

TABLE OF CONTENTS

1	Introduction	1
2	Related Work	7
2.1	GPUS for explicit and implicit methods	7
2.2	GPUS/RBF	7
2.3	RBF/Finite-Difference	7
2.3.1	Global RBF Method	7
2.3.2	RBF-FD	7
2.4	Preconditioners	7
2.5	Sparse matrix libraries	7
2.6	Various GPU-based libraries	7
2.6.1	CUSP	7
2.6.2	ViennaCL	7
2.7	Parallel implementations of RBF	8
2.8	Contributions of this Work	8
3	Fragments (integrate above)	9
3.1	Fragments	10
3.2	PDE as an Interpolation Problem	13
3.3	Reconstructing Solutions for PDEs	15
3.4	Kansa's Method	16
3.5	Fasshauer's Method	16
3.6	Direct Collocation	17
3.7	Local Methods	18
3.8	RBF-FD	19
3.9	Time Stepping	19
I	Preliminaries	21
4	RBF Methods for PDEs	22
4.1	Survey of Related Work	22
4.1.1	Global RBF Methods	24
4.1.2	Compactly Support RBFs	25
4.1.3	Local RBF Methods	27

4.1.4	RBF-FD	27
4.1.5	Recent Advances in Conditioning	27
4.1.6	Parallel RBF Implementations	28
4.2	PDE as an Interpolation Problem	29
4.3	Reconstructing Solutions for PDEs	30
4.4	Global RBF Method	31
4.5	Kansa's Method	33
4.6	Fasshauer's Method	34
4.7	Direct Collocation	35
4.8	Local Methods	35
4.9	RBF-FD	36
4.10	Time Stepping	37
II	The RBF-FD Method	39
5	The RBF-FD Method	40
6	RBF Methods for PDE Solutions	41
6.1	The RBF-generated Finite Differences Method	41
6.2	Stencil-based Derivative Approximation	42
6.3	RBF-FD weights	42
6.4	Weight Operators	44
6.4.1	Laplacian (∇^2)	44
6.4.2	Laplace-Beltrami (Δ_S) on the Sphere	44
6.4.3	Constrained Gradient ($P_x \cdot \nabla$) on the Sphere	45
6.5	Stabilization: Hyperviscosity	46
7	TEMP	48
7.1	Fragments (integrate above)	48
7.2	Approximate Nearest Neighbor (ANN) Query	49
7.2.1	k -D Tree	49
8	Parallel Solvers	50
8.1	On the use of libraries for parallel solvers	51
8.2	Partitioning	51
8.3	Local node ordering	52
8.4	Two level parallelism	55
8.4.1	Overlapping Computation and Communication	55
8.4.2	Explicit Solvers	56
8.4.3	Fragments (integrate above)	60
8.4.4	Implicit Solvers	60
8.5	RBF-FD as SpMV	61
9	Sparse Matrix Multiplication on the GPU	62

10 GMRES	63
11 Numerical Validation	64
11.1 Choosing ϵ	64
11.1.1 CVT epsilon tests	64
11.2 Parabolic	64
11.3 Hyperbolic	64
11.3.1 Vortex Rollup	66
11.3.2 Solid body rotation	68
11.4 Fragments (integrate above)	71
11.4.1 CFL	72
11.5 GFLOP Throughput	72
11.6 Elliptic	73
12 Stokes	74
12.1 Introduction	74
12.2 Bad Problem	77
12.2.1 Details	77
12.3 RBF-FD Weights	77
12.4 GPU Based Solver	78
12.4.1 GMRES Algorithm	78
12.4.2 Multiple GPUs	79
12.5 Multiple GPUs	80
12.5.1 Solution Ordering	80
12.6 Governing Equations	81
12.6.1 Constraints	84
12.6.2 Manufactured Solution	84
12.7 Preconditioning	85
12.8 GMRES Results	86
12.9 Fragments	86
13 Performance Benchmarks	87
13.1 Metrics	87
13.2 OpenCL	87
13.2.1 OpenCL vs CUDA	87
13.2.2 Asynchronous Queuing	88
13.3 Fermi Architecture	88
13.3.1 Double Precision	88
13.3.2 Local Caching	88
13.3.3 Multiple Kernel Scheduling	88
13.3.4 Future NVidia Hardware	88
13.4 HPC Spear Cluster	89
13.5 Keeneland	89
13.6 Future Hardware	89
13.7 MPI_Alltoall	89

13.8 Asynchronous OpenCL	92
13.9 Multi-Queue OpenCL	92
13.10 GPU Kernel Optimizations	92
13.10.1 Work-Group Size and Number of Stencils	92
13.10.2 Parallel Reduction in Shared Memory	92
13.10.3 Comparison: custom SpMV for explicit schemes vs ViennaCL	92
14 Neighbor Queries and Node Ordering	93
14.1 Neighbor Queries	93
14.1.1 k-D Tree	93
14.1.2 Locality Sensitive Hashing	93
14.2 Node Ordering	93
15 Community Outreach	96
A Avoiding Pole Singularities with RBF-FD	97
B Projected Weights on the Sphere	99
B.1 Direct Weights	99
B.2 Indirect Weights	100
B.3 Test Cases	100
B.4 Conclusions	101

CHAPTER 1

INTRODUCTION

outline:

- high performance computing demand for numerical modeling
 - complexity of physics terms plus the complexity of numerical methods push the limits of hardware
 - problem since the dawn of computing: limited registers/ALUs, unlimited computation. how do we triage?
 - utilize every available piece of hardware, parallelism, etc.
 - new algorithms think smarter, not just faster.
- a new era of numerical methods ushered by Radial Basis functions
 - scattered nodes
 - * place samples where they are necessary, not where the grid requires
 - * maximize the representation of nodes and minimize interpolation error
 - * higher order accuracy than standard
 -

Unanswered Questions:

- If RBF-FD is higher order accurate than FV, why would anyone opt for FV?
 - the conservative form FV is preferred in physical science since it does not allow loss of (what? mass/energy/???)
 - FV are typically low-order. what is the trade-off in complexity between methods. For a much higher-order RBF-FD if it is not conservative, how many iterations is it accurate consistent with the low-order FV? 1

Background on GPUs. GPUs were introduced in 80's see master's thesis.

Originally GPUs were designed as parallel rasterizing units. They had limited logic control in contrast to the serial CPUs and their advanced branching and looping logic.

Gradually new and complex logic was added to the GPU to produce the shader languages that allowed developers to customize specific parts of the rendering pipeline. This allowed

scientific problems such as the diffusion equation [cite Lore and others](#) to be solved in process of rendering. In other words, the GPU was tricked into computing.

The year 2006 brought the modern age of GPU computing with the introduction of CUDA from NVidia. The high level language allowed scientists to leverage the GPU as a parallel accelerator without all of the overhead of setting up graphics contexts and tricking the hardware into computing. Memory management is still the developer's responsibility, but compiler transforms generic C/Fortran code to GPU instruction set.

Scientific Computing has seen a widespread adoption of GPGPU because of the goal to get to "exa-scale" computing, which may only be possible in the near future with the help of GPU accelerators [?].

NVidia is not the only company involved in many core parallel accelerators. Other groups like AMD and Intel have been increasing the number of cores as well. The end effect is a hybridization where CPUs look similar to GPUs and vice-versa.

Until 2009, the hardware distinction required that developers target parallelism on CPUs and GPUs using different languages. Then the OpenCL standard was drafted and implemented. OpenCL is a parallel language that strives to provide functional portability rather than performance.

We focus on the OpenCL language within this dissertation with confidence that hardware will change frequently. In fact, every 18 months [cite](#) shows a new release of GPU hardware, manycore CPU hardware and extensions to parallel languages. But if hardware is constantly changing, then we need to focus on a high level implementation that allows portability. We need a language like OpenCL to carry our implementations into the future regardless of what hardware and which company survive.

Scientific problems and the need for computational methods. Many scientific problems of importance can be expressed as a collection of partial differential equations defined for some domain. In order to solve these problems , computational numerical methods are employed on a discretized version of the domain. Traditionally, three major categories exist for PDE solutions: [finite difference \(FD\)](#), [finite element \(FEM\)](#) and [spectral element \(SEM\)](#) [?]. Interestingly, all three of these methods rely on an underlying mesh, making them *meshed methods*. While each has had a turn in the spotlight, more recently a new category, or rather a generalization on all three previous categories, has emerged: *meshfree methods*.

Traditional methods (FD, FV, FEM, SEM) and their dependence on structured meshes and "nice" connections (e.g., Delaunay).. Note, finite element allows for triangle meshes, but well balanced Delaunay are preferred to limit ill conditioning. The same regularity in nodes is preferred with RBF methods. The reason for this is to keep best conditioning of the system.

Leaders in computational science are cobbled together with metaphorical bubble gum and duct tape. Many scientists neglect to plan in advance for items like:

- generic point clouds; most methods require a point cloud include some connectivity information for structured and unstructured mesh.
- we need to add adaptive mesh refinement
- we need to avoid pole singularities

- save on interpolations and differences but get similar answer in $O(nm)$.

recently, meshfree methods surfaced leveraging RBFs.

RBFs have interesting history. Started with interpolation, but went to collocation with global SEM scheme. The global scheme has $O(N^2)$ complexity..

Most recently RBFs went to RBF-FD. Introduces sparsity and reduces cost of scaling..

For RBF problems in general there is limited work that scales the methods to large problems..

Large problems require high performance computing. Given lack of scaling in literature, it has been irrelevant until recently to assess the possibility of solving PDEs with RBF-FD on GPUs..

what is new in the thesis (summarize: 1-2 pages). The large scale contributions of this thesis can be summarized as follows:

- This thesis covers details related to the application of the RBF-FD method for both explicit and implicit solutions for PDES.
- Our domain decomposition algorithm provides the first known parallelization of RBF-FD across multiple CPUs.
- Furthermore, we offload intense computational tasks to the GPU creating the first ever single- and multi-GPU implementation of RBF-FD.

As part of the research on RBF-FD within those three areas we additionally contributed the following:

- Application of RBF-FD to Centroidal Voronoi Tessellation grids.
- Approximate Nearest Neighbor methods for faster neighbor queries and improved system conditioning

Research Statement. What motivated this research?

The goal of this dissertation is to present a unified approach to parallel solutions of Partial Differential Equations (PDEs) with a method called Radial Basis Function-generated Finite Differences (RBF-FD).

fix: Many scientific problems of importance can be expressed as a collection of PDEs. Solutions to these problems provide answers to many simple questions such as the current temperature of a material, or perhaps the current position of a moving object. Complex and coupled PDEs can simulate the growth of zebra stripes or cheetah spots [?] or even model the flow of fluids. **Need refs for examples**

In order to solve these problems, computational numerical methods are employed on a discretized version of the domain. Traditionally, three (**and how would I classify FV? PartOfUnity?**) major categories exist for PDE solutions: finite difference (FD), finite element (FEM) and spectral element (SEM) [?]. Interestingly, all three of these methods rely on an underlying mesh, making them *meshed methods*. While each has had a turn in the spotlight,

more recently a new category, or rather a generalization on all three previous categories, has emerged: *meshfree methods*.

The first task in traditional meshed methods is to generate an underlying grid/mesh. Node placement can be done in a variety of ways including uniform, non-uniform and random (monte carlo) sampling, or through iterative methods like Lloyd's algorithm that generate a regularized sampling of the domain (see e.g., [?]). meshed methods have constraints on edge length and angle. Delaunay answers this, but is costly to compute Mesh2d, Triangle, DIstmesh In addition to choosing nodes, meshed methods require connectivity/adjacency lists to form stencils (FD) or elements (FEM, SEM)—this implies an added challenge to cover the domain closure with a chosen element type. While these tasks may be straightforward in one- or two-dimensions, the extension into higher dimensions becomes increasingly more cumbersome [?].

Complex geometries, irregular boundaries and mesh refinement also pose a problem for meshed methods. As the complexity of the geometry/boundaries increases, so too should the resolution of the approximating mesh in order to accurately reconstruct the detail present. A naïve approach to refinement increases the density of nodes uniformly across the domain, adding much more computation and memory storage than necessary for activity that is localized to sub-regions of the domain. Multiresolution methods attempt to compromise between accurate approximation of the domain and reduced resolution by one of two approaches: a) *multilevel methods* that decompose the model into a hierarchy with several levels of mesh detail, then only use a level when it is required to capture phenomena; and b) *adaptive irregular sampling* which has one level of detail, but non-uniform nodal density concentrated in areas of high activity [?]. Such techniques require robust methods and complex code capable of either coarsening/smoothing the approximate solutions to new level, or handling non-uniform node placement, element size etc.

Ideally, we seek a method defined on arbitrary geometries, that behaves regularly in any dimension, and avoids the cost of mesh generation. The ability to locally refine areas of interest in a practical fashion is also desirable. Fortunately, meshfree methods provide all of these properties: based wholly on a set of independent points in n -dimensional space, there is minimal cost for mesh generation, and refinement is as simple as adding new points where they are needed.

Since their adoption by the mathematics community in the 1980s ([?]), a plethora of meshfree methods have arisen for the solution of PDEs. For example, smoothed particle hydrodynamics, partition of unity method, element-free Galerkin method and others have been considered for fluid flow problems [?]. For a recent survey of methods see [?].

A subset of meshfree methods of particular interest to the community today revolves around Radial Basis Functions (RBFs). RBFs are a class of radially symmetric functions (i.e., symmetric about a point, x_j , called the *center*) of the form:

$$\phi_j(\mathbf{x}) = \phi(r(\mathbf{x})) \quad (1.1)$$

where the value of the univariate function ϕ is a function of the Euclidean distance from the center point \mathbf{x}_j given by $r(\mathbf{x}) = \|\mathbf{x} - \mathbf{x}_j\|_2 = \sqrt{(x - x_j)^2 + (y - y_j)^2 + (z - z_j)^2}$. Examples of commonly used RBFs are available in Table 4.1 with their corresponding plots in Figure 1.1. RBF methods are based on a superposition of translates of these radially symmetric functions, providing a linearly independent but non-orthogonal basis used to

interpolate between nodes in n -dimensional space. An example of RBF interpolation in 2D using 15 Gaussians is shown in Figure 2.1, where $\phi_j(r(\mathbf{x}))$ is an RBF centered at $\{\mathbf{x}_j\}_{j=1}^n$.

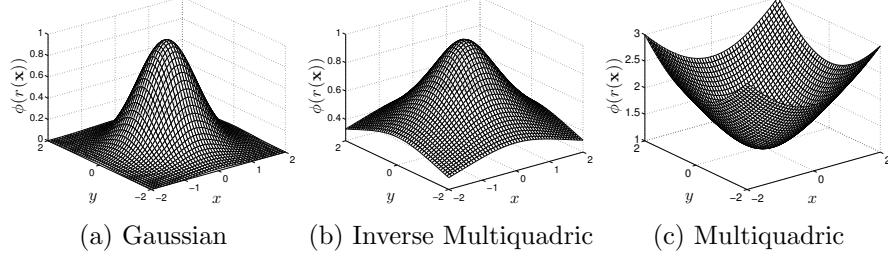


Figure 1.1: Commonly used RBFs.

With a history extending back four decades for RBF interpolation schemes [?], and two decades for RBFs applied to solving PDEs [?], many avenues of research remain untouched within their realm. Being a meshless method, RBF methods excel at solving problems that require geometric flexibility with scattered node layouts in n -dimensional space. They naturally extend into higher dimensions without significant increase in programming complexity [?, ?]. In addition to competitive accuracy and convergence compared with other state-of-the-art methods [?, ?, ?, ?, ?], they also boast stability for large time steps.

Like most numerical methods, RBFs come with certain limitations. For example, RBF interpolation is—in general—not a well-posed problem, so it requires careful choice of positive definite or conditionally positive definite basis functions [?, ?]. The example 2D RBFs presented in Figure 1.1 are infinitely smooth and satisfy the (conditional) positive definite requirements.

Infinitely smooth RBFs depend on a shape or support parameter ϵ that controls the width of the function. The functional form of the shape function becomes $\phi(\epsilon r)$. Decreasing ϵ increases the support of the RBF and in most cases, the accuracy of the interpolation, but worsens the conditioning of the RBF interpolation problem [?]. The **conditioning** of the system also dramatically **decreases** as the number of nodes in the problem increases. Fortunately, recent algorithms such as Contour-Padé [?] and RBF-QR [?, ?] allow for numerically stable computation of interpolants in the nearly flat RBF regime (i.e., $\epsilon \rightarrow 0$) where high accuracy has been observed [?, ?].

Historically, the most common way to leverage RBFs for PDE solutions is in a global interpolation sense. That is, the value of a function value or any of its derivatives at a node location is a linear combination of all the function values over the *entire* domain, just as in a pseudospectral method. If using infinitely smooth RBFs, this leads to spectral (exponential) convergence of the RBF interpolant for smooth data [?]. As discussed in [?], global RBF methods require $O(N^3)$ floating point operations (FLOPs) in pre-processing, where N is the total number of nodes, to assemble and solve a dense linear system for differentiation coefficients. The coefficients in turn are assembled into a dense Differentiation Matrix (DM) that is applied via matrix-vector multiply to compute derivatives at all N nodes for a cost of $O(N^2)$ operations. **assumes explicit scheme**

Alternatively, one can use RBF-generated finite differences (RBF-FD) to introduce sparse DMs (Note: for pure interpolation, compactly supported RBFs can also introduce

sparse matrices [?]). RBF-FD was first introduced by Tolstykh in 2000 [?], but it was the simultaneous, yet independent, efforts in [?], [?], [?] and [?] that gave the method its real start. RBF-FD share advantages with global RBF methods, like the ability to function without an underlying mesh, easily extend to higher dimensions and afford large time steps; however spectral accuracy is lost. Some of the advantages of RBF-FD include high computational speed together with high-order accuracy (6th to 10th order accuracy is common) and the opportunity for parallelization.

The RBF-FD method is similar in concept to classical finite-differences (FD): both methods approximate derivatives as a weighted sum of values at nodes within a nearby neighborhood. The two methods differ in that the underlying differentiation weights are exact for RBFs rather than polynomials.

As in FD, increasing the stencil size n increases the accuracy of the approximation. Given N total nodes in the domain (such as on the surface of a sphere), N linear systems, each of size $n \times n$, are solved to calculate the differentiation weights. Since $n \ll N$, the RBF-FD preprocessing complexity is dominated by $O(N)$ —much lower than for the global RBF method of $O(N^3)$ —with derivative evaluations on the order of $O(nN) \implies O(N)$ FLOPs.

CHAPTER 2

RELATED WORK

2.1 GPUS for explicit and implicit methods

2.2 GPUS/RBF

[?, ?]. Our paper [?] introduced the first implementation of RBF-FD to span multiple CPUs and multiple GPUs.

2.3 RBF/Finite-Difference

2.3.1 Global RBF Method

2.3.2 RBF-FD

RBF-FD have been successfully employed for a variety of problems including Hamilton-Jacobi equations [?], convection-diffusion problems [?, ?], incompressible Navier-Stokes equations [?, ?], transport on the sphere [?], and the shallow water equations [?].

2.4 Preconditioners

Many preconditioners are based on physical properties of the domain, PDE and the spectral properties of the numerical method.

2.5 Sparse matrix libraries

2.6 Various GPU-based libraries

2.6.1 CUSP

GMRES in CUSP is based on the Givens rotations for the Arnoldi process.

2.6.2 ViennaCL

GMRES in ViennaCL is based on Householder reflections for the Arnoldi process. We opted to leverage the ViennaCL package in most of our implementations.

$$\phi_j(\epsilon \|\mathbf{x} - \mathbf{x}_j\|) = e^{-(\epsilon \|\mathbf{x} - \mathbf{x}_j\|)^2}, (\epsilon = 2) \quad \hat{f}_N = \sum_{j=1}^N w_j \phi_j(\epsilon \|\mathbf{x} - \mathbf{x}_j\|)$$

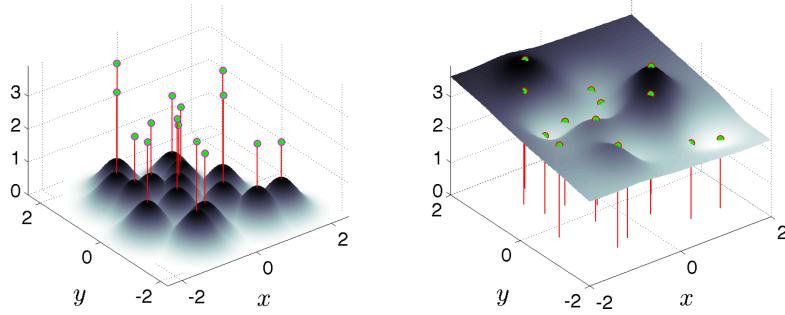


Figure 2.1: RBF interpolation using 15 translates of the Gaussian RBF with $\epsilon = 2$. One RBF is centered at each node in the domain. Linear combinations of these produce an interpolant over the domain passing through known function values.

2.7 Parallel implementations of RBF

As N grows larger, it behooves us to work on parallel architectures, be it CPUs or GPUs. With regard to the latter, there is some research on leveraging RBFs on GPUs in the fields of visualization [?, ?], surface reconstruction [?, ?], and neural networks [?]. However, research on the parallelization of RBF algorithms to solve PDEs on multiple CPU/GPU architectures is essentially non-existent. We have found three studies that have addressed this topic, none of which implement RBF-FD but rather take the avenue of domain decomposition for global RBFs (similar to a spectral element approach). In [?], Divo and Kassab introduce subdomains with artificial boundaries that are processed independently. Their implementation was designed for a 36 node cluster, but benchmarks and scalability tests are not provided. Kosec and Šarler [?] parallelize coupled heat transfer and fluid flow models using OpenMP on a single workstation with one dual-core processor. They achieved a speedup factor of 1.85x over serial execution, although there were no results from scaling tests. Yokota, Barba and Knepley [?] apply a restrictive additive Schwarz domain decomposition to parallelize global RBF interpolation of more than 50 million nodes on 1024 CPU processors. Only Schmidt et al. [?] have accelerated a global RBF method for PDEs on the GPU. Their MATLAB implementation applies global RBFs to solve the linearized shallow water equations utilizing the AccelerEyes Jacket [?] library to target a single GPU.

2.8 Contributions of this Work

Within this dissertation, we have developed the first implementation of RBF-FD to span multiple CPUs. Each CPU has a corresponding GPU attached to it in a one-to-one correspondence. We thus also introduced the first known implementation of accelerated RBF-FD on the GPU.

continue with outline of chapters We present our explicit and implicit solutions to PDEs with our multi-GPU RBF-FD implementation.

CHAPTER 3

Fragments (Integrate Above)

with the goal of solving coupled fluid flows modelling the physics (?) in the interior of the earth. We require both advective terms and diffusive terms. The components of a good pressure fluid solver are explicit and implicit differentiation. Whether we solve a steady-state PDE implicitly or a hyperbolic PDE with an implicit time-scheme, we require an *implicit solver*. An implicit solver is nothing but a method of assembling a differentiation matrix, forcing terms on the RHS and then solving the linear system. A direct LU factorization would suffice if our system is dense, but with RBF-FD the system is sparse. Sparse systems can be efficiently solved under the right conditions through sparse iterative solvers. :

Sufficient understanding of RBF interpolation led to the development of the first global RBF method for PDEs in [?]. Most popular among global methods is collocation, wherein RBF interpolation approximates the PDE solution by solving a large, dense linear system. In some cases RBF collocation demonstrates higher accuracy for the same number of nodes when compared to other state-of-the-art pseudospectral methods (e.g., [?] [?] [?]). In [?], spectral accuracy was demonstrated for hyperbolic PDEs even with local refinement of nodes in time.

Unfortunately—ill-conditioning aside—collocation methods are prohibitively expensive to solve when scaled to a large number of nodes. Assuming the collocation matrix does not change in time, global methods, with their dense systems, scale at $O(N^3)$ operations for initial preconditioning/preprocessing followed by $O(N^2)$ operations every time-step. This complexity is consistent with any collocation scheme. However, by introducing sparsity into the system (e.g., using compactly supported RBFs), the complexity is somewhat reduced.

General Purpose GPU (GPGPU) computing is one of today’s hottest trends within scientific computing. The release of NVidia’s CUDA at the end of 2006 marked both a redesign of GPU architecture, plus the addition of a new software layer that finally made GPGPU accessible to the general public. The CUDA API includes routines for memory control, interoperability with graphics contexts (i.e., OpenGL programs), and provides GPU implementation subsets of BLAS and FFTW libraries [?]. After the undeniable success of CUDA for C, new projects emerged to encourage GPU programming in languages like FORTRAN (see e.g., HMPP [?] and Portland Group Inc.’s CUDA-FORTRAN [?]).

In early 2009, the Khronos Group—the group responsible for maintaining OpenGL—announced a new specification for a general parallel programming language referred to as the Open Compute Language (OpenCL) [?]. Similar in design to the CUDA language—in

many ways it is a simple refactoring of the predecessor—the goal of OpenCL is to provide a mid-to-low level API and language to control any multi- or many-core processor in a uniform fashion. Today, OpenCL drivers exist for a variety of hardware including NVidia GPUs, AMD/ATI CPUs and GPUs, and Intel CPUs.

This *functional portability* is the cornerstone of the OpenCL language. However, functional portability does not imply performance portability. That is, OpenCL allows developers to write kernels capable of running on all types of target hardware, but optimizing kernels for one type of target (e.g., GPU) does not guarantee the kernel will run efficiently on another target (e.g., CPU). With CPUs tending toward many cores, and the once special purpose, many-core GPUs offering general purpose functionality, it is easy to see that soon the CPU and GPU will meet somewhere in the middle as general purpose many-core architectures. Already, ATI has introduced the Fusion APU (Accelerated Processing Unit) which couples an AMD CPU and ATI GPU within a single die. OpenCL is an attempt to standardize programming ahead of this intersection.

Petascale computing centers around the world are leveraging GPU accelerators to achieve peak performance. In fact, many of today's high performance computing installations boast significantly more GPU accelerators than CPU counterparts. The Keeneland project is one such example, currently with 240 CPUs accompanied by 360 NVidia Fermi class GPUs with at least double that number expected by the end of 2012 [?].

Such throughput oriented architectures require developers to decompose problems into thousands of independent parallel tasks in order to fully harness the capabilities of the hardware. To this end, a plethora of research has been dedicated to researching algorithms in all fields of computational science. Of interest to us are methods for atmospheric- and geo-sciences.

3.1 Fragments

- what is new in the thesis (summarize: 1-2 pages)
- get all your figures
- 1/2 page description per figures
 - All figure captions (self-contained). Don't skimp on words.
- Previous work
 - GPUS/implicit methods
 - GPUS/RBF
 - RBF/Finite-Difference
 - Preconditionners (info on that)
 - Sparse matrix libraries
 - Various GPU-based libraries
 - Parallel implementations of RBF (there are any?)

Benchmarking....

- Timing (serial, parallel)
- Timing with special node reorderings. Explain logic

Test cases ...

Get all your references in bibtex. Send them to me.

Need detailed table of content.

As will be demonstrated in Chapter ??, most of the literature surrounding the solution of PDEs with RBFs is based on collocation. Collocation finds an interpolant that passes through a set of *collocation points* (commonly chosen to coincide with RBF centers) satisfying the differential equations with zero residual. Collocation, then, is a global interpolation problem. Alternative methods exist based on local collocation formulations (see [?, ?, ?, ?, ?]). Recently, a new approach using FD-like stencils to approximate differential operators was proposed (see e.g., [?, ?, ?, ?]). The so-called RBF-FD method uses RBFs to interpolate initial conditions, but not the differential equations—generalized FD stencils approximate differential operators.

Even today, RBFs are still up-and-coming in the scientific world with many avenues of research left to consider. Global formulations are understood to have spectral convergence properties, high accuracy and other benefits like adaptivity and ease of implementation over meshed methods [?]. However, little is known about the behavior of local and RBF-FD methods. Open questions include (but are in no way limited to): a) ideal node placement to eliminate singularities; b) data-structures for stencil storage and evaluation; c) problem sizes larger than a few thousand nodes; and d) parallel implementations across new heterogeneous multi- and many-core architectures. In response to this, our group, in collaboration with researchers assembled from a national lab and four universities (see Chapter ??), has been granted funds by the National Science Foundation to collaboratively:

“Bring RBFs to the forefront of multi-scale geophysical modeling by developing fast, efficient, and parallelizable RBF algorithms in arbitrary geometries, with performance enhanced by hardware accelerators, such as graphic processing units (GPUs).” [?]

In the last few years, GPUs transitioned from hardware dedicated to the embarrassingly parallel tasks involved in graphics rendering (e.g., rasterization) into multi-core co-processors for high performance scientific computing. Thanks to the highly profitable and always demanding gaming industry, what began as a static rendering pipeline, was molded to allow fully dynamic execution with a SIMD-like programming model (Single Instruction Multiple Threads or SIMT). Changes in hardware were followed closely by evolving programming languages. Today, GPUs can be manipulated via high level languages similar to C/C++ and require no knowledge of computer graphics. In Chapters ?? and ?? we will discuss how GPU hardware and languages evolved to exceptionally higher compute capability than traditional CPUs, and became a popular platform for high performance computing.

True to our purpose statement above, the goal of this project is to integrate RBF methods for PDEs, Geophysics and large scale GPU computing. We begin in Chapter ?? with a discussion of related work on RBF-PDE methods, their applications, and related work on

GPUs. In Chapter ?? a formal introduction to RBFs for the solution of PDEs is provided. Chapter ?? considers various languages available for GPU computing and their appropriateness for our task. This is followed by a discussion in Chapter ?? of GPU hardware and the large scale clusters with integrated GPUs which will be used for heterogeneous multi-core computing. Finally, in Chapter ?? we present our proposal for research into efficient PDE solutions on multi-node, multi-GPU compute clusters using radial basis functions for Tsunami simulation.

This chapter outlines the process of solving PDEs using RBFs and a few methods developed since 1990. We start with a description of the general approximation problem and provide background on RBF scattered data interpolation that will be required for the remainder of the chapter. We categorize existing methods for solving PDEs with RBFs as either global or local. Global methods use collocation and invert a single large linear system to find the interpolant that satisfies the differential equations at RBF centers. Local methods limit the influence of basis functions and seek an interpolant at each RBF center defined in terms of neighboring basis functions (local collocation) or nodal values (RBF-FD).

We present three collocation methods: Kansa's method, Fasshauer's method and Direct collocation, and discuss extensions that lead to local interpolation matrices instead of a single global interpolation matrix. Finally, we discuss RBF-FD, the most recent method for solving PDEs.

3.2 PDE as an Interpolation Problem

Following [?], consider a PDE expressed in terms of a (linear) differential operator, \mathcal{L} :

$$\mathcal{L}u = f \quad \text{on } \Omega \tag{3.1}$$

$$u = g \quad \text{on } \Gamma \tag{3.2}$$

where Ω is the interior of the physical domain, Γ is the boundary of Ω and f, g are known explicitly. In the case of a non-linear differential operator, a Newton's iteration, or some other method, can be used to linearize the problem (see e.g., [?]); of course, this increases the complexity of a single time step. Then, the unknown solution, u , which produces the observations on the right hand side can be approximated by an interpolant function u_Φ expressed as a linear combination of radial basis functions, $\{\Phi_j(x) = \Phi(\|x - x_j\|)\}_{j=1}^N$, and polynomial functions $\{P_l(x)\}_{l=1}^M$:

$$u_\Phi(x) = \sum_{j=1}^N c_j \Phi_j(x) + \sum_{l=1}^M d_l P_l(x), \quad P_l(x) \in \Pi_p^D \tag{3.3}$$

where $\Phi_j(x) = \|x - x_j\|$ ($\|\cdot\|$ is standard Euclidean distance). The second sum represents a linear combination of polynomials that enforces zero approximation error when $u(x)$ is a polynomial of degree than or equal to p . The variable D is the problem dimension (i.e., $u_\Phi(x) \in \mathbb{R}^D$). To eliminate degrees of freedom for well-posedness, p should be greater than or equal to the order of the chosen RBF (see Table 4.1) [?]. Note that Equation 4.4 is evaluated at $x = \{x_i\}_{i=1}^n$ known data points through which the interpolant is required to pass with zero residual. We refer to the x_i 's (data points) points as *collocation points*, usually taken as the RBF centers, although this is not a requirement.

To clarify the role of the polynomial part in Equation 4.4, it is necessary to put aside the PDE for the moment and consider only the problem of *scattered data interpolation* with Radial Basis Functions. Borrowing notation from [?, ?], we seek an interpolant, \mathcal{P}_f of the

form

$$\begin{aligned}\mathcal{P}_f &= \sum_{j=1}^N c_j \Phi_j(x) = f(x) \\ &= \sum_{j=1}^N \Phi_j(x) c_j = f(x) \\ \vec{\mathcal{P}}_f &= \Phi^T(x) c = f(x)\end{aligned}$$

where \mathcal{P}_f is expressed as a scalar product between the unknown coefficient weights c_j and the radial basis functions $\Phi_j(x)$.

In many cases it is desirable to exactly reproduce functions of degree less than or equal to some order m . For RBF scattered data interpolation in one dimension, this can be achieved by adding a polynomial of order m with $M = \binom{m+1}{1}$ terms (e.g., x^0, x^1, \dots, x^m). In 2D, the terms would be: $1, x, y, xy, x^2y, xy^2, \dots, x^m y^{m-1}, x^{m-1} y^m, x^m y^m$. In \mathbb{R}^D , $M = \binom{m+D}{D}$ [?], giving

$$f(x) = \sum_{j=1}^N c_j \Phi_j(x) + \sum_{l=1}^M d_l P_l(x), \quad P_l(x) \in \Pi_m^D \quad (3.4)$$

$$f = \Phi^T c + P^T d \quad (3.5)$$

where the second summation (referred to as *interpolation conditions* [?]) ensures the minimum degree of the interpolant. Notice, however, that the interpolation conditions add M new degrees of freedom, so we must provide M *moment conditions* or constraints:

$$\sum_{j=1}^N c_j P_l(x_j) = 0, \quad l = 1, \dots, M$$

or

$$\mathbb{P}^T c = 0$$

to guarantee that the moments of the interpolant vanish for the coefficients obtained, where the matrix

$$\mathbb{P}^T = (P(x_1) \ P(x_2) \ \cdots \ P(x_N))$$

and

$$P(x_j) = (P_1(x_j) \ P_2(x_j) \ \cdots \ P_M(x_j))^T.$$

It is now possible again to write the interpolation as a linear system using Equations 4.6 and 4.2:

$$\vec{\mathcal{P}}_f = \begin{bmatrix} \Phi & \mathbb{P} \\ \mathbb{P}^T & 0 \end{bmatrix} \begin{pmatrix} \vec{c} \\ \vec{d} \end{pmatrix} = \begin{pmatrix} \vec{f} \\ 0 \end{pmatrix} \quad (3.6)$$

This system then produces an interpolant capable of exactly approximating data from polynomials of degree less than or equal to m [?].

Returning to the PDE, we postpone the issue of achieving polynomial precision in approximation. However, it has been shown (see [?, ?]) that some choices of RBFs (e.g. multiquadratics and thin-plate splines [?]) are not positive definite and therefore there is no guarantee that the approximation is well-posed. A sufficient condition for well-posedness is that the matrix is *conditionally positive definite*. In [?], Fasshauer demonstrates that conditional positive definiteness of order m is guaranteed when Equation 4.4 includes the full polynomial part (i.e., interpolation conditions) to match the order of the RBF.

Now, since $u_\Phi(x)$ from Equation 4.4 cannot (in general) satisfy the PDE everywhere, we enforce the PDE at a set of collocation points, which are distributed over both the interior and the boundary. Again, these points do not necessarily coincide with the RBF centers, but it is convenient for this to be true in practice.

3.3 Reconstructing Solutions for PDEs

In the next few sections, we will generate collocation equations based on this general form:

$$\mathcal{L}u_\Phi(x) = f(x) \quad \text{on } \Omega \tag{3.7}$$

$$\mathcal{B}u_\Phi(x) = g(x) \quad \text{on } \Gamma \tag{3.8}$$

where the methods presented below will apply the differential operators, \mathcal{L} and \mathcal{B} , to different choices of u_Φ and different sets of collocation points. In many applications \mathcal{L} is chosen as a differential operator (e.g., ∇ , ∇^2) and $\mathcal{B} = I$ (i.e. identity operator for Dirichlet boundary conditions) for PDEs. There are also applications where \mathcal{L} is a convolution operator (see e.g., [?, ?]) capable of smoothing/de-noising a surface reconstructed from point clouds.

For all the methods presented here a linear system is generated

$$\begin{aligned} A \begin{pmatrix} c \\ d \end{pmatrix} &= \begin{pmatrix} f \\ 0 \end{pmatrix} \\ \begin{pmatrix} c \\ d \end{pmatrix} &= A^{-1} \begin{pmatrix} f \\ 0 \end{pmatrix} \end{aligned} \tag{3.9}$$

where matrix A depends on the choice of collocation method. Once the linear system is solved, the value, $u(x_i)$, at an unknown point, x_i , can be reconstructed using the coefficients with the following inner product:

$$u(x_i) = [\Phi^T(x_i) \quad P^T(x_i)] \begin{pmatrix} c \\ d \end{pmatrix} \tag{3.10}$$

Likewise, to find differential quantities we have:

$$\mathcal{L}u(x_i) = [\mathcal{L}(\Phi^T)(x_i) \quad \mathcal{L}(P^T)(x_i)] \begin{pmatrix} c \\ d \end{pmatrix} = A_{\mathcal{L}} \begin{pmatrix} c \\ d \end{pmatrix} \tag{3.11}$$

Here we substitute $A_{\mathcal{L}}$ for the right hand side row vector. Also, Equation 4.10 can be substituted into Equation 4.12 to get

$$\mathcal{L}u(x_i) = A_{\mathcal{L}} A^{-1} \begin{pmatrix} f \\ 0 \end{pmatrix} \tag{3.12}$$

where the vector-matrix inner product $(A_{\mathcal{L}} A^{-1})$ is a row-vector. Since the coefficient vectors c and d are the same for all x_i , we can group the evaluation of $\mathcal{L}u(x_i)$ for $i = 1, \dots, n$ as a matrix-vector multiplication where the matrix rows correspond to $(A_{\mathcal{L}} A^{-1})$ for each x_i .

3.4 Kansa's Method

The first method of collocation, *Kansa's method* [?, ?], collocates the solution through known values on the boundary, while constraining the interpolant to satisfy the PDE operator on the interior. This is equivalent to choosing u_{Φ} according to Equation 4.4. The resulting system is given by [?]; assuming that \mathcal{L} is a linear operator,

$$\mathcal{L}u_{\Phi}(x_i) = \sum_{j=1}^N c_j \mathcal{L}\Phi_j(x_i) + \sum_{l=1}^M d_l \mathcal{L}P_l(x_i) = f(x_i) \quad i = 1, \dots, n_I \quad (3.13)$$

$$\mathcal{B}u_{\Phi}(x_i) = \sum_{j=1}^N c_j \mathcal{B}\Phi_j(x_i) + \sum_{l=1}^M d_l \mathcal{B}P_l(x_i) = g(x_i) \quad i = n_I + 1, \dots, n \quad (3.14)$$

$$\sum_{j=1}^N c_j P_l(x_j) = 0 \quad l = 1, \dots, M \quad (3.15)$$

where n_I are the number of interior collocation points, with the number of boundary collocation points equal to $n - n_I$. First, observe that the differential operators are applied directly to the RBFs inside summations, rather than first solving the scattered data interpolation problem and then applying the operator to the interpolant. Second, since the basis functions are known analytically, it is possible (although sometimes painful) to derive $\mathcal{L}\Phi$ (refer to [?] for RBF derivative tables); the same is true for the polynomials P_l .

We can now reformulate Kansa's method as the linear system:

$$\begin{bmatrix} \Phi_{\mathcal{L}} & P_{\mathcal{L}} \\ \Phi_{\mathcal{B}} & P_{\mathcal{B}} \\ P^T & 0 \end{bmatrix} \begin{pmatrix} c \\ d \end{pmatrix} = \begin{pmatrix} f \\ g \\ 0 \end{pmatrix} \quad (3.16)$$

where $\Phi_{\mathcal{L}} = \mathcal{L}\Phi$, $P_{\mathcal{L}} = \mathcal{L}P$ are the interior components (Equation 4.17), $\Phi_{\mathcal{B}}$ and $P_{\mathcal{B}}$ are the boundary components (Equation 4.18), and $P^T = [P_{\mathcal{L}}^T \ P_{\mathcal{B}}^T]$ are moment constraints for both interior and boundary polynomial parts (Equation 4.19). From Equation 4.20 it should be clear why Kansa's method is also known as the *Unsymmetric* collocation method.

Recall from Chapter ?? that the matrix in Equation 4.20 has no guarantee of non-singularity [?]; however, singularities are rare in practice [?].

3.5 Fasshauer's Method

Fasshauer's method [?] addresses the problem of singularity in Kansa's method by assuming the interpolation to be Hermite. That is, it requires higher differentiability of the

basis functions (they must be at least C^k -continuous if \mathcal{L} is of order k). Leveraging this assumption, Fasshauer's method chooses:

$$u_{\Phi}(x_i) = \sum_{j=1}^{N_I} c_j \mathcal{L}\Phi_j(x_i) + \sum_{j=N_I+1}^N c_j \mathcal{B}\Phi_j(x_i) + \sum_{l=1}^M d_l P_l(x_i) \quad (3.17)$$

as the interpolant passing through collocation points. Note N_I is used here to specify the number of RBF centers in the interior of Ω . Here the interpolant is similar to Equation 4.4, but a change of basis functions is used for the expansion: $\mathcal{L}\Phi_j(x)$ on the interior and $\mathcal{B}\Phi_j(x)$ on the boundary.

Collocating (i.e., substituting Equation 4.21 into Equations 4.17-4.19) we get:

$$\sum_{j=1}^{N_I} c_j \mathcal{L}^2 \Phi_j(x_i) + \sum_{j=N_I+1}^N c_j \mathcal{L}\mathcal{B}\Phi_j(x_i) + \sum_{l=1}^M d_l \mathcal{L}P_l(x_i) = f(x_i) \quad i = 1, \dots, n_I \quad (3.18)$$

$$\sum_{j=1}^{N_I} c_j \mathcal{B}\mathcal{L}\Phi_j(x_i) + \sum_{j=N_I+1}^N c_j \mathcal{B}^2 \Phi_j(x_i) + \sum_{l=1}^M d_l \mathcal{B}P_l(x_i) = g(x_i) \quad i = n_I + 1, \dots, M \quad (3.19)$$

$$\sum_{j=1}^{N_I} c_j \mathcal{L}P_l(x_j) + \sum_{j=N_I+1}^N c_j \mathcal{B}P_l(x_j) = 0 \quad l = 1, \dots, M \quad (3.20)$$

which is reformatted as the linear system:

$$\begin{bmatrix} \Phi_{\mathcal{L}\mathcal{L}} & \Phi_{\mathcal{L}\mathcal{B}} & P_{\mathcal{L}} \\ \Phi_{\mathcal{B}\mathcal{L}} & \Phi_{\mathcal{B}\mathcal{B}} & P_{\mathcal{B}} \\ P_{\mathcal{L}}^T & P_{\mathcal{B}}^T & 0 \end{bmatrix} \begin{pmatrix} c \\ d \end{pmatrix} = \begin{pmatrix} f \\ g \\ 0 \end{pmatrix} \quad (3.21)$$

Note that $\Phi_{\mathcal{L}\mathcal{L}}$ represents the first summation in Equation 4.22. The linear system generated by Fasshauer's method reveals an interesting structure: namely, the subscripts \mathcal{L} and \mathcal{B} show blocks of influence in the matrix. For example, the interior RBF centers influence collocation on the interior collocation points ($\Phi_{\mathcal{L}\mathcal{L}}$), boundary centers influence collocation on the interior ($\Phi_{\mathcal{L}\mathcal{B}}$), interior centers influence collocation on the boundary ($\Phi_{\mathcal{B}\mathcal{L}}$), and so forth. In the case where the collocation points and RBF centers do not coincide, the subscripts would also indicate which set of points the operators are applied to [?].

The symmetry of Fasshauer's (*symmetric collocation*) method is apparent in Equation 4.25. Likewise, it is clear that the symmetric method requires more storage and computation to solve compared to Kansa's method. However, based on the assumption that collocation points coincide with RBF centers, the symmetry reduces storage requirements by half.

3.6 Direct Collocation

In *Direct collocation* (see [?, ?], the interpolant is chosen as Equation 4.4 (the same as Kansa's method). However, the Direct method collocates both the interior and boundary

operators at the boundary points:

$$\sum_{j=1}^N c_j \mathcal{L} \Phi_j(x_i) + \sum_{l=1}^M d_l \mathcal{L} P_l(x_i) = f(x_i) \quad i = 1, \dots, n \quad (3.22)$$

$$\sum_{j=1}^N c_j \mathcal{B} \Phi_j(x_i) + \sum_{l=1}^M d_l \mathcal{B} P_l(x_i) = g(x_i) \quad i = 1, \dots, n_B = n - n_I \quad (3.23)$$

$$\sum_{j=1}^N c_j P_l(x_j) = 0 \quad l = 1, \dots, M \quad (3.24)$$

Reformulating as a linear system we get:

$$\begin{bmatrix} \Phi_{\mathcal{L}} & P_{\mathcal{L}} \\ \Phi_{\mathcal{B}} & P_{\mathcal{B}} \\ P^T & 0 \end{bmatrix} \begin{pmatrix} c \\ d \end{pmatrix} = \begin{pmatrix} f \\ g \\ 0 \end{pmatrix} \quad (3.25)$$

While the final system in Equation 4.29 is structured the same as Kansa's method (Equation 4.20), careful inspection of the index i in Equations 4.17 and 4.26 reveals that Direct collocation produces a larger system.

3.7 Local Methods

In all three cases above, the collocation required interpolation with globally supported RBFs. However, the latest trend is to use RBFs defined with some cut-off radius to enforce compact support. In some cases, authors have used the compact support to produce a single (large) sparse system for interpolation (see e.g., [?, ?, ?, ?, ?]). Other approaches use compact support to produce local linear systems defined at each collocation point. Examples of this include [?, ?] for Kansa's method, [?, ?, ?] for Fasshauer's method. To our knowledge no one has considered local Direct collocation. Also, instead of specifying a cut-off radius, some authors specify the exact stencil size (i.e., number of neighboring points to include); see e.g., [?, ?].

After observing the general structure of the symmetric and unsymmetric collocation methods above, it is necessary only to present the symmetric (i.e. Fasshauer's) local method and note that in the unsymmetric case certain blocks will be zero allowing the system to shrink.

The formula for the interpolant local to the (k) -th collocation point (i.e., RBF center) is given by:

$$u_{\Phi}^{(k)}(x_i) = \sum_{j(k)=1}^{N_I} c_j^{(k)} \mathcal{L} \Phi_j(x_i) + \sum_{j(k)=N_I+1}^{N_S} c_j^{(k)} \mathcal{B} \Phi_j(x_i) + \sum_{l=1}^M d_l^{(k)} P_l(x_i) \quad (3.26)$$

where N_S represents the number of points that defines the local stencil; N is possibly a function of the cut-off radius in the RBF, N_I is the number of interior stencil points (those points of the stencil that lie in the interior of Ω). The index j is a function of the stencil center k allowing the system to include a local neighborhood of stencil points.

Collocating produces a linear system with similar structure to the global collocation problem, but the dimensions are much smaller:

$$\begin{bmatrix} \Phi_{\mathcal{L}\mathcal{L}} & \Phi_{\mathcal{L}\mathcal{B}} & P_{\mathcal{L}} \\ \Phi_{\mathcal{B}\mathcal{L}} & \Phi_{\mathcal{B}\mathcal{B}} & P_{\mathcal{B}} \\ P_{\mathcal{L}}^T & P_{\mathcal{B}}^T & 0 \end{bmatrix} \begin{pmatrix} c^{(k)} \\ d^{(k)} \end{pmatrix} = \begin{pmatrix} f \\ g \\ 0 \end{pmatrix} \quad (3.27)$$

Solving this system gives an interpolant locally defined around the stencil center. Note that approximating the PDE solution $u(x)$ requires finding the stencil center nearest x , then using the local interpolant for that stencil. Since interpolation is local (i.e., $c_j^{(k)}$'s are unique to each RBF center), reconstructing the derivatives with Equation 4.12 is limited to an inner product for each center rather than the matrix-vector grouping possible with global RBFs. Note that because the interpolants are local, there is no notion of global continuity/smoothness of the solution.

3.8 RBF-FD

While most of the literature surrounding RBFs for PDEs involves collocation, an alternative method does exist: RBF-FD. RBF-FD is a hybrid of RBF scattered data interpolation and Finite Difference stencils. The idea behind finite-difference stencils is to express various derivative operators as a linear combination of the unknown functional values in the neighborhood of the point where an approximation to the derivative operator is desired. Common approximations such as upwind differencing, center differencing, higher order approximations, and even spectral operators, are of this form. A common approach to building such discrete operators is to form a local interpolant in a neighborhood of the target point, and simply differentiate it analytically. This is the approach taken in RBF-FD, which allows for stencils with irregular placement and number of nodes, and assigns their weights based on an RBF [?]. For a detailed derivation of the method see [?, ?, ?, ?]. Such an approach leads to very simple implementations of time-advancement schemes, whether explicit or implicit. The solution at the new time step is simply some linear (if $\mathcal{L}()$ is linear), nonlinear otherwise, combination of the unknown functional values (if implicit scheme) or known functional value (if explicit scheme).

Key challenges lie in the choice of grid, the choice of stencil, whether or not to change the support as a function of the stencil, how to guaranty the stability of the differentiation operator after discretization, etc.

3.9 Time Stepping

Until now we have assumed that the differential operator \mathcal{L} had no time dependency. However, many important problems are a function of time:

$$\frac{\partial u(x; t)}{\partial t} = \mathcal{L}u(x; t) \quad (3.28)$$

$$\mathcal{B}(u(x; t)) = g(x) \quad (3.29)$$

with some initial condition $u(x; 0) = f(x)$. Both collocation methods and RBF-FD methods can be used to discretize the right hand side. In the case of collocation, the coefficients c are now functions of time. On the other hand, it is the unknown functional values that are considered to be the unknown, time-dependent variables when RBF-FD is used.

The general approach for collocation methods is to first use the initial and boundary conditions to get an initial set of coefficients $c(t), d(t)$, whereas before, (c, d) denotes the list of unknown coefficients of the RBF basis functions and the monomials. Prior to spatial discretization, Equation 4.33 is updated using any of the standard methods for time-stepping, such as Euler, Crank-Nicholson, or higher order Runge-Kutta methods to name a few. Here we demonstrate the approach with the Euler method (which is explicit) for simplicity. In practice, a higher order method would be chosen, perhaps an implicit method. The Euler method can be written as

$$u(x; t_{n+1}) = u(x; t_n) + \Delta t(\mathcal{L}u(x; t_n)) \quad (3.30)$$

Since $u(x; 0) = u(x; t_0)$ is known as a function of $(c, d) = A^{-1}u(x; t_n)$, it is straightforward to compute the coefficients, and from there, $\mathcal{L}(u(x; t_n))$.

Equation 4.34 is solved via a two step algorithm:

1. solve Equation 4.13 to approximate $\mathcal{L}u(x; t_n)$ for the current time-step
2. update $u(x; t_{n+1})$

Initially, $A_{\mathcal{L}}A^{-1}$ is computed once during the first time-step, and the first part of the algorithm is reduced to an inner product to get $\mathcal{L}u(x; t_n)$. Once $u(x; t_{n+1})$ is updated, one repeats the procedure with $u(x; t_{n+1})$ as a new initial condition. For collocation methods, one must recompute all the coefficients at each timestep (i.e. recompute A^{-1}), which is expensive. On the other hand, for a static grid, $A_{\mathcal{L}}A^{-1}$ can be precomputed and there is no longer any need to work in coefficient space. In this case, we have a type of RBF-FD method, which is solely expressed in terms of functional values. Of course, in a "pure" RBF-FD method, one has expressions for the various derivatives and operators as a linear combination of functional values (the derived stencil being an implicit function of the RBFs and their location).

There is a huge literature on time-stepping schemes for finite-difference, finite-volume, spectral methods, etc., and many of these methods can be adopted to the solve time-dependent problems on meshless grids. Although there are surely issues that are particular to RBFS, we do not discuss them here. Instead, we will include our experiences in the final thesis.

For initial value problems with moving nodes, the coefficients must be updated at each time-step whether working with collocation or RBF-FD. We do not consider moving node problems in this thesis.

Part I

Preliminaries

CHAPTER 4

RBF METHODS FOR PDES

The process of solving partial differential equations (PDEs) using radial basis functions (RBFs) dates back to 1990 [?, ?]. However, at the core of all RBF methods lies the fundamental problem of approximation/interpolation. Some methods (e.g., global- and compact-RBF methods) apply RBFs to approximate derivatives directly. Others (e.g., RBF-generated Finite Differences) leverage the basis functions indirectly by generating weights for finite-differencing stencils that are used in-turn to approximate derivatives. Regardless, to track the history of RBF methods, one must look back to 1971 and R.L. Hardy's seminal research on interpolation with multi-quadric basis functions [?].

This chapter is dedicated to summarizing the four-decade history of RBF methods leading up to the development of the RBF-generated Finite Differences (RBF-FD) method. Beginning with a description of the general approximation problem and RBF scattered-data interpolation, we then categorize existing methods for solving PDEs with RBFs as either global or local. Global methods use collocation and invert a single large linear system to find the interpolant that satisfies the differential equations at RBF centers. Local methods limit the influence of basis functions and seek an interpolant at each RBF center defined in terms of neighboring basis functions (local collocation) or nodal values (RBF-FD).

We present three collocation methods: Kansa's method, Fasshauer's method and Direct collocation, and discuss extensions that lead to local interpolation matrices instead of a single global interpolation matrix. Finally, we discuss the most recent methods for solving PDEs, emphasizing RBF-FD as the focus of this work.

4.1 Survey of Related Work

Author's Note: Should I crop the intro on what RBFs are and stash it in a previous chapter (e.g., intro)?

In Radial Basis Function methods, radially symmetric functions provide a non-orthogonal basis used to interpolate between nodes of a point cloud. RBFs are univariate and a function of distance from a center point defined in \mathbb{R}^N , so they easily extend into higher dimensions without significant change in programming complexity. RBFs have been shown in some cases to have exponential convergence for function approximation [?]. It is also possible to reformulate RBF methods as pseudospectral methods that have generated solutions to ill-posed problems for which Chebyshev-based and other pseudospectral methods fail [?].

Name	Abbrev.	Formula	Order (p)
Multiquadric	MQ	$\sqrt{1 + (\varepsilon r)^2}$	1
Inverse Multiquadric	IMQ	$\frac{1}{\sqrt{1+(\varepsilon r)^2}}$	0
Gaussian	GA	$e^{-(\varepsilon r)^2}$	0
Thin Plate Splines	TPS	$r^2 \ln r $	2
Wendland (C^2)	W2	$(1 - \varepsilon r)^4(4\varepsilon r + 1)$	0

Table 4.1: Examples of frequently used RBFs based on [?, ?]. ε is the support parameter. For details on orders refer to Chapter ?? and [?]. All RBFs have global support. For compact support, enforce a cut-off radius (see Equation 4.1).

However, as with all methods, RBFs come with certain limitations. For example, RBF interpolation is—in general—not a well-posed problem, so it requires careful choice of positive definite or conditionally positive definite basis functions (see [?, ?] for details). Furthermore, in 1995, Schaback [?] observed an *Uncertainty Relation*: none of the commonly used RBFs combine good approximation with numerically stable interpolation. In other words, increasing accuracy can result in worse conditioning, whereas improving conditioning reduces accuracy [?, ?].

[Author's Note: \(end intro in question\)](#)

RBF methods for interpolation first appeared in 1971 with Hardy’s seminal research on multiquadratics [?]. In his 1982 survey of scattered data interpolation methods [?], Franke rated multiquadratics first-in-class against 28 other methods (3 of which were RBFs) [?]. Many other RBFs, including those presented in Table 4.1 have been applied in literature, but for PDEs in particular, none have rivaled the attention received by multiquadratics.

By 1990, the understanding of the scientific community regarding RBFs was sufficiently developed for application to PDEs through collocation. In Table 4.3 we classify references according to their choice of collocation method and RBF interpolation type.

We seek the solution to a PDE with the form

$$(\mathcal{L}u)(x_i) = \sum_{j=1}^N c_j \Phi_j(x_i) = f(x_i)$$

where \mathcal{L} is, in general, a nonlinear differential operator acting on $u(x)$. The solution $u(x)$ is expressed as a linear combination of N basis functions $\Phi_j(x)$, not necessarily RBFs:

$$u(x) = \sum_{i=1}^N c_i \Phi_i(x)$$

In the above expression, $\vec{c} = \{c_j\}$ is the unknown coefficient vector. Once one assumes \mathcal{L} is a linear operator, one has several choices. One can collocate the differential equation

Interpolation Type	Dense/Sparse A	Dim(A) ($N_S \ll N$)	# of A^{-1}	RBF Support
Global	Dense	$N \times N$	1	Global
Compact	Sparse	$N \times N$	1	Compact
Local	Dense	$N_S \times N_S$	N	Global/Compact

Table 4.2: RBF interpolation types and properties, assuming a problem with N nodes.

equation, or one can express individual derivative operators as linear combinations of the unknowns u_j (leading to the RBF-FD methods). In all cases, one ends up with a linear system of equations to invert, with different degrees of sparsity, dependent on the chosen basis functions and how the various constraints are enforced. While we restrict the $\Phi_j(x)$ to RBFs or various operators applied to the RBFs, we note that spectral methods, finite-element or spectral-element methods can be formulated in a similar way with different choices of basis functions. Of course, u can be a vector of unknown variables (c then becomes a matrix). In what follows, we refer to the matrix to invert as A .

There are three main categories of RBF interpolation; we list them in Table 4.2. The first is *Global* in the case that a single, large ($N \times N$) and dense matrix corresponding to globally supported RBFs is inverted; second, *Compact* if compactly supported RBFs are used to produce a single, large, but sparse matrix; and third, *Local* if compactly supported RBFs are used to produce many small but dense matrices with one corresponding to each collocation point. In all three cases the matrices are symmetric and with the correct choice of RBF they are at least conditionally positive definite.

We note that three types of collocation occur throughout the RBF literature: Kansa's unsymmetric collocation method [?, ?], Fasshauer's symmetric collocation method [?], and the Direct collocation method [?]. While not a collocation method, RBF-FD represents the latest trend in solving PDEs with RBFs. A full explanation of RBF based collocation is deferred to Chapter ??.

4.1.1 Global RBF Methods

Kansa's method [?, ?] (a.k.a. unsymmetric collocation) was the first RBF method for PDEs, and is still the most frequently used collocation type. The idea behind Kansa's method is that an approximate solution to the PDE can be found by finding an interpolant which satisfies the differential operator with zero residual at a set of *collocation points* (these coincide with the RBF centers). To find the interpolant, the differential equation is formulated as a two block (unsymmetric) linear system with: 1) the approximation of values at boundary points with boundary data only, and 2) the approximation of interior points by directly applying the differential operator. Hon et al. [?] employed Kansa's method to solve shallow water equations for Typhoon simulation. In [?], Flyer and Wright employed Kansa's method for the solution of shallow water equations on a sphere. Their results show that RBFs allow for longer time steps with spectral accuracy. It was shown in

	RBF Interpolation Type		
Method	Global (Dense)	Compact (Sparse Global)	Local
Kansa's Method	[?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?]	[?, ?]	[?, ?, ?, ?]
Fasshauer's Method	[?, ?, ?]	[?]	[?, ?, ?]
Direct Collocation	[?, ?]		[?, ?, ?, ?, ?, ?, ?, ?]
RBF-FD	N/A	N/A	

Table 4.3: Classification of references based on choice of RBF interpolation types and method for solving PDEs. References may appear in multiple cells according to the breadth of their research.

[?, ?] that the unsymmetric linear system produced by Kansa's method does not guarantee non-singularity; although it is noted that in practice singularities are rare encounters [?].

The second alternative for RBF collocation, is based on Hermite scattered data interpolation (see [?]). The so-called *Fasshauer* or *Symmetric Collocation* method ([?]) performs a change of basis for the interpolant by directly applying the differential operator to the RBFs. It then collocates using the same approach as Kansa's method [?, ?]. The resulting block structure of the linear system is symmetric and guaranteed to be non-singular [?]. In comparison to Kansa's method, the disadvantages of Fasshauer's method include: a) requirement of higher order differentiability of the basis functions (to satisfy double application of the differential operator) and b) the linear system is larger and more complex to form [?]. As [?] points out, the possible existence of a singularity in Kansa's method is not enough to justify the added difficulties of using Fasshauer's method.

The last method, *Direct Collocation*, was introduced by Fedoseyev, Friedman and Kansa [?] and satisfies the differential operator on the interior and the boundary. Larsson and Fornberg [?] observe that this third method has a matrix structure similar to that found in Kansa's method; however, it is noted that the dimensions of the matrix blocks for each method differ. This is due to collocation constraints added for the differential operator applied to the boundary. Aside from the survey on RBF collocation presented by Larsson and Fornberg [?], no related work was found that applied, or investigated, this method further.

4.1.2 Compactly Support RBFs

Thus far, all cases of collocation and interpolation mentioned have assumed globally supported RBFs. While global RBFs are well-studied and have nice properties, a major limitation is the large, dense system that must be solved. One alternative to global support

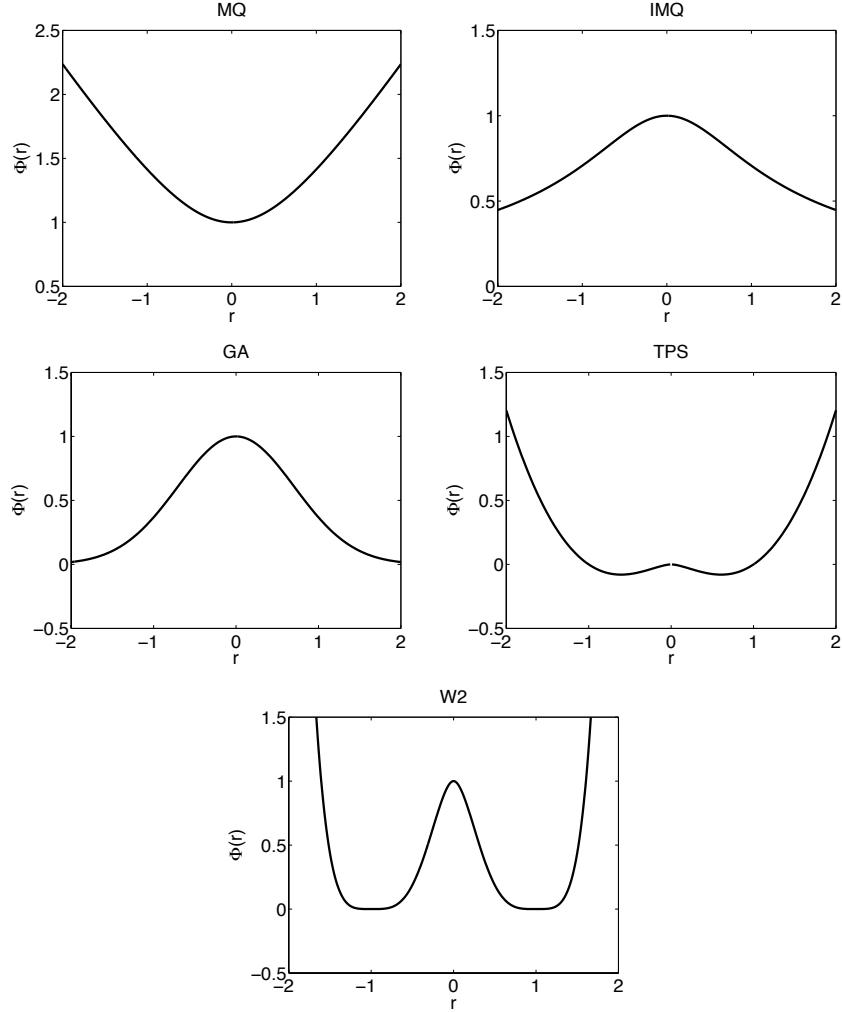


Figure 4.1: Example RBF shapes from Table 4.1 with parameter $\varepsilon = 1$.

is to use a set of compactly supported RBFs (CSRBFs) that are defined as:

$$\Phi(r) = \begin{cases} \varphi(r) & r \in [0, 1] \\ 0 & r > 1 \end{cases} \quad (4.1)$$

where a radius is defined past which the RBF has no influence on the interpolant. Note that the radius can be scaled to fit a desired support. Methods that leverage CSRBFs produce a global interpolation matrix that is *sparse* and therefore results in a system that is more efficiently assembled and solved [?]. While often referred to as CSRBF, a compact support method analogous to Kansa's method was also introduced under the name *radial point interpolation method (RPIM)* [?]. The CSRBF method analogue to Fasshauer's method is known as *radial point interpolation collocation method (RPICM)* [?]. A more thorough survey of CSRBF history can be found in [?]

CSRBFs have attracted a lot of attention in applications. For example, in the field of

dynamic surface and image deformation, compact support allows for local transformations which do not induce global deformation (see e.g., [?, ?, ?]).

We note that sparsity and spectral accuracy cannot be achieved simultaneously.

4.1.3 Local RBF Methods

Around 2005, Šarler and Vertnik [?, ?] demonstrated that if compactly supported RBFs are chosen, the traditional global collocation matrix from Kansa's method, can be avoided altogether in favor of small localized collocation matrices defined for each node. Local collocation still faces possible ill-conditioning and singularities like global collocation, but make it easier to distribute computation across parallel systems. Also, the smaller linear systems can be solved with more ease and less ill-conditioning. In [?], the authors consider 2D diffusion problems. Divo and Kassab [?] employ the method for Poisson-like PDEs including fluid flow and heat transfer. Kosec and Šarler [?] apply the same technique to solve coupled heat transfer and fluid flow problems.

In similar fashion, Stevens et al. [?] introduced a local version of Fasshauer's method called *local Hermitian interpolation*. The authors have applied their method to 3D soil problems based on transient Richards' equations [?, ?, ?].

4.1.4 RBF-FD

Another local alternative for solving PDEs with RBFs was presented by Wright and Fornberg [?, ?]. The method, called RBF-FD, combines scattered data interpolation to approximate function values and generalized finite difference stencils to approximate differential operators. This approach avoids collocation in the sense that the interpolant need not satisfy the differential operator at a set of collocation points.

According to Wright and Fornberg [?], RBF-FD (and *RBF-HFD* for the Hermite version) was independently proposed by many authors including Wright [?] in his dissertation, Shu et al. [?] and Cecil et al. [?]. A key difference between investigations was that Wright [?] was able to bypass ill-conditioning of RBF-FD and investigate its behavior in the limit as $\varepsilon \rightarrow 0$ by means of the Contour-Padé algorithm.

Shu et al. [?] compared the RBF-FD method to Least Squares FD (LSFD) in context of 2D incompressible viscous cavity flow, and found that under similar conditions, the RBF-FD method was more accurate than LSFD, but the solution required more iterations of an iterative solver. RBF-FD was applied to Poisson's equation in [?]. Chandhini and Sanyasiraju [?] studied it in context of 1D and 2D, linear and non-linear, convection-diffusion equations, demonstrating solutions that are non-oscillatory for high Reynolds number, with improved accuracy over classical FD. An application to Hamilton-Jacobi problems [?], and 2D linear and non-linear PDEs including Navier-Stokes equations [?] have all been considered.

4.1.5 Recent Advances in Conditioning

Recently, Fornberg and Wright [?] presented the *Contour-Padé algorithm*, which allows for numerically stable computation of highly accurate interpolants for (very small) cases typically associated with ill-conditioning induced by nearly flat RBFs (i.e., $\varepsilon \rightarrow 0$). Larsson and Fornberg [?] applied the algorithm to all three methods of collocation (Kansa's,

Fasshauer's and Direct Collocation) with considerable gain in accuracy over solutions from classical second-order FD and a pseudospectral method. Note that currently, the Contour-Padé algorithm was only studied for global RBF interpolation, not for compact or local methods.

The *RBF-QR* method, an alternative for numerically stable computation in the limit as $\varepsilon \rightarrow 0$, was introduced by Fornberg and Piret [?] in context of a sphere, and later extended to planar 2D problems in [?]. The RBF-QR method is simple to implement (less than 100 lines of Matlab code), and it allows solution of large problems that are typically ill-conditioned. Fornberg, Larsson and Flyer [?] successfully applied RBF-QR on large problems with 6000 nodes for globally supported basis functions.

With these two algorithms, global RBF methods have overcome most ill-conditioning issues for small to mid-sized problems. Unfortunately, both Contour-Padé and RBF-QR fail for large enough problems due to ill-conditioning. As the number of RBFs increases beyond a few thousand nodes it is impossible to avoid ill-conditioning of the extremely large interpolation matrix.

This reveals the benefit of local methods, which decrease the number of RBFs and ill-conditioning. However, in the limit as local stencil size increases to include all nodes in a domain, the local and global method are equivalent; thus it is known that local methods also suffer extreme ill-conditioning around 2000 nodes per stencil [?]. To our knowledge, no research has yet been published that applies the RBF-QR method to RBF-FD stencils.

4.1.6 Parallel RBF Implementations

Parallel implementations of RBF methods currently rely on parallel domain decomposition. Depending on the implementation, domain decomposition not only accelerates solution procedures, but can decrease the ill-conditioning that plague all global RBF methods [?]. The ill-conditioning is reduced if each domain is treated as a separate RBF domain, and the boundary update is treated separately. Domain decomposition methods for RBFs were introduced by Beatson et al. [?] in the year 2000 as a way to increase problem sizes into the millions of nodes.

Divo and Kassab [?] used a domain decomposition method with artificial subdomain boundaries for their implementation of a local collocation method [?]. Subdomains are processed independently. The derivative values at artificial boundary points are averaged to maintain global consistency of physical values. Their implementation was designed for a 36 node cluster, but benchmarks and scalability tests are not provided.

Kosec and Šarler [?] used OpenMP to parallelize coupled heat transfer and fluid flow problems on a single workstation. Their test cases used local collocation, explicit time-stepping and Neumann boundary conditions. A speedup factor of 1.85x over serial execution was achieved by executing on two CPU cores; no results from scaling tests were provided.

Stevens et al. [?] mention a parallel implementation under development, but no document is available yet.

Additional RBF implementations are discussed at the end of this chapter in the context of parallel co-processing with the GPU.

In this work, we will add to the above experiences, with the added twist of incorporating an implementation on the GPU (see later chapters).

4.2 PDE as an Interpolation Problem

Following [?], consider a PDE expressed in terms of a (linear) differential operator, \mathcal{L} :

$$\mathcal{L}u = f \quad \text{on } \Omega \tag{4.2}$$

$$u = g \quad \text{on } \Gamma \tag{4.3}$$

where Ω is the interior of the physical domain, Γ is the boundary of Ω and f, g are known explicitly. In the case of a non-linear differential operator, a Newton's iteration, or some other method, can be used to linearize the problem (see e.g., [?]); of course, this increases the complexity of a single time step. Then, the unknown solution, u , which produces the observations on the right hand side can be approximated by an interpolant function u_Φ expressed as a linear combination of radial basis functions, $\{\Phi_j(x) = \Phi(\|x - x_j\|)\}_{j=1}^N$, and polynomial functions $\{P_l(x)\}_{l=1}^M$:

$$u_\Phi(x) = \sum_{j=1}^N c_j \Phi_j(x) + \sum_{l=1}^M d_l P_l(x), \quad P_l(x) \in \Pi_p^D \tag{4.4}$$

where $\Phi_j(x) = \|x - x_j\|$ ($\|\cdot\|$ is standard Euclidean distance). The second sum represents a linear combination of polynomials that enforces zero approximation error when $u(x)$ is a polynomial of degree than or equal to p . The variable D is the problem dimension (i.e., $u_\Phi(x) \in \mathbb{R}^D$). To eliminate degrees of freedom for well-posedness, p should be greater than or equal to the order of the chosen RBF (see Table 4.1) [?]. Note that Equation 4.4 is evaluated at $x = \{x_i\}_{i=1}^n$ known data points through which the interpolant is required to pass with zero residual. We refer to the x_i 's (data points) points as *collocation points*, usually taken as the RBF centers, although this is not a requirement.

To clarify the role of the polynomial part in Equation 4.4, it is necessary to put aside the PDE for the moment and consider only the problem of *scattered data interpolation* with Radial Basis Functions. Borrowing notation from [?, ?], we seek an interpolant, \mathcal{P}_f of the form

$$\begin{aligned} \mathcal{P}_f &= \sum_{j=1}^N c_j \Phi_j(x) = f(x) \\ &= \sum_{j=1}^N \Phi_j(x) c_j = f(x) \\ \vec{\mathcal{P}}_f &= \Phi^T(x) c = f(x) \end{aligned}$$

where \mathcal{P}_f is expressed as a scalar product between the unknown coefficient weights c_j and the radial basis functions $\Phi_j(x)$.

In many cases it is desirable to exactly reproduce functions of degree less than or equal to some order m . For RBF scattered data interpolation in one dimension, this can be achieved by adding a polynomial of order m with $M = \binom{m+1}{1}$ terms (e.g., x^0, x^1, \dots, x^m). In 2D, the terms would be: $1, x, y, xy, x^2y, xy^2, \dots, x^m y^{m-1}, x^{m-1} y^m, x^m y^m$. In \mathbb{R}^D , $M = \binom{m+D}{D}$ [?],

giving

$$f(x) = \sum_{j=1}^N c_j \Phi_j(x) + \sum_{l=1}^M d_l P_l(x), \quad P_l(x) \in \Pi_m^D \quad (4.5)$$

$$f = \Phi^T c + P^T d \quad (4.6)$$

where the second summation (referred to as *interpolation conditions* [?]) ensures the minimum degree of the interpolant. Notice, however, that the interpolation conditions add M new degrees of freedom, so we must provide M *moment conditions* or constraints:

$$\sum_{j=1}^N c_j P_l(x_j) = 0, \quad l = 1, \dots, M$$

or

$$\mathbb{P}^T c = 0$$

to guarantee that the moments of the interpolant vanish for the coefficients obtained, where the matrix

$$\mathbb{P}^T = (P(x_1) \ P(x_2) \ \cdots \ P(x_N))$$

and

$$P(x_j) = (P_1(x_j) \ P_2(x_j) \ \cdots \ P_M(x_j))^T.$$

It is now possible again to write the interpolation as a linear system using Equations 4.6 and 4.2:

$$\vec{\mathcal{P}}_f = \begin{bmatrix} \Phi & \mathbb{P} \\ \mathbb{P}^T & 0 \end{bmatrix} \begin{pmatrix} \vec{c} \\ \vec{d} \end{pmatrix} = \begin{pmatrix} \vec{f} \\ 0 \end{pmatrix} \quad (4.7)$$

This system then produces an interpolant capable of exactly approximating data from polynomials of degree less than or equal to m [?].

Returning to the PDE, we postpone the issue of achieving polynomial precision in approximation. However, it has been shown (see [?, ?]) that some choices of RBFs (e.g. multiquadratics and thin-plate splines [?]) are not positive definite and therefore there is no guarantee that the approximation is well-posed. A sufficient condition for well-posedness is that the matrix is *conditionally positive definite*. In [?], Fasshauer demonstrates that conditional positive definiteness of order m is guaranteed when Equation 4.4 includes the full polynomial part (i.e., interpolation conditions) to match the order of the RBF.

Now, since $u_\Phi(x)$ from Equation 4.4 cannot (in general) satisfy the PDE everywhere, we enforce the PDE at a set of collocation points, which are distributed over both the interior and the boundary. Again, these points do not necessarily coincide with the RBF centers, but it is convenient for this to be true in practice.

4.3 Reconstructing Solutions for PDEs

In the next few sections, we will generate collocation equations based on this general form:

$$\mathcal{L}u_\Phi(x) = f(x) \quad \text{on } \Omega \quad (4.8)$$

$$\mathcal{B}u_\Phi(x) = g(x) \quad \text{on } \Gamma \quad (4.9)$$

where the methods presented below will apply the differential operators, \mathcal{L} and \mathcal{B} , to different choices of u_Φ and different sets of collocation points. In many applications \mathcal{L} is chosen as a differential operator (e.g., ∇, ∇^2) and $\mathcal{B} = I$ (i.e. identity operator for Dirichlet boundary conditions) for PDEs. There are also applications where \mathcal{L} is a convolution operator (see e.g., [?, ?]) capable of smoothing/de-noising a surface reconstructed from point clouds.

For all the methods presented here a linear system is generated

$$\begin{aligned} A \begin{pmatrix} c \\ d \end{pmatrix} &= \begin{pmatrix} f \\ 0 \end{pmatrix} \\ \begin{pmatrix} c \\ d \end{pmatrix} &= A^{-1} \begin{pmatrix} f \\ 0 \end{pmatrix} \end{aligned} \quad (4.10)$$

where matrix A depends on the choice of collocation method. Once the linear system is solved, the value, $u(x_i)$, at an unknown point, x_i , can be reconstructed using the coefficients with the following inner product:

$$u(x_i) = [\Phi^T(x_i) \quad P^T(x_i)] \begin{pmatrix} c \\ d \end{pmatrix} \quad (4.11)$$

Likewise, to find differential quantities we have:

$$\mathcal{L}u(x_i) = [\mathcal{L}(\Phi^T)(x_i) \quad \mathcal{L}(P^T)(x_i)] \begin{pmatrix} c \\ d \end{pmatrix} = A_{\mathcal{L}} \begin{pmatrix} c \\ d \end{pmatrix} \quad (4.12)$$

Here we substitute $A_{\mathcal{L}}$ for the right hand side row vector. Also, Equation 4.10 can be substituted into Equation 4.12 to get

$$\mathcal{L}u(x_i) = A_{\mathcal{L}} A^{-1} \begin{pmatrix} f \\ 0 \end{pmatrix} \quad (4.13)$$

where the vector-matrix inner product ($A_{\mathcal{L}} A^{-1}$) is a row-vector. Since the coefficient vectors c and d are the same for all x_i , we can group the evaluation of $\mathcal{L}u(x_i)$ for $i = 1, \dots, n$ as a matrix-vector multiplication where the matrix rows correspond to ($A_{\mathcal{L}} A^{-1}$) for each x_i .

4.4 Global RBF Method

Consider a PDE expressed in terms of (linear) differential operators, \mathcal{L} and \mathcal{B} :

$$\begin{aligned} \mathcal{L}u &= f && \text{on } \Omega, \\ \mathcal{B}u &= g && \text{on } \Gamma \end{aligned} \quad (4.14)$$

where Ω is the interior of the physical domain, Γ is the boundary of Ω and f, g are known explicitly. In the case of a non-linear differential operator, a Newton's iteration, or some other method, can be used to linearize the problem (see e.g., [?]); of course, this increases the complexity of a single time step.

We solve PDEs of this form with *collocation*. That is, given a number of points in the domain, we seek the derivative and solution values that best satisfy f, g at node locations. **Author's Note:** [Not clear yet. Review Lynch2005 for a better explanation\[?\]](#)

Global RBF collocation methods pose the problem of interpolating a multivariate function $f : \Omega \rightarrow \mathbb{R}$ where $\Omega \subset \mathbb{R}^m$. Given a set of sample values $\{f(x_j)\}_{j=1}^N$ on a discrete set of nodes $X = \{x_j\}_{j=1}^N \subset \Omega$, an approximation \hat{f}_N can be constructed through linear combinations of interpolation functions. Here, we choose univariate, radially symmetric functions based on Euclidean distance ($\|\cdot\|$), and use translates $\phi(x - x_j)$ of a single continuous real valued function ϕ defined on \mathbb{R} and centered at x_j :

$$\phi(x) := \varphi(\|x\|).$$

Here, φ is a Radial Basis Function and ϕ the associated kernel. For simplification $\phi_j(x)$ refers to a kernel centered at x_j ; i.e., $\varphi(\|x - x_j\|)$.

The interpolant $\hat{f}_N(x)$ requires a linear combination of translates:

$$\hat{f}_N(x) = \sum_{j=1}^N c_j \phi_j(x)$$

with real coefficients $\{c_j\}_{j=1}^N$. Assuming the interpolant passes through known values of f ; i.e.,

$$\hat{f}_N(x_i) = f(x_i), \quad 1 \leq i \leq N,$$

allows one to solve for coefficients if the following linear system is uniquely solvable:

$$\sum_{j=1}^N c_j \phi_j(x_i) = f(x_i), \quad 1 \leq i \leq N.$$

This is true if the $N \times N$ matrix Φ produced by the linear system

$$\begin{pmatrix} \phi_1(x_1) & \phi_2(x_1) & \cdots & \phi_N(x_1) \\ \phi_1(x_2) & \phi_2(x_2) & \cdots & \phi_N(x_2) \\ \vdots & \ddots & \ddots & \vdots \\ \phi_1(x_N) & \phi_2(x_N) & \cdots & \phi_N(x_N) \end{pmatrix} \begin{bmatrix} c_1 \\ c_2 \\ \vdots \\ c_N \end{bmatrix} = \begin{bmatrix} f(x_1) \\ f(x_2) \\ \vdots \\ f(x_N) \end{bmatrix}$$

$$\Phi \vec{c} = \vec{f}$$

is nonsingular.

When choosing an appropriate basis function for interpolation, a subset of RBFs have been shown to produce symmetric positive definite Φ , while others are only conditionally positive definite (for more details see [?]). With the latter set, additional polynomial terms are added to constrain the system and enforce positive definiteness, resulting in this expanded system of equations:

$$\begin{aligned} \sum_{j=1}^N c_j \phi_j(x_i) + \sum_{k=1}^M d_k p_k(x_i) &= f(x_i), \quad 1 \leq i \leq N, \\ \sum_{j=1}^N c_j p_k(x_j) &= 0, \quad 1 \leq k \leq M. \end{aligned}$$

$$\begin{pmatrix} \Phi & P \\ P^T & 0 \end{pmatrix} \begin{bmatrix} c \\ d \end{bmatrix} = \begin{bmatrix} \vec{f} \\ 0 \end{bmatrix} \tag{4.15}$$

where $\{p_k\}_{k=1}^M$ is a basis for Π_p^m (the set of polynomials in m variables of degree $\leq p$) and

$$M = \binom{p+m}{m}.$$

Given the coefficients \vec{c} , the function value at a test point x is interpolated by

$$\begin{aligned}\hat{f}_N(x) &= \sum_{j=1}^N c_j \Phi_j(x) + \sum_{l=1}^M d_l P_l(x) \\ &= [\Phi \quad \mathbb{P}] \begin{bmatrix} c \\ d \end{bmatrix} \\ &= \vec{\Phi}_x^T \vec{c} \\ &= \vec{\Phi}_x^T \Phi^{-1} \vec{f}\end{aligned}\tag{4.16}$$

where \vec{c} is substituted by the solution to Equation 4.15. In Equation 4.16 the term $\vec{\Phi}_x^T \Phi^{-1}$, dependent only on node positions, can be evaluated prior to knowing f .

4.5 Kansa's Method

The first method of collocation, *Kansa's method* [?, ?], collocates the solution through known values on the boundary, while constraining the interpolant to satisfy the PDE operator on the interior. This is equivalent to choosing u_Φ according to Equation 4.4. The resulting system is given by [?]; assuming that \mathcal{L} is a linear operator,

$$\mathcal{L}u_\Phi(x_i) = \sum_{j=1}^N c_j \mathcal{L}\Phi_j(x_i) + \sum_{l=1}^M d_l \mathcal{L}P_l(x_i) = f(x_i) \quad i = 1, \dots, n_I \tag{4.17}$$

$$\mathcal{B}u_\Phi(x_i) = \sum_{j=1}^N c_j \mathcal{B}\Phi_j(x_i) + \sum_{l=1}^M d_l \mathcal{B}P_l(x_i) = g(x_i) \quad i = n_I + 1, \dots, n \tag{4.18}$$

$$\sum_{j=1}^N c_j P_l(x_j) = 0 \quad l = 1, \dots, M \tag{4.19}$$

where n_I are the number of interior collocation points, with the number of boundary collocation points equal to $n - n_I$. First, observe that the differential operators are applied directly to the RBFs inside summations, rather than first solving the scattered data interpolation problem and then applying the operator to the interpolant. Second, since the basis functions are known analytically, it is possible (although sometimes painful) to derive $\mathcal{L}\Phi$ (refer to [?] for RBF derivative tables); the same is true for the polynomials P_l .

We can now reformulate Kansa's method as the linear system:

$$\begin{bmatrix} \Phi_{\mathcal{L}} & P_{\mathcal{L}} \\ \Phi_{\mathcal{B}} & P_{\mathcal{B}} \\ P^T & 0 \end{bmatrix} \begin{pmatrix} c \\ d \end{pmatrix} = \begin{pmatrix} f \\ g \\ 0 \end{pmatrix} \tag{4.20}$$

where $\Phi_{\mathcal{L}} = \mathcal{L}\Phi$, $P_{\mathcal{L}} = \mathcal{L}P$ are the interior components (Equation 4.17), $\Phi_{\mathcal{B}}$ and $P_{\mathcal{B}}$ are the boundary components (Equation 4.18), and $P^T = [P_{\mathcal{L}}^T \ P_{\mathcal{B}}^T]$ are moment constraints for both interior and boundary polynomial parts (Equation 4.19). From Equation 4.20 it should be clear why Kansa's method is also known as the *Unsymmetric* collocation method.

Recall from Chapter ?? that the matrix in Equation 4.20 has no guarantee of non-singularity [?]; however, singularities are rare in practice [?].

4.6 Fasshauer's Method

Fasshauer's method [?] addresses the problem of singularity in Kansa's method by assuming the interpolation to be Hermite. That is, it requires higher differentiability of the basis functions (they must be at least C^k -continuous if \mathcal{L} is of order k). Leveraging this assumption, Fasshauer's method chooses:

$$u_{\Phi}(x_i) = \sum_{j=1}^{N_I} c_j \mathcal{L}\Phi_j(x_i) + \sum_{j=N_I+1}^N c_j \mathcal{B}\Phi_j(x_i) + \sum_{l=1}^M d_l P_l(x_i) \quad (4.21)$$

as the interpolant passing through collocation points. Note N_I is used here to specify the number of RBF centers in the interior of Ω . Here the interpolant is similar to Equation 4.4, but a change of basis functions is used for the expansion: $\mathcal{L}\Phi_j(x)$ on the interior and $\mathcal{B}\Phi_j(x)$ on the boundary.

Collocating (i.e., substituting Equation 4.21 into Equations 4.17-4.19) we get:

$$\sum_{j=1}^{N_I} c_j \mathcal{L}^2 \Phi_j(x_i) + \sum_{j=N_I+1}^N c_j \mathcal{L}\mathcal{B}\Phi_j(x_i) + \sum_{l=1}^M d_l \mathcal{L}P_l(x_i) = f(x_i) \quad i = 1, \dots, n_I \quad (4.22)$$

$$\sum_{j=1}^{N_I} c_j \mathcal{B}\mathcal{L}\Phi_j(x_i) + \sum_{j=N_I+1}^N c_j \mathcal{B}^2 \Phi_j(x_i) + \sum_{l=1}^M d_l \mathcal{B}P_l(x_i) = g(x_i) \quad i = n_I + 1, \dots, \#(4.23)$$

$$\sum_{j=1}^{N_I} c_j \mathcal{L}P_l(x_j) + \sum_{j=N_I+1}^N c_j \mathcal{B}P_l(x_j) = 0 \quad l = 1, \dots, M \quad (4.24)$$

which is reformatted as the linear system:

$$\begin{bmatrix} \Phi_{\mathcal{L}\mathcal{L}} & \Phi_{\mathcal{L}\mathcal{B}} & P_{\mathcal{L}} \\ \Phi_{\mathcal{B}\mathcal{L}} & \Phi_{\mathcal{B}\mathcal{B}} & P_{\mathcal{B}} \\ P_{\mathcal{L}}^T & P_{\mathcal{B}}^T & 0 \end{bmatrix} \begin{pmatrix} c \\ d \end{pmatrix} = \begin{pmatrix} f \\ g \\ 0 \end{pmatrix} \quad (4.25)$$

Note that $\Phi_{\mathcal{L}\mathcal{L}}$ represents the first summation in Equation 4.22. The linear system generated by Fasshauer's method reveals an interesting structure: namely, the subscripts \mathcal{L} and \mathcal{B} show blocks of influence in the matrix. For example, the interior RBF centers influence collocation on the interior collocation points ($\Phi_{\mathcal{L}\mathcal{L}}$), boundary centers influence collocation on the interior ($\Phi_{\mathcal{L}\mathcal{B}}$), interior centers influence collocation on the boundary ($\Phi_{\mathcal{B}\mathcal{L}}$), and so forth. In the case where the collocation points and RBF centers do not coincide, the subscripts would also indicate which set of points the operators are applied to [?].

The symmetry of Fasshauer's (*symmetric collocation*) method is apparent in Equation 4.25. Likewise, it is clear that the symmetric method requires more storage and computation to solve compared to Kansa's method. However, based on the assumption that collocation points coincide with RBF centers, the symmetry reduces storage requirements by half.

4.7 Direct Collocation

In *Direct collocation* (see [?, ?], the interpolant is chosen as Equation 4.4 (the same as Kansa's method). However, the Direct method collocates both the interior and boundary operators at the boundary points:

$$\sum_{j=1}^N c_j \mathcal{L} \Phi_j(x_i) + \sum_{l=1}^M d_l \mathcal{L} P_l(x_i) = f(x_i) \quad i = 1, \dots, n \quad (4.26)$$

$$\sum_{j=1}^N c_j \mathcal{B} \Phi_j(x_i) + \sum_{l=1}^M d_l \mathcal{B} P_l(x_i) = g(x_i) \quad i = 1, \dots, n_B = n - n_I \quad (4.27)$$

$$\sum_{j=1}^N c_j P_l(x_j) = 0 \quad l = 1, \dots, M \quad (4.28)$$

Reformulating as a linear system we get:

$$\begin{bmatrix} \Phi_{\mathcal{L}} & P_{\mathcal{L}} \\ \Phi_{\mathcal{B}} & P_{\mathcal{B}} \\ P^T & 0 \end{bmatrix} \begin{pmatrix} c \\ d \end{pmatrix} = \begin{pmatrix} f \\ g \\ 0 \end{pmatrix} \quad (4.29)$$

While the final system in Equation 4.29 is structured the same as Kansa's method (Equation 4.20), careful inspection of the index i in Equations 4.17 and 4.26 reveals that Direct collocation produces a larger system.

4.8 Local Methods

In all three cases above, the collocation required interpolation with globally supported RBFs. However, the latest trend is to use RBFs defined with some cut-off radius to enforce compact support. In some cases, authors have used the compact support to produce a single (large) sparse system for interpolation (see e.g., [?, ?, ?, ?, ?]). Other approaches use compact support to produce local linear systems defined at each collocation point. Examples of this include [?, ?] for Kansa's method, [?, ?, ?] for Fasshauer's method. To our knowledge no one has considered local Direct collocation. Also, instead of specifying a cut-off radius, some authors specify the exact stencil size (i.e., number of neighboring points to include); see e.g., [?, ?].

After observing the general structure of the symmetric and unsymmetric collocation methods above, it is necessary only to present the symmetric (i.e. Fasshauer's) local method

and note that in the unsymmetric case certain blocks will be zero allowing the system to shrink.

The formula for the interpolant local to the (k)-th collocation point (i.e., RBF center) is given by:

$$u_{\Phi}^{(k)}(x_i) = \sum_{j(k)=1}^{N_I} c_j^{(k)} \mathcal{L}\Phi_j(x_i) + \sum_{j(k)=N_I+1}^{N_S} c_j^{(k)} \mathcal{B}\Phi_j(x_i) + \sum_{l=1}^M d_l^{(k)} P_l(x_i) \quad (4.30)$$

where N_S represents the number of points that defines the local stencil; N is possibly a function of the cut-off radius in the RBF, N_I is the number of interior stencil points (those points of the stencil that lie in the interior of Ω). The index j is a function of the stencil center k allowing the system to include a local neighborhood of stencil points.

Collocating produces a linear system with similar structure to the global collocation problem, but the dimensions are much smaller:

$$\begin{bmatrix} \Phi_{\mathcal{L}\mathcal{L}} & \Phi_{\mathcal{L}\mathcal{B}} & P_{\mathcal{L}} \\ \Phi_{\mathcal{B}\mathcal{L}} & \Phi_{\mathcal{B}\mathcal{B}} & P_{\mathcal{B}} \\ P_{\mathcal{L}}^T & P_{\mathcal{B}}^T & 0 \end{bmatrix} \begin{pmatrix} c^{(k)} \\ d^{(k)} \end{pmatrix} = \begin{pmatrix} f \\ g \\ 0 \end{pmatrix} \quad (4.31)$$

Solving this system gives an interpolant locally defined around the stencil center. Note that approximating the PDE solution $u(x)$ requires finding the stencil center nearest x , then using the local interpolant for that stencil. Since interpolation is local (i.e., $c_j^{(k)}$'s are unique to each RBF center), reconstructing the derivatives with Equation 4.12 is limited to an inner product for each center rather than the matrix-vector grouping possible with global RBFs. Note that because the interpolants are local, there is no notion of global continuity/smoothness of the solution.

4.9 RBF-FD

While most of the literature surrounding RBFs for PDEs involves collocation, an alternative method does exist: RBF-FD. RBF-FD is a hybrid of RBF scattered data interpolation and Finite Difference stencils. The idea behind finite-difference stencils is to express various derivative operators as a linear combination of the unknown functional values in the neighborhood of the point where an approximation to the derivative operator is desired. Common approximations such as upwind differencing, center differencing, higher order approximations, and even spectral operators, are of this form. A common approach to building such discrete operators is to form a local interpolant in a neighborhood of the target point, and simply differentiate it analytically. This is the approach taken in RBF-FD, which allows for stencils with irregular placement and number of nodes, and assigns their weights based on an RBF [?]. For a detailed derivation of the method see [?, ?, ?, ?]. Such an approach leads to very simple implementations of time-advancement schemes, whether explicit or implicit. The solution at the new time step is simply some linear (if $\mathcal{L}()$ is linear), nonlinear otherwise, combination of the unknown functional values (if implicit scheme) or known functional value (if explicit scheme).

Key challenges lie in the choice of grid, the choice of stencil, whether or not to change the support as a function of the stencil, how to guaranty the stability of the differentiation operator after discretizaiton, etc.

4.10 Time Stepping

Until now we have assumed that the differential operator \mathcal{L} had no time dependency. However, many important problems are a function of time:

$$\frac{\partial u(x; t)}{\partial t} = \mathcal{L}u(x; t) \quad (4.32)$$

$$\mathcal{B}(u(x; t)) = g(x) \quad (4.33)$$

with some initial condition $u(x; 0) = f(x)$. Both collocation methods and RBF-FD methods can be used to discretize the right hand side. In the case of collocation, the coefficients c are now functions of time. On the other hand, it is the unknown functional values that are considered to be the unknown, time-dependent variables when RBF-FD is used.

The general approach for collocation methods is to first use the initial and boundary conditions to get an initial set of coefficients $c(t), d(t)$, whereas before, (c, d) denotes the list of unknown coefficients of the RBF basis functions and the monomials. Prior to spatial discretization, Equation 4.33 is updated using any of the standard methods for time-stepping, such as Euler, Crank-Nicholson, or higher order Runge-Kutta methods to name a few. Here we demonstrate the approach with the Euler method (which is explicit) for simplicity. In practice, a higher order method would be chosen, perhaps an implicit method. The Euler method can be written as

$$u(x; t_{n+1}) = u(x; t_n) + \Delta t(\mathcal{L}u(x; t_n)) \quad (4.34)$$

Since $u(x; 0) = u(x; t_0)$ is known as a function of $(c, d) = A^{-1}u(x; t_n)$, it is straightforward to compute the coefficients, and from there, $\mathcal{L}(u(x; t_n))$.

Equation 4.34 is solved via a two step algorithm:

1. solve Equation 4.13 to approximate $\mathcal{L}u(x; t_n)$ for the current time-step
2. update $u(x; t_{n+1})$

Initially, $A_{\mathcal{L}}A^{-1}$ is computed once during the first time-step, and the first part of the algorithm is reduced to an inner product to get $\mathcal{L}u(x; t_n)$. Once $u(x; t_{n+1})$ is updated, one repeats the procedure with $u(x; t_{n+1})$ as a new initial condition. For collocation methods, one must recompute all the coefficients at each timestep (i.e. recompute A^{-1}), which is expensive. On the other hand, for a static grid, $A_{\mathcal{L}}A^{-1}$ can be precomputed and there is no longer any need to work in coefficient space. In this case, we have a type of RBF-FD method, which is solely expressed in terms of functional values. Of course, in a "pure" RBF-FD method, one has expressions for the various derivatives and operators as a linear combination of functional values (the derived stencil being an implicit function of the RBFs and their location).

There is a huge literature on time-stepping schemes for finite-difference, finite-volume, spectral methods, etc., and many of these methods can be adopted to solve time-dependent problems on meshless grids. Although there are surely issues that are particular to RBFS, we do not discuss them here. Instead, we will include our experiences in the final thesis.

For initial value problems with moving nodes, the coefficients must be updated at each time-step whether working with collocation or RBF-FD. We do not consider moving node problems in this thesis.

Part II

The RBF-FD Method

CHAPTER 5

THE RBF-FD METHOD

This is followed by a description of Radial Basis Function-generated Finite Differences and examples of operators approximated by the method within this work.

The choice to study RBF-FD within this dissertation is motivated by two factors. First, RBF-FD represents one of the latest developments within the RBF community. The method was formally introduced in 2003 but has yet to obtain the critical-mass following necessary for the method's use in large-scale scientific models. Our goal throughout the dissertation has been to scale RBF-FD to complex problems on high resolution meshes, and to lead the way for its adoption in high performance computational geophysics. Second, RBF-FD inherits many of the positive features from global and local collocation schemes, but sacrifices others for the sake of significantly reduced computational complexity and increased parallelism. We capitalize on the inherent parallelism to develop a collection of multi-GPU test cases that span the compute nodes of a Top 500 supercomputer. Our effort leads the way for application of RBF-FD in the modern field of high performance computing where GPU accelerators will be key to breaching the exa-scale barrier in computing [?].

Rather than solely focus on the optimizations of said algorithms on the GPU, we dedicate significant attention to the practical application of RBF-FD to interesting problems in geophysics. This means we have walked a fine line between research topics to both apply the method to

CHAPTER 6

RBF METHODS FOR PDE SOLUTIONS

6.1 The RBF-generated Finite Differences Method

For the RBF-FD method, derivatives of $u(x)$ are weighted combinations of a small neighborhood of N_s nodes (i.e., a stencil with $N_s \ll N$):

$$\mathcal{L}u(x_1) \approx \sum_{j=1}^{N_s} c_j u(x_j)$$

where $\mathcal{L}u$ denotes a linear differential quantity over u (e.g., $\mathcal{L}u = \frac{du}{dx}$). Here, c_j are unknown, and obtained by enforcing the reconstruction:

$$\mathcal{L}\phi_j(x_1) = \sum_{i=1}^{N_s} c_i \phi_j(x_i) \quad \text{for } j = 1, 2, \dots, N_s$$

with $\mathcal{L}\phi_j$ provided by analytically applying the differential operator to the kernel function. This is equivalent to:

$$\begin{pmatrix} \phi_1(x_1) & \phi_1(x_2) & \cdots & \phi_1(x_{N_s}) \\ \phi_2(x_1) & \phi_2(x_2) & \cdots & \phi_2(x_{N_s}) \\ \vdots & \ddots & \ddots & \vdots \\ \phi_{N_s}(x_1) & \phi_{N_s}(x_2) & \cdots & \phi_{N_s}(x_{N_s}) \end{pmatrix} \begin{bmatrix} c_1 \\ c_2 \\ \vdots \\ c_{N_s} \end{bmatrix} = \begin{bmatrix} \mathcal{L}\phi_1(x_1) \\ \mathcal{L}\phi_2(x_1) \\ \vdots \\ \mathcal{L}\phi_{N_s}(x_1) \end{bmatrix} \quad (6.1)$$

$$\Phi_s \vec{c} = \vec{\Phi}_{\mathcal{L}_s}. \quad (6.2)$$

Solving Equation 6.2 for stencils weights, \vec{c} , and substituting into Equation 12.3 reveals the similarity between RBF-FD and the general problem of RBF interpolation (see Equation 4.16 and discussion):

$$\begin{aligned} \mathcal{L}u(x_1) &\approx (\Phi_s^{-1} \vec{\Phi}_{\mathcal{L}_s})^T \vec{u} \\ &\approx \vec{\Phi}_{\mathcal{L}_s}^T \Phi_s^{-T} \vec{u} \end{aligned}$$

Again, based on the choice of RBF, positive definiteness of the system is ensured by adding the same constraints as Equation 4.15, but note differential quantities for $\vec{\Phi}$ on the

right hand side:

$$\begin{pmatrix} \Phi_s & \mathbb{P} \\ \mathbb{P}^T & 0 \end{pmatrix} \begin{bmatrix} c \\ d \end{bmatrix} = \begin{bmatrix} \vec{\Phi}_{\mathcal{L}_s} \\ \vec{\mathbb{P}}_{\mathcal{L}} \end{bmatrix}. \quad (6.3)$$

Equation 6.3 is strictly for the stencil centered at node x_1 . To obtain weights for all stencils in the domain, a total of N such systems must be solved.

6.2 Stencil-based Derivative Approximation

The RBF-FD method is similar to classical Finite Differences in that differential quantities at a single node referred to as a *center*, \mathbf{x}_j , is approximated by weighted combinations of function values at $n - 1$ neighboring nodes. The center node and its neighbors define a *stencil*, $\{\mathbf{x}\}_{i=1}^n$, in a localized (small) region or *neighborhood* of the domain.

Figure 6.1 provides two examples of RBF-FD stencils. First, Figure 6.1a illustrates a single stencil of size $n = 13$ in a domain of randomly distributed nodes. The stencil center is represented by a green square, with the 12 neighboring nodes connected via red edges. The purple circle, the minimum covering circle for the stencil, demonstrates that the stencil contains only the 12 nearest neighbors of the center node. In Figure 6.1b, a larger RBF-FD stencil of size $n = 75$ on the unit sphere is shown as red and blue disks surrounding the center represented as a square. Green disks are nodes outside of the stencil. The radii and color the red and blue disks represent the magnitude and sign of the RBF-FD weights determined to calculate a derivative quantity at the stencil center.

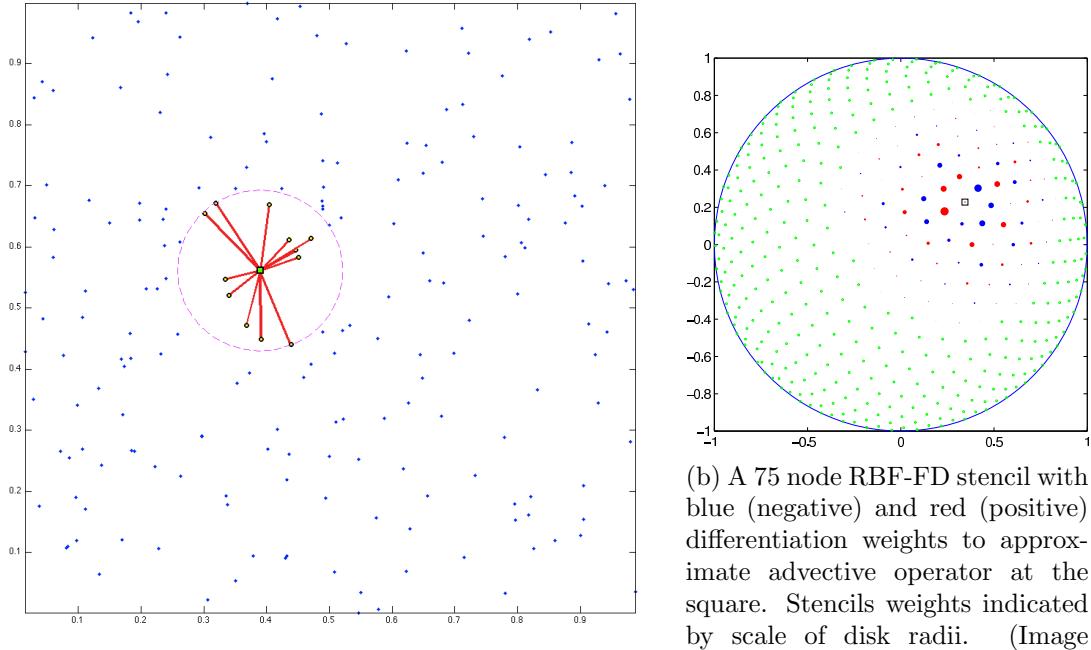
Weights for both classical Finite Difference and RBF-FD can be obtained through the solution of linear systems. In the case of Finite Difference, the system is a Vandermonde matrix. For RBF-FD the system is based on a symmetric distance matrix. [Author's Note: Need better description and ref:](#) The key difference between FD and RBF-FD systems is the singularity that occurs when two nodes swap.

6.3 RBF-FD weights

Given a set of function values, $\{u(\mathbf{x}_j)\}_{j=1}^N$, on a set of N nodes $\{\mathbf{x}_j\}_{j=1}^N$, the operator \mathcal{L} acting on $u(\mathbf{x})$ evaluated at \mathbf{x}_j , is approximated by a weighted combination of function values, $\{u(\mathbf{x}_i)\}_{i=1}^n$, in a small neighborhood of \mathbf{x}_j , where $n \ll N$ defines the size of the stencil.

$$\mathcal{L}u(\mathbf{x}) |_{\mathbf{x}=\mathbf{x}_j} \approx \sum_{i=1}^n w_i u(\mathbf{x}_i) + w_{n+1} p_0 \quad (6.4)$$

The RBF-FD weights, w_i , are found by enforcing that they are exact within the space spanned by the RBFs $\phi_i(\epsilon r) = \phi(\epsilon \|\mathbf{x} - \mathbf{x}_i\|)$, centered at the nodes $\{\mathbf{x}_i\}_{i=1}^n$, with $r = \|\mathbf{x} - \mathbf{x}_i\|$ being the distance between where the RBF is centered and where it is evaluated as measured in the standard Euclidean 2-norm. Various studies show [?, ?, ?, ?] that better



(a) A 13 node RBF-FD stencil of randomly distributed nodes. The stencil centered at the green square contains the 12 nearest neighbors contained within the minimum covering circle drawn in purple.

(b) A 75 node RBF-FD stencil with blue (negative) and red (positive) differentiation weights to approximate advective operator at the square. Stencils weights indicated by scale of disk radii. (Image courtesy of Bengt Fornberg and Natasha Flyer)

Figure 6.1: Examples of stencils computable with RBF-FD [Author's Note: Consider: show example FD stencil \(5pt with regular grid\)](#)

accuracy is achieved when the interpolant can exactly reproduce a constant, p_0 . Assuming $p_0 = 1$, the constraint $\sum_{i=1}^n w_i = \mathcal{L}1|_{\mathbf{x}=\mathbf{x}_j} = 0$ completes the system:

$$\begin{pmatrix} \phi(\epsilon \|\mathbf{x}_1 - \mathbf{x}_1\|) & \phi(\epsilon \|\mathbf{x}_1 - \mathbf{x}_2\|) & \cdots & \phi(\epsilon \|\mathbf{x}_1 - \mathbf{x}_n\|) & 1 \\ \phi(\epsilon \|\mathbf{x}_2 - \mathbf{x}_1\|) & \phi(\epsilon \|\mathbf{x}_2 - \mathbf{x}_2\|) & \cdots & \phi(\epsilon \|\mathbf{x}_2 - \mathbf{x}_n\|) & 1 \\ \vdots & \ddots & \ddots & \vdots & \vdots \\ \phi(\epsilon \|\mathbf{x}_n - \mathbf{x}_1\|) & \phi(\epsilon \|\mathbf{x}_n - \mathbf{x}_2\|) & \cdots & \phi(\epsilon \|\mathbf{x}_n - \mathbf{x}_n\|) & 1 \\ 1 & 1 & \cdots & 1 & 0 \end{pmatrix} \begin{bmatrix} w_1 \\ w_2 \\ \vdots \\ w_n \\ w_{n+1} \end{bmatrix} = \begin{bmatrix} \mathcal{L}\phi(\epsilon \|\mathbf{x} - \mathbf{x}_1\|)|_{\mathbf{x}=\mathbf{x}_j} \\ \mathcal{L}\phi(\epsilon \|\mathbf{x} - \mathbf{x}_2\|)|_{\mathbf{x}=\mathbf{x}_j} \\ \vdots \\ \mathcal{L}\phi(\epsilon \|\mathbf{x} - \mathbf{x}_n\|)|_{\mathbf{x}=\mathbf{x}_j} \\ 0 \end{bmatrix}, \quad (6.5)$$

where w_{n+1} is ignored after the matrix in (6.5) is inverted. This $n \times n$ system solve is repeated for each stencil center \mathbf{x}_j , $j = 1 \dots N$, to form the N rows of the DM with n non-zeros ($n \ll N$) per row. As an example, if \mathcal{L} is the identity operator, then the above procedure leads to RBF-FD interpolation. If $\mathcal{L} = \frac{\partial}{\partial x}$, one obtains the DM that approximates the first derivative in x . In the context of time-dependent PDEs, the stencil weights remain constant for all time-steps when the nodes are stationary. Therefore, the calculation of the differentiation weights is performed once in a single preprocessing step of $O(n^3N)$ FLOPs. Improved efficiency is achieved by processing multiple right hand sides in one pass, calculating the

weights corresponding to all required derivative quantities (i.e., $\frac{\partial}{\partial x}$, $\frac{\partial}{\partial y}$, ∇^2 , etc.).

For each of the N small system solves of Equation (6.5), the n nearest neighbors to \mathbf{x}_j need to be located. This can be done efficiently using neighbor query algorithms or spatial partitioning data-structures such as Locality Sensitive Hashing (LSH) and k D-Tree. Different query algorithms often have a profound impact on the DM structure and memory access patterns. We choose a Raster (ijk) ordering LSH algorithm [?] leading to the matrix structure in Figures 8.4 and 8.5. While querying neighbors for each stencil is an embarrassingly parallel operation, the node sets used here are stationary and require stencil generation only once. Efficiency and parallelism for this task has little impact on the overall run-time of tests, which is dominated by the time-stepping. We preprocess node sets and generate stencils serially, then load stencils and nodes from disk at run-time. In contrast to the RBF-FD view of a static grid, Lagrangian/particle based PDE algorithms promote efficient parallel variants of LSH in order to accelerate querying neighbors at each time-step [?, ?].

6.4 Weight Operators

Throughout the development of our parallel code we have verified code correctness through the solution of a variety of PDEs. Here we provide a list of operators we have tested and their corresponding equations for the RHS of Equation 6.5 necessary to compute RBF-FD weights.

The standard first derivatives $\frac{\partial}{\partial x}$, $\frac{\partial}{\partial y}$, $\frac{\partial}{\partial z}$ are produced by the chain rule

$$\frac{\partial \phi}{\partial x} = \frac{dr}{dx} \frac{d\phi}{dr} = \frac{(x - x_j)}{r} \frac{d\phi}{dr} \quad (6.6)$$

$$\frac{\partial \phi}{\partial y} = \frac{dr}{dy} \frac{d\phi}{dr} = \frac{(y - y_j)}{r} \frac{d\phi}{dr} \quad (6.7)$$

$$\frac{\partial \phi}{\partial z} = \frac{dr}{dz} \frac{d\phi}{dr} = \frac{(z - z_j)}{r} \frac{d\phi}{dr} \quad (6.8)$$

where $\frac{\partial \phi}{\partial r}$ for the Gaussian RBFs is given by:

6.4.1 Laplacian (∇^2)

2D:

3D:

6.4.2 Laplace-Beltrami (Δ_S) on the Sphere

The ∇^2 operator can be represented in spherical polar coordinates for \mathbb{R}^3 as:

$$\nabla^2 = \underbrace{\frac{1}{r} \frac{\partial}{\partial r} \left(r^2 \frac{\partial}{\partial r} \right)}_{\text{radial}} + \underbrace{\frac{1}{r^2} \Delta_S}_{\text{angular}} \quad (6.9)$$

where Δ_S is the Laplace-Beltrami operator—i.e., the Laplacian operator constrained to the surface of the sphere. This form nicely illustrates the separation of components into radial and angular terms.

In the case of PDEs solved on the unit sphere, there is no radial term, so we have:

$$\nabla^2 \equiv \Delta_S. \quad (6.10)$$

Although this originated in the spherical coordinate system, [?] introduced the following Laplace-Beltrami operator for the surface of the sphere:

$$\Delta_S = \frac{1}{4} \left[(4 - r^2) \frac{\partial^2}{\partial r^2} + \frac{4 - 3r^2}{r} \frac{\partial}{\partial r} \right], \quad (6.11)$$

where r is the Euclidean distance between nodes of an RBF-FD stencil and is independent of our choice of coordinate system.

6.4.3 Constrained Gradient ($P_x \cdot \nabla$) on the Sphere

Additionally following [?, ?], the gradient operator must also be constrained to the sphere with this projection matrix:

$$P = I - \mathbf{x}\mathbf{x}^T = \begin{pmatrix} (1 - x_1^2) & -x_1x_2 & -x_1x_3 \\ -x_1x_2 & (1 - x_2^2) & -x_2x_3 \\ -x_1x_3 & -x_2x_3 & (1 - x_3^2) \end{pmatrix} = \begin{pmatrix} P_{x_1} \\ P_{x_2} \\ P_{x_3} \end{pmatrix} \quad (6.12)$$

where \mathbf{x} is the unit normal at the stencil center.

The direct method of computing RBF-FD weights for the projected gradient for $\mathbf{P} \cdot \nabla$ is presented in [?]. When solving for the weights, we apply the projection on the right hand side of our small linear system. We let $\mathbf{x} = (x_1, x_2, x_3)$ be the stencil center, and $\mathbf{x}_k = (x_{1,k}, x_{2,k}, x_{3,k})$ indicate an RBF-FD stencil node.

Using the chain rule, and assumption that

$$r(\mathbf{x}_k - \mathbf{x}) = \|\mathbf{x}_k - \mathbf{x}\| = \sqrt{(x_{1,k} - x_1)^2 + (x_{2,k} - x_2)^2 + (x_{3,k} - x_3)^2},$$

we obtain the unprojected gradient of ϕ as

$$\nabla\phi(r(\mathbf{x}_k - \mathbf{x})) = \frac{\partial r}{\partial \mathbf{x}} \frac{\partial \phi(r(\mathbf{x}_k - \mathbf{x}))}{\partial r} = -(\mathbf{x}_k - \mathbf{x}) \frac{1}{r(\mathbf{x}_k - \mathbf{x})} \frac{\partial \phi(r(\mathbf{x}_k - \mathbf{x}))}{\partial r}$$

Applying the projection matrix gives

$$\begin{aligned} \mathbf{P}\nabla\phi(r(\mathbf{x}_k - \mathbf{x})) &= -(\mathbf{P} \cdot \mathbf{x}_k - \mathbf{P} \cdot \mathbf{x}) \frac{1}{r(\mathbf{x}_k - \mathbf{x})} \frac{\partial \phi(r(\mathbf{x}_k - \mathbf{x}))}{\partial r} \\ &= -(\mathbf{P} \cdot \mathbf{x}_k - 0) \frac{1}{r(\mathbf{x}_k - \mathbf{x})} \frac{\partial \phi(r(\mathbf{x}_k - \mathbf{x}))}{\partial r} \\ &= -(I - \mathbf{x}\mathbf{x}^T)(\mathbf{x}_k) \frac{1}{r(\mathbf{x}_k - \mathbf{x})} \frac{\partial \phi(r(\mathbf{x}_k - \mathbf{x}))}{\partial r} \\ &= \begin{pmatrix} x\mathbf{x}^T \mathbf{x}_k - x_k \\ y\mathbf{x}^T \mathbf{x}_k - y_k \\ z\mathbf{x}^T \mathbf{x}_k - z_k \end{pmatrix} \frac{1}{r(\mathbf{x}_k - \mathbf{x})} \frac{\partial \phi(r(\mathbf{x}_k - \mathbf{x}))}{\partial r} \end{aligned}$$

Thus, we directly compute the weights for $P_x \cdot \nabla$ using these three RHS in Equation 6.5:

$$P \frac{\partial}{\partial x_1} = (x_1 \mathbf{x}^T \mathbf{x}_k - x_{1,k}) \frac{1}{r(\mathbf{x}_k - \mathbf{x})} \frac{\partial \phi(r(\mathbf{x}_k - \mathbf{x}))}{\partial r} \Big|_{\mathbf{x}=\mathbf{x}_j} \quad (6.13)$$

$$P \frac{\partial}{\partial x_2} = (x_2 \mathbf{x}^T \mathbf{x}_k - x_{2,k}) \frac{1}{r(\mathbf{x}_k - \mathbf{x})} \frac{\partial \phi(r(\mathbf{x}_k - \mathbf{x}))}{\partial r} \Big|_{\mathbf{x}=\mathbf{x}_j} \quad (6.14)$$

$$P \frac{\partial}{\partial x_3} = (x_3 \mathbf{x}^T \mathbf{x}_k - x_{3,k}) \frac{1}{r(\mathbf{x}_k - \mathbf{x})} \frac{\partial \phi(r(\mathbf{x}_k - \mathbf{x}))}{\partial r} \Big|_{\mathbf{x}=\mathbf{x}_j} \quad (6.15)$$

Alternatively, assuming weights for operators $\nabla = \frac{\partial}{\partial x_1}, \frac{\partial}{\partial x_2}, \frac{\partial}{\partial x_3}$ are already computed, the projected operator could be constructed via weighting the unprojected gradient components. For example:

Author's Note: Show accuracy of derivative approximation when using projected operator vs linear combinations of dx,dy,dz operators

6.5 Stabilization: Hyperviscosity

For RBF-FD, differentiation matrices encode convective operators of the form

$$D = \alpha \frac{\partial}{\partial \lambda} + \beta \frac{\partial}{\partial \theta} \quad (6.16)$$

where α and β are a function of the fluid velocity. The convective operator, discretized through RBF-FD, has eigenvalues in the right half-plane causing the method to be unstable [?, ?]. Stabilization of the RBF-FD method is achieved through the application of a hyperviscosity filter to Equation (6.16) [?]. By using Gaussian RBFs, $\phi(r) = e^{-(\epsilon r)^2}$, the hyperviscosity (a high order Laplacian operator) simplifies to

$$\Delta^k \phi(r) = \epsilon^{2k} p_k(r) \phi(r) \quad (6.17)$$

where k is the order of the Laplacian and $p_k(r)$ are multiples of generalized Laguerre polynomials that are generated recursively (see [?]: Section 3.2). We assume a 2D Laplacian operator when working on the surface of the sphere since a local stencil can be viewed as lying on a plane.

In the case of parabolic and hyperbolic PDEs, hyperviscosity is added as a filter to the right hand side of the evaluation. For example, at the continuous level, the equation solved takes the form

$$\frac{\partial u}{\partial t} = -Du + Hu, \quad (6.18)$$

where D is the PDE operator, and H is the hyperviscosity filter operator. Applying hyperviscosity shifts all the eigenvalues of D to the left half of the complex plane. This shift is controlled by k , the order of the Laplacian, and a scaling parameter γ_c , defined by

$$H = \gamma \Delta^k = \gamma_c N^{-k} \Delta^k.$$

Given a choice of ϵ (see Section 11.3.1), it was found experimentally that $\gamma = \gamma_c N^{-k}$ provides stability and good accuracy for all values of N considered here. It also ensures

that the viscosity vanishes as $N \rightarrow \infty$ [?]. In general, the larger the stencil size, the higher the order of the Laplacian. This is attributed to the fact that, for convective operators, larger stencils treat a wider range of modes accurately. As a result, the hyperviscosity operator should preserve as much of that range as possible. The parameter γ_c must also be chosen with care and its sign depends on k (for k even, γ_c will be negative and for k odd, it will be positive). If γ_c is too large, the eigenvalues move outside the stability domain of our time-stepping scheme and/or eigenvalues corresponding to lower physical modes are not left intact, reducing the accuracy of our approximation. If γ_c is too small, eigenvalues remain in the right half-plane [?, ?].

CHAPTER 7

TEMP

7.1 Fragments (integrate above)

Stabilization of the RBF-FD method is achieved through the application of a hyperviscosity filter [?]. By assuming the use of Gaussian RBFs, $\phi(r) = e^{-(\epsilon r)^2}$, the hyperviscosity operator simplifies to

$$\Delta^k \phi(r) = \epsilon^{2k} p_k(r) \phi(r). \quad (7.1)$$

The multiples of generalized Laguerre polynomials, $p_k(r)$, are obtained through the following recursive relation:

$$\begin{cases} p_0(r) &= 1, \\ p_1(r) &= 4(\epsilon r)^2 - 2d, \\ p_k(r) &= 4((\epsilon r)^2 - 2(k-1) - \frac{d}{2})p_{k-1}(r) - 8(k-1)(2(k-1)-2+d)p_{k-2}(r), \quad k = 2, 3, \dots \end{cases}$$

where d is the dimension of the problem. We assume $d = 2$ below when working on the surface of the sphere.

Many algorithms exist to query the k -nearest neighbors (equivalently all nodes in the minimum/smallest enclosing circle). Some algorithms overlay a grid similar to Locality Sensitive Hashing and query such as... [?].

Conditioning. Conditioning is defined as:

With condition number estimates we can choose a proper support parameter for uniformly distributed node sets.

Alternative algorithms exist for solving for RBF-FD weights when the systems are overly ill-conditioned. Currently, we have an unpublished algorithm ContourSVD available to play with (demonstrate accuracy improvements).

Node Placement – Centroidal Voronoi Tessellation. We make the assumption for now that we use regularly distributed nodes. Centroidal Voronoi tessellations provide reasonably good distributions for solutions. Examples (heat ellipse, ellipsoid, sphere, square cavity).

The motivation behind RBF-FD is generality/functionality in the numerical method. Scattered nodes are supported. Distribution requires proper choice of support, and tight nodes result in increased conditioning

7.2 Approximate Nearest Neighbor (ANN) Query

RBF methods are traditionally described as general and meshless in that they apply to unstructured clouds of points in arbitrary dimensions. However, although the term meshless implies a method capable of operating with no node connectivity, all numerical methods—meshless RBF methods included—connect nodes in the domain. For example, the “meshless” global RBF method connects every node in the domain to all other nodes. Compact support or local RBF methods like RBF-FD limit connections to nodes that lie within a predetermined radius.

The connections between nodes form a directed adjacency graph with edges that dictate the paths along which data/phenomena can travel. For example, a plus shaped stencil of five points with a center node and four neighboring nodes allows values to propagate north, south, east and west; not northeast, southeast, etc.

They are robust and function on scattered point clouds. RBF-FD in particular requires stencils to be generated from n nearest neighbors to a stencil center. The cost of these neighbor queries can vary greatly depending on the choice of algorithm or data-structure used to make the query.

For example, in general brute force is inefficient. The author of [?] queries n nearest neighbors for a compact-support RBF partition of unity example with a k -D tree. In [?, ?] a k -D Tree is leveraged for all neighbor queries for RBF-FD.

In our work in [?] an alternative to k -D tree was leveraged, based loosely on Locality Sensitive Hashing.

7.2.1 k -D Tree

Most of the RBF community leverages the k -D tree, due to its low computational complexity for querying neighbors and its wide availability as standalone software in the public domain (e.g., matlab central has a few implementations for download, and the MATLAB Statistics Toolbox includes an efficient k-D Tree).

The complexity of assembling the tree is

The Matlab central k -D Tree is MEX compiled and efficient. We integrated the standalone C++ code into our library.

While the k -D Tree functions well for queries, its downfall is a large cost in preprocessing to build the tree. For moving nodes, such as in Lagrangian schemes, this cost is prohibitively high. In an attempt to reduce the cost, lagrangian schemes introduced approximate nearest neighbor queries based on

Approximate nearest neighbors will be nearly balanced. We observe that RBF-FD functions as well on stencils of true nearest neighbors as it does on approximate nearest neighbors.

CHAPTER 8

PARALLEL SOLVERS

Parallel solution of PDEs in a distributed computing environment requires three design decisions [?]. First, partitioning the domain and distributing work across compute nodes requires knowledge of *neighboring processors* with which communication will occur. Intelligent partitioning impacts load balancing on processors and the computation to communication ratio; imbalanced computation can cause excessive delay per iteration as processors wait to receive information. Second, one must decide what information each processor is able to access regarding node information, solution values, etc and establish index mappings that translate between a processor's local context and the global problem. In situations where processors are not aware of all nodes/solution values in the global domain, the index mappings are essential to maintaining solution consistency at each time-step. Last but not least, each processor can re-order nodes locally in an effort to improve solver efficiency and local system conditioning. Node ordering also allows us to minimize data transfer between CPU and GPU.

Parallelization of the RBF-FD method is achieved at two levels. First, the physical domain of the problem—in this case, the unit sphere—is partitioned into overlapping sub-domains, each handled by a different CPU process. All CPUs operate independently to compute/load RBF-FD stencil weights, run diagnostic tests and perform other initialization tasks. A CPU computes only weights corresponding to stencils centered in the interior of its partition. After initialization, CPUs continue concurrently to solve the PDE. Communication barriers ensure that the CPUs execute in lockstep to maintain consistent solution values in regions where partitions overlap. The second level of parallelization offloads time-stepping of the PDE to the GPU. Evaluation of the right hand side of Equation (6.18) is data-parallel: the solution derivative at each stencil center is evaluated independently of the other stencils. This maps well to the GPU, offering decent speedup even in unoptimized kernels. Although the stencil weight calculation is also data-parallel, we assume that in this context that the weights are precomputed and loaded once from disk during the initialization phase.

Author’s Note: Need partitioning, global and local index maps, and local ordering for a distributed matrix operation [?]

8.1 On the use of libraries for parallel solvers

Since our focus within this work is to lay the foundation for parallel computing with RBF-FD, we have made several simplifying assumptions in our code design. Libraries like PETsc, Hypre, Trilinos and Deal.ii distribute sparse matrix operations in similar fashion to our approach. When work initially began on this dissertation, none of these competing libraries contained support for the GPU. PETsc is currently developing support for the GPU, but we have not had the chance to consider it yet.

Our codebase began as a prototype demonstrating the feasibility of RBF-FD operating on the GPU. Starting from the perspective of RBF-FD as a Lagrangian method with stationary nodes, the code was developed as N independent dot products of weights and solution values to approximate derivatives. Weights were stored linearly in memory, with the solution values read randomly. On the GPU, stencils were evaluated independently by threads, or shared by a warp of threads. Operating on stencils in this way implied that GPU kernels were to be hand written, tuned and optimized.

Much later, our perspective evolved to see derivative approximation and time-stepping as sparse matrix operations. This opened new possibilities for optimization and allowed us to forego hand optimization and fine-tuning of GPU kernels. With all of the effort put into optimizing sparse operations within libraries like CUSP [?], ViennaCL [?] and even the Nvidia provided CUSPARSE [?], formulating the problem in terms of sparse matrix operations allows us to quickly prototype on the GPU and leverage all of the optimizations available within the third party libraries.

In the most recent version of ViennaCL a set of auto-tuning options were introduced. We have not investigated these options, but the idea is that they would only improve the efficiency of our code.

8.2 Partitioning

For ease of development and parallel debugging, partitioning is initially assumed to be linear within one physical direction (typically the x -direction). Figure 8.1 illustrates a partitioning of $N = 10,201$ nodes on the unit sphere onto four CPUs. Each partition, illustrated as a unique color, represents set \mathcal{G} for a single CPU. Alternating representations between node points and interpolated surfaces illustrates the overlap regions where nodes in sets \mathcal{O} and \mathcal{R} (i.e., nodes requiring MPI communication) reside. As stencil size increases, the width of the overlap regions relative to total number of nodes on the sphere also increases.

Author's Note: Q: what is the percentage overlap for n ? $\frac{1}{2}n^{\frac{1}{d}}$ gives depth into neighbor since n is uniformly sampled we expect a cube shape. (SHould be literature on this...no?)

The linear partitioning in Figure 8.1 was chosen for ease of implementation. Communication is limited for each processor to left and right neighbors only, which simplifies parallel debugging. This partitioning, however, does not guarantee properly balanced computational work-loads. **Author's Note:** Update w/ ParMETIS Other partitionings of the sphere exist but are not studied here because this paper's focus is neither on efficiency nor on selecting a partitioning strategy for maximum accuracy. Examples of alternative approaches include a cubed-sphere [?] or icosahedral geodesic grid [?], which can evenly balance the computational load across partitions. Other interesting partitionings can be generated with

software libraries such as the METIS [?] family of algorithms, capable of partitioning and reordering directed graphs produced by RBF-FD stencils.

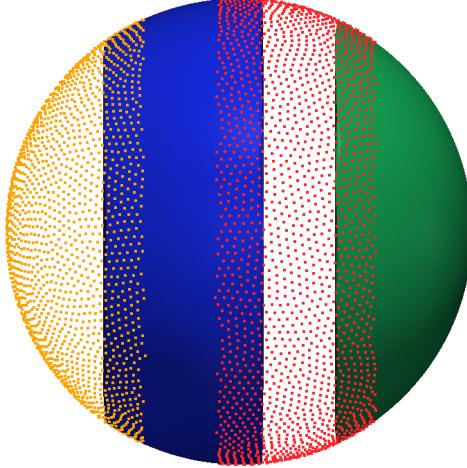


Figure 8.1: Partitioning of $N = 10,201$ nodes to span four processors with stencil size $n = 31$.

Author's Note: Need to flesh detail on ParMETIS, two dimensional partitioning and case when a node value is required by multiple processors

8.3 Local node ordering

After partitioning, each CPU/GPU is responsible for its own subset of nodes. To simplify accounting, we track nodes in two ways. Each node is assigned a global index, that uniquely identifies it. This index follows the node and its associated data as it is shuffled between processors. In addition, it is important to treat the nodes on each CPU/GPU in an identical manner. Implementations on the GPU are more efficient when node indices are sequential. Therefore, we also assign a local index for the nodes on a given CPU, which run from 1 to the maximum number of nodes on that CPU.

It is convenient to break up the nodes on a given CPU into various sets according to whether they are sent to other processors, are retrieved from other processors, are permanently on the processor, etc. Note as well, that each node has a home processor since the RBF nodes are partitioned into multiple domains without overlap. Table 8.1, defines the collection of index lists that each CPU must maintain for both multi-CPU and multi-GPU implementations.

Figure 8.2 illustrates a configuration with two CPUs and two GPUs, and 9 stencils, four on CPU1, and five on CPU2, separated by a vertical line in the figure. Each stencil has size $n = 5$. In the top part of the figures, the stencils are laid out with blue arrows pointing to stencil neighbors and creating the edges of a directed adjacency graph. Note that the connection between two nodes is not always bidirectional. For example, node 6 is in the stencil of node 3, but node 3 is not a member of the stencil of node 6. Gray arrows

\mathcal{G}	: all nodes received and contained on the CPU/GPU g
\mathcal{Q}	: stencil centers managed by g (equivalently, stencils computed by g)
\mathcal{B}	: stencil centers managed by g that require nodes from another CPU/GPU
\mathcal{O}	: nodes managed by g that are sent to other CPUs/GPUs
\mathcal{R}	: nodes required by g that are managed by another CPU/GPU

Table 8.1: Sets defined for stencil distribution to multiple CPUs

point to stencil neighbors outside the small window and are not relevant to the following discussion, which focuses only on data flow between CPU1 and CPU2. Since each CPU is responsible for the derivative evaluation and solution updates for any stencil center, it is clear that some nodes have a stencil with nodes that are on a different CPU. For example, node 8 on CPU1 has a stencil comprised of nodes 4,5,6,9, and itself. The data associated with node 6 must be retrieved from CPU2. Similarly, the data from node 5 must be sent to CPU2 to complete calculations at the center of node 6.

The set of all nodes that a CPU interacts with is denoted by \mathcal{G} , which includes not only the nodes stored on the CPU, but the nodes required from other CPUs to complete the calculations. The set $\mathcal{Q} \in \mathcal{G}$ contains the nodes at which the CPU will compute derivatives and apply solution updates. The set $\mathcal{R} = \mathcal{G} \setminus \mathcal{Q}$ is formed from the set of nodes whose values must be retrieved from another CPU. For each CPU, the set $\mathcal{O} \in \mathcal{Q}$ is sent to other CPUs. The set $\mathcal{B} \in \mathcal{Q}$ consists of nodes that depend on values from \mathcal{R} in order to evaluate derivatives. Note that \mathcal{O} and \mathcal{B} can overlap, but differ in size, since the directed adjacency graph produced by stencil edges is not necessarily symmetric. The set $\mathcal{B} \setminus \mathcal{O}$ represents nodes that depend on \mathcal{R} but are not sent to other CPUs, while $\mathcal{Q} \setminus \mathcal{B}$ are nodes that have no dependency on information from other CPUs. The middle section Figure 8.2 lists global node indices contained in \mathcal{G} for each CPU. Global indices are paired with local indices to indicate the node ordering internal to each CPU. The structure of set \mathcal{G} ,

$$\mathcal{G} = \{\mathcal{Q} \setminus \mathcal{B}, \mathcal{B} \setminus \mathcal{O}, \mathcal{O}, \mathcal{R}\}, \quad (8.1)$$

is designed to simplify both CPU-CPU and CPU-GPU memory transfers by grouping nodes of similar type. The color of the global and local indices in the figure indicate the sets to which they belong. They are as follows: white represents $\mathcal{Q} \setminus \mathcal{B}$, yellow represents $\mathcal{B} \setminus \mathcal{O}$, green indices represent \mathcal{O} , and red represent \mathcal{R} .

The structure of \mathcal{G} offers two benefits: first, solution values in \mathcal{R} and \mathcal{O} are contiguous in memory and can be copied to or from the GPU without the filtering and/or re-ordering normally required in preparation for efficient data transfers. Second, asynchronous communication allows for the overlap of communication and computation. This will be considered as part of future research on algorithm optimization. Distinguishing the set $\mathcal{B} \setminus \mathcal{O}$ allows the computation of $\mathcal{Q} \setminus \mathcal{B}$ while waiting on \mathcal{R} .

Author's Note: The local index set is ordered as QmB, BmO, O, R

Author's Note: Domain boundary nodes appear at beginning of the list

When targeting the GPU, communication of solution or intermediate values is a four step process:

1. Transfer \mathcal{O} from GPU to CPU

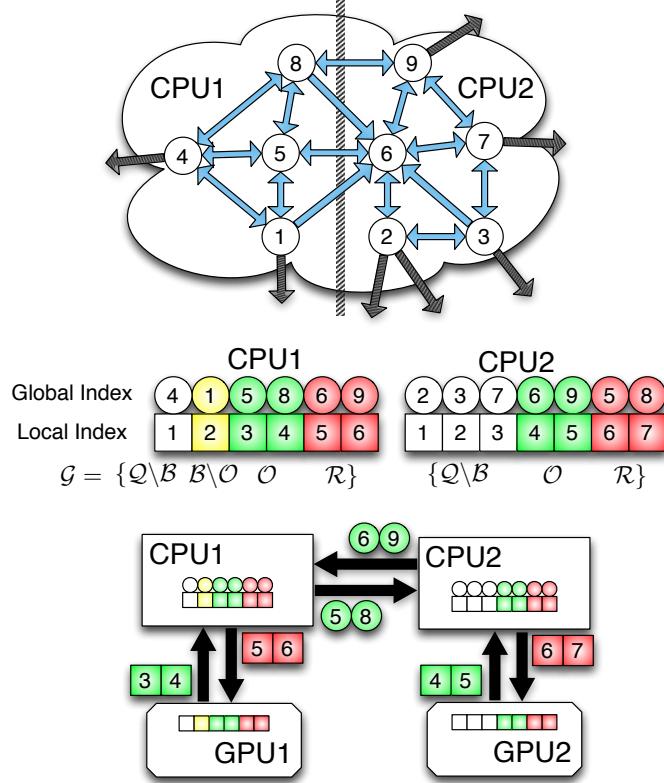


Figure 8.2: Partitioning, index mappings and memory transfers for nine stencils ($n = 5$) spanning two CPUs and two GPUs. Top: the directed graph created by stencil edges is partitioned for two CPUs. Middle: the partitioned stencil centers are reordered locally by each CPU to keep values sent to/received from other CPUs contiguous in memory. Bottom: to synchronize GPUs, CPUs must act as intermediaries for communication and global to local index translation. Middle and Bottom: color coding on indices indicates membership in sets from Table 8.1: $Q \setminus B$ is white, $B \setminus O$ is yellow, O is green and R is red.

2. Distribute O to other CPUs, receive R from other CPUs
3. Transfer R to the GPU
4. Launch a GPU kernel to operate on Q

The data transfers involved in this process are illustrated at the bottom of Figure 8.2. Each GPU operates on the local indices ordered according to Equation (8.1). The set O is copied off the GPU and into CPU memory as one contiguous memory block. The CPU then maps local to global indices and transfers O to other CPUs. CPUs send only the subset of node values from O that is required by the destination processors, but it is important to note that node information might be sent to several destinations. As the set R is received, the CPU converts back from global to local indices before copying a contiguous block of memory to the GPU.

This approach is scalable to a very large number of processors, since the individual processors do not require the full mapping between RBF nodes and CPUs.

8.4 Two level parallelism

Our current implementation assumes that we are computing on a cluster of CPUs, with one GPU attached to each CPU. The CPU maintains control of execution and launches kernels on the GPU that execute in parallel. Under the OpenCL standard [?], a tiered memory hierarchy is available on the GPU with *global device memory* as the primary and most abundant memory space. The memory space for GPU kernels is separate from the memory available to a CPU, so data must be explicitly copied to/from global device memory on the GPU.

- How to copy to/from GPU
- Where are the synchronization points (lockstep and overlapping)

Author's Note: [The basic primitives necessary to parallelize our problems: SpMV, AXPY and](#)

8.4.1 Overlapping Computation and Communication

Need to describe our approach to launch

In simplest form (& indicates an overlapped execution; overlapping is possible by non-blocking launches to the GPU before comm is started):

- $U'_{Q \setminus B} = A_{Q \setminus B} U_{Q \setminus B}$
- & MPI_alltoallv
- $U'_B = A_B U_B$

This form benefits because stencils are operated on in their entirety. For the GPU this is important because we maintain SIMD nature.

A more complex form which might overlap more comm and comp depending on the size of B (i.e., might be beneficial for large stencil sizes):

- $U'_{Q \setminus B} = A_{Q \setminus B} U_{Q \setminus B}$
- & MPI_alltoallv
- & $U'_B = A_{B \setminus R} U_{B \setminus R}$ (partial sparse vector product)
- $U'_B + = A_R U_R$

Note that there is no non-blocking version of MPI_alltoallv within the MPI standard, so the above two algorithms will not be overlapped when computing on the CPU. To overlap on the CPU, [?] refers to the non-blocking collective as MPI_ialltoallv. With a collective like this, we could enforce a pattern like:

- MPI_alltoallv
- & $U'_{Q \setminus B} = A_{Q \setminus B} U_{Q \setminus B}$
- & $U'_B = A_{B \setminus R} U_{B \setminus R}$ (partial sparse vector product)
- MPI_wait
- $U'_B += A_R U_R$

8.4.2 Explicit Solvers

Our implementation leverages the GPU for acceleration of the standard fourth order Runge-Kutta (RK4) scheme. Nodes are stationary, so stencil weights are calculated once at the beginning of the simulation, and reused in every iteration. To avoid the cost of calculating stencil weights each time a test case is run, they are written to disk and loaded from file on subsequent runs. There is one set of weights computed for each new grid. Ignoring code initialization, the cost of the algorithm is simply the explicit time advancement of the solution.

Figure 8.3 summarizes the time advancement steps for the multi-CPU/GPU implementation. The RK4 steps are:

$$\begin{aligned}\mathbf{k}_1 &= \Delta t f(t_n, \mathbf{u}_n) \\ \mathbf{k}_2 &= \Delta t f\left(t_n + \frac{1}{2}\Delta t, \mathbf{u}_n + \frac{1}{2}\mathbf{k}_1\right) \\ \mathbf{k}_3 &= \Delta t f\left(t_n + \frac{1}{2}\Delta t, \mathbf{u}_n + \frac{1}{2}\mathbf{k}_2\right) \\ \mathbf{k}_4 &= \Delta t f(t_n + \Delta t, \mathbf{u}_n + \mathbf{k}_3) \\ \mathbf{u}_{n+1} &= \mathbf{u}_n + \frac{1}{6}(\mathbf{k}_1 + 2\mathbf{k}_2 + 2\mathbf{k}_3 + \mathbf{k}_4),\end{aligned}$$

where each equation has a corresponding kernel launch. To handle a variety of Runge-Kutta implementations, steps $\mathbf{k}_{1 \rightarrow 4}$ correspond to calls to the same kernel with different arguments. The evaluation kernel returns two output vectors:

1. $\mathbf{k}_i = \Delta t f(t_n + \alpha_i \Delta t, \mathbf{u}_n + \alpha_i \mathbf{k}_{i-1})$, for steps $i = 1, 2, 3, 4$, and
2. $\mathbf{u}_n + \alpha_{i+1} \mathbf{k}_i$

We choose $\alpha_i = 0, \frac{1}{2}, \frac{1}{2}, 1, 0$ and $\mathbf{k}_0 = \mathbf{u}_n$. The second output for each $\mathbf{k}_{i=1,2,3}$ serves as input to the next evaluation, \mathbf{k}_{i+1} . In an effort to avoid an extra kernel launch—and corresponding memory loads—the SAXPY that produces the second output uses the same evaluation kernel. Both outputs are stored in global device memory. When the computation spans multiple GPUs, steps $\mathbf{k}_{1 \rightarrow 3}$ are each followed by a communication barrier to synchronize the subsets \mathcal{O} and \mathcal{R} of the second output (this includes copying the subsets between GPU and CPU). An additional synchronization occurs on the updated solution, \mathbf{u}_{n+1} , to ensure that all GPUs share a consistent view of the solution going into the next time-step.

To evaluate $\mathbf{k}_{1 \rightarrow 4}$, the discretized operators from Equation (6.18) are applied using sparse matrix-vector multiplication. If the operator D is composed of multiple derivatives,

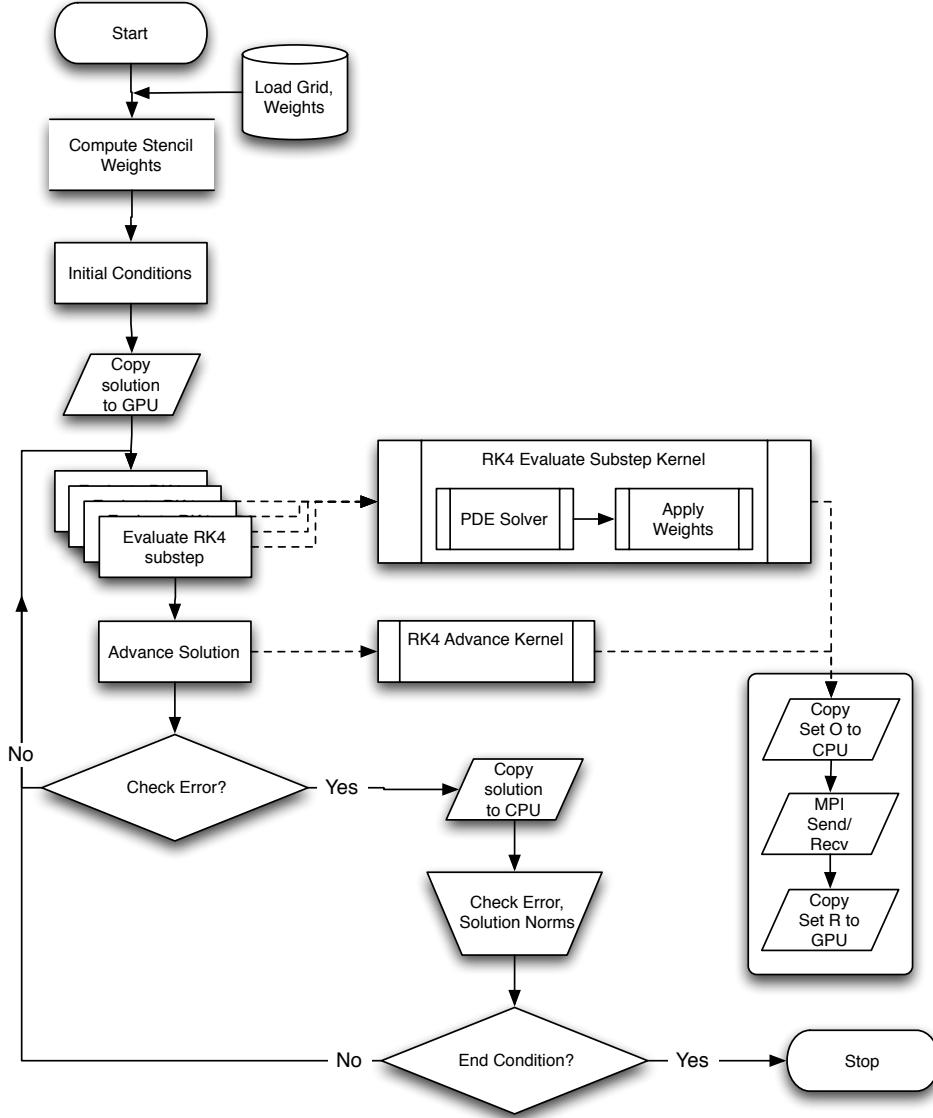


Figure 8.3: Workflow for RK4 on multiple GPUs.

a differentiation matrix for each derivative is applied independently, including an additional multiplication for the discretized H operator. On the GPU, the kernel parallelizes across rows of the DMs, so all derivatives for stencils are computed in one kernel call.

For the GPU, the OpenCL language [?] assumes a lowest common denominator of hardware capabilities to provide functional portability. For example, all target architectures are assumed to support some level of SIMD (Single Instruction Multiple Data) execution for kernels. Multiple *work-items* execute a kernel in parallel. A collection of work-items performing the same task is called a *work-group*. While a user might think of work-groups as executing all work-items simultaneously, the work-items are divided at the hardware level into one or more SIMD *warps*, which are executed by a single multiprocessor. On the family

of Fermi GPUs, a warp is 32 work-items [?]. OpenCL assumes a tiered memory hierarchy that provides fast but small *local memory* space that is shared within a work-group [?]. Local memory on Fermi GPUs is 48 KB per multiprocessor [?]. The *global device memory* allows sharing between work-groups and is the slowest but most abundant memory. In the GPU computing literature, the terms *thread* and *shared memory* are synonymous to *work-item* and *local memory* respectively, and are preferred below.

Although the primary focus of this paper is the implementation and verification of the RBF-FD method across multiple CPUs and GPUs, we have nonetheless tested two approaches to the computation of derivatives on the GPU to assess the potential for further improvements in performance. In both cases, the stencil weights are stored in CSR format [?], a packed one-dimensional array in global memory with all the weights of a single stencil in consecutive memory addresses. Each operator is stored as an independent CSR matrix. The consecutive ordering on the weights implies that the solution vector, structured according to the ordering of set \mathcal{G} is treated as random access.

All the computation on the GPU is performed in 8-byte double precision.

Naive Approach: One thread per stencil. In this first implementation, each thread computes the derivative at one stencil center (Figure 8.4). The advantage of this approach is trivial concurrency. Since each stencil has the same number of neighbors, each derivative has an identical number of computations. As long as the number of stencils is a multiple of the warp size, there are no idle threads. Should the total number of stencils be less than a multiple of the warp size, the final warp would contain idle threads, but the impact on efficiency would be minimal assuming the stencil size is sufficiently large.

Perfect concurrency from a logical point of view does not imply perfect efficiency in practice. Unfortunately, the naive approach is memory bound. When threads access weights in global memory, a full warp accesses a 128-byte segment in a single memory operation [?]. Since each thread handles a single stencil, the various threads in a warp access data in very disparate areas of global memory, rather than the same segment. This leads to very large slowdowns as extra memory operations are added for each 128-byte segment that the threads of a warp must access. However, with stencils sharing many common nodes, and the Fermi hardware providing caching, some weights in the unused portions of the segments might remain in cache long enough to hide the cost of so many additional memory loads.

Alternate Approach: One warp per stencil. An alternate approach, illustrated in Figure 8.5, dedicates a full warp of threads to a single stencil. Here, 32 threads load the weights of a stencil and the corresponding elements of the solution vector. As the 32 threads each perform a subset of the dot product, their intermediate sums are accumulated in 32 elements of shared memory (one per thread). Should a stencil be larger than the warp size, the warp iterates over the stencil in increments of the warp size until the full dot product is complete. Finally, the first thread of the warp performs a sum reduction across the 32 (warp size) intermediate sums stored in shared memory and writes the derivative value to global memory.

By operating on a warp by warp basis, weights for a single stencil are loaded with a reduced number of memory accesses. Memory loads for the solution vector remain random access but see some benefit when solution values for a stencil are in a small neighborhood in the memory space. Proximity in memory can be controlled by node indexing (see e.g.,

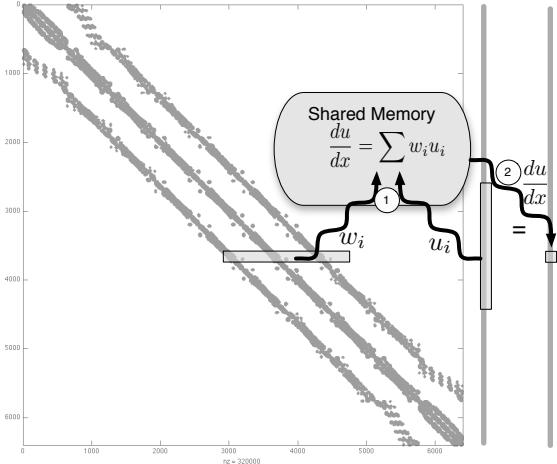


Figure 8.4: Naive approach to sparse matrix-vector multiply. Each thread is responsible for the sparse vector dot product of weights and solution values for derivatives at a single stencil.

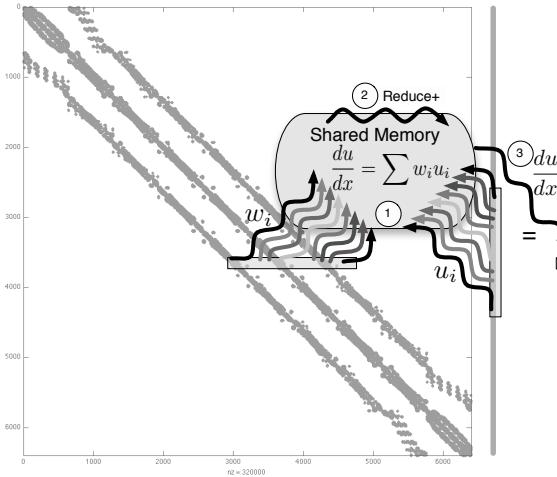


Figure 8.5: Alternative approach. A full warp (32 threads) collaborate to apply weights and compute the derivative at a stencil center.

[?] and [?]).

For stencil sizes smaller than 32, some threads in the warp always remain idle. Idle threads do not slow down the computation within a warp, but under-utilization of the GPU is not desirable. For small stencil sizes, caching on the Fermi can hide some of the cost of memory loads for the naive approach, with no idle threads, making it more efficient. The real strength of one warp per stencil is seen for large stencil sizes. As part of future work on optimization, we will consider a parallel reduction in shared memory, as well as assigning multiple stencils to a single warp for small n .

dirichlet: kernel copies dirichlet values into place [Author's Note: Need to dust off code](#)

8.4.3 Fragments (integrate above)

While the nomenclature used in this paper is typically associated with CUDA programming, the names *thread* and *warp* are used to clearly illustrate kernel execution in context of the NVidia specific hardware used in tests. OpenCL assumes a lowest common denominator of hardware capabilities to provide functional portability. However, intimate knowledge of hardware allows for better understanding of performance and optimization on a target architecture. For example, OpenCL assumes all target architectures are capable at some level of SIMD (Single Instruction Multiple Data) execution, but CUDA architectures allow for Single Instruction Multiple Thread (SIMT). SIMT is similar to traditional SIMD, but while SIMD immediately serializes on divergent operations, SIMT allows for a limited amount of divergence without serialization.

At the hardware level, a *thread* executes instructions on the GPU. On Fermi level GPUs, groups of 32 threads are referred to as *warps*. A warp is the unit of threads executed concurrently on a single *multi-processor*. In OpenCL (i.e., software), a collection of hardware threads performing the same instructions are referred to as a *work-group* of *work-items*. Work-groups execute as a collection of warps constrained to the same multiprocessor. Multiple work-groups of matching dimension are grouped into an *NDRRange*. The *kernel* provides a master set of instructions for all threads in an NDRRange [?].

NVidia GPUs have a tiered memory hierarchy related to the grouping of threads described above. In multiprocessors, each computing core executes a thread with a limited set of registers. The number of registers varies with the generation of hardware, but always come in small quantities (e.g., 32K shared by all threads of a multiprocessor on the Fermi). Accessing registers is free, but keeping data in registers requires an effort to maintain balance between kernel complexity and the number of threads per block. Threads of a single work-group can share information within a multiprocessor through *shared memory*. With only 48 KB available per multiprocessor [?], shared memory is another fast but limited resource on the GPU. OpenCL refers to shared memory as *local memory*. Sharing information across multiprocessors is possible in *global device memory*—the largest and slowest memory space on the GPU. To improve seek times into global memory, Fermi level architectures include L1 on each multiprocessor and a shared L2 cache for all multiprocessors.

8.4.4 Implicit Solvers

Perhaps this section should move up. I think it might be best to discuss GMRES and iterative methods before Distributed Solvers (that way we can say the general solution form is “matrix form”. but in parallel we need to use distributed algorithms.

8.5 RBF-FD as SpMV

From the definition of RBF-FD we can formulate the problem computationally in two ways. First, stencil operations are independent. Therefore, we can write kernels with perfect parallelism by dedicating a single thread per stencil or a group of threads per stencil.

Unfortunately, perfect concurrency does not imply perfect or even ideal concurrency on the GPU.

We first demonstrate the case where one thread is dedicated to each stencil. This is followed by dedicating a group of thread to the stencil. In each case we are operating under the assumption that each stencil is independent on the GPU.

To further optimize RBF-FD on the GPU, we formulate the problem in terms of a Sparse Matrix-Vector Mulitply (SpMV). When we consider the problem in this light we generate a single Differentiation Matrix that can see two optimizations not possible with our stencil-based view:

- First, the sparse containers used in SpMV allow for their own unique optimizations to compress storage and leverage hardware cache.
- Evaluation of multiple derivatives can be accumulated by association into one matrix operation. This reduces the total number of floating point operations required per iteration.

We compare the performance of our custom kernel to ViennaCL kernels (ELL, CSR, COO, HYB, DIAG), UBlas (COO, CSR) and Eigen (COO, CSR, ELL)

CHAPTER 9

SPARSE MATRIX MULTIPLICATION ON THE GPU

With the generalization of RBF-FD derivative computation formulated as a sparse matrix multiplication, we can consider the various sparse formats provided by CUSP and ViennaCL.

Consider the performance of the problem given

- All stencils with uniform size
- All stencils with non-uniform size

What is the optimal choice of sparse container? How do the sparse containers compare in performance to each other, and to our custom kernels? What can we conclude?

Compare formats:

- JAD
- ELL
- HYB
- COO
- CSR
- DIA

How is communication overlap handled with each format?

Conclude: sparse containers allow increased efficiency compared to our custom kernels.
The custom kernels compete with CSR and COO.

CHAPTER 10

GMRES

The GMRES algorithm follows Algorithm ??.

The Arnoldi process can be completed using in a variety of ways. Some libraries like CUSP and [?] prefer the straightforward Givens rotations because they are easily parallelizable. The givens algorithm is provided in Algorithm ??

Others like [...] prefer to use an alternate algorithm. This algorithm is demonstrated in Algorithm ??.

Parallelizing the algorithm is possible with communication points listed in red (*).

CHAPTER 11

NUMERICAL VALIDATION

11.1 Choosing ϵ

Present our attempts to get the right support parameter

We duplicate some results of [?] by choosing the same epsilon values specified in Table 11.1.

I also produced a Matlab script to generate the contours for any stencil size and produced the following figures.

11.1.1 CVT epsilon tests

[?] consider epsilon as a function of the number of nodes on the sphere. However, this assumption requires nodes to be distributed as MD nodes. When we apply the same functions to CVTs of larger problems we get varying results. The subtle perturbations in nodes

test on icosahedral grid

function based on min enclosing circle instead. Will be finer tuned to the grid. if nodes are uniform in stencil and stencil size locked then the only things that can influence epsilon and conditioning is the enclosing circle radius.

11.2 Parabolic

Include figure tracking max temperature and min temperature. If we have zero flux conditions then they should average out. If we have Dirichlet BCs the temperature will converge to 0.

11.3 Hyperbolic

Here, we present the first results in the literature for parallelizing RBF-FDs on multi-CPU and multi-GPU architectures for solving PDEs. To verify our multi-CPU, single GPU and multi-GPU implementations, two hyperbolic PDEs on the surface of the sphere are tested: 1) vortex roll-up [?, ?] and 2) solid body rotation [?]. These tests were chosen since they are not only standard in the numerical literature, but also for the development

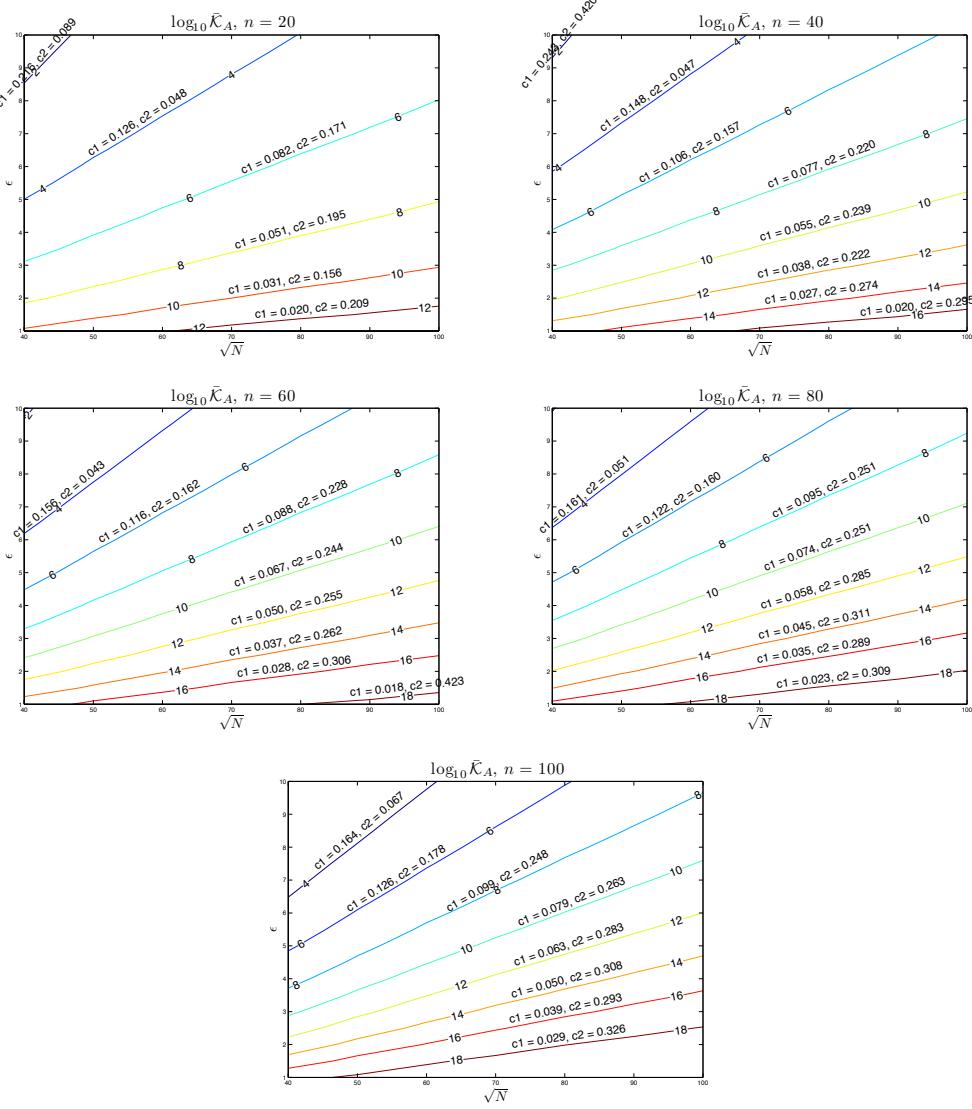


Figure 11.1: Contours for choosing ϵ for $n = 20, 40, 60, 80$ and 100 on the unit sphere as a function of \sqrt{N} . Contours assume near uniform distribution of nodes (e.g., maximum determinant nodes, icosahedral grids and spherical centroidal voronoi tessellations). Superimposed above each contour are parameters for the linear regression of the line, $c_1\sqrt{N} - c_2$.

of RBFs in solving PDEs on the sphere [?, ?, ?, ?]. Although any ‘approximately evenly’ distributed nodes on the sphere would suffice for our purposes, maximum determinant (MD) node distributions on the sphere are used (see [?] for details) in order to be consistent with previously published results (see e.g., [?] and [?]). Node sets from 1024 to 27,556 are considered with stencil sizes ranging from 17 to 101.

All results in this section are produced by the single-GPU implementation. Multi-CPU and multi-GPU implementations are verified to produce these same results. Synchronization of the solution at each time-step and the use of double precision on both the CPU and

GPU ensure consistent results regardless of the number and/or choice of CPU vs GPU. Eigenvalues are computed on the CPU by the Armadillo library [?].

11.3.1 Vortex Rollup

The first test case demonstrates vortex roll-up of a fluid on the surface of a unit sphere. An angular velocity field causes the initial condition to spin into two diametrically opposed but stationary vortices.

The governing PDE in latitude-longitude coordinates, (θ, λ) , is

$$\frac{\partial h}{\partial t} + \frac{u}{\cos \theta} \frac{\partial h}{\partial \lambda} = 0 \quad (11.1)$$

where the velocity field, u , only depends on latitude and is given by

$$u = \omega(\theta) \cos \theta.$$

Note that the $\cos \theta$ in u and $1/\cos \theta$ in (11.1) cancel in the analytic formulation, so the discrete operator approximates $\omega(\theta) \frac{\partial}{\partial \lambda}$.

Here, $\omega(\theta)$ is the angular velocity component given by

$$\omega(\theta) = \begin{cases} \frac{3\sqrt{3}}{2\rho(\theta)} \operatorname{sech}^2(\rho(\theta)) \tanh(\rho(\theta)) & \rho(\theta) \neq 0 \\ 0 & \rho(\theta) = 0 \end{cases}$$

where $\rho(\theta) = \rho_0 \cos \theta$ is the radial distance of the vortex with $\rho_0 = 3$. The exact solution to (11.1) at non-dimensional time t is

$$h(\lambda, \theta, t) = 1 - \tanh\left(\frac{\rho(\theta)}{\gamma} \sin(\lambda - \omega(\theta)t)\right),$$

where γ defines the width of the frontal zone.

From a method of lines approach, the discretized version of (11.1) is

$$\frac{d\mathbf{h}}{dt} = -\operatorname{diag}(\omega(\theta)) D_\lambda \mathbf{h}. \quad (11.2)$$

where D_λ is the DM containing the RBF-FD weights that approximate $\frac{\partial}{\partial \lambda}$ at each node on the sphere.

For stability, hyperviscosity is added to the right hand side of (11.2) in the form given in (6.18). The scaling parameter γ_c and the order of hyperviscosity k are given in Table 11.1. The goal when choosing k is to damp the higher spurious eigenmodes of $\operatorname{diag}(\omega(\theta)) D_\lambda$ while leaving the lower physical modes that can be resolved by the stencil intact. In this process, the eigenvalues will be pushed into the left half of the complex plane. Then, γ_c is used to condense the eigenvalues as near to the imaginary axis as possible. Figure 11.2b shows the effect of hyperviscosity on the eigenvalues of the DM, $-\operatorname{diag}(\omega(\theta)) D_\lambda$, in (11.2).

In order to scale to large node sets, the RBF shape parameter, ϵ , is chosen such that the mean condition number of the local RBF interpolation matrices $\bar{\kappa}_A = \frac{1}{N} \sum_{j=1}^N (\kappa_A)_j$ is kept constant as N increases ((κ_A) $_j$ is the condition number of the interpolation matrix in (?),

Table 11.1: Values for hyperviscosity and the RBF shape parameter ϵ for vortex roll-up test.

Stencil Size (n)	$\epsilon = c_1\sqrt{N} - c_2$		$H = -\gamma_c N^{-k} \Delta^k$	
	c_1	c_2	k	γ_c
17	0.026	0.08	2	8
31	0.035	0.1	4	800
50	0.044	0.14	4	145
101	0.058	0.16	4	40

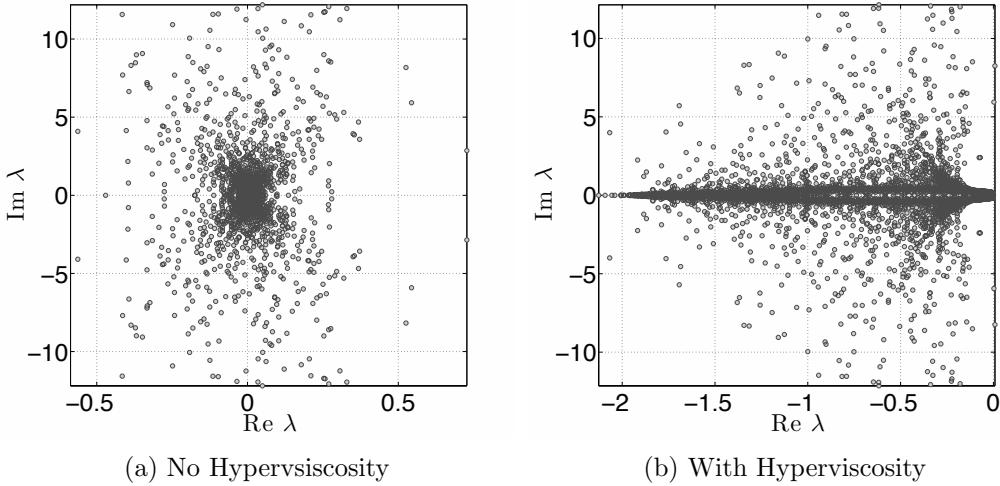


Figure 11.2: Eigenvalues of $\text{diag}(\omega(\theta))D_\lambda$ for the vortex roll-up test case for $N = 4096$ nodes, stencil size $n = 101$ and $\epsilon = 3.5$. Left: no hyperviscosity. Right: hyperviscosity enabled with $k = 4$ and $\gamma_c = 40$.

representing the j^{th} stencil). For a constant mean condition number, ϵ varies linearly with \sqrt{N} (see [?] Figure 4a and b). This is not surprising since the condition number strongly depends on the quantity ϵr , where $r \sim 1/\sqrt{N}$ on the sphere. Thus, to obtain a constant condition number, we let $\epsilon(N) = c_1\sqrt{N} - c_2$, where c_1 and c_2 are constants based on [?].

Figure 11.3 shows the solution to Equation (11.1) at $t = 10$, on $N = 10201$ nodes, with stencil size $n = 50$. This resolution is sufficient to properly capture the vortices at $t = 10$, but lower resolutions would suffer approximation errors associated with insufficient grid resolution. For this reason, the solution at $t = 3$ is considered in the normalized ℓ_2 error convergence study presented in Figure 11.4. The time step $\Delta t = 0.05$ for all resolutions.

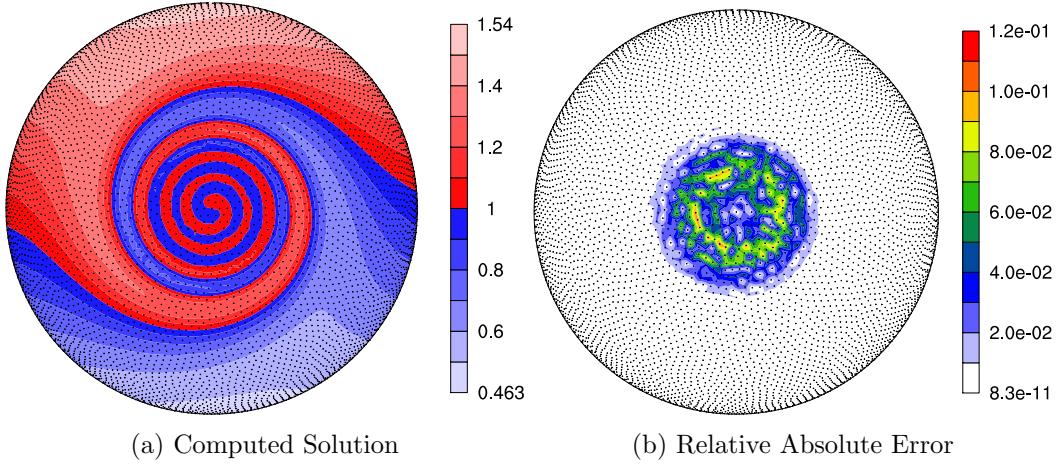


Figure 11.3: Vortex roll-up solution at time $t = 10$ using RBF-FD with $N = 10, 201$ and $n = 50$ point stencil. Normalized ℓ_2 error of solution at $t = 10$ is $1.25(10^{-2})$

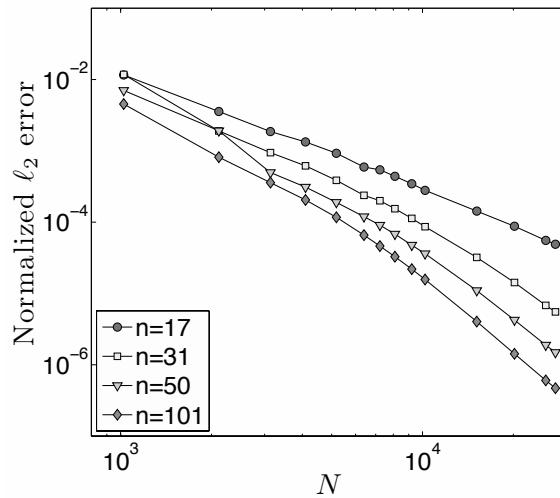


Figure 11.4: Convergence plot for vortex roll-up at $t = 3$.

11.3.2 Solid body rotation

The second test case simulates the advection of a cosine bell over the surface of a unit sphere at an angle α relative to the pole of a standard latitude-longitude grid. The governing PDE is

$$\frac{\partial h}{\partial t} + \frac{u}{\cos \theta} \frac{\partial h}{\partial \lambda} + v \frac{\partial h}{\partial \theta} = 0, \quad (11.3)$$

with velocity field,

$$\begin{cases} u = u_0(\cos \theta \cos \alpha + \sin \theta \cos \lambda \sin \alpha), \\ v = -u_0(\sin \lambda \sin \alpha) \end{cases} .$$

inclined at an angle α relative to the polar axis and velocity $u_0 = 2\pi/(1036800 \text{ seconds})$ to require 12 days per revolution of the bell as in [?, ?].

The discretized form of (11.3) is

$$\frac{d\mathbf{h}}{dt} = -\text{diag}\left(\frac{u}{\cos \theta}\right) D_\lambda \mathbf{h} - \text{diag}(v) D_\theta \mathbf{h} \quad (11.4)$$

where DMs D_λ and D_θ contain RBF-FD weights corresponding to all N stencils that approximate $\frac{\partial}{\partial \lambda}$ and $\frac{\partial}{\partial \theta}$ respectively. Rather than merge the differentiation matrices in (11.4) into one operator, our implementation evaluates them as two sparse matrix-vector multiplies. The separate matrix-vector multiplies are motivated by an effort to provide general and reusable GPU kernels. Additionally, they artificially increase the amount of computation compared to the vortex roll-up test case to simulate cases when operators cannot be merged into one DM (e.g., a non-linear PDE).

By splitting the DM, the singularities at the poles ($1/\cos \theta \rightarrow \infty$ as $\theta \rightarrow \pm \frac{\pi}{2}$) in (11.3) remain. However, in this case, the approach functions without amplification of errors because the MD node sets have nodes near, but not on, the poles. As noted in [?, ?], applying the entire spatial operator to the right hand side of Equation ?? generates a single DM that analytically removes the singularities at poles.

We will advect a C^1 cosine bell height-field given by

$$h = \begin{cases} \frac{h_0}{2}(1 + \cos(\frac{\pi\rho}{R})) & \rho \leq R \\ 0 & \rho \geq R \end{cases}$$

having a maximum height of $h_0 = 1$, a radius $R = \frac{1}{3}$ and centered at $(\lambda_c, \theta_c) = (\frac{3\pi}{2}, 0)$, with $\rho = \arccos(\sin \theta_c \sin \theta + \cos \theta_c \cos \theta \cos(\lambda - \lambda_c))$. The angle of rotation, $\alpha = \frac{\pi}{2}$, is chosen to transport the bell over the poles of the coordinate system.

Table 11.2: Values for hyperviscosity and RBF shape parameter for the cosine bell test.

	$\epsilon = c_1 \sqrt{N} - c_2$	$H = -\gamma_c N^{-k} \Delta^k$		
Stencil Size (n)	c_1	c_2	k	γ_c
17	0.026	0.08	2	$8 * 10^{-4}$
31	0.035	0.1	4	$5 * 10^{-2}$
50	0.044	0.14	6	$5 * 10^{-1}$
101	0.058	0.16	8	$5 * 10^{-2}$

Figure 11.5 compares eigenvalues of the DM for $N = 4096$ nodes and stencil size $n = 101$ before and after hyperviscosity is applied. To avoid scaling effects of velocity on the eigenvalues, they have been scaled by $1/u_0$. The same approach as in the vortex roll-up case is used to determine the parameters for hyperviscosity and ϵ . Our tuned parameters are presented in Table 11.2.

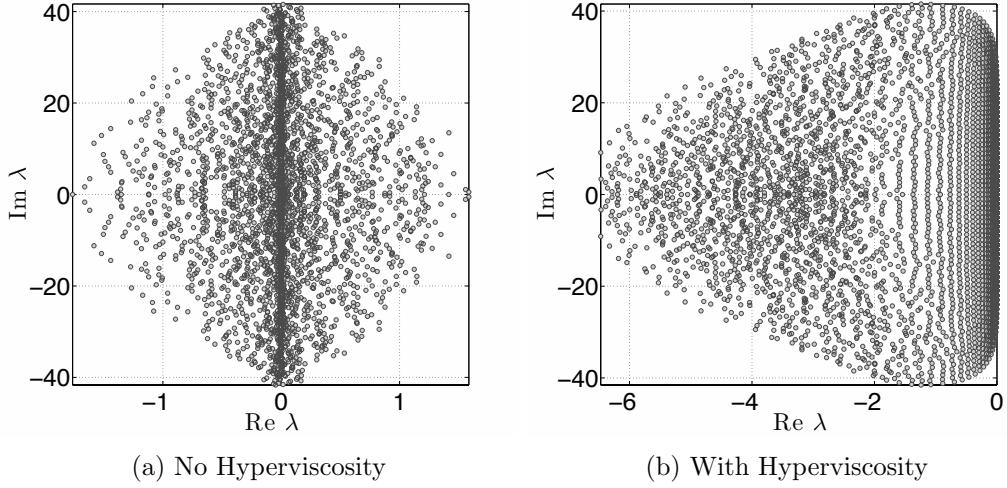


Figure 11.5: Eigenvalues of (11.4) for the cosine bell test case with $N = 4096$ nodes, stencil size $n = 101$, and $\epsilon = 3.5$. Left: no hyperviscosity. Right: hyperviscosity enabled with $k = 8$ and $\gamma_c = 5 * 10^{-2}$. Eigenvalues are divided by u_0 to remove scaling effects of velocity.

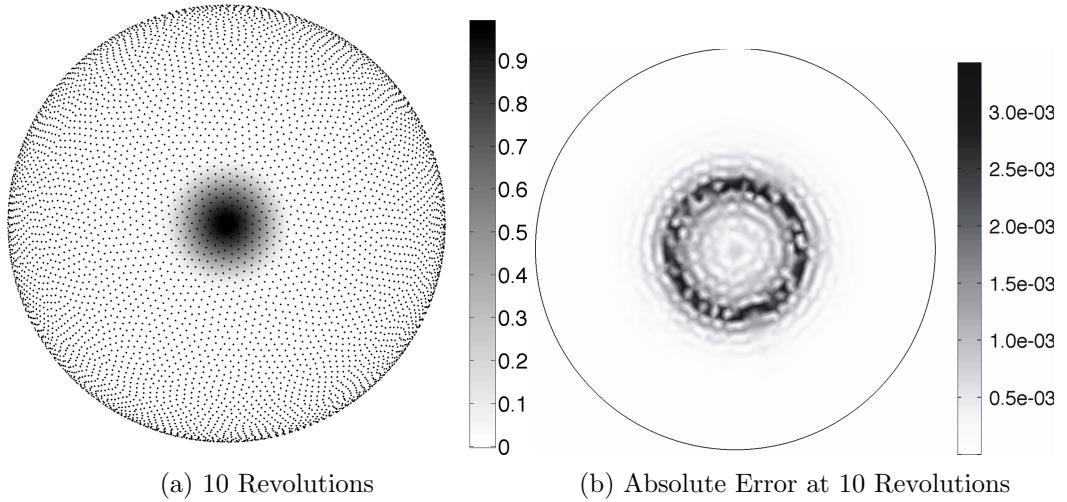


Figure 11.6: Cosine bell solution after 10 revolutions with $N = 10201$ nodes and stencil size $n = 101$. Hyperviscosity parameters are $k = 8$, $\gamma_c = 5(10^{-2})$.

Figure 11.6 shows the cosine bell transported ten full revolutions around the sphere. Without hyperviscosity, RBF-FD cannot complete a single revolution of the bell before instability takes over. However, adding hyperviscosity allows computation to extend to dozens or even thousands of revolutions and maintain stability (e.g., see [?]). After ten

revolutions, the cosine bell is still intact. The majority of the absolute error (Figure 11.6b) appears at the base of the C^1 bell where the discontinuity appears in the derivative. At ten revolutions, Figure 11.7 illustrates the convergence of the RBF-FD method. All tests in Figure 11.7 assume 1000 time-steps per revolution (i.e., $\Delta t = 1036.8$ seconds).

Complete: 17.28 minutes was a conservative step that allowed the problem to scale up to $N = 27556$ nodes. Compare this to the conservative 30 minute time-step taken for $N = 4096$ nodes in [?], which was already half necessary for DG and 8x less than necessary for both Spherical Harmonics and Double Fourier methods. **Author's Note:** It would be good to quantify the appropriate dt that would compare RBF-FD to their $N = 4096$ case with global RBFs.

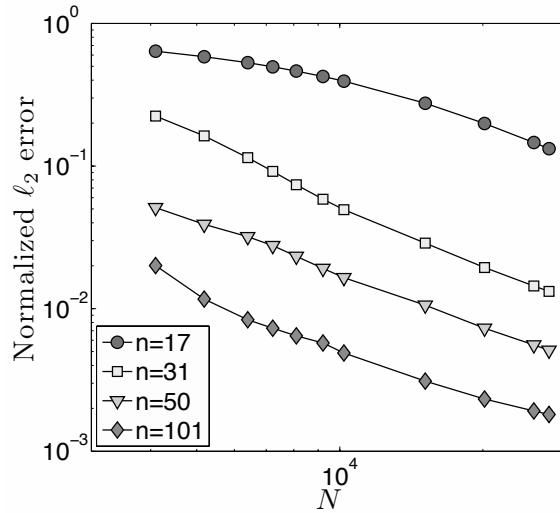


Figure 11.7: Convergence plot for cosine bell advection. Normalized ℓ_2 error at 10 revolutions with hyperviscosity enabled.

11.4 Fragments (integrate above)

To verify our multi-CPU and multi-CPU+GPU implementations, two hyperbolic PDEs on the surface of the sphere are tested. For both cases a spherical coordinate system is used in terms of latitude λ and longitude θ :

$$\begin{aligned} x &= \rho \cos \lambda \cos \theta, \\ y &= \rho \sin \lambda \cos \theta, \\ z &= \rho \sin \theta. \end{aligned}$$

Node sets are the Maximum Determinant point distributions on the sphere [?] consistent with previously published results (see e.g., [?] and [?]).

Hyperviscosity parameters γ_c and k depend on the RHS of the PDE. For this test case, hyperviscosity scaling parameters are listed in Table 11.1. Linear functions to choose the

RBF support parameter ϵ are also provided. The parameters γ_c and k were obtained via trial-and-error parameter searching on $N = 4096$ nodes. The goal when choosing parameters is to push all eigenvalues to the left half-plane, and then tweak γ_c up or down to condense the eigenvalues as near to the imaginary axis as possible. We try to keep the range of filtered eigenvalue real parts within twice the width of the unfiltered range, so hyperviscosity does not cause too much diffusion in the solution.

For the cosine bell we use the initial conditions

$$h = \begin{cases} \frac{h_0}{2}(1 + \cos(\frac{\pi\rho}{R})) & \rho \leq R \\ 0 & \rho \geq R \end{cases}$$

where the bell of radius $R = \frac{a}{3}$ is centered at (λ_c, θ_c) and provided by the expression,

$$\rho = a \arccos(\sin \theta_c \sin \theta + \cos \theta_c \cos \theta \cos(\lambda - \lambda_c)).$$

We assume $a = 1$, $h_0 = 1$, and $(\lambda_c, \theta_c) = (\frac{3\pi}{2}, 0)$. The angle $\alpha = \frac{\pi}{2}$ is chosen to transport the bell over the poles of the coordinate system, and $u_0 = \frac{2\pi a}{1036800}$.

[Author's Note: State that we can split operator to test cases like nonlinear PDEs.](#)

[Author's Note: Include figures from NCL or Paraview](#)

11.4.1 CFL

We constrict our timesteps according to the Courant-Friedrich-Lowy (CFL) condition:

$$C_{\max} \frac{\Delta x_{\min}}{v_{\max}} < \Delta t$$

where Δx_{\min} is the minimum distance between any two nodes in the domain, and the v_{\max} is the maximum velocity

For the cosine bell test cases we use a conservative $C_{\max} = 0.4$ to ensure stable transport in all cases with $n = 101$. However, in testing it was found that $n = 17$ is capable of stably advecting with $C_{\max} > 1$ for $n = 17$; $n = 101$ can go up to $C_{\max} = 0.51$ for $N = 27556$ (e.g. 650 timesteps per revolution).

apparently, canceling the cosine analytically causes the conditioning of the system to change slightly. The hyperviscosity parameters I have in the paper are for the case with the cosine present. The parameters continue to function well for the other cases, but their impact on the eigenvalue distributions is noticeably higher (further span to the left).

11.5 GFLOP Throughput

In order to quantify the performance of our implementation, we can measure two factors. First, we can check the speedup achieved on the GPU relative to the CPU to get an idea of how much return of investment is to be expected by all the effort in porting the application to the GPU. Speedup is measured as the time to execute on the CPU divided by the time to execute on the GPU.

The second quantification is to check the throughput of the process. By quantifying the GFLOP throughput we have a measure that tells us two things: first, a concrete number

quantifying the amount of work performed per second by either hardware, and second because we can calculate the peak throughput possible on each hardware, we also have a measure of how occupied our CPU/GPU units are. With the GFLOPs we can also determine the cost per watt for computation and conclude on what problem sizes the GPU is cost effective to target and use.

Now, as we parallelize across multiple GPUs, these same numbers can come into play. However we are also interested in the efficiency. Efficiency is the speedup divided by the number of processors. With efficiency we have a measure of how well-utilized processors are as we scale either the problem size (weak) or the number of processors (strong). As the efficiency diminishes we can conclude on how many stencils/nodes per processor will keep our processors occupied balanced with the shortest compute time possible (i.e., we are maximizing return of investment).

11.6 Elliptic

CHAPTER 12

STOKES

12.1 Introduction

We consider herein the solution to steady-state viscous Stokes flow on the surface of a sphere, governed by: [Author's Note: Where did we go wrong? Our vector laplacian does not have a curl component included. How do we derive that from here?](#)

$$\nabla \cdot [\eta(\nabla \mathbf{u} + (\nabla \mathbf{u})^T)] + RaT\hat{r} = \nabla p \quad (12.1)$$

$$\nabla \cdot \mathbf{u} = 0 \quad (12.2)$$

where the unknowns \mathbf{u} and p represent the vector velocity- and scalar pressure-field respectively, η is the viscosity tensor, Ra is the non-dimensional Rayleigh number, and T is an initial temperature profile. Many practical applications in sciences such as geophysics, climate modeling, and computational fluid dynamics must solve variations of the Navier-Stokes equations. The focus of this paper is on the implicit solve component for viscous (Stokes) flow, which amounts to the steady-state problem described by Equations 12.1 and 12.2.

This article introduces the first (to our knowledge) parallel approach to solve the steady-state equations on the surface of the unit sphere with the Radial Basis Function-generated Finite Differences (RBF-FD) method. Building on our work in [?], which parallelized explicit RBF-FD advection, our goal is to integrate both explicit and implicit components within a larger transient flow model.

[Author's Note: follow with details of RBF methods and novelty of RBF-FD.](#)

For decades, the demand for fast and accurate numerical solutions in fluid flow has lead to a plethora of computational methods for various geometries, discretizations and dimensions. On the sphere in \mathbb{R}^3 popular discretizations include the standard latitude-longitude grid, cubed-sphere [?], yin-yang overlapping grid [?], icosahedral grid [?] and centroidal voronoi tessellations [?]. Associate with each discretization is a mesh—specific to the choice of numerical method—that indicates connectivity of nodes for differentiation.

Two decades ago [?], meshless methods based on Radial Basis Functions (RBFs) were introduced for problems that require geometric flexibility with scattered node layouts in d -dimensional space, plus natural extensions to higher dimensions with minimal change in programming complexity [?, ?]. These RBF methods tout competitive accuracy with other

state-of-the-art and high-order methods [?, ?, ?, ?, ?], as well as stability for large time steps. A survey of RBF methods is provided by [?].

Author's Note: [Change first sentence](#) RBF-generated finite differences (RBF-FD) hold a promising future in that they share many advantages of global RBF methods, but reduce computational complexity to $O(N)$ and exhibit increased parallelism. The method was first suggested in 2000 [?], but made its grand debut a few years later in the simultaneous, yet independent, efforts of [?, ?, ?, ?]. It has been successfully employed for a variety of problems including Hamilton-Jacobi equations [?], convection-diffusion problems [?, ?], incompressible Navier-Stokes equations [?, ?], transport on the sphere [?], and the shallow water equations [?].

Compared to classical finite differences (FD) which calculate differentiation weights with one-dimensional polynomials, RBF-FD leverages d -dimensional RBFs as test functions. This allows for generalization to d -dimensional space on completely scattered node layouts. For both methods, a stencil of size n neighboring nodes determines the accuracy of the approximation. However, contrary to the regular stencil node distributions from FD, RBF-FD allows for completely scattered node distributions. In comparison with global RBF methods, spectral accuracy is sacrificed in exchange for computational speed and parallelism. Still, high-order accuracy is possible with RBF-FD—6th- to 10th-order accuracy is common.

Until now, most of the focus in RBF-FD has been on explicit methods. However, many practical applications in sciences such as geophysics, climate modeling, and computational fluid dynamics must solve variations of the navier stokes equations which include an implicit solve component. This paper develops multi-GPU algorithms for implicit RBF-FD systems toward the goal of integration within transient flow problems. The explicit component of transient flow is a natural extension of our work in [?].

Author's Note: [Flesh these points out and integrate them in surrounding paragraphs \(until "end"\)](#) Speed not the issue. Need less RBFs for given accuracy of steady state less nodes implies less memory general geometries are supported better distributions of nodes on spheres. competition like CitComS, etc. use cubed sphere, yinyang grids and triangular meshes in combination with low order methods (2nd and 3rd order). Increasing the order of the method or dimension can significantly increase the complexity of the algorithms. RBF-FD naturally extends to higher dimensions and increasing the order is as simple as increasing the number of nodes in the stencil. **Author's Note:** [end](#)

Related work ([list references](#)): RBF methods for Elliptic PDEs

- Global [?]
- Compact Support
- divergence-free spherical radial basis Glerkin method for Stokes on the unit sphere [?]
- RBF-FD
 - Incompressible Navier-Stokes using explicit (Euler) and semi-implicit (Crank Nicolson) time step. Small problem size (61×61) and small stencil sizes ($n = 9$); ghost nodes beyond boundary strategy. Both RBF-FD and RBF-HFD tested. [?]

Related work on Preconditioned iterative methods for Stokes Flow

- Survey of preconditioners used for Stokes flow problems [?] (limited applicability since they do not assume non-SPD matrices)
- Multi-GPU Jacobi iteration for Navier stokes flow in cavity [http://scholarworks.
boisestate.edu/cgi/viewcontent.cgi?article=1003&context=mecheng_facpubs](http://scholarworks.boisestate.edu/cgi/viewcontent.cgi?article=1003&context=mecheng_facpubs)

For decades, a major push has been executed in science to parallelize algorithms and leverage resources available on the increasingly capable supercomputers and clusters. Perhaps as soon as the next decade, we will see exascale level architectures ([?]). Given today's technology, those architecture will surely achieve their performance with the help of accelerator hardware such as GPUs.

To prepare the RBF community for the future, we develop algorithms employing two levels of parallelization: first the MPI standard spans computation across multiple CPUs, and second computation is distributed across the many processing cores of GPU accelerators.

The two level parallelization can even be extended to three level parallelism with pThreads or OpenMP [?]. While OpenCL provides the means to target parallelism on either multi-core CPUs or many-core GPUs, it does not allow a parallel kernel on one hardware interact with a parallel kernel on the another. That is to say, an OpenCL kernel on the CPU cannot launch kernels on the GPU.

Our goal is to demonstrate to the geosciences that RBF-FD can function well on both hyperbolic and elliptic problems. In [?] we introduced a multi-GPU implementation of RBF-FD and demonstrated the method's strong ability to stably advect solid bodies on the sphere. In this paper we continue toward the goal of RBF-FD solutions for fluid flow problems with a multi-GPU Poisson solver for steady-state Stokes flow. In this context, speed is not a paramount issue.

Related work on RBF methods targeting the GPU is quite limited. Schmidt et al. [?] solve an implicit system for a global RBF method using Accelerys Jacket in Matlab. Our work in [?] introduced the first parallel implementation of RBF-FD for explicit advection capable of spanning multiple CPUs as well as multiple GPUs.

Related work on multi-CPU or multi-GPU RBFs

- CPU [?] [?]
- single-GPU [?]
- multi-GPU
 - Preconditioned BiCGStab for Navier Stokes, Finite Element method [?]

While RBF-FD differentiation matrices are applied in the same fashion as standard FD methods, they are unique in that they are asymmetric, non-positive definite and potentially have high condition numbers. To solve an implicit system therefore, we require an iterative krylov solver like GMRES or BiCGStab which are applicable to matrices of this type. Additionally, preconditioned variants of these methods are required to reduce the complexity of the solution process.

Within this paper we implement a preconditioned GMRES method for RBF-FD on multiple GPUs.

Parallel GMRES

- CPU only: PETSc [?], Hypre [?]
- Parallel GMRES on single GPU available in ViennaCL [?] and CUSP [?]
- Parallel GMRES on Multiple GPUs [?]
- Reduced Communication with increased computation [?]

This article continues our effort with an implementation of RBF-FD on both single and multiple-GPUs for elliptic PDEs. In the next section we introduce

12.2 Bad Problem

Our initial derivation of the Stokes problem was incorrect. What we thought was the proper identity reduced the problem to simple scalar velocity laplacians and pressure gradients. The correct formulation of Stokes flow in 3D would have a curl component connecting each dimension.

Rather than start over and reformulate our problem, we opted to continue development of our GPU-based solver with the recognition that the problem we posed is actually a coupled Poisson problem and can be solved using the same iterative solver. By implementing and testing the solver and preconditioners for one problem, we prepare for the other.

12.2.1 Details

Items to test for our solver: GFLOPs throughput (CPU,GPU), Convergence of Solver (in iteration residual) vs Preconditioners.

Test problems: Sphere coupled poisson. Annulus. Stokes on annulus?

12.3 RBF-FD Weights

Given a set of function values, $\{u(\mathbf{x}_j)\}_{j=1}^N$, on a discrete set of nodes $\{\mathbf{x}_j\}_{j=1}^N$, the operator \mathcal{L} acting on $u(\mathbf{x}_j)$ is approximated by a weighted combination of function values, $\{u(\mathbf{x}_i)\}_{i=1}^n$, in a small neighborhood of $u(\mathbf{x}_j)$, where n defines the size of the stencil and $n \ll N$:

$$\mathcal{L}u(\mathbf{x})|_{\mathbf{x}=\mathbf{x}_j} \approx \sum_{i=1}^n w_i u(\mathbf{x}_i) \quad (12.3)$$

The RBF-FD weights, w_i , are found by enforcing that they are exact within the space spanned by the RBFs $\phi_i(\epsilon r) = \phi(\epsilon ||\mathbf{x} - \mathbf{x}_i||)$, centered at the nodes $\{\mathbf{x}_i\}_{i=1}^n$, with $r = ||\mathbf{x} - \mathbf{x}_i||$ being the distance between the RBF center and the evaluation point measured in the standard Euclidean 2-norm. It has also been shown through experience and studies [?, ?, ?, ?] that better accuracy is gained by the interpolant being able to reproduce a

constant. Hence, the constraint $\sum_{i=1}^n c_k = \mathcal{L}\mathbf{1}|_{\mathbf{x}=\mathbf{x}_j} = 0$ is added, where w_{n+1} is ignored after the matrix in (12.4) is inverted. That is,

$$\begin{pmatrix} \phi(\epsilon \|\mathbf{x}_1 - \mathbf{x}_1\|) & \phi(\epsilon \|\mathbf{x}_1 - \mathbf{x}_2\|) & \cdots & \phi(\epsilon \|\mathbf{x}_1 - \mathbf{x}_n\|) & 1 \\ \phi(\epsilon \|\mathbf{x}_2 - \mathbf{x}_1\|) & \phi(\epsilon \|\mathbf{x}_2 - \mathbf{x}_2\|) & \cdots & \phi(\epsilon \|\mathbf{x}_2 - \mathbf{x}_n\|) & 1 \\ \vdots & \ddots & \ddots & \vdots & \vdots \\ \phi(\epsilon \|\mathbf{x}_n - \mathbf{x}_1\|) & \phi(\epsilon \|\mathbf{x}_n - \mathbf{x}_2\|) & \cdots & \phi(\epsilon \|\mathbf{x}_n - \mathbf{x}_n\|) & 1 \\ 1 & 1 & \cdots & 1 & 0 \end{pmatrix} \begin{bmatrix} c_1 \\ c_2 \\ \vdots \\ c_n \\ c_{n+1} \end{bmatrix} = \begin{bmatrix} \mathcal{L}\phi(\epsilon \|\mathbf{x} - \mathbf{x}_1\|)|_{\mathbf{x}=\mathbf{x}_j} \\ \mathcal{L}\phi(\epsilon \|\mathbf{x} - \mathbf{x}_2\|)|_{\mathbf{x}=\mathbf{x}_j} \\ \vdots \\ \mathcal{L}\phi(\epsilon \|\mathbf{x} - \mathbf{x}_n\|)|_{\mathbf{x}=\mathbf{x}_j} \\ 0 \end{bmatrix} \quad (12.4)$$

This small system solve is repeated N times for each \mathbf{x}_j , $j = 1\dots N$, to form the DM associated with one derivative quantity. As an example, if \mathcal{L} is the identity operator then the above procedure will lead to RBF-FD interpolation. If \mathcal{L} is $\frac{\partial}{\partial x}$, it will lead to the DM that approximates the first derivative in x . While Equation 12.4 is symmetric, the added constraints for coefficient c_{n+1} detracts from the system's positive definite-ness. In lieu of this, a direct LU factorization solves for the weights. Also, observe that multiple right hand sides can be employed to efficiently obtain weights corresponding to all required derivative quantities (i.e., $\frac{\partial}{\partial x}$, $\frac{\partial}{\partial y}$, ∇^2 , etc.) in one system solve per stencil center.

12.4 GPU Based Solver

Author's Note: We should only discuss the ViennaCL implementation. Unless I can get my ILU preconditioner implemented for CUSP.

Our implementation leverages existing libraries for sparse matrix-vector operations on the GPU. Two libraries exist for sparse matrix linear algebra on the GPU: CUSP [?] and ViennaCL [?]. By implementing our algorithm in the context of ViennaCL, we directly benefit from improvements to the performance of underlying sparse matrix-vector product, vector dot vector and other linear algebra primitives. Also, ViennaCL provides seamless interoperability with the Boost::UBLAS, EIGEN and MTL libraries via C++ templates. We test the performance of our algorithm on one or more CPUs with the Boost::UBLAS library.

Table comparing CUSP and ViennaCL features

The GMRES algorithm was introduced in 1986 by Saad and Schultz [?]. The iterative solver support general matrix structures, whereas methods like Conjugate Gradient require symmetric positive definiteness.

12.4.1 GMRES Algorithm

At the core of the GMRES algorithm is an Arnoldi (orthogonalization) process. **Author's Note:** significance of orthogonalization.

Multiple variants of GMRES exist **Author's Note:** cite refs that utilize unique orthogonalization steps. The motivation behind alternative Arnoldi processes is to save both memory and operation counts. In some cases stability of the GMRES iterations can also be improved **Author's Note:** I recall a paper saved on my laptop. Saad [?] introduced a practical implementation of the GMRES method that utilizes Given's rotations to compute

an implicit QR factorization. The Given's based algorithm is part of the CUSP library; ViennaCL implements the Householder reflection algorithm.

We have implemented the Given's rotation algorithm within ViennaCL, because it is simpler to implement in parallel and increases parallelism [Author's Note: verify statement with reference](#).

[?] does not describe their use of rotations.)

Note that the application of a preconditioner such as ILU0 introduces an additional call to MPI_Alltoally before everywhere M^{-1} is present in Algorithm 12.2.

Algorithm 12.1 Left-preconditioned GMRES(k) with Given's Rotations

```

1:  $\varepsilon$  (tolerance for the residual norm  $r$ ),  $x_0$  (initial guess), and set  $convergence = false$ 
2: MPI_Alltoally( $x_0$ )
3: while  $convergence == false$  do
4:    $r_0 = M^{-1}(b - Ax_0)$ 
5:   MPI_Alltoally( $r_0$ )
6:    $\beta = \|r_0\|_2$                                       $\triangleright \text{MPI_Allreduce}(<r_0, r_0>)$ 
7:    $v_1 = r_0 / \beta$ 
8:   for  $j = 1$  to  $k$  do
9:      $w_j = M^{-1}Av_j$                                  $\triangleright \text{MPI_Alltoally}(w_j)$ 
10:    for  $i = 1$  to  $j$  do
11:       $h_{i,j} = < w_j, v_i >$                        $\triangleright \text{MPI_Allreduce}$ 
12:       $w_j = w_j - h_{i,j}v_i$ 
13:    end for
14:     $h_{j+1,j} = \|w_j\|_2$                            $\triangleright \text{MPI_Allreduce}$ 
15:     $v_{j+1} = w_j / h_{j+1,j}$ 
16:  end for
17:  Set  $V_k = [v_1, \dots, v_k]$  and  $\bar{H}_k = (h_{i,j})$  an upper Hesssenberg matrix of order  $(m + 1) \times m$ 
18:  Solve a least-square problem of size  $m$ :  $\min_{y \in \mathbb{R}^k} \|\beta e_1 - \bar{H}_k y\|_2$ 
19:   $x_k = x_0 + V_k y_k$ 
20:  if  $\|M^{-1}(b - Ax_k)\|_2 < \varepsilon$  then
21:     $convergence = true$ 
22:  end if
23: end while

```

12.4.2 Multiple GPUs

Domain Decomposition. A restricted additive Schwarz [?] domain decomposition is applied. Traditional additive Schwarz methods construct a restriction matrix R to constrain processor computation to a subdomain and R^T project it back onto the global domain. In contrast, restricted additive Schwarz replaces the restriction matrix R with some restriction operator R_P^0 where $R_P^0 \subset R$, but continues to use R^T to project the solution back onto the global domain. The main idea behind re , restricted additive schwarz assigns all nodes to exactly one subdomain. Regular additive Schwarz would allow nodes to be in one or more subdomains.

$$\mathbf{u} = \sum_{p=0}^{numprocs} R'_p (A(R_p^0)^T)^{-1} R_P^0 R_P F \quad (12.5)$$

where R_p is a restriction matrix (identity on diagonals for stencils operated on by a processor, zero elsewhere). In our case, the partitions are determined by node coordinates, so all non-zeros for a stencil row end up on the same processor (i.e., the decomposition of the matrix does not allow multiple domain blocks per row).

MPI Communication. Mention communication using MPI_AlltoAllv, MPI_Allreduce. (Show communication points in algorithm—be sure to explicitly show Givens rotations in algorithm. [?] did list rotations. We also use the CPU for Gram Schmidt process.)

12.5 Multiple GPUs

Scaling GMRES across multiple GPUs requires a domain decomposition. Domain decompositions can be interpreted as partitioning of nodes or cuts along

Domain Decomposition. The domain is decomposed naïvely by cutting the domain into domain bounding box into P equal sized partitions. Each processor is assigned one partition. Partitions may contain unequal number of nodes. Future work will integrate a domain partitioning library such as ParMETIS [?] or SCOTCH [?] to improve load balancing for processors.

$$\mathbf{u} = \sum_{p=0}^{numprocs} R_p (A R_p^T)^{-1} F \quad (12.6)$$

where R_p is a restriction matrix (identity on diagonals for stencils operated on by a processor, zero elsewhere). In our case, the partitions are determined by node coordinates, so all non-zeros for a stencil row end up on the same processor (i.e., the decomposition of the matrix does not allow multiple domain blocks per row).

The restriction matrix R_p^0 contains only zeros and ones. Cai et al. [?] let R_p^0 contain ones on the diagonal to indicate rows of the system for which a processor p is responsible. Our implementation similarly places I on the diagonal, but we then apply a general permutation matrix P to compress $R_p^0 A$ such that processors assemble an $N_p \times m$ local matrix (where $N_p \ll N$ and $m \ll N$). Likewise, processors allocate vectors for the reduced problem size. The solution is $N_p \times 1$ and RHS is $m \times 1$. Recall that the domain decomposition is determined by node partitioning, which does not guarantee rows of the global system corresponding to stencils in a processor’s partition will be a contiguous block of the full system.

12.5.1 Solution Ordering

To simplify distribution of solution values, we apply a reordering to the system to interleave components of the solution and group solutions by node. This allows us to directly copy a double4 for each

The DM assembly depends on all dimensions of solution values for a single node to be consecutive in memory. While Equation 12.15 has all components of U, V, W and P grouped together, the values of u_1, v_1, w_1 and p_1 correspond to node (x_1, y_1, z_1) .

Figure 12.1 demonstrates the effect of interleaving our solution. The sparsity pattern of the original DM with solutions grouped by component is shown in Figure 12.1a. Well defined blocks of non-zeros are filled with RBF-FD weights from Equation 12.7. Figure 12.1b presents the sparsity pattern for interleaved solution components. The pattern is similar to a single block of Figure 12.1a, but the sub-matrix $(10 : 50) \times (10 : 50)$ of each solution ordering, shown in Figures 12.1c and 12.1d, illustrate that non-zeros in Figure 12.1b are small 4×4 blocks with the structure of Equation 12.7.

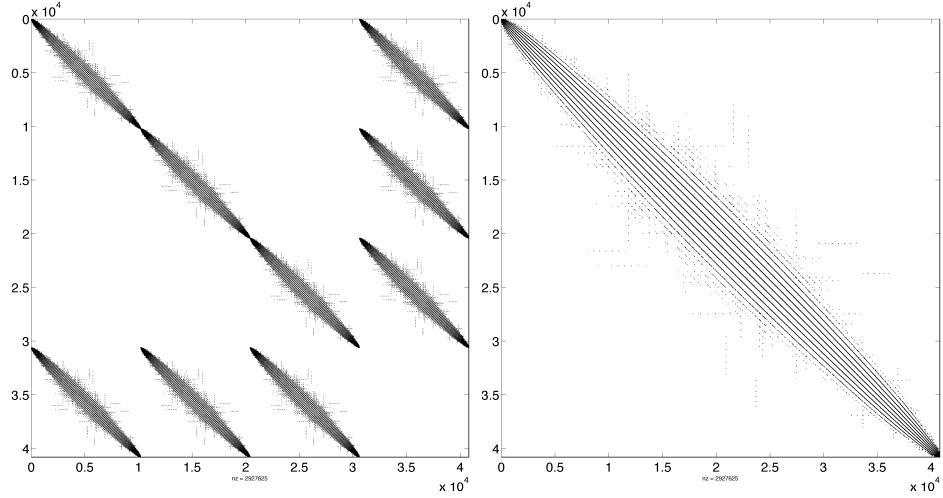
Node Order. *Author's Note: Should we discuss node ordering and its implications on convergence? if so, I should state:*

- Nodes are read from file
- Their order is either: unmodified or modified
- If modified, our goal is to improve memory access patterns by reducing the bandwidth of the DM. A bandwidth of n is the best case scenario where the solution vector is accessed linearly. The worst case scenario is when bandwidth is N and the solution is accessed randomly via stencils.
- Ordering nodes according to a space filling curve can reduce the “randomness” of solution access by placing elements of stencils nearby (sometimes sequential) in memory.
- Many space filling curves exist. Raster (IJK), Snake, Morton, Hilbert, U, X, etc. We consider Raster here, with other orderings left for future work.
- Our restriction operator for domain decomposition might reduce the bandwidth for each processor compared to the global DM. How does the combination of restriction and reordering work?
- Reordering the DM has implications on its condition number and the convergence rate of the GMRES algorithm. We should provide bandwidth of matrix before and after reordering, as well as the condition number before and after. We should monitor convergence with and without reordering, and compare the number of GMRES iterations per minute/second.

MPI Communication. The majority of communication within our implementation consists of two MPI routines: MPI_Alltoallv and MPI_Allreduce.

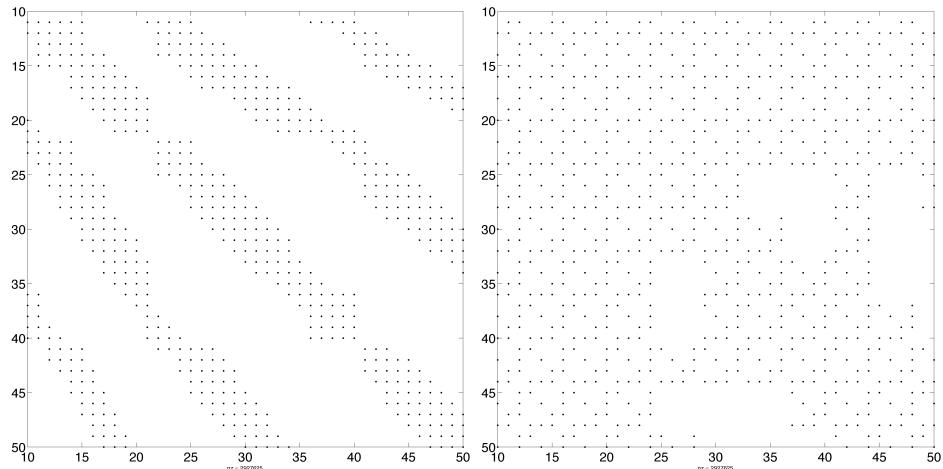
12.6 Governing Equations

We solve the PDE on the surface of the sphere as an example for one and multiple GPUs. Boundary conditions detract from the accuracy of RBF-FD and introduce other issues such as Runge phenomena [?], so we first verify the solution without boundaries.



(a) Grouped by Solution Component

(b) Interleaved Solution Components



(c) Grouped Submatrix $(10 : 50) \times (10 : 50)$
 (d) Interleaved Submatrix $(10 : 50) \times (10 : 50)$

Figure 12.1: Sparsity pattern of linear system in Equation 12.7. Solution values are either ordered by component (e.g., $(u_1, \dots, u_N, v_1, \dots, v_N, w_1, \dots, w_N, p_1, \dots, p_N)^T$) or interleaved (e.g., $(u_1, v_1, w_1, p_1, \dots, u_N, v_N, w_N, p_N)^T$).

Author's Note: Need to reference work that solves problem in two steps and justify our approach to solve in one step. Golub paper might be good for this. Or the Stoke preconditioners paper

Assuming η is a constant (i.e., $\nabla\eta = 0$), our system simplifies to

$$\begin{pmatrix} -\eta\nabla^2 & 0 & 0 & \frac{\partial}{\partial x_1} \\ 0 & -\eta\nabla^2 & 0 & \frac{\partial}{\partial x_2} \\ 0 & 0 & -\eta\nabla^2 & \frac{\partial}{\partial x_3} \\ \frac{\partial}{\partial x_1} & \frac{\partial}{\partial x_2} & \frac{\partial}{\partial x_3} & 0 \end{pmatrix} \begin{pmatrix} u_1 \\ u_2 \\ u_3 \\ p \end{pmatrix} = \frac{RaT}{\sqrt{x_1^2 + x_2^2 + x_3^2}} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ 0 \end{pmatrix}. \quad (12.7)$$

where the ∇^2 operator in spherical polar coordinates for \mathbb{R}^3 is:

$$\nabla^2 = \underbrace{\frac{1}{\hat{r}} \frac{\partial}{\partial \hat{r}} \left(\hat{r}^2 \frac{\partial}{\partial \hat{r}} \right)}_{\text{radial}} + \underbrace{\frac{1}{\hat{r}^2} \Delta_S}_{\text{angular}}. \quad (12.8)$$

Here Δ_S is the Laplace-Beltrami operator—i.e., the Laplacian constrained to the surface of the sphere with radius \hat{r} . This form nicely illustrates the components of the ∇^2 corresponding to the radial and angular terms.

On the surface of the unit sphere the radial term vanishes, so we are left with:

$$\nabla^2 \equiv \Delta_S. \quad (12.9)$$

The following RBF operator from [?—Equation (20) can be applied to the RHS of Equation 12.4 to generate Laplace-Beltrami RBF-FD weights:

$$\Delta_S = \frac{1}{4} \left[(4 - r^2) \frac{\partial^2}{\partial r^2} + \frac{4 - 3r^2}{r} \frac{\partial}{\partial r} \right], \quad (12.10)$$

where r is the Euclidean distance between nodes of an RBF-FD stencil and is independent of our choice of coordinate system.

Additionally following [?, ?], the off-diagonal blocks in Equation 12.7 must be constrained to the sphere via the projection matrix:

$$P = I - \mathbf{x}\mathbf{x}^T = \begin{pmatrix} (1 - x_1^2) & -x_1x_2 & -x_1x_3 \\ -x_1x_2 & (1 - x_2^2) & -x_2x_3 \\ -x_1x_3 & -x_2x_3 & (1 - x_3^2) \end{pmatrix} = \begin{pmatrix} P_{x_1} \\ P_{x_2} \\ P_{x_3} \end{pmatrix} \quad (12.11)$$

where \mathbf{x} is the unit normal at the stencil center, and [?, ?] show that with a little manipulation weights can be directly computed with these operators for Equation 12.4:

$$P \frac{\partial}{\partial x_1} = (x_1 \mathbf{x}^T \mathbf{x}_k - x_{1,k}) \frac{1}{r} \frac{\partial}{\partial r} |_{\mathbf{x}=\mathbf{x}_j} \quad (12.12)$$

$$P \frac{\partial}{\partial x_2} = (x_2 \mathbf{x}^T \mathbf{x}_k - x_{2,k}) \frac{1}{r} \frac{\partial}{\partial r} |_{\mathbf{x}=\mathbf{x}_j} \quad (12.13)$$

$$P \frac{\partial}{\partial x_3} = (x_3 \mathbf{x}^T \mathbf{x}_k - x_{3,k}) \frac{1}{r} \frac{\partial}{\partial r} |_{\mathbf{x}=\mathbf{x}_j} \quad (12.14)$$

12.6.1 Constraints

Due to the lack of boundary conditions on the sphere, the family of solutions that satisfy the PDE in Equation 12.7 includes four free constants (one for each u_1, u_2, u_3 and p).

One way to close the null-space of the solution is to augment Equation 12.7 with the following constraints:

$$\begin{pmatrix} -\eta \nabla^2 & 0 & 0 & \frac{\partial}{\partial x_1} \\ 0 & -\eta \nabla^2 & 0 & \frac{\partial}{\partial x_2} \\ 0 & 0 & -\eta \nabla^2 & \frac{\partial}{\partial x_3} \\ \frac{\partial}{\partial x_1} & \frac{\partial}{\partial x_2} & \frac{\partial}{\partial x_3} & 0 \\ \vdots & \vdots & \vdots & \vdots \\ 1_{N \times N} & 0 & 0 & 0 \\ 0 & 1_{1 \times N} & 0 & 0 \\ 0 & 0 & 1_{1 \times N} & 0 \\ 0 & 0 & 0 & 1_{1 \times N} \end{pmatrix} \begin{pmatrix} 1_{N \times 1} & 0 & 0 & 0 \\ 0 & 1_{N \times 1} & 0 & 0 \\ 0 & 0 & 1_{N \times 1} & 0 \\ 0 & 0 & 0 & 1_{N \times 1} \\ \vdots & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix} \begin{pmatrix} u_1 \\ u_2 \\ u_3 \\ p \\ c_1 \\ c_2 \\ c_3 \\ c_4 \end{pmatrix} = \frac{RaT}{\sqrt{x_1^2 + x_2^2 + x_3^2}} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ 0 \\ \int_{\Omega} u_1 \partial \bar{\Omega} \\ \int_{\Omega} u_2 \partial \Omega \\ \int_{\Omega} u_3 \partial \Omega \\ \int_{\Omega} p \partial \Omega \end{pmatrix}. \quad (12.15)$$

Author's Note: Need the integral of my manufactured solution on the RHS. ℓ_1 norm does not converge to 0, so it will screw solve with constraints where the subscript on $1_{N \times 1}$ indicates a $N \times 1$ vector of ones. These constraints add the unknowns (c_1, c_2, c_3, c_4) , which should solve to be zero. The four rows on the bottom require that the solution satisfy the integral over the domain for each solution component. In combination with the four added columns, the constraints indicate that the solution components must satisfy integrals using the same constant value. This is only possible if the constants are zero. Our added constraints are not chosen for physical significance, but for algebraic significance. When solving Equation 12.15 with GMRES, the constraints improve conditioning of the system and increase the rate of convergence.

We also investigate the use of GMRES without constraints. This increases the number of iterations required to converge, but allows increased parallelism (decreased data sharing).

Author's Note: Perhaps we can iterate without constraints until convergence slows then "restart" the problem on a single GPU with constraints included? Limits scalability but would allow more parallelism for part of iterations while also reasonable convergence.

12.6.2 Manufactured Solution

To verify our implementation, we manufacture a solution that satisfies the continuity equation. Using the identity

$$\nabla \cdot (\nabla \times g(\mathbf{x})) = 0, \quad (12.16)$$

for any function $g(\mathbf{x})$, we can easily manufacture a solution by choosing some vector function $g(\mathbf{x})$, projecting it onto the sphere via $P_x g(x)$ and applying the curl projection, Q_x :

$$Q_x = \begin{bmatrix} 0 & -x_3 & x_2 \\ x_3 & 0 & -x_1 \\ -x_2 & x_1 & 0 \end{bmatrix}. \quad (12.17)$$

Figure 12.2: A divergence free field is manufactured for the sphere.

Then, a manufactured solution that satisfies both momentum and continuity conditions on the surface of the sphere is given by:

$$\mathbf{u} = Q_x(g(\mathbf{x})) \quad (12.18)$$

Typically, on the surface of the sphere, the projection operator from Equation 12.11 must be applied to an arbitrary $g(\mathbf{x})$. Here, we choose the components of $g(\mathbf{x})$ to be various spherical harmonics in Cartesian coordinates where $P_x Y_l^m = Y_l^m$. Consequently, application of P_x can be ignored.

We select $g(x)$ to be:

$$g(x) = 8Y_3^2 - 3Y_{10}^5 + Y_{20}^{20} \quad (12.19)$$

and the pressure function:

$$P = Y_6^4 \quad (12.20)$$

The manufactured solution is shown in Figure 12.2. A Mollweide projection [?] maps the sphere onto the plane.

Convergence. As the problem size N increases, we expect the approximation to the solution to converge on the order of \sqrt{n} where n is the choice of stencil size. Figure ?? demonstrates the convergence of our solution with respect to \sqrt{N} for stencil sizes $n = 31, 101$

Author's Note: [update when figure is complete](#).

12.7 Preconditioning

GMRES is slow to converge when used without a preconditioner.

The differentiation matrix produced by RBF-FD is asymmetric, non-positive definite, and non-diagonally dominant. Thus, many of the popular choices for preconditioning provided by CUSP and ViennaCL do not apply.

Our tests show that an incomplete LU factorization with zero fill-in [?] functions well.

We also find that a large Krylov subspace must be saved. GMRES converges best when approximately 250 dimensions are saved between restarts.

Plot comparing residual of GMRES without precond and with ILU0

Author's Note: [Need to comment on the conditioning of the system and how stencils can influence convergence](#).

Need a table/plot comparing convergence of various preconditioners (ILU0, ILUT, AMG, etc.). We will justify our use of ILU0 even if it is the most basic and frequently least beneficial approach.

Demonstrate ILU0 is best for converging between

- Jacobi
- ILU0 on block $(1 : 3 * N)$

- ILU0 on full matrix
- ILUT
- AMG

Algorithm 12.2 Incomplete LU Factorization with Zero Fill-in (ILU0)

```

1: for i do = 0
2:    $a_{i,i} = a_{i,i}/a_{i,i}$ 
3: end for

```

12.8 GMRES Results

We want to express benchmarks in terms of the number of GMRES iterations per second, and the number of iterations required to converge. Readers wont care what percentage of peak we are getting, just how fast we get to the solution.

- One GPU
 - Convergence for stokes steady
 - GMRES iteration plot (assuming 1e-8 and restart=60)
- Multi-GPU
 - Number of iterations without constraints
 - Number of iterations with constraints
 - GMRES iteration plot
 - plot: number of GMRES iterations per second w.r.t. number of processors

12.9 Fragments

Need to describe the solver.

State that the conditions under which a problem is solved determines how quickly it will converge. For example, nodes too close together on delaunay meshes cause higher condition numbers and require more iterations. Proper choice of preconditioners can dramatically reduce the number of iterations required to converge. Although preconditioners incur an additional cost in preprocessing and at each iteration of the solver, the potential number of iterations they save

John Dennis dissertation has a list of nice datasets (compare the list to Hoth) and their preconditioners. We might list similar paramters used for results here.

CHAPTER 13

PERFORMANCE BENCHMARKS

Author's Note: First I need to describe the hardware used. This includes: Troi, Spear, Keeneland; their gpus and RAM, etc. Discuss interconnects (why can't we run multiple kernels on one node of spear)

We present our implementation of an efficient multi-node, multi-GPU RBF PDE package to run on clusters of GPU compute nodes. Each compute node has one or more GPUs attached. Specifically, we utilize the hardware available in the FSU HPC Spear cluster and the NFS funded Keeneland project. While the Spear cluster is only a handful of nodes with 2 GPUs each, Keeneland boasts a total of 240 CPUs and 320 GPUs. The large scale of Keeneland allows us to verify the scaling of our method.

13.1 Metrics

The code produces the same results on CPUs and GPUs. However, their performance differs, so we analyze this difference with the speedup metric:

$$S_p = T_{\text{serial}} / T_{\text{parallel}}$$

Author's Note: Include efficiency:

$$E_p = S_p / \# \text{ of processors}$$

13.2 OpenCL

We leverage the OpenCL language for functional portability.

Our dedication to OpenCL is a hedged bet that the future architectures will merge in the middle between many and multi-core architectures with co-processors alongside CPUs. By selecting an open standard parallel programming language, we increase the likelihood for future support of our programs.

13.2.1 OpenCL vs CUDA

The market is volatile. Companies survive by investing margins in their next great product. If a product fails or the company faces a recall, their survival may come into

question. Thus far, NVidia's CUDA has been wildly popular, but for the longest time (until May 2012) it was closed source. The closed source limited the language to NVidia hardware. As such, the OpenCL language gained popularity due to its support for AMD, Intel, mobile devices, web browsers, etc. NVidia's push to provide an open source compiler may be an attempt to regain the market share, but OpenCL appears to be on good footing. One other point: with an open source NVidia compiler, OpenCL can be optimized by the more mature NVidia compiler for their proprietary hardware. OpenCL compilers are also becoming more sophisticated at auto-optimization.

13.2.2 Asynchronous Queuing

Provide details and simple example of how asynchronous queueing can be used.

Need a figure showing the overlapping comm and comp in a general process with the wait points marked.

13.3 Fermi Architecture

In Spring 2010, NVidia will publicly release a new architecture code named "Fermi" [?]. The new hardware will support many features of interest, the most important being 8x faster double precision than the older Tesla C1060 (GT200 architecture). It will also allow for concurrent kernel execution (for up to 16 small kernels) making it easier to keep the GPU saturated with computation. Table 13.1 considers some of the more monumental differences between the Fermi and GT200 architectures.

13.3.1 Double Precision

Double precision operations take XXX cycles

13.3.2 Local Caching

Local caching allows us to bypass the need for

13.3.3 Multiple Kernel Scheduling

describe fermi's ability to schedule multiple kernels, what it means for our queues. Do we need multiple queues, or just one that is non-blocking. How do we indicate we are done communicating if there is no queue to add markers to?

13.3.4 Future NVidia Hardware

The latest hardware is the Kepler. It supports the following features

	Fermi	GT200
# of concurrent kernels	16	1
Warps scheduled concurrently	2	1
Clock cycles to dispatch instruction to warp	2	4
Caching on Global Device Memory	Yes	No
Shared memory	64 KB	16 KB
Shared memory banks	32	16
Bank conflict free FP64 access	Yes	No
Cycles to read FP64 (from shared memory)	2	32
Max allowed warps	48	32

Table 13.1: Comparison of NVidia’s new Fermi architecture to the GT200 architecture used for GTX 280, Tesla C1060 and other GPUs in use today.

13.4 HPC Spear Cluster

13.5 Keeneland

13.6 Future Hardware

These figures represent optimizations of the Cosine Bell and Vortex Roll-up test cases. Essentially, the optimizations here are general for multi-GPU SpMV. Improving these test cases improves all explicit schemes for RBF-FD (i.e., hyperbolic and parabolic equations, and various time stepping schemes like euler, RK4, Adams-Bashforth etc.).

For the MPI I might need to have multiple figures comparing performance. However, for the GPU optimizations I can show a single plot with all the curves on it. These sections will be good

13.7 MPI_Alltoallv

Change send/recv to alltoallv. Track wait time. Show limitation on scalability with GPU (sublinear) vs CPU (linear). How high can we get linear on CPU?

Communication between processors requires each processor to iterate through their neighboring processors and share information. This can be seen as a simple for loop allowing every processor to touch its neighbors in round-robin fashion. The benchmarks seen

in [Author's Note: figures from paper1](#) show the strong scaling of our implementation with a for loop and send/recv.

In Figs ?? we show the strong scaling of the cosine bell after

Alternatively

an all to all collective. That is, all processors share some information with potentially every other processor.

Author's Note: cleanup: These results were run on Keeneland. The changes to MPI communication are the result of changing from blocking communication (MPI_send/MPI_recv) to non-blocking communication (MPI_alltoall).

Figure 13.1a shows that our GPU kernel is not much different than in the paper. I have a list of optimizations I'm going through, but this test case focuses on improving the communication.

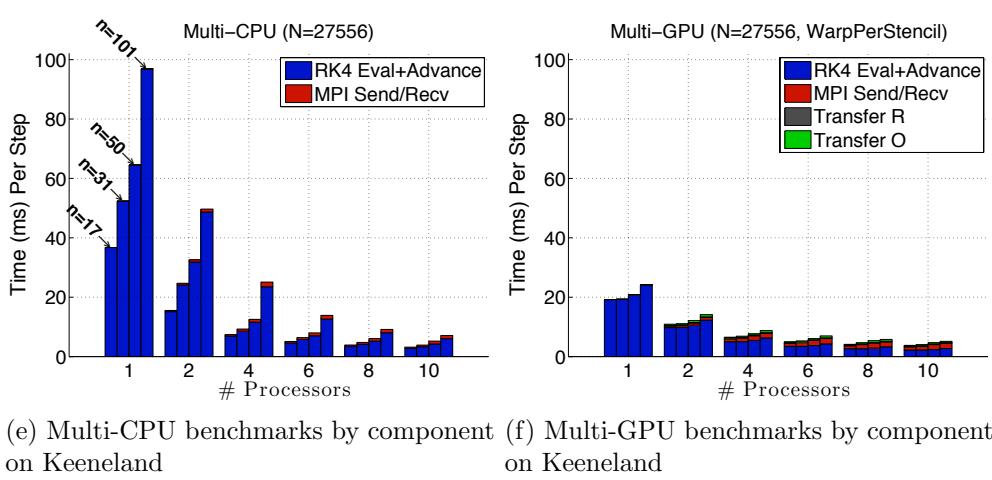
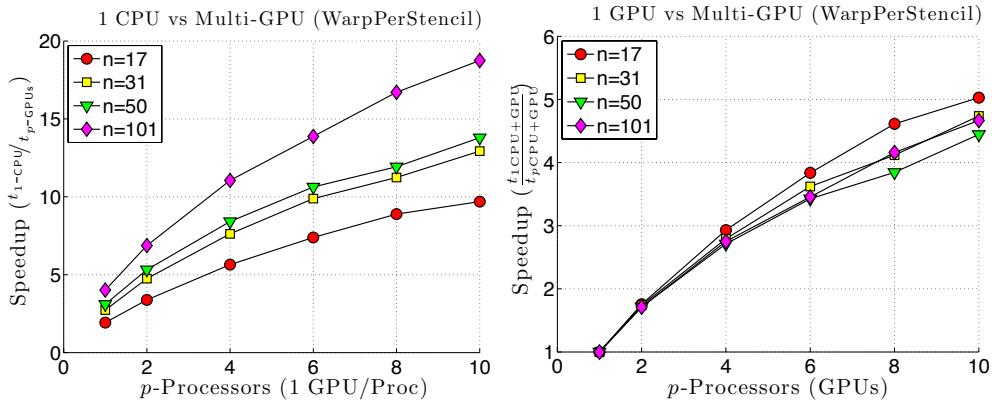
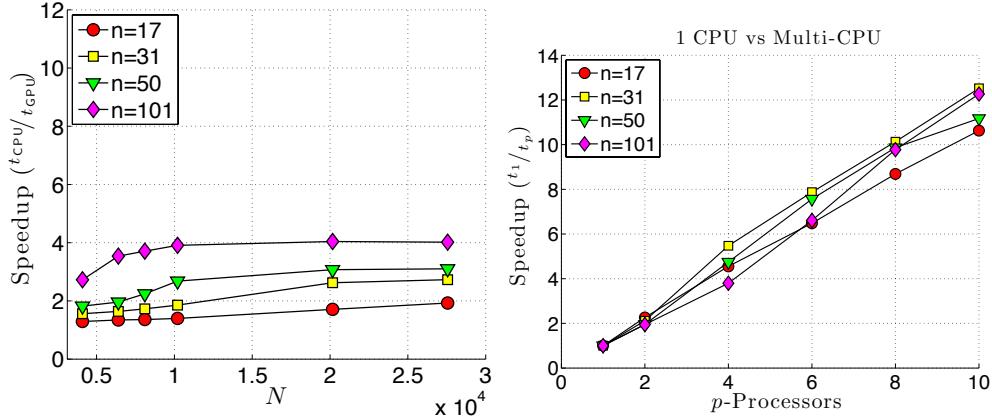
Figure 13.1b shows the strong scalability of our method on Multiple CPUs. In distributed computing, ideal scaling is linear. This figure demonstrates that our method does scale linearly (almost super-linearly) as the number of CPUs increases, so our prospect for spanning all CPUs on Keeneland is within reach for problem sizes large enough. The super-linear speedup seen for 10 processors results from improved caching on processors as their individual problem sizes decrease and the processors are able to keep a larger percentage of the problem within fast cache memory.

Figure 13.1c shows the scaling of multiple GPUs vs 1 CPU. Ideally, this figure would be the product of the previous two figures since the GPUs are attached to CPUs in a one to one correspondence. However, we see from the sub-linear scaling that while the GPU accelerators are decreasing the time to compute solutions, there is less and less return of investment as the number of processors increase. Between this Figure and the previous, the only thing that differs is the hardware on which stencils are evaluated. The cost of communication stays the same as in the previous figure. But that means the communication consumes a increasing percentage of the iteration time, until it dominates. Additionally, computing on the GPU requires transfer (additional communication) of data between CPU and GPU.

Figure 13.1d shows the scalability of multiple GPUs vs 1 GPU. Here we see a sub-linear behavior for all cases. This is attributed to both the cost of transfer between CPUs and GPUs and the decreasing problem size as number of processors increases, which underutilizes the GPUs.

Figure 13.1e and Figure 13.1f show the smaller percentage of time per iteration dedicated to communication compared to the figures in the paper. In the Figure 13.1f, the way the times bottom out indicates we are/have converged on the minimum time required to launch a GPU kernel, transfer to/from the GPU, and communicate the problem via MPI. To scale to more processors, a larger problem size is absolutely necessary.

I am generating another set of figures that demonstrate the scaling when we overlap communication and computation. MPI collectives do not allow overlap, but the asynchronous GPU kernel launches do. Therefore, I expect: - the scaling on multiple CPUs vs 1 CPU to be the same as it is now - the scaling on multiple GPUs vs 1 GPU will improve to linear/super-linear for problem sizes that occupy the hardware longer than the minimum kernel launch time. For N=27556 we might only see linear speedup up to 6 or 8 processors. - larger problem sizes will still be necessary (I have benchmarks for 100K, 500K and 1M on



the sphere).

Author's Note: Need to test weak scaling (problem size stays fixed per processor). This will require a modified code, but we can fill the weight matrices with anything and run the kernels. Author's Note: Need to compare one warp of threads to a full block of size n

13.8 Asynchronous OpenCL

What are the limitations if using just async and not the queues?

13.9 Multi-Queue OpenCL

How does performance improve if we use two queues (one for Q and one for R)?

13.10 GPU Kernel Optimizations

13.10.1 Work-Group Size and Number of Stencils

What if a work-group is larger than a warp? What if the group was occupied by multiple stencils. What improvements to speedup do we see?

How many stencils can each group handle (assuming values stay in shared memory)? Shared memory bank conflicts? How do we sort the values? What is the occupancy of the GPU?

13.10.2 Parallel Reduction in Shared Memory

What significant gain do we see from adding a segmented scan to the shared memory? What other improvements can we think of?

13.10.3 Comparison: custom SpMV for explicit schemes vs ViennaCL

CHAPTER 14

NEIGHBOR QUERIES AND NODE ORDERING

14.1 Neighbor Queries

As part of the preprocessing stage for RBF-FD, the scattered point cloud must be analyzed to generate stencils. To generate a stencil, any collection of nodes can be selected. However, by choosing nodes close to the stencil center and well balanced around it, we stand to get the best possible approximations to derivatives.

Why? well, the approximations are based on differences. Similar to classic FD, draw a secant connecting two nodes of a stencil. The slope of the secant determines the gradient at either point or a point in between. In the limit as the points are moved closer to the same spot, the approximation to the derivative at that point becomes exact.

So ideally every RBF-FD stencil will operate on nearest neighbors. One way to generate nearest neighbors is brute force $O(N^2)$; very inefficient.

Alternatives exist including k-D Tree

14.1.1 k-D Tree

Build complexity, seek times. Internal ordering

We use implementation provided by Andrea Tagliasacchi [?] as a MEX compiled set of routines for Matlab. These same routines are then linked into C/C++.

14.1.2 Locality Sensitive Hashing

[?] provide a fast parallel

[?] is working on parallel generation. [?] has OpenCL neighbor queries

We started with a CPU implementation to test appropriateness.

[Author's Note: Compare performance of stencil generation in C++](#)

14.2 Node Ordering

Locality sensitive hashing also allows us to reorder the nodes

[?] mentions the impact of ordering on conditioning.

Algorithms like Reverse Cuthill McKee and Approximate Minimum Degree ordering allow general restructuring of matrices.

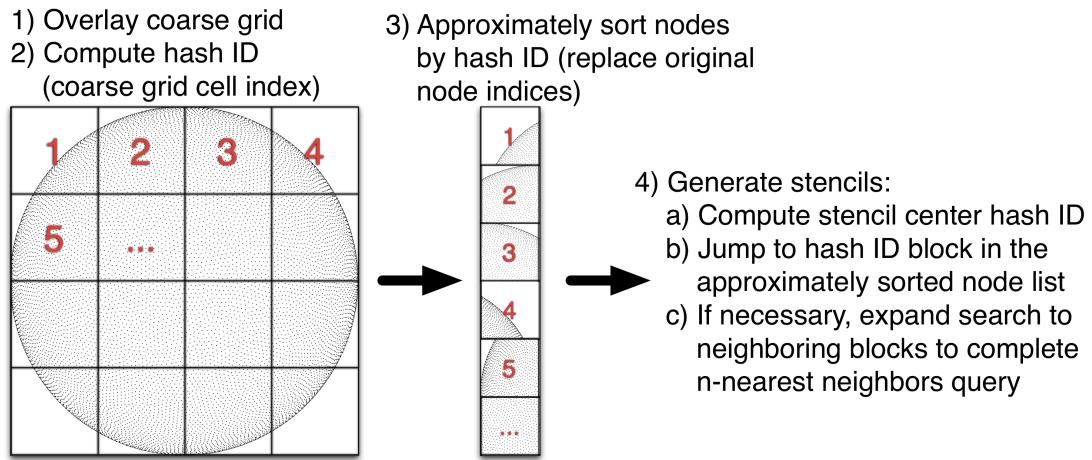


Figure 14.1: High level overview of Locality Sensitive Hashing algorithm. First we overlay a coarse regular grid on the bounding box of the domain. The cells of the coarse grid cells are reordered in memory according to a space filling curve. Then we query neighbors by starting search with [Author's Note: Illustrate the query process](#) cell containing stencil center and append neighboring cells until stencil size n nodes are found. We take n closest neighbors (brute force search) if more than n are appended to the list of candidates.

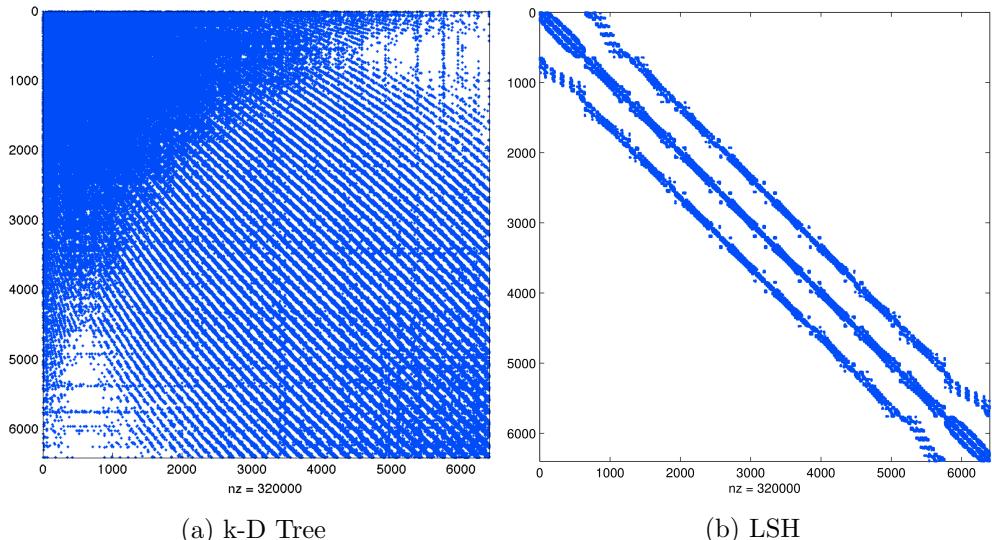


Figure 14.2: Example effects of node reordering within neighbor query algorithm for MD node set $N = 6400$ with $n = 50$. Matrix is 0.78% full. k-D Tree maintains original ordering of the nodes and deceptively appears nearly dense. LSH algorithm reorders nodes according to raster ordering and reveals sparsity of the problem.

Author's Note: NEed to compare conditioning of LSH and other algorithms in Matlab

Q: what is an ideal ordering? Q: what is the best conditioning from ordering? Q: what is the relative cost of ordering?

CHAPTER 15

COMMUNITY OUTREACH

-
-
-
-
-
-
-

TODO: add survey of goals stated in prospectus and whether they were completed or not
(Probably part of slides, not actual dissertation)

APPENDIX A

AVOIDING POLE SINGULARITIES WITH RBF-FD

This content follows [?, ?].

Within the test cases of this dissertation, we solve convective PDEs on the unit sphere with the form:

$$\frac{\partial h}{\partial t} = \mathbf{u} \cdot \nabla h$$

where \mathbf{u} is velocity. For example, the cosine bell advection has this particular form:

$$\frac{\partial h}{\partial t} = \frac{u}{\cos \theta} \frac{\partial h}{\partial \lambda} + v \frac{\partial h}{\partial \theta} \quad (\text{A.1})$$

in the spherical coordinate system defined by

$$\begin{aligned} x &= \cos \theta \cos \lambda \\ y &= \cos \theta \sin \lambda \\ z &= \sin \theta \end{aligned}$$

where $\theta \in (-\frac{\pi}{2}, \frac{\pi}{2})$ is the elevation angle and $\lambda \in (-\pi, \pi)$ is the azimuthal angle. Observe that as $\theta \rightarrow \pm \frac{\pi}{2}$, the $\frac{1}{\cos \theta}$ term goes to infinity as a discontinuity.

One of the many selling points for RBF-FD and other RBF methods is their ability analytically avoid pole singularities, which arise from the choice of coordinate system and not from the methods themselves. Since RBFs are inherently based on Euclidean distance between nodes, and not geodesic distance, it is said that they do not “feel” the effects of the geometry or recognize singularities naturally inherent in the coordinate system [?]. Here we demonstrate how pole singularities are analytically avoided with RBF-FD for cosine bell advection.

Let $r = \|\mathbf{x} - \mathbf{x}_j\|$ be the Euclidean distance which is invariant of the coordinate system. In Cartesian coordinates, we have

$$r = \sqrt{(x - x_j)^2 + (y - y_j)^2 + (z - z_j)^2}.$$

In spherical coordinates we have:

$$r = \sqrt{2(1 - \cos \theta \cos \theta_j \cos(\lambda - \lambda_j) - \sin \theta \sin \theta_j)}.$$

The RBF-FD operators for $\frac{d}{d\lambda}, \frac{d}{d\theta}$ are discretized with the chain rule:

$$\frac{d\phi_j(r)}{d\lambda} = \frac{dr}{d\lambda} \frac{d\phi_j(r)}{dr} = \frac{\cos \theta \cos \theta_j \sin(\lambda - \lambda_j)}{r} \frac{d\phi_j(r)}{dr}, \quad (\text{A.2})$$

$$\frac{d\phi_j(r)}{d\theta} = \frac{dr}{d\theta} \frac{d\phi_j(r)}{dr} = \frac{\sin \theta \cos \theta_j \cos(\lambda - \lambda_j) - \cos \theta \sin \theta_j}{r} \frac{d\phi_j(r)}{dr}, \quad (\text{A.3})$$

where $\phi_j(r)$ is the RBF centered at \mathbf{x}_j .

Plugging A.2 and A.3 into A.1, produces the following explicit form:

$$\frac{dh}{dt} = u(\cos \theta_j \sin(\lambda - \lambda_j) \frac{1}{r} \frac{d\phi_j}{dr}) + v(\sin \theta \cos \theta_j \cos(\lambda - \lambda_j) - \cos \theta \sin \theta_j \frac{1}{r} \frac{d\phi}{dr})$$

where $\cos \theta$ from A.2 analytically cancels with the $\frac{1}{\cos \theta}$ in A.1.

Then, formally, one would assemble differentiation matrices containing weights for the following operators:

$$\mathbf{D}_\lambda = \cos \theta_j \sin(\lambda - \lambda_j) \frac{1}{r} \frac{d\phi_j}{dr}, \quad (\text{A.4})$$

$$\mathbf{D}_\theta = \sin \theta \cos \theta_j \cos(\lambda - \lambda_j) - \cos \theta \sin \theta_j \frac{1}{r} \frac{d\phi}{dr}, \quad (\text{A.5})$$

and solve the explicit method of lines problem:

$$\frac{dh}{dt} = u\mathbf{D}_\lambda h + v\mathbf{D}_\theta h$$

where now the system is completely free of singularities at the poles [?].

We note that the expression $\cos(\frac{\pi}{2})$ evaluates on some systems to a very small number rather than zero (e.g., $6.1(10^{-17})$ on the Keeneland system with the GNU gcc compiler). The small value in turn allows $\frac{1}{\cos \theta}$ to evaluate to a large value (e.g., $1.6(10^{16})$) rather than “inf” or “NaN”. A large value allows the cosine terms to cancel in double precision, whereas an “inf” or “NaN” would corrupt the numerics. Rather than avoid placing nodes at the poles, or assuming the machine will numerically cancel the singularities, it is preferred to use operators A.4, A.5 on the RHS of Equation 6.5 to compute RBF-FD weights.

APPENDIX B

PROJECTED WEIGHTS ON THE SPHERE

It is shown in [?, ?] that a projection operator

$$\mathbf{P} = \mathbf{I} - \mathbf{x}\mathbf{x}^T = \begin{bmatrix} (1-x^2) & -xy & -xz \\ -xy & (1-y^2) & -yz \\ -xz & -yz & (1-z^2) \end{bmatrix} = \begin{bmatrix} \mathbf{p}_x^T \\ \mathbf{p}_y^T \\ \mathbf{p}_z^T \end{bmatrix}$$

where \mathbf{p}_x^T represents the projection operator in the x direction.

From [?], the projected RBF gradient operator is:

$$\begin{aligned} \mathbf{P} \cdot \nabla \phi_k(r(\mathbf{x})) &= \mathbf{P} \cdot \frac{(\mathbf{x} - \mathbf{x}_k)}{r(\mathbf{x})} \frac{d\phi_k(r(\mathbf{x}))}{dr(\mathbf{x})} \\ &= -\mathbf{P} \cdot \mathbf{x}_k \frac{1}{r(\mathbf{x})} \frac{d\phi_k(r(\mathbf{x}))}{dr(\mathbf{x})} \\ &= \begin{bmatrix} x\mathbf{x}^T \mathbf{x}_k - x_k \\ y\mathbf{x}^T \mathbf{x}_k - y_k \\ z\mathbf{x}^T \mathbf{x}_k - z_k \end{bmatrix} \frac{1}{r(\mathbf{x})} \frac{d\phi(r(\mathbf{x}))}{dr}. \end{aligned} \quad (\text{B.1})$$

The operator $\mathbf{I} - \mathbf{x}\mathbf{x}^T$ for $\mathbf{x} = (x, y, z)$ projects a vector onto the plane tangent to the unit sphere at (x, y, z) . Therefore, Equation B.1 gives the projection of the gradient operator at \mathbf{x}_k onto the plane tangent to \mathbf{x} .

B.1 Direct Weights

Following [?], B.1 takes on the following when adapted to RBF-FD:

$$[\mathbf{p}_x \cdot \nabla f(\mathbf{x})]|_{\mathbf{x}=\mathbf{x}_c} = \sum_{k=1}^n c_k \underbrace{\left[x_c \mathbf{x}_c^T \mathbf{x}_k - x_k \right]}_{B_{c,k}^{\mathbf{p}_x}} \frac{1}{r} \frac{d\phi(r(x_c))}{dr}. \quad (\text{B.2})$$

and so forth for the $\mathbf{p}_y \cdot \nabla, \mathbf{p}_z \cdot \nabla$ operators, where \mathbf{x}_c is the stencil center and \mathbf{x}_k are stencil nodes. To compute RBF-FD weights for the $\mathbf{p}_x \cdot \nabla$ operator, the RHS of Equation 6.5 is filled with elements $B_{c,k}^{\mathbf{p}_x}$. We will refer to this method of obtaining the weights as the *direct* method due to the ability to directly compute RBF-FD weights for the operators $\mathbf{P} \cdot \nabla$, and assemble the differentiation matrices $\mathbf{D}_{\mathbf{p}_x \cdot \nabla}, \mathbf{D}_{\mathbf{p}_y \cdot \nabla}, \mathbf{D}_{\mathbf{p}_z \cdot \nabla}$ without the need to compute and/or store other weights.

B.2 Indirect Weights

Alternatively, one is able to compute weights *indirectly* as a weighted combination of existing RBF-FD weights for the unprojected ∇ operator. Here we assume that differentiation matrices to compute the components of ∇ are readily available in memory:

$$\mathbf{D}_\nabla = \begin{bmatrix} \mathbf{D}_{\frac{d}{dx}} \\ \mathbf{D}_{\frac{d}{dy}} \\ \mathbf{D}_{\frac{d}{dz}} \end{bmatrix},$$

where each matrix contains weights computed with the operators of Equation ?? applied to the RHS of Equation 6.5.

The differentiation matrices for $\mathbf{P} \cdot \nabla$ can then be assembled as a weighted combination of the differentiation matrices for the unprojected operator:

$$\mathbf{D}_{\mathbf{P} \cdot \nabla} = \begin{bmatrix} \mathbf{D}_{\mathbf{p}_x \cdot \nabla} \\ \mathbf{D}_{\mathbf{p}_y \cdot \nabla} \\ \mathbf{D}_{\mathbf{p}_z \cdot \nabla} \end{bmatrix} = \begin{bmatrix} \text{diag}(1 - X^2) \mathbf{D}_{\frac{d}{dx}} - \text{diag}(XY) \mathbf{D}_{\frac{d}{dy}} - \text{diag}(XZ) \mathbf{D}_{\frac{d}{dz}} \\ -\text{diag}(XY) \mathbf{D}_{\frac{d}{dx}} + \text{diag}(1 - Y^2) \mathbf{D}_{\frac{d}{dy}} - \text{diag}(YZ) \mathbf{D}_{\frac{d}{dz}} \\ -\text{diag}(XZ) \mathbf{D}_{\frac{d}{dx}} - \text{diag}(YZ) \mathbf{D}_{\frac{d}{dy}} + \text{diag}(1 - Y^2) \mathbf{D}_{\frac{d}{dz}} \end{bmatrix} \quad (\text{B.3})$$

where $X = \{x_{c,i}\}_{i=1}^N$, $Y = \{y_{c,i}\}_{i=1}^N$, $Z = \{z_{c,i}\}_{i=1}^N$ are all x-components, y-components and z-components of the stencil centers $\{\mathbf{x}_{c,i}\}_{i=1}^N$ respectively.

Author's Note: [Cleanup](#): This concept equates to classical Finite Differences where for example, the standard 5-point finite difference formula for approximating the Laplacian can be expressed a weighted combination of differences for

Have not found in literature any mention of the fact that RBF-FD weights can be used to compose operators like this. Generally, its easier to directly compute weights and assumed to be more accurate. But how much more accurate?

One benefit of indirect weights is conservation of memory. For example, for complex operators, a single DM on $N = 1$ million nodes and $n = 101$ requires roughly 1.6 GB of memory. If the PDE is coupled and requires

This also allows us to compose complex operators with weights loaded from disk

$O(N * n)$ cost to assemble the indirect operators after $O(N * n^3)$ cost of assembling direct operators means this approach has potential to save FLOPs in the long run

But the question is, how accurate is it? In situations where memory is critical and these FLOPs need to be saved (i.e., large N and complicated equations), would it be useful?

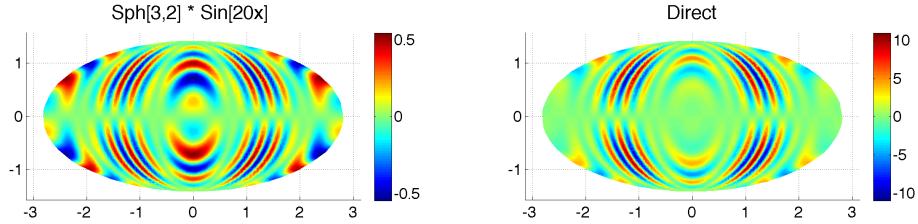
Author's Note: [Q: how do direct vs indirect weights compare? Is the lsfc different sparsity with same approximation potentials? Author's Note: :end](#)

B.3 Test Cases

We computed direct and indirect approaches for the MD-node sets with size $N = \{121, 256, 400, 841, 1024, 2500, 4096, 6400, 8100, 10201, 16384, 27556\}$.

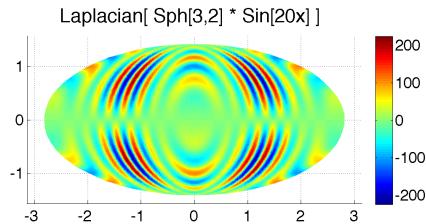
We check the relative error of the approximation:

$$\text{relative } \ell_2 \text{ error} = \frac{\|f_{\text{approx}} - f_{\text{exact}}\|_2}{\|f_{\text{exact}}\|_2}$$



(a) Manufactured test function:
 $f(\mathbf{x}) = Y_3^2(\mathbf{x}) \sin(20x)$.

(b) x -component of the projected gradient:
 $\mathbf{p}_x \cdot \nabla f(\mathbf{x})$.



(c) Surface Laplacian: $\Delta_S f(\mathbf{x})$.

Figure B.1: Test function and its projected derivatives on the surface of the unit sphere.

We also look at the difference of relative errors and its absolute value:

$$(\text{relative } \ell_2 \text{ error})_{\text{direct}} - (\text{relative } \ell_2 \text{ error})_{\text{indirect}}$$

We find that our indirect approach functions well compared to the direct method. For small node sizes ($N < 2500$ nodes) we see that the direct method has the advantage with

B.4 Conclusions

Although it is clear the indirect method functions well compared to the direct method, we must consider its usefulness. Typically, weights are computed only as necessary for the PDE. If the PDE is on the sphere, then directly computing the $\mathbf{P} \cdot \nabla$ operator would be most efficient for both memory and computation. However, one could imagine a scenario such as a 3-D spherical shell domain with physics on the boundaries that must be constrained to the surface, while the interior requires only an unprojected ∇ operator. In such cases, by simply computing for the ∇ operator, we assemble all necessary operators with minimal loss of accuracy and significant savings ($3Nn$ doubles) in storage.

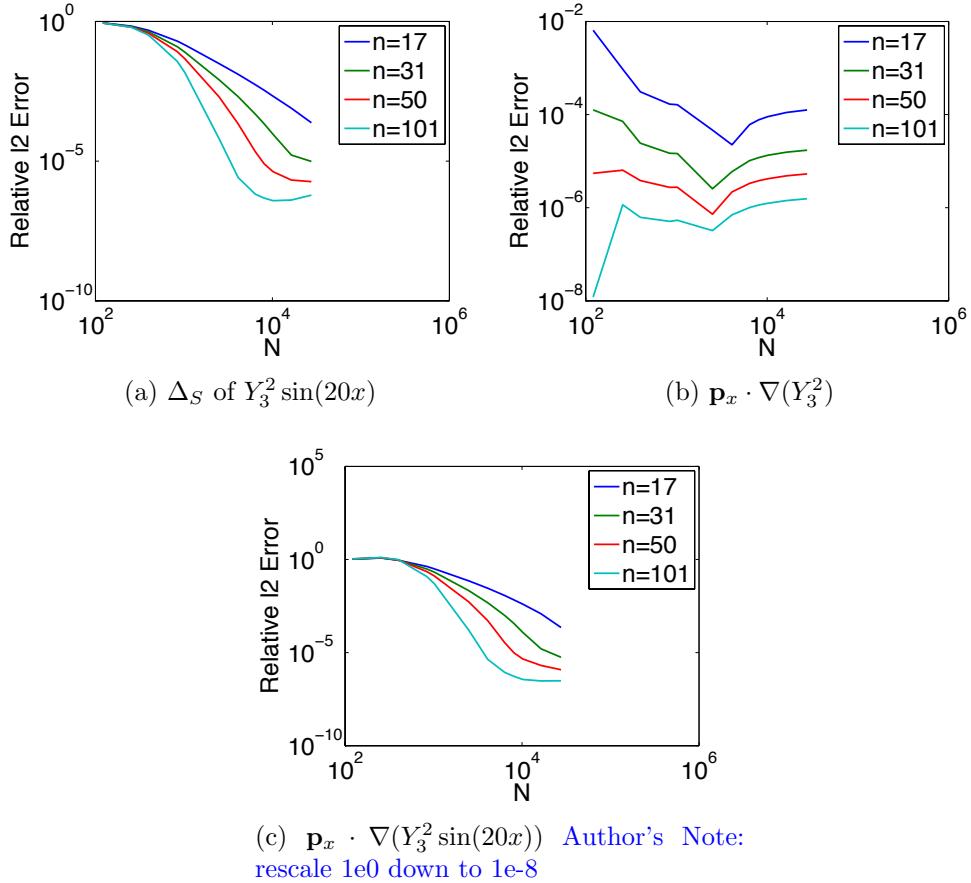


Figure B.2: Relative ℓ_2 error in differentiation.

Author's Note: Plot file size of matrix market file to give an idea of how much memory is saved by cutting out operators

With $N = 1e6$ nodes and stencil size $n = 101$, the matrix market file for weights is approximately 1.6 GB on disk. For a GPU with only 6 GB of global memory space available, it is worthwhile to consider possibilities for memory conservation.

Author's Note: Check to see $\nabla \cdot \nabla$ accuracy for heat equation.

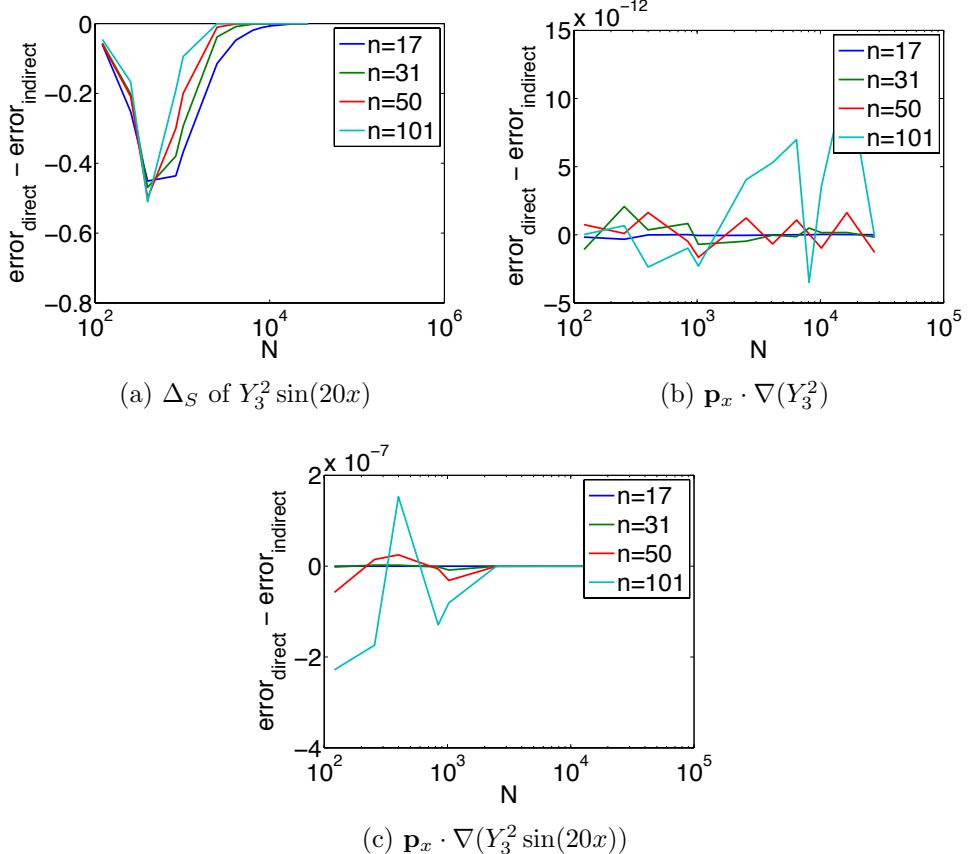


Figure B.3: Signed differences of relative ℓ_2 errors in differentiation between Direct and Indirect weights.

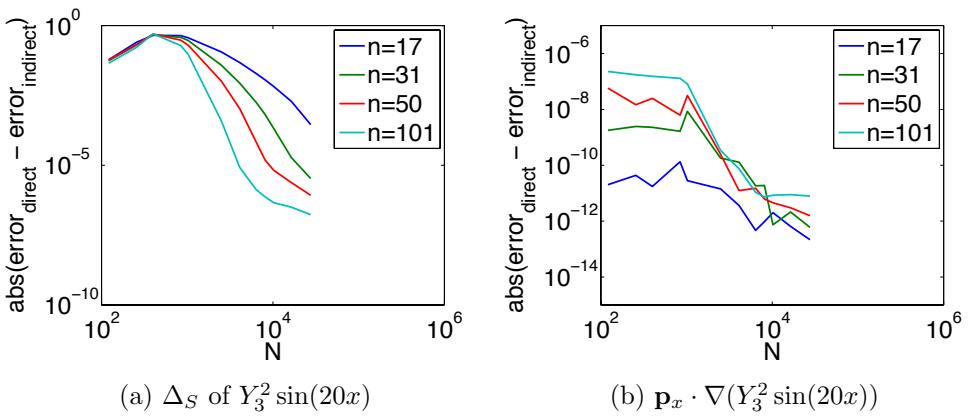


Figure B.4: Absolute differences of relative ℓ_2 errors in differentiation between Direct and Indirect weights.