PDE SOLVERS FOR HYBRID CPU-GPU ARCHITECTURES

Michael Malahe

A dissertation submitted to the faculty at the University of North Carolina at Chapel Hill in partial fulfillment of the requirements for the degree of Doctor of Philosophy in the Department of Mathematics in the College of Arts and Sciences.

Chapel Hill
2016

Approved by:

Sorin Mitran

Boyce Griffith

Jingfang Huang

Julia Kimbell

Richard McLaughlin

© 2016
Michael Malahe
ALL RIGHTS RESERVED

ABSTRACT

Michael Malahe: PDE Solvers for Hybrid CPU-GPU Architectures
(Under the direction of Sorin Mitran)

Many problems of scientific and industrial interest are investigated through numerically solving partial differential equations (PDEs). For some of these problems, the scope of the investigation is limited by the costs of computational resources. A new approach to reducing these costs is the use of coprocessors, such as graphics processing units (GPUs) and Many Integrated Core (MIC) cards, which can execute floating point operations at a higher rate than a central processing unit (CPU) of the same cost. This is achieved through the use of a large number of processors in a single device, each with very limited dedicated memory per thread. Codes for a number of continuum methods, such as boundary element methods (BEM), finite element methods (FEM) and finite difference methods (FDM) have already been implemented on coprocessor architectures. These methods were designed before the adoption of coprocessor architectures, so implementing them efficiently with reduced thread-level memory can be challenging. There are other methods that do operate efficiently with limited thread-level memory, such as Monte Carlo methods (MCM) and lattice Boltzmann methods (LBM) for kinetic formulations of PDEs, but they are not competitive on CPUs and generally have poorer convergence than the continuum methods.

In this work, we introduce a class of methods in which the parallelism of kinetic formulations on GPUs is combined with the better convergence of continuum methods on CPUs. We first extend an existing Feynman-Kac formulation for determining the principal eigenpair of an elliptic operator to create a version that can retrieve arbitrarily many eigenpairs. This new method is implemented for multiple GPUs, and combined with a standard deflation preconditioner on multiple CPUs to create a hybrid concurrent method with superior convergence to that of the deflation preconditioner alone. The hybrid method exhibits good parallelism, with an efficiency of 80% on a problem with 300 million unknowns, run on a configuration of 324 CPU cores and 54 GPUs.

To my mother, Lesley

ACKNOWLEDGEMENTS

This work would not have been possible without the support of my advisor, Sorin Mitran. I am thankful for his dedicated mentoring, not just in my PhD work, but also in planning for the career beyond. His guidance has been invaluable.

I am grateful to the members of my PhD committee, Jingfang Huang, Boyce Griffith, Richard McLaughlin and Julia Kimbell, for their advice and input at various stages of the work. I also appreciate the contributions of Cass Miller in this area.

I am grateful to the National Research Foundation of South Africa (NRF) and the United States Department of State for the award of an NRF/Fulbright Scholarship for Doctoral Studies Abroad for the first three years of my work. This work would never have begun without their support. For funding grants that supported the continuation of my work, I would like to thank the National Science Foundation and the National Institutes of Health.

I would also like to thank UNC Research Computing, in particular, Mark Reed, Jenny Williams and Sandeep Sarangi, for sharing their expertise, and for help with the logistics of running large scale computations.

I am also grateful to those whose contributions were not directly to my work, but nevertheless helped with its completion. Whether the contributions were to my wellbeing, sanity, or clarity of thought, they were greatly appreciated.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF ABBREVIATIONS

AMG	algebraic multigrid. 30
ARPACK	the Arnoldi Package. 54
BEM	boundary element methods. iii
CG	conjugate gradient. 28
CPU	central processing unit. iii
CUDA	Compute Unified Device Architecture. 8
DG	Discontinuous Galerkin. 15, 16
EM	electromagnetic. 15, 16
FDTD	finite-difference time-domain. 16
FEM	finite element methods. iii, 15, 16
FLOP	floating-point operation. 9
FLOPS	floating-point operations per second. 9
FVM	finite volume method. 16
GMRES	generalized minimum residual. 17
GPUs	graphics processing units. iii
HDGMRES	hybrid deflated GMRES. 61, 62
HEGMRES	hybrid enriched GMRES. 61, 62
KNF	Knights Ferry. 16
LBM	lattice Boltzmann methods. iii

MCM Monte Carlo methods. iii

MGCG the multigrid conjugate gradient method. 28

MIC Many Integrated Core. iii

PCIe Peripheral Component Interconnect Express. 5

PDEs partial differential equations. iii, 28

PFMG parallel full multigrid. 28, 30

RAM random access memory. 9

SMG semicoarsening multigrid. 28

SMP streaming multiprocessor. 9

UNC The University of North Carolina. 55

WENO Weighted Essentially Non-oscillatory. 16

CHAPTER 1

Introduction

1.1 Numerical Approaches to PDEs

The numerical solution of PDEs has relevance to a large number of scientific and industrial problems. These problems include those of fluid flow, heat conduction, and elasticity, and the PDEs often arise as the continuum limit of microscale interactions. In the case of heat conduction for example, the Laplace operator captures the spatially-averaged effects of elastic collisions at the molecular level.

This perspective, that interactions at the microscale underly all of these PDEs, is not typically leveraged by traditional solvers, such as Finite Element Methods (FEM) [7] and Boundary Element Methods (BEM) [8]. Instead, these methods take the continuum limit as the starting point, from which the domain is discretized into elements representative of a physical volume or surface.

These discretization approaches come with two main computational expenses, which are the discretization of the domain through generating volume and surface meshes, and the solution of linear systems. Solving the linear systems is often only feasible with the addition of mechanisms to accelerate convergence. Some simple examples are linear element preconditioning [9] and sparse approximate inverses [10], which are a part of a large catalogue of acceleration techniques.

The need for complex mesh generation and the solution of linear systems is avoided entirely by kinetic methods such as kinetic Monte Carlo methods [11] and Lattice-Boltzmann Methods (LBM) [12], both of which leverage descriptions of the interactions at the microscale. The major drawback is that these methods are only feasible for relatively low accuracy computations, because their computational expense grows rapidly with increased accuracy. This makes their use in isolation worthy of consideration for only a small class of problems.

There is the potential however, for these methods to be used in conjunction with standard continuum methods. A number of combined schemes are proposed here, in which a kinetic method is used to approximate the structure of the differential operator in the PDE, which is then used to accelerate a continuum solver. The kinetic method is implemented on a GPU, and the continuum method is implemented on a CPU, so that each method is running on the architecture where it is the most efficient. These proposed schemes with results and analysis are presented in Chapter 5 and Chapter 6.

The description of the schemes is preceded by a description of relevant applications in Section 1.2, a survey of existing processor architectures in Chapter 2, and a short review of the current state of continuum methods in Chapter 3 and linear solvers in Chapter 4.

1.2 Applications

This work deals primarily with PDEs of the form

$$\frac{\partial u(\boldsymbol{x},t)}{\partial t} = \mathcal{L}(u) + f(\boldsymbol{x},t), \tag{1.1}$$

where

$$\mathcal{L}(u) = \sum_{i=1}^{d} \alpha_i(\boldsymbol{x}) \frac{\partial u(\boldsymbol{x},t)}{\partial x_i} + \frac{1}{2} \sum_{i,j=1}^{d} \beta_{ij}(\boldsymbol{x}) \frac{\partial^2 u(\boldsymbol{x},t)}{\partial x_i \partial x_j} + q(\boldsymbol{x},t)u(\boldsymbol{x},t), \tag{1.2}$$

and d is the dimension of the problem, which is defined for $\boldsymbol{x} \in \Omega \subset \mathbb{R}^d$ and $t \in [0,\infty)$, with appropriate initial and boundary conditions. There are many possible applications of PDEs of this form, but here we choose one that is representative of the case where the linear solvers for discretizations of the PDE suffer from poor convergence. The application is described briefly in the following subsection.

1.2.1 Inhomogeneous diffusion

For inhomogeneous diffusion, the problem is given by

$$-\nabla \cdot (K(\boldsymbol{x})\nabla u(\boldsymbol{x})) = f(\boldsymbol{x}), \tag{1.3}$$

where $K(\boldsymbol{x})$ is the diffusivity, u is the scalar of interest, and $f(\boldsymbol{x})$ is the sum of the source terms. In the notation Equation (1.1), this corresponds to

$$\begin{aligned} \beta_{ij}(\boldsymbol{x}) &= 2\delta_{ij}K(\boldsymbol{x}) \\ \alpha_i &= \frac{\partial K}{\partial x_i} \\ q(\boldsymbol{x}) &= 0. \end{aligned}$$

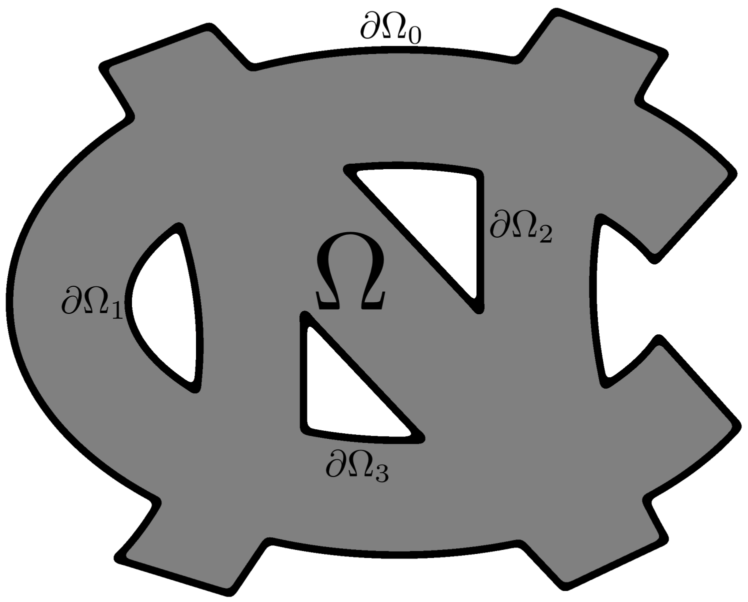


Figure 1.1: Example domain for Equation (1.3). Dirichlet conditions in the interior boundaries and Neumann conditions on the outer boundary would be notated $\Gamma_D = \partial\Omega_1 \cup \partial\Omega_2 \cup \partial\Omega_2$, and $\Gamma_N = \partial\Omega_0$.

The boundary conditions are

$$\begin{aligned} u(\boldsymbol{x}) = g(\boldsymbol{x}) &\quad \text{on} \quad \Gamma_D, \\ K(\boldsymbol{x})\nabla u(\boldsymbol{x}) \cdot \boldsymbol{n} = h(\boldsymbol{x}) &\quad \text{on} \quad \Gamma_N, \\ \Gamma_D \cup \Gamma_N &\quad = \quad \partial\Omega, \\ \Gamma_D \cap \Gamma_N &\quad = \quad \varnothing, \end{aligned}$$

where $g(\boldsymbol{x})$ is the Dirichlet condition on Γ_D, and $h(\boldsymbol{x})$ is the Neumann condition on Γ_N. The domain labeling is illustrated in Figure 1.1.

The application to time-steady porous media flow is chosen as a simple example. First, we take the Navier Stokes equations in the zero Reynolds number limit to recover the Stokes equation:

$$\nabla P = \mu\nabla^2\boldsymbol{u} + \boldsymbol{f}. \tag{1.4}$$

For steady flow in a porous medium, Darcy's Law gives

$$\boldsymbol{q}(\boldsymbol{x}) = \frac{-k(\boldsymbol{x})}{\mu}\nabla P(\boldsymbol{x}), \tag{1.5}$$

where $\boldsymbol{q}(\boldsymbol{x})$ is the flux and $k(\boldsymbol{x})$ is the permeability. Imposing incompressibility gives

$$\begin{aligned} \nabla \cdot \boldsymbol{q} &= 0, \\ \Rightarrow \nabla \cdot \left(\frac{-k(\boldsymbol{x})}{\mu}\nabla P(\boldsymbol{x}) \right) &= 0. \end{aligned}$$

Thus the problem has been reduced to solving

$$\nabla \cdot (K(\boldsymbol{x})\nabla P(\boldsymbol{x})) = 0, \tag{1.6}$$

where $K(\boldsymbol{x}) = \frac{k(\boldsymbol{x})}{\mu}$. The problems of interest are those with large and rapid changes in the diffusivity throughout the domain. One such problem is groundwater flow.

1.3 Processor architectures

The effectiveness of numerical methods for solving these elliptic PDEs depends both on the mathematical properties of the method and on the properties of the processor architectures on which the method is implemented. The most popular architecture currently available is on-die CPUs [13], for which all of the cores reside on the same continuous piece of silicon. This architecture has low latency and high memory throughput, which makes method design for a single machine quite straightforward. In high performance systems, the need for additional processing power is met by creating a network that connects multiple machines that are each based around this architecture. If more than one of these computational nodes is needed for a given application, the overhead from network communication can be a dominant factor in the overall processing time [14]. The difficulty then is designing algorithms that cut down as much as possible on this communication.

The extent of the network communication that is required can be reduced by improving the processing throughput of a single node, which is currently being done with two kinds of alternative architectures. The first is the Many Integrated Core (MIC) architecture [15, 16], developed by Intel, and the second is general-purpose Graphics Processing Units (GPUs) [17], chiefly supported by NVIDIA in the realm of scientific computing through their Tesla, Fermi and Kepler microarchitectures.

Both of these architectures can be regarded as coprocessors, which deliver their high throughput by using a discrete card with a large number of cores that is is connected to the motherboard through a Peripheral Component Interconnect Express (PCIe). The historical peak processing throughput for the CPU, MIC and GPU architectures is plotted in Figure 1.2.

It should be noted that reaching these peaks for the coprocessor architectures in non-trivial applications is still challenging [18, 19]. While there are many cores, each one has very limited dedicated memory, which poses a challenge for algorithm design. Despite this, a number of scientific computing applications and libraries have been ported to one or both of the coprocessor architectures. For both architectures the focus has mostly been on dense (MIC [20, 21], GPU [22, 23]) and sparse (MIC [24], GPU [25]) linear algebra libraries. The effectiveness of this approach is supported by the high performance computing community [26, 27, 28], and the current 1st, 2nd, 6th and 7th most powerful supercomputers in the world all rely heavily on coprocessors [4]. The top 10 supercomputers in June 2014 are listed with their coprocessor adoption in Table 1.1.

While these coprocessors are powerful and delay the need for inter-node communication, they increase the cost of intra-node communication by requiring communication between the CPU and the coprocessor through the PCIe bus. This again places constraints on algorithm design, where now the communication between the CPU and the coprocessor has to be minimized. It is this constraint that influences the computational approach put forward in this proposal.

Rank	Machine Name	Coprocessors	Throughput (TFlop/s)	Power (kW)
1	Tianhe-2	MIC	33862	17808
2	Titan	GPU	17590	8209
3	Sequoia	-	17173	7890
4	RIKEN K Computer	-	10510	12660
5	Mira	-	8586	3945
6	Piz Daint	GPU	6271	2325
7	Stampede	MIC	5168	4510
8	JUQUEEN	-	5008	2310
9	Vulcan	-	4293	1972
10	US Govt. Unnamed	-	3143	-

Table 1.1: Architecture adoption among the world's top 10 supercomputers in June 2014 (as measured by realized maximum FLOP rate) [4].

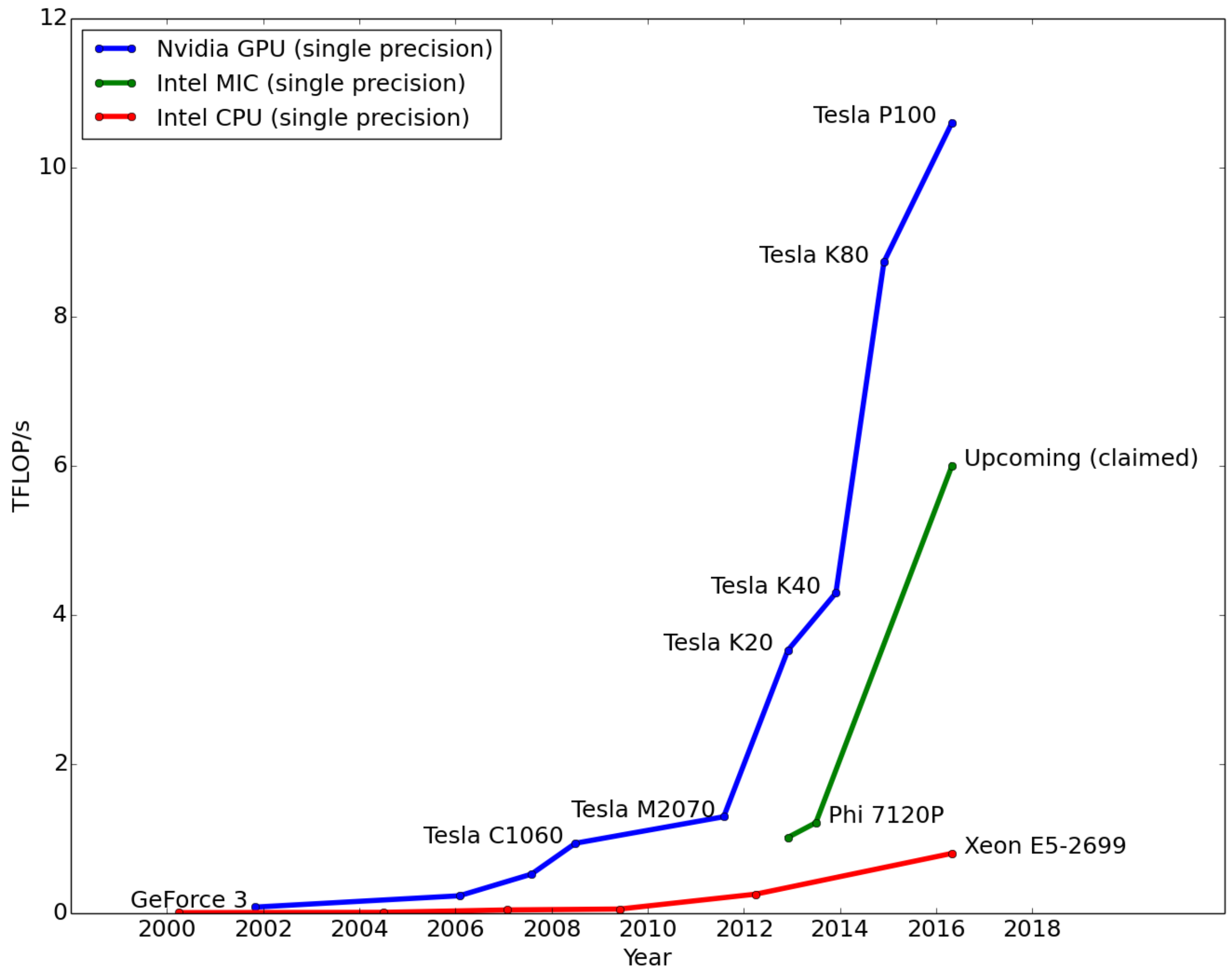


Figure 1.2: History of the three dominant processor architectures in terms of peak TFLOP/s throughput with perfect parallelization [1, 2, 3]

.

CHAPTER 2

Computer Architectures

In this chapter we describe the current computer architectures available for scientific computing. In particular, we highlight the features of each architecture that strongly influence the design of numerical methods for PDEs. The existing work on implementing continuum methods on these architectures is presented in Section 3.2.

2.1 CPUs

For describing CPUs, we take the Intel Xeon family of processors to be representative of modern multi-core CPUs. In the last decade, the process has gone from 65 nm dual-core [29] to 45 nm 8-core [30] to 22 nm 15-core [5] processors. In their current (2016) state, multi-core CPUs are distinguished from GPUs and MICs by their large cache and large instruction set. The memory transfer rates for a representative multi-core CPU are given in Table 2.1. In that table, and in the tables for GPUs and MICs, "throughput up" is the cost of transferring memory between the given tier and the one above it.

Memory type	Size	Throughput up
RAM	48+ GB	-
L3 cache	37.5 MB	1866 - 2667 MT/s (= 119 - 171) GB/s

Table 2.1: Memory transfer rates for a 22 nm, 15-core Intel Xeon processor [5]

2.2 GPUs

For describing GPUs, we take the Tesla architecture [17] to be representative of modern GPUs.

2.2.1 Memory

For GPUs, there are four relevant tiers of memory, each with significant transaction costs between them. They are random access memory (RAM), then the global, shared, and thread memory of the GPU. The sizes and throughput rates for the a representative GPU are given in Table 2.2. "Throughput up" is the cost of transferring memory between the given tier and the one above it. The key limitations are the 16 kB of shared memory and the 1 kB of thread memory.

Memory	Hardware	Software	Number	Size	Throughput up (GB/s)
Global	Card	Device	1	12GB	8
Shared	SMP	Block	15 per device	16kB	216
L1 cache (per core)	Core	Thread	192 per block	1 kB	216

Table 2.2: Memory hierarchy of a Tesla K40

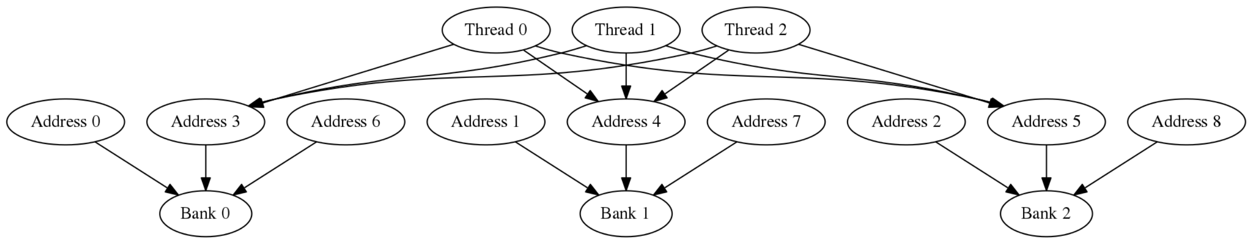


Figure 2.1: Threads accessing the same memory location in each bank. These broadcast operations do not incur any serialization.

Optimal memory transfers Due to the very small per-thread and shared memory, high throughput applications require a large number of memory transfers. These transfers are frequently the main bottleneck in these applications, so doing them optimally is essential. One area in which these transfers can be processed at drastically different speeds, despite delivering the same amount of data, is in the accessing of memory from the GPU's shared memory banks. If two threads attempt to access two *different* locations within the same memory bank, a *bank conflict* occurs, and the accesses must be serialized. Ideally, the threads are either all accessing the same memory location in each bank (Figure 2.1), or are accessing memory from different banks (Figure 2.2). Following the intuition from CPU programming, one might assign each thread a contiguous chunk of memory addresses to read from, but this leads to the worst case bank conflict (Figure 2.3).

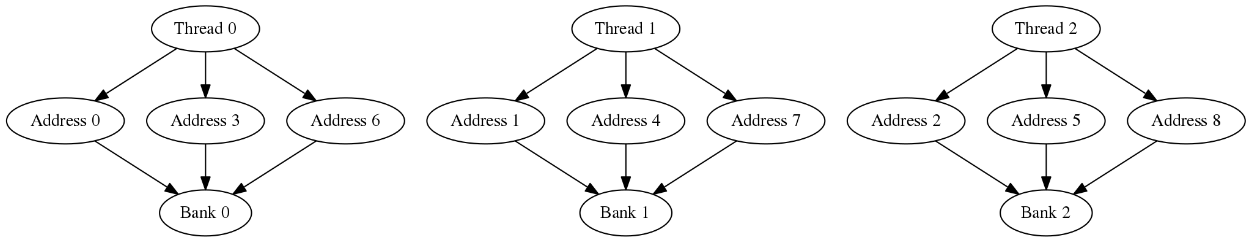


Figure 2.2: Threads accessing memory from completely separate banks. This is the best case scenario.

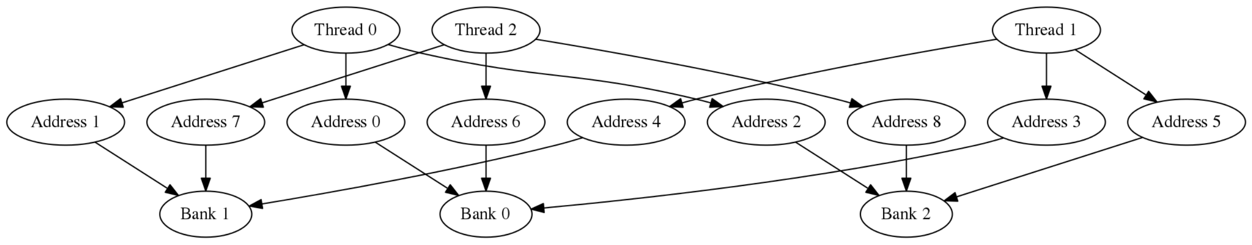


Figure 2.3: Threads accessing memory from contiguous memory addresses. This is the worst case scenario.

2.3 MICs and other architectures

The first attempt by Intel at creating a GPU-like device was the Larrabee architecture [31], which was designed to be capable of carrying out actual graphics processing, but was abandoned in 2010. Its successor was the MIC architecture, which was designed to function purely as a coprocessor [32, 33]. As with GPUs, MICs rely on a large number of smaller cores for their throughput, but have an instruction set very similar to a multi-core CPU, allowing for easier porting of existing CPU codes to the architecture. For MICs, there are 5 relevant tiers of memory, which are detailed in Table 2.3. While the memory hierarchy is similar to that of a GPU, and the memory sizes for global and on-core memory are similar, MICs are distinguished from GPUs by having significantly more shared memory, at the cost of reduced transfer rates between this shared memory and the other tiers of memory.

Memory type	Size	Throughput up (GB/s)
RAM	48+ GB	-
Main memory	2 GB	8
L2 Cache	8 MB	96
L1 Cache	1 MB	73
CPU registers (per core)	2 kB	20

Table 2.3: Memory transfer rates for a 32-core Intel Xeon Phi coprocessor[6]

CHAPTER 3

Continuum Methods

3.1 Finite Difference Methods

In this work, we use a finite difference discretization in all of the methods. For the inhomogeneous diffusion problem in 2D, the standard second-order discretization is

$$\begin{aligned}\nabla_h \cdot (K_h \nabla_h u_{ij}) &= \frac{1}{h^2}\left(K_{i+\frac{1}{2}}(u_{i+1}-u_i) - K_{i-\frac{1}{2}}(u_i - u_{i-1})\right) \\ &+ \frac{1}{h^2}\left(K_{j+\frac{1}{2}}(u_{j+1}-u_j) - K_{j-\frac{1}{2}}(u_j - u_{j-1})\right),\end{aligned}$$

where the indices are i and j if omitted. The grid dimensions are $i = 0, \ldots, m-1$ and $j = 1, \ldots, n-1$. We'll use the index renumbering $I_{ij} = i + mj$. Again, the indices are i and j if omitted. The full system is then

$$\begin{aligned}&-\nabla_h \cdot (K_h \nabla_h u_{ij}) = f_{ij} \\ \Rightarrow\quad & K_{i+\frac{1}{2}}(u_{i+1}-u_i) - K_{i-\frac{1}{2}}(u_i - u_{i-1}) + K_{j+\frac{1}{2}}(u_{j+1}-u_j) - K_{j-\frac{1}{2}}(u_j - u_{j-1}) = -h^2 f_{ij}\end{aligned}$$

The matrix A represents the action of $-\nabla_h \cdot (K_h \nabla_h \cdot)$, so the non-zero entries are

$$\begin{aligned}A_{I,I} &= \frac{1}{h^2}\left(K_{i-\frac{1}{2}} + K_{i+\frac{1}{2}} + K_{j-\frac{1}{2}} + K_{j+\frac{1}{2}}\right), \\ A_{I,I_{i-1}} &= -\frac{1}{h^2}K_{i-\frac{1}{2}}, \\ A_{I,I_{i+1}} &= -\frac{1}{h^2}K_{i+\frac{1}{2}}, \\ A_{I,I_{j-1}} &= -\frac{1}{h^2}K_{j-\frac{1}{2}}, \\ A_{I,I_{j+1}} &= -\frac{1}{h^2}K_{j+\frac{1}{2}}.\end{aligned}$$

The system matrix now has the following structure:

$$A_h = \frac{1}{h^2}\begin{pmatrix} K_{m-\frac{1}{2},0}+K_{\frac{1}{2},0}+K_{0,n-\frac{1}{2}}+K_{0,\frac{1}{2}} & -K_{\frac{1}{2},0} & 0\dots0 & -K_{0,\frac{1}{2}} & 0\dots0 & 0 & 0\dots0 & 0 \\ \ddots & \ddots & \ddots & \ddots & \ddots & \ddots & \ddots & \ddots \\ \ddots & \ddots & \ddots & \ddots & \ddots & \ddots & \ddots & \ddots \\ \ddots & \ddots & \ddots & \ddots & \ddots & \ddots & \ddots & \ddots \end{pmatrix}, \tag{3.1}$$

where Dirichlet boundary conditions are applied in the standard way, by modifying the right-hand side. For example, in the top left corner we have

$$\begin{aligned} & -\left(K_{i+\frac{1}{2}}(u_{i+1}-u_i) - K_{i-\frac{1}{2}}u_i + K_{j+\frac{1}{2}}(u_{j+1}-u_j) - K_{j-\frac{1}{2}}u_j\right) \\ = \quad & h^2 f_{ij} + K_{i-\frac{1}{2}}g_{i-1,0} + K_{j-\frac{1}{2}}g_{0,j-1}, \end{aligned}$$

and in the bottom right corner we have

$$\begin{aligned} & -\left(-K_{i+\frac{1}{2}}u_i - K_{i-\frac{1}{2}}(u_i-u_{i-1}) - K_{j+\frac{1}{2}} - u_j - K_{j-\frac{1}{2}}(u_j-u_{j-1})\right) \\ = \quad & h^2 f_{ij} + K_{i+\frac{1}{2}}g_{i+1,0} + K_{j+\frac{1}{2}}g_{0,j+1} \end{aligned}$$

3.2 Parallelization

In this section we take a broad review of the parallelization approaches for continuum methods on CPUs, GPUs and MICs. The performance for GPU and MIC codes for continuum methods at various computational scales is summarized in Table 3.1.

3.2.1 CPU

Finite element methods on CPUs are typically parallelized through domain decomposition, the parallelization of linear solvers [34, 35, 36], and adaptive mesh refinement [37]. These approaches have been successfully implemented on petascale computers [38, 39], including applications to reactor hydrodynamics [40] and mantle convection [41].

3.2.2 GPU

The development of finite element methods on GPUs has mostly been focused on the parallelization of the numerical integration, matrix assembly and linear solution steps [42, 43, 44, 25]. These approaches have been used in a number of applications, including fluid flow [45, 46] and soft-tissue surgery simulation [47, 48].

Ports of entire FEM codes to GPUs have primarily focused on nodal Discontinuous Galerkin (DG) methods [49]. These methods have good performance on GPUs due to a large component of the DG operator being evaluated at the element level, with less computation devoted to the coupling between elements [49]. For a single GPU on an electromagnetic (EM) scattering problem, the efficiency of DG was found to increase with higher order elements, which was attributed to a greater fraction of the work being done at the element level. This was also found in multiple GPU implementations [50]. In an EM wave propagation problem using 6th-order elements, a parallel efficiency of 90.5% was achieved with 8 GPUs on 2 nodes [50].

At the low terascale, finite difference Weighted Essentially Non-oscillatory (WENO) methods and finite volume method (FVM) have been implemented on GPUs for rectangular grids, using standard domain decomposition techniques with ghost cells. For the Favre-averaged Navier-Stokes equations in 3D, a finite difference implementation using 4 GPUs on a single node achieved a parallel efficiency of 67.3% [51]. For the shallow water equations, a finite volume implementation using 4 GPUs on a single node achieved a parallel efficiency of 86.0% [52].

At the medium terascale, applications of finite-difference time-domain (FDTD) and FEM are dominated by seismic wave propagation problems [53, 54, 55, 56]. These methods achieve near-perfect weak scaling, but have limited strong scaling (see Table 3.1).

3.2.3 MIC

The development of FEM code for the MIC architecture is currently fairly limited, presumably due to the novelty of the architecture. Preliminary work has used a MIC port [27] of the libMesh library [57]. The speedups were capped at around 14 times for 2D and 20 times for 3D using a 30-core Knights Ferry (KNF) card. The authors pointed to FEM steps that were difficult or inefficient to parallelize, such as matrix and mesh allocations and the application of constraints.

Application	Method	Hardware	# of unknowns	Throughput	Weak/strong scaling
EM	DG	1×GTX 280	1.9×10^6	200 GFlops	N/A [49]
EM	DG	8×Tesla T10	1.04×10^8	1.64 TFlops	-/90.5% [50]
Fluids	WENO	4×Tesla C1070	1.20×10^8	-	-/67.3% [51]
Fluids	FVM	4×Tesla C2050	4.7×10^8	-	86.0%/- [52]
Seismic	FEM	128×Tesla X2090	5.2×10^9	10.0 TFlops	98.0%/50% [56]
Seismic	FDTD	270×Tesla M2090	1.68×10^{10}	30.5 TFlops	- [55]
Seismic	FDTD	952×Tesla X2090	5.88×10^{11}	101.4 TFlops	- [55]
Seismic	FEM	896×Tesla X2090	5.2×10^9	135.0 TFlops	-/- [56]
Radiation	KBA	928×Tesla K20m	-	35.0 TFlops	64%/58% [58]
Fluids	FEM	4096×Tesla K20m	-	-	46.4%/99% [59]

Table 3.1: Summary of performance on GPUs and MICs at various scales for implementations of continuum method formulations of PDEs. Entries of "-" correspond to quantities that were not explicitly reported by the authors.

CHAPTER 4

Linear Solvers

In this chapter, we review a number of classical linear solvers, and their adaptations to high performance computing environments. We focus primarily on Krylov subspace methods, and in particular we examine the GMRES method and its derivatives. The enriched subspace method (Section 4.1.2) and the deflation preconditioned method (Section 4.3) are given in full detail, as they form the basis of the hybrid methods presented in this work (Chapter 6). We also examine a number of high performance multigrid preconditioners for Krylov methods (Section 4.4), which are used as state-of-the-art comparisons for these hybrid methods.

4.1 GMRES

GMRES is an iterative method for solving nonhermitian linear systems, introduced by Saad *et. al* [60]. Suppose we have a subspace $\mathcal{V}_n = \langle \boldsymbol{v}_1, \boldsymbol{v}_2, \ldots, \boldsymbol{v}_n \rangle$. If we want to minimize the residual on $\mathcal{V}_n$, we can write this as

$$\min_{\boldsymbol{y} \in \mathbb{R}^n} \|\boldsymbol{b} - AV_n\boldsymbol{y}\|,$$

where V_n is the matrix whose columns are the vectors $\boldsymbol{v}_i$, and the approximate solution is $\boldsymbol{x}_n = V_n\boldsymbol{y}$. A simple approach to this would be to use a QR factorization of AV_n, from which the problem can be represented as

$$\min_{\boldsymbol{y} \in \mathbb{R}^n} \|\boldsymbol{b} - QR\boldsymbol{y}\|,$$

which reduces to solving

$$R\boldsymbol{x}_n = Q^*\boldsymbol{b},$$

which is a single matrix-vector product and a back-substitution. GMRES is just least squares minimization on the Krylov subspace $\mathcal{K}_n = \langle \boldsymbol{b}, A\boldsymbol{b}, \ldots, A^{n-1}\boldsymbol{b} \rangle$. Explicitly constructing the QR factorization is unstable, however. Instead, the approach is to find a matrix Q_n whose columns span

$\mathcal{K}_n$, from which the problem becomes minimizing

$$\min_{\boldsymbol{y}\in\mathbb{R}^n} \|\boldsymbol{b} - AQ_n\boldsymbol{y}\|,$$

where $\boldsymbol{x}_n = Q_n\boldsymbol{y}$.

This can be done with the Arnoldi iteration, which reduces a nonhermitian matrix to Hessenberg form by orthogonal similarity transformations. Write this as decomposition $H_n = Q_n^* AQ_n$. This gives $AQ_n = Q_{n+1}\tilde{H}_n$, where

$$\tilde{H}_n = \begin{pmatrix} h_{11} & \cdots & \cdots & h_{1n} \\ h_{21} & h_{22} & & \vdots \\ 0 & \ddots & \ddots & \vdots \\ 0 & \ddots & h_{n,n-1} & h_{\text{nn}} \\ 0 & 0 & 0 & h_{n+1,n} \end{pmatrix}.$$

The method is given in Algorithm 1. The result is that the columns of Q_n do in fact span $\mathcal{K}_n$. Additionally, the recurrence allows the minimization problem to be reduced to

$$\begin{aligned} & \min_{\boldsymbol{y}\in\mathbb{R}^n} \|\boldsymbol{b} - Q_{n+1}\tilde{H}_n\boldsymbol{y}\| \\ = & \min_{\boldsymbol{y}\in\mathbb{R}^n} \|Q_{n+1}^*\boldsymbol{b} - \tilde{H}_n\boldsymbol{y}\| \\ = & \min_{\boldsymbol{y}\in\mathbb{R}^n} \|\tilde{H}_n\boldsymbol{y} - \|\boldsymbol{b}\|\boldsymbol{e}_1\|. \end{aligned}$$

The minimization step can be done via a QR decomposition of $\tilde{H}_n$ [60]. Combining the Arnoldi iteration and the minimization gives GMRES, shown in Algorithm 2. If we denote the jth approximate solution obtained by GMRES by $\boldsymbol{x}_j$ and the corresponding residual by $\boldsymbol{r}_j = \boldsymbol{b} - A\boldsymbol{x}_j$, the residuals satisfy the property that [60, 61]

$$\|\boldsymbol{r}_j\| = \min_{p\in\mathcal{P}_j^0} \|p(A)\boldsymbol{r}_0\|, \tag{4.1}$$

where $\mathcal{P}_j^0$ are the monic polynomials with degree less than or equal to j. This guarantees a nondecreasing residual, but for larger problems, maintaining a linearly increasing Krylov subspace for all of the iterations becomes prohibitive. This can be remedied by using a restarted version of

GMRES, in which a maximum number of iterations, m, is chosen, after which the equation $A\boldsymbol{e} = \boldsymbol{r}$ is solved in order to update the current approximation by $\boldsymbol{x} = \boldsymbol{x} + \boldsymbol{e}$. This equation itself is solved by applying GMRES with a Krylov subspace generated from the current residual:

$$\mathcal{K}_n = \langle \boldsymbol{r}, A\boldsymbol{r}, \ldots, A^{n-1}\boldsymbol{r} \rangle.$$

These corrections proceed iteratively, to produce restarted GMRES, which is given in Algorithm 3.

4.1.1 Convergence

There are a number of convergence theorems for GMRES that are based on creating bounds for Equation (4.1) by characterizing how it is affected by the properties of A and $\boldsymbol{r}_0$ [61]. Some are based on the singular values of A [62], but the most relevant ones for this work are bounds based on spectrum of A, which we denote by $\sigma(A)$. The first step in constructing such a bound is to take Equation (4.1) and bound it above by the worst case $\boldsymbol{r}_0$, leading to

$$\|\boldsymbol{r}_j\| \leq \min_{p \in \mathcal{P}_j^0} \|p(A)\| \|\boldsymbol{r}_0\|, \tag{4.2}$$

so that we're now concerned only with the properties of $\|p(A)\|$. If A is diagonalizable, with $A = S\Lambda S^{-1}$, then the inequality becomes [63, 60, 61]

$$\|\boldsymbol{r}_j\| \leq \min_{p \in \mathcal{P}_j^0} \|Sp(\Lambda)S^{-1}\| \|\boldsymbol{r}_0\|, \tag{4.3}$$

which gives [61]

$$\frac{\|\boldsymbol{r}_j\|}{\|\boldsymbol{r}_0\|} \leq \kappa(S) \min_{p \in \mathcal{P}_j^0} \max_{\lambda \in \Lambda(A)} |p(\lambda)|, \tag{4.4}$$

where $\kappa(S) = \|S\| \|S^{-1}\|$ is the condition number of S. In the methods described in Section 4.1.2 and Section 4.3, this bound is lowered by effectively removing eigensubspaces within span(S), and eigenvalues from $\Lambda(A)$ through Krylov subspace enrichment or deflation. For both methods, following the notation in [64], we denote the subset of k eigenvectors that are removed by $S_1 = \{\varphi_1, \cdots, \varphi_k\}$, and the remaining eigenvectors by $S_2 = \{\varphi_{k+1}, \cdots, \varphi_n\}$. We also split the initial residual into $\boldsymbol{r}_0 = \boldsymbol{r}_{0,1} + \boldsymbol{r}_{0,2}$, where $\boldsymbol{r}_{0,1}$ is the residual projected onto S_1, and $\boldsymbol{r}_{0,2}$ is the residual projected onto S_2.

Algorithm 1 Arnoldi Iteration

```
Input: $A, \boldsymbol{b}$
Output: $Q_{n+1}, \tilde{H}_n$
  $\mathbf{q}_1 = \mathbf{b}/\|\mathbf{b}\|$
  for n=1,... do
    $\mathbf{v} = A\mathbf{q}_n$
    for j=1,..,n do
      $h_{jn} = \mathbf{q}_j^*\mathbf{v}$
      $\mathbf{v} = \mathbf{v} - h_{jn}\mathbf{q}_j$
    end for
    $h_{n+1,n} = \|\mathbf{v}\|$
    $\mathbf{q}_{n+1} = \mathbf{v}/h_{n+1,n}$
  end for
```

Algorithm 2 GMRES

```
Input: $A$,$\boldsymbol{b}$,$\boldsymbol{x}_0$,$\epsilon$,$m$
Output: $\boldsymbol{x}$
  Initialization
  $\mathbf{r}_0 = \mathbf{b} - A\mathbf{x}_0$
  $\mathbf{q}_1 = \mathbf{r}_0/\|\mathbf{r}_0\|$

  Arnoldi Iteration
  for $j = 1$ to $m$ do
    $\mathbf{v} = A\mathbf{q}_j$
    for $i$=1 to $j$ do
      $h_{ij} = \mathbf{q}_i^*\mathbf{v}$
      $\mathbf{q}_{j+1} = \mathbf{q}_{j+1} - h_{ij}\mathbf{q}_i$
    end for
    $h_{j+1,j} = \|\mathbf{q}_{j+1}\|$
    $\mathbf{q}_{j+1} = \mathbf{q_{j+1}}/h_{j+1,j}$
  end for

  Find approximate solution
  $\beta = \|\mathbf{r}_0\|$
  Find $\hat{\mathbf{d}} \in \mathbb{R}^l$ that minimizes $\|\beta\mathbf{e}_1 - \tilde{H}\mathbf{d}\|$
  $\hat{\mathbf{x}} = \mathbf{x}_0 + Q\hat{\mathbf{d}}$
```

Algorithm 3 Restarted GMRES

```
Input: $A, \boldsymbol{b}, \boldsymbol{x}_0, \epsilon, m, n_{\max}$
Output: $\boldsymbol{x}$
  for $n = 0$ to $n_{\max} - 1$ do
      Initialization
      $\mathbf{r}_0 = \mathbf{b} - A\mathbf{x}_0$
      $\mathbf{q}_1 = \mathbf{r}_0 / \|\mathbf{r}_0\|$

      Arnoldi Iteration
      for $j = 1$ to $m$ do
          $\mathbf{v} = A\mathbf{q}_j$
          for $i$=1 to $j$ do
              $h_{ij} = \mathbf{q}_i^* \mathbf{v}$
              $\mathbf{q}_{j+1} = \mathbf{q}_{j+1} - h_{ij}\mathbf{q}_i$
          end for
          $h_{j+1,j} = \|\mathbf{q}_{j+1}\|$
          $\mathbf{q}_{j+1} = \mathbf{q_{j+1}} / h_{j+1,j}$
      end for

      Find approximate solution
      $\beta = \|\mathbf{r}_0\|$
      Find $\hat{\mathbf{d}} \in \mathbb{R}^m$ that minimizes $\|\beta\mathbf{e}_1 - \tilde{H}\mathbf{d}\|$
      $\hat{\mathbf{x}} = \mathbf{x}_0 + Q\hat{\mathbf{d}}$

      Restart
      $\mathbf{r} = \mathbf{b} - A\hat{\mathbf{x}}$
      if $\|\mathbf{r}\| < \epsilon$ then
          return $\boldsymbol{x}$
      else
          $\mathbf{x}_0 = \hat{\mathbf{x}}$
      end if
  end for
```

4.1.2 Enriched subspace GMRES

The Krylov subspace can be extended to created the enriched subspace

$$\mathcal{K}_n^m = \langle \boldsymbol{b}, A\boldsymbol{b}, \ldots, A^{n-1}\boldsymbol{b}, \boldsymbol{v}_1, \ldots, \boldsymbol{v}_m \rangle,$$

where $\mathbf{v}_1, \ldots, \mathbf{v}_m$ is a set of vectors that each ideally have large components in the solution. The goal is then to determine

$$\min_{\boldsymbol{y} \in \mathbb{R}^n} \|\boldsymbol{b} - AK_n^m \boldsymbol{y}\|, \tag{4.5}$$

where K_n^m is a matrix whose columns span $\mathcal{K}_n^m$:

$$K_n^m = (K_n | \boldsymbol{v}_1 | \ldots | \boldsymbol{v}_2) \,. \tag{4.6}$$

A method by Morgan takes such an approach, in which the Krylov subspace is enriched with approximate eigenvectors that are calculated during the course of each restart [65]. In that work, the method is called "augmented GMRES", but in this work it will be referred to as "enriched subspace GMRES", or simply "enriched GMRES". In enriched subspace GMRES, the bound in Equation (4.4) is lowered by seeking approximate eigenvectors that have the smallest associated eigenvalues [65]. In the method, the Arnoldi iteration is carried out as in GMRES to produce an orthonormal matrix V whose columns span the Krylov subspace (previously labeled Q). This matrix is then extended to create a matrix W, formed by appending the previously-computed k approximate eigenvectors, labeled $\varphi_1, \cdots, \varphi_k$ to V [65]:

$$W = (V | \varphi_1 | \varphi_2 | \cdots | \varphi_k)$$

The number of columns of W is then labeled $l = m + k$. It is the subspace spanned by the columns of W in which approximate eigenpairs with the smallest eigenvalues in magnitude are computed using a particular Rayleigh-Ritz procedure [66, 67] for the reduced eigenvalue problem [65]:

$$W^* A^* W \bar{\varphi}_i = \frac{1}{\bar{\lambda}_i} W^* A^* A W \bar{\varphi}_i.$$

The solution of this generalized eigenvalue problem yields an approximate eigenvector through $\varphi_i = W\bar{\varphi}_i$ [65]. For the minimization problem, Q is created by orthogonalizing the vectors $A\varphi_i$

against the Arnoldi vectors, through [65]

$$AW = Q\tilde{H}.$$

The full method is given in Algorithm 4 [65], and continues to Algorithm 5. The quantities F and G are shorthand for $F = W^*A^*W$ and $G = W^*A^*AW$. The convergence bound for the method using only approximate eigenvectors, and no Krylov vectors, is given in Theorem 1 [67, 64]. This bound is constructed in a similar manner to that in Equation (4.4), but with the effect of the smallest eigenvalues removed.

Theorem 1. *Let $S_1 = \{\varphi_1, \cdots, \varphi_k\}$ and $S_2 = \{\varphi_{k+1}, \cdots, \varphi_n\}$. For $m = 0$, the residual r_l computed with Algorithm 4 satisfies*

$$\|\boldsymbol{r}_l\| \leq \|\boldsymbol{r}_{0,2}\| \min_{p \in \mathcal{P}_l^0} \max_{k+1 \leq i \leq n} |p(\lambda_i)| \text{cond}(S_2), \tag{4.7}$$

where $\text{cond}(S_2) = \|S_2\| \|(S_2^* S_2)^{-1} S_2^*\|$.

4.2 Classical preconditioners

With left preconditioning, we solve the system

$$M^{-1}A\boldsymbol{x} = M^{-1}\boldsymbol{b},$$

and with right preconditioning we solve the system

$$AM^{-1}M\boldsymbol{x} = \boldsymbol{b},$$

with a preconditioner M^{-1} designed to reduce the condition number of A. For GMRES, applying preconditioning is straightforward, and we present right-preconditioned GMRES in Algorithm 6. We briefly mention two simple preconditioners here that are used for benchmarking purposes in Chapter 6, before describing deflation preconditioners in the next section. The two simple preconditioners both start with the decomposition

$$A = D + L + U,$$

Algorithm 4 Restarted GMRES enriched with approximate eigenvectors

```
Input: $A, \boldsymbol{b}, \boldsymbol{x}_0, \epsilon, m, n_{\max}$
Output: $\boldsymbol{x}$
  $k = 0$
  for $n = 0$ to $n_{\max} - 1$ do
      Pick enrichment vectors $y_1, ..., y_k$ from the previous approximate eigenvectors.
      $l = m + k$

      Initialization
      $\mathbf{r}_0 = \mathbf{b} - A\mathbf{x}_0$
      $\mathbf{q}_1 = \mathbf{r}_0/\|\mathbf{r}_0\|$
      $\mathbf{w}_1 = \mathbf{q}_1$
      for $i$=1 to $k$ do
          $\mathbf{w}_{m+i} = \varphi_i$
      end for
      for $j$=$m + 1$ to l do
          for $i$=$m + 1$ to l do
              $\mathbf{f}_{ij} = \bar{\varphi}_i^* F_{\text{old}} \bar{\varphi}_j$
          end for
      end for

      Arnoldi Iteration
      for $j = 1$ to $m$ do
          $\mathbf{v} = A\mathbf{q}_j$
          $\mathbf{q}_{j+1} = \mathbf{v}$
          for $i$=1 to $j$ do
              $h_{ij} = \mathbf{q}_i^* \mathbf{q}_{j+1}$
              $\mathbf{f}_{ji} = h_{ij}$
              $\mathbf{q}_{j+1} = \mathbf{q}_{j+1} - h_{ij}\mathbf{q}_i$
          end for
          for $i$=1 to $k$ do
              $\mathbf{f}_{j,m+i} = \varphi_i * \mathbf{v}$
          end for
          $h_{j+1,j} = \|\mathbf{q}_{j+1}\|$
          $\mathbf{q}_{j+1} = \mathbf{q}_{\mathbf{j+1}}/h_{j+1,j}$
          if $j < m$ then
              $\mathbf{w}_{j+1} = \mathbf{q}_{j+1}$
              $\mathbf{f}_{j,j+1} = h_{j+1,j}$
          end if
      end for
      *Continues to Algorithm 5*
```

Algorithm 5 Restarted GMRES enriched with approximate eigenvectors

```
    *Continued from Algorithm 4*
    **Addition of approximate eigenvectors**
    **for** $j = 1$ to $m + 1$ **do**
        $\mathbf{v} = A\mathbf{w}_j$
        $\mathbf{q}_{j+1} = v$
        **for** $i$=1 to $j$ **do**
            $h_{ij} = \mathbf{q}_i^* \mathbf{q}_{j+1}$
            $\mathbf{q}_{j+1} = \mathbf{q}_{j+1} - h_{ij}\mathbf{q}_i$
        **end for**
        **for** $i$=1 to $m$ **do**
            $\mathbf{f}_{ji} = h_{ij}$
        **end for**
        $h_{j+1,j} = \|\mathbf{q}_{j+1}\|$
        $\mathbf{q}_{j+1} = \mathbf{q_{j+1}}/h_{j+1,j}$
    **end for**

    **Find approximate solution**
    $\beta = \|\mathbf{r}_0\|$
    Find $\hat{\mathbf{d}} \in \mathbb{R}^l$ that minimizes $\|\beta\mathbf{e}_1 - \tilde{H}\mathbf{d}\|$
    $\hat{\mathbf{x}} = \mathbf{x}_0 + W\hat{\mathbf{d}}$

    **Form the new approximate eigenvectors**
    $k = k + 1$
    $G = R^*R$
    Solve $F\bar{\varphi}_k = \frac{1}{\theta_k}G\bar{\varphi}_k$
    $\varphi_k = Q\bar{\varphi}_k$
    $A\varphi_k = Q\tilde{H}\bar{\varphi}_k$
    $F_{\text{old}} = F$

    **Restart**
    $\mathbf{r} = \mathbf{b} - A\hat{\mathbf{x}}$
    **if** $\|\mathbf{r}\| < \epsilon$ **then**
        **return** $\boldsymbol{x}$
    **else**
        $\mathbf{x}_0 = \hat{\mathbf{x}}$
    **end if**
**end for**
```

Algorithm 6 Right-preconditioned restarted GMRES

```
**Input:** $A, \boldsymbol{b}, \boldsymbol{x}_0, \epsilon$, $m$, $n_{\max}$
**Output:** $\boldsymbol{x}$
  **for** $n = 0$ to $n_{\max} - 1$ **do**
      **Initialization**
      $\mathbf{r}_0 = \mathbf{b} - AM^{-1}\mathbf{x}_0$
      $\mathbf{q}_1 = \mathbf{r}_0/\|\mathbf{r}_0\|$

      **Arnoldi Iteration**
      **for** $j = 1$ to $m$ **do**
          $\mathbf{v} = A\mathbf{q}_j$
          **for** $i$=1 to $j$ **do**
              $h_{ij} = \mathbf{q}_i^*\mathbf{v}$
              $\mathbf{q}_{j+1} = \mathbf{q}_{j+1} - h_{ij}\mathbf{q}_i$
          **end for**
          $h_{j+1,j} = \|\mathbf{q}_{j+1}\|$
          $\mathbf{q}_{j+1} = \mathbf{q_{j+1}}/h_{j+1,j}$
      **end for**

      **Find approximate solution**
      $\beta = \|\mathbf{r}_0\|$
      Find $\hat{\mathbf{d}} \in \mathbb{R}^m$ that minimizes $\|\beta\mathbf{e}_1 - \tilde{H}\mathbf{d}\|$
      $\hat{\mathbf{x}} = \mathbf{x}_0 + M^{-1}Q\hat{\mathbf{d}}$

      **Restart**
      $\mathbf{r} = \mathbf{b} - A\hat{\mathbf{x}}$
      **if** $\|\mathbf{r}\| < \epsilon$ **then**
          **return** $\boldsymbol{x}$
      **else**
          $\mathbf{x}_0 = \hat{\mathbf{x}}$
      **end if**
  **end for**
```

where D is a diagonal matrix, L is a strictly lower triangular matrix, and U is a strictly upper triangular matrix. The simplest of these, diagonal preconditioning, simply sets $M^{-1} = D^{-1}$, which can be effective for diagonally dominant matrices. A more viable extension of this idea is the two-step Jacobi iteration [68]. First define $r = \rho(L + U)$ to be the spectral radius of $L + U$, and premultiply the system by D^{-1}, defining $\bar{\boldsymbol{b}} = D^{-1}\boldsymbol{b}$. Then, given a parameter $\alpha > 0$, the iteration is [68]

$$((\alpha + r)I + L + U)\,\boldsymbol{x} = \bar{\boldsymbol{b}} + (\alpha + r - 1)\,\boldsymbol{x}.$$

The method converges if $\rho(M(\alpha)) < 1$, where [68]

$$M(\alpha) = (\alpha + r - 1)\left[(\alpha + r)I + L + U\right]^{-1}.$$

4.3 Deflation preconditioned GMRES

A number of methods take the approach of constructing a preconditioner for GMRES based on approximations to the eigenvalues and eigenvectors of A. In one approach, the Implicitly Restart Arnoldi method (IRA) [69, 70] is used to construct a preconditioner. In another approach, which we focus on here, approximate eigenvectors are used at each restart to construct a deflation preconditioner that is persistent throughout the method [71]. The aim of the preconditioner is primarily to create an AM^{-1} for which the smallest magnitude eigenvalues of A have been removed. These eigenvalues are replaced with eigenvalues equal to the magnitude of the largest magnitude eigenvalue of A [71]. If we assume that A is nondefective, and label the smallest r eigenvalues by $|\lambda_1| \leq |\lambda_2| \leq \cdots \leq |\lambda_r|$, the preconditioner is constructed in order to solve $A\boldsymbol{x} = \boldsymbol{b}$ exactly in the subspace

$$P = \langle \varphi_1, \varphi_2, \cdots, \varphi_r \rangle,$$

where φ_i is the eigenvector corresponding to the eigenvalue λ_i [71]. Given an orthonormal basis U of P, the authors provide a method for constructing such a preconditioner, given in Theorem 2 [71].

Theorem 2. *If $T = U^T A U$ and $M = I_n + U\left(1/|\lambda_n| T - I_r\right) U^T$, then M is nonsingular, $M^{-1} = I_n + U\left(|\lambda_n| T^{-1} - I_r\right) U^T$, and the eigenvalues of AM^{-1} are $\lambda_{r+1}, \lambda_{r+2}, \cdots, \lambda_n, |\lambda_n|$, with the multiplicity of $|\lambda_n|$ at least r.*

With that established, the remaining step is to construct an approximation to P using the quantities available from a standard restarted GMRES. The authors naturally use the Ritz values and vectors from the Arnoldi Hessenberg matrix $\tilde{H}_m$. Specifically, they decompose it into the Schur form $\tilde{H}_m = \tilde{S}B\tilde{S}^*$, ordered by increasing eigenvalue, and use the vectors $U = Q\tilde{S}$ to approximate the Schur vectors of A. The method, known as DGMRES, is given in full in Algorithm 7. The authors noted instances of increasing residuals for left-preconditioning in their implementation, so we only deal with the case of right-preconditioning here, which guarantees a nonincreasing residual [71]. The convergence bound for the method is given in Theorem 3 [64]. Other variants of this method include ones that use harmonic projections for the eigenvector approximations, and a scheme in which the vectors in U are continuously updated [64].

Theorem 3. *Let $Y_2 = \{y_{r+1}, y_{r+2}, \cdots, y_n\}$ be the eigenvectors of AM^{-1} corresponding to the eigenvalues $\lambda_{r+1}, \lambda_{r+2}, \cdots, \lambda_n$. The residual $\boldsymbol{r}_m$ computed with Algorithm 7 satisfies*

$$\|\boldsymbol{r}_m\| \leq \min_{p \in \mathcal{P}_m^0} \left(p(|\lambda_n|)\|\boldsymbol{r}_{0,1}\| + \max_{k+1 \leq i \leq n} |p(\lambda_i)| \|\boldsymbol{r}_{0,2}\| \mathrm{cond}(Y_2) \right), \tag{4.8}$$

where $\boldsymbol{r}_{0,1}$ is the projection of $\boldsymbol{r}$ onto S_1, $\boldsymbol{r}_{0,2}$ is the projection of $\boldsymbol{r}$ onto Y_2, and $\mathrm{cond}(Y_2) = \|Y_2\|(Y_2^* Y_2)^{-1} Y_2^*\|$.

4.4 Multigrid methods

In this section we cover multigrid methods, which are used in a number of preconditioners for Krylov methods. We first present a framework for multigrid methods (Section 4.4.1), and some convergence theorems for a two-grid geometric multigrid method as both a standalone method, and as a preconditioner for GMRES (Section 4.4.2). We then highlight the particular methods that are implemented in the Lawrence Livermore National Laboratory (LLNL) package HYPRE [72], which are taken to be representative of what is used in practice by the high performance computing community. These methods are parallel full multigrid (PFMG) (Section 4.4.3) and SMG (Section 4.4.4).

4.4.1 Two-grid multigrid

We adopt the framework used in [73] for the Poisson equation,

$$-\nabla^2 u = f,$$

Algorithm 7 DGMRES

```
Pick an arbitrary $\mathbf{x}_0$.
Set a convergence tolerance $\epsilon$
Pick a number of eigenvectors to approximate per restart, $l$
Pick a maximum number of restarts to update the preconditioner, $k_{\text{precond}}$
$M^{-1} = I$
for $k$=0 to $k_{\text{max}}$-1 do
    Pick the dimension of the base Krylov subspace, $m$

    Initialization
    $\mathbf{r}_0 = \mathbf{b} - M^{-1}A\mathbf{x}_0$
    $\mathbf{q}_1 = \mathbf{r}_0/\|\mathbf{r}_0\|$

    Arnoldi Iteration
    Get $M^{-1}A = Q_m\tilde{H}_mQ_m^*$

    Find approximate solution
    $\beta = \|\mathbf{r}_0\|$
    Find $\hat{\mathbf{d}} \in \mathbb{R}^m$ that minimizes $\|\beta\mathbf{e}_1 - \tilde{H}\mathbf{d}\|$
    $\hat{\mathbf{x}} = \mathbf{x}_0 + M^{-1}Q\hat{\mathbf{d}}$
    $\mathbf{r} = \mathbf{b} - M^{-1}A\hat{\mathbf{x}}$
    if $\|\mathbf{r}\| < \epsilon$ then
        return $\boldsymbol{x}$
    else
        $\mathbf{x}_0 = \hat{\mathbf{x}}$
    end if

    Update preconditioner
    if k<$k_{\text{precond}}$ then
        Schur factorize $\tilde{H}_m = \tilde{S}B\tilde{S}^*$
        Order Schur decomposition by increasing eigenvalue
        $\lambda_m = max(\sigma_B)$
        $S = Q_m\tilde{S}$
        for $j$=1 to $l$ do
            $J = j + kl$
            $u_J = s_j$
            for $i$=1 to $j + kl - 1$ do
                $u_J = u_J - (u_i^*u_J)u_i$
            end for
            $u_J = u_J/\|u_J\|$
        end for
        $T = U^*AU$
        $M^{-1} = I + U^*(T^{-1}\|\lambda_m\| - I_{(k+1)l})U$
    end if
end for
```

and give their convergence theorems in Section 4.4.2 for multigrid used both as a standalone method, and as a preconditioner for GMRES. Given the current error in the approximation, v_{j-1}, the effect of a two-grid iteration is given by [73]

$$v_j = M_h^{2h} v_{j-1},$$

where v_j is the error at iteration j, and M_h^{2h} represents one multigrid cycle. We now break this operator down this cycle into its smallest components. First, we have

$$M_h^{2h} = S_h^{\nu_2} K_h^{2h} S_h^{\nu_1},$$

where $S_h^{\nu_1}$ are ν_1 iterations of pre-smoothing, K_h^{2h} is the coarse-grid correction, and $S_h^{\nu_2}$ are ν_2 iterations of post-smoothing. The coarse grid correction is given by

$$K_h^{2h} = I_h - I_{2h}^h (\nabla_{2h}^2)^{-1} I_h^{2h} \nabla_h^2,$$

where ∇_h^2 is the operator discretized on the fine grid, I_h^{2h} is the fine-to-coarse transfer operator, $(\nabla_{2h}^2)^{-1}$ is the coarse grid solve, I_{2h}^h is the coarse-to-fine transfer operator, and I_h is the fine-grid identity. Overall, this gives

$$v_j = S_h^{\nu_2} (I_h - I_{2h}^h (\nabla_{2h}^2)^{-1} I_h^{2h} \nabla_h^2) S_h^{\nu_1} v_{j-1}.$$

If we use red-black Gauss-Seidel smoothing, this can be broken down further into a red relaxation followed by a black relaxation, given by

$$S_h = S_B S_R,$$

where the effect of S_R is

$$S_R v = \frac{1}{2}\left(\begin{pmatrix} 0 & 1 & 0 & & & & & \\ 0 & 2 & 0 & \ddots & & & & \\ 0 & 1 & 0 & 1 & \ddots & & & \\ 0 & 0 & 0 & 2 & 0 & \ddots & & \\ & \ddots & 0 & 1 & 0 & 1 & \ddots & \\ & & \ddots & & \ddots & & & 0 \\ & & & \ddots & 1 & 0 & 1 & 0 \\ & & & & \ddots & 0 & 2 & 0 \\ & & & & & 0 & 1 & 0 \end{pmatrix}\begin{pmatrix} v_1 \\ v_2 \\ \vdots \\ \vdots \\ \vdots \\ \vdots \\ \vdots \\ \vdots \\ v_{n-1} \end{pmatrix} + h^2 \begin{pmatrix} f_1 \\ 0 \\ f_3 \\ 0 \\ \vdots \\ 0 \\ f_{n-3} \\ 0 \\ f_{n-1} \end{pmatrix}\right),$$

and the effect of S_B is

$$S_B v = \frac{1}{2}\left(\begin{pmatrix} 2 & 0 & 0 & & & & & \\ 1 & 0 & 1 & \ddots & & & & \\ 0 & 0 & 2 & \ddots & \ddots & & & \\ 0 & 0 & 1 & \ddots & \ddots & \ddots & & \\ & \ddots & 0 & \ddots & \ddots & \ddots & \ddots & \\ & & \ddots & \ddots & \ddots & & & 0 \\ & & & \ddots & 0 & 2 & 0 & 0 \\ & & & & \ddots & 1 & 0 & 1 \\ & & & & & 0 & 0 & 2 \end{pmatrix}\begin{pmatrix} v_1 \\ v_2 \\ \vdots \\ \vdots \\ \vdots \\ \vdots \\ \vdots \\ \vdots \\ v_{n-1} \end{pmatrix} + h^2 \begin{pmatrix} 0 \\ f_2 \\ 0 \\ f_4 \\ 0 \\ \vdots \\ 0 \\ f_{n-2} \\ 0 \end{pmatrix}\right).$$

4.4.2 Two-grid convergence

Standalone multigrid The discretization and approximation of ∇^2 with 5-point finite-difference stencil can be written as [73].

$$\begin{aligned} & \nabla_h^2 u_h(\boldsymbol{x}) \\ \approx \ & \sum_{\kappa \in J} \alpha_\kappa u_h(\boldsymbol{x} + \kappa h), \end{aligned}$$

where J is the set of stencil offsets, $\{(0,0),(-1,0),(1,0),(0,-1),(0,1)\}$, and κ is a multi-index. More directly as a matrix:

$$A\boldsymbol{u} = \sum_{\kappa \in J} \alpha_\kappa \boldsymbol{u}_\kappa,$$

where $\boldsymbol{u}_\kappa$ is $\boldsymbol{u}$ shifted by the stencil offset. If we now call $F(\Omega_h)$ the space of all grid functions on the domain, one basis for $F(\Omega_h)$ is the Fourier basis [73]:

$$\varphi_h^{k,l}(x,y) = \sin(k\pi x)\sin(l\pi y),$$

with $k,l = 1,\ldots,n-1$, for $(n-1)^2$ total basis functions. It's possible to divide $F(\Omega_h)$ into a direct sum of at most 4-dimensional spaces. There are $\left(\frac{n}{2}\right)^2$ of these spaces, denoted by

$$E_h^{k,l} = \operatorname{span}\{\varphi_h^{k,l}, \varphi_h^{n-k,n-l}, \varphi_h^{n-k,l}, \varphi_h^{k,n-l}\},$$

with $k,l = 1,\ldots,\frac{n}{2}$. The dimension is lower than 4 when k or l or both are equal to $n/2$. Elsewhere it is proved that each of these subspaces is invariant under K_h^{2h}. We can now take a vector $\boldsymbol{x}$ represented in the Fourier basis,

$$\boldsymbol{x} = \sum_{k,l=1}^{n-1} c_{k,l}\varphi_h^{k,l},$$

where $c_{k,l}$ are the Fourier coefficients, and write it in terms of the invariant subspaces $E_h^{k,l}$ [73]:

$$\boldsymbol{x} = \sum_{k,l=1}^{n/2} \sum_{i=1}^{4} c_{k,l}^i E_i^{k,l},$$

where $E_i^{k,l}$ is the ith vector in $E^{k,l}$ and $c_{k,l}^i$ is the coefficient for that vector. Applying K_h^{2h}, we get

$$\begin{aligned} K_h^{2h}\boldsymbol{x} &= \sum_{k,l=1}^{n/2} K_h^{2h} \sum_{i=1}^{4} c_{k,l}^i E_i^{k,l} \\ &= \sum_{k,l=1}^{n/2} \sum_{i=1}^{4} d_{k,l}^i E_i^{k,l}, \end{aligned}$$

where the $d_{k,l}^i$, $i = 1,\ldots,4$ for a given k and l can be written as a linear combination of the $c_{k,l}^i$. If we write K_h^{2h} in the Fourier basis and call it $\tilde{K}$, it will be a block matrix of 4×4 blocks. Additionally, the red-black Gauss-Seidel relaxations $S_h^{\nu_1}$ and $S_h^{\nu_2}$ leave the same invariant subspaces, so the matrix M_h^{2h} is orthogonally equivalent to a matrix $\tilde{M}$ of 4×4 blocks [73]. More specifically, it can be written as $\tilde{M} = UM_h^{2h}U^{-1}$, where U is a unitary matrix. Calling the blocks $\hat{M}(k,l)$, the two-grid convergence factor can be written as [73]

$$\rho_F = \rho(\tilde{M}) = \max_{1 \leqslant k,l \leqslant \frac{n}{2}} \rho(\hat{M}(k,l)).$$

That is, the convergence factors can be computed per block, with the worst determining the overall convergence factor.

Multgrid preconditioned GMRES The multigrid cycle can be represented through the matrix splitting [73]

$$Cu_j + (A - C)u_{j-1} = f,$$

or more explicitly

$$u_j = u_{j-1} + C^{-1}(f - Au_{j-1}).$$

Using this as a right-preconditioner

$$AM^{-1}Mu = f,$$

where M^{-1} is the multigrid cycle, the problem is re-written as

$$\begin{aligned} AM^{-1}y &= f, \\ u &= M^{-1}y. \end{aligned}$$

This leads to GMRES(m) finding a correction $C(u_j - u_{j-m})$ in the Krylov subspace $K_m = \text{span}\{r_{j-m}, (AC^{-1})r_{j-m}, \ldots, (AC^{-1})^{n-1}r_{j-m}\}$. Any element w in the affine subspace $u_{j-m} + C^{-1}K^m(AC^{-1}, r_{j-m})$ can be represented by [73]

$$w = u_{j-m} + C^{-1}(\alpha_1 r_{j-m} + \alpha_2 AC^{-1} + \ldots + \alpha_m (AC^{-1})^{m-1} r_{j-m}),$$

where $\alpha_1, \ldots, \alpha_m$ are scalars. Substituting the vector into the residual equation gives

$$\begin{aligned} r_w &= b - Aw \\ &= r_{j-m} - AC^{-1}(\alpha_1 r_{j-m} + \alpha_2 AC^{-1} + \ldots + \alpha_m (AC^{-1})^{m-1} r_{j-m}). \end{aligned}$$

This leads to the representation [73]

$$r_w = P_m(AC^{-1}) r_{j-m},$$

where $P_m = 1 - \sum_{k=1}^{m} \alpha_k \lambda^k$. In this case, r_j satisfies the property [73]

$$\|r_j\|_2 = \min_{p \in \mathcal{P}_m^0} \|p(AC^{-1}) r_{j-m}\|.$$

An analysis with the spectrum of the iteration matrix gives the bound [73]

$$\|r_{i \cdot m}\|_2 \leqslant (1 - \alpha/\beta)^{m/2} \|r_{(i-1)m}\|_2,$$

where $\alpha = \left(\lambda_{\min}\left(\frac{1}{2}(AC^{-1} + (AC^{-1})^T)\right)\right)^2$ and $\beta = \lambda_{\max}((AC^{-1})^T AC^{-1})$.

4.4.3 Parallel full multigrid

A parallel multigrid preconditioner was developed by Ashby *et. al.* for use with conjugate gradient (CG), specifically for groundwater flow problems of the form [74]

$$-\nabla \cdot (K \nabla u) = f,$$

called the multigrid conjugate gradient method (MGCG). When the multigrid component alone from MGCG is used in other contexts, it's simply referred to as PFMG [75].

4.4.4 Semicoarsening multigrid

The SMG method is a method designed for rectangular (at least in connectivity) discretizations of general elliptic PDEs [76]. A highly scalable version of the method for distributed memory machines was created for the subset of these problems of the form [77, 75]

$$-\nabla \cdot (K \nabla u) - qu = f$$

This method is more robust than PFMG, but it's slightly less efficient per V-cycle [72].

CHAPTER 5

Kinetic Methods

In Chapter 4, we presented methods of preconditioning and Krylov subspace enrichment that relied on the availability of approximate eigenvectors of the differential operator. In this chapter, the focus is on obtaining these approximations by the use of methods from kinetic formulations of the PDE. As will be seen in Chapter 6, where the concurrent schemes are presented, ideal methods are ones that can be halted at an arbitrary point in the computation. Further, the methods will need to be highly parallelizable, to allow for efficient GPU computation. The kinetic formulation upon which these methods will be built is the Feynman-Kac formulation.

5.1 Feynman-Kac formulation

A link can be made between stochastic processes and second-order PDEs of the form in Equation (1.1). The link is made via the Feynman-Kac formula [78, 79], for which a special case is given in Equation 5.2. The special case under consideration is the subset of problems that are time-independent and have $q(\boldsymbol{x}) = 0$, given by

$$\sum_{i=1}^{d} \alpha_i(\boldsymbol{x}) \frac{\partial u}{\partial x_i} + \frac{1}{2} \sum_{i,j=1}^{d} \beta_{ij}(\boldsymbol{x}) \frac{\partial^2 u}{\partial x_i \partial x_j} + f(\boldsymbol{x}) = 0. \tag{5.1}$$

Furthermore, we consider the cases where the coefficients on the second order terms can be represented as $\beta_{ij}(\boldsymbol{x}) = \sum_{k=1}^{d} \sigma_{ik}(\boldsymbol{x})\sigma_{jk}(\boldsymbol{x})$. The solution to this class of equations can be written as [79]

$$u(\boldsymbol{x}) = \mathbb{E}^{\boldsymbol{x}} \left[\int_0^T f(\boldsymbol{X}(s))ds \right] + \mathbb{E}^{\boldsymbol{x}}[u(\boldsymbol{X}(T))], \tag{5.2}$$

where the path $\boldsymbol{X}(t)$ is an Ito diffusion given by [79]

$$d\boldsymbol{X}(t) = \alpha(\boldsymbol{X}(t))dt + \sigma(\boldsymbol{X}(t))d\boldsymbol{B}(t), \tag{5.3}$$

terminating when it exits the domain at time T, and the expectations $\mathbb{E}^x$ are taken over sample paths of this diffusion originating at $\boldsymbol{X}(0) = x$.

5.1.1 Sample path integration

The Ito diffusions can be integrated in a number of ways. The simplest is the Euler-Maruyama scheme, which is an analog of forward Euler, and reads [80]

$$\boldsymbol{X}_{n+1} = \boldsymbol{X}_n + \alpha(\boldsymbol{X}_n)\Delta t + \sigma(\boldsymbol{X}_n)\Delta \boldsymbol{B}_n, \tag{5.4}$$

where $\Delta \boldsymbol{B}_n = \sqrt{\Delta t}\mathcal{N}(0,1)$, with $\mathcal{N}(0,1)$ a random variable drawn from a multivariate normal of mean 0 and variance 1. This scheme is order 0.5 [80]. Alternative schemes are Milstein schemes of order 1 [81, 82], and stochastic Runge-Kutta schemes of order 1 [83, 84], 1.5 [85] and 2 [86, 87]. However, the convergence in the number of samples is only of order 0.5, so using these schemes is not worth the additional computation.

5.1.2 Adaptive time-stepping

In this work, there is a need to set timestep sizes to guarantee a particular probability of random walks exiting a certain region. We start by calculating the expected displacement for a single step of the Euler-Maruyama scheme. The displacement is given by

$$\begin{aligned} d\boldsymbol{X} &= \boldsymbol{X}_{n+1} - \boldsymbol{X}_n \\ &= \alpha(\boldsymbol{X}_n)\Delta t + \sigma(\boldsymbol{X}_n)\sqrt{\Delta t}\mathcal{N}(0,1). \end{aligned}$$

We consider the 1D case where $\alpha(x)$ and $\sigma(x)$ are constant, and use it to bound the cases of non-constant coefficients.

Diffusion only Label the displacement from diffusion by $d\boldsymbol{X}_\sigma = \sigma\sqrt{\Delta t}\mathcal{N}(0,1)$. The probability that the displacement from diffusion is greater than some constant D_σ is

$$P(d\boldsymbol{X}_\sigma > D_\sigma) = \frac{1}{2} - \frac{1}{2}\left[1 + \operatorname{erf}\left(\frac{D_\sigma}{\sqrt{2}\sigma\sqrt{\Delta t}}\right)\right],$$

and the probability that the displacement from diffusion is less than some constant $-D_\sigma$ is

$$P(d\boldsymbol{X}_\sigma < -D_\sigma) = \frac{1}{2} - \frac{1}{2}\left[1 + \operatorname{erf}\left(\frac{D_\sigma}{\sqrt{2}\sigma\sqrt{\Delta t}}\right)\right],$$

to give

$$P(\|d\boldsymbol{X}_\sigma\| \leq D_\sigma) = \operatorname{erf}\left(\frac{D_\sigma}{\sqrt{2}\sigma\sqrt{\Delta t}}\right). \tag{5.5}$$

From these, it follows that

$$\Delta_\sigma(P) = 2\sigma\sqrt{\Delta t}\operatorname{erf}^{-1}(P)$$

is the displacement for which the probability of $\|d\boldsymbol{X}_\sigma\|$ being less than $\Delta_\sigma(P)$ is P. For n steps, Equation (5.5) becomes

$$P(\|d\boldsymbol{X}_\sigma\| \leq D_\sigma) = \operatorname{erf}\left(\frac{D_\sigma}{\sqrt{2n}\sigma\sqrt{\Delta t}}\right),$$

which gives

$$\Delta_\sigma(P) = \sqrt{2n}\sigma\sqrt{\Delta t}\operatorname{erf}^{-1}(P).$$

Now for a given number of steps n, we can now recover Δt such that the walks remain with a domain of width $2D_\sigma$ with probability P:

$$\Delta t = \frac{1}{2n}\left(\frac{D_\sigma}{\sigma\operatorname{erf}^{-1}(P)}\right)^2. \tag{5.6}$$

Diffusion and drift Now restoring the drift term, we label the constant displacement due to drift as $D_\alpha = \alpha\Delta t$. The probability that the displacement is greater than some constant D_R is

$$P(d\boldsymbol{X} > D_R) = \frac{1}{2} - \frac{1}{2}\left[1 + \operatorname{erf}\left(\frac{D_R - D_\alpha}{\sqrt{2}\sigma\sqrt{\Delta t}}\right)\right],$$

and the probability that the displacement is less than some constant $-D_L$ is

$$P(d\boldsymbol{X} < -D_L) = \frac{1}{2} - \frac{1}{2}\left[1 + \operatorname{erf}\left(\frac{D_L + D_\alpha}{\sqrt{2}\sigma\sqrt{\Delta t}}\right)\right],$$

to give

$$P(-D_L \leq d\boldsymbol{X} \leq D_R) = \frac{1}{2}\left[\operatorname{erf}\left(\frac{D_R - D_\alpha}{\sqrt{2}\sigma\sqrt{\Delta t}}\right) + \operatorname{erf}\left(\frac{D_L + D_\alpha}{\sqrt{2}\sigma\sqrt{\Delta t}}\right)\right].$$

For n steps, this becomes,

$$P(-D_L \leq d\boldsymbol{X} \leq D_R) = \frac{1}{2}\left[\operatorname{erf}\left(\frac{D_R - nD_\alpha}{\sqrt{2n}\sigma\sqrt{\Delta t}}\right) + \operatorname{erf}\left(\frac{D_L + nD_\alpha}{\sqrt{2n}\sigma\sqrt{\Delta t}}\right)\right].$$

Simplifying for the case of a symmetric domain, this is

$$P(\|d\boldsymbol{X}\| \leq D) = \frac{1}{2}\left[\operatorname{erf}\left(\frac{D - nD_\alpha}{\sqrt{2n}\sigma\sqrt{\Delta t}}\right) + \operatorname{erf}\left(\frac{D + nD_\alpha}{\sqrt{2n}\sigma\sqrt{\Delta t}}\right)\right].$$

While Δt cannot be solved for directly as in Equation (5.6), finding the roots of

$$F(\Delta t) = -P + \frac{1}{2}\left[\operatorname{erf}\left(\frac{D - n\alpha\Delta t}{\sqrt{2n}\sigma\sqrt{\Delta t}}\right) + \operatorname{erf}\left(\frac{D + n\alpha\Delta t}{\sqrt{2n}\sigma\sqrt{\Delta t}}\right)\right] \tag{5.7}$$

numerically is trivial.

Non-constant coefficients For the case of non-constant coefficients, we simply bound the probability of the walks staying in the domain by:

$$P(\|d\boldsymbol{X}\| \leq D) \leq \min_{\boldsymbol{x}} \frac{1}{2}\left[\operatorname{erf}\left(\frac{D - n\alpha(\boldsymbol{x})\Delta t}{\sqrt{2n}\sigma(\boldsymbol{x})\sqrt{\Delta t}}\right) + \operatorname{erf}\left(\frac{D + n\alpha(\boldsymbol{x})\Delta t}{\sqrt{2n}\sigma(\boldsymbol{x})\sqrt{\Delta t}}\right)\right].$$

5.2 Feynman-Kac formulation for principal eigenpair

An alternative to using the Feynman-Kac formulation directly for solving PDEs is to use it to approximate the structure of differential operators. This approximate structure can then be used to improve the convergence of deterministic solvers. The details of such approximations are examined in this section, and their application to improving convergence are examined in Chapter 6.

Here, an approach by Lejay *et al.* [88] is summarized for the case of determining the the principal eigenpair of the Laplacian on a domain through sampling of Ito diffusions. Extensions to multiple eigenpairs and general semi-elliptic operators are presented in Subsections 5.3 and 5.4 respectively.

The principal eigenpair of the Laplacian can be approximated by stochastic processes that solve the Cauchy problem [88]. The Cauchy problem for $u(\boldsymbol{x},t)$ is

$$\frac{\partial u(\boldsymbol{x},t)}{\partial t} = \frac{1}{2}\nabla^2 u(\boldsymbol{x},t) \quad \text{in} \quad \Omega \subset \mathbb{R}^d, \tag{5.8}$$

$$u(\boldsymbol{x},t) = 0 \quad \text{on} \quad \partial\Omega, \tag{5.9}$$

with $u(\boldsymbol{x},0) = u_0(\boldsymbol{x})$. For this problem, the solution is given by [88]

$$u(\boldsymbol{x},t) = \mathbb{P}(\tau_\Omega^{\boldsymbol{x}} > t), \tag{5.10}$$

where $\tau_\Omega^{\boldsymbol{x}}$ is the exit time from the domain Ω, of an Ito diffusion (Equation (5.3)) with zero drift ($a(\boldsymbol{X}_t) = 0$) starting at $\boldsymbol{x}$.

Given some initial condition $u_0(\boldsymbol{x})$ for the Cauchy problem (Section 5.2), taking an eigenfunction expansion of $u(\boldsymbol{x},t)$ yields a solution that can be written in terms of the evolution of the decomposition of the initial condition in the eigenfunctions of the Laplacian [89]:

$$u(\boldsymbol{x},t) = \langle \varphi_1^*(\boldsymbol{x}), u_0\rangle \exp(\lambda_1 t)\varphi_1(\boldsymbol{x}) + R(\boldsymbol{x},t), \tag{5.11}$$

where $\varphi_1(\boldsymbol{x})$ is the principal eigenfunction and $R(\boldsymbol{x},t) = o(\exp(\lambda_1 t))$ contains the evolution the components of $u_0(\boldsymbol{x})$ in the remaining eigenfunctions. Using this along with Equation (5.10) yields the following approximation to the principal eigenvalue of $\frac{1}{2}\nabla^2 u$ [88]:

$$\lambda_1 = \lim_{t\to\infty} \frac{1}{t} \log \mathbb{P}(\tau_\Omega^{\boldsymbol{x}} > t), \tag{5.12}$$

with $\boldsymbol{x}$ taken as any $\boldsymbol{x} \in \Omega$. Furthermore, an estimate for the principal eigenfunction of the adjoint (which in this case is just $\frac{1}{2}\Delta$ itself) is given by [89]:

$$\begin{aligned} \mathbb{E}^{\boldsymbol{x}}[u_0(\boldsymbol{X}_t) | t < \tau_\Omega^{\boldsymbol{x}}] &= \frac{\mathbb{E}^{\boldsymbol{x}}[u_0(\boldsymbol{X}_t); t < \tau_\Omega^{\boldsymbol{x}}]}{\mathbb{E}^{\boldsymbol{x}}[1; t < \tau_\Omega^{\boldsymbol{x}}]} \\ &= \frac{\langle u_0, \varphi_1^*\rangle \varphi_1(\boldsymbol{x}) \exp(\lambda_1 t) + o(\exp(\lambda_1 t))}{\langle 1, \varphi_1^*\rangle \varphi_1(\boldsymbol{x}) \exp(\lambda_1 t) + o(\exp(\lambda_1 t))} \\ &\backsimeq \frac{\langle u_0, \varphi_1^*\rangle}{\langle 1, \varphi_1^*\rangle}, \end{aligned} \tag{5.13}$$

which gives that the density of $\boldsymbol{X}_t$ is $\frac{\varphi_1^*}{\langle 1, \varphi_1^* \rangle}$ for large t [89].

Since only the large time behaviour is required, the authors instead consider estimating [89]

$$\frac{F(t) - F(T)}{1 - F(T)} = \mathbb{P}_x[\tau < t | \tau \geqslant T], \tag{5.14}$$

for some T, with $t \geqslant T$ and the shorthand $\mathbb{P}_x(\tau < t) \equiv \mathbb{P}(\tau_D^x < t)$. If the distribution π_T of the process X_T is known, then the Markov property of the process can be exploited to write the exit time in terms of walks whose positions are initialized at T with the known distribution π_T:

$$\mathbb{P}_x[\tau < t | \tau \geqslant T] = \mathbb{P}_{\pi_T}[\tau < t] = \int_D \mathbb{P}_y[\tau < t] d\pi_T(y). \tag{5.15}$$

An estimator for π_T itself is acquired using walks up to time T. This procedure can be applied to a sequence of times $T_1 < \ldots < T_k$, where estimators $\hat{\pi}_{T_1}, \ldots, \hat{\pi}_{T_k}$ are generated and the walks that inform $\hat{\pi}_{T_{i+1}}$ are seeded from $\hat{\pi}_{T_i}$. The algorithm is give in Algorithm 8[89]. The authors used no more than 3 branching times in their numerical experiments. They propose a number of ways of estimating λ_1 from Equation (5.12). As long as t is large enough so that $1 - F(t) \backsimeq C \exp((\lambda_1 - c)t)$, λ_1 can be estimated as [89]:

$$\lambda_1 = \frac{1}{t_1 - t_0} \log \left(\frac{\hat{F}(t_1)}{\hat{F}(t_0)} \right), \tag{5.16}$$

with $t_1 > t_0 > t$.

5.3 Multiple eigenpairs

This approach can be generalized to multiple ordered eigenpairs by exploiting the relative decay rates for each of the eigenpairs.

5.3.1 Cascading scheme

With this mechanism in mind, a cascading scheme can be designed where the first k eigenfunctions are recovered. First, we introduce a notation for expanding Equation (5.11):

$$\begin{aligned} u(\boldsymbol{x}, t) &= \langle \varphi_1^*, u_0 \rangle \exp(\lambda_1 t) \varphi_1 + R_1(\boldsymbol{x}, t), \\ R_i(\boldsymbol{x}, t) &= \langle \varphi_{i+1}^*, u_0 \rangle \exp(\lambda_{i+1} t) \varphi_{i+1} + R_{i+1}(\boldsymbol{x}, t), \end{aligned}$$

Algorithm 8 Branching Monte-Carlo method for the principal eigenpair of the Laplacian

```
Initialize
Fix the branching times $T_0 = 0 < T_1 < \ldots < T_k$
Pick the number of samples per branch, $N$
Pick the point from which the walks originate, $x$
Set $\hat{\pi}_{T_0} = \delta_x$ (i.e. the walks of the first branch start at $x$)

for $i = 0, \ldots, k-1$ do
    Simulate $N$ independent walks with starting points drawn from $\hat{\pi}_{T_i}$ for a time $T_{i+1} - T_i$. Denote the final positions by $\{X^{(j)}\}_{j=1,\ldots,N}$. Particles reaching the boundary are absorbed (removed).
    Let $N(i)$ be the set of the indices of the walks that were not absorbed.
    Set $\hat{\pi}_{T_{i+1}} = \frac{1}{|N(i)|}\sum_{j \in N(i)} \delta_{X^{(j)}}$.
end for

Finalize
Simulate $N$ independent walks with starting points drawn from $\hat{\pi}_{T_k}$ until they exit the domain.
Estimate $\lambda_1$ from the exit time distribution. (Equation 5.12)
Estimate $\varphi_1^*$ from the realizations $\{X^{(j)}\}_{j=1,\ldots,N}$ of the position of $X_{T_k}$. (Equation 5.13)
```

where $R_i(\boldsymbol{x}, t) = o(\exp(\lambda_i t))$. Denote the approximation of the solution to the Cauchy problem at time T_i as $\bar{u}(\boldsymbol{x}, T_i) = \mathbb{E}_{\boldsymbol{x}}[u_0(\boldsymbol{X}_t) | t < T_i]$. We know that

$$\begin{aligned}
\bar{u}(\boldsymbol{x}, T_i) &= \langle \varphi_1^*, u_0 \rangle \exp(\lambda_1 T_i) \varphi_1 + \bar{R}_1(\boldsymbol{x}, T_i) \\
\Rightarrow \varphi_1 &\approx \frac{\bar{u}(\boldsymbol{x}, T_i)}{\langle \varphi_1^*, u_0 \rangle e^{\lambda_1 T_i}} + \mathcal{O}(e^{(\lambda_2 - \lambda_1) T_i}),
\end{aligned}$$

and we also have that

$$\begin{aligned}
\bar{R}_i(\boldsymbol{x}, t) = \bar{u}(\boldsymbol{x}, t) - \sum_{j=1}^{i} \langle \varphi_j^*, u_0 \rangle \exp(\lambda_j t) \varphi_j &= \langle \varphi_{i+1}^*, u_0 \rangle \exp(\lambda_{i+1} t) \varphi_{i+1} + \bar{R}_{i+1}(\boldsymbol{x}, t) \\
\Rightarrow \varphi_{i+1} &\approx \frac{\bar{R}_{i+1}(\boldsymbol{x}, t)}{\langle \varphi_{i+1}^*, u_0 \rangle e^{\lambda_{i+1} t}} + \mathcal{O}(e^{(\lambda_{i+2} - \lambda_{i+1}) t}),
\end{aligned}$$

Suppose now that we have approximations $\bar{u}(\boldsymbol{x}, T_j)$ for $j = 1, \ldots, k$. For convenience, denote $R_0 = \langle \varphi_1^*, u_0 \rangle \exp(\lambda_1 t) \varphi_1$. Combining these equations we can write

$$\begin{aligned}
\varphi_1 &\approx \frac{\bar{u}(\boldsymbol{x}, T_k)}{\langle \varphi_1^*, u_0 \rangle e^{\lambda_1 T_k}} + \mathcal{O}(e^{(\lambda_2 - \lambda_1) T_i}), \\
\varphi_{i+1} &\approx \frac{\bar{R}_{i+1}(\boldsymbol{x}, T_{k-i})}{\langle \varphi_{i+1}^*, u_0 \rangle e^{\lambda_{i+1} T_{k-i}}} + \mathcal{O}(e^{(\lambda_{i+2} - \lambda_{i+1}) T_{k-i}}),
\end{aligned}$$

which recovers the eigenfunctions $\varphi_1, \ldots, \varphi_k$. The eigenvalues can then be recovered by writing

$$\langle \varphi_i^*, u(\boldsymbol{x}, t)\rangle \quad = \quad \langle \varphi_i^*, u_0\rangle \exp(\lambda_i t) \|\varphi_i\|^2,$$

and then using $u(\boldsymbol{x}, t)$ at two branching times to recover

$$\begin{aligned} \frac{\langle \varphi_i^*, u(\boldsymbol{x}, T_2)\rangle}{\langle \varphi_i^*, u(\boldsymbol{x}, T_1)\rangle} &= \exp(\lambda_i(T_2 - T_1)), \\ \Rightarrow \quad \lambda_i &= \log\left(\frac{\langle \varphi_i^* u, (\boldsymbol{x}, T_2)\rangle}{\langle \varphi_i^* u, (\boldsymbol{x}, T_1)\rangle}\right) / (T_2 - T_1). \end{aligned}$$

5.3.2 Dealing with degeneracy

Suppose that some eigenvalues are very poorly separated or are truly degenerate. For the purpose of illustration, suppose that the principal eigenvalue is degenerate:

$$u(\boldsymbol{x}, t) \quad = \quad \exp(\lambda_1 t)(\langle \varphi_{11}^*, u_0\rangle \varphi_{11} + \langle \varphi_{12}^*, u_0\rangle \varphi_{12}) + R_1(\boldsymbol{x}, t), \tag{5.17}$$

where φ_{11} and φ_{12} are linearly independent eigenfunctions that share the eigenvalue λ_1. First, we need to be able to detect that this is the case. Consider two initial conditions, u_0 and v_0, such that $(\langle \varphi_{11}, u_0\rangle, \langle \varphi_{12}, u_0\rangle)$ and $(\langle \varphi_{11}, v_0\rangle, \langle \varphi_{12}, v_0\rangle)$ are linearly independent. Now, using the cascading scheme with u_0 as the initial condition, we'll naively recover

$$\varphi_{1,u} = (\langle \varphi_{11}, u_0\rangle \varphi_{11} + \langle \varphi_{12}, u_0\rangle \varphi_{12}) / \exp(\lambda_1 t), \tag{5.18}$$

where $\varphi_{1,u}$ is the approximation calculated using u_0 as the initial condition. If instead we use v_0 as the initial condition, we'll recover

$$\varphi_{1,v} = (\langle \varphi_{11}, v_0\rangle \varphi_{11} + \langle \varphi_{12}, v_0\rangle \varphi_{12}) / \exp(\lambda_1 t), \tag{5.19}$$

where $\varphi_{1,v}$ is the approximation calculated using v_0 as the initial condition. If $\varphi_{1,u}$ and $\varphi_{1,v}$ are not linearly dependent, then the eigenvalue λ_1 is degenerate. In this case, we can simply orthogonalize $\varphi_{1,u}$ and $\varphi_{1,v}$ to recover φ_{11} and φ_{12} (or rather two functions that span the same subspace as φ_{11}

and φ_{12}):

$$\begin{aligned}\varphi_{11} &= \varphi_{1,u},\\ \varphi_{12} &= \varphi_{1,v} - \langle\varphi_{1,v},\varphi_{11}\rangle\varphi_{11},\end{aligned}$$

where $\varphi_{1,u}$ and $\varphi_{1,v}$ have been normalized beforehand. For numerical purposes, we can say that $\varphi_{1,u}$ and $\varphi_{1,v}$ are orthogonal if

$$|\langle\varphi_{1,u},\varphi_{1,v}\rangle| < \varepsilon_D, \tag{5.20}$$

where ε_D that sets the tolerance, with $\varepsilon_D = 0$ recovering the strict definition of orthogonality.

5.3.3 Restarted scheme with dynamic branch time choice

In order for the scheme above to be effective, it is necessary to choose the branch times very carefully so that succesive eigenfunctions can be well separated by their differing decay rates, yet not have been completely dominated by the slower decaying eigenpairs. It's impossible to optimally determine this in advance, and if we could then we'd effectively have a very good sense of the eigenvalues to begin with.

However, these differences can be assessed dynamically, and this section describes a scheme that does so at the small additional cost of restarting the walks after each additional eigenpair recovery. The idea is that once an eigenpair has been recovered, the solution history can be examined to determine when last there were not insignificant components in directions other than the currently-known dominant ones. This time is then set as the end time for the determination of the next eigenpair. If the time is set much later than this then the contribution from this eigenpair will be lost and if it is set much earlier than this, it will not be distinctly separated from the next smallest eigenpair.

First, we need to define what we mean by significant components. We'll say that significant components remain after eigenpair k if

$$\left\|\sum_{i=1}^{k}\langle\varphi_i^*,\bar{u}(\boldsymbol{x},t)\rangle\right\| / \|\bar{u}(\boldsymbol{x},t)\| < 1-\varepsilon, \tag{5.21}$$

where ε is a small parameter that defines what proportion of the solution must be in directions orthogonal to the first k eigenfunctions. We must prescribe how many branching times are taken for each eigenfunction. Call this parameter N_B. Finally, we must define a criterion for ending the determination of the first eigenpair, since there will be no reference time to work with. We choose to end the computation of the first eigenpair when

$$\|\lambda_1^{(i)} - \lambda_1^{(i-1)}\| / \|\lambda_1^{(i)}\| < \delta, \tag{5.22}$$

where $\lambda_1^{(i)}$ is the approximation to the first eigenvalue at branch i, and δ determines the tolerance for determining that the leading eigenvalue has converged. The algorithm is presented in Algorithm 9. Here $u_0^{(i)}$ denotes the initial condition used for determining the ith eigenpair and $T_j^{(i)}$ denotes the jth branch time for determining eigenpair i. The scheme is illustrated in Figure 5.1.

Algorithm 9 Restarted cascading scheme for eigenpairs of the Laplacian

```
Input: $u_0$
Output: $\varphi_0, \varphi_1, ...$
  Initialize
  Choose $\varepsilon$, $\delta$ and $N_B$.

  First eigenpair
  Choose some branch spacing $\Delta T$.
  for $i = 1, \ldots.$ do
      Step to branch time $T_i = i\Delta T$.
      Estimate $\lambda_1^{(i)}$ and check if it satisfies Equation 5.22. If yes, continue to the remaining eigenpairs.
  end for

  Remaining eigenpairs
  for $i = 2, \ldots$ do
      for $j = N_B, N_B - 1, \ldots, 1$ do
          Check if Equation (5.21) is satisfied at branch time $T_j^{(i-1)}$ of the previous eigenpair computation. If yes, set the maximum time for the current eigenpair computation to $T_{N_B}^{(i)} = T_j^{(i-1)}$ and continue to the current eigenpair computation.
      end for
      Set the initial condition $u_0^{(i)} = u_0 - \sum_{j=1}^{i-1} \langle \varphi_j^*, u_0 \rangle \varphi_j$.
      Take $N_B$ steps to branch time $T_{N_B}^{(i)}$. Estimate $\varphi_i$ and $\lambda_i$.
  end for
```

Choosing end times from eigenvalues If we have another source of approximations to the eigenvalues of the system, we can choose the branch times directly based on preserving a certain fraction of an eigenvector. That is, if the remaining fraction for the eigenvector φ_i is $w_i = \exp(-\lambda_i t_i)$

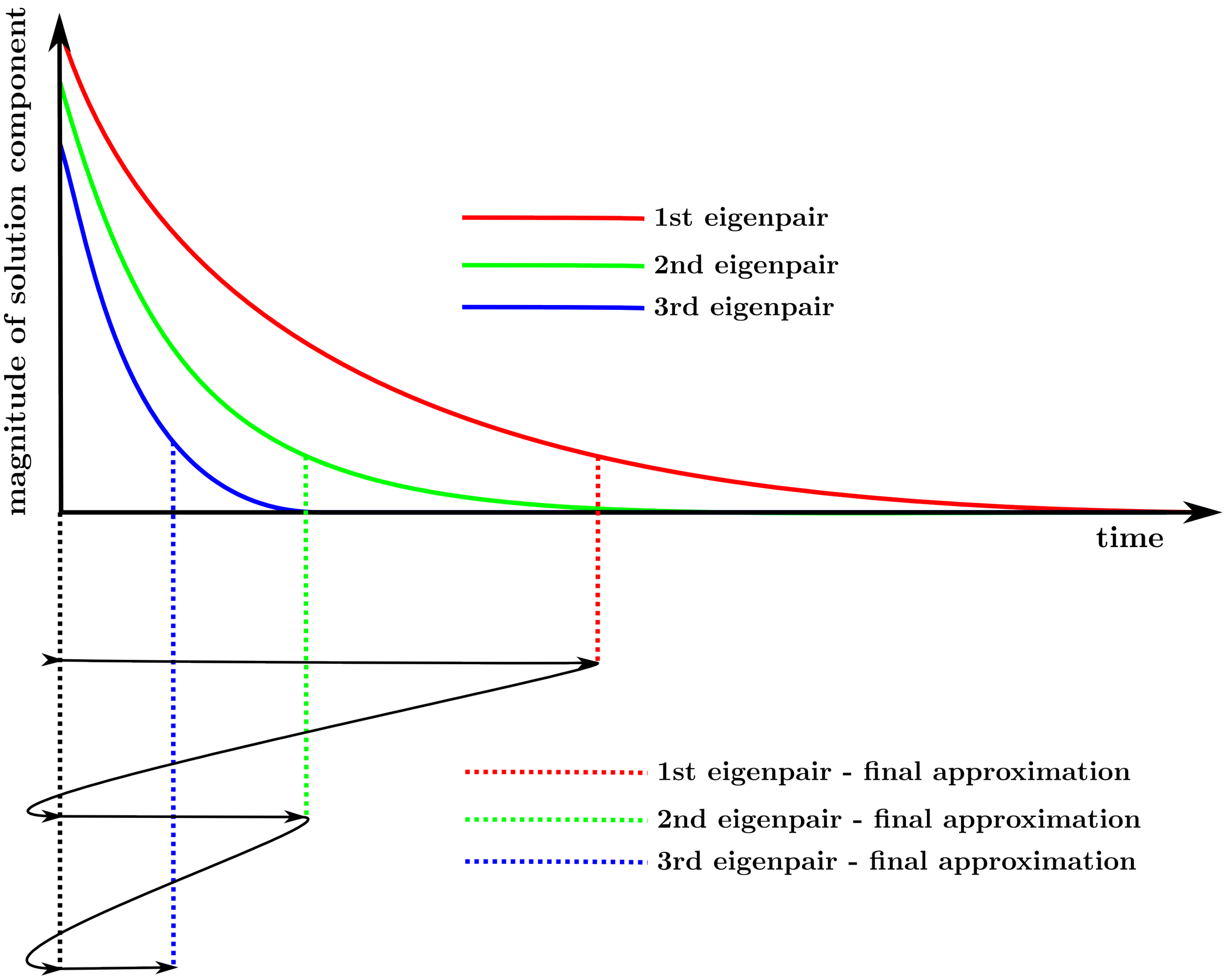

magnitude of solution component
time
1st eigenpair
2nd eigenpair
3rd eigenpair
1st eigenpair - final approximation
2nd eigenpair - final approximation
3rd eigenpair - final approximation

Figure 5.1: The restarted cascading scheme

we invert this to get the end time to recover that fraction of the ith eigenvector:

$$t_i = -\frac{\log(w_i)}{\lambda_i} \tag{5.23}$$

5.4 Generalization

The approach in Section 5.2 can be applied to any differential operator of the kind described in Equation 5.1. The following is a presentation of the approach developed in [89] for this purpose. In this case, the operator is no longer self-adjoint, and the eigenvalues are no longer strictly real. However, under certain regularity assumptions, there is a real eigenvalue λ_1 such that the real parts of all other eigenvalues are larger in magnitude. This allows us to recover the principal eigenpair in the same manner as in Section 5.2. The time dependent problem is examined first, given by

$$\frac{\partial u(\boldsymbol{x},t)}{\partial t} = \sum_{i=1}^{d} \alpha_i(\boldsymbol{x}) \frac{\partial u(\boldsymbol{x},t)}{\partial x_i} + \frac{1}{2}\sum_{i,j=1}^{d} \beta_{ij}(\boldsymbol{x}) \frac{\partial^2 u(\boldsymbol{x},t)}{\partial x_i \partial x_j} + q(\boldsymbol{x})u(\boldsymbol{x},t) \tag{5.24}$$

with $u(\boldsymbol{x},0) = u_0(\boldsymbol{x})$ and $\beta_{ij}(\boldsymbol{x}) = \sum_{k=1}^{d} \sigma_{ik}(\boldsymbol{x})\sigma_{jk}(\boldsymbol{x})$. If we call the operator

$$\mathcal{L} = \sum_{i=1}^{d} \alpha_i(\boldsymbol{x}) \frac{\partial}{\partial x_i} + \frac{1}{2}\sum_{i,j=1}^{d} \beta_{ij}(\boldsymbol{x}) \frac{\partial^2}{\partial x_i \partial x_j}, \tag{5.25}$$

then the adjoint of the operator is

$$\mathcal{L}^* = -\sum_{i=1}^{d} \frac{\partial}{\partial x_i}(\alpha_i(\boldsymbol{x})\cdot) + \frac{1}{2}\sum_{i,j=1}^{d} \frac{\partial^2}{\partial x_i \partial x_j}(\beta_{ij}(\boldsymbol{x})\cdot). \tag{5.26}$$

The algorithms in Sections 5.2 and 5.3 compute the principal eigenfunction, φ_1^*, of $\mathcal{L}^*$. If we want to compute the principal eigenfunction of $\mathcal{L}$, we use the fact that $\mathcal{L}^{**} = \mathcal{L}$, and rather apply the algorithm to $\mathcal{L}^*$ as the operator. Now, $\mathcal{L}^*$ is not in an appropriate form, but it can be converted into [89]

$$\mathcal{L}^* = \sum_{i=1}^{d} \bar{\alpha}_i(\boldsymbol{x}) \frac{\partial}{\partial x_i} + \frac{1}{2}\sum_{i,j=1}^{d} \beta_{ij}(\boldsymbol{x}) \frac{\partial^2}{\partial x_i \partial x_j} + \gamma(\boldsymbol{x}), \tag{5.27}$$

where

$$\bar{\alpha}_i(\boldsymbol{x}) = \frac{1}{2}\sum_{j=1}^{d} \frac{\partial \beta_{ij}}{\partial x_i} - \alpha_i, \tag{5.28}$$

and

$$\gamma(\boldsymbol{x}) = \frac{1}{2}\sum_{i,j=1}^{d} \frac{\partial^2}{\partial x_i \partial x_j}(\beta_{ij}(\boldsymbol{x})) - \sum_{i=1}^{d} \frac{\partial \alpha_i(\boldsymbol{x})}{\partial x_i}. \tag{5.29}$$

This is now a more general case, where the evolution equation for the adjoint is:

$$\frac{\partial u(\boldsymbol{x},t)}{\partial t} = \sum_{i=1}^{d} \bar{\alpha}_i(\boldsymbol{x}) \frac{\partial u(\boldsymbol{x},t)}{\partial x_i} + \frac{1}{2}\sum_{i,j=1}^{d} \beta_{ij}(\boldsymbol{x}) \frac{\partial^2 u(\boldsymbol{x},t)}{\partial x_i \partial x_j} + \gamma(\boldsymbol{x}) u(\boldsymbol{x},t). \tag{5.30}$$

To begin with, we solve a modified equation [89]. Denote

$$L = \sum_{i=1}^{d} \bar{\alpha}_i(\boldsymbol{x}) \frac{\partial}{\partial x_i} + \frac{1}{2}\sum_{i,j=1}^{d} \beta_{ij}(\boldsymbol{x}) \frac{\partial^2}{\partial x_i \partial x_j} \tag{5.31}$$

Suppose that γ is bounded above by a constant Γ. Now let v be the solution to

$$\begin{aligned} \frac{\partial v}{dt} &= Lv + \gamma v - \Gamma v, \\ v(0,\boldsymbol{x}) &= v_0, \end{aligned}$$

with a (zero) Dirichlet boundary condition on the cylinder $\mathbb{R}_+ \times \partial D$, that is, having zero Dirichlet boundary conditions over all time $t \geqslant 0$. The solution v can be represented by the Feynman-Kac formula:

$$v(t,x) = \mathbb{E}_x \left[v_0(\boldsymbol{X}_t) \exp\left(\int_0^t \gamma(\boldsymbol{X}_s)\,\mathrm{ds} - \Gamma t \right); t < \tau \right], \tag{5.32}$$

where τ is the exit time from the domain, and $\boldsymbol{X}$ is the process generated by L. That is, the process is governed by

$$d\boldsymbol{X}(t) = \bar{\alpha}(\boldsymbol{X}(t))dt + \sigma(\boldsymbol{X}(t))d\boldsymbol{B}(t),$$

where, as usual, σ is given by

$$\beta_{ij}(\boldsymbol{x}) = \sum_{k=1}^{d} \sigma_{ik}(\boldsymbol{x})\sigma_{jk}(\boldsymbol{x}). \tag{5.33}$$

This can also be written as [89]

$$v(t,x) = \mathbb{E}_x \left[v_0(\boldsymbol{X}_t); \int_0^t \gamma(\boldsymbol{X}_s)\,\mathrm{ds} - \Gamma t > \zeta \,\mathrm{and}\, t < \tau \right], \tag{5.34}$$

where ζ is an exponential random variable with parameter 1. That is, we first sample ζ, then run the diffusion until either inequality is violated, and stop there. Now, returning to the decay arguments from before, we know that

$$v(t,x) \backsimeq \exp(\lambda_1^*)\langle v_0, \psi_1^*\rangle \psi_1^*(\boldsymbol{x}), \tag{5.35}$$

when t is large. We also know that $\psi_1^* = \varphi_1$, and $\lambda_1^* = \lambda_1 - \Gamma$. This can now be plugged into the formulations in Sections 5.2 and 5.3.

5.5 Implementation

When implementing these schemes, the vast majority of the work comes in propagating the Ito Diffusions. Here we define a subroutine that carries out Ito diffusions in 2D of an initial condition $\boldsymbol{u}^{(0)}$, given Ito coefficients $\boldsymbol{\sigma}$ and $\boldsymbol{\alpha}$ over the whole domain, a diffusion time T, a timestep Δt and a number of samples n_{samples}. This basic subroutine is labeled `ItoDiffuse`, and is given in Algorithm 10. A naive implementation of this subroutine on GPUs is relatively straightforward, but not optimal. In the following subsections, we describe modifications to this approach and implementation specifics that drastically improve its efficiency, based on the considerations described in Chapter 2.

5.5.1 Pseudorandom number generator

In choosing a psuedorandom number generator to generate the samples of $\mathcal{N}(0,1)$, we need to evaluate it based on the following properties, in order of decreasing importance:

1. The correlation between successive numbers
2. The memory footprint for the state of the generator
3. The number of FLOPs per iteration

Having a low correlation between successive numbers is a basic requirement for any good pseudorandom number generator. Having a high memory footprint has huge consequences for performance, as we need to keep the number of registers per thread as low as possible. The number of FLOPs per iteration is less important, as the performance is generally constrained by memory operations. However, with very careful management of memory, it may again be relevant. An ideal candidate that satisfies all three criteria is the Philox [90] counter-based PRNG, which has been demonstrated to be highly efficient on GPUs for applications as varied as lattice QCD [91] and genetic algorithms

Algorithm 10 `ItoDiffuse`

```
Input: $\boldsymbol{u}^{(0)}$,$\boldsymbol{\sigma}$,$\boldsymbol{\alpha}$,T,$\Delta t$,$n_{\text{samples}}$
Output: $\boldsymbol{u}^{(T)}$
  $mx$ is the number of grid points in the $x$ dimension
  $my$ is the number of grid points in the $y$ dimension
  $\boldsymbol{u}^{(T)} = \mathbf{0}$
  for $i = 0, ..., mx - 1$ do
      for $j = 0, ..., my - 1$ do
          for $k = 0, ..., n_{\text{samples}} - 1$ do
              $\boldsymbol{x}^{(0)} = x_{ij}$
              $u^{(0)} = \boldsymbol{u}_{ij}^{(0)}$
              $t = 0$
              while $t < T$ do
                  $\boldsymbol{x} = \boldsymbol{x} + \boldsymbol{\alpha}(\boldsymbol{x})\Delta t + \boldsymbol{\sigma}(\boldsymbol{x})\sqrt{\Delta t}\mathcal{N}(0, 1)$
                  $t = t + \Delta t$
                  if $\boldsymbol{x} \notin \Omega$ then
                      Break
                  end if
              end while
              if $\boldsymbol{x} \in \Omega$ then
                  Determine $ix, iy$ for the nearest grid point $x_{ix,iy}$ to $\boldsymbol{x}$
                  $\boldsymbol{u}_{ix,iy}^{(T)} = u^{(0)}/n_{\text{samples}}$
              end if
          end for
      end for
  end for
```

[92]. In this work we use the Philox4x32-10 variant.

5.5.2 Memory management

A potential bottleneck in the code is the reading of the coefficients and source terms from global GPU memory to local thread memory. This is mitigated with a system in which coefficients from global memory are loaded into shared block memory in a radius around the points from which walks are emanating. Given the size of this block, an appropriate timestep size and/or number of timesteps is chosen for the walks to remain in the region, using the approach described in Section 5.1.2.

Assigning threads to blocks The first step is to set the main block dimensions, $b_x \times b_y$, which are the dimensions of a standard block in the interior. These are chosen to be as close to square as possible, while satisfying $b_x \times b_y = n_{\text{threads}}$. The second step is to decompose the domain into blocks of this size, with smaller blocks for edge and corner regions if necessary. A method for achieving these steps is presented in Algorithm 11, and the result of the method method is illustrated in Figure 5.2 for a domain with 11×9 interior nodes and $n_{\text{threads}} = 32$.

Algorithm 11 Block decomposition for Ito Diffusions

```
Input: $n_{\text{threads}}$, $mx$, $my$
  Set block size
  $b_x = \lceil \sqrt{n_{\text{threads}}} \rceil$ is the block size in the $x$ dimension
  while $n_{\text{threads}} \mod b_x \neq 0$ do
      $b_x = b_x + 1$
  end while

  Block decomposition
  for $j_{\min} = 0, b_y, 2b_y, ...$ do
      $j_{\max} = \min(j_{\min} + b_y - 1, my - 1)$
      for $i_{\min} = 0, b_x, 2b_x, ...$ do
          $i_{\max} = \min(i_{\min} + b_x - 1, mx - 1)$
          Add a new block with extents $[i_{\min}, i_{\max}] \times [j_{\min}, j_{\max}]$.
      end for
  end for
```

Allocating shared memory to blocks With threads assigned to blocks, the next task is to load coefficients and source terms into shared memory around these blocks. Suppose that we have sufficient shared memory to load these variables for n_{shared} grid points into a block of shared memory of dimension $s_x \times s_y$. We simply take the thread block dimensions computed by Algorithm 11,

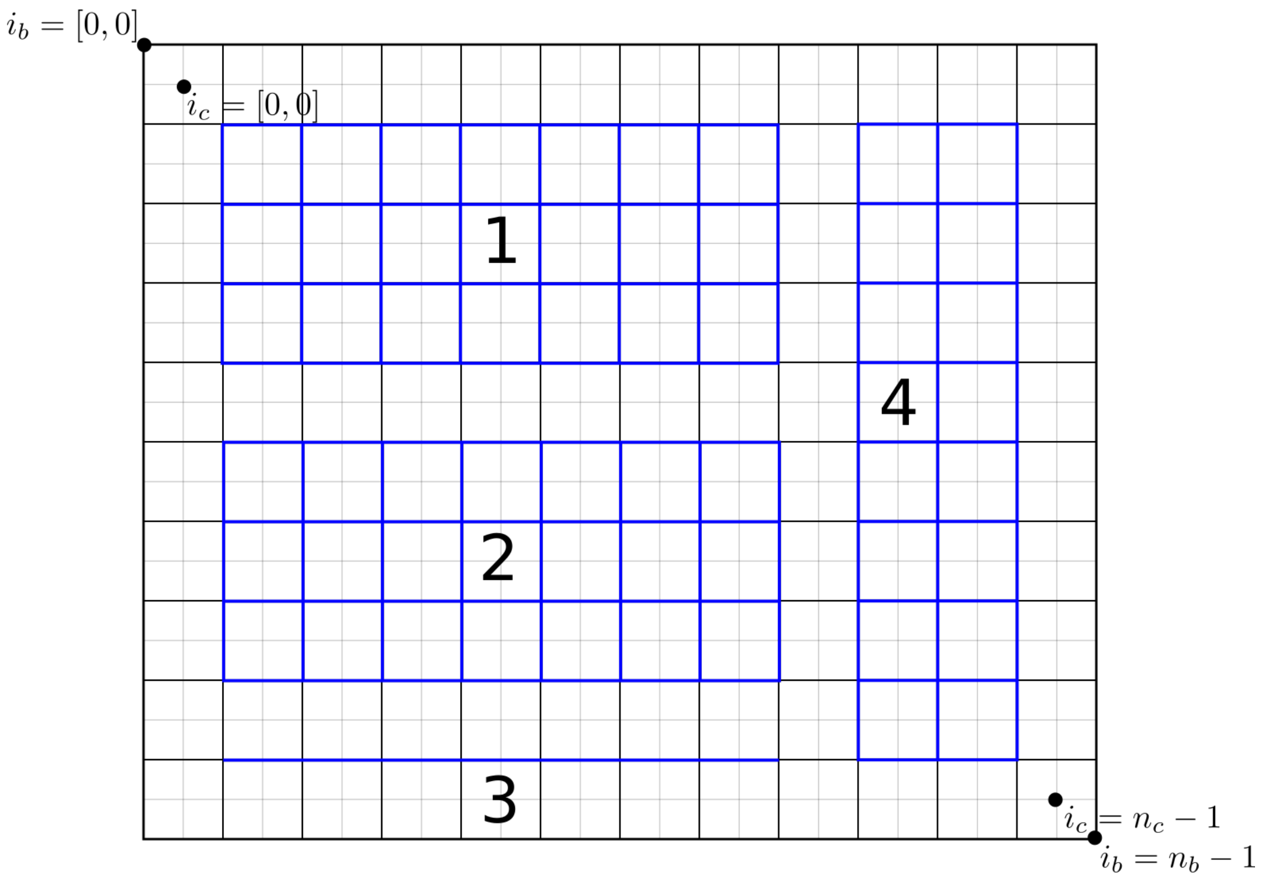


Figure 5.2: The block assignment scheme

and extend them until we run out of shared memory. This process is given in Algorithm 12. This approach applied to block 2 in Figure 5.2, with $n_{\text{shared}} = 85$, is shown in Figure 5.3.

Algorithm 12 Shared memory allocations for Ito Diffusions

```
Input: $i_{\min}$, $i_{\max}$, $j_{\min}$, $j_{\max}$, $mx$, $my$
  Initialize shared block dimensions to thread block dimensions
  $k_{\min} = i_{\min}$
  $k_{\max} = i_{\max}$
  $l_{\min} = j_{\min}$
  $l_{\max} = j_{\max}$

  Extend shared block dimensions
  $n_s = (k_{\max} - k_{\min} + 1) \times (l_{\max} - l_{\min} + 1)$
  while $n_s \leq n_{\text{shared}}$ do
      Decrement/Increment the next one of $k_{\min}, k_{\max}, l_{\min}, l_{\max}$
      $n_s = (k_{\max} - k_{\min} + 1) \times (l_{\max} - l_{\min} + 1)$
  end while
```

Copying coefficients from global to shared memory When copying coefficients from global memory to shared memory, care needs to be taken to avoid bank conflicts (see 2.2.1) when reading. This is achieved by having thread n read into the shared memory indices given by

$$I_{\text{shared}} = \{n, n + n_{\text{threads}}, n + 2n_{\text{threads}}, ..., n + kn_{\text{threads}}, \} \tag{5.36}$$

where k is the total number of reads done by the thread. With 32 banks, this gives the bank indices that the thread reads from as

$$I_{\text{bank}} = \{n \bmod 32, n + n_{\text{threads}} \bmod 32, n + 2n_{\text{threads}} \bmod 32, ..., n + kn_{\text{threads}} \bmod 32\}, \tag{5.37}$$

Since n_{threads} should be a factor of 32, this simplifies to:

$$I_{\text{bank}} = \{n \bmod 32, ..., n \bmod 32\}. \tag{5.38}$$

This guarantees that no successive 32 threads will have bank conflicts.

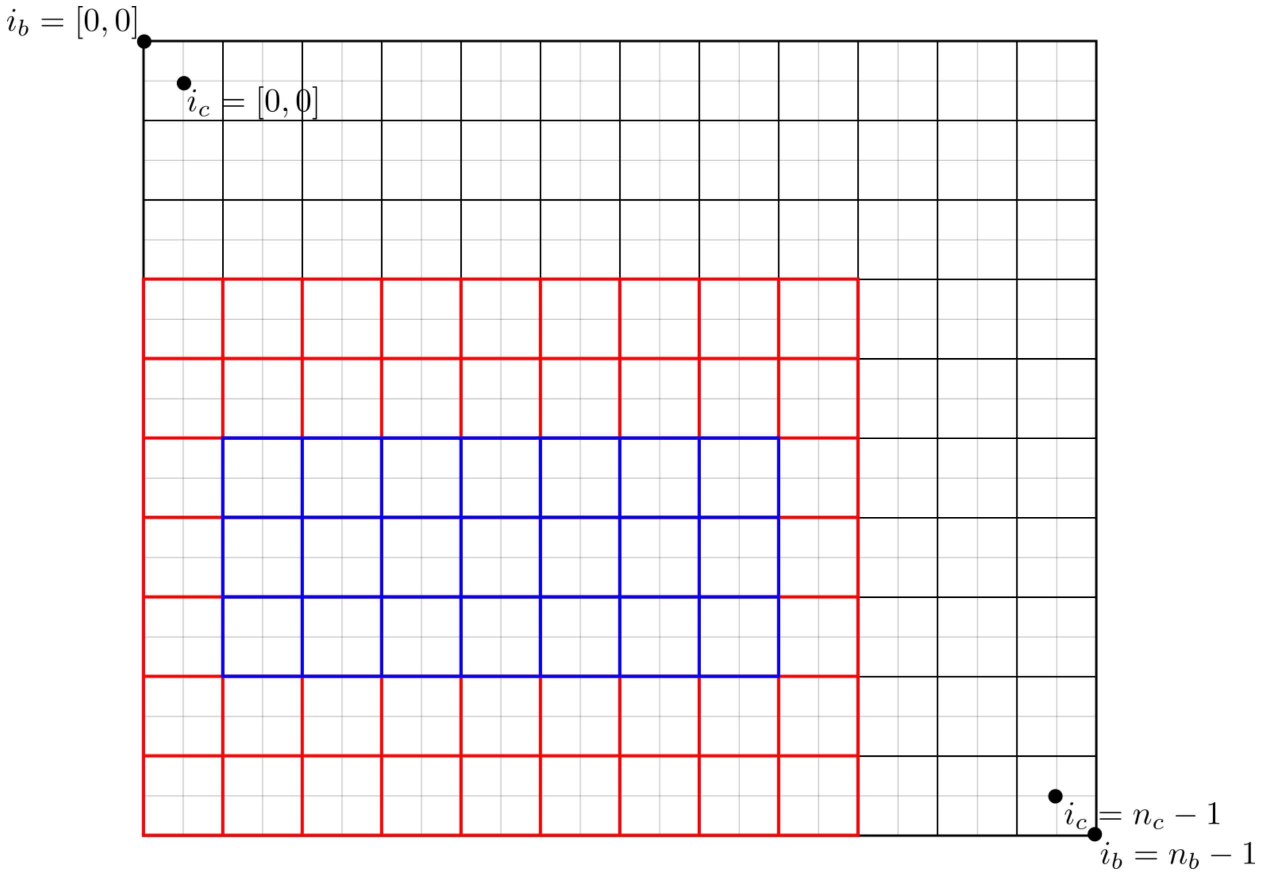
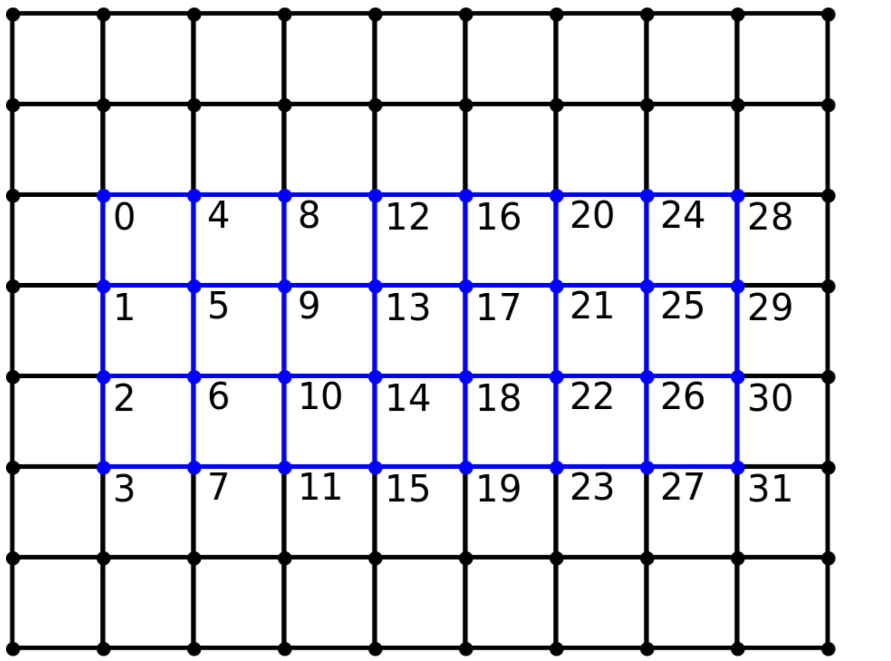


Figure 5.3: The shared memory assignment scheme. Blue: dimensions of the thread block. Red: dimensions of the shared memory.



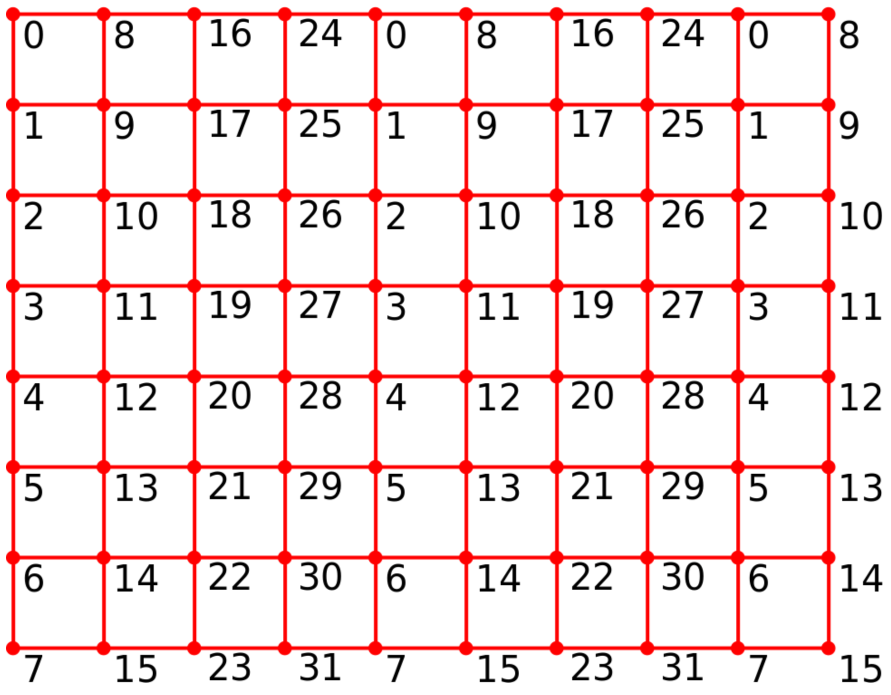


Figure 5.4: The scheme for reading coefficients from global memory to shared memory

5.6 Multiscale Random Walks

For the cascading scheme we initially wish to simulate the long-time evolution of the Cauchy problem. If we use a small timestep and the full resolution of the coefficients in the domain, this first simulation can possibly take a very long time. Since our aim is to recover the lowest frequency modes of the operator first, a sensible step in mitigating the long simulation time is to initially propagate the random walks on a coarser grid, with a larger time-step, before taking smaller timesteps on successively finer grids. This approach is inspired by multigrid, so we use similar nomenclature. We label the quantities on the finest grid by $\boldsymbol{u}^h, \boldsymbol{\sigma}^h, \boldsymbol{\alpha}^h, T^h, \Delta t^h$, and the quantities on the ith coarsened grid by $\boldsymbol{u}^{2^ih}, \boldsymbol{\sigma}^{2^ih}, \boldsymbol{\alpha}^{2^ih}, T^{2^ih}, \Delta t^{2^ih}$. The total number of grids, including the finest one, is labeled as L. The restriction or prolongation operations that transfer variables from one grid to another are labeled as I_{mh}^{nh}, where mh is the refinement level being transferred from, and nh is the refinement level being transferred to. The modification of the random walks to fit this multiscale approach is presented in Algorithm 13.

Algorithm 13 Multiscale Ito Diffusion

```
Choose a decomposition $T = \sum_{i=0}^{L-1} T^{2^ih}$.
Set timesteps $\Delta t^h, \cdots, \Delta t^{2^ih}$.
for i=0,..,$L-2$ do
    $m = 2^i$, $n = 2^{i+1}$
    $\boldsymbol{\sigma}^{nh} = I_{mh}^{nh}\boldsymbol{\sigma}^{mh}$
    $\boldsymbol{\alpha}^{nh} = I_{mh}^{nh}\boldsymbol{\alpha}^{mh}$
end for
$\boldsymbol{u}_0^{2^{L-1}h} = \left(\prod_{i=0}^{L-2} I_{2^ih}^{2^{i+1}h}\right)\boldsymbol{u}_0^h$
for i=$L-1$, $L-2$, ..., 0 do
    $m = 2^i$, $n = 2^{i-1}$
    $\boldsymbol{u}^{mh} = \texttt{ItoDiffuse}(\boldsymbol{u}_0^{mh}, \boldsymbol{\sigma}^{mh}, \boldsymbol{\alpha}^{mh}, T^{mh}, \Delta t^{mh})$
    $\boldsymbol{u}_0^{nh} = I_{mh}^{nh}\boldsymbol{u}^{mh}$
end for
```

5.6.1 Restriction and prolongation in 2D

The restriction operation is a nearest neighbor averaging defined by

$$v_{i,j} = \frac{1}{8}(4u_{2i,2j} + u_{2i-1,2j} + u_{2i+1,2j} + u_{2i,2j-1} + u_{2i,2j+1}), \tag{5.39}$$

where v is the vector of averaged values, with $i = 0, \ldots, \frac{m}{2} - 1$ and $j = 0, \ldots, \frac{n}{2} - 1$. The averaging matrix then has the non-zero entries

$$\begin{aligned}
C_{H,I} &= \frac{1}{2}, \\
C_{H,I_{2i-1}} &= \frac{1}{8}, \\
C_{H,I_{2i+1}} &= \frac{1}{8}, \\
C_{H,I_{2j-1}} &= \frac{1}{8}, \\
C_{H,I_{2j+1}} &= \frac{1}{8},
\end{aligned}$$

where now the definition for the relabeled index H is $H_{ij} = i + \frac{m}{2}j$, where the indices are i and j if omitted. The matrix now has the following structure:

$$C_h^{2h} = \frac{1}{8}\begin{pmatrix} 4 & 1 & 0 & \cdots & 0 & 1 & 0 & \ldots & 0 & 1 & 0 & \ldots & 0 & \ldots \\ \ddots & \ddots & \ddots & \ddots & \ddots & \ddots & \ddots & \ddots & \ddots & \ddots & \ddots & \ddots & \ddots & \ddots \\ \ddots & \ddots & \ddots & \ddots & \ddots & \ddots & \ddots & \ddots & \ddots & \ddots & \ddots & \ddots & \ddots & \ddots \end{pmatrix}. \tag{5.40}$$

The prolongation operation is a linear interpolation defined by

$$\begin{aligned}
u_{2i,2j} &= v_{i,j}, \\
u_{2i+1,2j} &= \frac{1}{2}(v_{i,j} + v_{i+1,j}), \\
u_{2i,2j+1} &= \frac{1}{2}(v_{i,j} + v_{i,j+1}), \\
u_{2i+1,2i+j} &= \frac{1}{4}(v_{i,j} + v_{i+1,j} + v_{i,j+1} + v_{i+1,j+1}).
\end{aligned}$$

The interpolation matrix then has the non-zero entries

$$\begin{aligned}
C_{I,H} &= 1, \\
C_{I_{i-1},H} &= \frac{1}{2}, \\
C_{I_{i+1},H} &= \frac{1}{2}, \\
C_{I_{j-1},H} &= \frac{1}{2}, \\
C_{I_{j+1},H} &= \frac{1}{2}, \\
C_{I_{i-1,j-1},H} &= \frac{1}{4}, \\
C_{I_{i+1,j-1},H} &= \frac{1}{4}, \\
C_{I_{i-1,j+1},H} &= \frac{1}{4}, \\
C_{I_{i+1,j+1},H} &= \frac{1}{4}.
\end{aligned}$$

The matrix now has the following structure:

$$C_{2h}^{h} = \frac{1}{4}\begin{pmatrix}
4 & 0 & \ddots & \ddots & \ddots & \ddots & \ddots & \ddots & \ddots & \ddots \\
2 & 2 & \ddots & \ddots & \ddots & \ddots & \ddots & \ddots & \ddots & \ddots \\
0 & 4 & 0 & \ddots & \ddots & \ddots & \ddots & \ddots & \ddots & \ddots \\
0 & 2 & 2 & \ddots & \ddots & \ddots & \ddots & \ddots & \ddots & \ddots \\
\ddots & \ddots & \ddots & \ddots & \ddots & \ddots & \ddots & \ddots & \ddots & \ddots \\
\ddots & \ddots & 2 & 0 & \dots & \dots & 0 & 2 & 0 & \ddots \\
\ddots & \ddots & 1 & 1 & 0 & \dots & 0 & 1 & 1 & \ddots \\
\ddots & \ddots & \ddots & \ddots & \ddots & \ddots & \ddots & \ddots & \ddots & \ddots \\
\ddots & \ddots & \ddots & \ddots & \ddots & \ddots & \ddots & \ddots & \ddots & \ddots
\end{pmatrix}. \tag{5.41}$$

5.7 Results

In this section we present results for the restarted cascading eigensolver (Algorithm 9) in 2D. The operator whose eigenvectors are being solved for arises from the 2D case of the general inhomogeneous diffusion problem Equation (1.3), given by:

$$\begin{aligned} -\nabla \cdot (K(\boldsymbol{x})\nabla u(\boldsymbol{x})) = f(\boldsymbol{x}) \quad &\text{in} \quad \Omega \subset \mathbb{R}^2, \\ u(\boldsymbol{x}) = g(\boldsymbol{x}) \quad &\text{on} \quad \Gamma_D, \\ K(\boldsymbol{x})\nabla u(\boldsymbol{x}) \cdot \boldsymbol{n} = h(\boldsymbol{x}) \quad &\text{on} \quad \Gamma_N, \\ \Gamma_D \cup \Gamma_N \quad &= \quad \partial\Omega, \\ \Gamma_D \cap \Gamma_N \quad &= \quad \varnothing, \end{aligned} \tag{5.42}$$

In the notation of the Feynman Kac formulation (see Equation (5.1)), this corresponds to

$$\begin{aligned} \beta_{ij}(\boldsymbol{x}) &= 2\delta_{ij}K(\boldsymbol{x}) \\ \alpha_i &= \frac{\partial K}{\partial x_i} \\ q(\boldsymbol{x}) &= 0. \end{aligned}$$

In the following sections, the method is applied on the domain $\Omega = [0,1] \times [0,1]$, and the effect of various parameter choices is investigated. In Sections 5.7.1-5.7.4, the problem is defined by

$$\begin{aligned} K(\boldsymbol{x}) &= 1 \\ f(\boldsymbol{x}) &= 1 \\ g(\boldsymbol{x}) &= x \\ \Gamma_D &= \partial\Omega, \end{aligned} \tag{5.43}$$

and the method is applied to the domain discretized with 256×256 cells. For all of the comparisons, the initial condition for the cascading scheme is given by the residual from a concurrently running linear solver, as this is how the eigensolver is used in the hybrid method. In Section 5.7.5, the method is applied to a moderately complex domain.

5.7.1 Decay cutoff times

In this section we investigate the effect of the fraction of eigenvector decay at which the method restarts. This is the parameter ϵ in Equation (5.21). If ϵ is set too small, then only the very slowest decaying eigenvectors remain, and information about the remainder of the spectrum is lost, as shown in Figure 5.5 for $\epsilon = 0.01$. On the other hand, if ϵ is too large, the highest frequency eigenvectors do not decay quickly enough, and no information is gathered about the spectrum at all, as illustrated in Figure 5.6. If ϵ is in a moderate range, of approximately $0.3 - 0.7$, the eigenvectors are recovered by ascending eigenvalue magnitude in the low end of the spectrum. This is illustrated in Figure 5.7.

5.7.2 Timestep size

In this section we investigate the effect of the timestep on the eigenvectors computed by the method. The timestep is not set directly in the method, but is rather determined by the number of desired substeps within in a shared memory block, and the allowed probability of random walks exiting that shared memory block before completing the full duration of the given walk (see Section 5.1.2). We fix the number of substeps at 4, and vary the exit probability, which we label p. If p is too high, walks exit their shared memory region too early, and are excluded from the full Feynman-Kac evolution. This leads to clear aliasing of the regions, as shown for $p = 0.5$ in Figure 5.8 and for $p = 0.1$ in Figure 5.9. This effect disappears or is negligible for sufficiently small p, as shown for $p = 0.0001$ in Figure 5.10.

5.7.3 Number of samples

In this section we investigate the effect of the number of samples taken on the eigenvectors computed by the method. As expected, increasing the number of samples reduces the noise in the eigenvectors, as shown for 4 samples in Figure 5.11, 16 samples in Figure 5.12, and 128 samples in Figure 5.13.

5.7.4 Filtering

An alternative (or supplement) to additional sampling is filtering to reduce high frequency noise. The effect of a nearest neighbor averaging filter is shown for 1 iteration in Figure 5.14, 4 iterations in Figure 5.15, and 16 iterations in Figure 5.16. It's possible for too many filtering iterations to wash out useful information however, which is shown to slow convergence when the eigenvectors are used in a deflation preconditioner (Section 6.4.1).

5.7.5 Complex domains

It is also possible to implement this approach on complex domains. Here we show an implementation on a moderately complex domain based on the The University of North Carolina (UNC) logo (Figure 1.1). This domain is particularly challenging for this approach, as the near

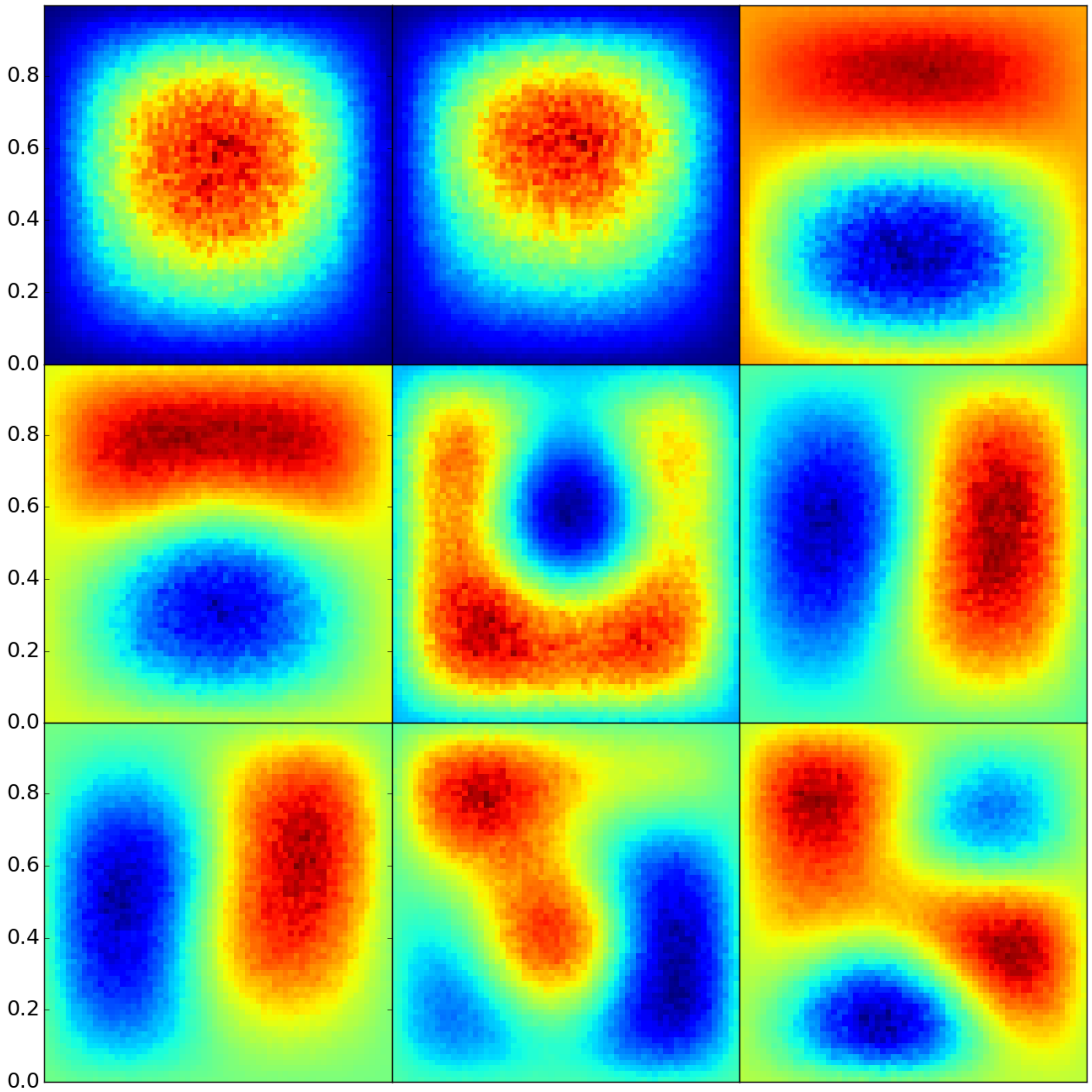


Figure 5.5: Eigenvectors from the restarted cascading eigensolver, with decay fraction $\epsilon = 0.01$

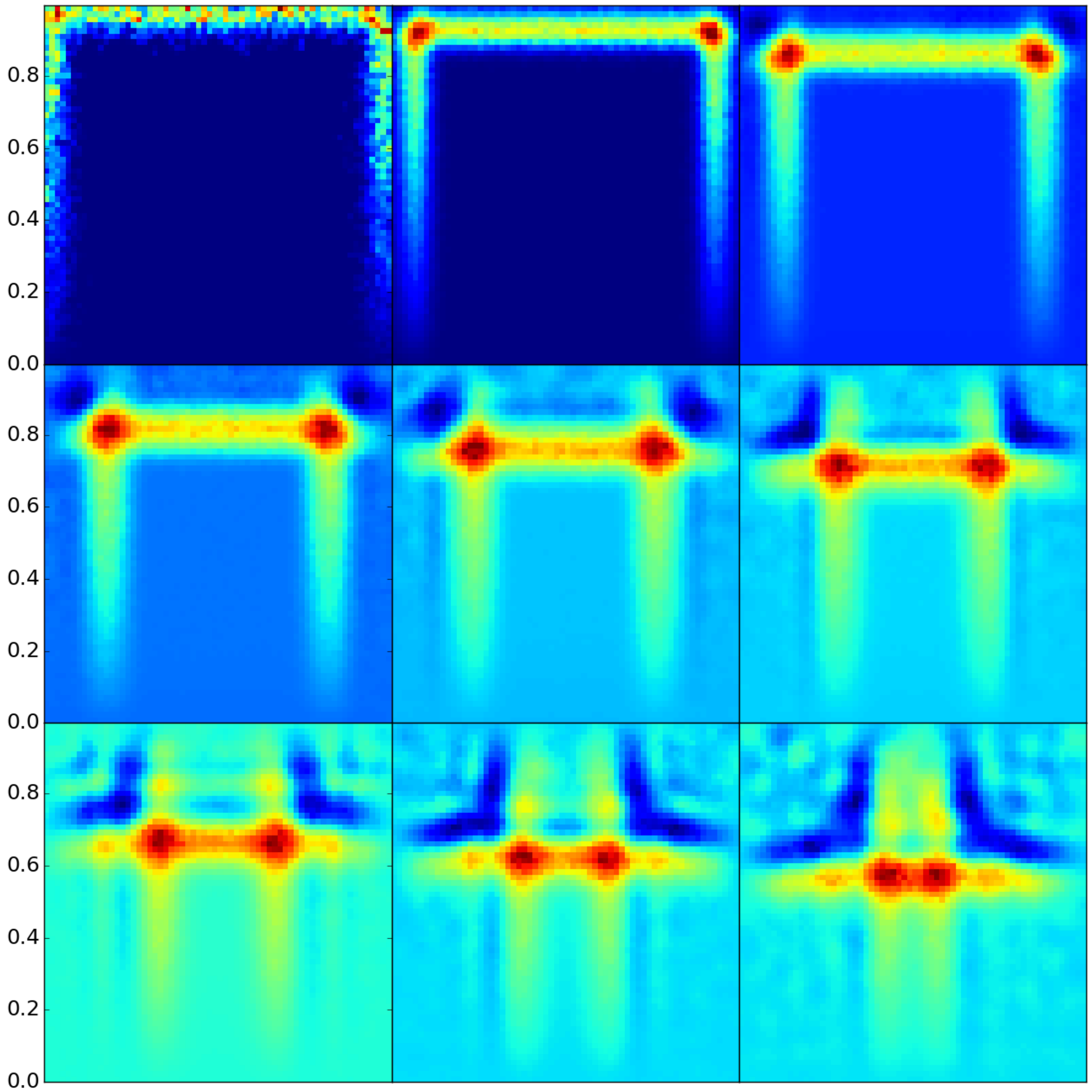


Figure 5.6: Eigenvectors from the restarted cascading eigensolver, with decay fraction $\epsilon = 0.99$

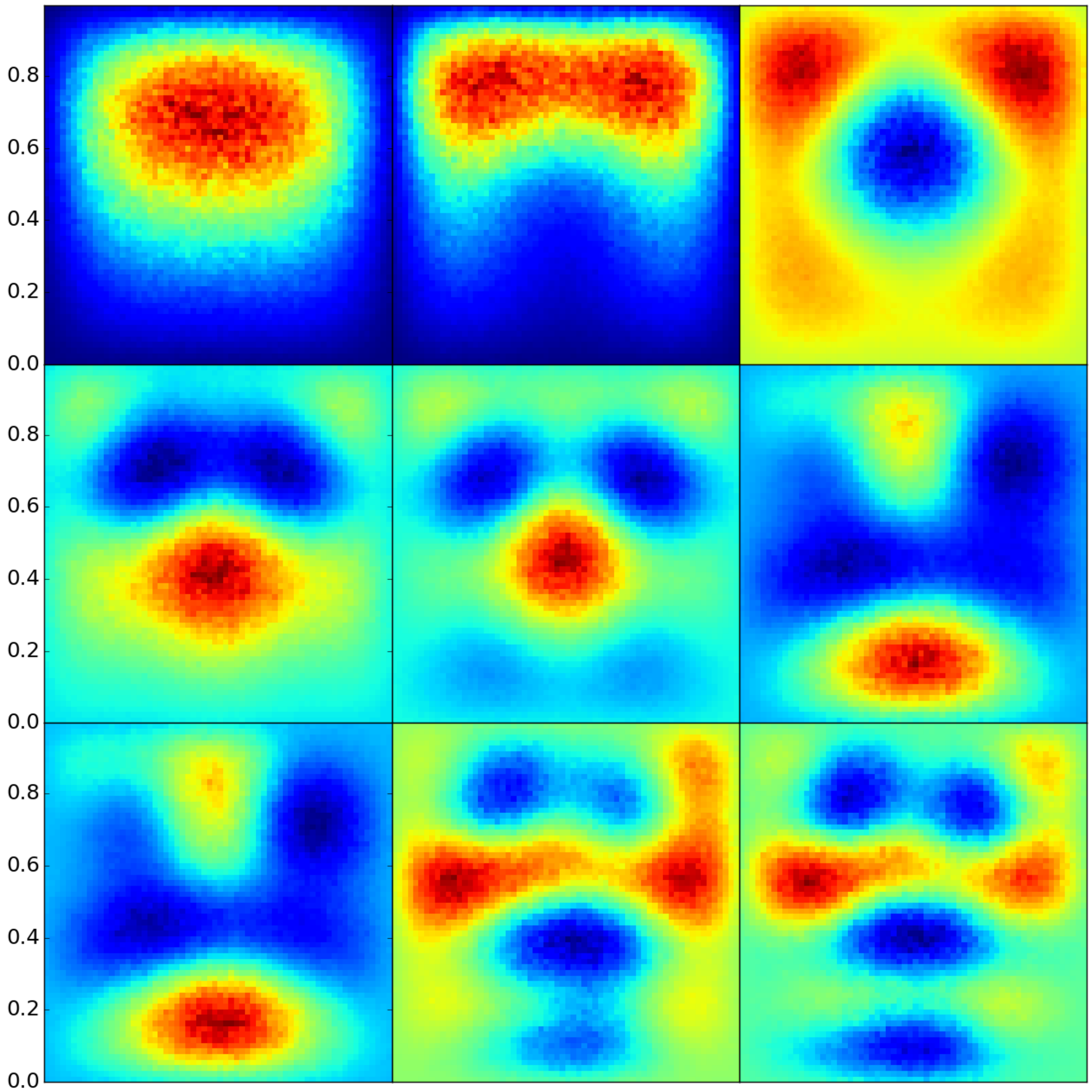


Figure 5.7: Eigenvectors from the restarted cascading eigensolver, with decay fraction $\epsilon = 0.4$

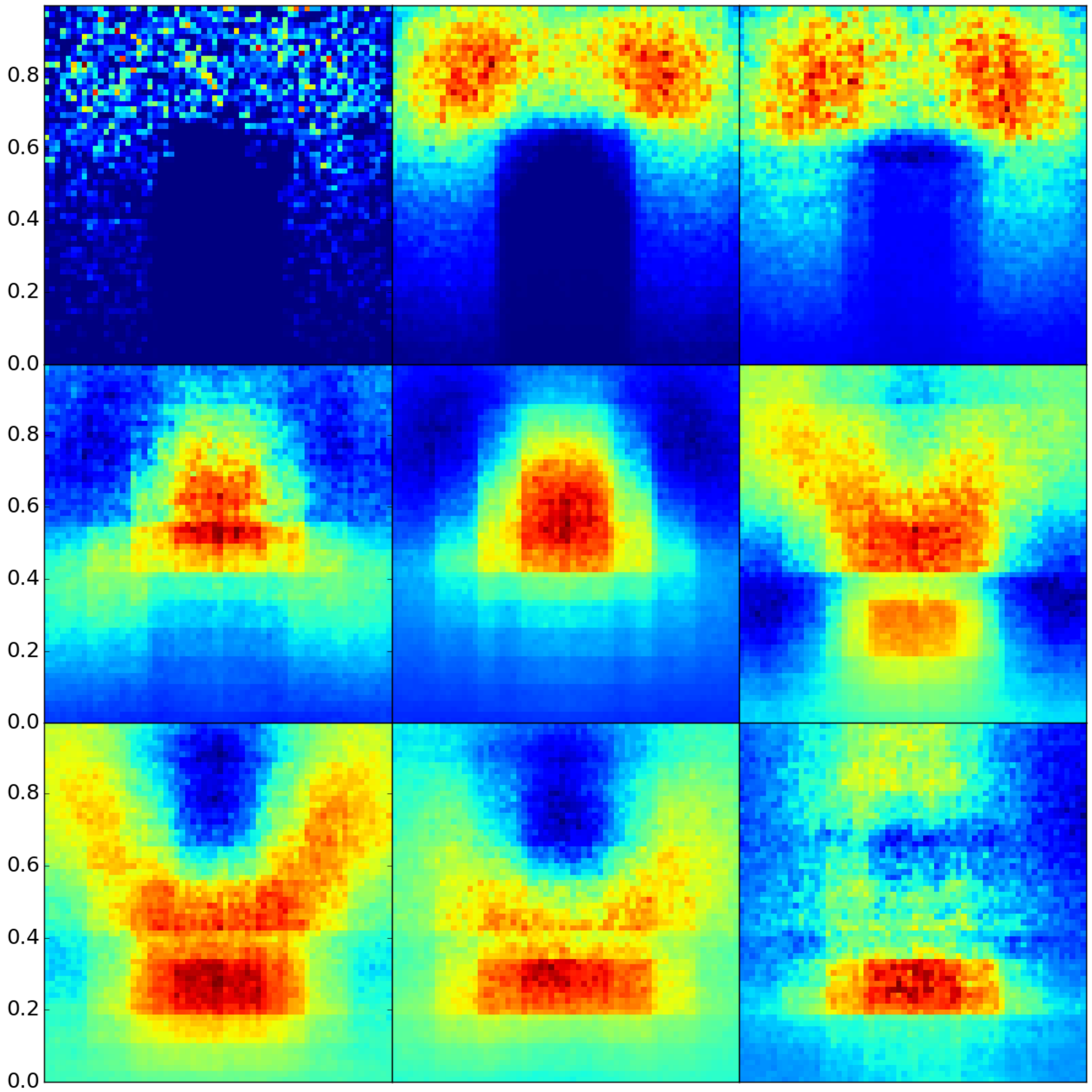


Figure 5.8: Eigenvectors from the restarted cascading eigensolver, with exit probability $p = 0.5$

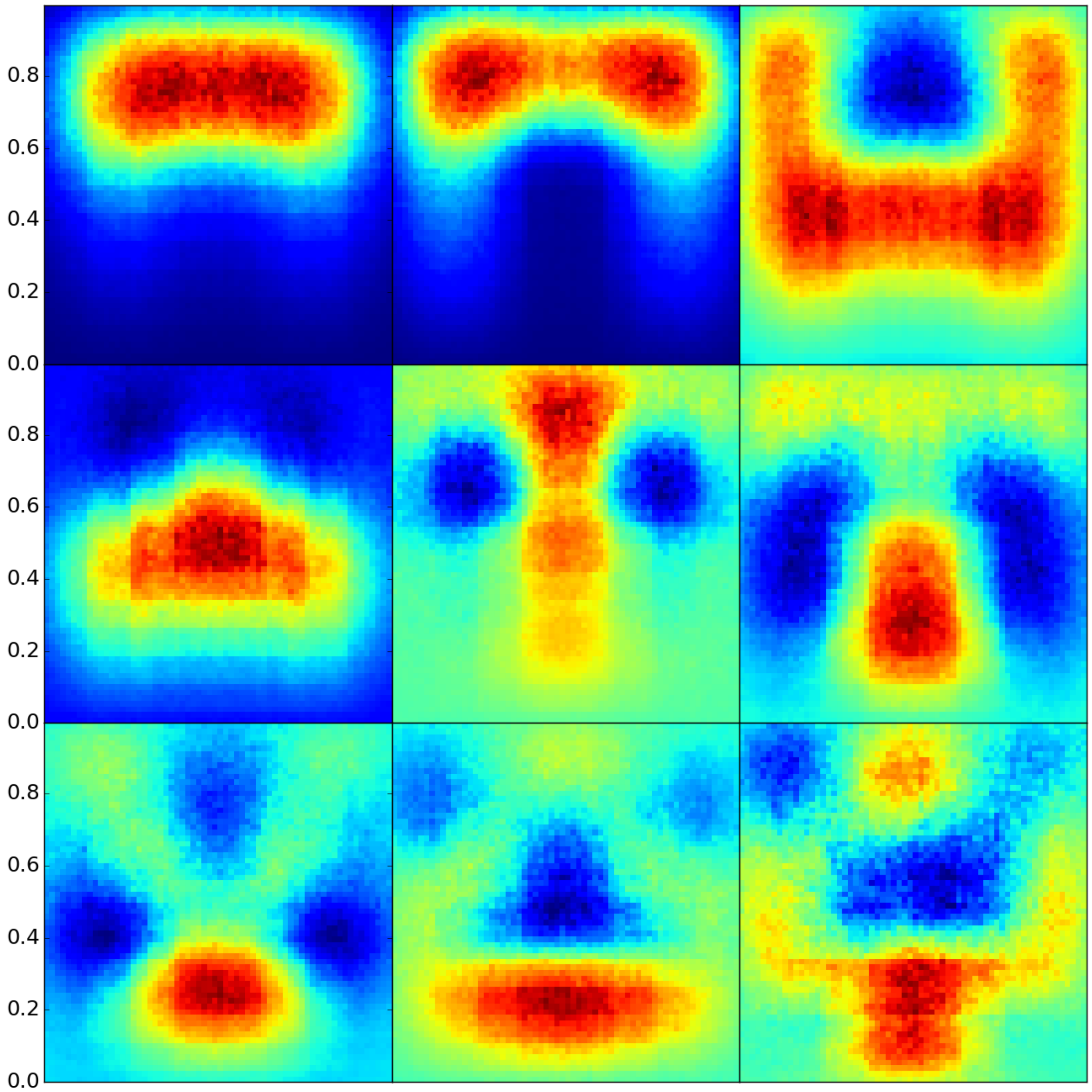


Figure 5.9: Eigenvectors from the restarted cascading eigensolver, with exit probability $p = 0.1$

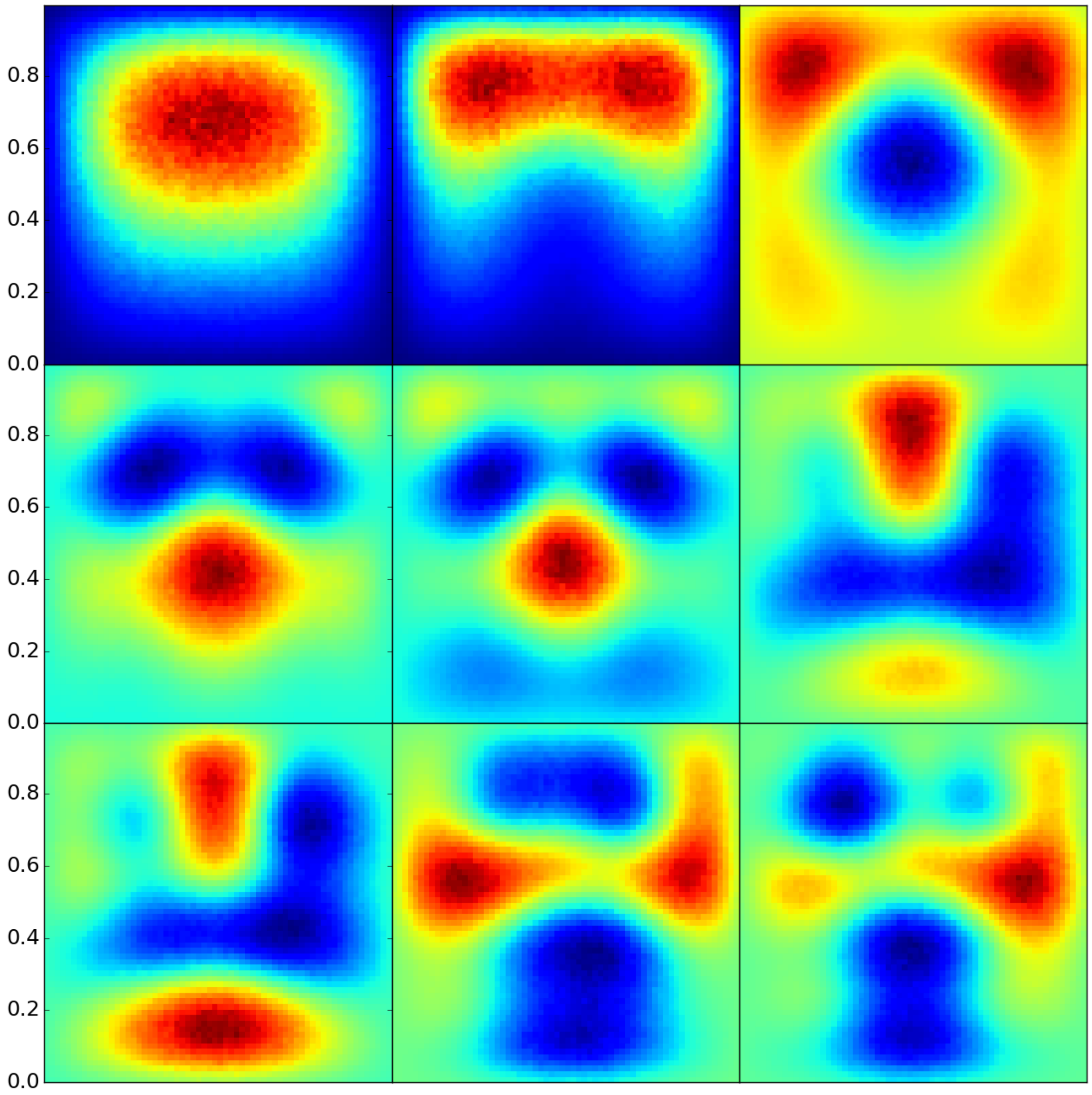


Figure 5.10: Eigenvectors from the restarted cascading eigensolver, with exit probability $p = 0.0001$

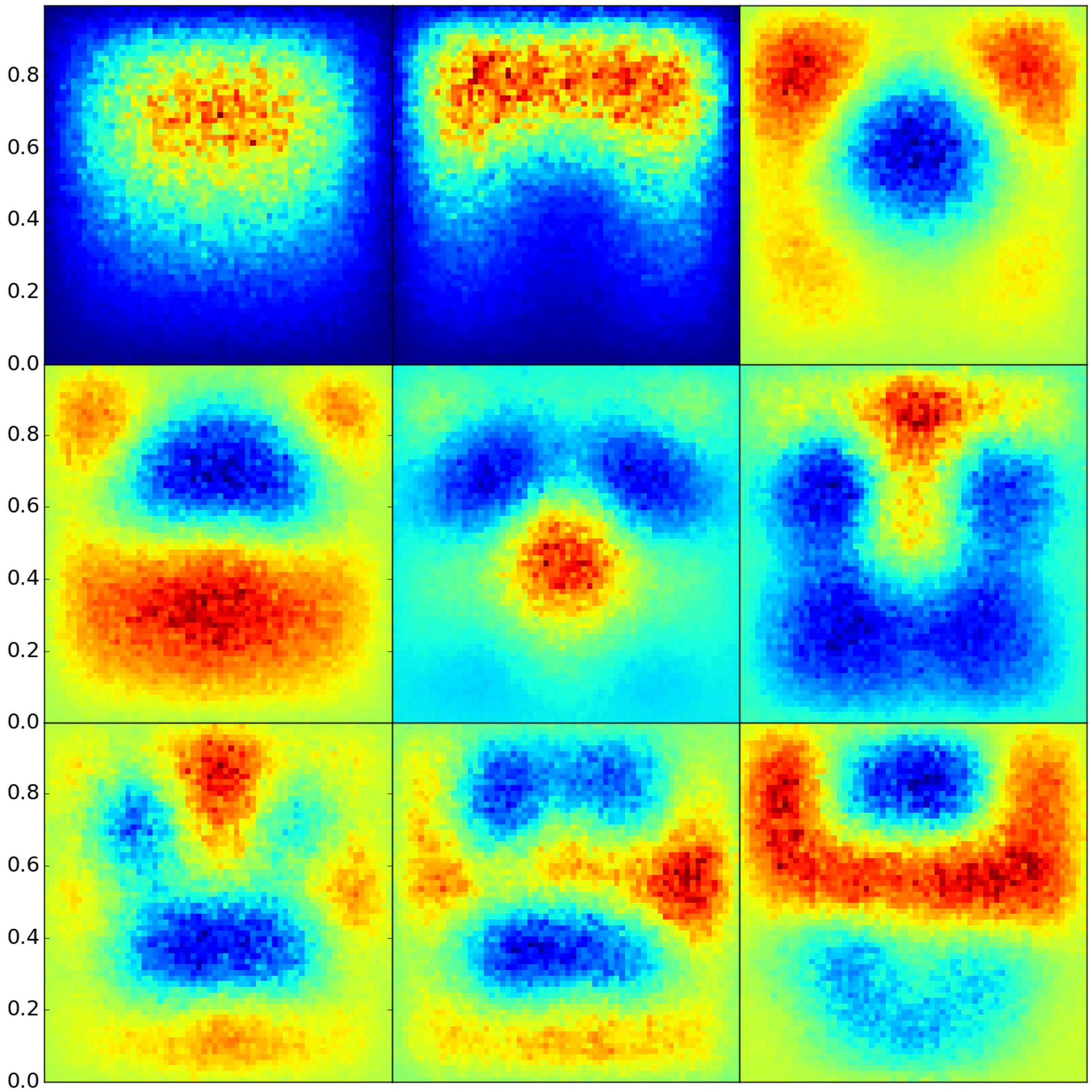


Figure 5.11: Eigenvectors from the restarted cascading eigensolver, with 4 samples

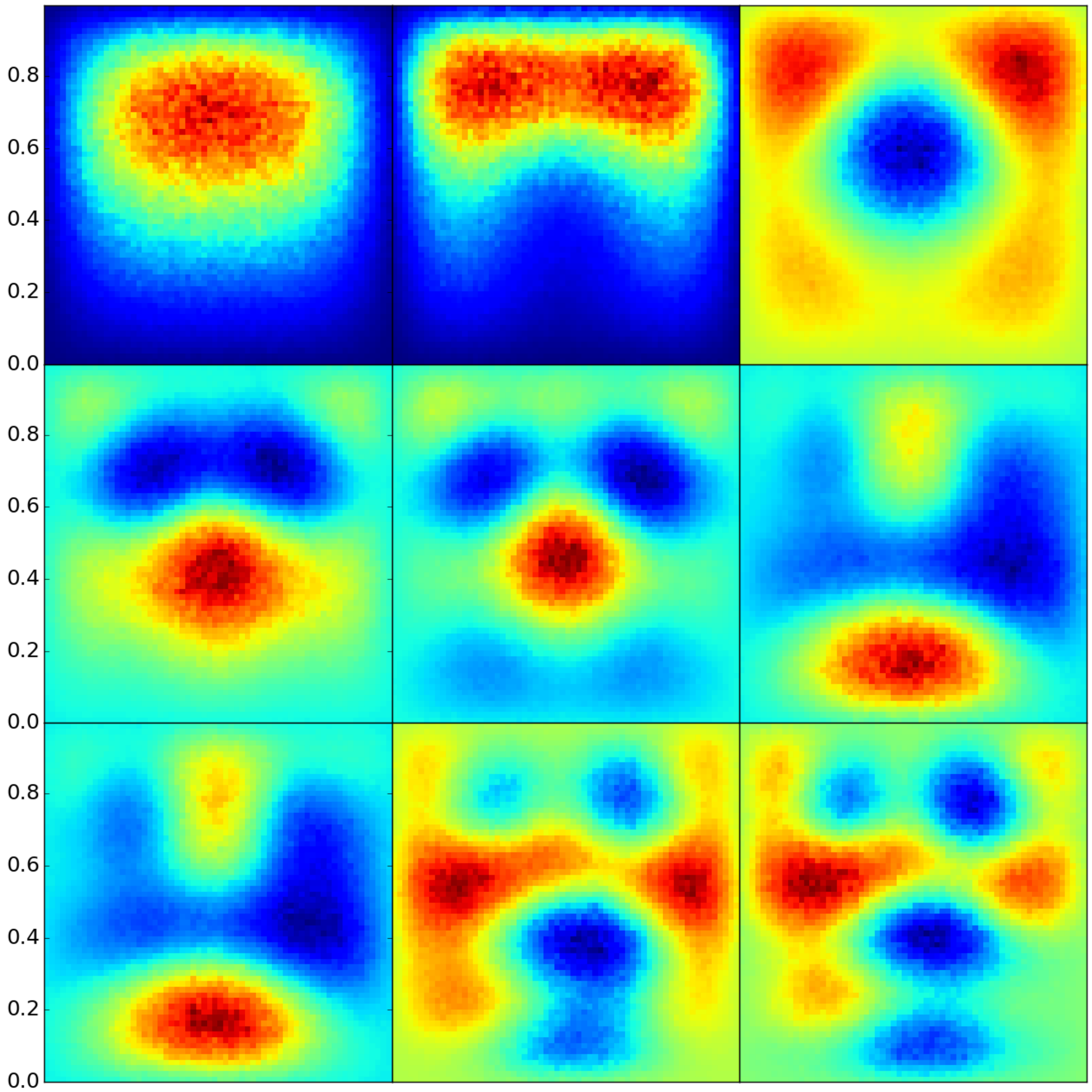


Figure 5.12: Eigenvectors from the restarted cascading eigensolver, with 16 samples

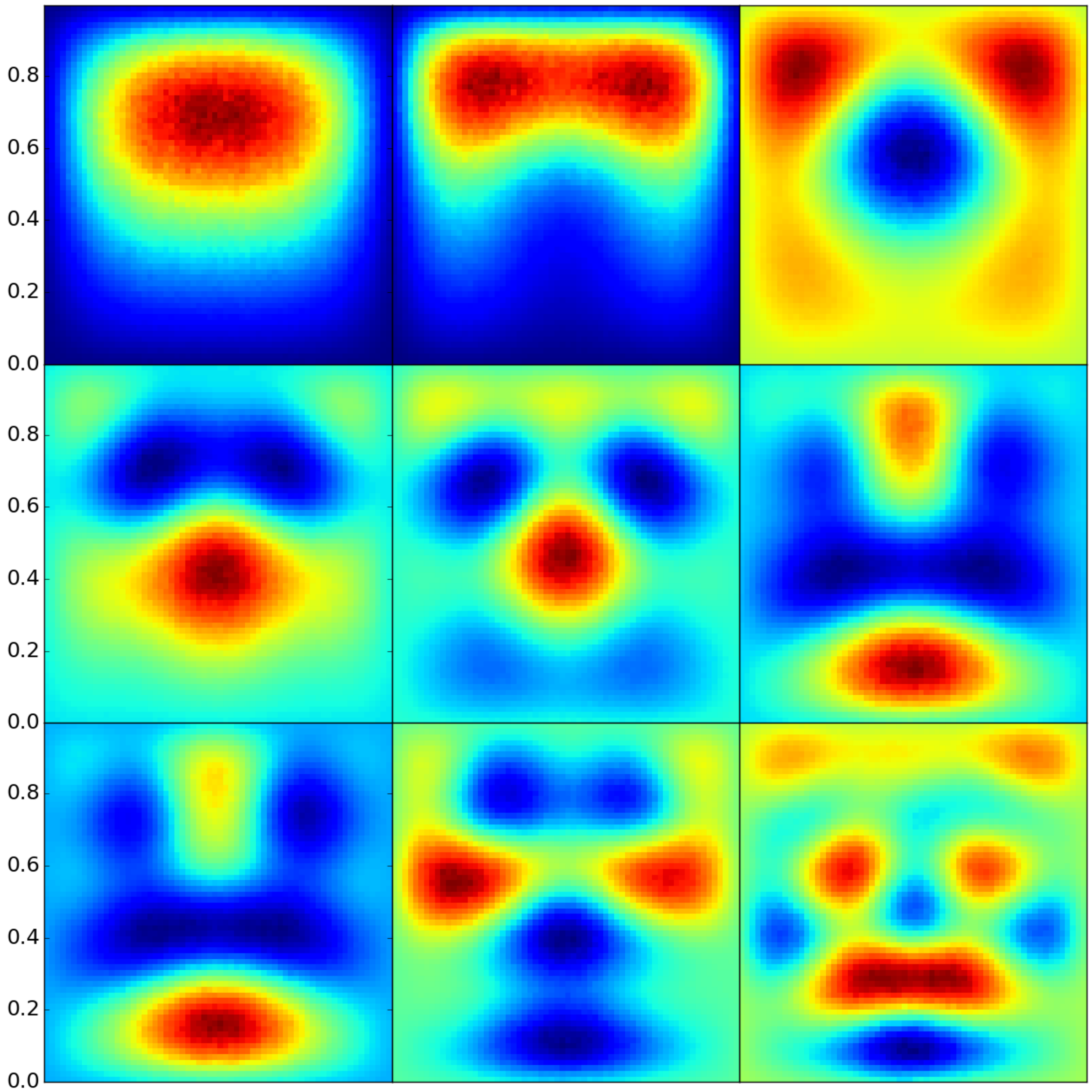


Figure 5.13: Eigenvectors from the restarted cascading eigensolver, with 128 samples

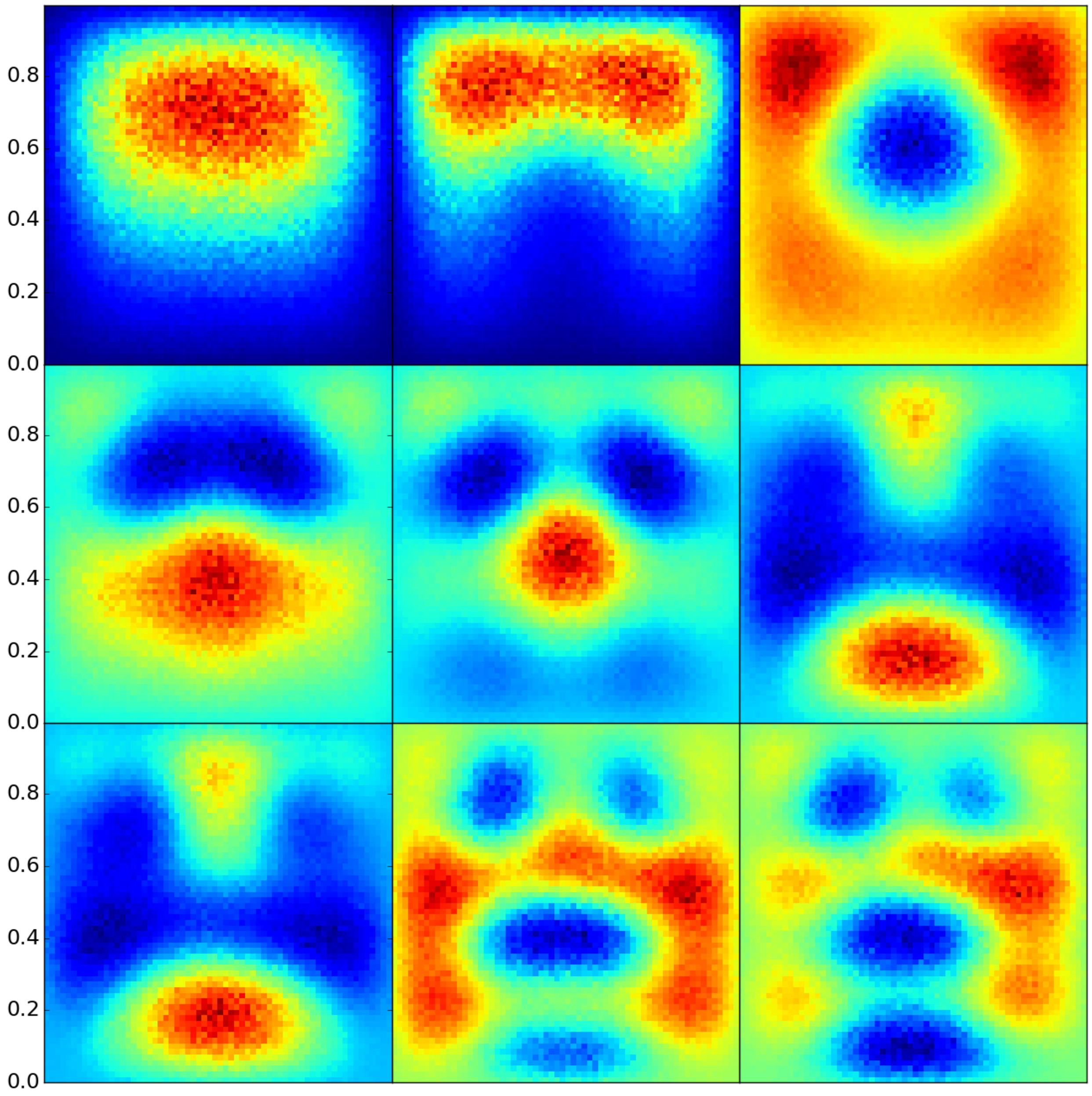


Figure 5.14: Eigenvectors from the restarted cascading eigensolver, with 1 filter iteration

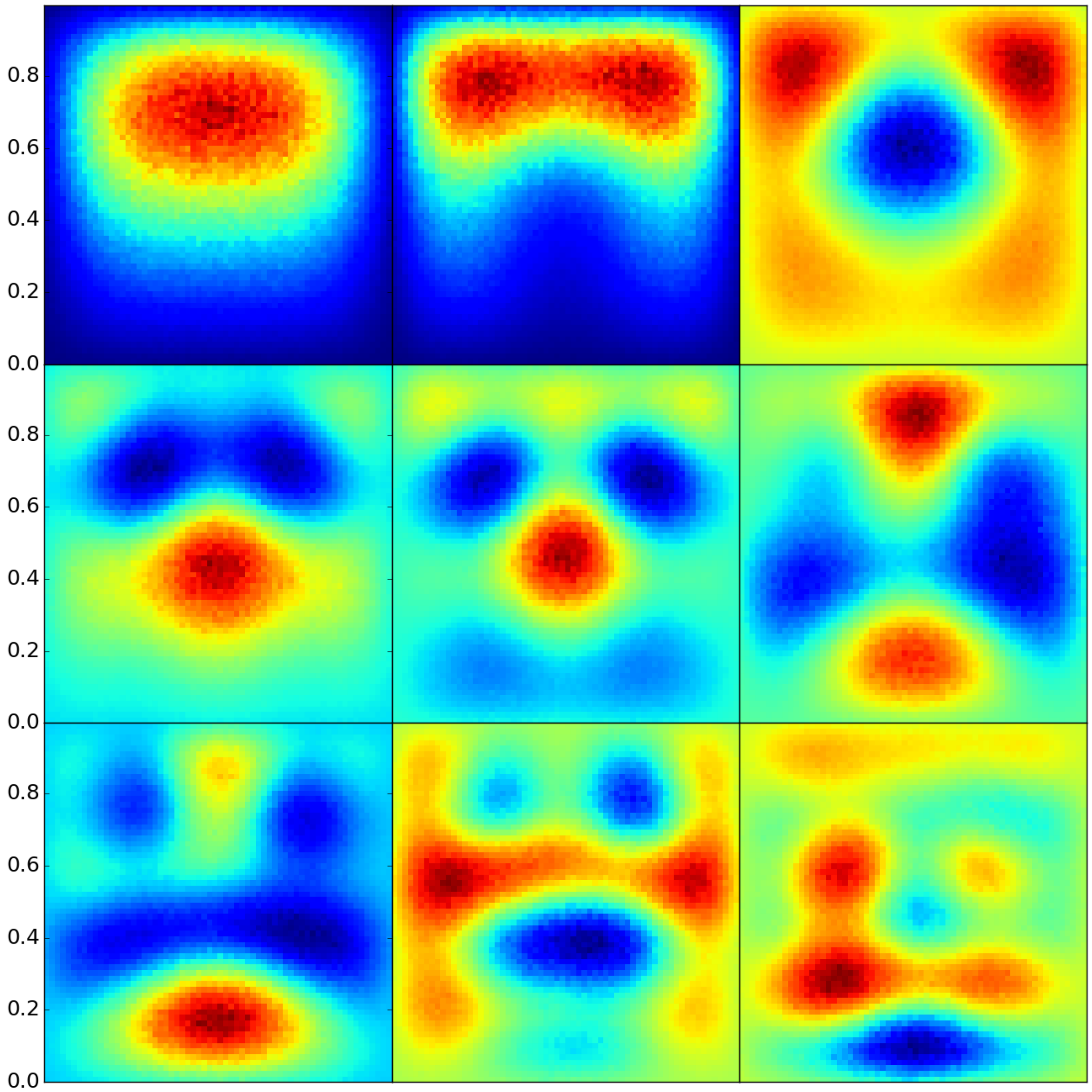


Figure 5.15: Eigenvectors from the restarted cascading eigensolver, with 4 filter iterations

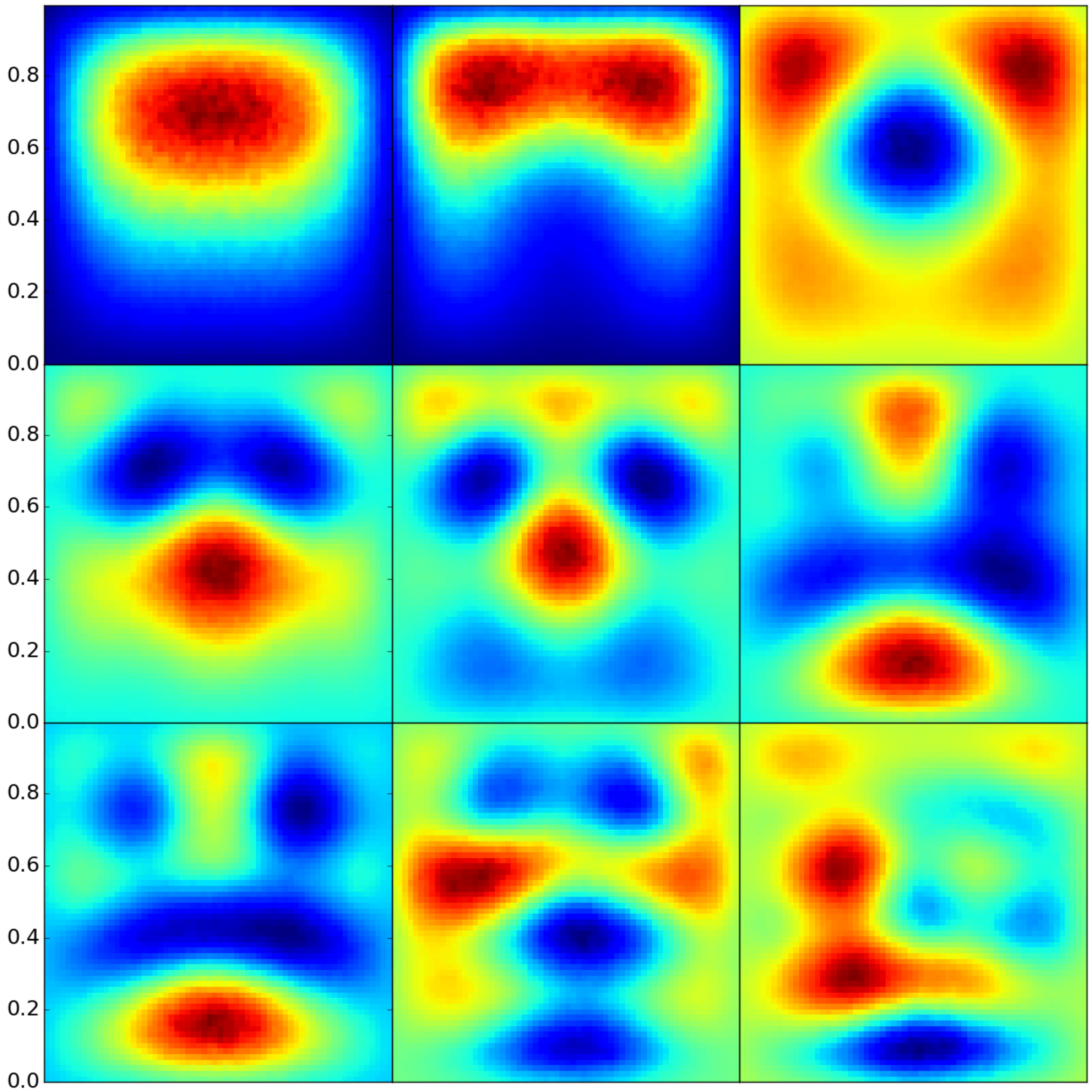


Figure 5.16: Eigenvectors from the restarted cascading eigensolver, with 16 filter iterations

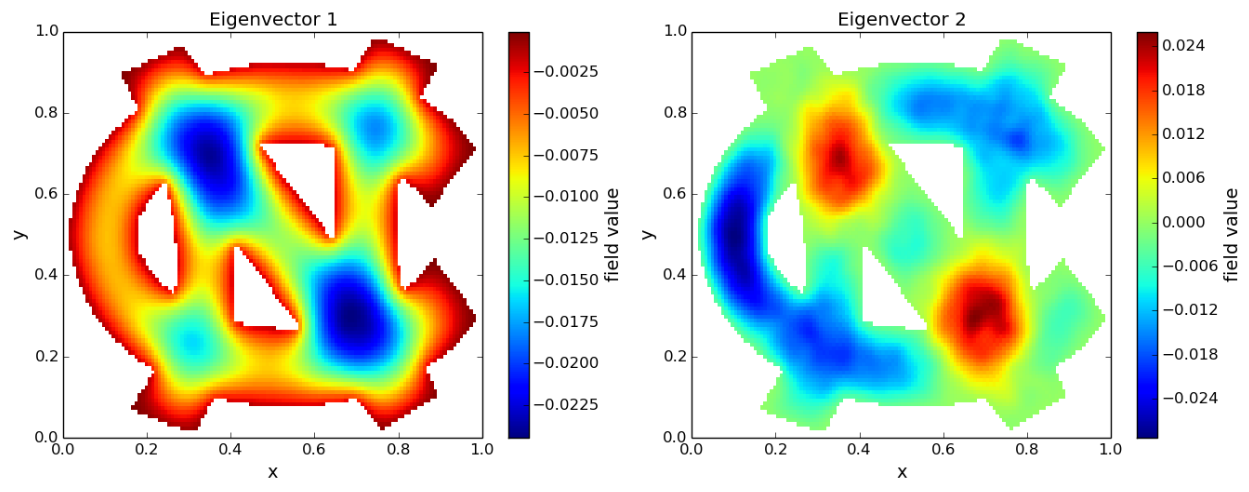


Figure 5.17: The first two approximations to the eigenvectors of the Laplacian as determined by the restarted cascading eigensolver.

symmetries create clusters of eigenvectors with nearly identical eigenvalues. The first two eigenvector approximations computed by this approach are shown in Figure 5.17. For the remainder of this work, the implementation is on a rectangular domain, for ease of implementation, analysis, and comparison with external packages that take structured grid inputs, such as HYPRE.

CHAPTER 6

Continuum-Kinetic Hybrid Methods

The eigenvector approximations calculated by the method in Chapter 5 can be used to improve the convergence of standard iterative linear solvers as described in Chapter 4 applied to the continuum formulations presented in Chapter 3. Two approaches are examined here. The first approach is to use the approximate eigenvectors to enrich a Krylov subspace, and the second approach is to use them to construct a deflation preconditioner. In both cases, the eigenvector computations are carried out on a GPU, while the linear solver computations are carried out concurrently on a CPU. The general concurrent computation framework is described Section 6.1, with the application of that framework to Krylov subspace enrichment in Section 6.2, and to deflation preconditioning in Section 6.3. This class of methods will be referred to as hybrid augmented GMRES, while the enriched subspace method in particular will be referred to as hybrid enriched GMRES (HEGMRES), and deflation preconditioner will be referred to as HDGMRES.

6.1 Concurrent computation

The general hybrid augmented GMRES method is designed so that there is no latency for either the CPU or the GPU caused by waiting for the other device to complete its operations. The first useful property of each method is that they're both performing some sort of iterative process, which is capable of taking new inputs at each new iteration. For the CPU, this is each restart of augmented GMRES, where the new inputs are additional eigenvectors. For the GPU, this is each cycle of the restarted eigensolver (Algorithm 9), where the new inputs are initial conditions for the cascading scheme. The second useful property is that each algorithm can also proceed without these new inputs at each iteration. With these two properties, it's easy to design a concurrent scheme in which each device:

1. At some point in each iteration, writes new inputs for the other device into common memory, without interrupting it.

Algorithm 14 Hybrid augmented GMRES - CPU Side

```
Input: $A$, $\boldsymbol{b}$
  $\boldsymbol{r} = \boldsymbol{b} - A\boldsymbol{x}_0$
  Launch GPU eigensolver (Algorithm 15) with initial condition $\boldsymbol{u}_0 = \boldsymbol{r}$
  for $n_{restart}$=1 to $max_{restarts}$ do
      Perform Arnoldi iteration
      Minimize and update approximate solution $\boldsymbol{x}$
      $\boldsymbol{r} = \boldsymbol{b} - A\boldsymbol{x}_0$
      Write new $\boldsymbol{u}_0 = \boldsymbol{r}$ to common memory
      Set flag notifying GPU that a new $\boldsymbol{u}_0$ is available
      if Algorithm 15 has returned new eigenvectors then
          Use the eigenvectors $\varphi_i, \varphi_{i+1}, ...$ to enrich the Krylov subspace, or to update the preconditioner
      end if
  end for
```

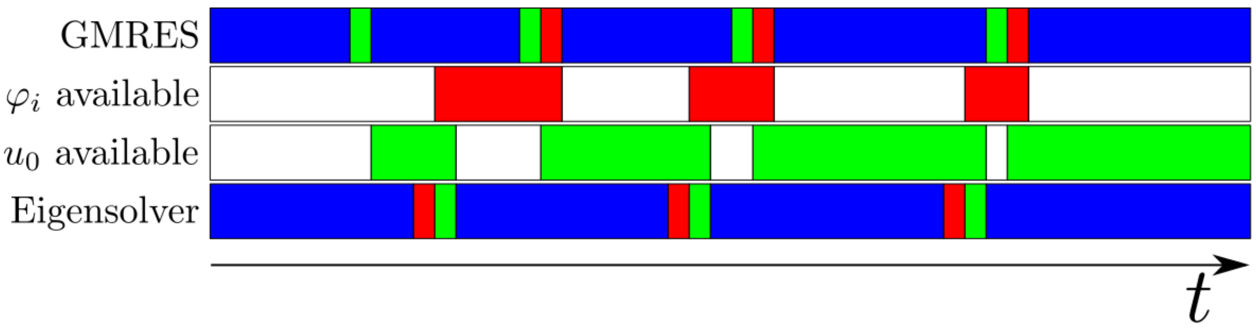


Figure 6.1: Timeline for a concurrent hybrid augmented GMRES implementation. Blue indicates the main computations for both timelines. Green on the GMRES timeline indicates writing of $\boldsymbol{u}_0$ from the CPU to common memory, while on the eigensolver timeline, it indicates the reading of $\boldsymbol{u}_0$ from common memory to GPU memory. Red on the eigensolver timeline indicates writing of φ_i from the GPU to common memory, while on the GMRES timeline, it indicates the reading of φ_i from common memory to CPU memory. This particular timeline shows a computation in which the eigensolver iterations take slightly longer than a single GMRES restart.

2. At another point in each iteration, checks for new inputs in the common memory, and uses them if available.

A template concurrent computation method for implementing these steps is given in Algorithm 14 (CPU side) and Algorithm 15 (GPU side). A timeline for this general concurrent method is illustrated in Figure 6.1. Extensions of the general method are given with results for HEGMRES in Section 6.2 and for HDGMRES in Section 6.3.

Algorithm 15 Hybrid augmented GMRES - GPU Side

```
Input: $\boldsymbol{u}_0$
Output: $\varphi_0, \varphi_1, ...$
  Receive $\boldsymbol{u}_0$ from CPU
  while not converged do
      if CPU has provided new $\boldsymbol{u}_0$ then
          Subtract components in previously computed eigenvectors from $\boldsymbol{u}_0$
          Begin cascading eigenvector scheme (Algorithm 9) with initial condition $\boldsymbol{u}_0$
      else
          Continue cascading eigenvector scheme
      end if
      if Algorithm 9 returns a new $\varphi_i$ then
          Write $\varphi_i$ to common memory
          Set flag notifying CPU that $\varphi_i$ available
      end if
      if CPU set converged flag then
          Break
      end if
  end while
```

6.2 Hybrid enriched GMRES

For hybrid enriched GMRES, the eigenvectors provided by the GPU eigensolver are used to enrich the Krylov subspace, taking the place of the approximate eigenvectors in the method by Morgan [65] (see Section 4.1.2). The extension of the template concurrent method (Algorithm 14) to hybrid enriched GMRES is given in Algorithm 16, and the progression of the method is illustrated in Figure 6.2.

6.2.1 Results in 1D

Based on 1D experiments, it was found that enriched GMRES would not be ideal for this concurrent approach, as even when given the analytic eigenvectors directly, convergence isn't guaranteed to be much better than diagonal preconditioning. This is shown in Figure 6.3, and is caused by previously eliminated eigenvectors being re-excited on subsequent restarts, as illustrated in Figure 6.4. Since all of the work done on the GPU is to generate approximate eigenvectors, it would be preferable to have that information explicitly retained once computed, which is why we explore HDGMRES in much more depth.

6.3 Hybrid deflated GMRES

For hybrid deflated GMRES, the eigenvectors provided by the GPU eigensolver are used to construct a deflation preconditioner, complementing or replacing the Schur decomposition vectors

Algorithm 16 Hybrid enriched GMRES - CPU Side - continues to Algorithm17

```
Input: $A$,$\boldsymbol{b}$
Output: $\boldsymbol{x}$
  Pick an arbitrary $\mathbf{x}_0$.
  Set a convergence tolerance $\epsilon$
  Allocate common memory for CPU and GPU
  Launch GPU eigensolver (Algorithm 15) with initial condition $\boldsymbol{u}_0 = \boldsymbol{r} = \boldsymbol{b} - A\boldsymbol{x}_0$

  for $n = 0$ to $n_{\max} - 1$ do
      Pick the dimension of the base Krylov subspace, $m$
      Check common memory for $k$ new eigenvectors $\phi_1, ..., \phi_k$
      $l = m + k$

      Initialization
      $\mathbf{r}_0 = \mathbf{b} - A\mathbf{x}_0$
      $\mathbf{q}_1 = \mathbf{r}_0/\|\mathbf{r}_0\|$
      $\mathbf{w}_1 = \mathbf{q}_1$
      for $i$=1 to $k$ do
          $\mathbf{w}_{m+i} = \mathbf{y}_i$
      end for

      Arnoldi Iteration
      for $j = 1$ to $m$ do
          $\mathbf{v} = A\mathbf{q}_j$
          $\mathbf{q}_{j+1} = \mathbf{v}$
          for $i$=1 to $j$ do
              $h_{ij} = \mathbf{q}_i^*\mathbf{q}_{j+1}$
              $\mathbf{q}_{j+1} = \mathbf{q}_{j+1} - h_{ij}\mathbf{q}_i$
          end for
          $h_{j+1,j} = \|\mathbf{q}_{j+1}\|$
          $\mathbf{q}_{j+1} = \mathbf{q}_{\mathbf{j+1}}/h_{j+1,j}$
          if $j < m$ then
              $\mathbf{w}_{j+1} = \mathbf{q}_{j+1}$
          end if
      end for
```

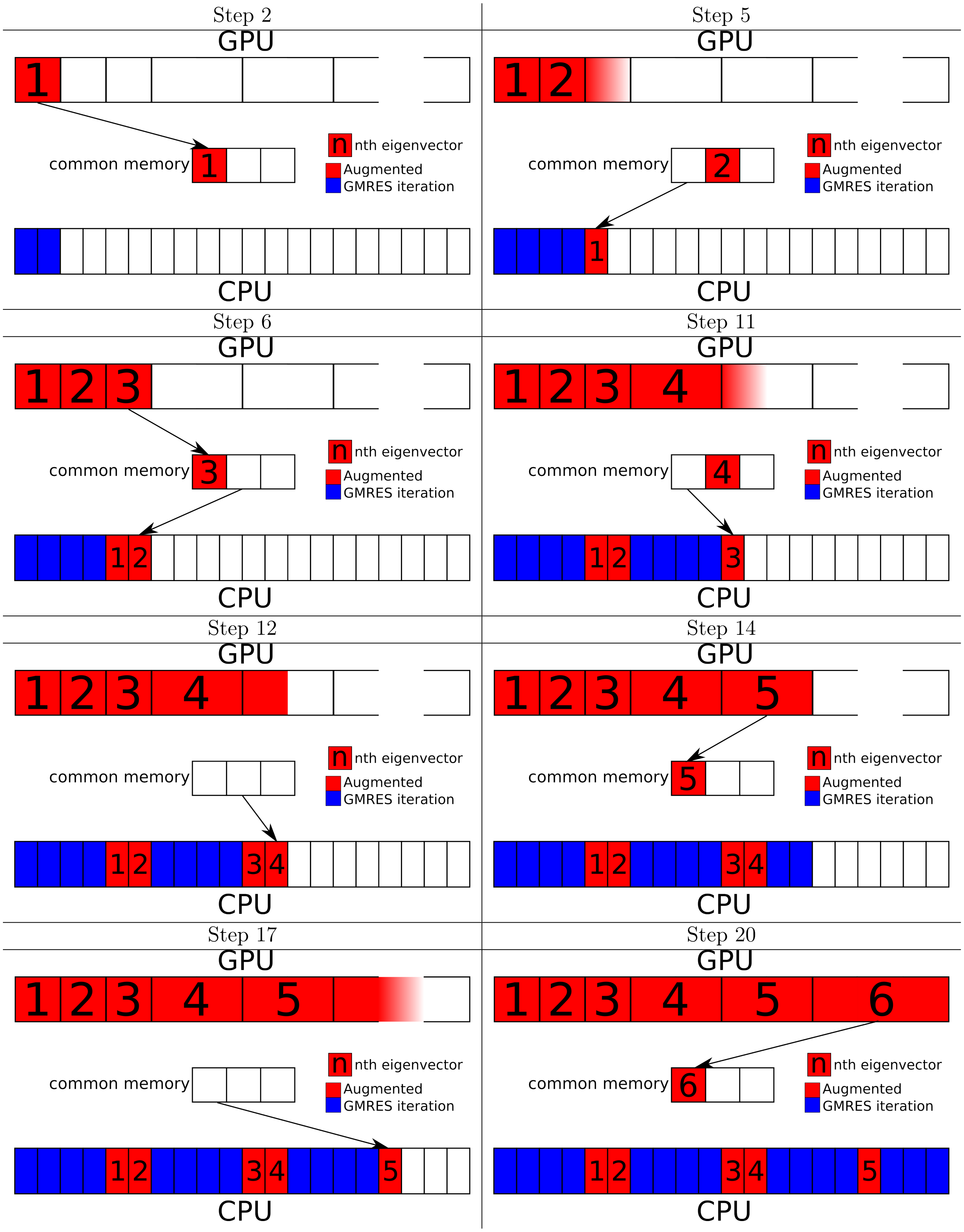


Figure 6.2: Augmented GMRES timeline

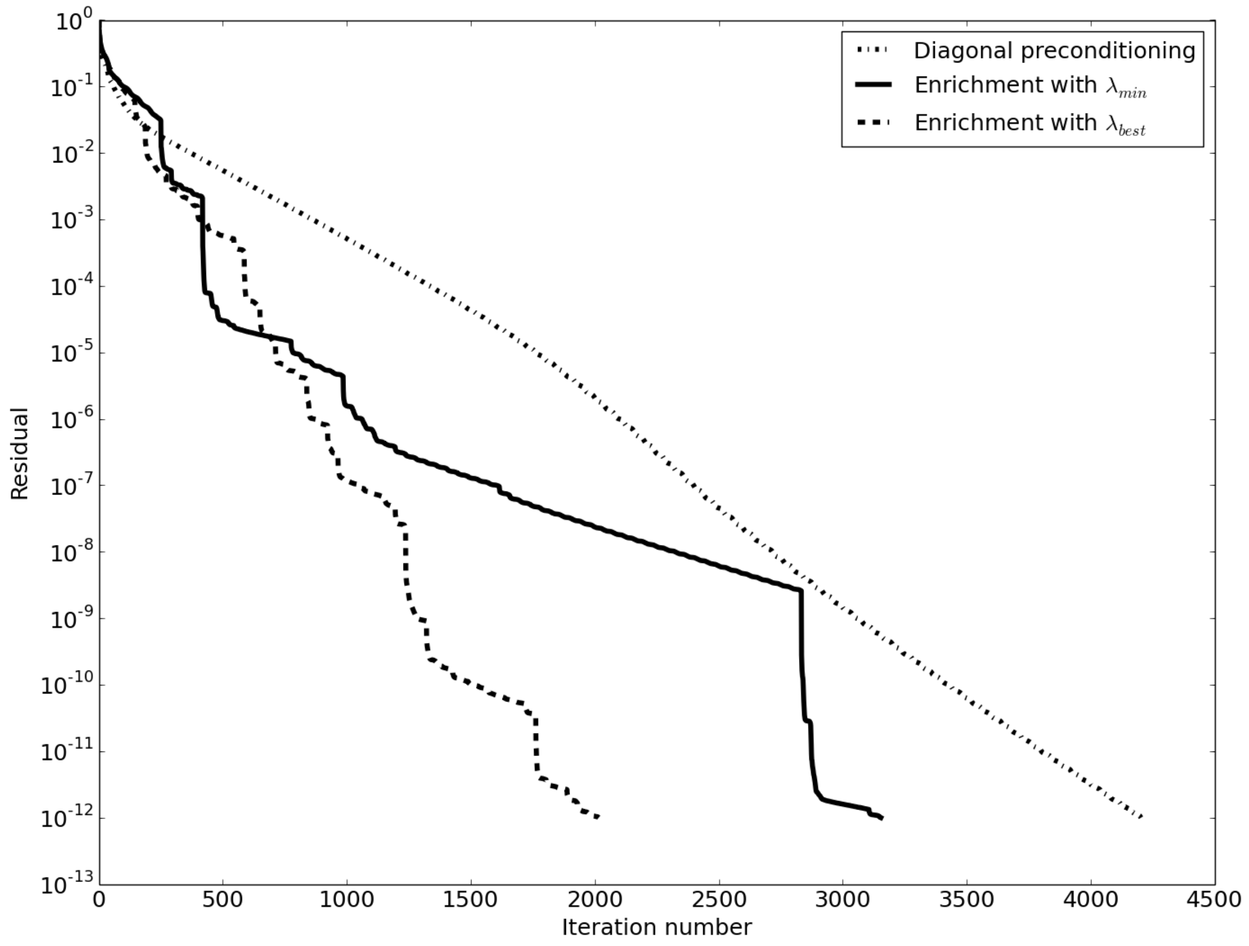


Figure 6.3: Enriched GMRES convergence

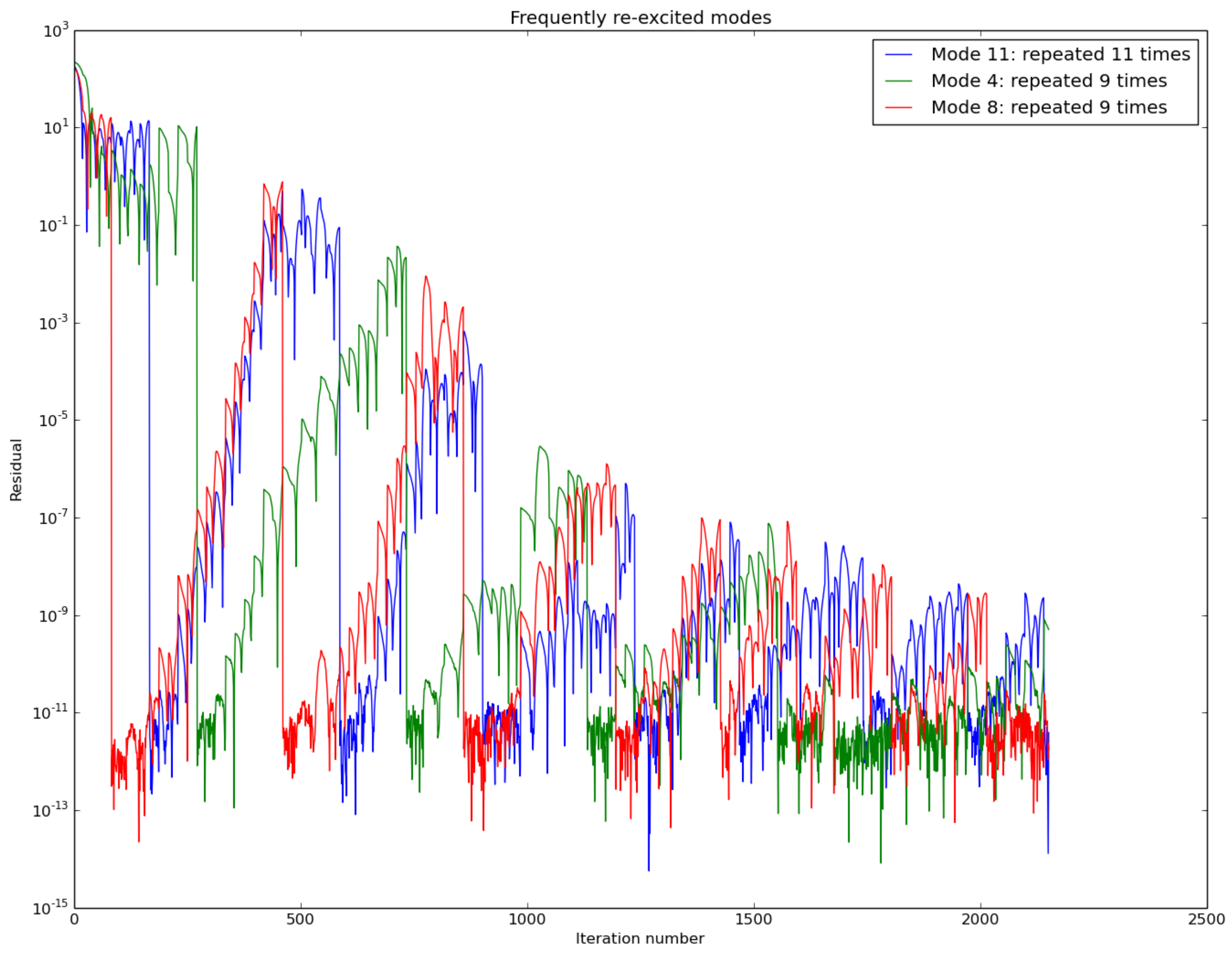


Figure 6.4: Previously eliminated eigenvectors being re-excited by enriched GMRES

Algorithm 17 Hybrid enriched GMRES - CPU Side - continued from Algorithm 16

```
    **Addition of approximate eigenvectors**
    for $j = 1$ to $m + 1$ do
        $\mathbf{v} = A\mathbf{w}_j$
        $\mathbf{q}_{j+1} = v$
        for $i$=1 to $j$ do
            $h_{ij} = \mathbf{q}_i^* \mathbf{q}_{j+1}$
            $\mathbf{q}_{j+1} = \mathbf{q}_{j+1} - h_{ij}\mathbf{q}_i$
        end for
        $h_{j+1,j} = \|\mathbf{q}_{j+1}\|$
        $\mathbf{q}_{j+1} = \mathbf{q_{j+1}}/h_{j+1,j}$
    end for

    **Find approximate solution**
    $\beta = \|\mathbf{r}_0\|$
    Find $\hat{\mathbf{d}} \in \mathbb{R}^l$ that minimizes $\|\beta\mathbf{e}_1 - \tilde{H}\mathbf{d}\|$
    $\hat{\mathbf{x}} = \mathbf{x}_0 + W\hat{\mathbf{d}}$

    **Restart**
    $\mathbf{r} = \mathbf{b} - A\hat{\mathbf{x}}$
    if $\|\mathbf{r}\| < \epsilon$ then
        Set converged flag for GPU
        return $\boldsymbol{x}$
    else
        $\mathbf{x}_0 = \hat{\mathbf{x}}$
    end if
    Write new $\boldsymbol{u}_0 = \boldsymbol{r}$ to common memory
    Set flag notifying GPU that a new $\boldsymbol{u}_0$ is available
end for
```

in the method by Erhel [71] (see Section 4.3). The extension of the template concurrent method (Algorithm 14) to hybrid deflated GMRES is given in Algorithm 18. A full parallel implementation in 2D is described in Section 6.3.1, with results and scaling studies presented in Section 6.4.

6.3.1 Implementation in 2D

For a parallel implementation in 2D, we consider a cluster comprised of p computational nodes, with q logical CPU cores per node, and r GPUs per node. For the purposes of this description, it's assumed that all of the processors have identical characteristics. We label the nodes as n_i, with $i = 0, .., p-1$. We label the logical CPU cores an index c_{ij}, where $j = 0, \cdots, q-1$, and assign a single Message Passing Interface (MPI) thread to each logical core. The "root" core is $c_{0,0}$, and the "root" thread is the one assigned to that core. We define an index set I_p that contains the global

Algorithm 18 Hybrid deflated GMRES - CPU Side

```
Input: $A$,$\boldsymbol{b}$,$\boldsymbol{x}_0$,$\epsilon$,$k_{\text{precond}}$
Output: $\boldsymbol{x}$
  Allocate common memory for CPU and GPU
  Launch GPU eigensolver (Algorithm 15) with initial condition $\boldsymbol{u}_0 = \boldsymbol{r} = \boldsymbol{b} - A\boldsymbol{x}_0$
  $M^{-1} = I$
  for $k$=0 to $k_{\max}$-1 do
      Pick the dimension of the base Krylov subspace, $m$

      Initialization
      $\mathbf{r}_0 = \mathbf{b} - M^{-1}A\mathbf{x}_0$
      $\mathbf{q}_1 = \mathbf{r}_0/\|\mathbf{r}_0\|$

      Arnoldi Iteration
      Get $M^{-1}A = Q_m\tilde{H}_mQ_m^*$

      Find approximate solution
      $\beta = \|\mathbf{r}_0\|$
      Find $\hat{\mathbf{d}} \in \mathbb{R}^m$ that minimizes $\|\beta\mathbf{e}_1 - \tilde{H}\mathbf{d}\|$
      $\hat{\mathbf{x}} = \mathbf{x}_0 + M^{-1}Q\hat{\mathbf{d}}$
      $\mathbf{r} = \mathbf{b} - M^{-1}A\hat{\mathbf{x}}$
      if $\|\mathbf{r}\| < \epsilon$ then
          Set converged flag for GPU
          return $\boldsymbol{x}$
      else
          $\mathbf{x}_0 = \hat{\mathbf{x}}$
      end if
      Write new $\boldsymbol{u}_0 = \boldsymbol{r}$ to common memory
      Set flag notifying GPU that a new $\boldsymbol{u}_0$ is available

      Update preconditioner
      Check common memory for $l$ new eigenvectors $\phi_1, ..., \phi_l$
      if k<$k_{\text{precond}}$ then
          Schur factorize $\tilde{H}_m = \tilde{S}B\tilde{S}^*$
          Order Schur decomposition by increasing eigenvalue
          $\lambda_m = max(\sigma_B)$
          $S = Q_m\tilde{S}$
          for $j$=1 to $l$ do
              $J = j + kl$
              $u_J = \phi_j$
              for $i$=1 to $j + kl - 1$ do
                  $u_J = u_J - (u_i^* u_J)u_i$
              end for
              $u_J = u_J/\|u_J\|$
          end for
          $T = U^*AU$
          $M^{-1} = I + U^*(T^{-1}\|\lambda_m\| - I_{(k+1)l})U$
      end if
  end for
```

index of a single thread per node, which is referred to as the primary thread on the node. This set has m elements (one per node), which for many MPI implementations and cluster configurations can be given simply by

$$I_p = \{0, q-1, 2q-1, \cdots, (p-1)q-1\}.$$

For other configurations, we group threads within a node using a hash table based on the name of the node, as returned by `MPI_Get_processor_name`. The first thread within the grouping on node i then has its index placed in I_p, and is labeled $c_{i,0}$. The GPUs themselves are labeled g_{ik}, with $k = 0, \cdots, r-1$. The thread hierarchy is illustrated for the case of $p = 3, q = 3, r = 2$ in Figure 6.5. Within this hierarchy, particular operations are carried out only by certain subsets of these (co)processors. The operations carried out by all threads are:

- Sparse matrix-vector multiplications
- Dot products
- Norms

The operations carried out only by primary threads (with global indices in I_p) are:

- Communication with the GPU(s) on the node
- Gathering/scattering of GPU outputs/inputs between nodes

The operations carried out only by the root thread are:

- Averaging of GPU samples
- Dense linear algebra

The assignment of operations to certain threads is highlighted in an annotated version of the hybrid deflated GMRES, given in Algorithm 19. Green operations are carried out by all threads, blue are carried out only by primary threads, and red are carried out only by the root thread.

6.4 HDGMRES results in 2D

In the following section, in a similar manner to Section 5.7, the method is applied on the domain $\Omega = [0,1] \times [0,1]$, and the effect of various eigensolver parameter choices on the convergence

Algorithm 19 Hybrid deflated GMRES - CPU Side - Parallel annotations

```
Input: $A$,$\boldsymbol{b}$,$\boldsymbol{x}_0$,$\epsilon$,$k_{\text{precond}}$
Output: $\boldsymbol{x}$
  Allocate common memory for CPU and GPU
  Launch GPU eigensolver with initial condition $\boldsymbol{u}_0 = \boldsymbol{r} = \boldsymbol{b} - A\boldsymbol{x}_0$
  $M^{-1} = I$
  for $k$=0 to $k_{\max}$-1 do
      Pick the dimension of the base Krylov subspace, $m$

      Initialization
      $\mathbf{r}_0 = \mathbf{b} - M^{-1}A\mathbf{x}_0$
      $\mathbf{q}_1 = \mathbf{r}_0/\|\mathbf{r}_0\|$

      Arnoldi Iteration
      Get $M^{-1}A = Q_m\tilde{H}_mQ_m^*$

      Find approximate solution
      $\beta = \|\mathbf{r}_0\|$
      Find $\hat{\mathbf{d}} \in \mathbb{R}^m$ that minimizes $\|\beta\mathbf{e}_1 - \tilde{H}\mathbf{d}\|$
      $\hat{\mathbf{x}} = \mathbf{x}_0 + M^{-1}Q\hat{\mathbf{d}}$
      $\mathbf{r} = \mathbf{b} - M^{-1}A\hat{\mathbf{x}}$
      if $\|\mathbf{r}\| < \epsilon$ then
          Set converged flag for GPU
          return $\boldsymbol{x}$
      else
          $\mathbf{x}_0 = \hat{\mathbf{x}}$
      end if
      Write new $\boldsymbol{u}_0 = \boldsymbol{r}$ to common memory
      Set flag notifying GPU that a new $\boldsymbol{u}_0$ is available

      Update preconditioner
      Check common memory for $l$ new eigenvectors $\phi_1, ..., \phi_l$
      if k<$k_{\text{precond}}$ then
          Schur factorize $\tilde{H}_m = \tilde{S}B\tilde{S}^*$
          Order Schur decomposition by increasing eigenvalue
          $\lambda_m = max(\sigma_B)$
          $S = Q_m\tilde{S}$
          for $j$=1 to $l$ do
              $J = j + kl$
              $u_J = \phi_j$
              for $i$=1 to $j + kl - 1$ do
                  $u_J = u_J - (u_i^*u_J)u_i$
              end for
              $u_J = u_J/\|u_J\|$
          end for
          $T = U^*AU$
          $M^{-1} = I + U^*(T^{-1}\|\lambda_m\| - I_{(k+1)l})U$
      end if
  end for
```

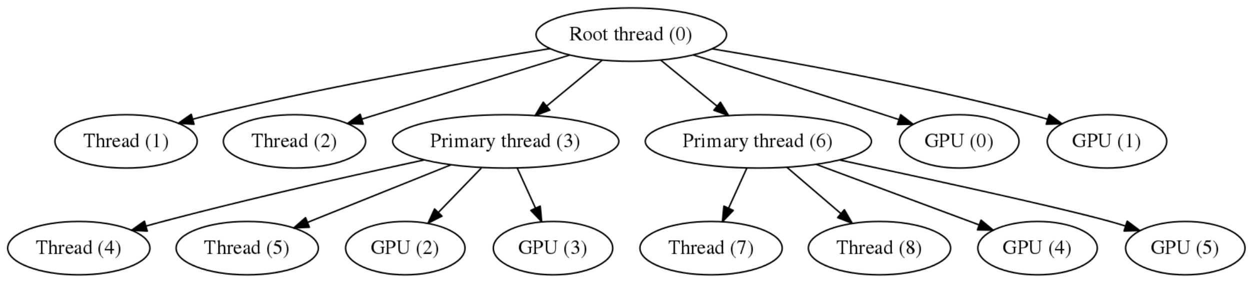


Figure 6.5: Hierarchy of threads for a parallel implementation of HDGMRES with $p = 3$ nodes, $q = 3$ threads per node, and $r = 2$ GPUs per node.

of HDGMRES is investigated. For all of these variations, the method is run on a single node configuration with 12 CPU cores and 2 GPUs. As in Section 5.7, the problem is defined by

$$\begin{aligned} K(\boldsymbol{x}) &= 1 \\ f(\boldsymbol{x}) &= 1 \\ g(\boldsymbol{x}) &= x \\ \Gamma_D &= \partial\Omega, \end{aligned} \tag{6.1}$$

and the method is applied to the domain discretized with 256×256 cells. In Section 6.4.2, the effect of different diffusivity distributions on the convergence of HDGMRES is investigated.

6.4.1 Effect of eigensolver parameters on convergence

Initial condition The parameter with the greatest significance is the choice of vector used for the initial condition of the eigensolver. If a uniform vector $\boldsymbol{u}_0 = 0$ is used, the convergence is significantly worse than if the initial condition is set to the current residual, $\boldsymbol{u}_0 = \boldsymbol{r}$. This effect is shown in Figure 6.6. This is due to the fact that given no additional information about the problem, the eigensolver may compute eigenvectors that have insignificant components in the residual. However, if the initial condition is simply set to the current residual, the eigensolver naturally recovers those that do have components in the residual.

Decay ratio In Section 5.7, it was suggested that the decay fraction ϵ should be in the range 0.3-0.7 to sufficiently resolve a broad range of the spectrum. This is borne out in the effect of the decay fraction on the convergence of HDGMRES, which is given in Figure 6.7. For very large

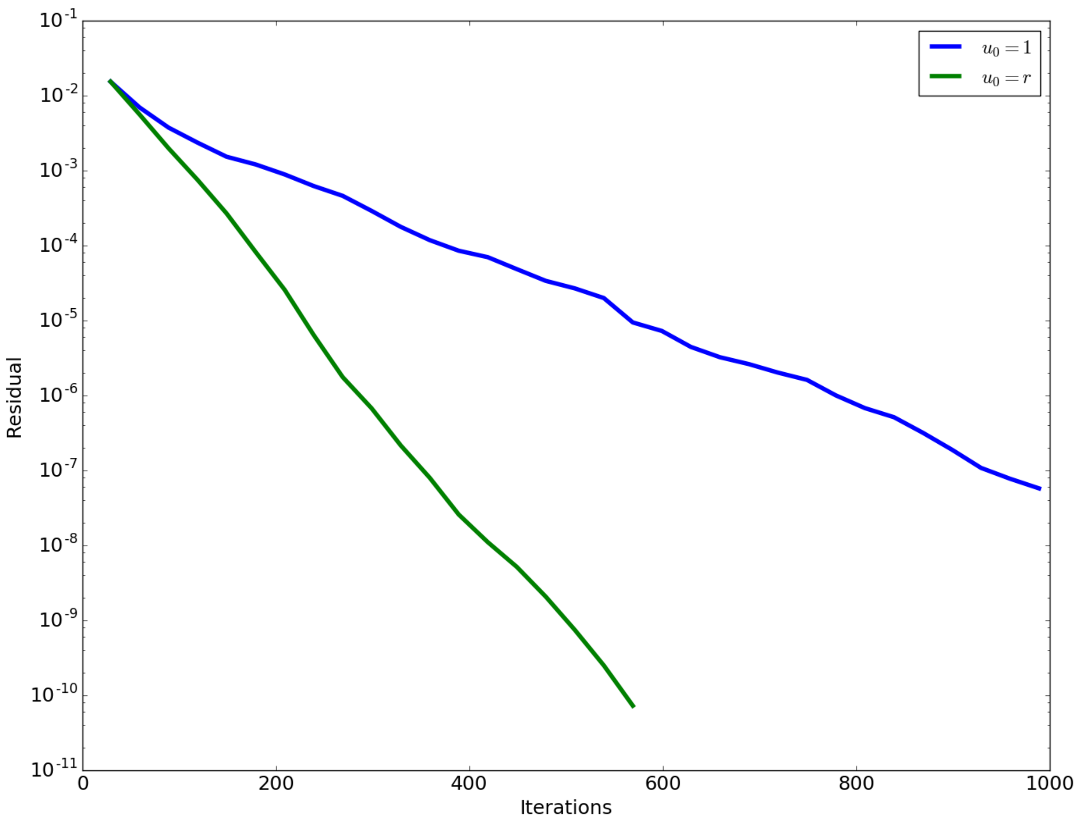


Figure 6.6: Convergence of HDGMRES as a function of the initial condition. The convergence is improved significantly by setting the initial condition to the current residual as opposed to an arbitrary uniform vector.

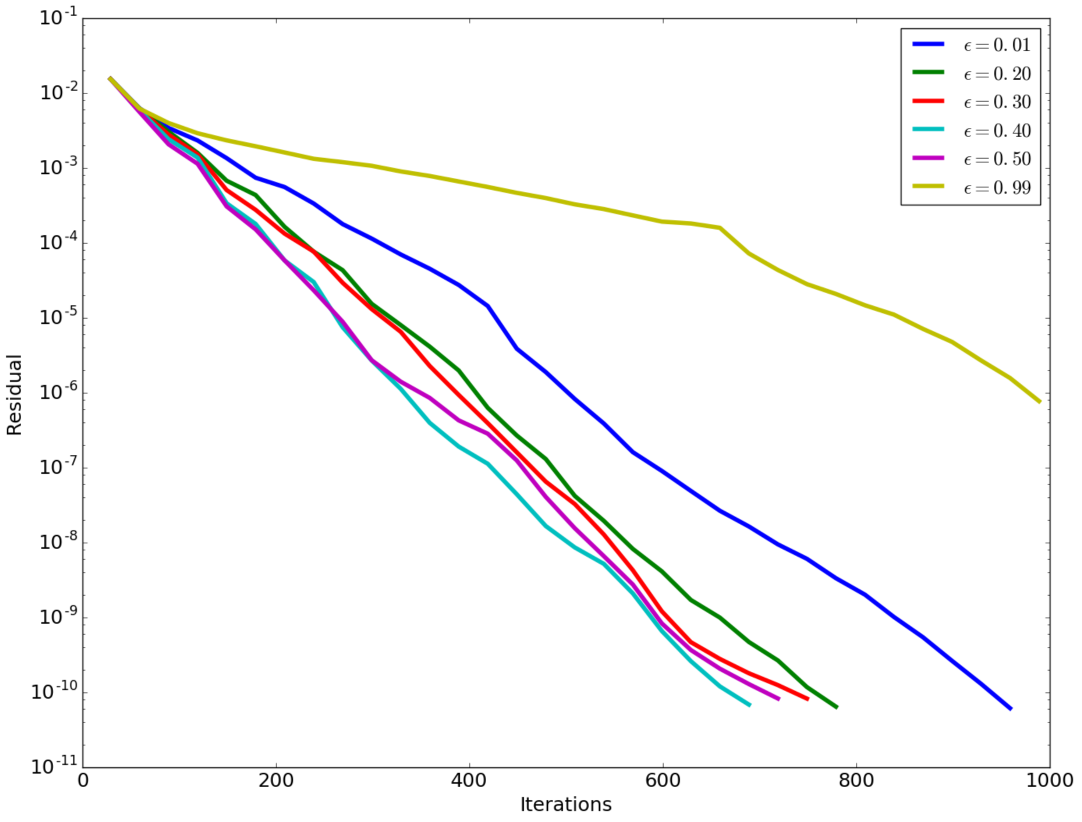


Figure 6.7: Convergence of HDGMRES as a function of the decay fraction ϵ

decay ratios, convergence is worse than base DGMRES, as no eigenvectors are resolved at all. For very small decay ratios, only a few eigenvectors are resolved very well, while the remainder of the eigenvectors decay entirely, slowing convergence.

Eigenvector resolution The remaining parameters do not affect which eigenvectors are recovered, but how well they are resolved. The effect of the exit probability p on convergence is given in Figure 6.8, showing that lower exit probabilities reduce the aliasing of shared memory regions, improving the quality of the eigenvectors, and therefore improving convergence. The effect of the number of samples on convergence is given in Figure 6.9, showing that a greater number of samples reduces the noise in the eigenvectors, improving convergence. Finally, the effect of the number of filtering iterations on convergence is given in Figure 6.10, showing that a moderate number of filtering iterations reduces very high frequency noise in the eigenvectors, improving convergence, but

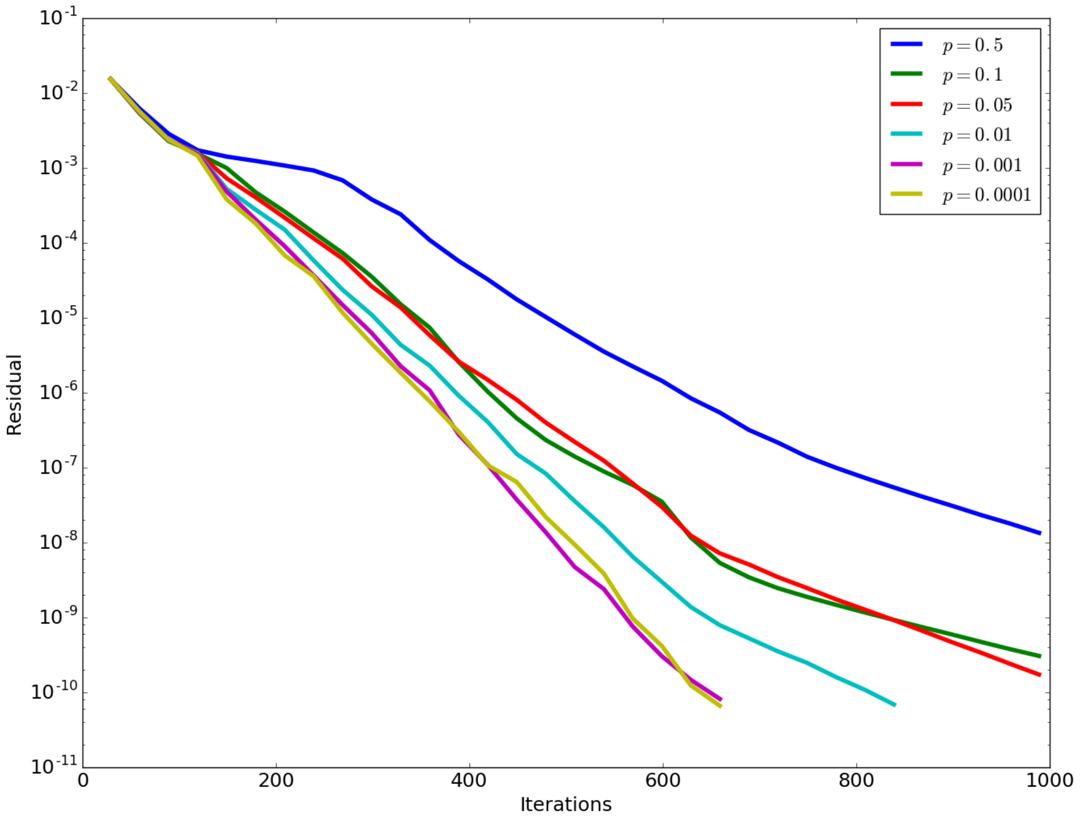


Figure 6.8: Convergence of HDGMRES as a function of the exit probability p

that excessive filtering ultimately washes out high frequency eigenvectors, worsening convergence.

Comparison with existing solvers Finally, a comparison of DGMRES with the PETSc implementation of GMRES and DGMRES, and the HYPRE implementation of the 2-step Jacobi preconditioner is given in Figure 6.11.

6.4.2 Varying diffusivities

In this section we examine a number of cases with varying diffusivity distributions. Depending on the context, we use either diffusivity or permeability to mean the quantity $K(\boldsymbol{x})$.

Smooth variations The first case is a smoothly varying vertical diffusivity gradient, given by $K(\boldsymbol{x}) = 1 + (\kappa - 1)y$. The parameter κ controls the ratio of the largest diffusivity to the smallest. Figure 6.12 shows a comparison of the convergence for 2-step Jacobi preconditioned GMRES,

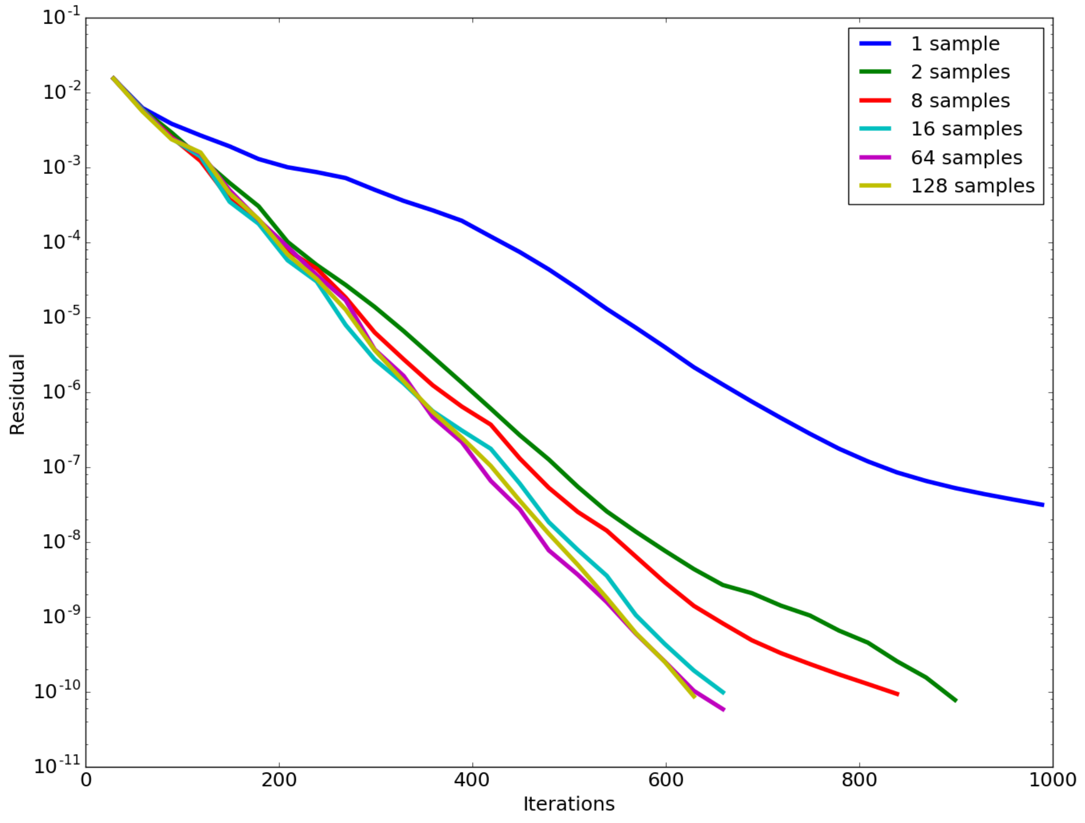


Figure 6.9: Convergence of HDGMRES as a function of the number of samples

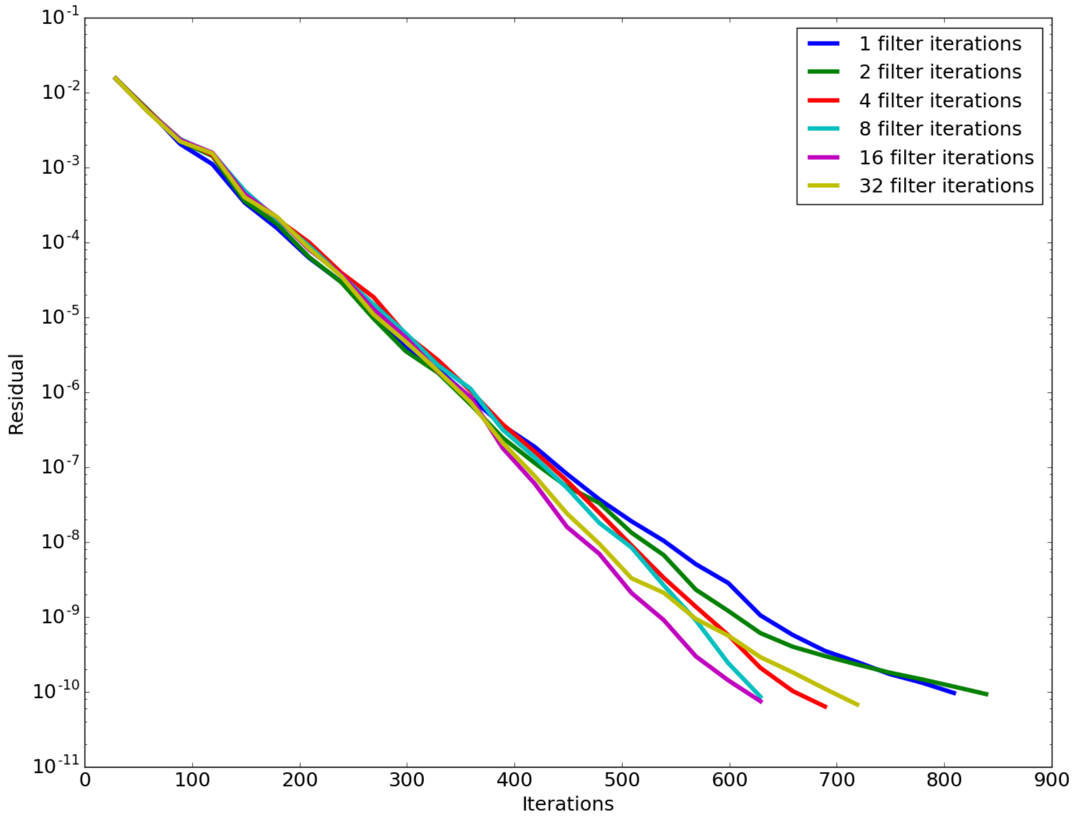


Figure 6.10: Convergence of HDGMRES as a function of the number of filtering iterations

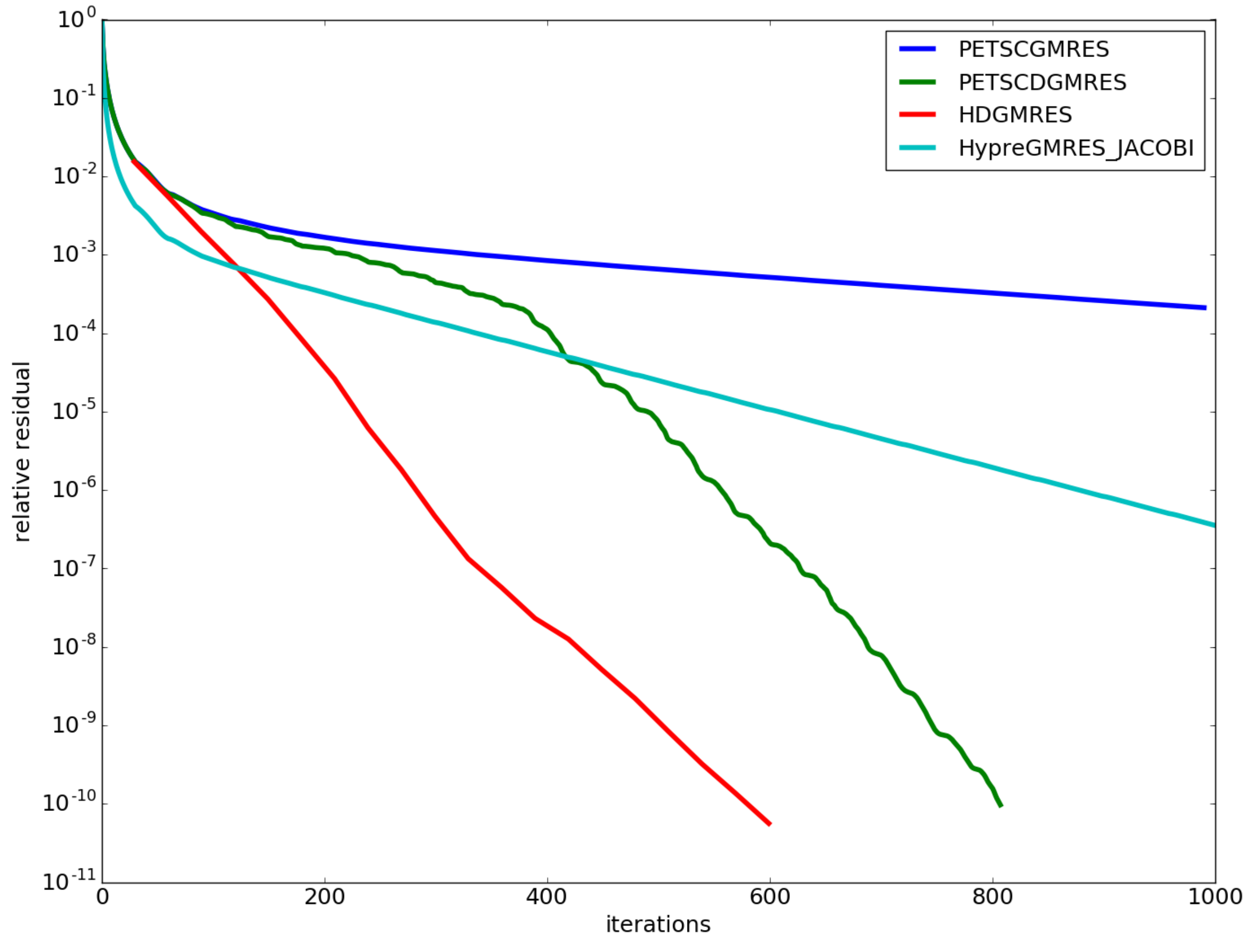


Figure 6.11: Convergence of HDGMRES compared with the PETSc implementation of GMRES and DGMRES, and the HYPRE implementation of the 2-step Jacobi preconditioner

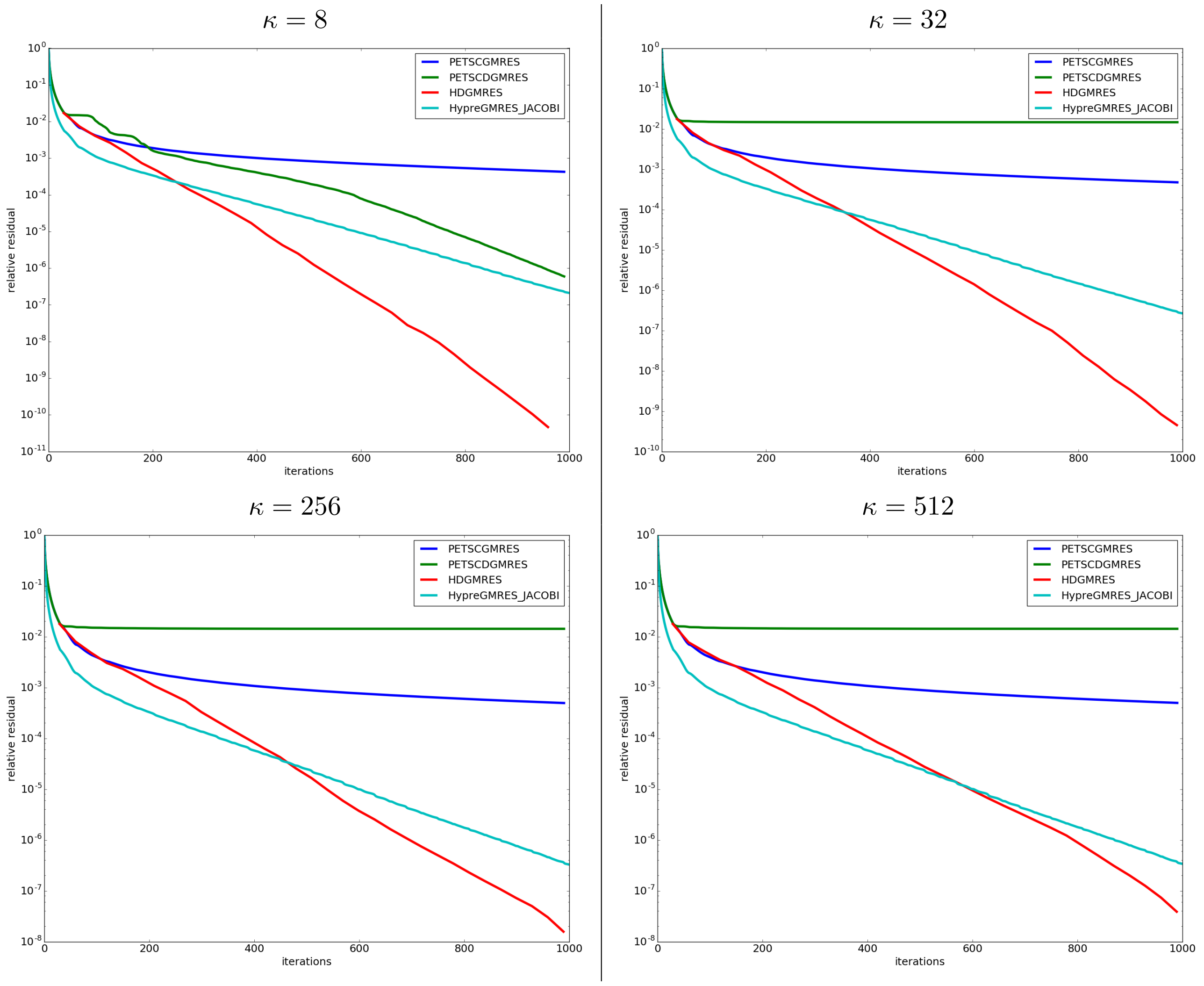


Figure 6.12: Convergence of HDGMRES for a smoothly varying vertical diffusivity gradient, compared with the PETSc implementation of DGMRES, and the HYPRE implementation of the 2-step Jacobi preconditioner

DGMRES and HDGMRES. The convergence of HDGMRES is maintained throughout a range of diffusivity ratios, while the parent method, DGMRES, begins to stagnate at around $\kappa = 32$. The second case is a smoothly varying horizontal gradient, given by $K(\boldsymbol{x}) = 1 + (\kappa - 1)x$. Figure 6.13 shows a comparison of the convergence for the three methods, with similar results to the vertical gradient case.

Stratified variations For the second case, we use a problem similar to the porous media flow problem in [93], where alternating layers of shale, sandstone and a combination of both are used. Here we define a permeability that is stratified with three layers, defined by $y < 0.25$, $0.25 \le y \le 0.75$

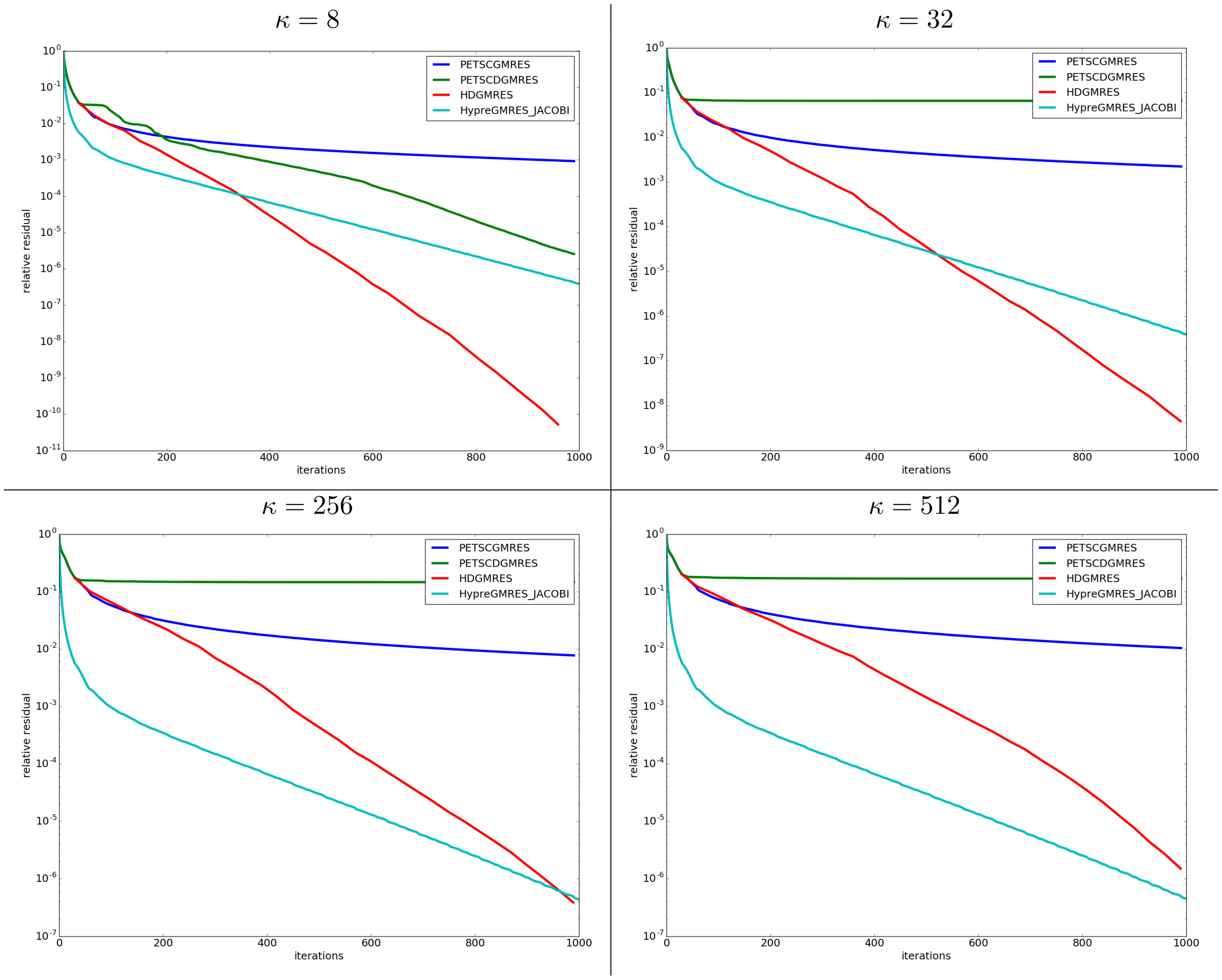


Figure 6.13: Convergence of HDGMRES for a smoothly varying horizontal diffusivity gradient, compared with the PETSc implementation of DGMRES, and the HYPRE implementation of the 2-step Jacobi preconditioner

and $y > 0.75$. The permeabilities are 1, $\sqrt{\kappa}$ and κ in these layers respectively. The convergence of the methods for this vertical stratification is shown in Figure 6.14, and the convergence for a horizontal stratification is shown in Figure 6.17. In both cases, as with in the smooth variation cases, HDGMRES consistently outperforms DGMRES, which stagnates for $\kappa > 32$. However, HDGMRES also stagnates at around $\kappa = 512$, which can be explained by the fact that diffusion now occurs at vastly different timescales on this domain. The effect this has is that initial conditions for the eigensolver rapidly diffuse out of the lower strata completely, leading to poor approximations of the eigenvectors. The onset of this effect is shown for $\kappa = 8$ in Figure 6.15, and the full effect is shown for $\kappa = 256$ in Figure 6.16

Low permeability barriers Instead of full strata of sharp variations, here we examine the effect of many distinct regions of sharply different permeabilities. We take a problem from an application of long term migration of CO2 in saline formations [94], in which the domain has uniform permeability, with the exception of a number of randomly distributed shale barriers that are impermeable. The barriers are oriented horizontally, and have fixed height and a variable width. We generate these barriers with a Monte Carlo method until a given fraction of the domain is taken up by the shale. Instead of having the barrier be fully impermeable, we set the barriers to have a permeability of $K = 1/\kappa$, where κ is a large constant, and study the effect of varying κ. The convergence for a domain composed of 10% impermeable barriers is given in Figure 6.19, with an example permeability distribution shown in Figure 6.18. The convergence for a domain composed of 20% impermeable barriers is given in Figure 6.21, with an example permeability distribution shown in Figure 6.20. For both this problem, as with sharp stratification problem, DGMRES, HDGMRES have limited effectiveness for very high diffusivity/permeability ratios. The key limitation in this case is the fact that these methods aim to improve convergence through completely removing components of the error in a few low frequency eigenvectors. However, in problems where there are significant components of the error in the remainder of the spectrum, this approach is less effective. In these cases, a more appropriate choice of method is one that quickly eliminates high frequency components in the error. Multigrid methods (Section 4.4) for example are more effective in these situations, both as standalone methods and as preconditioners. This is illustrated in Figure 6.22, where SMG is applied as a preconditioner to GMRES for the same low permeability barrier problem, with better

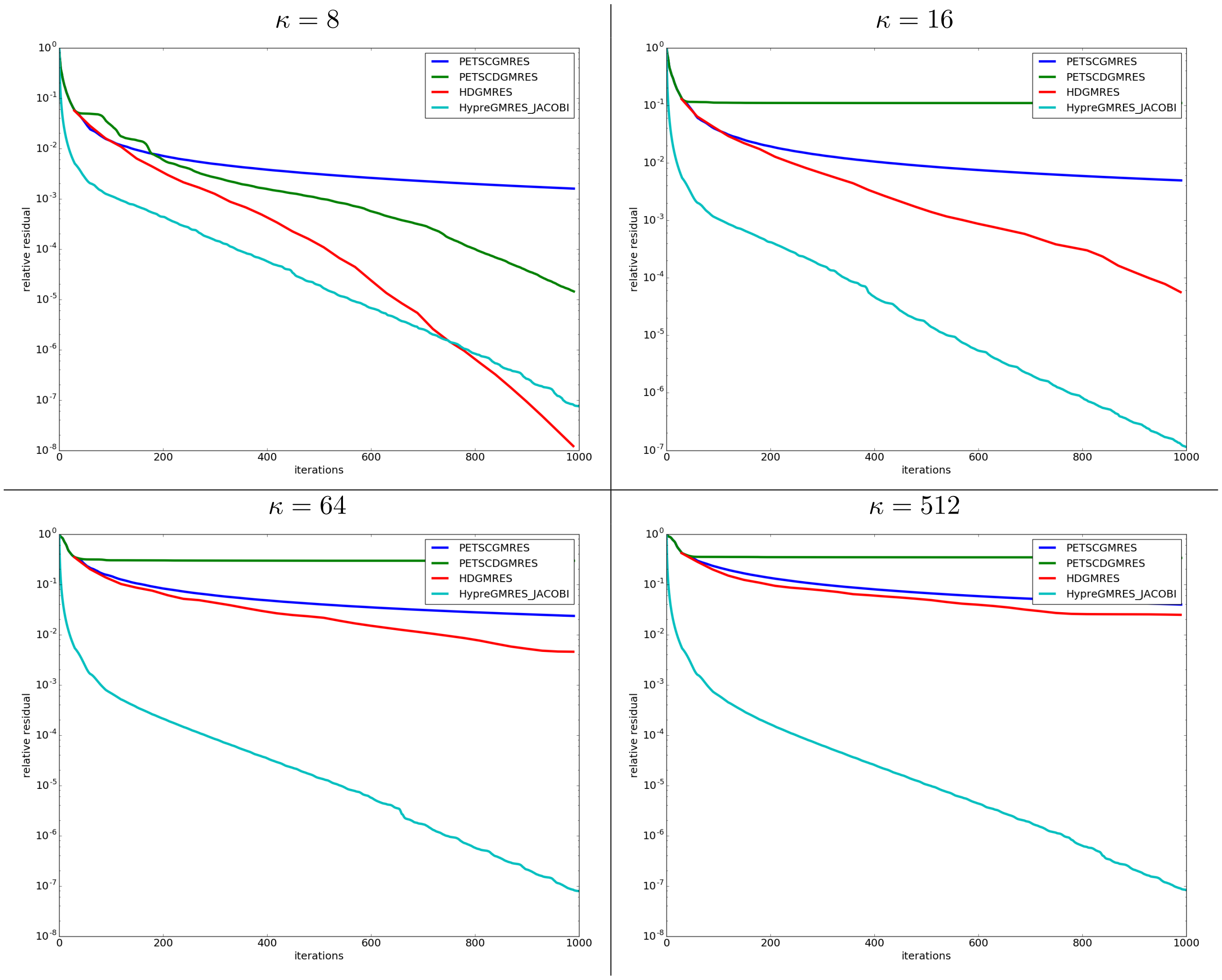


Figure 6.14: Convergence of HDGMRES for a vertically stratified diffusivity gradient, compared with the PETSc implementation of GMRES and DGMRES, and the HYPRE implementation of the 2-step Jacobi preconditioner

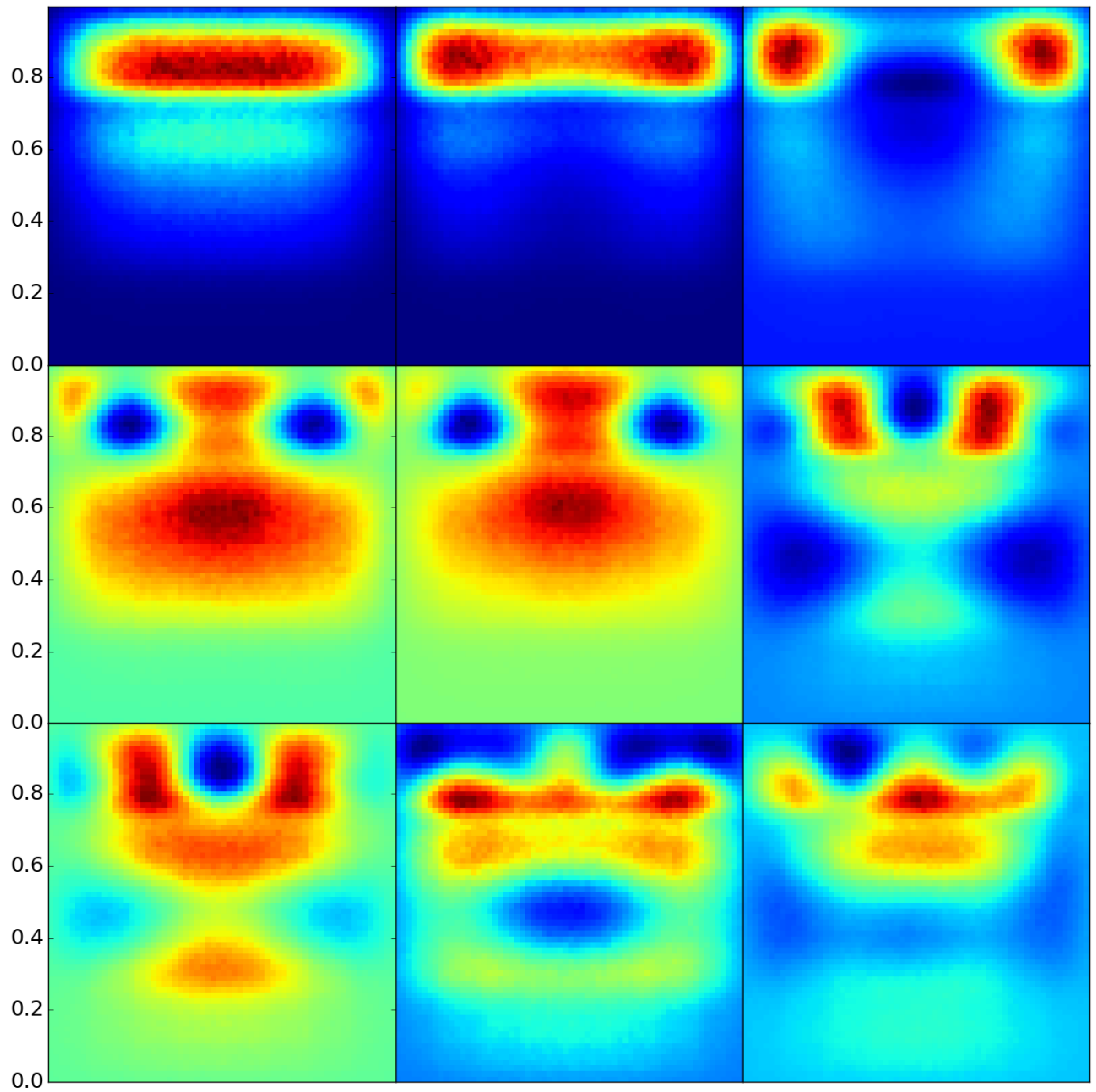


Figure 6.15: Eigenvectors from the cascading eigensolver for a vertically stratified diffusivity gradient, with $\kappa = 8$.

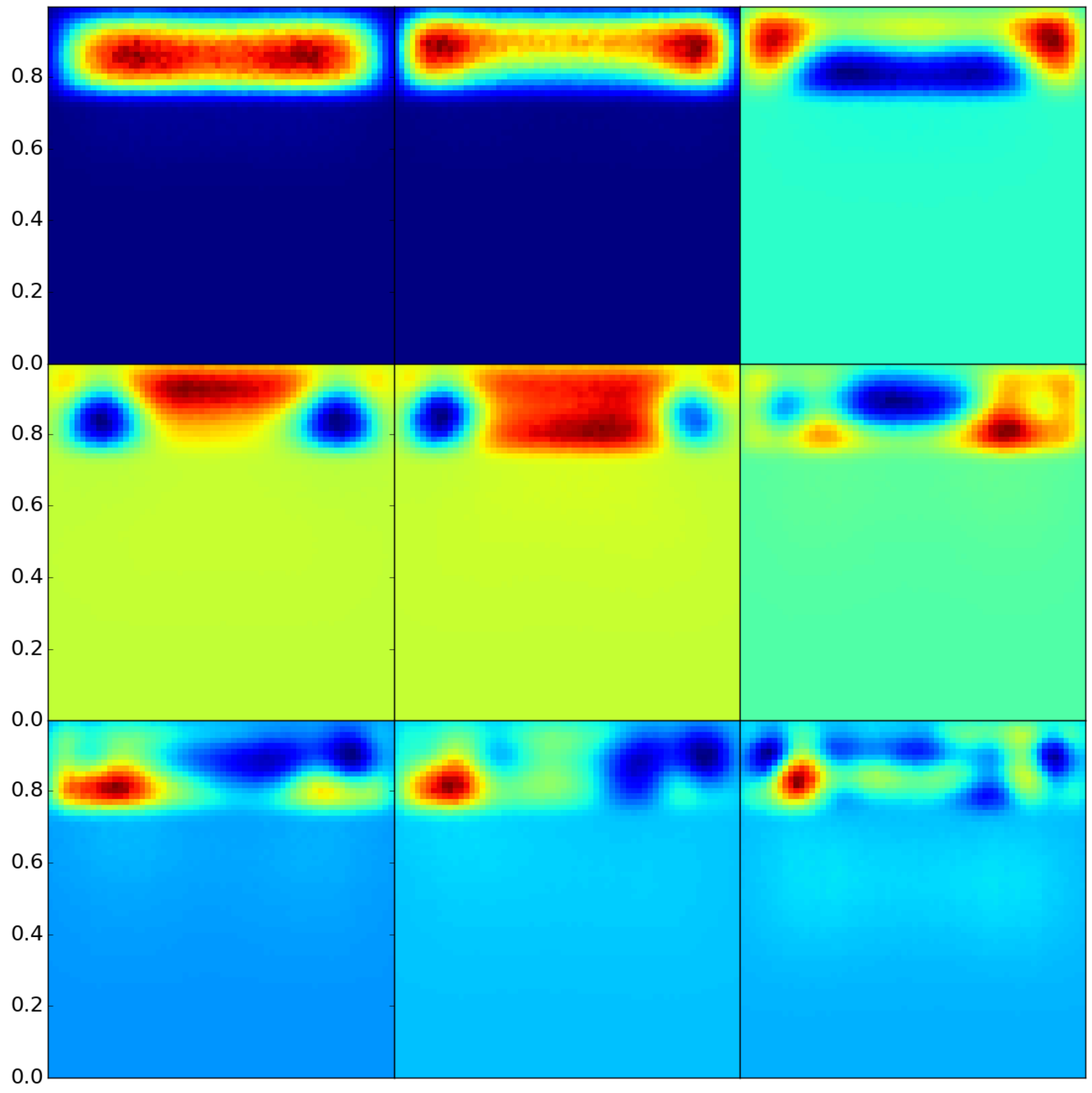


Figure 6.16: Eigenvectors from the cascading eigensolver for a vertically stratified diffusivity gradient, with $\kappa = 256$.

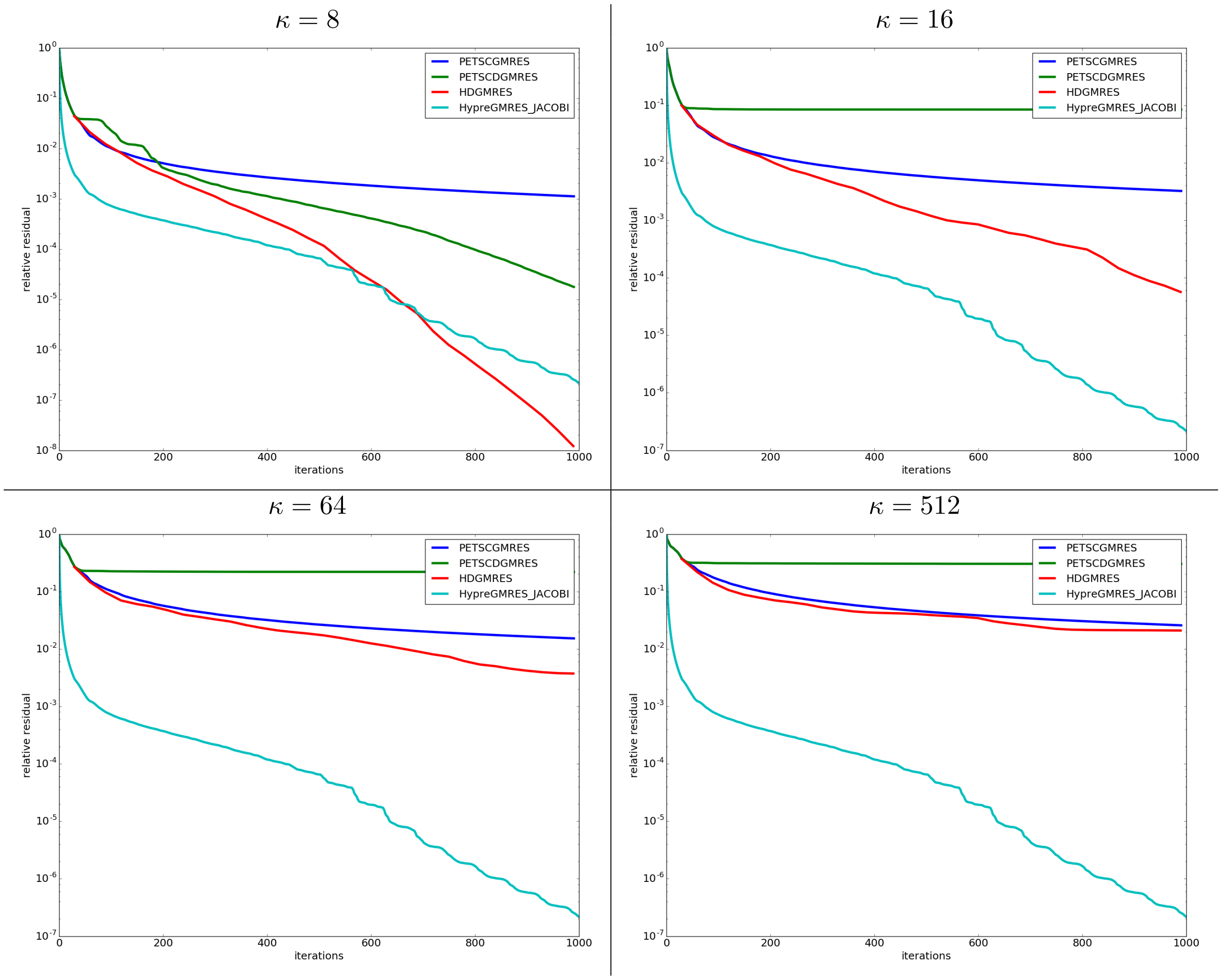


Figure 6.17: Convergence of HDGMRES for a horizontally stratified diffusivity gradient, compared with the PETSc implementation of GMRES and DGMRES, and the HYPRE implementation of the 2-step Jacobi preconditioner

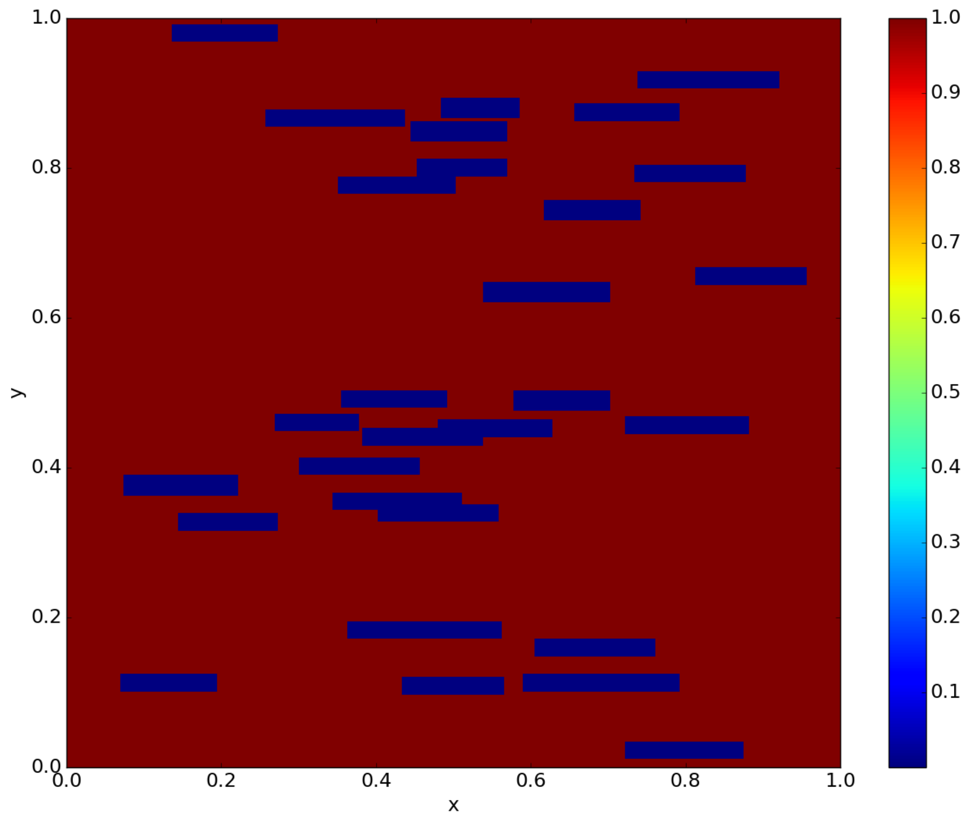


Figure 6.18: Permeability distribution for a domain composed of 10% low permeability barriers for $\kappa = 4096$

convergence than any of the other methods.

6.5 HDGMRES parallel efficiency

Parallel efficiency studies were done for problems ranging in size from 1024×1024 to 17324×17324 on configurations of 1-27 nodes, each with 12 CPU cores and 2 GPUs. Two different method configurations were used; one with large RAM requirements (Section 6.5.1), and one with lower RAM requirements (Section 6.5.2).

6.5.1 Large memory configuration

In the first configuration, HDGMRES was run with a Krylov subspace of dimension 30, and a deflation preconditioner of dimension at most 20. The run times this configuration are given in Table 6.1, the weak scaling is given in Figure 6.23, and the strong scaling is given in Figure 6.24.

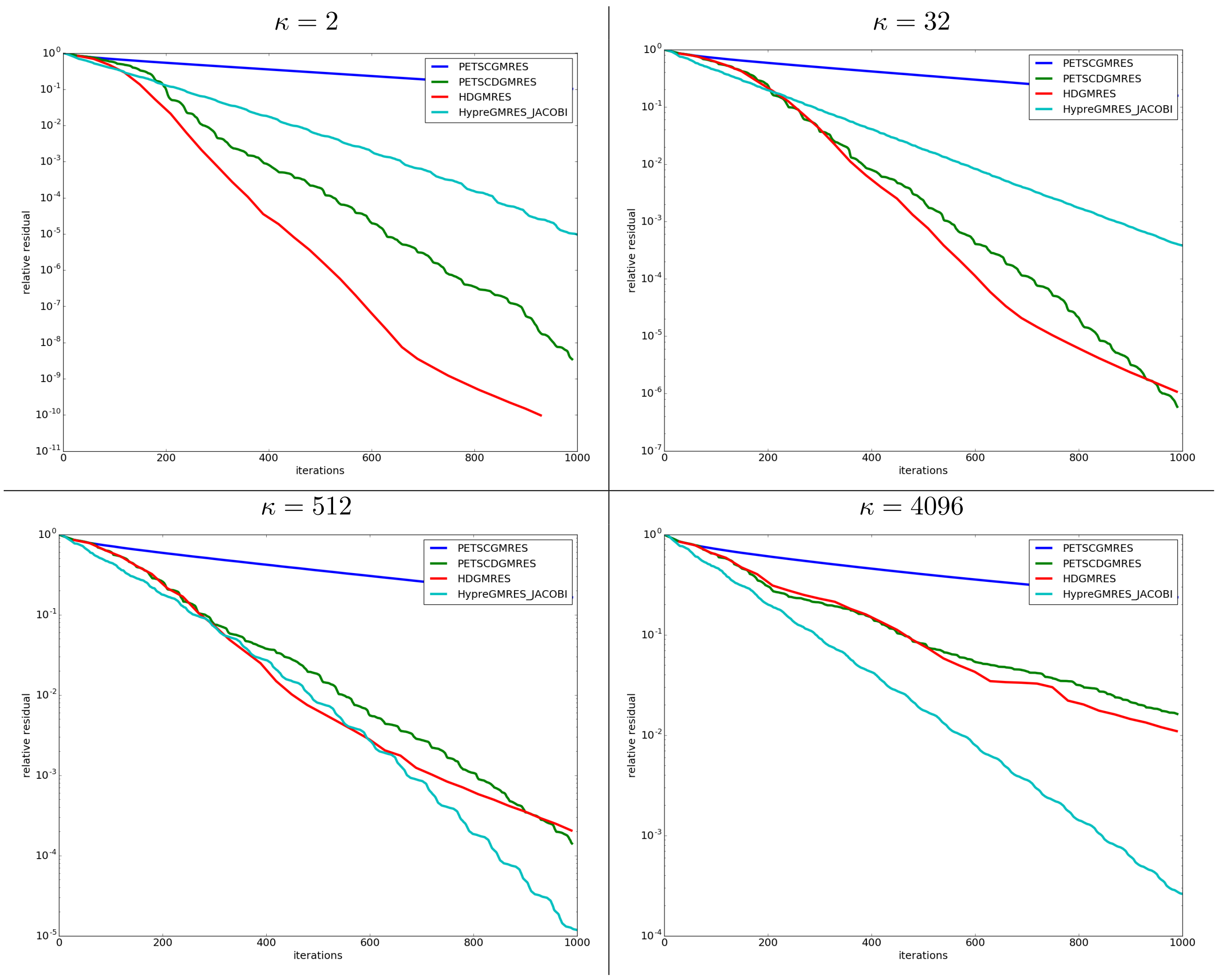


Figure 6.19: Convergence of HDGMRES for a domain with horizontally aligned low permeability barriers composing 10% of the domain. This is compared with the PETSc implementation of GMRES and DGMRES, and the HYPRE implementation of the 2-step Jacobi preconditioner

Number of nodes	1	2	4	8	16	27
1024×1024 problem	29.78	15.28	9.41	7.57	9.56	21.38
2048×2048 problem	165.77	78.95	33.55	20.98	20.78	26.47
4096×4096 problem	-	358.41	183.20	97.91	53.49	48.07
8192×8192 problem	-	-	619.23	385.74	240.85	175.62
16384×16384 problem	-	-	-	1803.83	1099.35	763.67
17324×17324 problem	-	-	-	2467.68	1698.92	950.86

Table 6.1: Run times in seconds of HDGMRES with a Krylov subspace of dimension 30, and a deflation preconditioner of dimension at most 20. Entries of "-" correspond to configurations where the number of nodes was too small to fit the problem into memory.

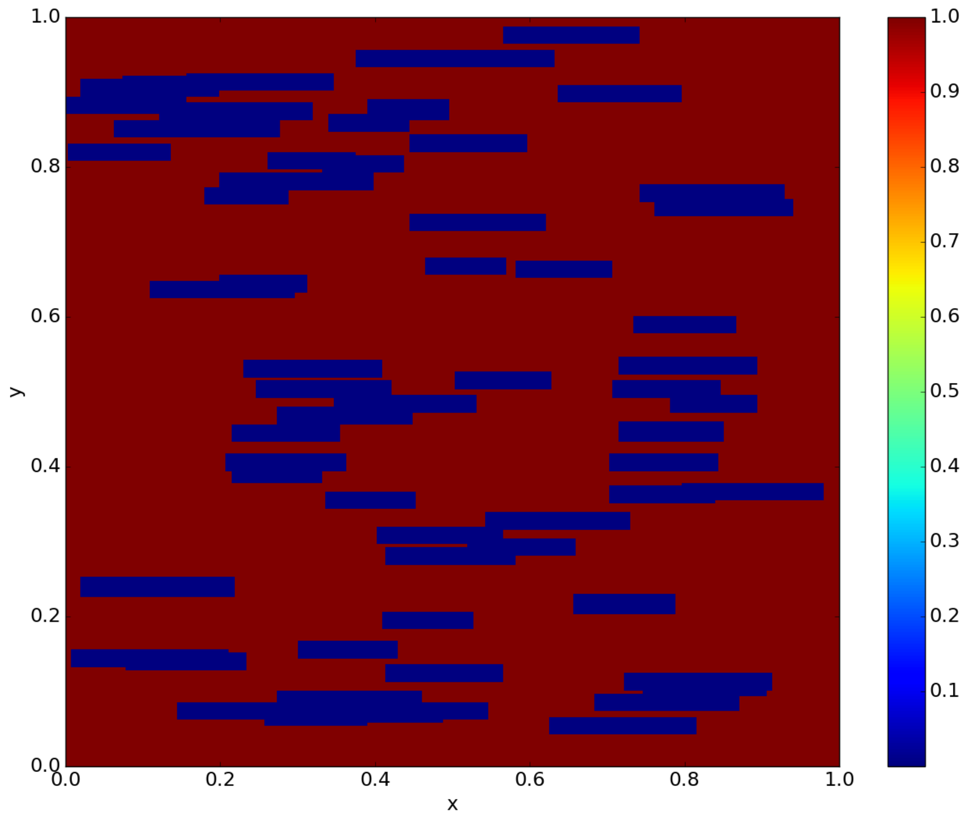


Figure 6.20: Permeability distribution for a domain composed of 20% low permeability barriers for $\kappa = 4096$

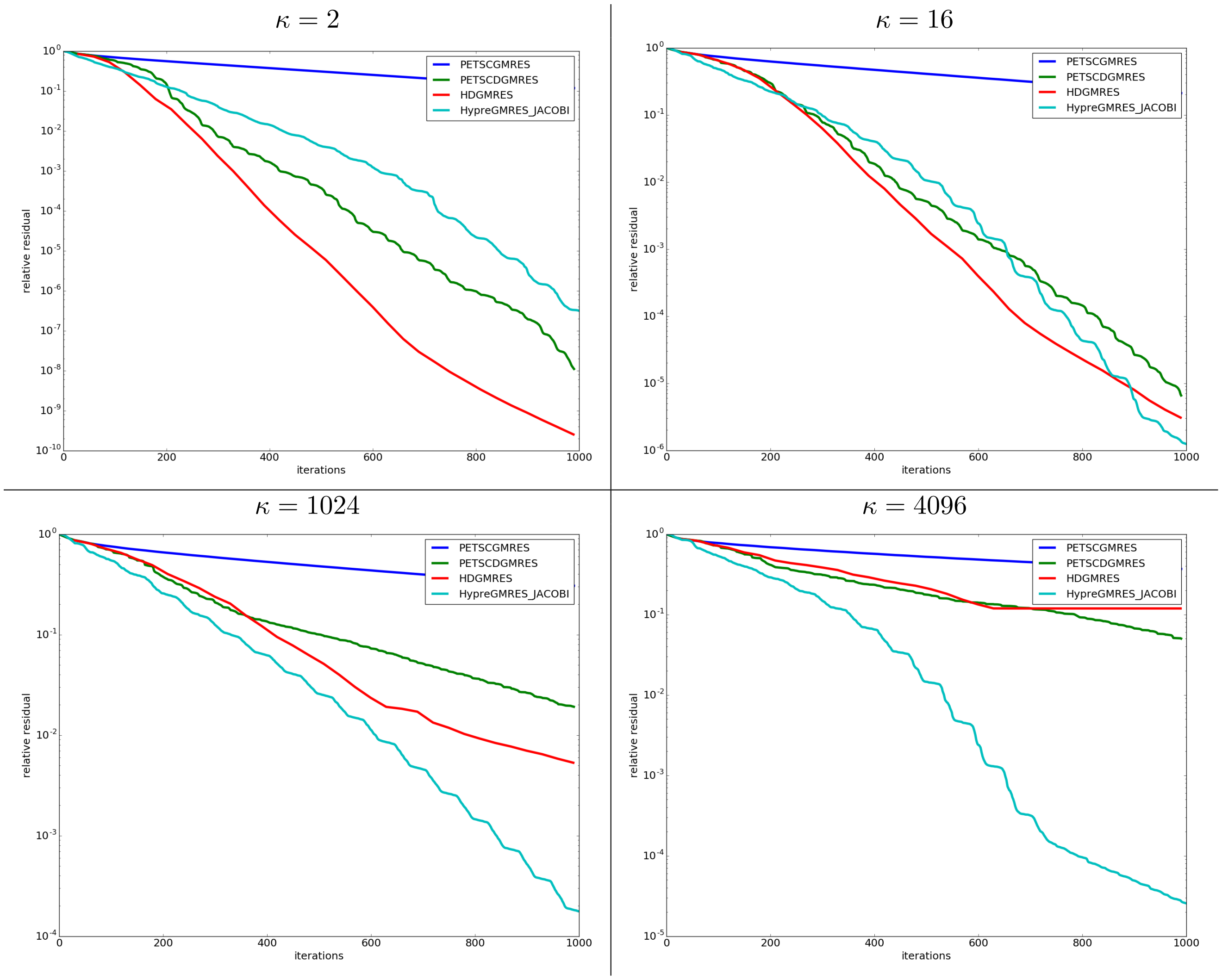


Figure 6.21: Convergence of HDGMRES for a domain with horizontally aligned low permeability barriers composing 20% of the domain. This is compared with the PETSc implementation of GMRES and DGMRES, and the HYPRE implementation of the 2-step Jacobi preconditioner

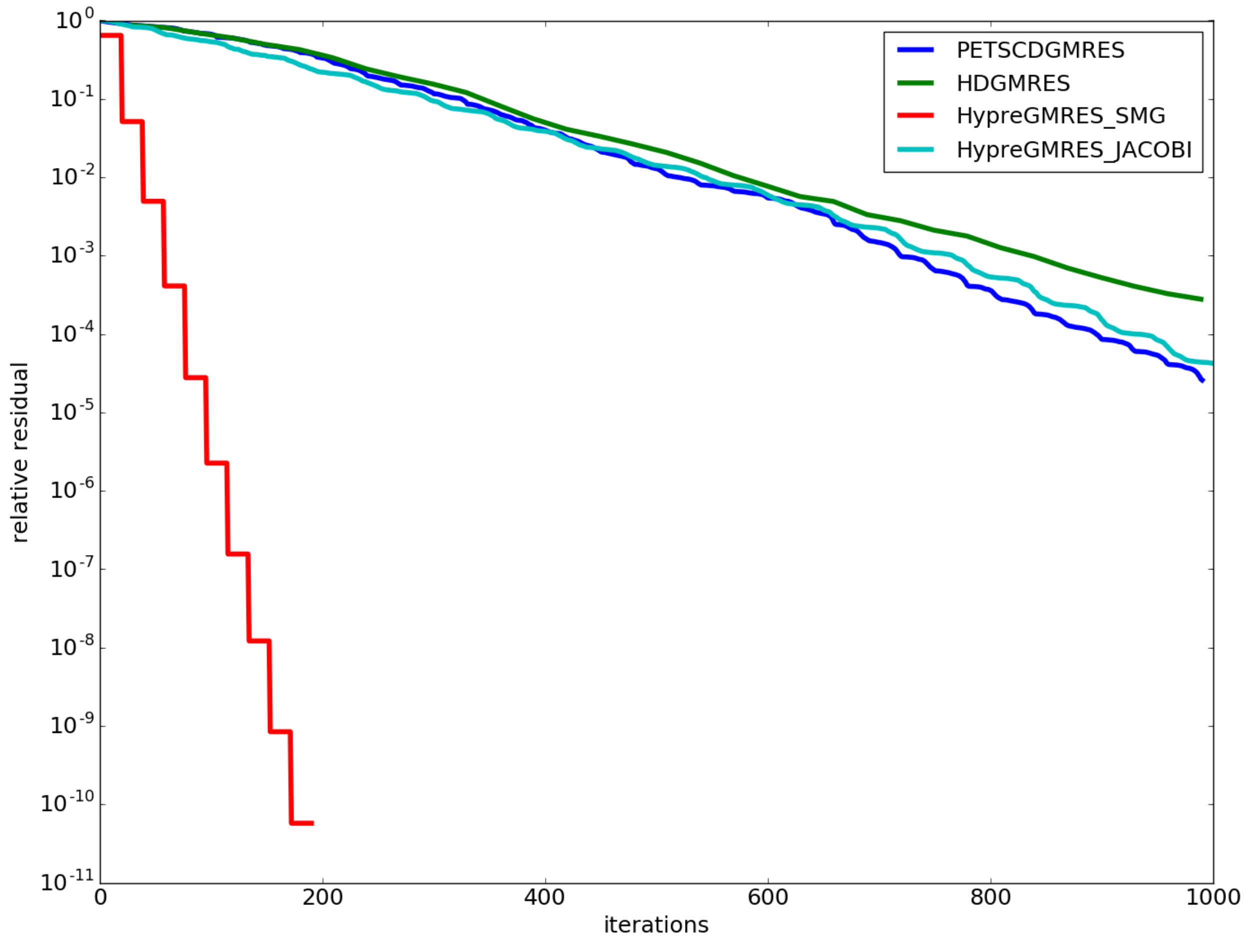


Figure 6.22: Convergence of HDGMRES for a domain with horizontally aligned low permeability barriers composing 20% of the domain, with $\kappa = 64$. This is compared with the PETSc implementation of DGMRES, and the HYPRE implementation of the 2-step Jacobi and SMG preconditioners

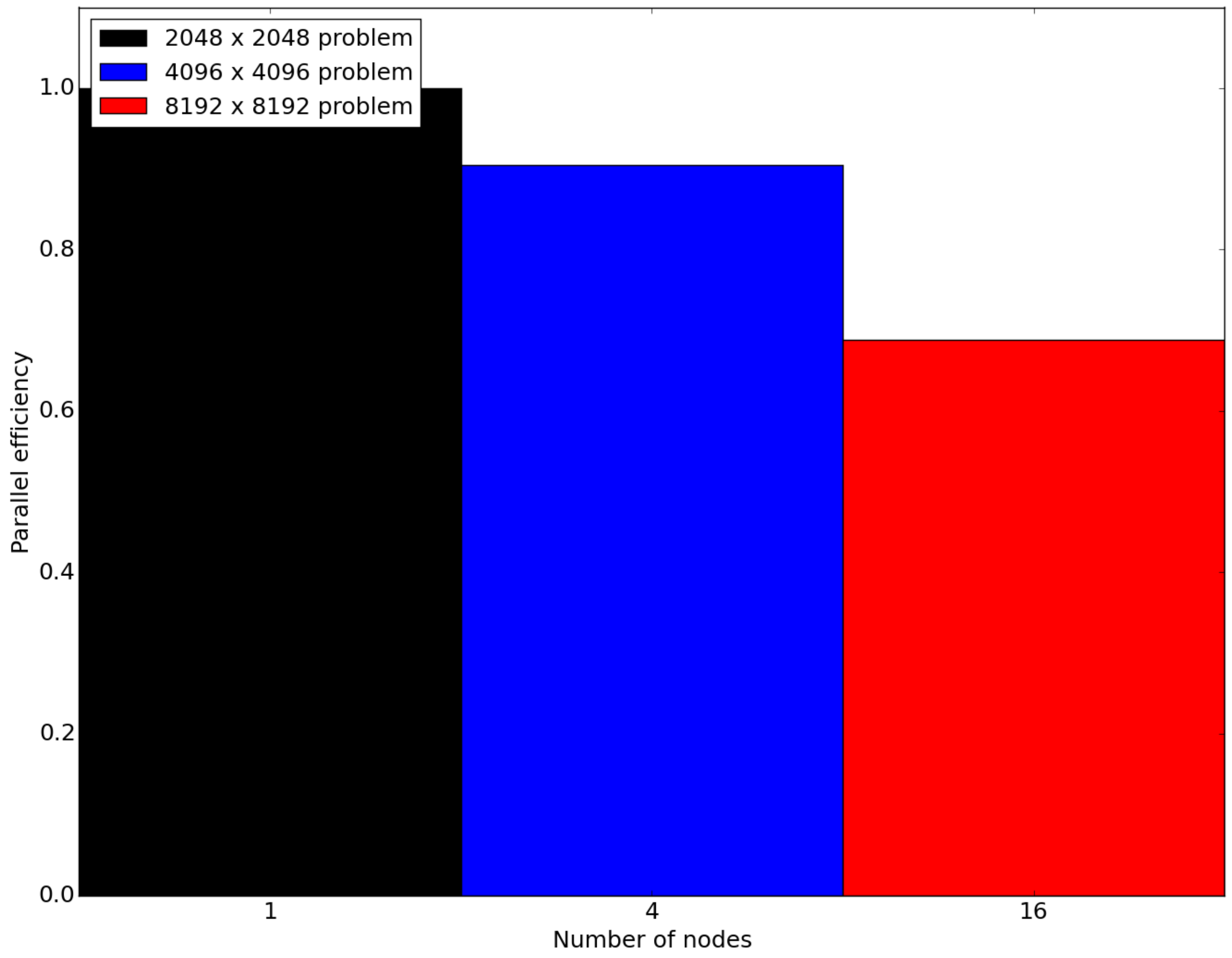


Figure 6.23: Weak scaling of HDGMRES with a Krylov subspace of dimension 30, and a deflation preconditioner of dimension at most 20

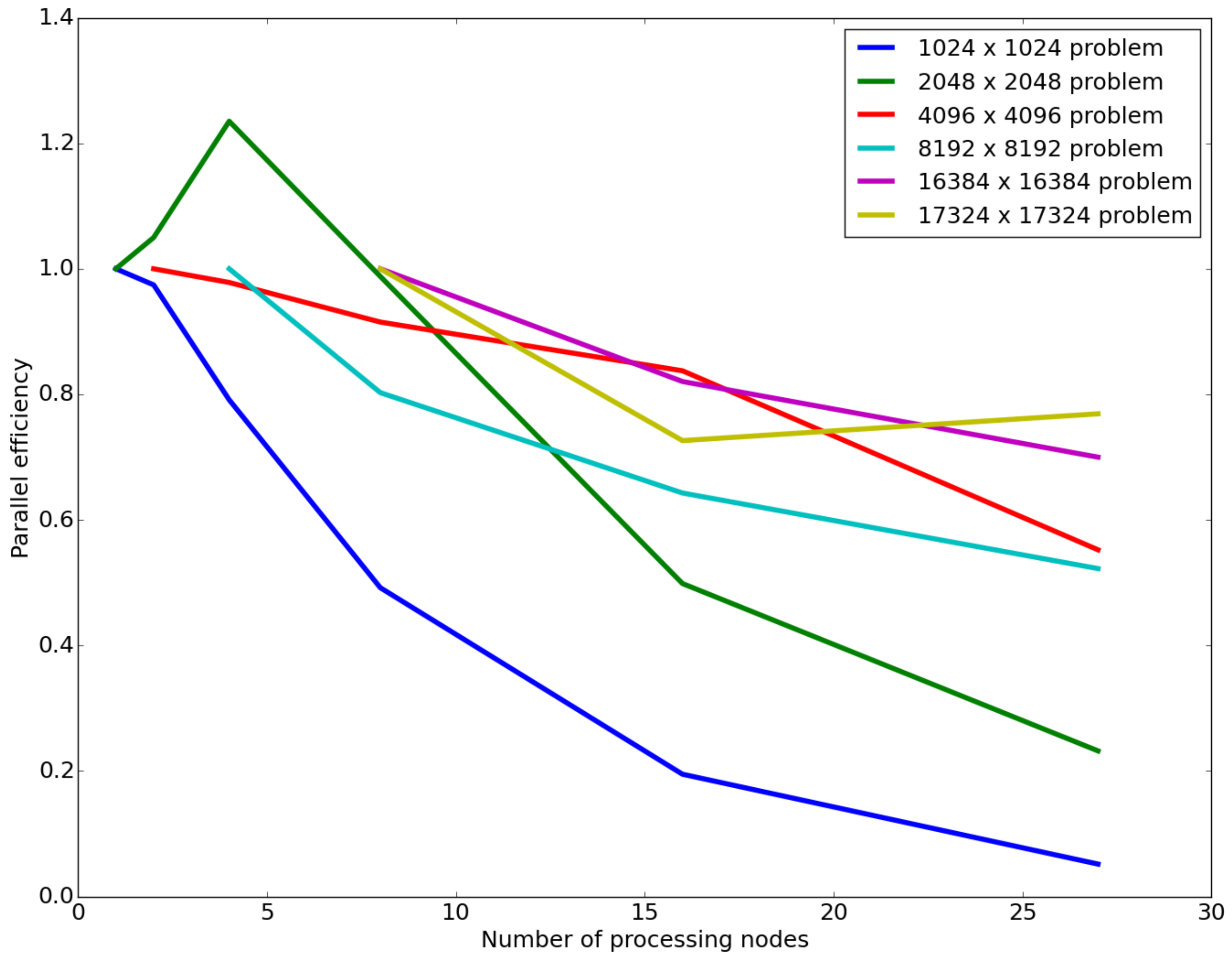


Figure 6.24: Strong scaling of HDGMRES with a Krylov subspace of dimension 30, and a deflation preconditioner of dimension at most 20

Number of nodes	1	4	8	16	27
1024×1024 problem	19.32	7.97	8.45	15.36	39.26
2048×2048 problem	106.63	23.71	18.55	23.04	44.04
4096×4096 problem	428.13	118.24	66.83	50.02	60.96
8192×8192 problem	1513.97	413.64	243.46	148.07	132.60
16384×16384 problem	-	2345.86	1117.41	749.66	479.55
17324×17324 problem	-	3296.18	1528.77	904.50	819.77

Table 6.2: Run times in seconds of HDGMRES with a Krylov subspace of dimension 15, and a deflation preconditioner of dimension at most 10. Entries of "-" correspond to configurations where the number of nodes was too small to fit the problem into memory.

6.5.2 Small memory configuration

In the second configuration, HDGMRES was run with a Krylov subspace of dimension 15, and a deflation preconditioner of dimension at most 10. This allowed for larger problem sizes to be run for the 1, 2, and 4 node configurations. The run times for this configuration are given in Table 6.2, the weak scaling is given in Figure 6.25, and the strong scaling is given in Figure 6.26.

6.5.3 Modifications for petascale implementations

The method has been shown to scale effectively up to 8-16 nodes, but with a dropoff in efficiency for larger configurations. We identify the primary bottleneck in the method as being the step in which eigenvectors are fetched from the GPUs and averaged. This can be seen clearly in comparing the timelines for a run with 8 nodes (Figure 6.27), and one with 27 nodes (Figure 6.28). A potential way to mitigate this effect is to distribute the work between the GPUs not by the number of samples, but by a full domain decomposition. In order for such a scheme to be concurrent, it would be necessary to transfer grid quantities between GPUs on the network without any involvement by the CPUs. This could be facilitated by the Remote Direct Memory Access (RDMA) feature in Compute Unified Device Architecture (CUDA) 5.0 and higher, but care would need to be taken to ensure that the networked GPU to GPU transfers don't significantly impede the networked CPU-CPU transfers occurring concurrently.

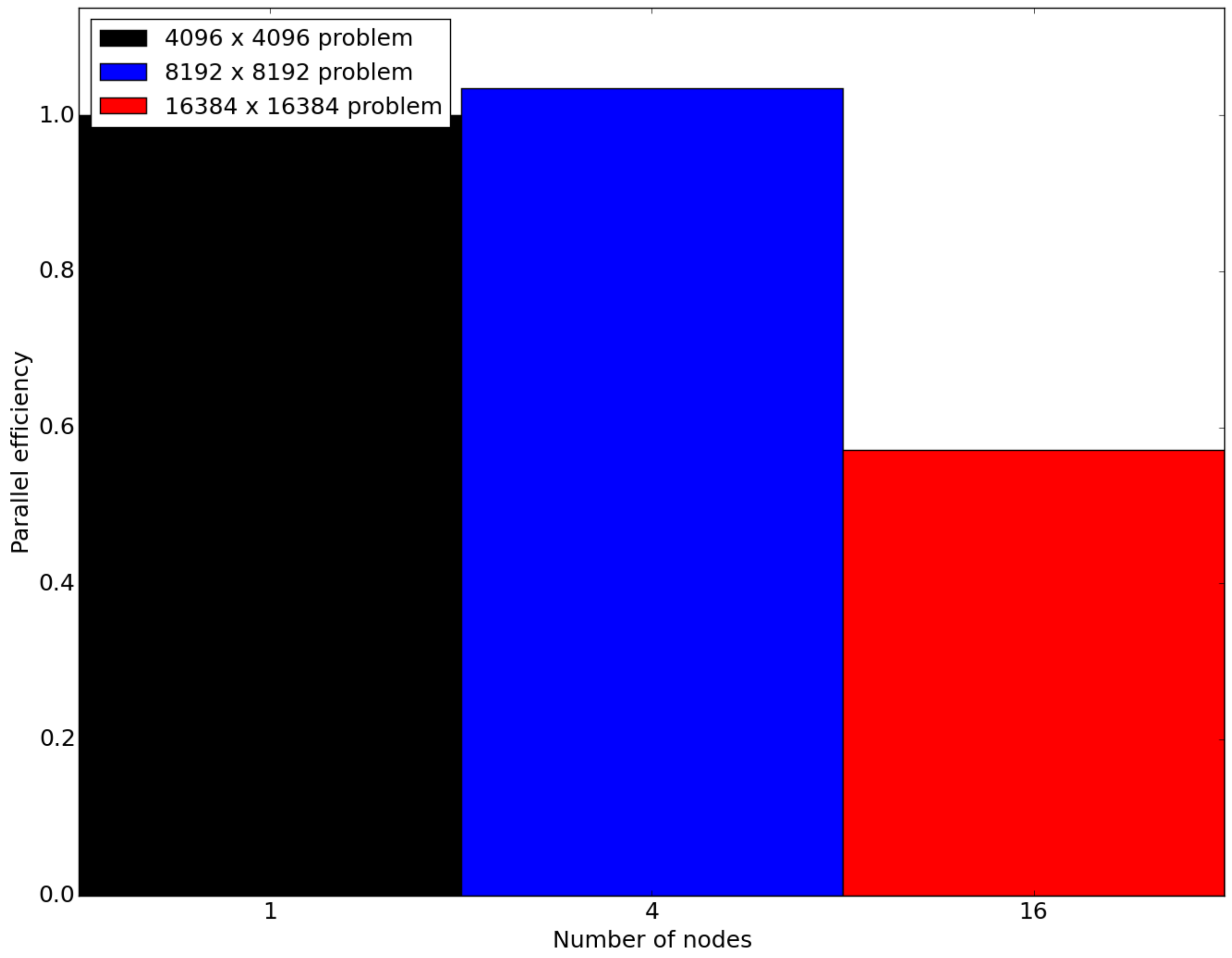


Figure 6.25: Weak scaling of HDGMRES with a Krylov subspace of dimension 15, and a deflation preconditioner of dimension at most 10

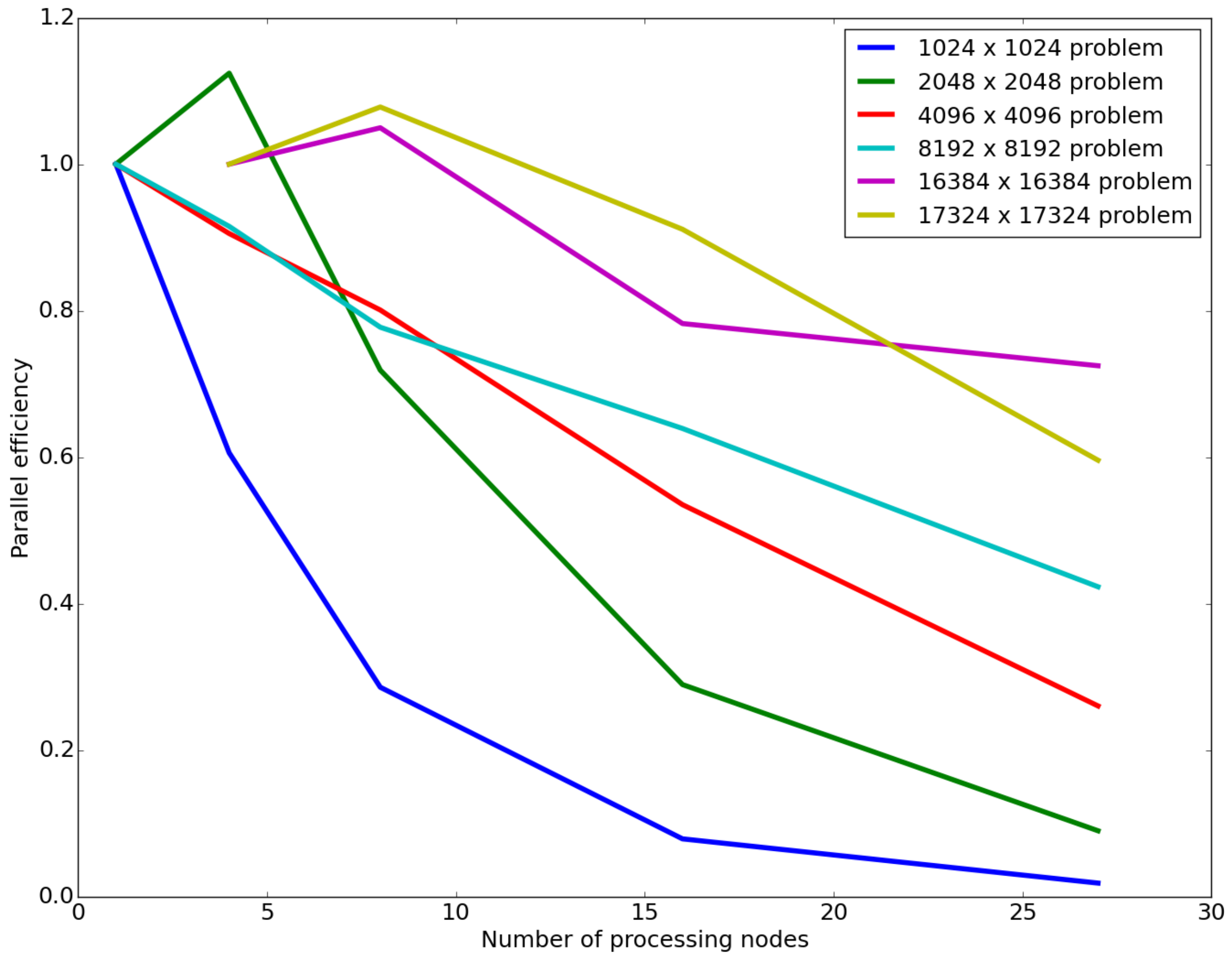


Figure 6.26: Strong scaling of HDGMRES with a Krylov subspace of dimension 15, and a deflation preconditioner of dimension at most 10

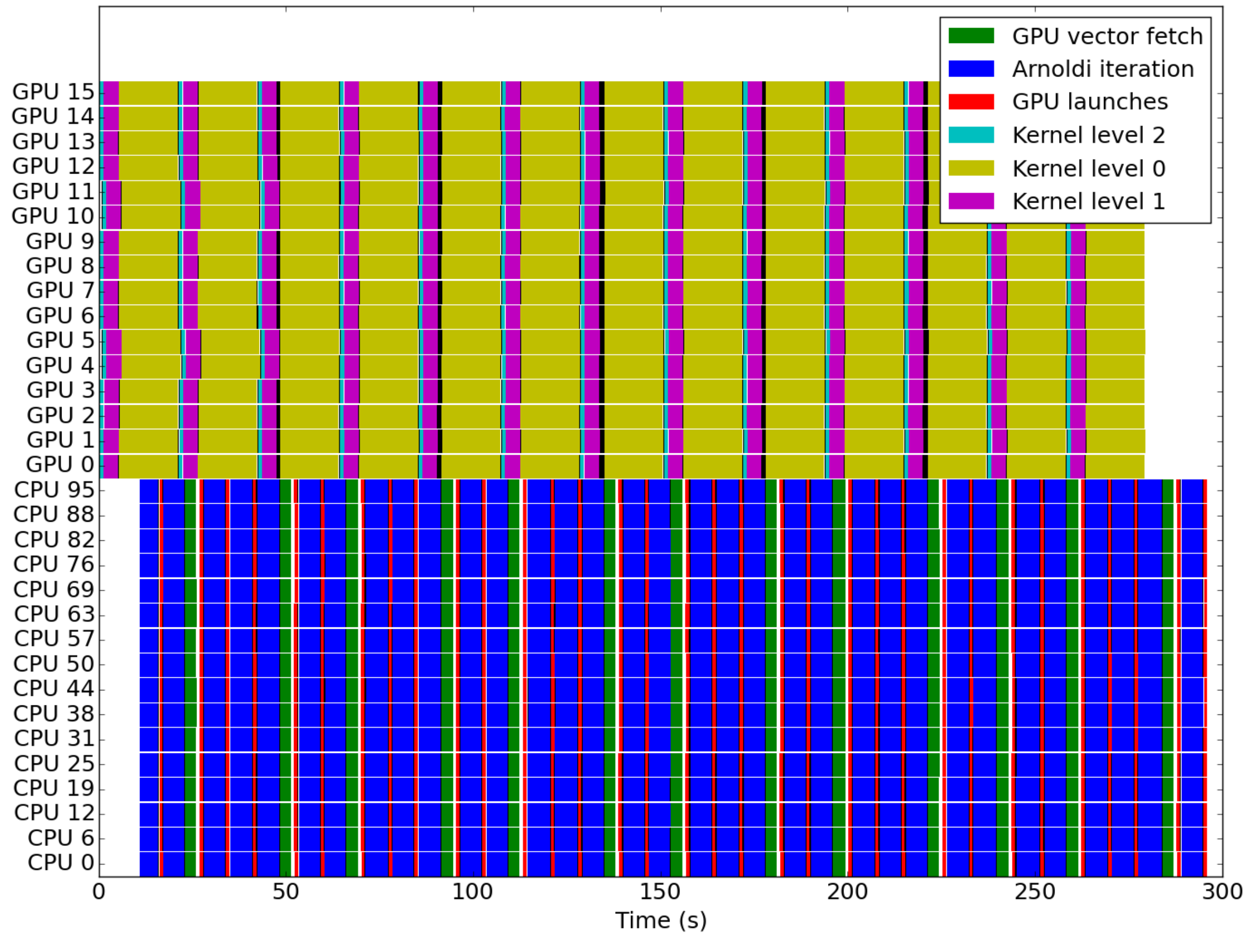


Figure 6.27: Timeline for HDGMRES with 8 nodes. The cost of gathering all 16 GPU eigenvectors for this configuration is significant.

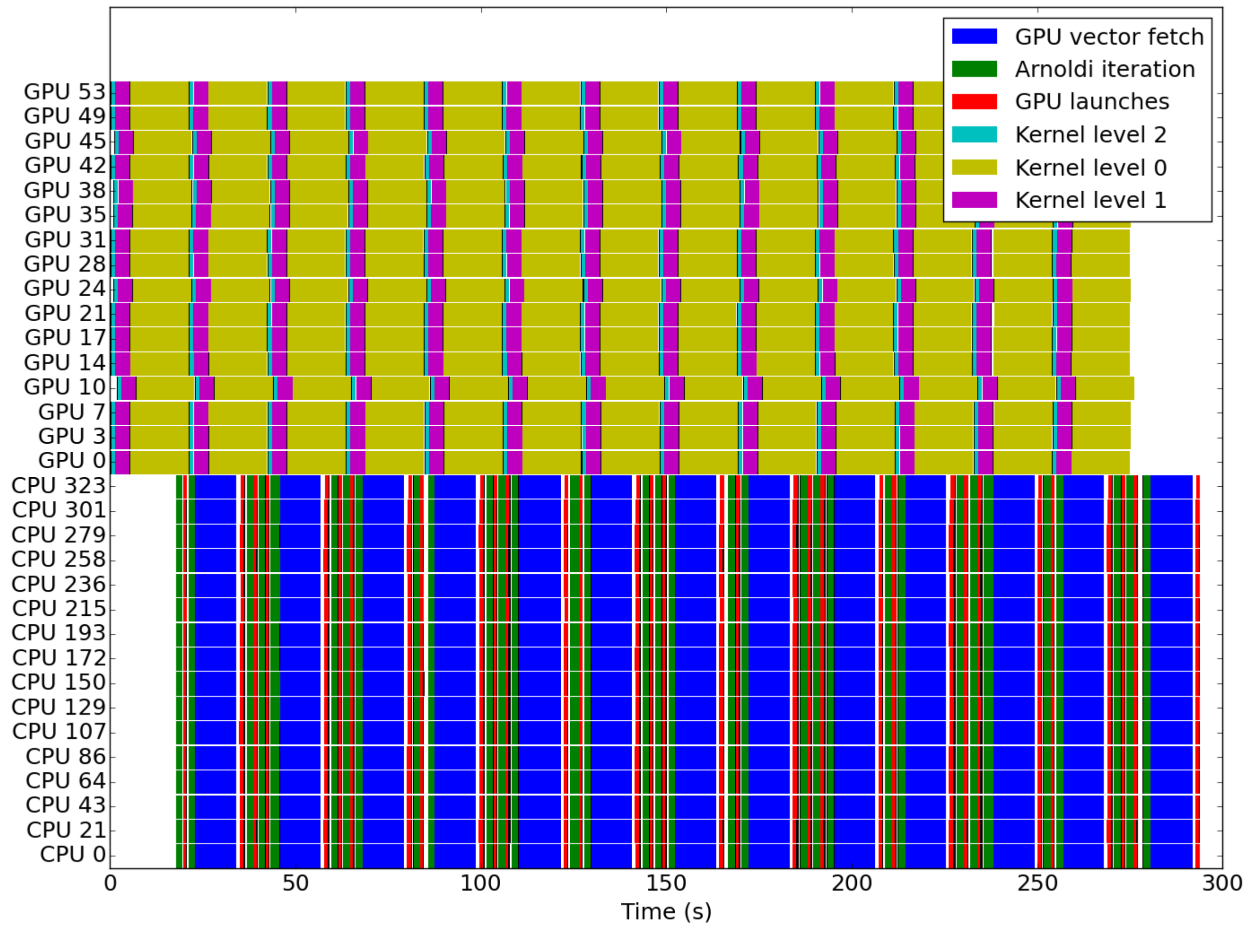


Figure 6.28: Timeline for HDGMRES with 27 nodes. The cost of gathering all 54 GPU eigenvectors for this configuration is the primary bottleneck.

CHAPTER 7

Discussion and Conclusion

7.1 Discussion

In this work, we have developed a novel parallel hybrid linear solver for elliptic equations. This development has two novel components. The first is a Feynman-Kac based eigensolver, which extends prior work on a solver for the principal eigenvector to the case of arbitrarily many eigenvectors. The eigensolver was further modified to facilitate efficient implementation on GPUs, by decomposing the domain into shared memory regions to bound the range of memory accesses by the random walks in the Feynman-Kac method. This efficient method was found to be effective at recovering eigenvectors with small associated eigenvalues, but was not as effective for eigenvectors on the high end of the spectrum. The second novel component is a concurrent computation scheme that combines the new eigensolver with a deflation preconditioner for GMRES. This new scheme allows for information to be traded between the CPU and GPU without a need for explicit synchronization between the devices. These exchanges can also occur at arbitrarily irregular intervals, allowing for both devices to be maximally utilized, regardless of their relative capabilities.

The full hybrid concurrent method, named HDGMRES, was found to outperform its parent method, DGMRES, in a number of convergence studies for varied distributions of the elliptic operator coefficients within the domain. With these coefficients interpreted as diffusivities, the method performed well for smooth diffusivity variations with the ratio between the largest and smallest diffusivities in the range $\kappa = 1 - 512$, which was representative of solutions with large components in eigenvectors at the low end of the spectrum. For sharply stratified variations, the method was found to stagnate for $\kappa > 64$, which was representative of solutions with large components throughout the spectrum. For a similar case, with a random placement of low diffusivity barriers with permeability $1/\kappa$, the method was found to converge more slowly, but not stagnate for increasing κ. The relative performance of these cases is attributed to the relative quality of the eigenvectors recovered by the eigensolver, which was shown earlier to perform best on the lower end of the spectrum.

The method was also shown to exhibit a parallel efficiency of 80% in the low-medium terascale, as demonstrated on a problem with 300 million variables, run on a configuration of 324 CPU cores and 54 GPUs. For the larger configurations, the most significant bottleneck in the method was identified as the gather operation required to average the random walk samples across all of the GPUs.

7.2 Future Directions

The immediate future developments for the current method involve improving the performance of the eigensolver on the higher end of the spectrum, and improving the parallelism to make the method viable at the high terascale and low petascale. For improving the performance of the eigensolver, one possible modification of the Feynman-Kac evolution algorithm is to create an analog of adaptive mesh refinement, where coarser subdomains with larger timesteps are used for low diffusivity regions, and finer subdomains with smaller timesteps are used for high diffusivity regions.

For the development of the method to larger computational scales, the two issues to be addressed are domains that are too large to fit into a single GPU's memory, and the bottleneck of gathering samples from multiple GPUs. The most direct remedy both of these issues is to use a domain decomposition technique on the GPUs, but this will come with its own challenges in managing networked GPU to GPU transfers without slowing down those of the CPUs.

7.3 Conclusion

In the broader scope of highly parallel linear solvers on heterogeneous architectures, the method presented in this work has demonstrated the viability of using a concurrently computed GPU preconditioner to improve upon the CPU-only version of the same preconditioner. This approach in general, where CPUs and GPUs are never explicitly synchronized, has the potential to be highly effective for large scale heterogeneous computing, where the cost of networked transfers and synchronization is high.

REFERENCES

[1] NVIDIA Corporation, "High Performance Computing for Servers | Tesla GPUs | NVIDIA." http://www.nvidia.com/object/tesla-servers.html.

[2] Intel Corporation, "Intel Xeon Phi Product Family." http://www.intel.com/content/www/us/en/processors/xeon/xeon-phi-detail.html.

[3] Intel Corporation, "Intel Core i7 Processors." http://www.intel.com/content/www/us/en/processors/core/core-i7-processor.html.

[4] "June 2014 | TOP500 Supercomputer Sites." http://www.top500.org/lists/2014/06/.

[5] S. Rusu, H. Muljono, D. Ayers, S. Tam, W. Chen, A. Martin, S. Li, S. Vora, R. Varada, and E. Wang, "A 22 nm 15-Core Enterprise Xeon Processor Family," *IEEE Journal of Solid-State Circuits*, vol. 50, no. 1, pp. 35–48, 2015.

[6] J. Fang, A. L. Varbanescu, H. Sips, L. Zhang, Y. Che, and C. Xu, "An Empirical Study of Intel Xeon Phi," *arXiv preprint*, no. Section III, pp. 137–148, 2013.

[7] G. Dhatt, E. Lefrançois, and G. Touzot, *Finite element method.* John Wiley & Sons, 2012.

[8] S. A. Sauter and C. Schwab, *Boundary element methods.* Berlin Heidelberg: Springer, 2011.

[9] I. Babuška, A. Craig, J. Mandel, and J. Pitkäranta, "Efficient preconditioning for the p-version finite element method in two dimensions," *SIAM Journal on Numerical Analysis*, vol. 28, no. 3, pp. 624–661, 1991.

[10] M. J. Grote and T. Huckle, "Parallel preconditioning with sparse approximate inverses," *SIAM Journal on Scientific Computing*, vol. 94305, pp. 1–25, 1997.

[11] M. Kalos and P. A. Whitlock, *Monte carlo methods.* 2008.

[12] S. Chen and G. Doolen, "Lattice Boltzmann method for fluid flows," *Annual review of fluid mechanics*, vol. 30, no. 1, pp. 329–364, 1998.

[13] S. Borkar and A. Chien, "The future of microprocessors," *Communications of the ACM*, vol. 54, no. 5, pp. 67–77, 2011.

[14] F. Bustamante, "Efficient wire formats for high performance computing," in *Supercomputing, ACM/IEEE 2000 Conference*, 2000.

[15] J. Reinders, "An Overview of Programming for Intel Xeon processors and Intel Xeon Phi coprocessors," tech. rep., Intel Corporation, 2012.

[16] J. Jeffers and J. Reinders, *Intel Xeon Phi coprocessor high performance programming.* Newnes, 2013.

[17] E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym, "NVIDIA Tesla: A unified graphics and computing architecture," *IEEE Micro*, vol. 28, no. 2, pp. 39–55, 2008.

[18] V. W. Lee, C. Kim, J. Chhugani, M. Deisher, D. Kim, A. D. Nguyen, N. Satish, and M. Smelyanskiy, "Debunking the 100X GPU vs . CPU Myth : An Evaluation of Throughput Computing on CPU and GPU," *ACM SIGARCH Computer Architecture News*, vol. 38, no. 3, pp. 451–460, 2010.

[19] A. Danalis, G. Marin, and C. McCurdy, "The scalable heterogeneous computing (SHOC) benchmark suite," in *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units*, pp. 63–74, 2010.

[20] M. Deisher, M. Smelyanskiy, B. Nickerson, V. W. Lee, M. Chuvelev, and P. Dubey, "Designing and dynamically load balancing hybrid LU for multi/many-core," *Computer Science - Research and Development*, vol. 26, pp. 211–220, apr 2011.

[21] A. Heinecke, K. Vaidyanathan, M. Smelyanskiy, A. Kobotov, R. Dubtsov, G. Henry, A. Shet, G. Chrysos, and P. Dubey, "Design and Implementation of the Linpack Benchmark for Single and Multi-node Systems Based on Intel Xeon Phi Coprocessor," in *2013 IEEE 27th International Symposium on Parallel and Distributed Processing*, pp. 126–137, Ieee, may 2013.

[22] S. Barrachina, M. Castillo, and F. D. Igual, "Solving dense linear systems on graphics processors," in *Euro-Par 2008 - Parallel Processing*, pp. 739–748, 2008.

[23] F. D. Igual, G. Quintana-Orti, and R. A. van de Geijn, "Level-3 BLAS on a GPU : Picking the Low Hanging Fruit," tech. rep., 2009.

[24] E. Saule, K. Kaya, and Ü. V. Çatalyürek, *Performance evaluation of sparse matrix multiplication kernels on intel xeon phi.* Berlin, Heidelberg: Springer, feb 2014.

[25] M. Wang, H. Klie, M. Parashar, and H. Sudan, "Solving sparse linear systems on NVIDIA Tesla GPUs," in *Computational Science- ICCS*, pp. 864–873, 2009.

[26] C. Yang, C. Huang, and C. Lin, "Hybrid CUDA, OpenMP, and MPI parallel programming on multicore GPU clusters," *Computer Physics Communications*, vol. 182, pp. 266–269, jan 2011.

[27] K. W. Schulz and R. Ulerich, "Early experiences porting scientific applications to the Many Integrated Core (MIC) platform," in *TACC-Intel Highly Parallel Computing Symposium*, (Schulz2012), 2012.

[28] N. Rajovic, L. Vilanova, C. Villavieja, N. Puzovic, and A. Ramirez, "The low power architecture approach towards exascale computing," *Journal of Computational Science*, vol. 4, pp. 439–443, nov 2013.

[29] I. Journal, S.-s. Circuits, R. Intel, H. M. Intel, and R. V. Intel, "A 65-nm Dual-Core Multithreaded Xeon Processor With 16-MB L3 Cache," no. April 2016, 2007.

[30] S. Rusu, S. Tam, H. Muljono, J. Stinson, D. Ayers, J. Chang, R. Varada, M. Ratta, S. Kottapalli, and S. Vora, "A 45 nm 8-Core Enterprise Xeon Processor," *IEEE Journal of Solid-State Circuits*, vol. 45, no. 1, pp. 7–14, 2010.

[31] L. Seiler, R. Cavin, R. Espasa, E. Grochowski, T. Juan, P. Hanrahan, D. Carmean, E. Sprangle, T. Forsyth, M. Abrash, P. Dubey, S. Junkins, A. Lake, and J. Sugerman, "Larrabee: A Many-Core x86 Architecture for Visual Computing," *ACM Transactions on Graphics*, vol. 27, no. 3, p. 1, 2008.

[32] G. Chrysos, "Intel Xeon Phi Coprocessor - the Architecture," *Proceedings of the 24th Hot Chips Symposium*, pp. 1–8, 2012.

[33] A. Duran and M. Klemm, "The Intel many integrated core architecture," *Proceedings of the 2012 International Conference on High Performance Computing and Simulation, HPCS 2012*, no. mic, pp. 365–366, 2012.

[34] K. H. Law, "A parallel finite element solution method," *Computers & Structures*, vol. 23, pp. 845–858, jan 1986.

[35] C. Farhat and F. Roux, "An unconventional domain decomposition method for an efficient parallel solution of large-scale finite element systems," *SIAM Journal on Scientific and Statistical Computing*, vol. 13, no. 1, pp. 379–396, 1992.

[36] T. Tezduyar, S. Aliabadi, M. Behr, A. Johnson, and S. Mittal, "Parallel finite-element computation of 3D flows," *Computer*, vol. 26, no. 10, pp. 27–36, 1993.

[37] W. Bangerth, C. Burstedde, T. Heister, and M. Kronbichler, "Algorithms and data structures for massively parallel generic adaptive finite element codes," *ACM Transactions on Mathematical Software*, vol. 38, no. 2, pp. 1–28, 2011.

[38] O. Sahni, M. Zhou, M. S. Shephard, and K. E. Jansen, "Scalable implicit finite element solver for massively parallel processing with demonstration to 160K cores," in *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis - SC '09*, (New York, New York, USA), p. 1, ACM Press, 2009.

[39] C. Burstedde, O. Ghattas, and G. Stadler, "Towards adaptive mesh PDE simulations on petascale computers," in *Teragrid*, 2008.

[40] P. Fischer, J. Lottes, D. Pointer, and A. Siegel, "Petascale algorithms for reactor hydrodynamics," *Journal of Physics: Conference Series*, vol. 125, jul 2008.

[41] C. Burstedde, O. Ghattas, and M. Gurnis, "Scalable adaptive mantle convection simulation on petascale supercomputers," in *ACM/IEEE conference on Supercomputing*, no. November, 2008.

[42] J. Bolz, I. Farmer, E. Grinspun, and P. Schroder, "Sparse Matrix Solvers on the GPU: Conjugate Gradients and Multigrid," *ACM Transactions on Graphics (TOG)*, vol. 22, no. 3, pp. 917–924, 2003.

[43] A. Dziekonski, P. Sypek, A. Lamecki, and M. Mrozowski, "Finite Element Matrix Generation on a GPU," *Progress In Electromagnetics Research*, vol. 128, pp. 249–265, 2012.

[44] C. Cecka, A. J. Lew, and E. Darve, "Assembly of Finite Element Methods on Graphics Processors," vol. 85, no. 5, pp. 1–6, 2002.

[45] D. Goddeke and S. H. M. Buijssen, "GPU acceleration of an unmodified parallel finite element Navier-Stokes solver," in *High Performance Computing & Simulation*, 2009.

[46] D. Jacobsen, J. Thibault, and I. Senocak, "An MPI-CUDA Implementation for Massively Parallel Incompressible Flow Computations on Multi-GPU Clusters," in *48th AIAA Aerospace Sciences Meeting Including the New Horizons Forum and Aerospace Exposition*, (Reston, Virigina), American Institute of Aeronautics and Astronautics, 2010.

[47] W. Wu and P. A. Heng, “A hybrid condensed finite element model with GPU acceleration for interactive 3D soft tissue cutting,” *Computer Animation and Virtual Worlds*, vol. 15, pp. 219–227, jul 2004.

[48] G. R. Joldes, A. Wittek, and K. Miller, “Real-Time Nonlinear Finite Element Computations on GPU - Application to Neurosurgical Simulation.,” *Computer methods in applied mechanics and engineering*, vol. 199, pp. 3305–3314, dec 2010.

[49] A. Klöckner, T. Warburton, J. Bridge, and J. S. Hesthaven, “Nodal discontinuous Galerkin methods on graphics processors,” *Journal of Computational Physics*, vol. 228, no. 21, pp. 7863–7882, 2009.

[50] N. Godel, N. Nunn, T. Warburton, and M. Clemens, “Scalability of higher-order discontinuous galerkin FEM computations for solving electromagnetic wave propagation problems on GPU clusters,” *IEEE Transactions on Magnetics*, vol. 46, no. 8, pp. 3469–3472, 2010.

[51] A. S. Antoniou, K. I. Karantasis, E. D. Polychronopoulos, and J. A. Ekaterinaris, “Acceleration of a Finite-Difference WENO Scheme for Large-Scale Simulations on Many-Core Architectures,” *Computer Engineering*, no. January, pp. 1–12, 2010.

[52] A. R. Brodtkorb and M. L. Sætra, *Shallow Water Simulations on Multiple GPUs*. Springer Berlin Heidelberg, 2012.

[53] D. Komatitsch, D. Michéa, and G. Erlebacher, “Porting a high-order finite-element earthquake modeling application to NVIDIA graphics cards using CUDA,” *Journal of Parallel and Distributed Computing*, vol. 69, pp. 451–460, may 2009.

[54] D. Komatitsch, G. Erlebacher, D. Göddeke, and D. Michéa, “High-order finite-element seismic wave propagation modeling with MPI on a large GPU cluster,” *Journal of Computational Physics*, vol. 229, pp. 7692–7714, oct 2010.

[55] J. Zhou, Y. Cui, E. Poyraz, D. J. Choi, and C. C. Guest, “Multi-GPU Implementation of a 3D finite difference time domain earthquake code on heterogeneous supercomputers,” *Procedia Computer Science*, vol. 18, pp. 1255–1264, 2013.

[56] M. Rietmann, P. Messmer, T. Nissen-meyer, D. Peter, and P. Basini, “Forward and Adjoint Simulations of Seismic Wave propagation on Emerging Large Scale GPU Architectures,” in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, 2012.

[57] B. S. Kirk, J. W. Peterson, R. H. Stogner, and G. F. Carey, “libMesh : a C++ library for parallel adaptive mesh refinement/coarsening simulations,” *Engineering with Computers*, vol. 22, pp. 237–254, nov 2006.

[58] C. Baker, G. Davidson, T. M. Evans, S. Hamilton, J. Jarrell, and W. Joubert, “High performance radiation transport simulations: Preparing for TITAN,” *International Conference for High Performance Computing, Networking, Storage and Analysis*, 2012.

[59] T. Dong, V. Dobrev, T. Kolev, R. Rieben, S. Tomov, and J. Dongarra, “A step towards energy efficient computing: Redesigning a hydrodynamic application on CPU-GPU,” *Proceedings of the International Parallel and Distributed Processing Symposium, IPDPS*, no. June, pp. 972–981, 2014.

[60] Y. Saad and M. H. Schultz, "GMRES: A Generalized Minimal Residual Algorithm for Solving Nonsymmetric Linear Systems," *SIAM Journal on Scientific and Statistical Computing*, vol. 7, no. 3, pp. 856–869, 1986.

[61] M. Embree, "How descriptive are GMRES convergence bounds?," 1999.

[62] V. Simoncini, "On the convergence of restarted Krylov subspace methods," *Siam Journal on Matrix Analysis and Applications*, vol. 22, no. 2, pp. 430–452, 2000.

[63] S. C. Eisenstat, H. C. Elman, and M. H. Schultz, "Variational Iterative Methods for Nonsymmetric Systems of Linear Equations," *SIAM Journal on Numerical Analysis*, vol. 20, no. 2, pp. 345–357, 1983.

[64] K. Burrage and J. Erhel, "On the Performance of Various Adaptive Preconditioned GMRES Strategies," *Numerical Linear Algebra with Applications*, vol. 1, no. February 1997, pp. 1–20, 1997.

[65] R. B. Morgan, "A Restarted GMRES Method Augmented with Eigenvectors," *SIAM Journal on Matrix Analysis and Applications*, vol. 16, no. 4, pp. 1154–1171, 1995.

[66] R. B. Morgan, "Computing interior eigenvalues of large matrices," *Linear Algebra and Its Applications*, vol. 154-156, no. C, pp. 289–309, 1991.

[67] R. B. Morgan and M. Zeng, "Estimates for interior eigenvalues of large nonsymmetric matrices," tech. rep., tech. rep, 1996.

[68] V. S. Manoranjan and M. O. Gomez, "A Two-Step Jacobi-Type Iterative Method," *Computers and Mathematics with Applications*, vol. 34, no. 1, pp. 1–9, 1997.

[69] D. Sorensen, "Implicit Application of Polynomial Filters in a k-step Arnoldi Method," *SIAM Journal on Matrix Analysis and Applications*, vol. 13, no. 1, pp. 357–385, 1992.

[70] J. Baglama, D. Calvetti, G. H. Golub, and L. Reichel, "Adaptively Preconditioned GMRES Algorithms," *SIAM Journal on Scientific Computing*, vol. 20, no. 1, pp. 243–269, 1998.

[71] J. Erhel, K. Burrage, and B. Pohl, "Restarted GMRES preconditioned by deflation," *Journal of Computational and Applied Mathematics*, vol. 69, no. 2, pp. 303–318, 1996.

[72] "Hypre User's Manual," tech. rep., 2015.

[73] R. Wienands, C. W. Oosterlee, and T. Washio, "Fourier Analysis of GMRES(m) Preconditioned by Multigrid," *SIAM Journal on Scientific Computing*, vol. 22, no. 2, pp. 582–603, 2000.

[74] S. Ashby and R. Falgout, "A parallel multigrid preconditioned conjugate gradient algorithm for groundwater flow simulations," *Nuclear Science and Engineering*, vol. 124, no. 1, pp. 145–159, 1996.

[75] R. D. Falgout and J. E. Jones, "Multigrid on massively parallel architectures," *Multigrid Methods VI*, vol. 14, pp. 101–107, 2000.

[76] S. Schaffer, "A Semicoarsening Multigrid Method For Elliptic Partial Differential Equations With Highly Discontinuous And Anisotropic Coefficients," *SIAM Journal on Scientific Computing*, vol. 20, no. 1, pp. 228–242, 1998.

[77] P. N. Brown, R. D. Falgout, and J. E. Jones, “Semicoarsening multigrid on distributed memory machines,” *SIAM Journal on Scientific Computing*, vol. 21, no. 5, pp. 1823–1834, 2000.

[78] M. Kac, “On distributions of certain Wiener functionals,” *Trans. Amer. Math. Soc*, 1949.

[79] M. K. Chati, M. D. Grigoriu, S. S. Kulkarni, and S. Mukherjee, “Random walk method for the two- and three-dimensional Laplace, Poisson and Helmholtz’s equations,” *International Journal for Numerical Methods in Engineering*, vol. 51, pp. 1133–1156, aug 2001.

[80] D. Higham, “An algorithmic introduction to numerical simulation of stochastic differential equations,” vol. 43, no. 3, pp. 525–546, 2001.

[81] G. N. Milstein, *Numerical integration of stochastic differential equations.* Dordrecht: Kluwer Academic, 1995.

[82] G. N. Milstein, *Stochastic Numerics for Mathematical Physics.* Berlin: Springer, 2004.

[83] W. Rümelin, “Numerical treatment of stochastic differential equations,” *SIAM Journal on Numerical Analysis*, vol. 151, pp. 859–60, feb 1982.

[84] N. J. Newton, “Asymptotically efficient Runge-Kutta methods for a class of Ito and Stratonovich equations,” *SIAM Journal on Applied Mathematics*, vol. 51, no. 2, pp. 542–567, 1991.

[85] P. E. Kloeden and E. Platen, *Numerical Solution of Stochastic Differential Equations.* Berlin: Springer, 1999.

[86] A. Rossler, “Second order Runge-Kutta methods for Itô stochastic differential equations,” *SIAM Journal on Numerical Analysis*, vol. 47, no. 3, pp. 1713–1738, 2009.

[87] A. Rössler, “Runge-Kutta methods for the strong approximation of solutions of stochastic differential equations,” *SIAM Journal on Numerical Analysis*, vol. 48, no. 3, pp. 922–952, 2010.

[88] A. Lejay and S. Maire, “Computing the principal eigenvalue of the Laplace operator by a stochastic method,” *Mathematics and computers in simulation*, pp. 1–21, 2007.

[89] A. Lejay and S. Maire, “Computing the principal eigenelements of some linear operators using a branching Monte Carlo method,” *Journal of Computational Physics*, vol. 227, pp. 9794–9806, dec 2008.

[90] J. K. Salmon, M. A. Moraes, R. O. Dror R, and D. E. Shaw, “Parallel Random Numbers: As Easy as 1, 2, 3,” in *2011 International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2011.

[91] M. Schröck and H. Vogt, “Gauge fixing in lattice QCD with multi-GPUs,” in *Acta Physica Polonica B, Proceedings Supplement*, vol. 6, pp. 763–768, 2013.

[92] J. Jaros, “Multi-GPU island-based genetic algorithm for solving the knapsack problem,” in *2012 IEEE Congress on Evolutionary Computation, CEC 2012*, pp. 10–15, 2012.

[93] R. Nabben and C. Vuik, “A Comparison of Deflation and Coarse Grid Correction Applied to Porous Media Flow,” *SIAM Journal on Numerical Analysis*, vol. 42, no. 0, pp. 1631–1647, 2004.

[94] C. P. Green and J. Ennis-King, “Effect of vertical heterogeneity on long-term migration of CO2 in saline formations,” *Transport in Porous Media*, vol. 82, no. 1, pp. 31–47, 2010.