8 processes per node is slightly less that 17 hours, and the runtime using 4 nodes and 4 processes per node is just over 9 hours. Depending on which combination of nodes and processes per node are considered, the runtime is either more than halved or nearly halved up to 128 parallel processes, and we still observe a significant reduction in runtime up to all 512 parallel processes used.

The performance for $N = 2048$ is due to memory access, which can be a limiting factor in the performance of memory-bound code. For this case, we expect a bottleneck when the 8 processes on each CPU attempt to access the memory through 4 memory channels. If we examine the performance studies run for the Poisson test problem we observe similar behavior in the results for the same mesh size [10]. That is, for the elliptic test problem, (3.1.1), for mesh resolution $N = 2048$, the runtimes are nearly halved using up to 4 processes per node. However, we observe much more limited improvements in runtimes going to 8 and 16 processes per node for the elliptic test problem as well.

We can draw several conclusions from Table 4.1.1. The first basic, but important conclusion we reach is that for all problem sizes and node choices it is most efficient to use all 16 processes per node. That is, it is never more efficient to leave cores idling on a node. The next is that as the problem sizes increase, we observe the desired halving of the code for larger numbers of parallel processes. Finally, we observe that this problem scales to multiple nodes very well on maya, with the runtime being halved or more than halved in many cases as we double the amount of nodes.

In order to observe the parallel scalability of our results for the test problem we

can arrange our results by total number of parallel processes and present speedup and efficiency data. Table 4.1.2 does this. Each of the subtables arranges the columns by an increasing number of parallel processes p and the rows by increasing mesh resolution N. In Table 4.1.2 (a), the wall clock time is listed in HH:MM:SS format. This is similar to Table 4.1.1, except that each row represents one mesh resolution, and the timing results are listed for an increasing number of parallel processes. For most values of p there are several possible combinations of nodes and processes per node that can be chosen. Due to our observation that using as many processes per node as possible performs best for $N = 128, \ldots, 1024$, we maximize processes per node used for each choice of p for each of these mesh resolutions. So for $p = 1, 2, 4, 8,$ and 16, we use only one node and p processes per node, and for larger choices of p, we use 16 processes per node. For $N = 2048$, we make use of our observation that performance is limited when the 8 processes on each CPU try to access memory through only 4 memory channels to arrange our results using only 4 processes per node.

Tables 4.1.2 (b) and (c) list speedup and efficiency numbers for our results. Speedup and efficiency tables and plots allow us to view parallel scalability in a different way. Ideal behavior for the parallel code would be for a problem of size N to be p times faster than the serial code using p processes. Denoting $T_p(N)$ as the wall clock time of the test problem with mesh resolution N using p processes, the speedup of the code from 1 to p processes is $S_p(N) = T_1(N)/T_p(N)$. The optimal value for speedup is $S_p(N) = p$. Similarly, efficiency $E_p(N) = S_p(N)/p$ characterizes how close a run with p parallel processes is to the optimal value. If the code is behaving optimally then $E_p = 1$. Parallel code that behaves optimally is strongly scalable. Figures 4.1.1 (a) and (b) visualize the numbers in Tables 4.1.2 (b) and (c) respectively.

Table 4.1.2 Wall clock time arranged by total number of parallel processes and speedup and efficiency on CPUs only using MPI only code for the parabolic test problem on the maya 2013 cluster.

(a) Wall clock time in HH:MM:SS

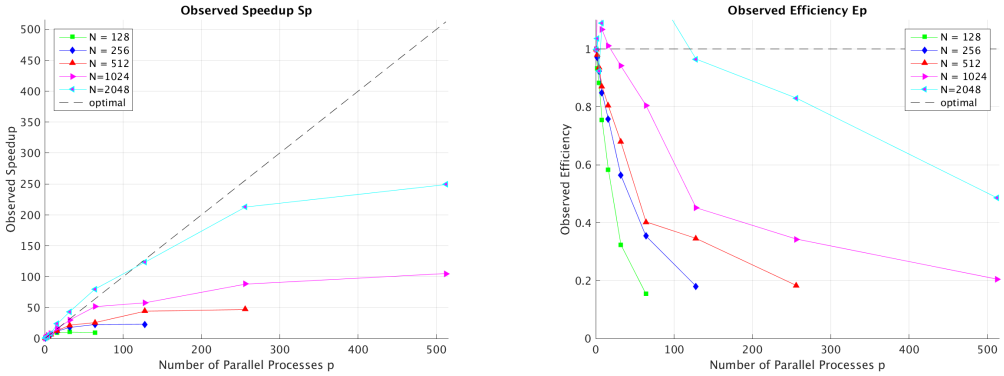
N	$p=1$	$p=2$	$p=4$	$p=8$	$p=16$	$p=32$	$p=64$	$p=128$	$p=256$	$p=512$
128	00:09:29	00:05:02	00:02:42	00:01:34	00:01:01	00:00:55	00:00:58	—	—	—
256	00:42:49	00:22:05	00:11:34	00:06:18	00:03:32	00:02:22	00:01:53	00:01:51	—	—
512	03:31:05	01:47:36	00:56:24	00:30:21	00:16:25	00:09:43	00:08:11	00:04:46	00:04:30	—
1024	24:59:09	10:07:35	05:16:24	02:55:43	01:32:36	00:49:47	00:29:05	00:25:56	00:17:03	00:14:16
2048	216:39:07	104:43:22	58:38:03	24:50:14	09:04:05	04:57:30	02:42:05	01:45:15	01:01:10	00:52:14

(b) Observed speedup S_p

N	$p=1$	$p=2$	$p=4$	$p=8$	$p=16$	$p=32$	$p=64$	$p=128$	$p=256$	$p=512$
128	1.00	1.86	3.53	6.03	9.33	10.37	9.86	—	—	—
256	1.00	1.94	3.70	6.79	12.11	18.04	22.65	23.11	—	—
512	1.00	1.96	3.74	6.95	12.86	21.72	25.80	44.25	46.91	—
1024	1.00	2.47	4.74	8.53	16.19	30.11	51.54	57.82	87.96	105.11
2048	1.00	2.07	3.70	8.72	23.89	43.70	79.99	123.51	212.52	248.87

(c) Observed Efficiency E_p

N	$p=1$	$p=2$	$p=4$	$p=8$	$p=16$	$p=32$	$p=64$	$p=128$	$p=256$	$p=512$
128	1.00	0.93	0.88	0.75	0.58	0.32	0.15	—	—	—
256	1.00	0.97	0.92	0.85	0.76	0.56	0.35	0.18	—	—
512	1.00	0.98	0.94	0.87	0.80	0.68	0.40	0.35	0.18	—
1024	1.00	1.23	1.18	1.07	1.01	0.94	0.81	0.45	0.34	0.21
2048	1.00	1.03	0.92	1.09	1.49	1.37	1.25	0.96	0.83	0.49



(a) Speedup Plot (b) Efficiency Plot

Figure 4.1.1 Speedup and efficiency plots for CPUs.

Table 4.1.2 confirms our conclusions for Table 4.1.1. We observe near optimal speedup and efficiency up to 16 processes for all mesh resolutions except $N = 128$. We also observe that as the problem size increases there is near optimal speedup for an increasing number of parallel processes. We observe optimal or better than optimal speedup for $N = 2048$ up to 128 parallel processes. There is suboptimal speedup past this point although we still observe a significant reduction in runtime as we further increase the number of parallel processes. These results show that the efficiency eventually decreases very substantially for code with small memory usage. Hence, there is an intermediate, moderate number of processes that is the optimal choice. For example, we see that for $N = 1024$ there are very good efficiency numbers up to 64 parallel processes. However, we see a severe drop-off in efficiency past this point. For $N = 2048$, where memory usage is not as small, we see optimal efficiency up to 128 processes. However, there is a drop-off in efficiency past this point.

4.1.2 Summary of Performance on CPUs only using 1 Hybrid Node on maya and Stampede

Next we present parallel performance studies on maya and Stampede using only 1 node with 2 CPUs. We present these studies in order to allow for a direct comparison for our performance studies in subsequent sections on the Intel Phi. The runs on both maya and Stampede use the default setting for `I_MPI_FABRICS` as this is the setting that is used for all runs on the Intel Phi in native mode and symmetric mode. Here we note that results were generated using multiple nodes on Stampede and the speedup and efficiency in these cases was analogous to the results on the maya 2013 cluster but we do not include results for multiple nodes because this is not relevant to our studies for the Intel Phi.

In Table 4.1.3 we present results using 1 node and 1,2,4,8, and 16 processes per node for the clusters maya and Stampede. We present results for $N = 128, 256$, and 512 as these results are sufficient to show the speedup from using more processes per node and confirm that we obtain similar results on both clusters. We will report results for $N = 1024$ and 2048 as well for 1 node and 16 processes per node to confirm that maya and Stampede give similar results, but we do not present the full results for these mesh resolutions in Table 4.1.3.

Table 4.1.3 confirms that we obtain similar results on maya and Stampede for CPU only runs using MPI only. We observe that runs are generally faster on the slightly newer nodes of the 2013 portion of the maya cluster. In each case except for $N = 128$, we observe near optimal halving up to 16 processes per node. Recall however that we did not see this halving up to 16 processes per node for $N = 2048$ in Table 4.1.1 because of the bottleneck caused by each CPU with 8 cores attempting to access memory through only 4 memory channels. Here we report the timing for $N = 1024$ using 16 processes on 1 node is 1 hour, 32 minutes, and 24 seconds on

Table 4.1.3 Wall clock times in HH:MM:SS on 1 node using CPUs only with MPI only code for the parabolic test problem on the clusters maya and Stampede.

maya					
	MPI processes per node				
N	1	2	4	8	16
128	00:09:37	00:05:00	00:02:40	00:01:35	00:01:04
256	00:42:56	00:22:08	00:11:24	00:06:21	00:03:41
512	03:36:02	01:48:06	00:56:16	00:30:21	00:16:25
Stampede					
	MPI processes per node				
N	1	2	4	8	16
128	00:11:01	00:05:44	00:03:01	00:01:36	00:00:59
256	00:49:23	00:25:34	00:13:08	00:06:59	00:03:31
512	04:03:33	02:02:01	01:04:17	00:32:44	00:17:29

maya and 1 hour, 35 minutes, and 4 seconds on Stampede. The timing for $N = 2048$ using 16 processes on 1 node is 36 hours, 58 minutes, and 11 seconds on maya and 37 hours, 9 minutes and 24 seconds on Stampede. Thus, we can conclude that we obtain similar results on maya and Stampede nodes for CPUs only using MPI only code, and that it is most efficient to use all 16 cores on 1 hybrid node.

4.2 Parallel Performance Studies on CPUs using Hybrid MPI+OpenMP Code

This section presents the results of the parallel performance studies for the solution of the test problem on the 2013 portion of the maya cluster on CPUs using hybrid MPI+OpenMP code. We modify the existing MPI only code so that it is a hybrid MPI+OpenMP code. This is done by taking the existing MPI only code for the parabolic test problem and parallelizing the function that computes v=Au, matrix-vector multiplication, and dot product functions with OpenMP. The advantage of using this hybrid code is that we can use MPI for message passing between processes and OpenMP for multithreading inside of each process. In order to run parallel performance studies using the hybrid code several options must be set in the SLURM submission script. The number of nodes, processes per nodes, and OpenMP threads per process are set using the `nodes`, `ntasks-per-node`, and `export OMP_NUM_THREADS` options respectively. In addition, we set `OMP_PROC_BIND=spread` equal to the number of OpenMP threads used for the run. Before each run on maya was submitted, `I_MPI_FABRICS` was set to `shm:tmi`. This is because just like our code in the previous section, we tried using several different settings for `I_MPI_FABRICS` and this setting ended up producing the best results.

Performance studies were run using up to 4 nodes. On each node, 1, 2, 4, 8, and 16 MPI processes were used with 16, 8, 4, 2, and 1 OpenMP threads respectively. This study was run using mesh resolutions $N = 128$, 256, 512, 1024, and 2048. For each mesh resolution, optimal performance is obtained using all 16 cores on each node, so we restrict our study to cases that use all 16 cores. Table 4.2.1 collects the results of our study. We restrict our studies to only 4 nodes as this is enough to draw conclusions about the effectiveness of using hybrid MPI+OpenMP code for CPUs on multiple nodes compared to pure MPI only code.

We observe in Table 4.2.1 that in most cases there is not an advantage to using a combination of MPI processes and OpenMP threads. In several cases, the code performs optimally using 16 MPI processes and 1 OpenMP thread. In the cases where the code does not perform optimally using 16 processes and 1 OpenMP thread, the advantage to using a combination of MPI processes and OpenMP threads is not very significant and it is not clear from this table which combination of MPI processes and OpenMP threads performs optimally. We can conclude from these results that using a combination of MPI processes and OpenMP threads does not provide a significant advantage over MPI only code using CPUs only. We also observe that for $N = 128, \ldots, 1024$, the hybrid code performs slightly better than the MPI only code in cases that use 16 processes per node. However, for $N = 2048$ the MPI only code performs slightly better when using 1 and 2 nodes.

We do not present results using hybrid code on Stampede except to note that we reach the same conclusion that we did on maya. Results on Stampede show that it is generally most effective to use 16 MPI processes with 1 OpenMP process. This is reinforced by the results obtained by Khuvis [8] for the elliptic test problem that support the conclusion there is no clear advantage to using a combination of MPI processes and OpenMP threads. We conclude that using MPI only is a reasonable choice for runs using CPUs only.

Table 4.2.1 Wall clock times using CPUs only with hybrid MPI+OpenMP code for the parabolic test problem on maya on 1, 2, and 4 nodes.

(a) Mesh Resolution $N = 128$

MPI processes	threads	1 node	2 nodes	4 nodes
1	16	00:01:33	00:01:45	00:01:53
2	8	00:01:06	00:01:15	00:01:21
4	4	00:01:02	00:01:09	00:01:12
8	2	00:01:01	00:01:06	00:01:10
16	1	00:00:56	00:01:02	00:01:08

(b) Mesh Resolution $N = 256$

MPI processes	threads	1 node	2 nodes	4 nodes
1	16	00:03:43	00:03:21	00:03:09
2	8	00:03:11	00:02:33	00:02:22
4	4	00:03:01	00:02:27	00:02:12
8	2	00:02:59	00:02:22	00:02:16
16	1	00:02:51	00:02:17	00:02:06

(c) Mesh Resolution $N = 512$

MPI processes	threads	1 node	2 nodes	4 nodes
1	16	00:13:38	00:09:39	00:07:43
2	8	00:13:21	00:08:44	00:06:39
4	4	00:13:24	00:08:43	00:06:16
8	2	00:13:02	00:08:24	00:06:11
16	1	00:13:04	00:08:29	00:06:29

(d) Mesh Resolution $N = 1024$

MPI processes	threads	1 node	2 nodes	4 nodes
1	16	01:17:06	00:46:15	00:31:09
2	8	01:15:38	00:44:54	00:28:49
4	4	01:16:40	00:44:28	00:33:58
8	2	01:17:59	00:45:18	00:27:28
16	1	01:18:30	00:43:28	00:27:09

(e) Mesh Resolution $N = 2048$

MPI processes	threads	1 node	2 nodes	4 nodes
1	16	40:55:47	15:02:43	02:39:58
2	8	40:42:31	14:27:31	02:35:06
4	4	41:10:41	15:47:27	02:35:03
8	2	37:47:35	13:58:08	02:34:48
16	1	37:43:24	13:36:51	02:49:39

4.3 Parallel Performance Studies for the Intel Phi in Native Mode Using MPI Only Code

This section describes parallel performance studies for the solution of the test problem using an Intel Phi 5110P in native mode on maya and an Intel SE10P in native mode on Stampede using MPI only code. We conduct numerical experiments on five mesh sizes, $N = 128, 256, 512, 1024$, and 2048. Parallel performance studies are run using the same MPI only code developed for parallel performance studies for our test problem on CPUs only with MPI only. To run this code on the Phi, the `-mmic` must be added to the compilation command of the Intel compiler.

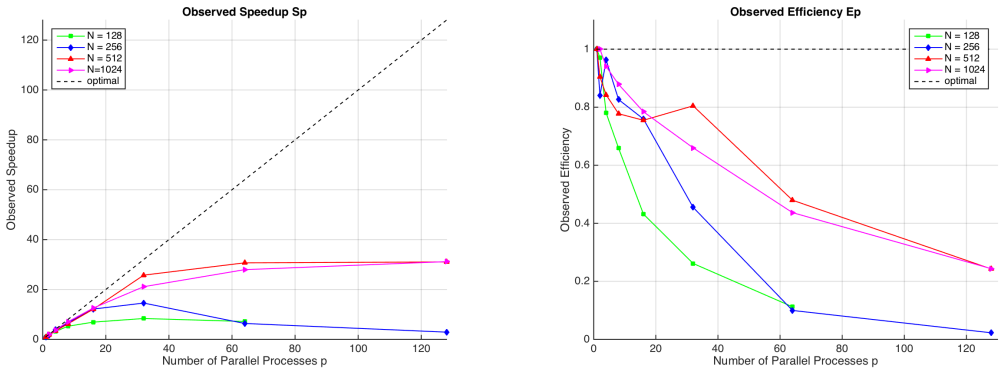
Table 4.3.1 presents the results of performance studies using one Intel Phi in native mode with MPI only code. The Intel Phi on maya is capable of 4 threads on each of the 60 cores on the 5110P for a total of 240 threads. Restricting our code to powers of 2 like we did in the previous sections, this means we can run the code using up to 128 processes per Phi. Each of the results in this table were obtained using a compute node on the maya 2013 cluster with 2 Intel Phis.

We report results for $N = 128, \ldots, 1024$ in Table 4.3.1 with the number of MPI processes per Phi increasing by powers of 2 from 1 MPI process per Phi to 128 MPI processes per Phi. We note that for $N = 1024$ we do not present results using 1 or 2 MPI processes per Phi because in each of these cases we expect the runs would take several days to complete and would not provide any additional insight into performance on the Phi in native mode. We report the observed wall clock times in Table 4.3.1 (a), speedup in Table 4.3.1 (b) and efficiency in Table 4.3.1 (c). We can draw several conclusions from Table 4.3.1. The first observation we make is that performance on the Phi scales much better for larger problem sizes. In particular, we observe for $N = 128$ and 256 that the performance degrades severely as we move past 16 MPI processes and that runtimes actually increase using 64 or 128 processes. We

Table 4.3.1 Wall clock times in HH:MM:SS and speedup and efficiency for 1 Phi in native mode with MPI only code for the parabolic test problem on maya. ET denotes excessive time requirement.

	MPI processes per Phi							
N	1	2	4	8	16	32	64	128
Observed wall clock time in HH:MM:SS								
128	00:48:45	00:25:02	00:15:33	00:09:13	00:07:02	00:05:48	00:06:46	—
256	04:09:10	02:28:14	01:04:44	00:37:41	00:20:30	00:17:07	00:39:04	01:25:54
512	23:16:03	12:50:14	06:53:49	03:44:32	01:55:34	00:54:13	00:45:29	00:44:58
1024	ET	79:35:19	42:22:18	22:37:32	12:40:43	07:31:58	05:41:45	05:06:21
Observed Speedup S_p								
128	1.00	1.94	3.12	5.27	6.91	8.37	7.18	—
256	1.00	1.68	3.85	6.61	12.15	14.56	6.38	2.90
512	1.00	1.81	3.37	6.22	12.08	25.75	30.69	31.05
1024	ET	2.00	3.76	7.03	12.55	21.12	27.95	31.17
Observed Efficiency E_p								
128	1.00	0.97	0.78	0.66	0.43	0.26	0.11	—
256	1.00	0.84	0.96	0.83	0.76	0.46	0.10	0.02
512	1.00	0.91	0.84	0.78	0.75	0.80	0.48	0.24
1024	ET	1.00	0.94	0.88	0.78	0.66	0.44	0.24

see that as we move to $N = 512$ and 1024, runtimes decrease up to 128 processes, although the decrease in runtime is minimal when we increase the number of processes to 64 and 128. Comparing the results in Table 4.3.1 to the results in Table 4.1.3, we observe that for these problem sizes, the Intel Phi in native mode performs far worse than 1 node using 2 CPUs. For example, the best runtime we observe for $N = 1024$ on the Phi is 5 hours, 6 minutes, and 21 seconds. Using all 16 cores on 1 node for $N = 1024$ results in a runtime of only 1 hour, 33 minutes, and 19 seconds. We observe this poor performance for all mesh resolutions in Table 4.3.1, so we can conclude from



(a) Speedup Plot (b) Efficiency Plot

Figure 4.3.1 Speedup and efficiency plots for the Intel Phi in native mode using MPI only code on maya.

these results that the Intel Phi in native mode using MPI only code does not perform well for these problem sizes.

Table 4.3.2 Best observed wall clock times in HH:MM:SS for 1 Phi in native mode with MPI only code for the parabolic test problem on maya and Stampede.

	Stampede		maya	
N	MPI processes	1 Phi	MPI processes	1 Phi
128	32	00:06:39	32	00:05:48
256	32	00:14:58	32	00:17:07
512	32	00:56:20	128	00:44:58
1024	64	06:03:45	128	05:06:21
2048	128	34:16:21	128	31:28:13

Table 4.3.2 collects the best observed wall clock times for each mesh resolution on the maya and Stampede Phis and the corresponding MPI processes per Phi for these times. We observe similar behavior and runtime on the maya and Stampede Phis for $N = 128$, 256, and 512. In these cases the Phis on Stampede perform slightly better because they have a slightly higher clock rate. However, the Intel Phis on

maya outperform the Intel Phis on Stampede when using 64 and 128 processes per Phi. We observe for $N = 1024$, there was an increase in runtime on the Stampede Phis when using 128 MPI processes, while on maya, it is more effective to use 128 MPI processes and the runtime is significantly faster than on Stampede. Similarly, the Phis on maya outperform the Phis on Stampede for $N = 2048$. The cases that use 64 and 128 processes are the only cases where multiple processes are allocated to the same core. In these cases, the slightly newer Phis on maya are able to handle this issue better.

The Intel Phi in native mode performs much better for $N = 2048$. We observe in Table 4.3.2 that it is more efficient to use 128 MPI processes per Phi on both maya and Stampede. Comparing the results in Table 4.3.2 to those in Section 4.1 Section 4.2, we also observe that for $N = 2048$ the Intel Phi in native mode outperforms 1 node using both CPUs. We can conclude from these results that the Intel Phi is more effective for larger problem sizes for the test problem.

4.4 Parallel Performance Studies for the Intel Phi in Native Mode Using Hybrid MPI+OpenMP Code

This section describes parallel performance studies for the solution of the test problem using an Intel Phi 5110P in native mode on maya with modified hybrid MPI+OpenMP code. We use the same hybrid code used in Section 4.2 for the test problem using hybrid code on CPUs. In this section we restrict our studies to parallel processes increasing by powers of 2 up to 128 total parallel processes.

Table 4.4.1 collects the results of performance studies for $N = 128$, 256, and 512 on the maya cluster using 1, 2, 4, 8, 16, 32, 64, and 128 MPI processes and OpenMP threads up to 128 total parallel processes. For each mesh resolution in Table 4.4.1 different combinations of MPI processes and OpenMP threads produce optimal results. We cannot conclude from Table 4.4.1 that it is beneficial to use any particular combination of MPI processes and OpenMP threads. Comparing Table 4.4.1 with Table 4.3.1, we observe that there is no clear advantage to using hybrid MPI+OpenMP code instead of MPI Only code for these problem sizes. The best observed runtimes for each mesh resolutions are very similar, and in some cases the MPI Only code slightly outperforms the hybrid code.

Table 4.4.1 Wall Clock Time in HH:MM:SS on 1 Phi in native mode using hybrid MPI+OpenMP code on for the parabolic test problem for $N = 128, 256$ and 512 on the maya cluster.

(a) $N = 128$

	MPI Processes per node							
	1	2	4	8	16	32	64	128
$t = 1$	00:38:18	00:22:32	00:16:02	00:09:47	00:28:55	00:15:02	01:02:21	NA
$t = 2$	00:21:32	00:16:08	00:24:49	00:10:23	00:17:05	00:17:14	NA	NA
$t = 4$	00:14:42	00:12:21	00:23:43	00:09:38	00:18:32	NA	NA	NA
$t = 8$	00:10:25	00:16:35	00:20:44	00:10:43	NA	NA	NA	NA
$t = 16$	00:09:37	00:18:58	00:24:50	NA	NA	NA	NA	NA
$t = 32$	00:10:32	00:12:45	NA	NA	NA	NA	NA	NA
$t = 64$	00:13:09	NA	NA	NA	NA	NA	NA	NA
$t = 128$	NA	NA	NA	NA	NA	NA	NA	NA

(b) $N = 256$

	MPI Processes per node							
	1	2	4	8	16	32	64	128
$t = 1$	03:45:23	02:04:13	01:09:00	00:31:24	00:31:07	00:16:58	00:18:23	00:24:19
$t = 2$	01:56:08	01:03:37	00:35:19	00:23:21	00:20:40	00:20:44	00:40:56	NA
$t = 4$	00:59:43	00:45:38	00:25:37	00:20:11	00:21:15	00:20:49	NA	NA
$t = 8$	01:09:27	00:35:32	00:19:29	00:18:48	00:23:26	NA	NA	NA
$t = 16$	00:36:56	00:32:11	00:19:48	00:21:17	NA	NA	NA	NA
$t = 32$	00:35:51	00:34:38	00:20:04	NA	NA	NA	NA	NA
$t = 64$	00:38:20	00:33:37	NA	NA	NA	NA	NA	NA
$t = 128$	00:39:39	NA	NA	NA	NA	NA	NA	NA

(c) $N = 512$

	MPI Processes per node							
	1	2	4	8	16	32	64	128
$t = 1$	20:50:55	12:23:38	06:25:41	03:48:54	01:48:43	01:10:04	00:59:18	01:03:45
$t = 2$	11:25:04	06:22:41	03:51:09	01:59:16	01:04:16	00:59:51	01:00:57	NA
$t = 4$	06:00:16	03:46:02	01:47:27	01:13:57	00:52:14	00:59:03	NA	NA
$t = 8$	03:35:14	02:02:24	01:11:13	00:59:06	00:55:10	NA	NA	NA
$t = 16$	01:38:21	01:19:34	01:01:29	01:08:20	NA	NA	NA	NA
$t = 32$	01:02:27	01:00:40	01:00:58	NA	NA	NA	NA	NA
$t = 64$	01:04:53	01:07:13	NA	NA	NA	NA	NA	NA
$t = 128$	01:09:08	NA	NA	NA	NA	NA	NA	NA

For mesh resolutions $N = 1024$ and 2048, we restrict our studies to cases that use 128 threads over all MPI processes. We present the results for $N = 1024$ using 128 total threads on maya and Stampede and 2048 on maya in Table 4.4.2. We observe that it is more efficient to use a combination of MPI processes and OpenMP threads for $N = 2048$ and more efficient to use only 1 MPI process and 128 OpenMP threads for $N = 1024$. Similar to the results in Table 4.4.1, it is not clear from Table 4.4.2 which combination of MPI processes and OpenMP threads generally produces optimal results.

Table 4.4.2 Parallel performance studies for the Intel Phi in native mode on maya and Stampede using hybrid MPI+OpenMP code.

		maya		Stampede
MPI processes	threads	$N = 1024$	$N = 2048$	$N = 1024$
1	128	04:21:18	33:00:02	03:40:11
2	64	05:00:42	33:55:02	04:37:42
4	32	04:55:27	33:01:44	04:30:03
8	16	05:13:55	39:31:24	04:30:11
16	8	04:40:37	33:25:30	04:04:45
32	4	04:46:34	31:40:09	04:24:39
64	2	05:16:48	31:28:06	05:41:08
128	1	26:51:58	33:12:12	07:33:09

Table 4.4.2 displays one of the differences between performance on maya and on Stampede. On maya, we do not see a clear advantage to using a combination of MPI processes and OpenMP threads. For $N = 2048$, MPI only code outperforms the hybrid code for all combinations of MPI processes and OpenMP threads in Table 4.4.2. However, on Stampede we observe a clear advantage to using a combination of MPI processes and OpenMP threads. On Stampede for $N = 1024$ we observed an increase in runtime going from 64 to 128 processes using MPI only code, and the best observed runtime was slightly more than 6 hours. Using a combination of MPI processes and

OpenMP threads significantly reduces the runtime except for in the case that uses 128 MPI processes. Similarly, for $N = 2048$ the best runtime using MPI only code is 34 hours, 16 minutes, and 21 seconds. However, using 16 MPI processes and 8 OpenMP threads per process results in a runtime of only 29 hours, 14 minutes, and 16 seconds. Thus, we can conclude that on Stampede there is a clear advantage to using a combination of MPI processes and OpenMP threads for the Intel Phi in native mode.

Up to this point in our performance studies, we have limited the number of cores used on the Intel Phi to powers of 2. However, this does not utilize all 240 cores on the Intel Phi. We modify our existing code to test performance using all 240 cores available on the Intel Phi on maya. Using hybrid MPI+OpenMP code on all 240 threads of the Intel Phi results in much better performance than using MPI only code for all 240 processes. However, the runtime using 240 total threads over all MPI processes still results in a significant increase in runtime compared to using only 128 total threads. The best observed runtime on maya using 240 total threads is 34 hours, 57 minutes, and 57 seconds. The best observed runtime on Stampede using 240 total threads in 36 hours, 29 minutes, and 32 seconds. On both maya and Stampede these runtimes are significantly longer than those observed using only 128 total threads.

We see that the runtimes in some cases using 240 threads is comparable to the runtimes using 128 threads. However, we still find that it is more efficient to not use all available threads. This may be due to the added communication between processes slowing down performance because the problem size is not large enough. In this case the benefit of using more cores does not outweigh the added cost of communication from having more processes. There may be a slightly better number of threads than 128, but it does not appear to be 240.

4.5 Parallel Performance Studies for the Test Problem on CPUs and Intel Phis in Symmetric Mode

This section describes the performance of the test problem on CPUs and Intel Phis in symmetric mode. In this mode, MPI ranks are run on the CPU and Intel Phi allowing for communication between the two architectures. All results in this section were generated using the Stampede cluster because we are currently unable to complete runs on the maya cluster in symmetric mode. This is because the maya cluster uses an Infiniband from QLogic and Stampede uses an Infiniband from Mellanox. Issues with the Infiniband from QLogic result in an inability to communicate over the network on maya.

The key to good performance in symmetric mode is proper load balancing between the Intel Phis and CPUs. Since symmetric mode requires communication between CPUs and Phis, larger problems observe better speedup, so we will restrict our studies in this section to the mesh resolution $N = 2048$. Recall from Sections 4.3 that we observe better performance on the Intel Phi than on 2 CPUs. Optimal performance on the CPUs was observed using 16 MPI processes with MPI only code. On Stampede, optimal performance on 1 Intel Phi was observed using a combination of MPI processes and OpenMP threads.

Table 4.5.1 summarizes the results using CPUs only, 1 and 2 Phis in native mode, and 1 hybrid node in symmetric mode. Recall that optimal performance on Stampede was obtained using a total of 128 threads across all MPI processes. For the runs using 1 and 2 Phis in native mode, 16 MPI processes with 8 threads per process were used on each Phi. For runs in symmetric mode, MPI only code is used on the CPUs. Several different combinations of processes and threads were tested on the Phi. Runs were submitted using both hybrid MPI+OpenMP code on the Phi as well as MPI only code. We did not observe good results using hybrid code on the Phi

in symmetric mode. Using 16 MPI processes with 8 OpenMP threads on each Phi resulted in an increase in runtime compared to using 2 Phis in native mode. Runs with this configuration on the Phis did not complete in the 24 hour time limit placed on nodes on Stampede containing 2 Intel Phis. This is likely caused by the workload not being properly balanced between the CPUs and the Phis. We observe that using MPI processes only on both the Phi and the CPUs better balances the workload and leads to a further reduction in runtime in symmetric mode. We observe that using 128 total MPI processes with 16 MPI processes on the CPUs and 56 MPI processes on each Phi results in a runtime of only 16 hours, 31 minutes, and 15 seconds. Increasing the number of processes on each Phi results in an increase in runtime. This points further to the importance of balancing the workloads between different architectures in symmetric mode. We can conclude from these results that using 2 CPUs and 2 Phi produces optimal performance on a node with 2 CPUs and 2 Phis.

Table 4.5.1 Symmetric mode for the parabolic test problem on 1 node on Stampede.

CPUs	Phis	Mode	CPU prog. model	Phi prog. model	Runtime
2	0	CPU only	MPI Only	NA	37:09:24
0	1	Native	NA	MPI+OpenMP	29:14:16
0	2	Native	NA	MPI+OpenMP	17:16:39
2	2	Symmetric	MPI only	MPI Only	16:31:15

4.6 Summary of Performance on One Hybrid Node

This section provides a summary of all relevant results on 1 hybrid node. Performance studies were run for the test problem using mesh resolutions $N = 128$, 256, 512, 1024, and 2048. For mesh resolutions $N = 128$, 256, 512, and 1024 we observe good performance using CPUs only. In each case it is most efficient to use all 16 cores on the node. These mesh resolutions scale well on the CPUs, as we observe that the runtime is nearly halved each time we double the number of cores used up all 16 CPU cores on the node. We found that there was no clear advantage to using a combination of MPI processes and OpenMP threads on the CPUs as opposed to using MPI only code. We observed this behavior on both maya and Stampede.

We found that the Intel Phi in native mode does not perform well for $N = 128$, 256, 512, 1024. For each of these mesh resolutions we ran studies on the Intel Phi using both MPI only code and hybrid MPI+OpenMP code. Similar to our results for CPUs, we found that there was no clear advantage to using a combination of MPI processes and OpenMP threads. For each of these mesh resolutions we found the Intel Phi performed significantly worse than the CPUs on both maya and Stampede. Performance on maya and Stampede was similar for the Intel Phi in native mode. However, we found that the slightly newer Intel Phis on maya outperformed those on Stampede for runs that used 64 or 128 cores. None of these mesh resolutions were found to perform well in symmetric mode as the Intel Phis perform poorly for these runs, resulting in slower runs than those conducted using CPUs only.

Table 4.6.1 summarizes our results for $N = 2048$ on maya and Stampede. We report the best runtimes for each different mode on the maya and Stampede clusters. For $N = 2048$, we reach the conclusion that there is no clear advantage to using hybrid MPI+OpenMP code instead of MPI only code on both the CPUs. On Stampede, there is a clear advantage to using hybrid MPI+OpenMP code on the Intel Phis in

native mode. We found that the Intel Phi in native mode performs much better for $N = 2048$. The Intel Phi outperforms 1 node using 2 CPUs only. We also found that symmetric mode performs much better for this problem size. We observe using both CPUs and Intel Phis on 1 hybrid node on the Stampede cluster more than halves the runtime using only 1 Phi or both CPUs. Our results confirm that the Intel Phi and symmetric mode perform better for larger problem sizes.

Table 4.6.1 Summary of parabolic test problem results for $N = 2048$ on 1 hybrid node.

Stampede					
CPUs	Phis	Mode	CPU prog. model	Phi prog. model	Runtime
2	0	CPU only	MPI only	NA	37:58:11
0	1	Native	NA	MPI only	34:16:21
0	1	Native	NA	MPI+OpenMP	29:14:16
0	2	Native	NA	MPI+OpenMP	17:16:39
2	2	Symmetric	MPI only	MPI only	16:31:15
maya					
CPUs	Phis	Mode	CPU prog. model	Phi prog. model	Runtime
2	0	CPU only	MPI only	NA	36:19:55
2	0	CPU only	MPI+OpenMP	NA	37:43:24
0	1	Native	NA	MPI only	31:28:13
0	1	Native	NA	MPI+OpenMP	31:28:06

CHAPTER 5

CONCLUSIONS

In this chapter, we draw conclusions for the performance of one or more hybrid nodes with two CPUs and two Phis for a linear parabolic test problem whose structure is representative of kernels of real-world application codes. This test problem is the linear heat equation with homogeneous Dirichlet boundary conditions, (1.1.1). This test problem lies in complexity between the linear stationary elliptic test problem and the CICR problem. We use this test problem to evaluate performance on CPUs, the Intel Phis, and CPUs in combination with Phis. We evaluate parallel performance studies for the 2013 portion of the maya cluster in the UMBC High Performance Computing Facility and on TACC's Stampede system.

We find that for the test problem MPI only code is sufficient for good performance on the CPUs and the Intel Phis on maya. On Stampede, MPI only code is sufficient for good performance on the CPUs and a combination of MPI processes and OpenMP threads results in optimal performance on the Intel Phi. The CPUs perform well for all mesh resolutions studied. We found that the performance on CPUs scales well to an intermediate number of nodes and this scalability improves as the mesh resolutions are increased. We can conclude that the Intel Phis perform significantly worse than the CPUs for cases with low memory usage, as we observe that runtimes on the CPUs were significantly faster up to the mesh resolution $N = 1024$. However, for $N = 2048$ the Intel Phi performs excellently compared to the CPUs. We observe that 1 Intel Phi in native mode outperforms two CPUs for $N = 2048$ and that symmetric mode halves the runtimes using 2 CPUs or 1 Intel Phi. We can conclude that code with a high degree of parallelism is required to take advantage of the many cores of the Phi and to achieve better performance than on the CPUs.

One of the fundamental differences between the linear parabolic test problem studies in this paper and the linear stationary elliptic test problem studied by Khuvis is the choice of mesh resolutions studied [8]. Each of these test problems has different limitations that determined the choice of mesh resolutions studied. For the elliptic test problem, the primary limitation is memory usage during a run. The Intel Phi is limited to problem sizes that use 8 GB or less of memory during a run, and the CPUs are limited to problem sizes that use 64 GB or less of memory. For the elliptic test problem, this limits the choice of mesh resolutions to 8192 when running experiments using the Phis and 32768 when running experiments using only the CPUs. For the elliptic test problem, runtime is not a limiting factor. For the parabolic test problem studied here, memory usage is not a limiting factor. The largest mesh resolution studied is 2048. Memory usage for this problem size is less than 1 GB, so the problem can easily fit on both the Phis and the CPUs. The limiting factor for our test problem is runtime. Mesh resolutions larger than 2048 could not be studied due to the excessive runtimes associated with them.

The differences between the elliptic and parabolic test problems that were studied leads to some differences in the results between the two test problems. These differences in results can be attributed to the limiting factors for each of the test problems. For the elliptic test problem, each of the large mesh resolutions studied exhibited typical performance for memory-bound code. Using 8 or 16 processes results in less than optimal speedup as the 8 processes on each CPU attempt to access memory through only 4 memory channels. For our parabolic test problem only the largest mesh resolution studied, 2048, exhibited this characteristic of memory-bound code. For each of the smaller mesh resolutions studied, memory access did not limit performance up to 16 processes per node, however the problem only scales well up to an intermediate number of nodes because of the cost of communication between an

increasing number of parallel processes. For each of the smaller mesh resolutions with low memory usage it is not possible to compare performance between the elliptic test problem and the parabolic test problem. This is because the runtime in these cases is trivial for the elliptic test problem, with even the serial runs for each of these mesh resolutions lasting only a couple of seconds. Even for $N = 2048$, we are only able to compare performance for a limited number of parallel processes, as past this point the runtimes for this problem size become trivial for the elliptic test problem as well.

For $N = 2048$ the results for the elliptic and parabolic test problem show similar speedup and efficiency up to 4 nodes. Further increasing the nodes used for this problem size results in trivial runtimes for the elliptic test problem, which does not allow for a direct comparison of the results. We observe similar behavior using the CPUs, with the runtime being halved up to 4 processes per node. Past this point the runtime is still reduced using 8 and 16 processes per node, but the improvement in performance is much less significant than the optimal halving that occurs using a smaller number of processes. For both test problems it is more efficient to use 1 Phi in native mode than to use the CPUs. On Stampede, we see a further improvement in performance using both Phis in native mode and both the CPUs and the Phis in symmetric mode.

We can conclude that while there are many similarities between the elliptic test problem and the linear parabolic test problem (1.1.1), the different limitations between the two problems lead to different performance on 1 hybrid node. A direct comparison between equivalent mesh resolutions for each test problem is not possible as the mesh resolutions studied for the parabolic test problem produce mostly trivial runtimes for the elliptic test problem.

For the parabolic test problem, for code with low memory usage such as the mesh resolutions $N = 128, \ldots, 1024$, the code seems to be compute-bound rather

than memory-bound for the mesh resolutions considered, and we can conclude that it is most efficient to use MPI only code on the CPUs, and performance scales well up to 16 processes per node. Performance for these problem sizes on the Phis is significantly worse as there is not a sufficiently high degree of parallelism in the code to take advantage of the many cores of the Phis. For the case of $N = 2048$ we observe optimal performance on the CPUs is still obtained using all 16 cores of the CPUs, however we do not observe an optimal halving of the runtime using 8 and 16 processes per node, as this mesh resolution makes the code memory-bound rather than compute-bound. Further, we find that for this problem size 1 Phi in native mode outperforms the CPUs and that on Stampede using a combination of MPI processes and OpenMP threads is key to optimal performance on the Phis. However, we find equivalent performance for the Intel Phi on maya using MPI only code and hybrid MPI+OpenMP code. Performance is further improved on Stampede using both CPUs and Phis in symmetric mode. Thus, we can conclude that for problems with higher memory usage and a high degree of parallelism the Phis outperform CPUs and optimal performance on one hybrid node is obtained using both CPUs and Phis in symmetric mode.

BIBLIOGRAPHY

[1] Kendall E. Atkinson. *An Introduction to Numerical Analysis*. John Wiley & Sons, second edition, 1989.

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

[3] Lawrence C. Evans. *Partial Differential Equations*, vol. 19 of *Graduate Studies in Mathematics*. American Mathematical Society, second edition, 2010.

[4] HPCF. How to run programs on the intel phi on maya introduction. `http://hpcf.umbc.edu/resources-for-hpcf-users/how-to-run-programs-on-the-intel-phi-on-maya`, accessed on April 10, 2016.

[5] Intel. I_MPI_FABRICS. `https://software.intel.com/en-us/node/535585`, accessed on March 25, 2016.

[6] Intel. Selecting fabrics. `https://software.intel.com/en-us/node/535532`, accessed on March 25, 2016.

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

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

[9] Samuel Khuvis and Matthias K. Gobbert. Parallel performance studies for an elliptic test problem on maya 2013. Technical Report HPCF–2014–6, UMBC High Performance Computing Facility, University of Maryland, Baltimore County, 2014.

[10] Samuel Khuvis and Matthias K. Gobbert. Parallel performance studies for an elliptic test problem on the cluster maya. Technical Report HPCF–2015–6, UMBC High Performance Computing Facility, University of Maryland, Baltimore County, 2015.

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

[12] John Towns, Timothy Cockerill, Maytal Dahan, Ian Foster, Kelly Gaither, Andrew Grimshaw, Victor Hazlewood, Scott Lathrop, Dave Lifka, Gregory D. Peterson, Ralph Roskies, J. Ray Scott, and Nancy Wilkens-Diehr. XSEDE: Accelerating Scientific Discovery. *Computing in Science and Engineering*, vol. 16, no. 5, pp. 62–74, 2014.

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