

THE FLORIDA STATE UNIVERSITY
COLLEGE OF ARTS AND SCIENCE

MULTI-GPU SOLUTIONS OF GEOPHYSICAL PDES WITH RADIAL BASIS
FUNCTION-GENERATED FINITE DIFFERENCES

By

EVAN F. BOLLIG

A Dissertation submitted to the
Department of Scientific Computing
in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy

Degree Awarded:
Fall Semester, 2012

Evan F. Bollig defended this dissertation on October 1, 2012.

The members of the supervisory committee were:

Gordon Erlebacher
Professor Directing Thesis

Natasha Flyer
Outside University Representative

Mark Sussman
University Representative

Dennis Slice
Committee Member

Ming Ye
Committee Member

Janet Peterson
Committee Member

The Graduate School has verified and approved the above-named committee members, and certifies that the dissertation has been approved in accordance with the university requirements.

TABLE OF CONTENTS

List of Tables	vi
List of Figures	vii
1 Introduction	1
1.1 On Parallel/Distributed RBF-FD	4
1.2 On GPU RBF Methods	5
1.3 On Multi-GPU Methods	7
I Preliminaries	8
2 RBF Methods for PDEs	9
2.1 Survey of Related Work	10
2.1.1 Global RBF Methods	13
2.1.2 Compactly Support RBFs	15
2.1.3 Local RBF Methods	16
2.2 Comparison of RBF Methods	16
2.2.1 RBF Scattered Data Interpolation	17
2.2.2 Reconstructing Solutions for PDEs	18
2.2.3 PDE Methods	19
2.2.4 Local Methods	22
2.3 Recent Advances in Conditioning	22
II RBF-FD for HPC Environments	24
3 Introduction to RBF-FD	25
3.1 Multiple Operators	28
3.2 Differentiation Matrices and Sparse Matrix-Vector Multiply (SpMV)	28
3.3 Weight Operators	30
3.3.1 First and Second Derivatives ($\frac{1}{r} \frac{\partial \phi}{\partial r}, \frac{\partial^2 \phi}{\partial r^2}$)	31
3.3.2 Cartesian Gradient (∇)	31
3.3.3 Cartesian Laplacian (∇^2)	31
3.3.4 Laplace-Beltrami (Δ_S) on the Sphere	32
3.3.5 Constrained Gradient ($P_x \cdot \nabla$) on the Sphere	32
3.3.6 Hyperviscosity Δ^k for Stabilization	33

3.4	RBF-FD Implementation for Time-dependent PDEs	34
3.5	Grids	35
3.6	On Choosing the Right ϵ	38
4	An Alternative Stencil Generation Algorithm for RBF-FD	41
4.1	k -D Tree	42
4.2	A Fixed-Grid Algorithm	45
4.2.1	Fixed-grid Construction	47
4.2.2	Fixed-Grid Neighbor Query	49
4.3	Performance Comparison	51
4.3.1	Impact on SpMV	53
4.4	Alternative Orderings	55
4.4.1	Cuthill-McKee	56
4.5	Conclusion and Future Work	59
4.5.1	Future Work	59
4.6	Conclusions on Stencil Generation	60
5	GPU SpMV	61
5.1	Related Work	62
5.2	GPGPU	62
5.2.1	OpenCL	62
5.2.2	Hardware Layout	62
5.3	Performance	62
5.3.1	GFLOP Throughput	62
5.3.2	Expectations in Performance	63
5.4	Targeting the GPU	63
5.4.1	OpenCL	63
5.4.2	Naive Kernels	63
5.4.3	SpMV Formats/Kernels	63
5.5	Performance Comparison	63
5.5.1	Performance of Cosine CL vs VCL	63
5.5.2	VCL Formats Comparison	63
6	Distributed RBF-FD	65
6.1	Partitioning	65
6.2	Index Mappings and Local Node Ordering	68
6.3	Local node ordering	69
6.4	Test Case	72
6.5	Communication Collectives	72
6.5.1	Alltoally	73
6.5.2	Isend/Irecv	73
6.5.3	No Decode	74
6.5.4	Immediate Isend on Encode	74
6.6	CPU Scaling	74
6.6.1	Strong Scaling	74

6.6.2	Weak Scaling	74
6.6.3	Bandwidth	75
7	Distributed GPU SpMV	78
7.1	Overlapped Queues	78
7.2	Avoiding Copy Out	78
7.2.1	Avoiding Copy-Out on CPU	78
7.3	Scaling	78
7.3.1	Fermi	78
7.3.2	Kepler	78
7.3.3	Shared K20s	78
8	Numerical Validation	79
8.0.4	Vortex Rollup	79
8.0.5	Solid body rotation	81
8.1	Fragments (integrate above)	85
8.1.1	CFL	86
9	Performance Benchmarks	87
9.1	Metrics	87
9.2	OpenCL	87
9.2.1	OpenCL vs CUDA	87
9.2.2	Asynchronous Queueing	88
9.3	Fermi Architecture	88
9.3.1	Double Precision	88
9.3.2	Local Caching	88
9.3.3	Multiple Kernel Scheduling	88
9.3.4	Future NVidia Hardware	88
9.4	HPC Spear Cluster	89
9.5	Keeneland	89
9.6	Future Hardware	89
9.7	MPI_Alltoally	89
9.8	Asynchronous OpenCL	92
9.9	Multi-Queue OpenCL	92
9.10	GPU Kernel Optimizations	92
9.10.1	Work-Group Size and Number of Stencils	92
9.10.2	Parallel Reduction in Shared Memory	92
9.10.3	Comparison: custom SpMV for explicit schemes vs ViennaCL	92

LIST OF TABLES

2.1	Examples of frequently used RBFs based on [5?]. ε is the support parameter. All RBFs have global support. For compact support, enforce a cut-off radius (see Equation 2.1).	11
2.2	RBF interpolation types and properties, assuming a problem with N nodes.	13
2.3	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.	15
6.1	Sets defined for stencil distribution to multiple CPUs	70
8.1	Values for hyperviscosity and the RBF shape parameter ϵ for vortex roll-up test.	81
8.2	Values for hyperviscosity and RBF shape parameter for the cosine bell test.	83
9.1	Comparison of NVidia's new Fermi architecture to the GT200 architecture used for GTX 280, Tesla C1060 and other GPUs in use today.	89

LIST OF FIGURES

1.1	Commonly Used Radial Basis Functions (RBFs)	2
2.1	Example RBF shapes from Table 2.1 with parameter $\varepsilon = 1$	11
2.2	The Gaussian (GA) RBF (Table 2.1) with parameter $\varepsilon = 1$ and r in $D = 1, 2$ and 3	12
2.3	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.	12
3.1	Examples of stencils computable with RBF-FD	26
3.2	Differentiation matrix D_x is applied explicitly to calculate derivative approximations, $\frac{d}{dx} u(x)$	29
3.3	Quasi-regular nodes with $N = 4096$ maximum determinant (MD) node sets on the unit sphere.	37
3.4	$N = 2562$ icosahedral nodes on the unit sphere.	37
3.5	(Left) $N=100,000$ Spherical Centroidal Voronoi Tessellation nodes. (Right) Close-up of the same $N=100,000$ nodes to illustrate the irregularities in the grid.	38
3.6	Contours for ϵ as a function of \sqrt{N} for stencil sizes $n = 20, 40, 60, 80$ and 100 on the unit sphere. Contours assume near uniform distribution of nodes (e.g., maximum determinant (MD) nodes). Parameters superimposed above each contour provide coefficients for function $\epsilon(\sqrt{N}) = c_1\sqrt{N} - c_2$	40
4.1	A stencil center in green finds neighboring stencil nodes in blue. Two ball queries are shown as dashed and dash-dot circles to demonstrate the added difficulty of finding the right query radius to obtain the k -nearest neighbors.	42
4.2	An example k -D Tree in 2-Dimensions. Nodes are partitioned with a cyclic dimension splitting rule (i.e., splits occur first in X , then Y , then X , etc.); all splits occur at the median node in each dimension.	44

4.3	Two example space filling curves to linearize the same fixed-grid. Left: Raster-ordering (ijk); Right: Morton-/Z-ordering.	46
4.4	Example effects of node reordering for MD node set $N = 6400$ with $n = 50$. The differentiation matrices are permuted equivalents and roughly 0.78% full. Stencils generated based on k -D Tree maintain the original node ordering, while a fixed-grid with $h_n = 10$ condenses non-zeros for improved memory access patterns (i.e., cache reuse).	49
4.5	Querying the $n = 50$ nearest neighbors on a regular grid up to $N = 160^3$ demonstrates the gains achieved by the fixed-grid neighbor query method.	52
4.6	Fixed-grid speedup versus k -D Tree with stencil size $n = 50$	53
4.7	Fixed-grid speedup versus k -D Tree on a one-million node CVT (unit sphere), with stencil size $n = 50$	53
4.8	Fixed-grid impact on SpMV for stencil size $n = 50$	54
4.9	Example space filling curves used to reorder cells/nodes. The Raster, Z, X, U and 4-nodes per Edge are space filling curves applied to reorder cells of the fixed-grid stencil queries. Reverse Cuthill-McKee (RCM) operates on the output stencils to the associated adjacency graph. The RCM shown here is a special case with 3 neighbors per node.	55
4.10	An RCM reordering of nodes based on stencils $n = 5$ nearest neighbors per node.	56
4.11	In order: a) node ordering test cases; b) original ordering of regular grid (raster); c) coarse grid overlay for hash functions ($hn_x = 6$); d) example stencil ($n = 31$) spanning multiple Z's; e) spy of DM after orderings. 3) Spy impact on MD node set $N = 4096$, stencil size $n = 31$, $h_n = 6$	58
4.12	The impact of reordering on sparsity patterns for $N = 4096$ MD nodes on the sphere. Stencil size $n = 31$, fixed-grid resolution per dimension $h_n = 10$	59
6.1	Partitioning of $N = 10,201$ nodes to span four processors with stencil size $n = 31$	66
6.2	A simple adjacency graph and corresponding matrices. Edges connecting nodes of RBF-FD stencils produce (a) a directed adjacency matrix. To partition RBF-FD stencils, METIS requires conversion to (b) an undirected graph/matrix. 67	67
6.3	METIS partitioning of $N = 10,201$ nodes to span four processors with stencil size $n = 31$	68

6.4	Decomposition for one processor selects a subset of rows from the DM. Blocks corresponding to node sets $\mathcal{Q} \setminus \mathcal{B}$, \mathcal{O} , and \mathcal{R} are labeled for clarity. The subdomain for the processor is outlined by dashed lines.	68
6.5	Spy of the sub-DM view on processor 3 of 4 from a METIS partitioning of $N = 4096$ nodes with stencil size $n = 31$ and stencils generated with Algorithm ?? ($hnx = 100$). Blocks are highlighted to distinguish node sets $\mathcal{Q} \setminus \mathcal{B}$, \mathcal{B} , and \mathcal{R} . \mathcal{R} Stencils involved in MPI communications have been permuted to the bottom of the matrix. The split in \mathcal{R} indicates communication with two neighboring partitions.	69
6.6	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 6.1: $\mathcal{Q} \setminus \mathcal{B}$ is white, $\mathcal{B} \setminus \mathcal{O}$ is yellow, \mathcal{O} is green and \mathcal{R} is red.	71
6.7	The MPI_Alltoall collective allows processors to interchange transpose data by passing an equivalent number of bytes to every other processor.	73
6.8	The MPI_Alltoally collective compresses the interchange from MPI_Alltoall by allowing for variable message sizes between all processors. Assume message sizes are proportional to square size in figure. When packet sizes are null MPI_Alltoally has undefined behavior.	73
6.9	The “subset-to-subset” (MPI_Isend/MPI_Irecv) collective allows for variable message sizes, and truncates the number of connections between processors to only required connections.	74
6.10	Strong scaling the distributed SpMV for $N = 4096000$ nodes (i.e., a 160^3 regular grid) and various stencil sizes. Here the MPI_Alltoally collective operation is used. (Left) Strong scaling of SpMV (including cost of communication). (Center) Strong scaling of computation only. (Right) Strong scaling of communication only.	75
6.11	Scaling comparison of MPI_Alltoally and two types of MPI_Isend/MPI_Irecv collectives: one with MPI_Irecv issued after filling the MPI_Isend send buffer (post-fill), and the other issued before filling the MPI_Isend buffer (pre-fill).	76
6.12	Weak scaling of the SpMV	76
6.13	Scaling of SpMV with MPI_Isend/MPI_Irecv	77

8.1	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 $\gamma_c = 40$. Author's Note: color	81
8.2	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 $1.25(10^{-2})$. Author's Note: add initial condition figure	82
8.3	Convergence plot for vortex roll-up at $t = 3$. Author's Note: color	82
8.4	Eigenvalues of (8.4) 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. Author's Note: color	84
8.5	Cosine bell solution after 10 revolutions with $N = 10201$ nodes and stencil size $n = 101$. Hyperviscosity parameters are $k = 8$, $\gamma_c = 5(10^{-2})$. Author's Note: Insert color figures	84
8.6	Convergence plot for cosine bell advection. Normalized ℓ_2 error at 10 revolutions with hyperviscosity enabled.	85

LIST OF ALGORITHMS

3.1	A High-Level View of RBF-FD	35
4.1	BuildKDTree(P , $depth$)	43
4.2	KNNSearchKDTree(X_q , n , $root$, $depth$)	45
4.3	BuildFixedGrid(P , h_n)	48
4.4	QueryFixedGrid(X_q , n , P , Q)	50

CHAPTER 1

INTRODUCTION

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, numerical methods have been *meshed methods*, in that they rely on some underlying grid/lattice to connect discretized points in a well-defined manner. More recently a new class of methods surfaced, called *meshfree methods*, which discard the requirement for connectivity information and operate only on point clouds. Each class of methods offers numerous and often complementary benefits. This work focuses on a recent method that draws inspiration from both meshed and meshfree methods, called *Radial Basis Function-generated Finite Differences (RBF-FD)*.

Author's Note: [this paragraph does not fit well](#) 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., [?]). In addition to choosing nodes, meshed methods require connectivity/adjacency lists to form stencils (e.g., Finite Differences) or elements (e.g., Finite Element Method).

Well balanced meshes play a significant role in isolating nodes and allowing methods to be partitioned for distributed computing.

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 [?]. Also, it is often the case that methods have limited support for irregular node distributions (e.g., FD)

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 ([5]), a plethora of meshfree methods have arisen for the solution of partial differential equations (PDEs). For example, smoothed particle hydrodynamics, partition of unity method, element-free Galerkin method and others have been considered for fluid flow problems [?]. A good

survey of meshfree methods can be found in [?].

A subset of meshfree methods of particular interest to the numerical modeling 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 shown in Figure 1.1 (for corresponding equations refer to Table 2.1). 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.

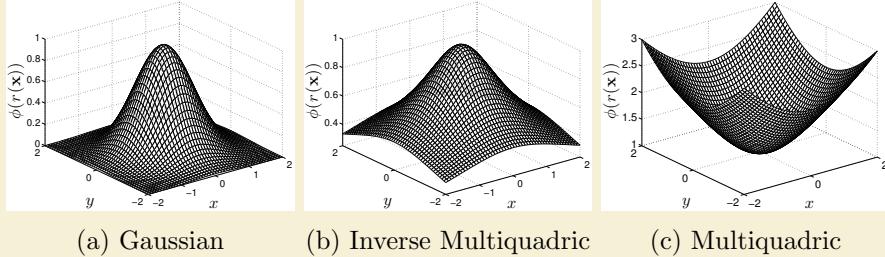


Figure 1.1: Commonly Used Radial Basis Functions (RBFs).

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 [?]. Solving PDEs with RBFs dates back to 1990 [? ?]. At the core of all RBF-based PDE 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 to generate stencil weights for finite-difference approximations of derivatives.

As “meshless” methods, RBF methods excel at solving problems that require geometric flexibility with scattered node layouts in d -dimensional space. They naturally extend into higher dimensions without significant increase in programming complexity [8?]. In addition to competitive accuracy and convergence compared with other state-of-the-art methods [6, 8, 9? ?], they also boast stability for large time steps.

While most of the literature surrounding RBFs for PDEs involves collocation (see Chapter 2), trends have shifted to place RBF-FD top of the list in the community today. RBF-FD is a hybrid of RBF scattered data interpolation and classical Finite Difference (FD). It shares many of the benefits from other RBF methods, like the ability to handle scattered node layouts in any dimension, as well as high-order accurate solutions. Likewise, RBF-FD exhibits many benefits akin to classical FD including compact stencils for localized derivative approximation, and low computational complexity.

The idea behind classical FD is to express derivatives at a single node (center) as a weighted combination/difference of solution values from a small neighborhood (i.e., a stencil) around the center. Common approximations such as upwind differencing, center differ-

encing, and other higher order approximations are of this form. In similar fashion, RBF-FD combines solutions values based on stencils, but it does so in a more generalized sense than standard FD. For example, classical FD is typically restricted to regular meshes and often symmetric stencils in practice with the same set of weights for each stencil. Weights can be derived from polynomial expansion and obtained in 1D by solving a Vandermonde interpolation matrix [10]. Higher dimension FD stencils are composed from combinations of 1D formulas applied to each dimension. This implies restrictions on the shape/layout of stencils. In contrast to this, RBF-FD is designed for stencils with irregular node placement and can easily provide a unique set of weights for each stencil with no restrictions on stencil shape.

The concept of RBF-FD was first introduced in 2000 [?], but it took another few years for the method to really get a start ([?], [?], [?] and [?]). Introduced over a decade ago, the method is only recently showing signs that it has obtained the critical-mass following necessary for the method's use in large-scale scientific models. At the onset of this work, most of the literature considered RBF-FD for problem sizes up to a few thousand nodes; at most tens of thousands of nodes. Similar to most RBF methods, RBF-FD is predominantly implemented within small-scale, serial computing environments. Under most circumstances the community at large continues investigation and extension development within MATLAB.

The goal of this dissertation is to scale RBF-FD solutions on high resolution meshes across high performance clusters, and to lead the way for its adoption within HPC and supercomputing circles. Chapter 6 focuses on the problem of distributing RBF-FD across independent compute nodes, and demonstrates the scalability of RBF-FD on over a thousand processors. As part of the push to HPC, leveraging Graphics Processing Units (GPUs) for computation is considered critical. GPUs, introduced in Chapter 5, are many-core accelerators capable of general purpose, embarrassingly parallel computations. Accelerators represent the latest trend in HPC, where compute nodes are commonly supplemented by one or more accessory boards for offload parallel tasks. Chapter 7 continues the discussion of RBF-FD on GPUs by tackling the problem of spanning a GPU cluster with an algorithm for overlapping communication and computation to hide the latency in data transfer between accelerators. Our effort leads the way for RBF-FD applicationss in an age when compute nodes with attached accelerator boards are considered key to breaching the exascale computing barrier [?].

The layout of this document is as follows. This chapter continues with a survey of work related on parallelizing RBF-FD, targeting the GPU, and spanning a multi-GPU cluster. Chapter 2 provides a historical survey of RBF methods as a backdrop to present RBF-FD in Chapter 3. Chapter 4 introduces a novel, fast algorithm for generating RBF-FD stencils as a faster alternative to the k -D Tree algorithm widely in use by the RBF community. In Chapter 6, the first scalable implementation of RBF-FD to span one thousand processors is described in detail. Chapter 5 continues with the challenge of offloading computation to GPUs, and Chapter 7 expands the discussion to a multi-GPU cluster. Chapter ?? verifies the parallel RBF-FD implementation with both explicit and implicit solutions to geophysical problems. Finally, this document concludes with a summary of results and discussion of future directions in Chapter ??.

Author's Note: [TODO: iterate through remainder of this chapter again](#)

1.1 On Parallel/Distributed RBF-FD

Parallel implementations of RBF methods rely on 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.

This work leverages a domain decomposition, but not for the purpose of conditioning. Instead the focus is on decomposing the domain in order to scale RBF-FD across more than a thousand CPU cores of an HPC cluster. Add to this the twist of incorporating a novel implementation on the GPU with overlapping communication and computation. This combination is unmatched in related work. However, RBF methods do have a bit of history of parallel implementations.

In 2007, Divo and Kassab [?] used a domain decomposition method with artificial subdomain boundaries for their implementation of a local collocation method [?]. The subdomains are processed independently, with derivative values at artificial boundary points 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.

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.

Kosec and Šarler [?] have the only known (to our knowledge) OpenMP implementation for RBFs. The authors parallelize coupled heat transfer and fluid flow problems on a single workstation. The application involves the local RBF collocation method, 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 further results from scaling tests were provided.

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

Perhaps the most competitive parallel implementation of RBFs is the PetRBF [?] branch of PETSc [?]. The authors of PetRBF (also developers for PETSc) have implemented a highly scalable, efficient RBF interpolation method based on compact RBFs (i.e., they operate on sparse matrices). The authors demonstrate efficient weak scaling of PetRBF across 1024 processes on a Blue Gene/L, and strong scaling up to 128 processes

on the same hardware. Additionally, strong scaling was tested on a Cray XT4. On the Blue Gene/L, PetRBF is demonstrated to achieve an impressive 74% parallel weak scaling efficiency on 1024 processes (operating on over 50 million points), and 84% strong scaling efficiency for 128 processes. For the Cray XT4, strong scaling tops out at 36% for 128 processes, a respectable number—and similar to observed results for our own code on 128 processes.

Parallelization of RBF-FD is achieved at two levels. First, the physical domain of the problem is partitioned into overlapping subdomains, each handled by a different MPI 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 (3.14) 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.

1.2 On GPU RBF Methods

With regard to the latter, there is some research on leveraging RBFs on GPUs in the fields of visualization [? ?], surface reconstruction [? ?], and neural networks [?].

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.

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.

Related work on RBFs and GPUs is sparse. In 2009, Schmidt et al. [? ?] implemented a global RBF method for Tsunami simulation on the GPU using the AccelerEyes Jacket [?] add-on for MATLAB. Jacket provides a MATLAB interface to data structures and routines that internally call to the NVidia CUDA API. Their model was based on a single large dense matrix solve, and with the help of Jacket the authors were able to achieve approximately 7x speedup over the standard MATLAB solution on the then current generation of the MacBook Pro laptop. The authors compared the laptop CPU (processor details not specified) to the built-in NVidia GeForce 8600M GT GPU. Schmidt et al.'s implementation was the first contribution to the RBF community to leverage accelerators. The results were significant and promising, but no further contributions were made on the topic.

While both Schmidt et al.'s method and the method presented here are based on RBFs, the two problems are only distantly related when it comes to implementation on the GPU. Dense matrix operations have a high computational complexity, are considered ideal (or near to) by linear algebra libraries like BLAS [?] and LAPACK [?], and were demonstrated to fit well on GPUs from the onset of General Purpose GPU (GPGPU) Computing. In fact, NVidia included CUBLAS [?] (a GPU based BLAS library for their hardware) with their initial public release of the game-changing CUDA development kit in 2006. In stark contrast to this, sparse matrix operations have minimal computational complexity and are less than ideal for the GPU.

Earlier this year (2013), Cuomo et al. [?] implemented RBF-interpolation on the GPU for surface reconstruction. Their implementation utilizes PetRBF [?], and new built-in extensions that allow GPU access within PETSc. PETSc internally wraps the CUSP project [?] for sparse matrix algebra on the GPU. With the help of these libraries, Cuomo et al. solve and apply sparse interpolation systems on the GPU for up to three million nodes on an NVidia Fermi C1060 GPU (4GB). They compare results to a single core CPU implementation on an Intel i7-940 CPU and demonstrate that the GPU accelerate their solutions between 6x and 25x. Unfortunately, the authors do not show evidence of scaling the interpolation across multiple GPUs; so while evidence exists that PetRBF now has full GPU support, it remains to be seen how well the code can scale in GPU mode.

1.3 On Multi-GPU Methods

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)
Thibault et al. have multiple works on Multi-GPU and overlapping comm and comp.

Part I

Preliminaries

CHAPTER 2

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 to generate weights for finite-differencing stencils, utilizing the weights 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 [?].

As “meshless” methods, RBF methods excel at solving problems that require geometric flexibility with scattered node layouts in d -dimensional space. They naturally extend into higher dimensions without significant increase in programming complexity [8?]. In addition to competitive accuracy and convergence compared with other state-of-the-art methods [6, 8, 9? ?], they also boast stability for large time steps.

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 brief introduction to RBFs and a historical survey, related methods are into classified into three types: global, compact, and local methods. Following this, the general approximation problem is introduced, with a look at the core of all three method classifications: RBF scattered-data interpolation.

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 [5?]. 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 degrades 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 can lead to spectral (exponential) convergence of the RBF interpolant for smooth data [?].

Three global RBF collocation methods are presented here: Kansa’s method, Fasshauer’s method and Direct collocation. Additionally, the RBF-pseudospectral (RBF-PS) method is shown as an extension to fit global RBF methods into the framework of lower complexity pseudo-spectral methods.

This survey of RBF PDE methods frames the context in which RBF-FD was developed, and illustrates both the benefits and pitfalls inherited from its predecessors.

2.1 Survey of Related Work

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}^d , so they easily extend into higher dimensions without significant change in programming complexity. Examples of commonly used RBFs from the literature are provided in Table 2.1; 2D representations of the same functions can be found in Figure 2.1. Figure 2.2 illustrates the radial symmetry of RBFs—in this case, a Gaussian RBF—in the first three dimensions.

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 d -dimensional space. The interpolation problem—referred to as *RBF scattered data interpolation*—seeks the unknown coefficients, $\mathbf{c} = \{c_j\}$, that satisfy:

$$\sum_{j=1}^N \phi_j(r(\mathbf{x})) c_j = f(\mathbf{x}),$$

where $\phi_j(r(\mathbf{x}))$ is an RBF centered at $\{\mathbf{x}_j\}_{j=1}^n$. In theory the radial coordinate, $r(\mathbf{x})$, could be any distance metric, but is most often assumed to be $r(\mathbf{x}) = \|\mathbf{x} - \mathbf{x}_j\|_2$ (i.e., Euclidean distance), as it is here. The coefficients \mathbf{c} result in a smooth interpolant that collocates sample values $f(\mathbf{x}_j)$. An example of RBF interpolation in 2D using 15 Gaussians is shown in Figure 2.3.

RBFs have been shown in some cases to have exponential convergence for function approximation [5]. 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 [?]. 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 [5?] for details).

RBFs depend on a shape or support parameter ϵ that controls the width of the function. The functional form of the shape function becomes $\phi(\epsilon r(\mathbf{x}))$. For simplicity in what follows, the notation $\phi_j(\mathbf{x})$ implies $\phi(\epsilon \|\mathbf{x} - \mathbf{x}_j\|_2)$. Decreasing ϵ increases the support of the RBF and

Name	Abbrev.	Formula	Order (m)
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 2.1: Examples of frequently used RBFs based on [5?]. ε is the support parameter. All RBFs have global support. For compact support, enforce a cut-off radius (see Equation 2.1).

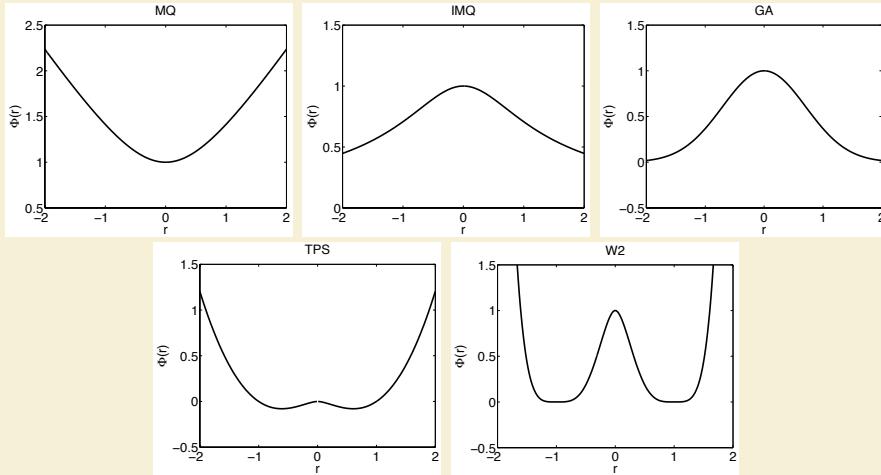


Figure 2.1: Example RBF shapes from Table 2.1 with parameter $\varepsilon = 1$.

in most cases, the accuracy of the interpolation, but worsens the conditioning of the RBF interpolation problem [?]. This inverse relationship is widely known as the *Uncertainty Relation* [? ?]. 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., $\varepsilon \rightarrow 0$) where high accuracy has been observed [? ?].

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 2.1 have been applied in literature, but for PDEs in particular, few can rival the attention received by multiquadratics. Recently, however, Gaussian RBFs are on the rise due to recent advances in eigenvalue stabilization and new methods for investigating the $\varepsilon \rightarrow 0$ regime (see e.g., [10?]).

By 1990, the understanding of the scientific community regarding RBFs was sufficiently

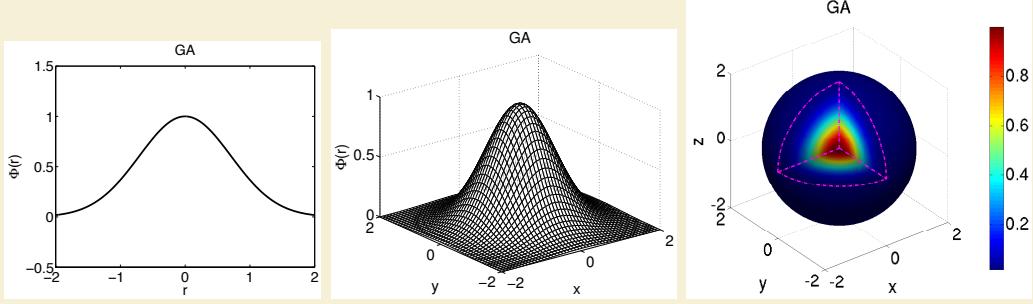


Figure 2.2: The Gaussian (GA) RBF (Table 2.1) with parameter $\epsilon = 1$ and r in $D = 1, 2$ and 3 .

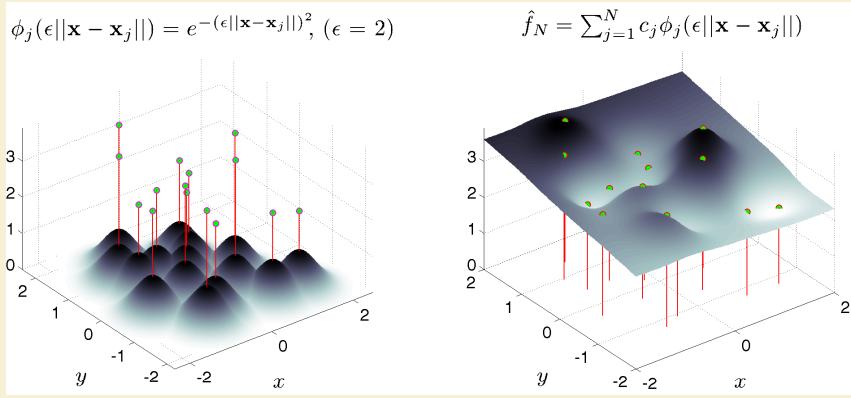


Figure 2.3: RBF interpolation using 15 translates of the Gaussian RBF with $\epsilon = 2$. One RBF is centered at each node in the domain. Linear combinations of these produce an interpolant over the domain passing through known function values.

developed for collocating PDEs [? ?]. PDE collocation seeks a solution of the form

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

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:

$$u(x) = \sum_{i=1}^N \phi_j(x) c_j$$

As in the problem of RBF scattered data interpolation, $\mathbf{c} = \{c_j\}$ is the unknown coefficient vector. Under the assumption that \mathcal{L} is a linear operator, one can collocate the differential equation. Alternatively, individual derivative operators can be expressed as linear combinations of the unknowns u_j (leading to the RBF-FD methods). In all cases, a linear system of equations arises, with different degrees of sparsity, dependent on the chosen basis functions and how the various constraints are enforced. While $\phi_j(x)$ is restricted to RBFs in

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

this context, note that spectral methods, finite-element or spectral-element methods can be formulated in similar fashion with alternative basis functions. Of course, u can be a vector of unknown variables (\mathbf{c} then becomes a matrix).

Table 2.3 classifies references according to their choice of collocation method and RBF interpolation type. There are three main categories of RBF interpolation listed in Table 2.2. 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. The final row of Table 2.3 considers literature on the RBF-FD method and is discussed in depth in Chapter 3.

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 [?].

We now turn to discussion of the benefits and shortcomings of each RBF method, before covering derivation of the methods.

2.1.1 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 method. 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. It was shown in [? ?] that the unsymmetric linear system produced by Kansa's method does not guarantee non-singularity; although it is also 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 [5]. 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 collocation 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.

Both Kansa’s method and Fasshauer’s methods were shown in [?] to fit well in the generalized framework of pseudo-spectral methods with a subtle change in algorithm. While collocation methods explicitly compute the coefficients for a continuous derivative approximation, their alternates, referred to in literature as RBF-pseudospectral (RBF-PS) methods, never explicitly compute the interpolant coefficients. Instead, a differentiation matrix (DM) is assembled and used to approximate derivates at the collocation points only [?]. Since most computational models are simply concerned with the solution at collocation points, the change to assemble DMs as in RBF-PS is organic.

Following the evolution of the RBF-PS algorithm, applications of global RBFs in the classic collocation sense (i.e., without the RBF-PS DMs) become impractical. This statement stems from the algorithmic complexity of each method. Global RBF methods result in full matrices [5]. The global collocation methods then scale on the order of $O(N^3)$ floating point operations (FLOPs) to solve for weighting coefficients on a given node layout, plus $O(N^2)$ to apply the weights for derivatives. If time-stepping is required, global collocation methods must recompute the time-dependent coefficients with additional cost dominated by $O(N^3)$ operations. RBF-PS methods have similar requirements for $O(N^3)$ operations to assemble the differentiation matrix and $O(N^2)$ to apply for derivatives. However, by avoiding time-dependent coefficients, the differentiation matrix application at each time-step is only $O(N^2)$ operations. As an aside, the $O(N^3)$ complexity for each method—typically due to an LU-decomposition, with subsequent forward- and back-solves—could be reduced. While not in mainstream use by the RBF community, [?] correctly points out that iterative solvers could be employed for $O(N^2)$ complexity.

Hon et al. [?] employed Kansa’s method to solve shallow water equations for Typhoon simulation. In [9], Flyer and Wright employed RBF-PS (Kansa method) for the solution of shallow water equations on a sphere. Their results show that RBFs allow for longer time steps with spectral accuracy. The survey [?] by Flyer and Fornberg showcases RBF-PS (Kansa’s method) out-performing some of the best available methods in geosciences, namely: Finite Volume, Spectral Elements, Double Fourier, and Spherical Harmonics. When applied to problems such as transport on the sphere [8], shallow water equations [9], and 3D mantle convection [?], RBF-PS consistently required fewer time steps, and a fraction of the nodes for similar accuracy [?].

	RBF Interpolation Type		
Method	Global (Dense)	Compact (Sparse Global)	Local
Kansa's Method	[8, 9? ? ? ? ? ? ? ? ? ? ? ? ? ?]	[? ?]	[? ? ? ?]
Fasshauer's Method	[? ? ?]	[?]	[? ? ?]
Direct Collocation	[? ?]		[6, 7, 10? ? ?]
RBF-FD	N/A	N/A	? ? ? ? ? ?]

Table 2.3: 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.

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

$$\phi(r) = \begin{cases} \varphi(r) & r \in [0, 1] \\ 0 & r > 1 \end{cases} \quad (2.1)$$

where a cut-off radius is defined past which the RBF (in this case $\varphi(r)$) 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 with smaller memory requirements [5]. The actual complexity estimate of the CSRBF method depends on the sparsity of the problem as well as the ordering of the assembled system. Assuming $n \ll N$ where n represents the number of nodes in support, [?] lists the complexity as dominated by $O(N)$ for properly structured systems within MATLAB, and the investigation in [?] found $O(N^{1.5})$ consistent with the estimate provided by their choice of general sparse solver package. A multi-level CSRBF method, introduced by Fasshauer [5], collocates solutions over multiple grid refinements to achieve reduced $O(N)$ complexity, but the method is plagued by poor convergence. It is also worth noting that in the context of CSRBFS, analogues to Kansa's method and Fasshauer's method are known by the names *radial point interpolation method (RPIM)* [?] and *radial point interpolation collocation method (RPICM)* [?], respectively. A more thorough survey of CSRBF history can be found in [5?].

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., [? ? ?]).

2.1.3 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 less conditioning issues. 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 [? ? ?].

2.2 Comparison of RBF Methods

We now detail RBF methods for PDEs leading up to the derivation of RBF-FD.

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

$$\begin{aligned}\mathcal{L}u &= f && \text{on } \Omega \\ u &= g && \text{on } \Gamma\end{aligned}$$

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 $\{P_l(x)\}_{l=1}^M$:

$$u_\phi(x) = \sum_{j=1}^N \phi_j(x) c_j + \sum_{l=1}^M P_l(x) d_l, \quad P_l(x) \in \Pi_p^d \quad (2.2)$$

where $\phi_j(x) = \|x - x_j\|_2$ (Euclidean distance). The second sum represents a linear combination of polynomials that enforces zero approximation error when $u(x)$ is a polynomial of degree less 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 2.1) [?]. Note that Equation 2.2 is evaluated at $\{x_j\}_{j=1}^N$ data points through which the interpolant is required to pass with zero residual. The x_j 's are known as *collocation points* (a.k.a. trial points), taken as the RBF centers. The test points, x , usually coincide with collocation points, although this is not a requirement.

To clarify the role of the polynomial part in Equation 2.2, it is necessary to put aside the PDE for the moment and consider only the problem of *scattered data interpolation* with Radial Basis Functions.

2.2.1 RBF Scattered Data Interpolation

Borrowing notation from [5?], we seek an interpolant of the form

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

where $f(x)$ is expressed as a scalar product between the unknown coefficient weights c_j and the radial basis functions $\phi_j(x)$.

To obtain the unknown coefficients, c_j , form a linear system in terms of the N RBF centers:

$$\begin{aligned} f(x) &= \sum_{j=1}^N c_j \phi_j(x) \quad \text{for } x = \{x_j\}_{j=1}^N \\ (\mathbf{f}) &= [\phi] (\mathbf{c}) \end{aligned}$$

The invertibility of this system depends on the choice of RBF, so one typically chooses a function that is positive definite to avoid issues. It has been shown (see [5?]) 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 be *conditionally positive definite*. In [5], Fasshauer demonstrates that conditional positive definiteness is guaranteed when Equation 2.2 exactly reproduces functions of degree less than or equal 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 \mathbb{R}^d , $M = \binom{m+d}{d}$ [?], giving

$$\begin{aligned} \sum_{j=1}^N c_j \phi_j(x) + \sum_{l=1}^M d_l P_l(x) &= f(x), \quad P_l(x) \in \Pi_m^d \\ [\phi \ P] \begin{pmatrix} \mathbf{c} \\ \mathbf{d} \end{pmatrix} &= (\mathbf{f}) \end{aligned} \tag{2.3}$$

where the second summation (referred to as *interpolation conditions* [?]) ensures the minimum degree of the interpolant. Refer to Table 2.1 for a short list of recommended RBFs and minimally required orders of m . This document prefers the Gaussian RBF. Notice, in Equation 2.3, that the interpolation conditions add M new degrees of freedom, so M additional constraints are necessary to square the system. In this case:

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

or

$$P^T \mathbf{c} = 0. \tag{2.4}$$

It is now possible again to write the interpolation problem as a complete linear system using Equations 2.3 and 2.4:

$$\underbrace{\begin{bmatrix} \phi & P \\ P^T & 0 \end{bmatrix}}_A \begin{pmatrix} c \\ d \end{pmatrix} = \begin{pmatrix} f \\ 0 \end{pmatrix} \quad (2.5)$$

Equation 2.5—typically a dense system except in the case of RBFs with compact support—can be solved efficiently via standard methods like LU-decomposition. With the coefficients, the interpolant can be sampled at any test points, $\{x_i\}_{i=1}^n$, by substitution into Equation 2.3:

$$\begin{aligned} f(x_i) &= \sum_{j=1}^N c_j \phi_j(x_i) + \sum_{l=1}^M d_l P_l(x_i) \\ &= \underbrace{\begin{bmatrix} \phi & P \end{bmatrix}}_B \begin{pmatrix} c \\ d \end{pmatrix} \Big|_{x=x_i} \end{aligned} \quad (2.6)$$

2.2.2 Reconstructing Solutions for PDEs

In the next few subsections, collocation equations are considered based on this general form:

$$\begin{aligned} \mathcal{L}u_\phi(x) &= f(x) && \text{on } \Omega \\ \mathcal{B}u_\phi(x) &= g(x) && \text{on } \Gamma \end{aligned}$$

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., $\frac{\partial}{\partial x}$, ∇ , ∇^2) and $\mathcal{B} = I$ (i.e. identity operator for Dirichlet boundary conditions) for PDEs. For RBF scattered data interpolation, $\mathcal{L} = I$. 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 that follow a linear system is generated:

$$\begin{aligned} A_{\mathcal{L}} \begin{pmatrix} c \\ d \end{pmatrix} &= \begin{pmatrix} f \\ 0 \end{pmatrix} \\ \begin{pmatrix} c \\ d \end{pmatrix} &= A_{\mathcal{L}}^{-1} \begin{pmatrix} f \\ 0 \end{pmatrix} \end{aligned}$$

where matrix $A_{\mathcal{L}}$ depends on the choice of collocation method.

Once the linear system is solved, the value $u(x)$ is reconstructed at the test points following Equation 2.6:

$$\begin{aligned} u(x) &\approx \left[\begin{array}{cc} \phi & P \end{array} \right] \begin{pmatrix} c \\ d \end{pmatrix} \Big|_{x=x_i} \\ &\approx BA_{\mathcal{L}}^{-1} \begin{pmatrix} f \\ 0 \end{pmatrix} \end{aligned} \quad (2.7)$$

Likewise, to obtain differential quantities,

$$\begin{aligned}\mathcal{L}u(x) &\approx [\phi_{\mathcal{L}} \quad P_{\mathcal{L}}] \begin{pmatrix} c \\ d \end{pmatrix} \Big|_{x=x_i} \\ &\approx B_{\mathcal{L}} A_{\mathcal{L}}^{-1} \begin{pmatrix} f \\ 0 \end{pmatrix}\end{aligned}$$

2.2.3 PDE Methods

Now, since $u_{\phi}(x)$ from Equation 2.2 cannot (in general) satisfy the PDE everywhere, the PDE is enforced 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. Also, for each of the methods the choice of RBF can be either global, resulting in a large dense system, or compact, resulting in a large sparse system.

Kansa's Method. The first global RBF method for PDEs, *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 2.2. The resulting system is given by [?]; assuming that \mathcal{L} is a linear operator,

$$\mathcal{L}u_{\phi}(x_i) = \sum_{j=1}^N c_j \mathcal{L}\phi_j(x_i) + \sum_{l=1}^M d_l \mathcal{L}P_l(x_i) = f(x_i) \quad i = 1, \dots, n_I \quad (2.8)$$

$$\mathcal{B}u_{\phi}(x_i) = \sum_{j=1}^N c_j \mathcal{B}\phi_j(x_i) + \sum_{l=1}^M d_l \mathcal{B}P_l(x_i) = g(x_i) \quad i = n_I + 1, \dots, n \quad (2.9)$$

$$\sum_{j=1}^N c_j P_l(x_j) = 0 \quad l = 1, \dots, M \quad (2.10)$$

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 [5] for RBF derivative tables); the same is true for the polynomials P_l .

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

$$\underbrace{\begin{bmatrix} \phi_{\mathcal{L}} & P_{\mathcal{L}} \\ \phi_{\mathcal{B}} & P_{\mathcal{B}} \\ P^T & 0 \end{bmatrix}}_{A_{\mathcal{L}}} \begin{pmatrix} c \\ d \end{pmatrix} = \begin{pmatrix} f \\ g \\ 0 \end{pmatrix} \quad (2.11)$$

where $\phi_{\mathcal{L}} = \mathcal{L}\phi$, $P_{\mathcal{L}} = \mathcal{L}P$ are the interior components (Equation 2.8), $\phi_{\mathcal{B}}$ and $P_{\mathcal{B}}$ are the boundary components (Equation 2.9), and $P^T = [P_{\mathcal{L}}^T \quad P_{\mathcal{B}}^T]$ are constraints for both interior

and boundary polynomial parts (Equation 2.10). From Equation 2.11 it should be clear why Kansa's method is also known as the *Unsymmetric* collocation method.

Recall that the matrix in Equation 2.11 has no guarantee of non-singularity [?]; however, singularities are rare in practice [?].

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:

$$u_\phi(x_i) = \sum_{j=1}^{N_I} c_j \mathcal{L}\phi_j(x_i) + \sum_{j=N_I+1}^N c_j \mathcal{B}\phi_j(x_i) + \sum_{l=1}^M d_l P_l(x_i) \quad (2.12)$$

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 2.2, 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.

Substituting Equation 2.12 into Equations 2.8-2.10 gives

$$\begin{aligned} \sum_{j=1}^{N_I} c_j \mathcal{L}^2 \phi_j(x_i) + \sum_{j=N_I+1}^N c_j \mathcal{L}\mathcal{B}\phi_j(x_i) + \sum_{l=1}^M d_l \mathcal{L}P_l(x_i) &= f(x_i) \quad i = 1, \dots, n_I \\ \sum_{j=1}^{N_I} c_j \mathcal{B}\mathcal{L}\phi_j(x_i) + \sum_{j=N_I+1}^N c_j \mathcal{B}^2 \phi_j(x_i) + \sum_{l=1}^M d_l \mathcal{B}P_l(x_i) &= g(x_i) \quad i = n_I + 1, \dots, n \\ \sum_{j=1}^{N_I} c_j \mathcal{L}P_l(x_j) + \sum_{j=N_I+1}^N c_j \mathcal{B}P_l(x_j) &= 0 \quad l = 1, \dots, M \end{aligned} \quad (2.13)$$

which becomes the following:

$$\underbrace{\begin{bmatrix} \phi_{\mathcal{L}\mathcal{L}} & \phi_{\mathcal{L}\mathcal{B}} & P_{\mathcal{L}} \\ \phi_{\mathcal{B}\mathcal{L}} & \phi_{\mathcal{B}\mathcal{B}} & P_{\mathcal{B}} \\ P_{\mathcal{L}}^T & P_{\mathcal{B}}^T & 0 \end{bmatrix}}_{A_{\mathcal{L}}} \begin{pmatrix} c \\ d \end{pmatrix} = \begin{pmatrix} f \\ g \\ 0 \end{pmatrix} \quad (2.14)$$

Note that $\phi_{\mathcal{L}\mathcal{L}}$ represents the first summation in Equation 2.13.

The symmetry of Fasshauer's (*symmetric collocation*) method is apparent in Equation 2.14. 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.

Direct Collocation. In *Direct collocation* (see [? ?]), the interpolant is chosen as Equation 2.2 (the same as Kansa's method). However, the Direct method collocates both

the interior and boundary operators at the boundary points:

$$\begin{aligned} \sum_{j=1}^N c_j \mathcal{L} \phi_j(x_i) + \sum_{l=1}^M d_l \mathcal{L} P_l(x_i) &= f(x_i) \quad i = 1, \dots, n \\ \sum_{j=1}^N c_j \mathcal{B} \phi_j(x_i) + \sum_{l=1}^M d_l \mathcal{B} P_l(x_i) &= g(x_i) \quad i = 1, \dots, n_B = n - n_I \\ \sum_{j=1}^N c_j P_l(x_j) &= 0 \quad l = 1, \dots, M \end{aligned} \quad (2.15)$$

Reformulating as a linear system provides:

$$\begin{bmatrix} \phi_{\mathcal{L}} & P_{\mathcal{L}} \\ \phi_{\mathcal{B}} & P_{\mathcal{B}} \\ P^T & 0 \end{bmatrix} \begin{pmatrix} c \\ d \end{pmatrix} = \begin{pmatrix} f \\ g \\ 0 \end{pmatrix} \quad (2.16)$$

While the final system in Equation 2.16 is structured the same as Kansa's method (Equation 2.11), careful inspection of the index i in Equations 2.8 and 2.15 reveals that Direct collocation produces a larger system.

RBF-PS. The extension of global collocation to traditional pseudo-spectral form was introduced by Fasshauer in [?]. Dubbed RBF-PS, the method utilizes the same logic from Kansa's and Fasshauer's collocation methods to form matrix $A_{\mathcal{L}}$ (i.e., $A_{\mathcal{L}}$ can be either Equation 2.11 or 2.14). However, RBF-PS subtly assumes the solution, $u(x)$, is only required at collocation points (i.e., $\{x_i\} = \{x_c\}$) [5?]. Then, extending Equation 2.7, RBF-PS gives:

$$\begin{aligned} u(x) &= (BA_{\mathcal{L}}^{-1}) \begin{pmatrix} f \\ 0 \end{pmatrix} \\ &= D_{\mathcal{L}}^T \begin{pmatrix} f \\ 0 \end{pmatrix}. \end{aligned} \quad (2.17)$$

where $D_{\mathcal{L}}$ is a discrete differentiation matrix (DM) for the operator \mathcal{L} . Here, $D_{\mathcal{L}}$ is independent of the function $f(x)$ and is assembled by solving the system:

$$D_{\mathcal{L}} = A_{\mathcal{L}}^{-T} B^T \quad (2.18)$$

An LU-decomposition ($O(N^3)$) in preprocessing with forward- and back-solves ($O(N^2)$) are fitting to efficiently solve the multiple RHS system[5?].

Since matrix $D_{\mathcal{L}}$ is independent of functions $u(x)$ and $f(x)$, the matrix requires update only if the RBF centers move—a compelling benefit for time-dependent problems on stationary nodes. The complexity of RBF-PS for time-dependent solutions is then reduced to a matrix-vector multiply ($O(N^2)$) for each time-step. In contrast, classic RBF collocation methods also construct LU factors of $A_{\mathcal{L}}^{-1}$ in preprocessing, but delay application of forward- and back-solves to acquire time-dependent weighting coefficients at each time-step. This is then followed by the pre-multiply of B (i.e., additional $O(N^2)$) to complete the time-step.

2.2.4 Local Methods

Another trend in RBF methods is to 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 for RBF support, 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:

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

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.

This results in a linear system with similar structure to the global collocation problem, but the dimensions are much smaller:

$$\underbrace{\begin{bmatrix} \phi_{\mathcal{L}\mathcal{L}} & \phi_{\mathcal{L}\mathcal{B}} & P_{\mathcal{L}} \\ \phi_{\mathcal{B}\mathcal{L}} & \phi_{\mathcal{B}\mathcal{B}} & P_{\mathcal{B}} \\ P_{\mathcal{L}}^T & P_{\mathcal{B}}^T & 0 \end{bmatrix}}_{A_{\mathcal{L}}} \begin{pmatrix} c^{(k)} \\ d^{(k)} \end{pmatrix} = \begin{pmatrix} f \\ g \\ 0 \end{pmatrix} \quad (2.19)$$

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 2.8 is limited to an inner product for each center rather than the matrix-vector grouping possible with global RBFs. This approach decomposes the problem into smaller and more manageable parts. However, because the interpolants are local, there is no notion of global continuity/smoothness of the solution.

2.3 Recent Advances in Conditioning

The most limiting factor in the success of RBF methods has not been the complexity of the methods, nor the task of approximating derivatives. Rather, it is the support parameter, ϵ , and the dilemma one faces in the *Uncertainty Relation* [?]. Recall that as $\epsilon \rightarrow 0$, ill-conditioning of the RBF interpolation matrices increases, but so too does the approximation accuracy—that is, assuming a stable solution can be found. Likewise, as

the number of collocation points increases, the range of ϵ for which the linear system has acceptable conditioning narrows. In [?], the authors observe that much of the literature on RBF methods seek to find optimal values of the support parameter ϵ for the highest accuracy in applications. Occasionally the optimal values lie within a range of acceptable conditioning to solve the linear systems directly (a.k.a. RBF-Direct solutions). More often, one must compromise between the accuracy loss for large ϵ and accuracy loss in RBF-Direct solutions due to lower values of ϵ . Many attempts to express the optimal ϵ as a function of problem size have also been thwarted as the impact on the optimal ϵ values in the face of small node perturbations is still not fully understood. This makes refinements a challenge to manage.

In an effort to overcome limitations due to conditioning, Fornberg and Wright [?] presented the *Contour-Padé* algorithm, which allows for numerically stable computation of highly accurate interpolants with nearly flat RBFs (i.e., $\epsilon \rightarrow 0$). 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. The Contour-Padé algorithm is not overly competitive due to the fact that it only supports fewer than a hundred in 2D and slightly more in 3D [?].

The *RBF-QR* method, was later introduced by Fornberg and Piret [?] in context of a sphere to let $\epsilon \rightarrow 0$ for a few thousand nodes. It was later extended to general 1D, 2D and 3D problems in [?]. The RBF-QR method uses a truncated expansion of RBFs in terms of spherical harmonics or Chebyshev polynomials and leverages QR factorization to create a new well-conditioned set of basis functions to reproduce the original RBF space. The well-conditioned basis set allows stable solution independent of the value ϵ . The cost of the method is demonstrated to increase as ϵ increases. Benchmarks in [?] show that double precision RBF-QR is between 3x-7x slower than RBF-Direct for the same values of ϵ . Fornberg, Larsson and Flyer [?] successfully implemented the 2D method in less than 100 lines of MATLAB code and apply RBF-QR to problems with 6000 quasi-uniform nodes and globally supported RBFs.

Between Contour-Padé and RBF-QR, global RBF methods overcame many conditioning issues for small to mid-sized problems. The lack of support for large problem sizes is discouraging, but it leads to an argument in favor of local methods like RBF-FD, which decrease the problem size to fit nicely within the scope of stable methods.

Most recently, Fornberg et al. [?] introduced a new method called RBF-GA, which performs a similar change of basis as RBF-QR, but the method avoids truncated infinite expansions by expressing the new basis functions in terms of an incomplete Gamma function. Unlike RBF-QR, this method is limited to Gaussian basis functions only. Benchmarks provided in [?] rank stable methods for RBFs from fastest to slowest as: RBF-Direct, RBF-GA, and then RBF-QR. RBF-GA is also effective for a small number of nodes: a few hundred in 2D, and at least 500 in 3D. Unlike RBF-QR, which performs a change of basis on the interpolating matrix only, the RBF-GA method requires a complicated change of basis for the RHS as well [?].

Part II

RBF-FD for HPC Environments

CHAPTER 3

INTRODUCTION TO RBF-FD

RBF-generated Finite Differences (RBF-FD) were first introduced by Tolstykh in 2000 [?], but it was the simultaneous, yet independent, efforts in [?], [?], [?] and [?] that gave the method its real start.

The RBF-FD method is similar in concept to classical finite-differences (FD), in that derivatives of a function $u(x)$ are approximated by weighted combinations of n function values in a small neighborhood around a single *center* node, x_c . That is:

$$\mathcal{L}u(x) \Big|_{x=x_c} \approx \sum_{j=1}^n c_j u(x_j) \quad (3.1)$$

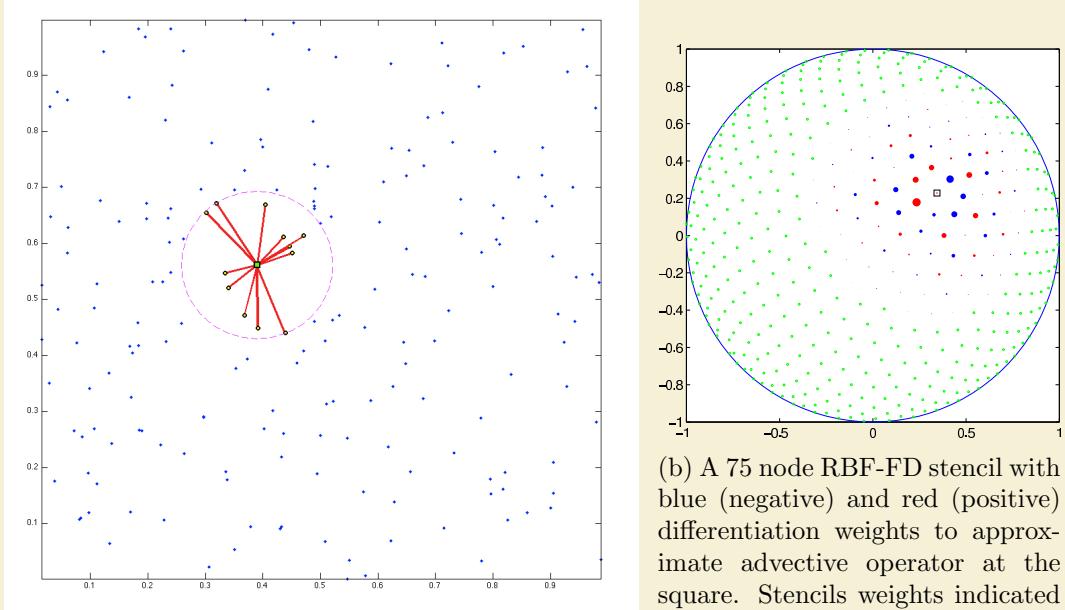
where $\mathcal{L}u$ again represents a differential operator on $u(x)$ (e.g., $\mathcal{L} = \frac{\partial}{\partial x}$). Here the n nodes are known as a *stencil* with size n . The c_j are *stencil weights*. In practice stencils include the center, x_c , plus the $n - 1$ nearest neighboring nodes. The definition of “nearest” can depend the choice of distance metric, but in all discussions to follow it is assumed to be Euclidean distance ($\|x - x_c\|_2$).

Figure 3.1 provides two examples of RBF-FD stencils. A single stencil of size $n = 13$ is depicted in Figure 3.1a within a domain of random points. The center, x_c , is represented by a green square, with 12 neighbors connected via red edges. The purple circle—the minimum covering circle for the stencil—illustrates that the 12 nearest neighbors are selected. Figure 3.1b presents a larger stencil ($n = 75$) on the unit sphere with red and blue disks surrounding the square center. Green disks are nodes outside of the stencil. Radii and color of the disks indicate magnitude and alternating sign of the weights, c_j .

Following [10], weights for a 1-D classical-FD stencil can be obtained by solving a Vandermonde system,

$$\begin{bmatrix} 1 & 1 & \cdots & 1 \\ x_1 & x_2 & \cdots & x_n \\ x_1^2 & x_2^2 & \cdots & x_n^2 \\ \vdots & \ddots & \ddots & \vdots \\ x_1^{n-1} & x_2^{n-1} & \cdots & x_n^{n-1} \end{bmatrix} \begin{pmatrix} c_1 \\ c_2 \\ c_3 \\ \vdots \\ c_n \end{pmatrix} = \begin{pmatrix} \mathcal{L}1|_{x=x_c} \\ \mathcal{L}x|_{x=x_c} \\ \mathcal{L}x^2|_{x=x_c} \\ \vdots \\ \mathcal{L}x^{n-1}|_{x=x_c} \end{pmatrix}, \quad (3.2)$$

where the the x_j are assumed to be distinct for guaranteed nonsingularity. In higher dimensions, multivariate polynomials dissolve the guaranteed nonsingularity of the Vandermonde



(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 3.1: Examples of stencils computable with RBF-FD

system, so FD stencils are typically composed by adding weights from individual spatial directions.

In contrast to Equation 3.2, RBF-FD weights arise by enforcing that they be exact within the space spanned by the RBFs that are centered at stencil nodes (i.e., $\phi_j(x) = \phi(\epsilon\|x - x_j\|_2)$; an RBF centered at x_j). This amounts to replacing each polynomial basis function $\{1, x, x^2, \dots, x^{n-1}\}$ in Equation 3.2 with a d -dimensional RBF, $\phi_j(x)$, which allows for nonsingularity in d -dimensions on irregular node placements. Various studies [7, 10? ?] show that better accuracy is achieved when the interpolant can exactly reproduce a constant, p_0 , such that

$$\mathcal{L}\phi_i(x) |_{x=x_c} = \sum_{j=1}^n c_j \phi_j(x_i) + c_{n+1} p_0 \quad \text{for } i = 1, 2, \dots, n$$

with $\mathcal{L}\phi_i$ provided by analytically applying the differential operator to the RBF. Assuming

$p_0 = 1$, the constraint $\sum_{i=1}^n c_i = \mathcal{L}p_0|_{x=x_c} = 0$ completes the system:

$$\begin{bmatrix} \phi_1(x_1) & \phi_2(x_1) & \cdots & \phi_n(x_1) & 1 \\ \phi_1(x_2) & \phi_2(x_2) & \cdots & \phi_n(x_2) & 1 \\ \vdots & \ddots & \ddots & \vdots & \vdots \\ \phi_1(x_n) & \phi_2(x_n) & \cdots & \phi_n(x_n) & 1 \\ 1 & 1 & \cdots & 1 & 0 \end{bmatrix} \begin{pmatrix} c_1 \\ c_2 \\ \vdots \\ c_n \\ c_{n+1} \end{pmatrix} = \begin{pmatrix} \mathcal{L}\phi_1(x)|_{x=x_c} \\ \mathcal{L}\phi_2(x)|_{x=x_c} \\ \vdots \\ \mathcal{L}\phi_n(x)|_{x=x_c} \\ 0 \end{pmatrix} \quad (3.3)$$

$$\begin{bmatrix} \phi & P \\ P^T & 0 \end{bmatrix} \begin{pmatrix} c_{\mathcal{L}} \\ d_{\mathcal{L}} \end{pmatrix} = \begin{pmatrix} \phi_{\mathcal{L}} \\ 0 \end{pmatrix}.$$

The resulting structure of Equation 3.3 is the same structure found in RBF scattered data interpolation (see Equation 2.5). As with other RBF methods, the choice of \mathcal{L} can be any linear operator. If \mathcal{L} is the identity operator, then the above procedure leads to RBF-FD weights for interpolation. If $\mathcal{L} = \frac{\partial}{\partial x}$, one obtains the weights to approximate the first derivative in x . Refer to [5] for a table of commonly used RBF derivatives. Section 3.3 provides a list of derivatives used in this work.

The small $(n+1) \times (n+1)$ system in Equation 3.3 is dense, and is easily solved at a cost of $O(n^3)$ floating point operations (FLOPs) using direct methods like LU-decomposition. The resulting stencil weights, $c_{\mathcal{L}} = \{c_j\}_{j=1}^n$ can be substituted into Equation 3.1 for the derivative approximation at x_c . Coefficient c_{n+1} ($d_{\mathcal{L}} = c_{n+1}$), included in the solution of Equation 3.3, is of no use and discarded once the system has been solved.

Based on the choice of support parameter, ϵ , the Equation 3.3 may suffer problems with conditioning. In such cases, stable methods for solving the system like Contour–Padé [?], RBF-QR [? ?], or RBF-GA [?] may be preferred.

RBF-FD shares many advantages with global RBF methods. For example, the ability to function without an underlying mesh, easily extend to higher dimensions, and (in some cases) stability for large time steps. Unfortunately, spectral accuracy is lost due to the local nature of this stencil method. Other advantages of RBF-FD include low computational complexity together with high-order accuracy (6th to 10th order accuracy is common). As in classical FD methods, increasing the stencil size, n , increases the order accuracy of approximations. While not a panacea for PDEs, RBF-FD is simple to code, feature rich, and powerful in its ability to avoid singularities introduced by coordinate systems that might negatively impact other methods (see e.g., [8, 10]).

Author's Note: [Expand these reference descriptions](#)

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 [10], and the shallow water equations [7]. 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.

3.1 Multiple Operators

In many cases, multiple derivatives (e.g., $\mathcal{L} = \nabla^2, \frac{\partial}{\partial x}, \frac{\partial}{\partial y}$, etc.) are required at stencil centers. This is common, for example, when solving coupled PDEs. For RBF-FD, acquiring weights for each additional operator can be both straight-forward and computationally efficient. For each change of differential operator, observe that only the RHS of Equation 3.3 is modified. Thus, new operators amount to extending Equation 3.3 to solve

$$\begin{bmatrix} \phi & P \\ P^T & 0 \end{bmatrix} \begin{bmatrix} c_{\nabla^2} & c_x & c_y & \cdots \\ d_{\nabla^2} & d_x & d_y & \cdots \end{bmatrix} = \begin{bmatrix} \phi_{\nabla^2} & \phi_x & \phi_y & \cdots \\ 0 & 0 & 0 & \cdots \end{bmatrix}. \quad (3.4)$$

where multiple sets of weights (c_{∇}, c_x, c_y) are obtained simultaneously. This dense, symmetric, multiple RHS linear system is considered ideal by linear algebra packages. Many highly optimized routines exist to solve Equation 3.4 (e.g., LAPACK “dgesv”) [?].

3.2 Differentiation Matrices and Sparse Matrix-Vector Multiply (SpMV)

Typically, one needs derivatives at every node in the discretized domain to solve PDEs. To achieve this with RBF-FD, stencils are generated around every node in the domain. Stencils need not have the same size (n), but this is assumed here for simplicity in discussion, and is most common in literature. Furthermore, the number of stencils need not match the number of nodes in the domain, but this is also assumed. The small system solve in Equation 3.3 or 3.4 is repeated N times—once for each stencil—to obtain a total of $N \times n$ stencil weights per operator.

For PDEs, it is common practice to assemble a *differentiation matrix* (DM); a discrete representation of the PDE operator on the domain. Given the set of nodes in the domain, $\{x_k\}_{k=1}^N$, the c -th row of the DM represents the discrete PDE operator for the stencil centered at node x_c with stencil nodes $\{x_j\}_{j=1}^n$:

$$\begin{aligned} \mathcal{L}u(x) &\approx D_{\mathcal{L}}u \\ D_{\mathcal{L}}^{(c,k)} &= \begin{cases} c_j & x_k = x_j \\ 0 & x_k \neq x_j \end{cases} \end{aligned}$$

where (c, k) represents the (row, column) index of $D_{\mathcal{L}}$ and vector $u = \{u(x_k)\}_{k=1}^N$. Equation 3.1 can be rewritten as:

$$\mathcal{L}u(x) |_{x=x_c} \approx D_{\mathcal{L}}^{(c)}u .$$

DMs are utilized in both explicit and implicit modes. Here explicit implies evaluating the matrix-vector multiply to get derivative values, u' , from explicitly known vector of solution values u :

$$u' = D_{\mathcal{L}}u \quad (3.5)$$

whereas implicit solves for unknown u :

$$D_{\mathcal{L}}u = f \quad (3.6)$$

Note that the non-zeros in $D_{\mathcal{L}}$ are independent of the values in u . Approximating \mathcal{L} over any function reduces to sampling the function values at nodes $\{x_k\}_{k=1}^N$ and performing a matrix-vector multiply.

An example RBF-FD DM is illustrated in Figure 3.2. In this example, assume operator $\mathcal{L} = \frac{\partial}{\partial x}$ is approximated at all N stencil centers of an arbitrary domain. RBF-FD weights assemble the rows of the differentiation matrix, D_x . On each row, weights are indicated by blue dots. The sparsity of rows reflects the subset of $\{x_k\}_{k=1}^N$ included in corresponding stencils of size n . A mapping is assumed to exist that translates non-consecutive indices, k , to consecutive indices, j , and vice-versa. On the right hand side, discrete derivative values $\frac{du}{dx}$ are approximated at all stencil centers.

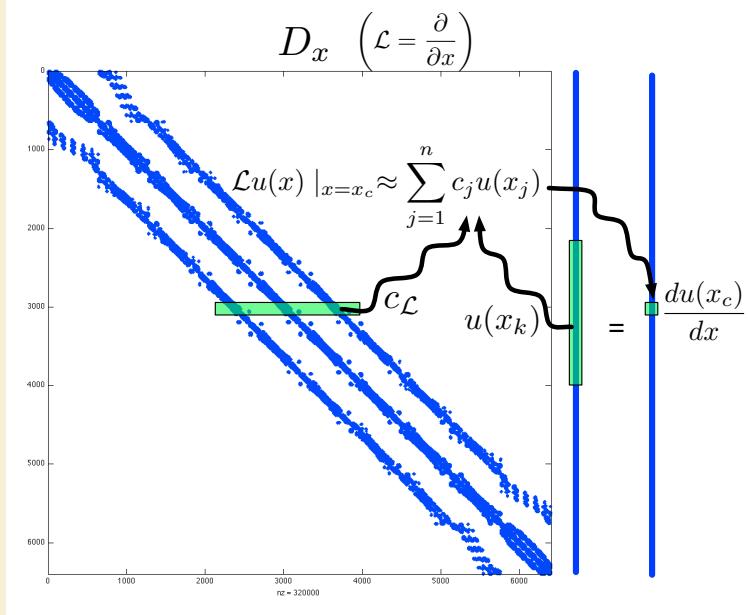


Figure 3.2: Differentiation matrix D_x is applied explicitly to calculate derivative approximations, $\frac{d}{dx}u(x)$.

Differentiation matrices are assembled at a cost of $O(n^3N)$ FLOPs. However, since the goal of RBF-FD is to keep stencils small ($n \ll N$), the cost of assembly scales as $O(N)$. Furthermore, RBF-FD weights are independent of function values ($u(x)$) and rely only on stencil node locations. The implications of this are as profound as in the context RBF-PS for time-dependent PDEs: the stencil weights are constant so long as the nodes are stationary. Thus, the DM assembly is part of a one-time preprocessing step.

For simple PDEs one often assembles a single DM to represent the operator for the entire differential equation, but RBF-FD allows flexibility in how operators are handled. Rather than a single DM, with weights from a new operator on the RHS of Equation 3.4, one may approximate the operator based on lower order derivatives. Consider for example,

the 2-D Laplacian operator, $\nabla^2 = \frac{\partial^2}{\partial x^2} + \frac{\partial^2}{\partial y^2}$:

$$\nabla^2 u \approx D_{\nabla^2} u$$

which can be expanded as:

$$\nabla^2 u \approx (D_{x^2} + D_{y^2}) u = D_{x^2} u + D_{y^2} u .$$

where either a single DM is composed by adding two lower order DMs, or the lower order DMs are directly multiplied against the vector u . Another option applies even lower order operators:

$$\nabla^2 u \approx D_x D_x u + D_y D_y u . \quad (3.7)$$

The choice of how the operators are approximated depends on the PDE and can be influenced by system memory limitations. For example, assume a coupled system of equations in 2-D where operators ∇ and ∇^2 are required. Then, Equation 3.4 is assembled and solved for the operators $\{\frac{\partial}{\partial x}, \frac{\partial}{\partial y}, \nabla^2\}$, with each DM stored in memory. This process is sufficient assuming all DMs fit adequately in the memory available. As an alternative, one may solve Equation 3.4 for operators $\{\frac{\partial}{\partial x}, \frac{\partial}{\partial y}\}$, reproduce ∇^2 with Equation 3.7, and reduce memory usage by 30%. On top of memory savings, this concept of composing DMs based on lower order operators extends to cases where PDEs require complex operators that are sufficiently difficult to apply to RBFs when deriving a new RHS for Equation 3.4.

The sparsity exhibited by the DM in Figure 3.2 is typical for RBF-FD. Consider that a problem size of $N = 10,000$ nodes and stencil size $n = 31$ is only 0.31% full (i.e., 99.79% of the matrix elements are zeros), and the percentage continues to decrease as N increases. Contrary to initial appearance, RBF-FD DMs like Figure 3.2 are not symmetric. This is true for two reasons: a) a stencil is generated based on $n - 1$ nearest neighbors to a center with no guarantee that any of the stencil nodes will include the center in their stencils; and b) even if the stencil connectivity were symmetric, each row of the DM contains a distinct set of weights that are a function of independent stencils. The only way to guarantee a symmetric system is to ensure all stencils are shaped the same, with the same number of nodes and node distribution relative to the center, plus all stencil nodes must have a bijective relationship (i.e., if A is connected to B , then B connects to A).

Best practices dictate that the DMs be stored in some type of compressed sparse storage that retains only non-zeros and their corresponding indices in memory. Note that with DMs stored as sparse representations, the matrix-vector multiply operation is distinguished as a *sparse matrix-vector multiply* (SpMV). SpMVs avoid unnecessary operations by only multiplying the nonzero elements of the matrix matched to corresponding values in the vector. The actual algorithm for SpMV depends on the choice of sparse storage. Chapter 5 demonstrates the pivotal role that sparse formats play in the performance of SpMV, and thus RBF-FD.

3.3 Weight Operators

In the course of this work a variety of operators are tested to solve PDEs. This section enumerates the operators and their corresponding equations for the RHS of Equation 3.3.

Whenever possible the general form of $\mathcal{L}\phi$ is provided; otherwise the Gaussian RBF ($\phi(r) = e^{-(\epsilon r)^2}$) is assumed.

3.3.1 First and Second Derivatives ($\frac{1}{r} \frac{\partial \phi}{\partial r}, \frac{\partial^2 \phi}{\partial r^2}$)

The following are used in subsequent derivatives:

$$\frac{1}{r} \frac{d}{dr} \phi(r) = -2\epsilon^2 \phi(r) \quad (3.8)$$

$$\frac{\partial^2 \phi}{\partial r^2} = \epsilon^2 (-2 + 4(\epsilon r)^2) \phi(r) \quad (3.9)$$

3.3.2 Cartesian Gradient (∇)

The first derivatives in Cartesian coordinates ($\frac{\partial}{\partial x}, \frac{\partial}{\partial y}, \frac{\partial}{\partial z}$) are produced via the chain rule:

$$\begin{aligned} \frac{\partial \phi}{\partial x} &= \frac{\partial r}{\partial x} \frac{\partial \phi}{\partial r} = \frac{(x - x_j)}{r} \frac{\partial \phi}{\partial r} \\ \frac{\partial \phi}{\partial y} &= \frac{\partial r}{\partial y} \frac{\partial \phi}{\partial r} = \frac{(y - y_j)}{r} \frac{\partial \phi}{\partial r} \\ \frac{\partial \phi}{\partial z} &= \frac{\partial r}{\partial z} \frac{\partial \phi}{\partial r} = \frac{(z - z_j)}{r} \frac{\partial \phi}{\partial r} \end{aligned}$$

where $\frac{\partial \phi}{\partial r}$ for the Gaussian RBFs is given in Equation 3.8.

3.3.3 Cartesian Laplacian (∇^2)

Fasshauer [5] provides the general form of ∇^2 in 2D as:

$$\nabla^2 = \frac{\partial^2}{\partial r^2} \phi(r) + \frac{1}{r} \frac{\partial}{\partial r} \phi(r)$$

For Gaussian RBFs in particular we have the following operators:

- 1D:

$$\nabla^2 = \epsilon^2 (-2 + 4(\epsilon r)^2) \phi(r)$$

- 2D:

$$\nabla^2 = \epsilon^2 (-4 + 4(\epsilon r)^2) \phi(r)$$

- 3D:

$$\nabla^2 = \epsilon^2 (-6 + 4(\epsilon r)^2) \phi(r)$$

which all fit $\nabla^2 = \frac{\partial^2}{\partial r^2} \phi(r) + \frac{d-1}{r} \frac{\partial}{\partial r} \phi(r)$ for dimension d .

3.3.4 Laplace-Beltrami (Δ_S) on the Sphere

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

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

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, resulting in:

$$\nabla^2 \equiv \Delta_S. \quad (3.10)$$

Although this originated in the spherical coordinate system, [?] gives the Laplace-Beltrami operator as

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

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

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

Following [7, 9], the gradient operator can be constrained to the sphere with this projection matrix:

$$P = I - \mathbf{x}\mathbf{x}^T = \begin{pmatrix} (1 - x_1^2) & -x_1x_2 & -x_1x_3 \\ -x_1x_2 & (1 - x_2^2) & -x_2x_3 \\ -x_1x_3 & -x_2x_3 & (1 - x_3^2) \end{pmatrix} = \begin{pmatrix} P_{x_1} \\ P_{x_2} \\ P_{x_3} \end{pmatrix} \quad (3.11)$$

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

The direct method of computing RBF-FD weights for the projected gradient for $\mathbf{P} \cdot \nabla$ comes from [9]. First, let $\mathbf{x} = (x_1, x_2, x_3)$ be the stencil center Cartesian coordinates, and $\mathbf{x}_k = (x_{1,k}, x_{2,k}, x_{3,k})$ be the coordinates of an RBF-FD stencil node.

Using the chain rule, and assumption that

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

the unprojected gradient of ϕ becomes

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

Applying the projection matrix gives

$$\begin{aligned}
\mathbf{P} \nabla \phi(r(\mathbf{x}_k - \mathbf{x})) &= -(\mathbf{P} \cdot \mathbf{x}_k - \mathbf{P} \cdot \mathbf{x}) \frac{1}{r(\mathbf{x}_k - \mathbf{x})} \frac{\partial}{\partial r} \phi(r(\mathbf{x}_k - \mathbf{x})) \\
&= -(\mathbf{P} \cdot \mathbf{x}_k - 0) \frac{1}{r(\mathbf{x}_k - \mathbf{x})} \frac{\partial}{\partial r} \phi(r(\mathbf{x}_k - \mathbf{x})) \\
&= -(I - \mathbf{x}\mathbf{x}^T)(\mathbf{x}_k) \frac{1}{r(\mathbf{x}_k - \mathbf{x})} \frac{\partial}{\partial r} \phi(r(\mathbf{x}_k - \mathbf{x})) \\
&= \begin{pmatrix} x_1 \mathbf{x}^T \mathbf{x}_k - x_{1,k} \\ x_2 \mathbf{x}^T \mathbf{x}_k - x_{2,k} \\ x_3 \mathbf{x}^T \mathbf{x}_k - x_{3,k} \end{pmatrix} \frac{1}{r(\mathbf{x}_k - \mathbf{x})} \frac{\partial}{\partial r} \phi(r(\mathbf{x}_k - \mathbf{x}))
\end{aligned}$$

Thus, weights for $P_x \cdot \nabla$ can be computed directly by using these three operators on the RHS in Equation 3.3:

$$\begin{aligned}
P \frac{\partial}{\partial x_1} &= (x_1 \mathbf{x}^T \mathbf{x}_k - x_{1,k}) \frac{1}{r(\mathbf{x}_k - \mathbf{x})} \frac{\partial}{\partial r} \phi(r(\mathbf{x}_k - \mathbf{x}))|_{\mathbf{x}=\mathbf{x}_j} \\
P \frac{\partial}{\partial x_2} &= (x_2 \mathbf{x}^T \mathbf{x}_k - x_{2,k}) \frac{1}{r(\mathbf{x}_k - \mathbf{x})} \frac{\partial}{\partial r} \phi(r(\mathbf{x}_k - \mathbf{x}))|_{\mathbf{x}=\mathbf{x}_j} \\
P \frac{\partial}{\partial x_3} &= (x_3 \mathbf{x}^T \mathbf{x}_k - x_{3,k}) \frac{1}{r(\mathbf{x}_k - \mathbf{x})} \frac{\partial}{\partial r} \phi(r(\mathbf{x}_k - \mathbf{x}))|_{\mathbf{x}=\mathbf{x}_j}
\end{aligned}$$

3.3.6 Hyperviscosity Δ^k for Stabilization

The hyperviscosity filter for stabilization is introduced in [10] and was included in our previous investigations in [2]. When explicitly solving hyperbolic equations, differentiation matrices encode convective operators of the form

$$D = \alpha \frac{\partial}{\partial \lambda} + \beta \frac{\partial}{\partial \theta} \quad (3.12)$$

The convective operator, discretized through RBF-FD, has eigenvalues in the right half-plane causing the method to be unstable [7, 10]. Stabilization of the RBF-FD method is achieved through the application of a hyperviscosity filter to Equation (3.12) [10]. By using Gaussian RBFs, $\phi(r) = e^{-(\epsilon r)^2}$, the hyperviscosity (a high order Laplacian operator) simplifies to

$$\Delta^k \phi(r) = \epsilon^{2k} p_k(r) \phi(r) \quad (3.13)$$

where k is the order of the Laplacian and $p_k(r)$ are multiples of generalized Laguerre polynomials that are generated recursively ([10]):

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

where d is the dimension. The application of hyperviscosity in Chapter ??, utilizes the operator as a filter to shift eigenvalues and stabilize advection equations on the surface of

the unit sphere. In that case, $d = 2$ can be assumed because individual RBF-FD stencils are viewed as (nearly) lying on a plane. A word of caution: for small N , the diameter of the stencil may not be sufficiently small compared to the radius of the sphere, and hyperviscosity might not work as advertised.

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

$$\frac{\partial u}{\partial t} = -\mathcal{L}u + Hu, \quad (3.14)$$

where \mathcal{L} is the PDE operator, and H is the hyperviscosity filter operator. Applying hyperviscosity shifts all the eigenvalues of L (the discrete form of \mathcal{L}) 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

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

It was found in [7], and verified in our own application, that $\gamma = \gamma_c N^{-k}$ provides stability and good accuracy as a function of N on the unit sphere. It also ensures that the viscosity vanishes as $N \rightarrow \infty$ [7]. In general, the larger the stencil size, the higher the order of the Laplacian required as a filter. 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 [7, 10].

Tuned parameters for hyperviscosity are provided in Chapter ??.

3.4 RBF-FD Implementation for Time-dependent PDEs

This section considers at a high level how one leverages RBF-FD to solve PDEs. To simplify in the discussion, consult Algorithm 3.1, which is split into two phases: preprocessing and application. The same processes would be The complexity of each phase can vary based the algorithms utilized for each task.

The Preprocessing phase encompasses tasks such as grid setup/generation, stencil generation and stencil weight calculations. As output the phase produces one or more DMs. Note that Preprocessing is a one time cost: grids, stencils, and DMs can be loaded from disk on subsequent runs to effectively bypass the all cost in this phase.

The algorithm starts by loading/generating a grid. RBF-FD requires only node coordinates, and in some cases an indication of whether nodes are on a boundary. Information on mesh edges/connectivity is optional, but could be used to bypass stencil generation. The QueryNeighbors step forms stencils and constructs a directed adjacency matrix that indicates the connectivity of stencils. Either the grid or the adjacency matrix can be partitioned (see Chapter 6) for distribution across multiple processors in the DecomposeDomain step. Finally, one or more processors operate independently to compute weights and assemble

Algorithm 3.1 A High-Level View of RBF-FD

Preprocessing:

```
{ $x$ } $_{j=1}^N$  = GenerateGrid()
for  $j = 1$  to  $N$  do
    Stencil { $S_{j,i}$ } $_{i=1}^n$  = QueryNeighbors( $x_j$ )
end for
{ $x$ } $_{j=1}^{N_p}$  = DecomposeDomain( $D_{\mathcal{L}}$ , { $x$ } $_{j=1}^N$ )
for  $j = 1$  to  $N_p$  do
    { $w_{j,i}$ } $_{i=1}^n$  = SolveForWeights({ $S_j$ })
     $D_{\mathcal{L}}^{(j)}$  = AssembleDM({ $w_j$ })
end for
```

Application:

```
 $t = t_{min}$ 
while  $t < t_{max}$  do
    { $u'$ } = SolvePDE( $D_{\mathcal{L}}$ , { $u$ })
    { $u$ } = UpdateSolution({ $u$ }, { $u'$ },  $\Delta t$ )
     $t += \Delta t$ 
end while
```

local DMs. The assumption in this work is that grids do not evolve in time, so DMs remain constant for the duration of the phase 2 (Application). In the event of moving nodes, weight calculation and DM assembly would move into the second phase and the method would become a Lagrangian particle method (e.g., [?]).

Once in the Application phase, the constructed DMs are applied to solve a PDE either explicitly (e.g., Equation 3.5) or implicitly (e.g., Equation 3.6). In the case of time-dependent PDEs, RBF-FD is applied the same as any classical FD method with a time-stepping scheme (represented by UpdateSolution). Examples of valid time-schemes include Runge-Kutta, Adams-Bashforth, and Crank Nicholson methods among many others. Based on the choice of time-scheme, one frequently needs multiple iterations through SolvePDE to assemble intermediate solutions that are weighted and combined in the the UpdateSolution step.

In general the performance of RBF-FD hinges on the cost of SolvePDE. This is especially true for time-dependent PDEs where an increase in grid resolution results in proportional increase in the number time-steps to satisfy the CFL condition [?]. Chapter 4 demonstrates that choosing the right algorithms for preprocessing tasks can also directly impact performance SpMV. Beyond this, the choice of stencils and accuracy of weights can impact the overall accuracy of approximation, stability and conditioning of the method. As each of these properties improve, the overall time to solution can decrease thanks to larger stable time-steps, smaller required grid size, and faster convergence.

3.5 Grids

RBF-FD has no requirement for structured grids, or need for a well-refined mesh/lattice to define and limit connectivity between nodes. It functions the same on existing meshes

and randomly generated point clouds; although, the actual node placement can impact accuracy of the method. This functional portability on domains of any shape, dimension, and granularity is a major selling point, and something often out of reach for other methods.

- 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

The RBF-FD implementation presented in this work is verified and benchmarked on a number of grids. The following grids are chosen for two reasons: a) to easily construct refinements in 2D/3D for consistent benchmarking and convergence studies, and b) to verify solutions against existing methods. All grids mentioned here are available for public download (see [?]).

Regular Grid. For basic debugging and benchmarking purposes the most natural choice is to start with a regular or Cartesian grid. Equally spaced nodes in multiple dimensions are simple to generate. Additionally, refinements—for scaling benchmarks and convergence tests—are direct subsamples.

In theory, RBF-FD functions the same whether nodes are uniformly spaced or random. However, regular grids do not fully exercise advantages that RBF-FD has over other methods with its ability to operate on scattered nodes. For this reason regular grids are only used here for benchmarking purposes and range in size from $N = 10^3$ to $N = 160^3 = 4,096,000$ nodes.

Maximum Determinant Nodes. Chapter ?? applies RBF-FD to solve PDEs on the unit sphere. For consistency with respect to related investigations (e.g., [10? ?]), the Maximum Determinant (MD) node sets [? ?] are chosen.

MD node sets were originally utilized by the RBF community due to their success in spherical harmonics interpolation [?]. For spherical harmonics, the seemingly irregular node distributions achieve an order of magnitude higher accuracy compared to regular looking node distributions (i.e., minimum energy nodes) [?]. In similar fashion, RBF methods have been shown to benefit from a subtle irregularity in node locations on the sphere due to the tendency in RBF interpolation to reproduce spherical harmonics interpolants when $\epsilon \rightarrow 0$ [?].

The MD node files are available for download on the authors' web site [?], and range in size from $N = 4$ up to $N=27,556$ nodes on the sphere. Figure 3.3 plots the $N = 4096$ node set to illustrate the irregularity in distribution. Node sets greater than $N=27,556$ are not available. Unlike regular grids, each MD node set is a refinement of the sphere, but not a subdivision, so extending beyond $N=27,556$ nodes would require complete regeneration.

Icosahedral Nodes on the Sphere. Figure 3.4 shows Icosahedral nodes on the sphere. Icosahedral grids are nearly homogenous and isotropic, and have been in use since the 1960s [?]. The grids originate as an icosahedron which is refined by subdividing edges equally and projecting back onto the unit sphere. The direct subdivisions imply that tests on icosahedral grids are true refinements of previous grids (in contrast to MD-nodes).

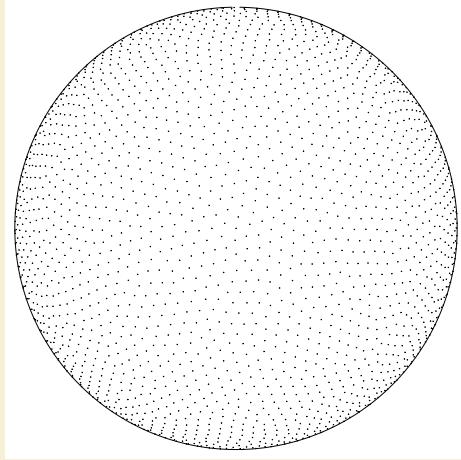


Figure 3.3: Quasi-regular nodes with $N = 4096$ maximum determinant (MD) node sets on the unit sphere.

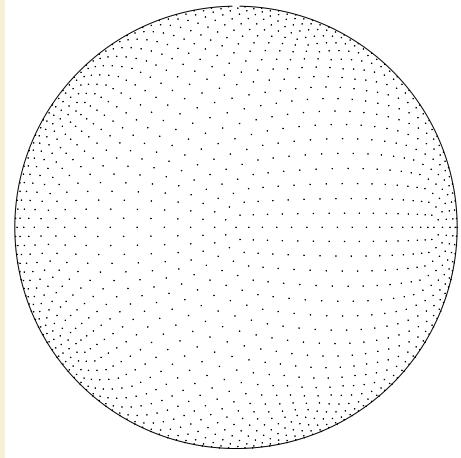


Figure 3.4: $N = 2562$ icosahedral nodes on the unit sphere.

This work tests Icosahedral grids with $N=42$ up to $N=163842$ (i.e., the first through 7th refinements).

Centroidal Voronoi Tessellations. On the sphere, MD nodes suffice for verification against related work on small to mid size grids (i.e., 30,000 nodes), and Icosahedral grid subdivisions allow for scaling tests and slightly larger mid-sized problems (i.e., 160,000). However, the objective is to scale RBF-FD to large problem sizes that can justify the need for HPC. For this, approximately regular grids on the sphere on the order of millions of nodes are needed. To this end, Spherical Centroidal Voronoi Tessellations (SCVTs) are leveraged to generate high resolution, approximately regular node distributions on the sphere [? ?].

The process to generate SCVTs involves constructing a Voronoi diagram, computing the mass centroids for each Voronoi partition, and updating node locations to the mass centroids projected onto the sphere. After a number of iterations, the nodes converge to

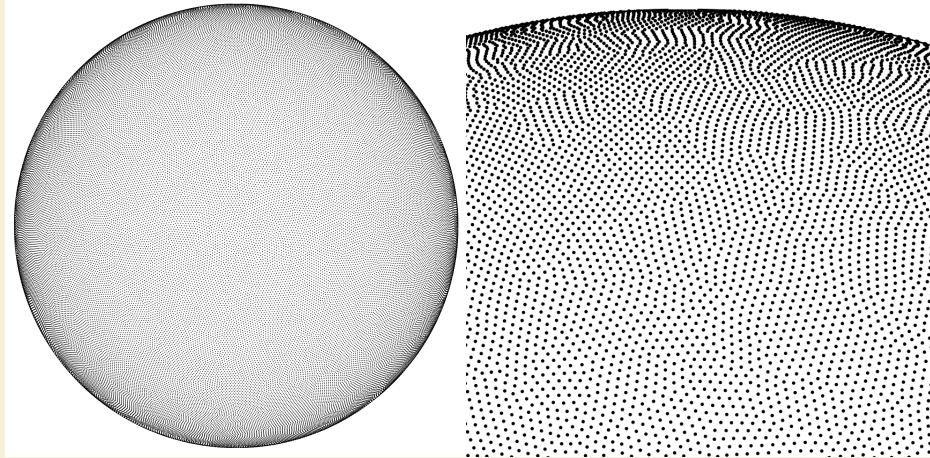


Figure 3.5: (Left) $N=100,000$ Spherical Centroidal Voronoi Tessellation nodes. (Right) Close-up of the same $N=100,000$ nodes to illustrate the irregularities in the grid.

nearly coincide with the projected mass centroids, and the resulting distribution is a SCVT. SCVTs come with a sense of “optimality” in node locations due to energy minimizing properties (see [?]). In most large-scale applications, the iteration in SCVT generation is a probabilistic Lloyd’s algorithm, with integrals computed through random sampling [? ?]. While SCVTs in theory converge to a near isotropic node distribution, the probabilistic nature of the centroid calculation introduces irregularities reminiscent of MD nodes.

Figure 3.5 provides an example SCVT grid with $N=100,000$ nodes. On the left, the full sphere; on the right, a close-up of the same node set. The close-up perspective clearly demonstrates the random artifacts/scarring of irregularly distributed nodes. For benchmarking purposes we use node sets $N=100,000$, $N=500,000$ and $N=1,000,000$ generated by the SCVT library from [?].

3.6 On Choosing the Right ϵ

If solving for RBF-Direct weights directly (i.e., inverting Equation 3.3 directly), one must balance the choice of ϵ to avoid ill-conditioning but achieve a reasonable accuracy for the weights. Numerous attempts exist in literature to provide “good” functions for ϵ based on node spacing (h), stencil size (n), and total number of nodes in the domain (N). No fool-proof method exists for RBF-Direct—the “optimal” (in most cases this reads: “acceptable”) value of ϵ depends on the node distribution and varies by application.

This work utilizes a moderately reliable method, proposed in [7], for choosing ϵ . The method expresses ϵ as a function of the grid resolution, N , and the desired mean condition number for Equation 3.3, $\bar{\kappa}_A = \frac{1}{N} \sum_{i=1}^N (\kappa_A)$. Here κ_A is the condition number of the interpolation matrix from Equation 3.3.

The tiles of Figure 3.6 illustrate a number of contours generated in MATLAB. Each contour is numbered according to $\log_{10} \bar{\kappa}_A$. Data was generated by uniformly sampling parameter spaces on ϵ and \sqrt{N} for the MD node-sets. For each $\sqrt{N} = \{40, \dots, 100\}$, in steps of 10, the code makes a sweep through $\epsilon = \{1, \dots, 10\}$ in steps of 0.5 and assembles

N RBF-FD interpolation matrices. A call to MATLAB’s “cond()” routine evaluates κ_A for each matrix. The resulting values of $\bar{\kappa}_A$ produce remarkably linear contours, with slopes that fan out from the ϵ axis. Note that κ_A depends on the stencil size, n , as evidenced by shifting contour fans for $n = 20, 40, 60, 80$, and 100. The simultaneous increase in slope and decreasing separation between contours leaves little “wiggle” room in guessing ϵ under RBF-Direct.

Regression slope and intercept parameters (c_1 and c_2) are superimposed in Figure 3.6 to aid others in choosing ϵ in new applications. These parameters reproduce contours as $\epsilon(\sqrt{N}) = c_1\sqrt{N} - c_2$. The authors of [7] find condition numbers on the order 10^{10} to 10^{12} to function well for competitive accuracy compared to other methods in literature; the method begins to degrade on large grids (i.e., $N > 1e5$). Similar observations are made in Chapter ??.

Stable weight algorithms like Contour-Padé [?], RBF-QR [? ? ?], and RBF-GA [?] can allow RBF-FD to scale beyond the limits of RBF-Direct and will be included in future investigation.

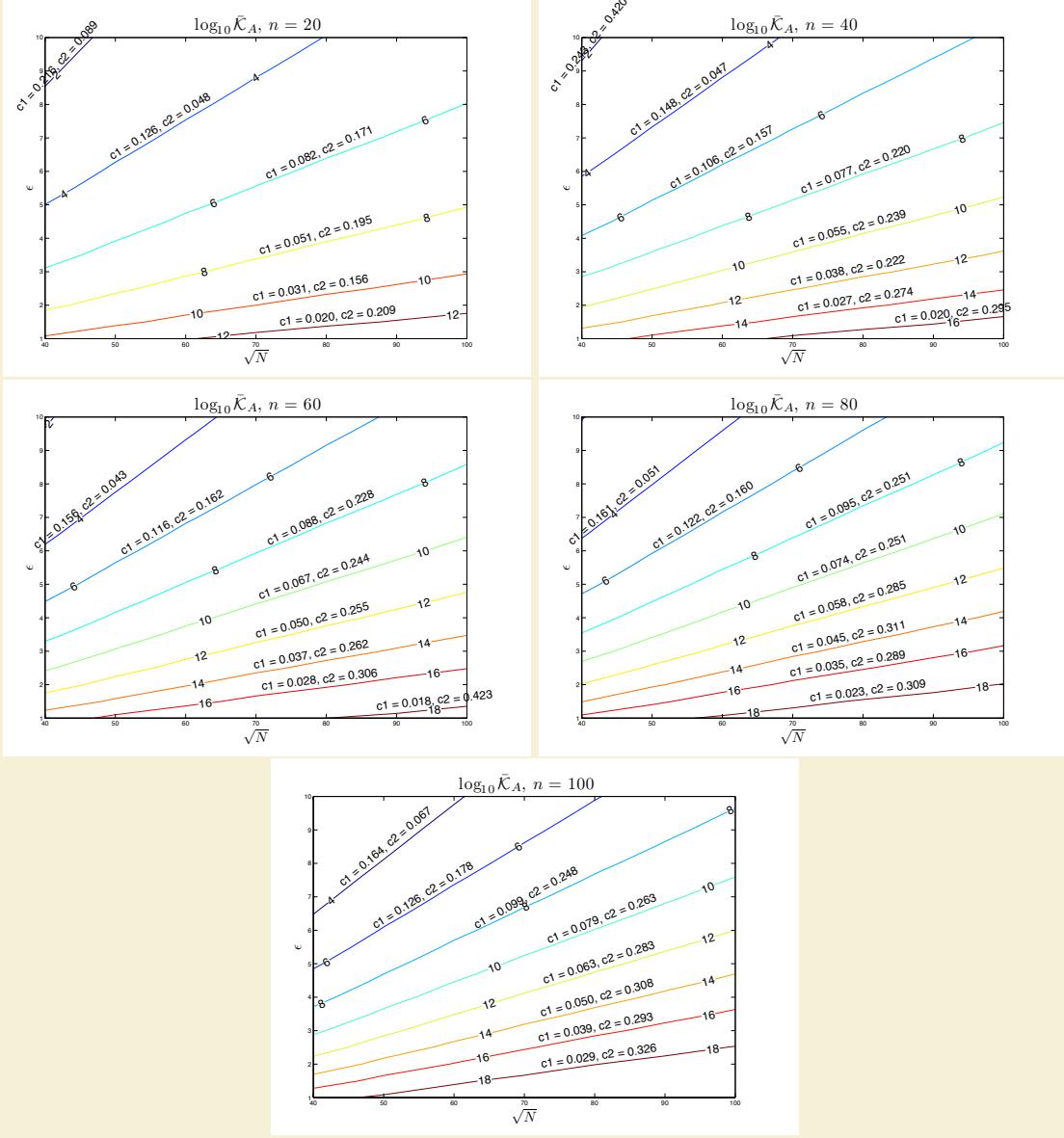


Figure 3.6: Contours for ϵ as a function of \sqrt{N} for stencil sizes $n = 20, 40, 60, 80$ and 100 on the unit sphere. Contours assume near uniform distribution of nodes (e.g., maximum determinant (MD) nodes). Parameters superimposed above each contour provide coefficients for function $\epsilon(\sqrt{N}) = c_1\sqrt{N} - c_2$.

CHAPTER 4

AN ALTERNATIVE STENCIL GENERATION ALGORITHM FOR RBF-FD

Like all RBF methods, RBF-FD is designed to handle irregular node distributions, so the emphasis in literature focuses on how the method manages point clouds. While nothing prevents implementations of RBF-FD from utilizing existing meshes/lattices, most work in the field concentrates on simple geometries to better understand properties of the method and develop extensions. Without mesh/lattice connectivity available, stencils are generated by choosing the n -nearest neighbors to a center node, inclusive of the center. This is known more formally as a *k-nearest neighbor (k-NN)* problem [24] (a.k.a. ℓ -nearest neighbor search [27]). Here “nearest” is defined with the Euclidean distance metric, although it is possible to generalize to other metrics (see e.g., [1]).

In comparison to the RBF-FD method, global RBF methods with infinite support connect all nodes to all other nodes, so there is no need for neighbor queries. On the other hand, compact RBF methods require all nodes—with no limit on the count—that lie within the support/radius of the RBF centered at each node. This type of neighbor query is referred to as a *ball query* (a.k.a. range query [27]) due to the closed ball created by the radius of support for a compact RBF (see Equation 2.1).

The *k*-NN and ball query share many similarities, but the former can be harder to solve. Consider, for example, the scenario in Figure 4.1. Two ball queries around a green stencil center are represented as dashed and dash-dot circles. The inner query returns four neighbors, and the outer returns six. If a stencil of size $n = 6$ is desired, then the outer query can be truncated to give the five required neighbors shown in blue. In this example the red node and the farthest blue node are equidistant from the center, and ties are broken arbitrarily. Although *k*-NN is simply a truncated ball query, the real challenge lies in finding the proper search radius to enclose at least the n desired neighbors. To find the radius in practice depends on the choice of data structure used to access node locations.

A naïve approach for neighbor queries would be a brute-force search that checks distances from all nodes to every other node. Obviously the cost of such a method is high: $O(N^2)$ for all stencils. Multi-dimensional data structures, such as those discussed here, can limit the scope of searching and reduce the cost of stencil generation to $O(N \log N)$.

For the most part, investigations in RBF communities that delve into efficient neighbor queries are limited to ball queries. For example, the Partition of Unity method for approximation (e.g., [26, 27]), and particle methods like the Fast Multipole Method (e.g., [14, 28])

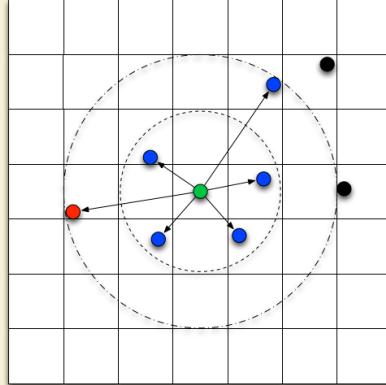


Figure 4.1: A stencil center in green finds neighboring stencil nodes in blue. Two ball queries are shown as dashed and dash-dot circles to demonstrate the added difficulty of finding the right query radius to obtain the k -nearest neighbors.

or Smoothed Particle Hydrodynamics (e.g., [16]). Examples of fast algorithms employed in these fields include the fixed-grid method [16, 27], k -D Trees [27], Range Trees [26, 27], and 2^d -Trees (i.e., Quad- and Octrees) [14, 28]. Surprisingly, while other communities continue the quest for fast neighbor queries, RBF collocation and RBF-FD communities have been slow on the uptake. For many years, the standard in the community has been to use k -D Trees (see e.g., [5, 7, 10]).

This chapter considers the use of an alternative neighbor query algorithm to generate RBF-FD stencils. It is based loosely on the fixed-grid method from [13, 15, 16]. [22] would classify the algorithm as a *fixed grid “bucket” method with one-dimensional spatial ordering*. The fixed-grid method loosens the requirements for finding the k -nearest neighbors (k -NN) stencils to accept k -“approximately nearest” neighbors (k -ANN). It also reorders nodes according to space-filling curves. In what follows, the fixed-grid method is compared to an efficient implementation of k -D Tree available for use in C++ and MATLAB ([24]). Benchmarks demonstrate that, with the proper choice of parameters for the fixed-grid, the method can be up to 2x faster than k -D Tree, and it comes with a free bonus: up to 5x faster SpMV performance due to the impact of spatial reordering that occurs during stencil generation.

4.1 k -D Tree

A k -D Tree is a spatial data structure that generally decomposes a space/volume into a small number of cells. All k -D Trees are binary and iteratively subdivide volumes and sub-volumes at each level into two parts. The “ k ” in k -D Tree refers to the dimensionality of the data/volume partitioned—that is $k \equiv d$.

Given a set of points bounded by a d -dimensional volume, a k -D Tree applies a hierarchy of $(d - 1)$ -dimensional axis aligned *splitting planes* to cut the space. At each level of the hierarchy the splitting planes result in two new *half-planes* [23]. Consecutive splits intercept one another at a *splitting value*. k -D Trees do not require that half-planes equally subdivide

a volume; more often it is the data contained within the volume that is equally partitioned. The choice of dimension for the splitting plane, in conjunction with a variety of methods for choosing the splitting values allows for many flavors of k -D Trees (see e.g., [4, 22, 23] for comprehensive lists). *Point k-D Trees*, 2^d *Trees* (i.e., quad-/octrees), *BSP-Trees*, and *R-trees* are all members of the general k -D Tree class [23, 28].

This work considers *Point k-D Trees* [22], which partition a set of discrete points/nodes as outlined by the recursive procedure in Algorithm 4.1. Point k -D Trees assume that splitting planes intercept nodes rather than occur arbitrarily along the half-plane. The splitting value at each level of the tree is set to the *median coordinate* of the points in the half-plane, which ensures the tree is well balanced on initial construction. All nodes with coordinate (in the current dimension) less than or equal to the splitting value are contained by the left half-plane, and all nodes with coordinate greater than the splitting value are contained by the right. Half-planes containing only one element correspond to leaves of the tree. The median coordinate of a half-plane is found by sorting the n node coordinates contained by the partition and selecting the $\lceil \frac{n}{2} \rceil$ -th element [4].

Algorithm 4.1 BuildKDTree(P , $depth$)

- 1: **Input:** A set of d -dimensional points P and the current $depth$.
- 2: **Output:** The root of the k -D Tree for P .
- 3:
- 4: **if** $\text{size}(P) = 1$ **then**
- 5: **return** a new leaf storing P
- 6: **end if**
- 7: $L_i := \text{median}(\text{coord}(P, depth))$
- 8: $v_l := \text{BuildKDTree}(\text{coord}(P, depth) \leq L_i, (\text{depth} + 1) \text{ modulo } d)$
- 9: $v_r := \text{BuildKDTree}(\text{coord}(P, depth) > L_i, (\text{depth} + 1) \text{ modulo } d)$
- 10: **return** A new node $v := \begin{cases} \text{value} := L_i \\ \text{left} := v_l \\ \text{right} := v_r \end{cases}$

The k -D Tree in Figure 4.2 is an example of a Point k -D Tree. Given a set of eight nodes in two dimensions, the tree is constructed by applying one-dimensional cuts along the x -dimension, then the y -dimension, then back to the x -dimension. This approach is referred to as *cyclic splitting*, as consecutive cuts are applied by iterating dimensions in a round-robin fashion [22]. The first cut, L_1 , shown in green, splits the nodes into two sets on either side of A . The corresponding tree in the center of Figure 4.2 shows L_1 as the tree root with all nodes having x -coordinates less than or equal to A to the left, and all nodes having x -coordinates greater than A to the right. The second level of the tree, L_2 and L_3 (in blue), splits the half-planes on either side of A at nodes B and C . The axis parallel splits for each half-plane intercept L_1 independently to partition half-planes along the y -dimension; once again, nodes with coordinates less than or equal (i.e., below) to the splitting value branch left in the tree, and y -coordinates greater than (i.e., above) the value branch right. The third level (red) returns to splitting half-planes in the x -dimension. Nodes D and H are not intersected by a splitting plane; their half-planes contain only one node so they immediately become leaves of the tree. This process to build a Point k -D Tree

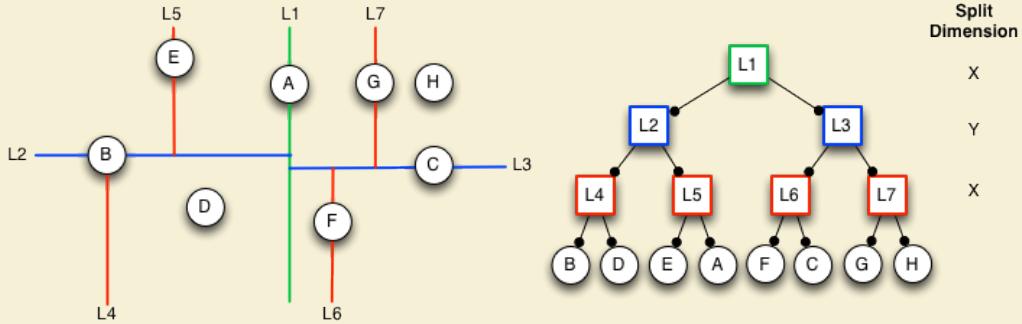


Figure 4.2: An example k -D Tree in 2-Dimensions. Nodes are partitioned with a cyclic dimension splitting rule (i.e., splits occur first in X , then Y , then X , etc.); all splits occur at the median node in each dimension.

has a complexity of $O(N \log N)$ with $O(N)$ storage required [4, 22].

Frequently, the terms *k -D Tree* and *Point k -D Tree* are used synonymously by the RBF community (see e.g., [5, 7, 10]); the same is convention adopted here.

Generating an RBF-FD stencil with a k -D Tree can be efficiently accomplished in $O(n \log N)$ time—where n is the stencil size—following an approach introduced in [11], and presented in Algorithm 4.2. The k -NN search starts a depth-first recursive search of the k -D Tree to find the nearest neighbor to a query point, X_q . Traversal of the tree occurs by following branches left or right based on comparison of X_q coordinates to the splitting value stored at each node of the tree, with the objective to find the smallest half-plane containing X_q . The search traverses the height of the tree in $O(\log N)$ steps to find the leaf that stores the nearest neighbor to X_q . The neighbor point and its distance from X_q are inserted into a global priority queue, pq . Points in the priority queue are sorted in descending order according to distance.

After finding the nearest neighbor the algorithm returns to the previous split in the tree and traverses onto the opposing half-plane (i.e., down the far branch) to look for other leaves. So long as the size of pq is at less than capacity (n) the search automatically adds points to the priority queue. If pq reaches capacity the algorithm starts to pop off excess points with the understanding that the action removes those points farthest from X_q .

In order to prune branches from the search and reduce complexity, Algorithm 4.2 makes use of a routine called “BoundsOverlapBall”, which checks if any boundaries of the current level half-plane intersect/overlap with a closed ball centered at X_q . The ball is given a radius equal to the maximum distance in pq . Then, if the ball and a boundary intersect, the search will continue onto the half-plane on the opposite side of that boundary. This step handles the possibility that nearer nodes occur within the overlapped region in the other half-plane. If the ball and boundary do not intersect, the opposing half-plane and its related subtree are pruned from the search. Additional details on the implementation of “BoundsOverlapBall” can be found in [11, 25].

The authors of [11] find Algorithm 4.2 capable of efficiently querying the n -nearest neighbors with a complexity proportional to $O(\log N)$ (dominated by the cost of tree traversal). The relationship between stencil size n , and grid size, N , is better expressed as $O(n \log N)$

Algorithm 4.2 KNNSearchKDTree($X_q, n, root, depth$)

- 1: **Input:** A query node X_q , number of desired neighbors (n), the current *root* of the k -D Tree, and the current *depth* of traversal.
- 2: **Output:** A global priority queue, pq , containing the n -nearest neighbors to X_q sorted by distance from X_q in descending order.
- 3: **Assume:** A routine named “BoundsOverlapBall” exists to determine if the boundaries of the current half-plane are intersected by the ball centered at X_q with radius equal to the maximum distance in pq . As long as $pq.size < n$, “BoundsOverlapBall” defaults to true.

- 4:
- 5: **if** *root* is leaf **then**
- 6: Insert $\{root, \text{dist}(X_q, root)\}$ into pq
- 7: **if** $pq.size > n$ **then**
- 8: $pq.pop$ ▷ Keep only n -nearest neighbors
- 9: **end if**
- 10: **return**
- 11: **end if**
- 12:
- 13: **if** $\text{coord}(X_q, depth) \leq root.value$ **then**
- 14: KNNSearchKDTree($X_q, n, root.left, (depth + 1) \% d$)
- 15: **else**
- 16: KNNSearchKDTree($X_q, n, root.right, (depth + 1) \% d$)
- 17: **end if**
- 18:
- 19: **if** $\text{coord}(X_q, depth) \leq root.value$ **then**
- 20: **if** BoundsOverlapBall(X_q) **then**
- 21: KNNSearchKDTree($X_q, n, root.right, (depth + 1) \% d$)
- 22: **end if**
- 23: **else**
- 24: **if** BoundsOverlapBall(X_q) **then**
- 25: KNNSearchKDTree($X_q, n, root.left, (depth + 1) \% d$)
- 26: **end if**
- 27: **end if**
- 28: **return**

for one stencil.

RBF-FD only needs to generate stencils once, so the overall time for the step subsumes the cost of tree construction and N queries. The resulting total complexity of stencil generation for all stencils is then proportional to $O(N \log N)$.

4.2 A Fixed-Grid Algorithm

While a k -D Tree functions well for queries, the cost to build the tree structure is unnecessary overhead. Among the many data-structures that exist for nearest neighbor queries, alternatives like fixed-grid methods [22, 26, 27] (a.k.a. uniform grid [13, 16]) bypass

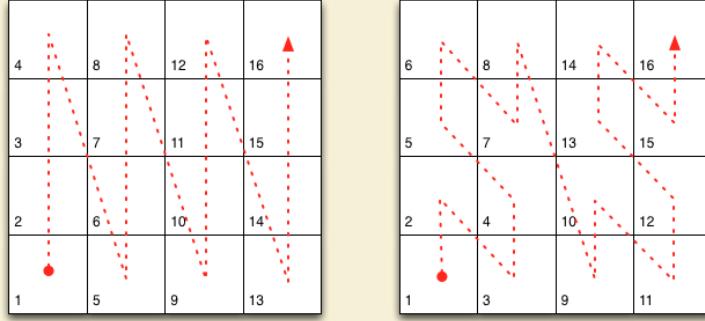


Figure 4.3: Two example space filling curves to linearize the same fixed-grid. Left: Raster-ordering (ijk); Right: Morton-/Z-ordering.

much of the cost in construction with an assumption that only lower spatial dimensions (e.g., 2-D or 3-D) are significant for choosing neighbors. This discards the need to build a tree and shifts focus onto querying neighbors.

Fixed-grid methods get their name from a coarse 2-D or 3-D regular grid that is overlaid on the domain. The d -dimensional grid divides the domain's axis aligned bounding box (AABB)—that is, the minimum bounding box containing the entire domain with edges parallel to axes—into $(h_n)^d$ cells. Subdivisions are uniform, so one can easily identify the cell containing any sample point, p , given the coordinates of the AABB and $(h_n)^d$. For example, let (c_x, c_y, c_z) be the desired cell in 3-D, and $(x_{min}, y_{min}, z_{min})$ and $(x_{max}, y_{max}, z_{max})$ be the minimum and maximum coordinates of the AABB (resp.). Then the cell coordinates are found by:

$$(dx, dy, dz) = \left(\frac{(x_{max} - x_{min})}{h_n}, \frac{(y_{max} - y_{min})}{h_n}, \frac{(z_{max} - z_{min})}{h_n} \right)$$

$$(c_x, c_y, c_z) = \left(\left\lfloor \frac{(p_x - x_{min})}{dx} \right\rfloor, \left\lfloor \frac{(p_y - y_{min})}{dy} \right\rfloor, \left\lfloor \frac{(p_z - z_{min})}{dz} \right\rfloor \right). \quad (4.1)$$

Cells neighboring (c_x, c_y, c_z) are trivial to find by adding positive and negative offsets to each coordinate.

Fixed grid methods also make use of *space filling curves*. Space filling curves pass through every point in d -dimensional space, and through each point only once. Equivalently, space filling curves map d -dimensional space down to 1-D, where every point is converted to a unique index or traversal order based on its spatial coordinates. These mapping properties make space filling curves ideal for use as hash functions. Traversing the d -dimensional points (i.e., playing “connect the dots”) draws the space filling curve. Figure 4.3 presents two common orderings of a 2-D fixed-grid. Note that one-dimensional orderings are not unique. On the left is a *Raster*-ordering (a.k.a. Scanline- or ijk -ordering): $f(c_x, c_y, c_z) = ((c_x * h_n) + c_y) * h_n + c_z$. The right half of Figure 4.3 shows an ordering known as *Morton*- or *Z*-ordering. *Z*-ordering construction is discussed later in this chapter. On both sides of Figure 4.3, the lower left corner of each cell indicates the mapped index. Traversing the cells in order produces the curves superimposed in red.

At a high level, fixed-grid methods have the following construction steps [16]:

1. Subdivide the domain with the overlay grid.
2. For each node, identify the containing cell coordinates.
3. For each node, use the cell coordinates as input to a spatial hash function (i.e., a space-filling curve).
4. Sort the nodes according to their spatial hash.

Particular details of how nodes are sorted, the choice of hashing function, the number of nodes allowed per cell, etc. determine the specific class of fixed-grid method and corresponding complexity. A comprehensive list of options and classifications can be found in [22].

4.2.1 Fixed-grid Construction

The algorithm in this work is inspired by fixed-grid approaches for GPU particle simulations ([13, 15, 16]). Particle methods require a ball query at each time-step. With time-steps often dominated by the cost of querying neighbors, the community understandably devotes significant effort to seek out the most efficient solutions possible [12]. The fixed-grid method is competitive for at least two reasons: a) by bypassing the need to build a tree, half the cost in querying neighbors is avoided; and b) nodes sorted according to a spatial hash reside closer in memory to nearby neighbors than in the case of unsorted nodes. The spatial locality results in a higher likelihood that data will be cached when required. Note that reordering by cell hash sorts nodes across cells but not within them—that is, nodes contained by the same cell are contiguous in memory, but remain arbitrarily ordered with respect one another. Fortunately, with contiguous groups of nodes, nearest neighbor queries can directly access all nodes per cell.

The authors of [13, 15, 16] assume a raster-ordering on cells, and that the uniform grid is sufficiently refined to ensure cells contain at most eight nodes. Particle interactions are limited to ball queries on the containing cell plus one valence of neighboring cells (i.e., 8 surrounding cells in 2-D and 26 cells in 3-D). Since the number of cells to check is fixed, the neighboring nodes can be obtained by direct access in constant time. A similar approach is taken by [12], but the authors opt for a Z -ordering of cells. As a point of difference in implementations, the authors of [12, 13, 16] leverage a fast radix sort algorithm to order nodes based on hash index, while [15] utilizes a slower bitonic sort algorithm. The fixed-grid in [26, 27] forgoes logic to refine the grid and enforce a maximum limit on the number of nodes per cell. The author also avoids sorting nodes based on cell hashes. Instead, a list is maintained that stores the indices of all contained nodes per cell.

The implementation presented here is a hybrid of the related algorithms. For example, cells are sorted based on raster-ordering, but without the restriction on max number of nodes per cell. Rather than a radix- or bitonic sort to reorder nodes, the list of node indices for each cell ([26, 27]) is constructed as part of a single-pass bucket sort. Finally, in stark contrast to [13, 15, 16, 26, 27], querying neighbors is not restricted to a fixed radius, or number of cell valences. To satisfy the k -NN query, this implementation iteratively increases the query radius to include a new valence of cells at each iteration. This multi-pass ball-

query was demonstrated in Figure 4.1. The iteration terminates when the desired count of neighboring nodes is satisfied or exceeded.

Algorithm 4.3 BuildFixedGrid(P, h_n)

```

1: Input: A set points  $P$ , and the fixed-grid resolution,  $h_n$ .
2: Output: The reordered points in  $P$ , and corresponding cell buckets  $Q$ .
3:
4: Create  $Q$ : an  $(h_n)^d$  array of empty buckets.
5: for point  $p_i$  in  $P$  do
6:    $c := \text{CellCoords}(p_i)$ 
7:    $ind := \text{SpatialHash}(c)$ 
8:   Append index  $i$  onto  $Q[ind]$ 
9: end for
10: for  $j = 0, 1, \dots, (h_n)^d$  do
11:   if  $Q[j]$  is not empty then
12:     Append the set  $P[Q[j]]$  onto  $\hat{P}$ 
13:     Overwrite the set  $Q[j]$  with new indices of  $\hat{P}$ 
14:   end if
15: end for
16:  $P := \hat{P}$ 
17: return

```

Algorithm 4.3 presents the fixed-grid build process. The routine starts with the allocation an array of empty buckets, Q . Next Q is populated based on the spatially hashed cell coordinates. The second for-loop in Algorithm 4.3 iterates through Q , looking for non-empty buckets. When one is found, nodes referenced by that bucket are transcribed/appended onto the “sorted” list of nodes \hat{P} . This way the nodes in each cell are contiguous, but maintain the original ordering with respect to one another. Additionally, node indices in \hat{P} replace the old indices within Q .

The entire build process complexity is proportional to $O(N)$, and requires $O((h_n)^d + N)$ storage. Samet [22] would classify this approach as a *fixed-grid bucket method with one-dimensional ordering*. The term *bucket* refers to the allowance for each cell to contain an arbitrary number of nodes. *One-dimensional ordering* is indicative of attempts later in the chapter to employ alternative space-filling curves in place of raster-ordering.

A special note: the final step of Algorithm 4.3 overwrites the original list of nodes with the sorted equivalent. The spatially sorted list is included in the cost of stencil generation, but available for reuse elsewhere. Since the first step in RBF-FD applications is to generate stencils, overwriting the input node set can guarantee that the node values throughout the entire life-cycle of an RBF-FD application will benefit from the same spatial locality as stencil generation. This benefit is (almost) free.

Consider Figure 4.4, which shows two differentiation matrices generated based on the same $N = 6400$ MD-node set (unit sphere), with each row representing an RBF-FD stencil of $n = 50$ non-zeros. The left matrix in Figure 4.4 is generated with stencils queried by a k -D Tree. The k -D Tree maintains the original ordering on P . The matrix on the right of Figure 4.4 is a permuted equivalent of the left matrix with fixed-grid cells sorted based on a raster-ordering for $h_n = 10$.

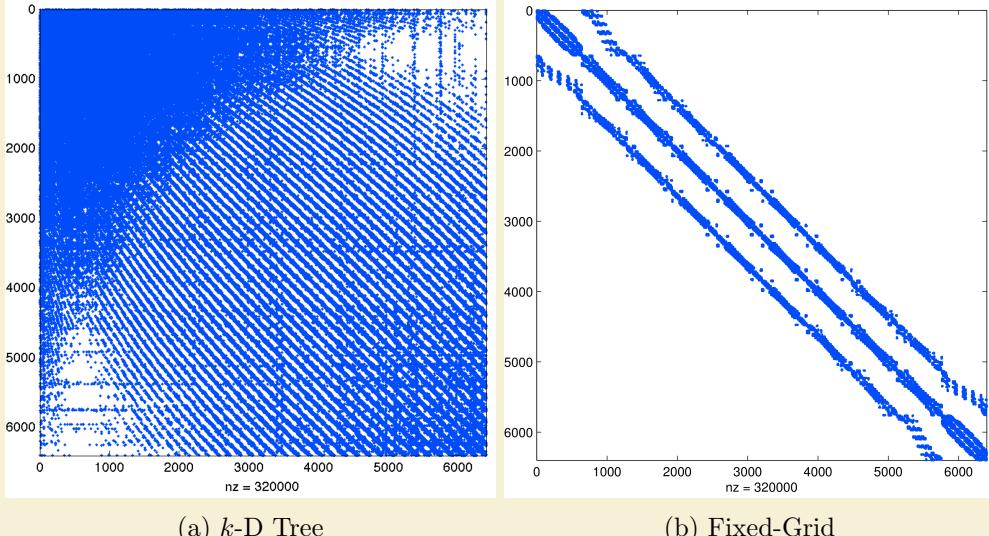


Figure 4.4: Example effects of node reordering for MD node set $N = 6400$ with $n = 50$. The differentiation matrices are permuted equivalents and roughly 0.78% full. Stencils generated based on k -D Tree maintain the original node ordering, while a fixed-grid with $h_n = 10$ condenses non-zeros for improved memory access patterns (i.e., cache reuse).

Looking at Figure 4.4 it should be obvious that reordering the nodes can improve memory access patterns for SpMV. If each row is applied as a sparse dot-product with a dense vector, the more condensed non-zeros are in the row, the more likely values from the dense vector will be resident in cache when needed. Likewise, non-zeros that appear on consecutive rows can benefit from cache reuse. Later in this chapter, the impact of spatial orderings are compared to determine how RBF-FD can benefit the most.

4.2.2 Fixed-Grid Neighbor Query

Querying the k -nearest neighbors for a single query node, X_q , is the subject of Algorithm 4.4. The process begins by finding X_q 's containing cell, c . The hash value of c is used to identify (in Q) the list of nodes contained within that cell, all of which are appended to a vector of neighbor candidates, pq .

It is possible for the number nodes in c to exceed n ; however, the algorithm conservatively assumes that it is necessary to search at least one valence of cells around it. This ensures that nodes near the cell boundaries will find nearby neighbors outside of c , and the stencils will be balanced. For certain fixed-grid resolutions, a single valence may not satisfy the stencil size requirements, so the algorithm iterates outward onto new valences. As cells are checked, their nodes are appended onto pq .

The final stage of Algorithm 4.4 calculates the distance from all candidate nodes to X_q , and uses that metric to sort pq in ascending order. The first n nodes in pq are returned as the stencil.

Complexity of Algorithm 4.4 can vary based on the choice of h_n . For a sufficiently refined fixed-grid the k -ANN is dominated by the cost of the *while-loop* and behaves as

Algorithm 4.4 QueryFixedGrid(X_q, n, P, Q)

- 1: **Input:** A query point, X_q ; the desired number of neighbors, n ; a set of d -dimensional points P ; and the matching cell bucket list, Q .
- 2: **Output:** The n -nearest neighbors list pq .
- 3:
- 4: $valence := 1$
- 5: $c := \text{CellCoords}(X_q)$
- 6: $ind := \text{SpatialHash}(cells)$
- 7: Append $P[Q[ind]]$ onto pq
- 8: **while** $pq.size < n$ OR $valence < 2$ **do**
- 9: $cells := \text{NeighboringCellCoords}(c, valence)$
- 10: $inds := \text{SpatialHash}(cells)$
- 11: **for** each q in $Q[inds]$ **do**
- 12: **if** q is not empty **then**
- 13: Append node list $P[q]$ onto pq
- 14: **end if**
- 15: **end for**
- 16: increment $valence$
- 17: **end while**
- 18: $dists := \text{ComputeDistances}(pq)$
- 19: Sort pq by $dists$
- 20: **return** the first n nodes in pq

$O(\log h_n)$ per stencil. In the worst case, when h_n is small, the cost of sorting pq dominates, and is proportional to $O(N \log N)$ (using a C++ STL Sort) in the worst case. Results below demonstrate that proper choice of h_n can maintain logarithmic complexity similar to the k -D Tree query.

The fixed grid query algorithm is considered an *approximate nearest neighbor* (ANN) search. Consider again the nodes in Figure 4.1. A k -NN stencil of size $n = 8$ should contain the green center, all blue nodes, plus the red node and one black node. The true k -NN would select the black node in the right-most column of the grid (i.e. the node closer to the dashed ball query). Under the fixed-grid method, however, the alternative black node is selected even though it is more distant. This is true because the more distant black node occupies the second valence of cells around the stencil center, whereas the true near neighbor is in the third valence. Algorithm 4.4 is able to truncate the search in the second valence by satisfying the requirement on n .

Similarly, if cells are rectangular in shape, the ball-query under fixed-grid functions as an ellipsoid. In this case, stencils are biased with more nodes in one direction. To combat this, and ensure spherical stencils, this work assumes the AABB bounding the domain is a cube (i.e., $dx = dy = dz$).

The difference between a true k -NN and k -ANN is insignificant from the perspective of RBF-FD. The method compensates automatically for differences in node locations when weights are calculated. On top of this, the only differences in stencils generated by k -NN versus k -ANN occur at nodes on the outermost reaches of the stencil (i.e., nodes with the least impact on the stencil center).

4.3 Performance Comparison

The implementation of k -D tree compared in this work, the *kd-Tree Matlab* library, was originally posted to the Matlab FileExchange in 2008 [24] and now maintained as an independent Google Code project [25]. The implementation is written in C++, but includes a MEX compiled interface, allowing for a consistent and efficient k -D Tree API in both languages. The original release of *kd-Tree Matlab* (pre-2012) was in use throughout the RBF community at the onset of this work. The dual language API was appealing for rapid-prototyping with MATLAB, and then porting applications to C++.

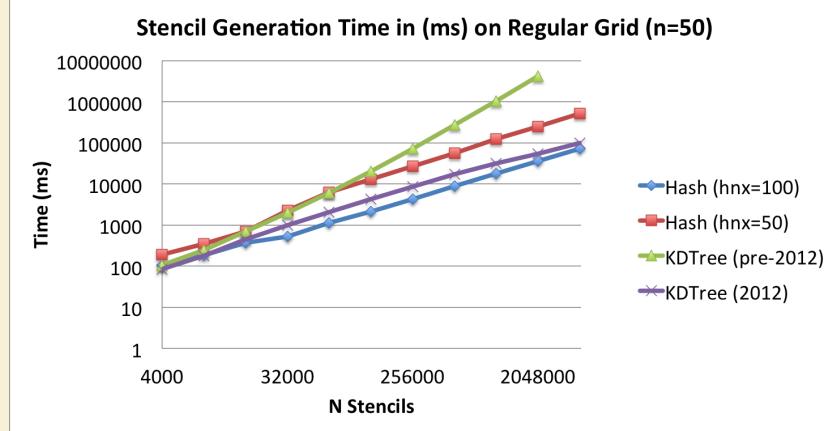
The pre-2012 implementation followed the $O(N \log N)$ expected complexity for neighbor queries, but the cost to build the tree was another story. Build times for small and medium sized grids (i.e., less than $N = 50000$ nodes) were small enough to be inconspicuous. However, large grid sizes were found to be prohibitively high, with the implementation behaving as $O(N^2)$. This issue ultimately led to work on a MATLAB prototype of the fixed-grid method ([?]) to test the concept of neighbor queries with low build costs.

The fixed-grid implementation, originally a pure MATLAB script, outperformed the “efficient” MEX-compiled k -D Tree for problem sizes $N > 20000$, and lead to the development of a faster C++ implementation tested here. However, with the 2012 release of *kd-Tree Matlab* on Google Code ([25]), the author has significantly improved the performance of the build process to achieve the $O(N \log N)$ behavior expected for a Point k -D Tree with cyclic splitting.

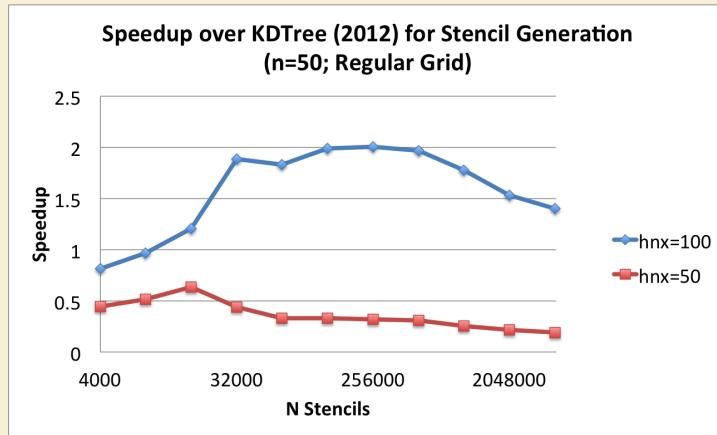
All benchmarks in this section were performed on the Itasca cluster at the Minnesota Supercomputing Institute. Itasca is an HPLinux cluster with 1,134HP ProLiant blade servers, each with two-socket, quad-core 2.8 GHz Intel Xeon processors sharing at least 24GB of RAM [?]. Both the k -D Tree and fixed-grid implementations are compiled with the Intel compiler toolchain (v13) and with the “-O3” optimizations for auto-vectorization, loop unrolling, etc.

Figure 4.5 demonstrates the performance of the k -D Tree and the fixed-grid method on increasing 3-D regular grid resolutions up to four million nodes. Both the current k -D Tree implementation (2012) and the original implementation (pre-2012) are shown on top to demonstrate the significant improvement in the latest release. For comparison, the C++ fixed-grid method is shown with two resolutions: $h_n = 50$ and $h_n = 100$. On the bottom, Figure 4.5 shows the associated speedups—defined as the ratio of time to compute k -D Tree stencils over the time for the same work in fixed-grid—achieved by the fixed-grid method over the 2012 release of *kd-Tree Matlab*. These timings cover the total time to build the data-structure and query all stencils. Also, the original node ordering is in raster-order, as is the fixed-grid.

Prior to the 2012 improvements, the C++ implementation was over 130x faster for four million nodes. With the newer and more reasonable k -D Tree benchmarks, the fixed-grid method is only 2x faster in the best case shown here. Although 2x is not as impressive, it is a notable improvement. One issue in Figure 4.5 is that the fixed-grid with $h_n = 50$ is consistently more than twice as slow as the k -D Tree. This is due to under-resolved cells with 32 nodes per cell; $h_n = 100$ has only 4 nodes per cell. For small N the k -D Tree is faster than both resolutions of fixed grid due to over-resolution. As N increases beyond one million nodes the $h_n = 100$ fixed-grid loses ground on the k -D Tree due to under-resolution.



(a) 3-D Regular Grid with stencil size $n = 50$



(b) Speedup of fixed-grid method versus k -D Tree

Figure 4.5: Querying the $n = 50$ nearest neighbors on a regular grid up to $N = 160^3$ demonstrates the gains achieved by the fixed-grid neighbor query method.

Similar behavior is seen in Figure 4.6 where the fixed-grid and k -D Tree are compared for various discretizations of the unit sphere. Each of the node sets described in Figure 4.6 are discussed in Chapter 3 and available for download ([?]). A range of resolutions are covered from a few hundred up to one million with some overlap. Each distribution (MD, Icosahedral, CVT) are generated differently and the nodes are naturally spatially sorted (icosahedral) or random (MD, CVT). Due to sorting in the fixed-grid method, node sets that are originally random exhibit more gain over k -D Tree. This is most evident in Figure 4.6, where different slopes appear for the segment corresponding to MD nodes and similar resolutions of Icosahedral nodes. The speedup curves for $h_n = 50$ and $h_n = 100$ demonstrate the dependence of fixed-grid method's success on the proper choice of h_n . When over-resolved (i.e., for $N < 10000$) the k -D Tree performs best. Under-resolution degrades performance rapidly starting at $N = 100000$ for $h_n = 50$, and $N = 500000$ for $h_n = 100$.

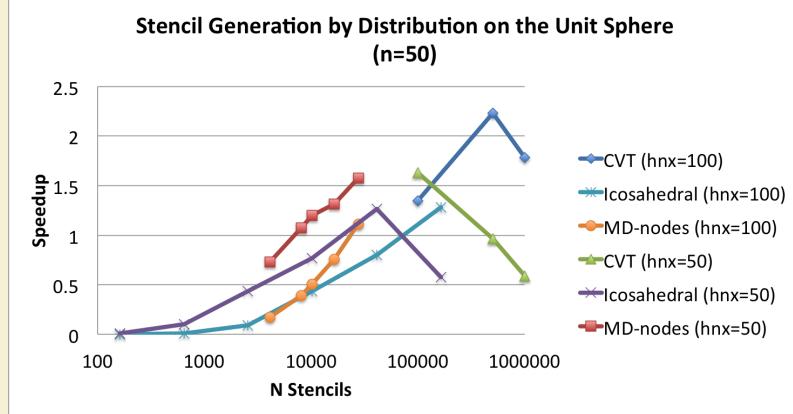


Figure 4.6: Fixed-grid speedup versus k -D Tree with stencil size $n = 50$.

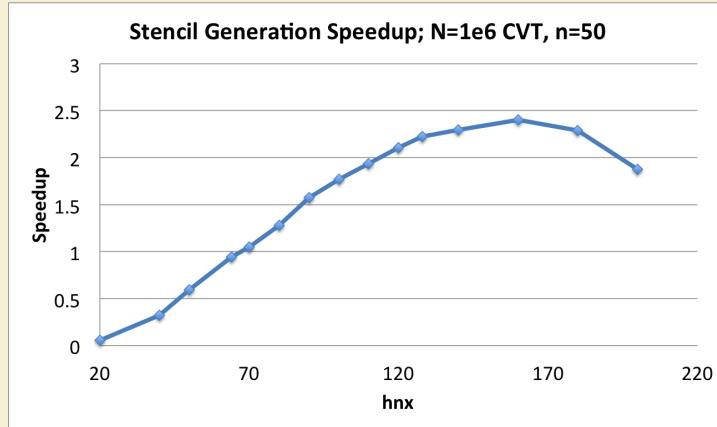
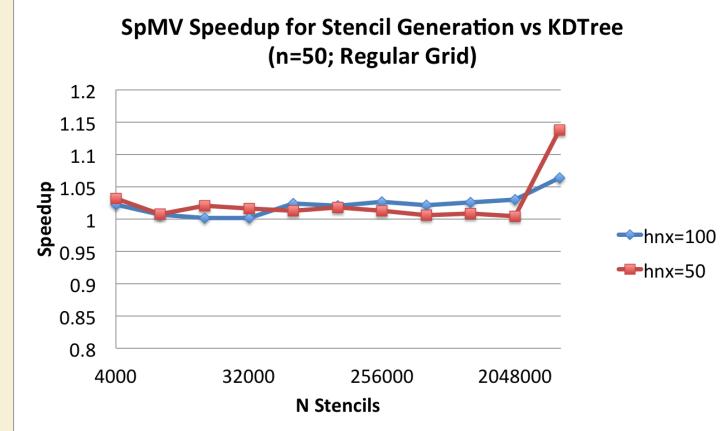


Figure 4.7: Fixed-grid speedup versus k -D Tree on a one-million node CVT (unit sphere), with stencil size $n = 50$.

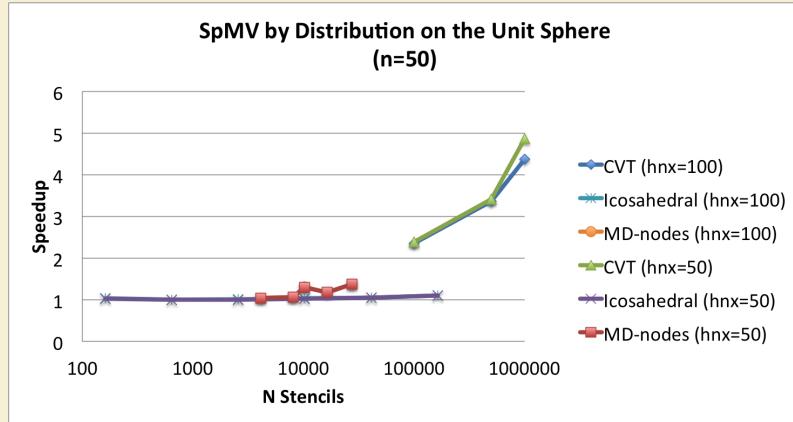
The speedup curves in Figures 4.5 and 4.6 hint at a maximum value h_n for which the fixed-grid will outperform its competitor, and with that value: a maximum speedup possible. In Figure 4.7, the $N = 10^6$ resolution CVT is used to generate stencils of $n = 50$, with range of values for h_n . The maximum of this curve shows that the fixed-grid can achieve up to 2.4x better than the k -D Tree for $h_n = 160$. In this case the number of cells, $(h_n)^3$, sufficiently resolves the domain for most cells intersecting the sphere to have one or two nodes. At $h_n = 70$ the fixed-grid is on par with k -D Tree; most cells have less than 8 nodes, which is consistent with the resolution sought by [12, 13, 16].

4.3.1 Impact on SpMV

Based on evidence so far, the fixed-grid method is not a big winner against the k -D Tree, but rather, it ekes out a small victory with the right choice of h_n for fewer than 8 nodes per cell. The reality is that since stencil generation only occurs once, and the difference between k -D Tree and fixed-grid is easily amortized by iterations during the bulk of the RBF-



(a) 3-D Regular Grid



(b) Unit Sphere

Figure 4.8: Fixed-grid impact on SpMV for stencil size $n = 50$.

FD application phase. However, as Figure 4.8 illustrates, the fixed-grid method includes another longer lasting benefit. Namely, the Sparse Matrix-Vector Multiply (SpMV)—the major overhead in RBF-FD applications—benefits positively due to the reordering that occurs during stencil generation. Reordering cells improves locality of nearby nodes and associated values, and leads to up to 5x speedup in the SpMV for random node distributions.

Note that regular grid and icosahedral test cases in Figure 4.8 see no more than 5% improvement, which is to be expected as they are previously spatially sorted node distributions. CVT and MD node sets, however, see a maximum speedup of 5x and 1.4x respectively. Furthermore, none of the test cases result in SpMV slowdown, so a case can be made for use of the fixed-grid query as a general safety net to precondition the system when properties of the input grid are unknown.

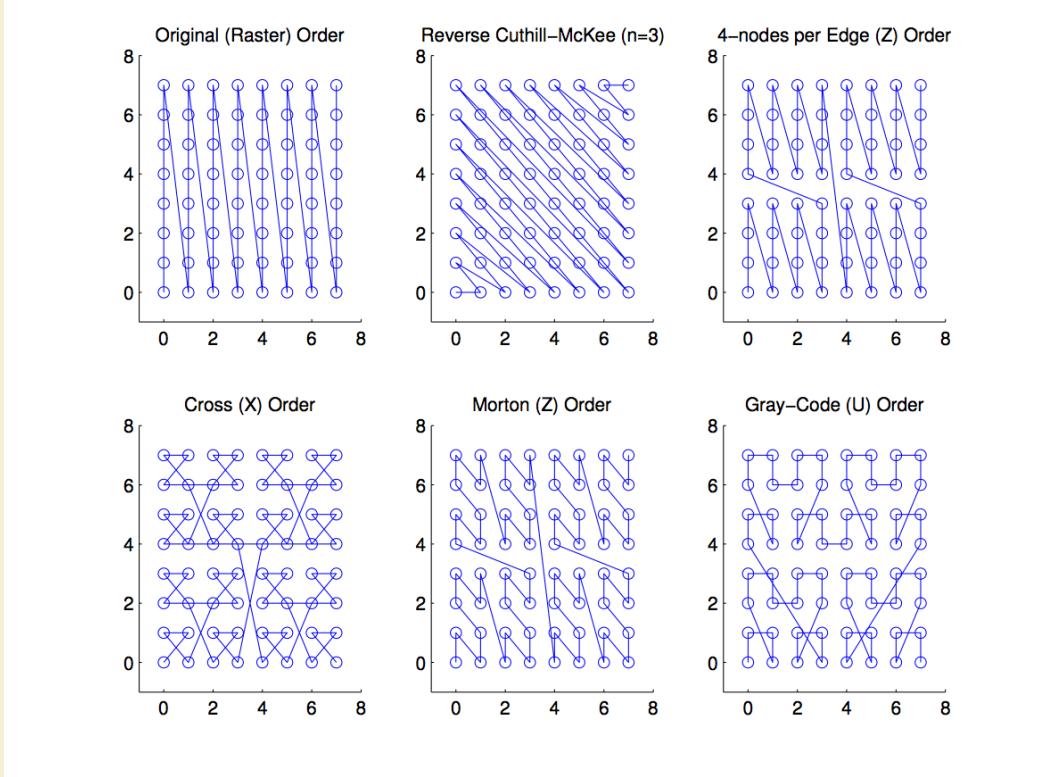


Figure 4.9: Example space filling curves used to reorder cells/nodes. The Raster, Z, X, U and 4-nodes per Edge are space filling curves applied to reorder cells of the fixed-grid stencil queries. Reverse Cuthill-McKee (RCM) operates on the output stencils to the associated adjacency graph. The RCM shown here is a special case with 3 neighbors per node.

4.4 Alternative Orderings

As mentioned, the choice of space filling curve can impact the sparsity pattern of differentiation matrices, which in turn correlates to different memory access patterns. The thought of using *Z*-ordering in a fixed-grid method for optimized memory access is not unknown (e.g., see the suggestions by [13, 15, 16] and implementation in [12]).

Morton, or *Z*-ordering curves can be constructed using integer dilation and bit interleaving [?]. The process

Similar techniques can be used for other curves as well. For example:

Figure 4.12 presents a variety of curves possible for a 2D node set. Options are endless, but the curves presented in Figure 4.12 include:

- Raster (IJK) –
- U order

Examples include *Z*- or *Morton* ordering, *U*- or *Greycode*-ordering, *X*-ordering, etc. Hilbert curves have been demonstrated to be the most efficient [?].

The authors of [12] utilize *Z*-ordering in their

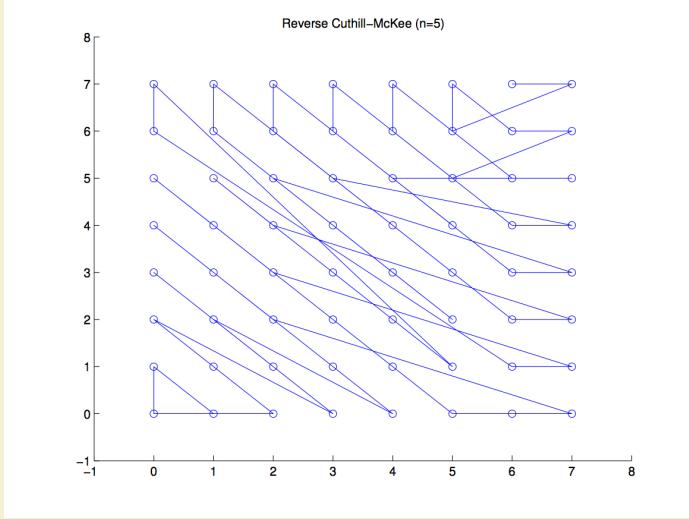


Figure 4.10: An RCM reordering of nodes based on stencils $n = 5$ nearest neighbors per node.

The ideal differentiation matrix, corresponding to discretization of a line, would have all non-zeros on the first $\frac{n-1}{2}$ to each side of the diagonal. For 2- or 3-D domains the ideal case is not possible to achieve, but the nodes can be reordered with the goal to condense. At the end of `BuildFixedGrid` the original set of nodes, P is overwritten by \hat{P} . While nothing requires that \hat{P} replace the usage of P

[18] found that reordering nodes via RCM and space filling curves offer similar benefits in terms of reduced TLB misses and better cache coherency.

4.4.1 Cuthill-McKee

Another option for reordering nodes is to focus on ordering the differentiation matrix directly to reduce bandwidth. A commonly used algorithm for this is called *Reverse Cuthill-McKee* (RCM) reordering. RCM takes as input a square, symmetric adjacency graph

Another option for reordering nodes arises after stencils have been generated. The stencils

A variety

The Cuthill McKee algorithms can be equated to a breadth-first search over the adjacency graph. The algorithm starts by

Step 0: Prepare an empty queue Q and an empty result array R . Step 1: Select the node in $G(n)$ with the lowest degree (ties are broken arbitrarily) that hasn't previously been inserted in the result array. Let us name it P (for Parent). Step 2: Add P in the first free position of R . Step 3: Add to the queue all the nodes adjacent with P in the increasing order of their degree. Step 4.1: Extract the first node from the queue and examine it. Let us name it C (for Child). Step 4.2: If C hasn't previously been inserted in R , add it in the first free position and add to Q all the neighbours of C that are not in R in the increasing order of their degree. Step 5: If Q is not empty repeat from Step 4.1 . Step 6: If there are unexplored nodes (the graph is not connected) repeat from Step 1 . Step 7: Reverse the

order of the elements in R. Element $R[i]$ is swapped with element $R[n+1-i]$.

The result array will be interpreted like this: $R[L] = i$ means that the new label of node i (the one that had the initial label of i) will be L .

The algorithm queues nodes in order of degree at each level of the search and traverses the lowest degree priority. The Reverse variant of Cuthill-McKee inverts the node order so that the lowest degree and top level node are at the end of the matrix rather than the beginning. Aside from ordering, the Reverse and Standard Cuthill McKee algorithms are identical processes. However, RCM is the more popular of the variants due to reduced fill-in that occurs when some matrix decompositions are applied to the reordered matrix ([17]).

Cuthill McKee implementations assume the algorithm operates on square, symmetric adjacency graphs.

Rather than complete a full analysis of the impact from space filling curves, we only consider the ideal cases from CVT and RG (because they are used in the thesis). The integer dilation is costly and does not appear to significantly improve the performance of the best cases.

RCM is effective and can be used as post-processing once stencils are generated and/or DMs have been assembled.

In Figure 4.12 typical stencils in RBF-FD easily span multiple cells. Shown is $n = 31$ which spans 7 cells. The use of Z-ordering (e.g., [12]) is often associated with improved memory access patterns. However,

The RCM algorithm is a breadth first search over stencils to order them based on the minimum degree (number of connections per stencil). The zig-zag ordering shown in Figure 4.12 is an ideal case where all nodes are connected to two neighbors (i.e., $n = 3$). In practice the curve trends diagonally across the domain due to the BFS property, but the ordering often appears to connect nodes randomly

Obviously, the ideal case for bandwidth is when all rows contain the $\frac{n}{2}$ nodes corresponding to solution value to either side of u_j . In 1-D this corresponds to every node containing the $\frac{n}{2}$ nodes to the left and right of x_j . In 2-D this is only possible if the nodes in the domain are properly indexed such that stencils contain the proper set of neighbors—a stringent requirement that will

impact from ordering on matrix sparsity. Bandwidth impact. Bandwidth impact on condition considered in future chapter.

One frequently hears that ordering via space filling curves like Morton Ordering and/or gray codes can benefit memory access patterns.

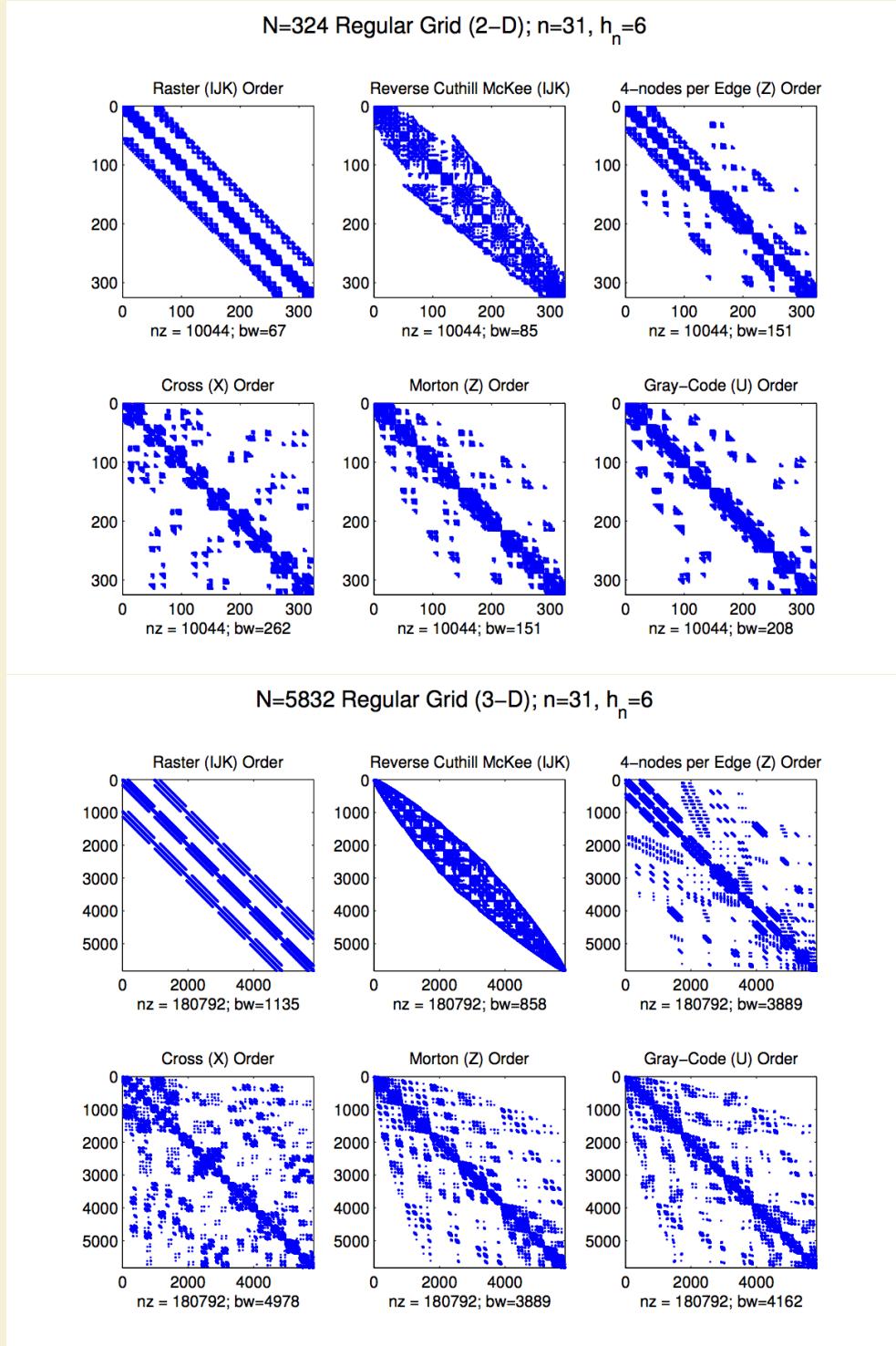


Figure 4.11: In order: a) node ordering test cases; b) original ordering of regular grid (raster); c) coarse grid overlay for hash functions ($hn_x = 6$); d) example stencil ($n = 31$) spanning multiple Z's; e) spy of DM after orderings. 3) Spy impact on MD node set $N = 4096$, stencil size $n = 31$, $h_n = 6$

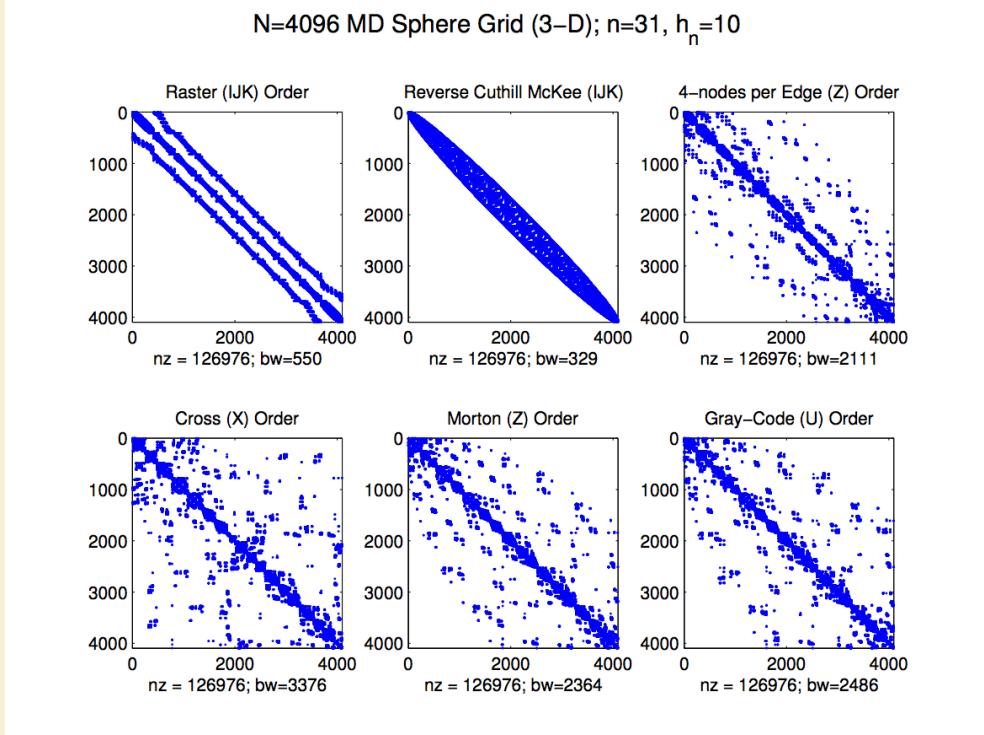


Figure 4.12: The impact of reordering on sparsity patterns for $N = 4096$ MD nodes on the sphere. Stencil size $n = 31$, fixed-grid resolution per dimension $h_n = 10$.

4.5 Conclusion and Future Work

Although RBF-FD only requires neighbor queries once, the results that follow reveal a long lasting positive impact on memory with a fixed-grid method, which is sufficient to justify its use. Investigations into moving node coordinates and/or local refinements for RBF methods (e.g., [6]) would find the fixed-grid method significantly more beneficial. As of this writing no known applications of RBF-FD consider moving nodes

Due to the limited significance of stencil generation under RBF-FD, the overhead in implementing and debugging the fixed-grid method on the GPU is difficult to justify. The implementation tested here was developed as a pure CPU prototype with minimal attention to optimization. The added complexity in reproducing the efficient fixed-grid method on the GPU could be the subject of future work for moving nodes.

[3] could benefit the algorithm by sorting nodes completely based on floating point Z-ordering.

4.5.1 Future Work

While more efficient implementations are possible, the savings demonstrated by Figure ?? the savings are significant with the right choice of h_n . Generating stencils for RBF-FD is a preprocessing cost, so we do not dedicate an excessive amount of attention to this algorithm. However, a few ideas that would improve: hilbert ordering, choose AABB

resolution based on N not user parameters, faster sorting, GPU implementation

Open item from to this section include:

- An efficient GPU k -NN implementation for the investigation of RBF-FD in a moving nodes or adaptive mesh refinement situation. Examples include [3, 12, 19]
- Space filling curves are limited to a MATLAB prototype at the moment. These can be implemented in C++ for integers, with a potential investigation into floating point dilation and orderings [3].
- [21] mentions the impact of ordering on conditioning.

4.6 Conclusions on Stencil Generation

For quasi-regular distributions and small to medium sized grids the k -D Tree performs well enough in comparison to the fixed-grid method. However, the difference in

There is an ideal h_n .

TODO:

- Node orderings based on integer dilation
 - Dilation and interleave process (include description of groups by dimension)
 - Give formulas for interleaving
 - RCM algorithm high level (?)
 - bandwidth implications on memory access (lower bandwidth is usually better)
 - table showing the impact of orderings on bandwidths

CHAPTER 5

GPU SPMV

As N grows larger, it behooves us to work on parallel architectures, be it CPUs or 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 coprocessors 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.

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.

5.1 Related Work

[?] [?] [?] etc.

5.2 GPGPU

GPGPU evolution

5.2.1 OpenCL

OpenCL is chosen with the future in mind. Hardware changes rapidly and vendors often leapfrog one another in the performance race. By selecting OpenCL, we hedge our bets on the functional portability

5.2.2 Hardware Layout

Modern GPUs have a memory hierarchy and hardware layout.

5.3 Performance

5.3.1 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).

5.3.2 Expectations in Performance

Many GPU applications claim a 50x or higher speedup. This will never be the case for RBF-FD for the simple reason that the method reduces to an SpMV. The SpMV is a low computational complexity operation with only two operations for every one memory load.

5.4 Targeting the GPU

5.4.1 OpenCL

5.4.2 Naive Kernels

5.4.3 SpMV Formats/Kernels

CSR Bytes:Flop ratio: <http://arxiv.org/pdf/1101.0091v1.pdf>

5.5 Performance Comparison

5.5.1 Performance of Cosine CL vs VCL

5.5.2 VCL Formats Comparison

Our assumption with RBF-FD in this manuscript is that all stencils will have equal size. Due to this, the ELL format is preferred as the default.

We are investigating optimizations that target both GPUs and Phi cards for a class of numerical methods based on Radial Basis Functions (RBFs) to solve Partial Differential Equations. RBF methods are increasingly popular across disciplines due to their low complexity, natural ability to function in higher dimension with minimal requirements for an underlying mesh, and high-order—in many cases, spectral—accuracy. RBF methods can be viewed as generalizations of many traditional methods such as Finite Difference and Finite Element to allow for truly unstructured grids. This generalization allows one to reuse many of the same techniques (e.g., sparse matrices, iterative solvers, domain decompositions, etc.) to efficiently obtain solutions. The variety of hardware available on Cascade will help us establish a clear argument in the choice of accelerator type and resolve the dilemma between

choosing Phi vs GPU for our method. Since RBFs generalize other methods, our results should have broad reaching impact to answer similar questions for related methods.

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.

Compare formats:

- ELL
- COO
- CSR
- Other formats such as HYB, JAD, DIA are considered on the GPU

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.

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 Multiply (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)

When matrix is sparse, a direct LU decomposition causes fill-in on factorization. In some cases the fill-in can be minimal, but in general one must assume that fill in can turn the sparse matrix into a dense matrix. To invert and solve Equation ??, use an iterative solver like GMRES. The GMRES algorithm (described further in Chapter ?? applies successive SpMVs along with other vector operations to converge on a solution. Due to the dominance of SpMV in GMRES, the performance of RBF-FD reduces once again to SpMV.

CHAPTER 6

DISTRIBUTED RBF-FD

Parallelizing RBF-FD in a distributed environment requires three design decisions [21]. First, the problem is partitioned in some fashion to distribute work across multiple processes. Intelligent partitioning impacts load balancing of processors and the ratio of computation versus communication; imbalanced computation can result in excessive delay per iteration as some processors tackle larger problem sizes with others sitting idle. Second, one must determine whether processes have access to all or a subset of node information, solution values, etc. and establish index mappings that translate between a local context and the global problem. Third, the local ordering of indices is established to improve solver efficiency and/or simplify operations. Node ordering is also significant in the future discussion of offloading computation to GPUs as it can help to minimize data transfer between CPU and GPU.

The following sections detail the approach to distributed computing and a few optimizations that allow RBF-FD to scale over a thousand processes. In later chapters these same decisions will tie into the resulting performance of the distributed multi-GPU implementations.

6.1 Partitioning

For ease of development and parallel debugging, partitioning is initially assumed to be linear within one physical direction (typically the x -direction). Figure 6.1 illustrates a partitioning of $N = 10,201$ nodes on the unit sphere onto four CPUs.

Each partition, illustrated as a unique color, contains many *stencil centers*. Although *stencil centers* are contained within a partition, there is no requirement for all *stencil nodes* to be contained within the same partition. As a result, many stencils require information updates from neighboring partitions for nodes that are referred to as *ghost nodes* [?]. In many cases, *ghost nodes* are treated the same as any other stencil node. The CPU process in charge of a partition is fully aware of the ghost node coordinate, current solution value(s), etc.. However, values at ghost nodes are modified by another process, so changes must be explicitly synchronized via an MPI collective for dependent processes to maintain consistency.

In Figure 6.1, alternating representations between node points and interpolated surfaces illustrates the overlap regions where ghost nodes reside. Due to stencil dependencies in

each partition, the overlap region representations are double-wide—i.e., they contain a set of ghost nodes for both the left and right partitions.

As the stencil size increases, the width of the overlap regions relative to total number of nodes on the sphere proportionally increases. In the case of the unit sphere from Figure 6.1, the width of the overlap is roughly \sqrt{n} for stencil size n . Figure 6.1 shows the case of $n = 31$ nodes per stencil. Higher order RBF-FD stencils (i.e., larger stencil sizes) exacerbate the situation by further increasing the number of bytes that must be sent via MPI. Observe that since stencils need not have symmetric dependencies (i.e., if stencil s_1 depends on s_2 , s_2 need not depend on s_1), the number of ghost nodes for each partition can vary.

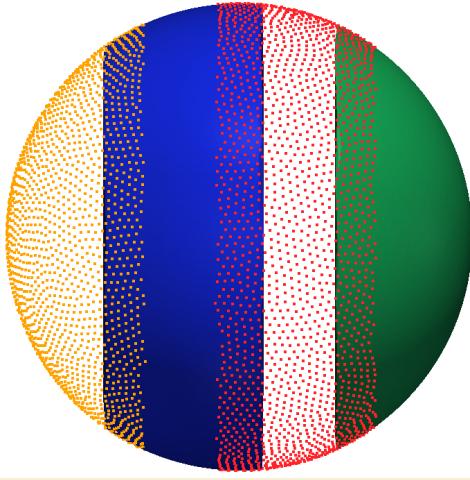


Figure 6.1: Partitioning of $N = 10,201$ nodes to span four processors with stencil size $n = 31$.

The choice for a linear partitioning is simple and easy to code. Each MPI process has a left and right neighbor, so communication is straightforward. On a high number of processors, there are two issues: 1) increasing the number of processors quickly reduces the width of each partition and can result in stencils dependent on more than one partition in each direction introducing the need for more complex collectives; and 2) with near uniform node distributions the resulting partitions are of unequal size and processors are improperly balanced. Thus, in the case of the sphere, linear partitioning is not ideal.

Many other options for partitioning the sphere exist. In atmospheric and geophysical communities for example, one often finds the cubed-sphere [? ?], which transcribes a subdivided cube onto the sphere and assigns projected rectangular elements to individual processors. Another option is the icosahedral geodesic grid [?], which evenly balances the computational load by distributing equal sized geodesic triangles across processors. The options for partitioning the sphere are endless, and are outside the scope of this work.

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.

In order to partition our nodes, METIS requires an undirected adjacency graph repre-

senting the edges that connect nodes. In this case the adjacency graph represents edges connecting nodes in a mesh. For RBF-FD there is no well-defined mesh. Rather, every node is connected to multiple stencil centers. An undirected To produce this we generate

The undirected graph is used only for partitioning and subsequently discarded.

METIS divides stencils into contiguous partitions of nearly equivalent size.

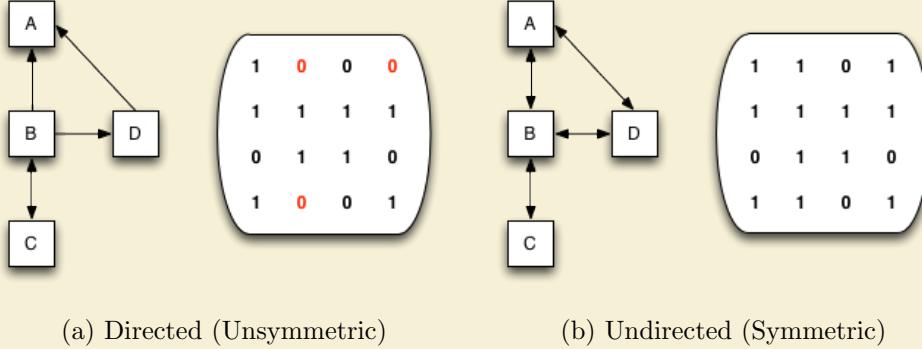


Figure 6.2: A simple adjacency graph and corresponding matrices. Edges connecting nodes of RBF-FD stencils produce (a) a directed adjacency matrix. To partition RBF-FD stencils, METIS requires conversion to (b) an undirected graph/matrix.

In Figure 6.2b new non-zeros are introduced to induce symmetry. Since the goal is to partition the physical domain, this added connectivity is harmless to RBF-FD. However, with respect to load balancing, this may not be ideal. For every new nonzero introduced to a row, METIS assumes an additional node utilized by the partition. When the node is a false connection, it is one fewer centers that METIS will assign to the partition in an attempt to keep all partitions balanced. [Author's Note: show example of 4 or 8 partitions in table to see how Q, O, R can be disproportionate.](#)

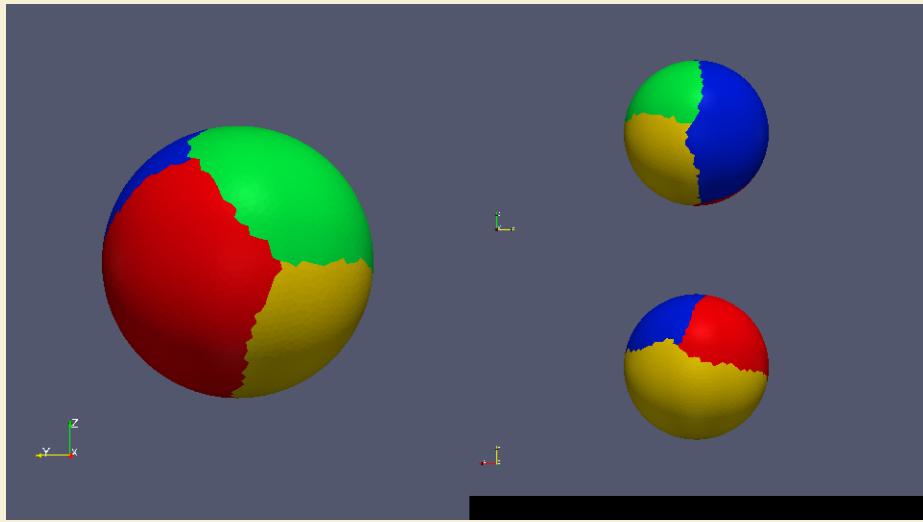


Figure 6.3: METIS partitioning of $N = 10,201$ nodes to span four processors with stencil size $n = 31$.

6.2 Index Mappings and Local Node Ordering

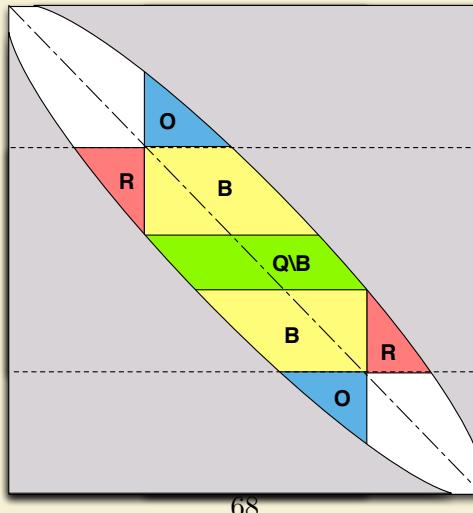


Figure 6.4: Decomposition for one processor selects a subset of rows from the DM. Blocks corresponding to node sets $\mathcal{Q} \setminus \mathcal{B}$, \mathcal{O} , and \mathcal{R} are labeled for clarity. The subdomain for the processor is outlined by dashed lines.

6.3 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 6.1, defines the collection of index lists that each CPU must maintain for both multi-CPU and multi-GPU implementations.

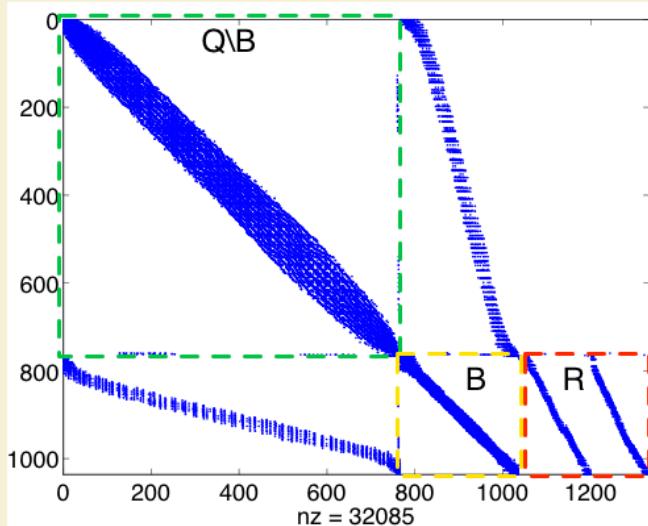


Figure 6.5: Spy of the sub-DM view on processor 3 of 4 from a METIS partitioning of $N = 4096$ nodes with stencil size $n = 31$ and stencils generated with Algorithm ?? ($hnx = 100$). Blocks are highlighted to distinguish node sets $\mathcal{Q} \setminus \mathcal{B}$, \mathcal{B} , and \mathcal{R} . \mathcal{R} Stencils involved in MPI communications have been permuted to the bottom of the matrix. The split in \mathcal{R} indicates communication with two neighboring partitions.

Figure 6.6 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 point to stencil neighbors outside the small window and are not relevant to the following

\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 6.1: Sets defined for stencil distribution to multiple CPUs

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 6.6 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} ,

$$\mathcal{G} = \{\mathcal{Q} \setminus \mathcal{B}, \mathcal{B} \setminus \mathcal{O}, \mathcal{O}, \mathcal{R}\}, \quad (6.1)$$

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

Figure 6.1 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)

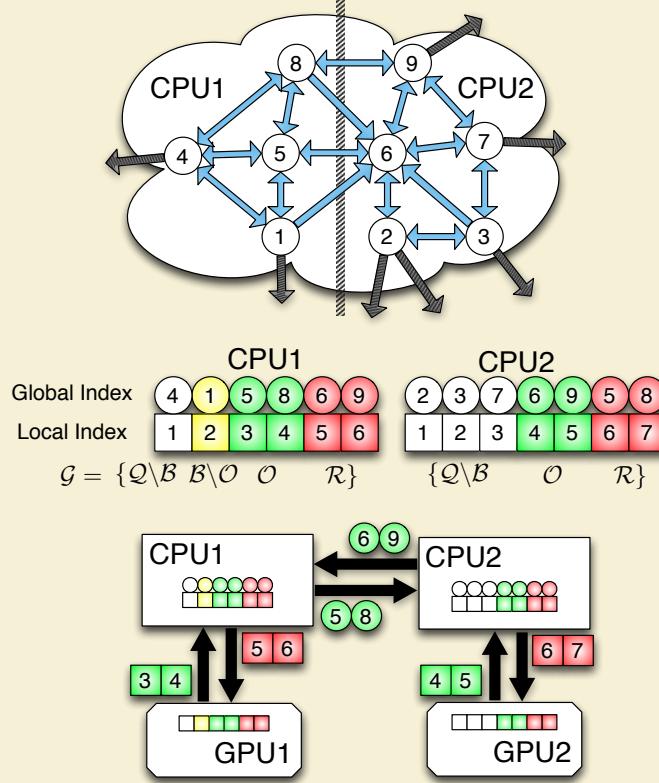


Figure 6.6: Partitioning, index mappings and memory transfers for nine stencils ($n = 5$) spanning two CPUs and two GPUs. Top: the directed graph created by stencil edges is partitioned for two CPUs. Middle: the partitioned stencil centers are reordered locally by each CPU to keep values sent to/received 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 6.1: $Q \setminus B$ is white, $B \setminus O$ is yellow, O is green and R is red.

reside. As stencil size increases, the width of the overlap regions relative to total number of nodes on the sphere also increases.

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

1. Transfer \mathcal{O} from GPU to CPU
2. Distribute \mathcal{O} to other CPUs, receive \mathcal{R} from other CPUs
3. Transfer \mathcal{R} to the GPU
4. Launch a GPU kernel to operate on \mathcal{Q}

The data transfers involved in this process are illustrated at the bottom of Figure 6.6. Each GPU operates on the local indices ordered according to Equation (6.1). The set \mathcal{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 \mathcal{O} to other CPUs. CPUs send only the subset of node values from \mathcal{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 \mathcal{R} is received, the CPU converts back from global to local indices before copying a contiguous block of memory to the GPU. [Author's Note: remember distributed case: no decode](#)

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.

By scalable here we imply total problem size and processor count. The performance scalability of the code depends on the problem size and the MPI collective. In Figure ?? the strong scaling of $N = 10^6$ nodes is tested on Itasca, a supercomputer at the Minnesota Supercomputing Institute.

6.4 Test Case

To test and demonstrate scaling of our method, we consider an idealized regular grid in three dimensions.

verification here is only significant to ensure we are applying all weights. We apply weights to calculate derivatives of a test function in X, Y, Z, and the Laplacian. the grid is regular and 3D. We test strong scaling on a $N = 160^3$ grid, and weak scaling with $N_p = 4000$. This way at $p = 1024$ processes we have weak scaling testing the full $N = 160^3$ grid.

6.5 Communication Collectives

MPI collectives allow information sharing between processes. Our code leverages three collectives: MPI_Alltoall, MPI_Alltoally and MPI_Isend/MPI_Irecv.

The collective operation is essentially transposing information as seen in Figure 6.8.

`MPI_Alltoall` requires that all processors send and receive an equivalent number of bytes to one another. Since the size must be equivalent for all processors, the send and receive buffers are padded to the maximum message size for any one connection between processors. `MPI_Alltoally` reduces the number of bytes sent and received by allowing processors to specify variable message sizes when communicating. For a small number of processors the variable message size will function well. However, `MPI_Alltoally` requires all processes to connect with every other process, even in the event that 0 bytes are to be sent. Based on the grid decomposition, processors compute on contiguous partitions with a small number of neighboring partitions. By replacing the `MPI_Alltoally` with a `MPI_Isend/MPI_Irecv` combination, the number of collective connections are truncated such that processors only connect to and communicate with essential neighbors that need/provide data.

The actual implementation of `MPI_Alltoall` and `MPI_Alltoally` likely use `Isend` and `Irecv` internally.

`MPI_Isend/MPI_Irecv` also allows for overlapping communication and computation by posting receives early

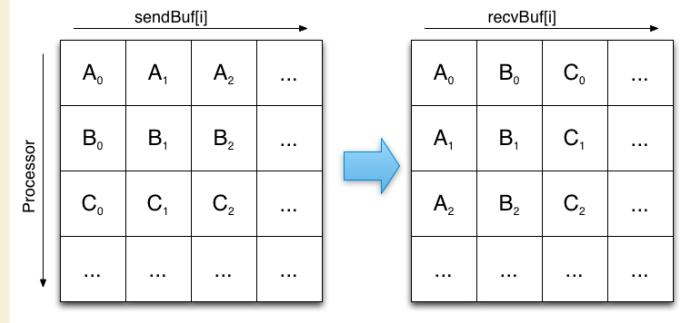


Figure 6.7: The MPI_Alltoall collective allows processors to interchange/transpose data by passing an equivalent number of bytes to every other processor.

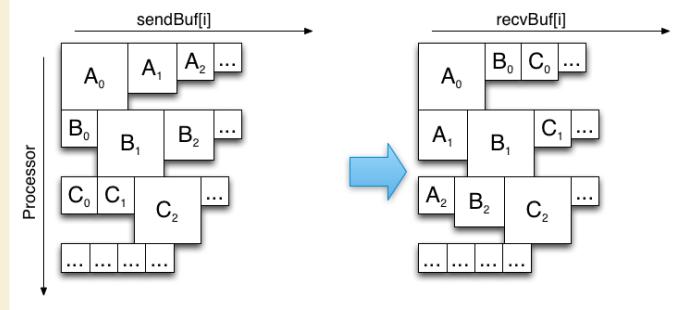


Figure 6.8: The MPI_Alltoally collective compresses the interchange from MPI_Alltoall by allowing for variable message sizes between all processors. Assume message sizes are proportional to square size in figure. When packet sizes are null MPI_Alltoally has undefined behavior.

Figure 6.8: <http://www.mcs.anl.gov/papers/P1699.pdf> observes that the zero-byte messages still add up due to processing required to analyze send-buffer and determine when connection is required.

6.5.1 Alltoally

As a baseline for scaling we start with MPI_Alltoally.

[Author's Note: figure: alloall visual](#)

6.5.2 Isend/Irecv

The first improvement on Alltoally collectives is to truncate the number of connections made between processes. Compact stencils implies an overlap region for each processor that draws values from a limited number of neighboring processors.

[Author's Note: figure: isend visual](#)

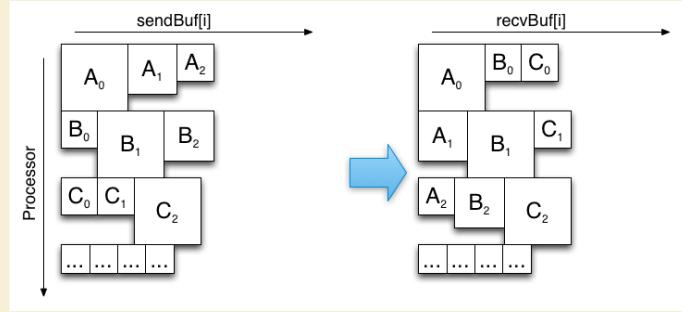


Figure 6.9: The “subset-to-subset” (MPI_Isend/MPI_Irecv) collective allows for variable message sizes, and truncates the number of connections between processors to only required connections.

6.5.3 No Decode

Author’s Note: [figure: per iteration stacked bar for n=50 and 16 processes to show cost of decode](#)
 Author’s Note: [figure: algorithm for collective](#)

Author’s Note: [figure: alltoall to isend improvement. justify comm_combo for up to 16 procs.](#)
 Author’s Note: [figure: comm_combo gains](#)
 Author’s Note: [figure: algorithm for collective](#)

6.5.4 Immediate Isend on Encode

Author’s Note: [figure: algorithm for collective](#)

Author’s Note: [back to section: figure: improvement on all CPU collectives \(n=50\)](#)

Author’s Note: [table: show percentage of comm time for actual mpi time. busy network can cause slower comm times. but the decode cost is gone. it can also be an issue if we have saturated comm pipes](#)

6.6 CPU Scaling

Author’s Note: [Show the strong and weak scaling here](#)

To demonstrate the effectiveness of our decomposition and indexing, we perform scaling experiments.

6.6.1 Strong Scaling

Strong scaling tests the growth in time for a fixed total problem size, and a variable number of processors.

6.6.2 Weak Scaling

Weak scaling considers the amount of time for a fixed problem size per process and variable number of processors. That is to say, each processor has roughly the same amount

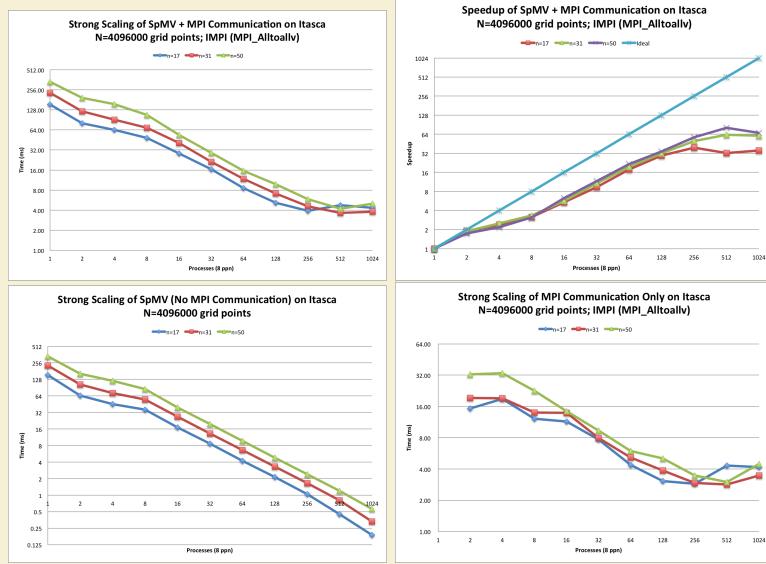


Figure 6.10: Strong scaling the distributed SpMV for $N = 4096000$ nodes (i.e., a 160^3 regular grid) and various stencil sizes. Here the MPI_Alltoally collective operation is used. (Left) Strong scaling of SpMV (including cost of communication). (Center) Strong scaling of computation only. (Right) Strong scaling of communication only.

of work, so as we scale to a large number of processors, changes in time will be the result of increased communication overhead.

Although our weak scaling results are promising, they also contain a problem. First, since we are subsampling a 160^3 regular grid to get the first $N = p * 4000$ nodes, many of the tests consider domains that are “L” shaped and have odd partitions with limited connectivity.

Author’s Note: [Here and strong scaling: table showing the min and max Osize,Rsize](#)

6.6.3 Bandwidth

To understand the impact of MPI on these benchmarks we calculate the average and aggregate collective bandwidths. The average bandwidth considers the MPI throughput from the perspective of one processor.

The aggregate bandwidth reveals when processes saturate the interconnects.

We consider a simple idealized problem where derivatives are computed over a regular grid generated in 3-D. The experiment computes the SpMV one thousand times. At the end of each SpMV the MPI_Alltoallv collective is used to synchronize the local derivative vectors. After one thousand iterations, each process computes the local norm of the resulting vector and an MPI_Reduce collective dra

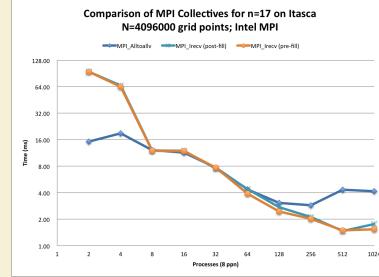


Figure 6.11: Scaling comparison of MPI_Alltoallv and two types of MPI_Irecv/MPI_Ircv collectives: one with MPI_Irecv issued after filling the MPI_Irecv send buffer (post-fill), and the other issued before filling the MPI_Irecv buffer (pre-fill).

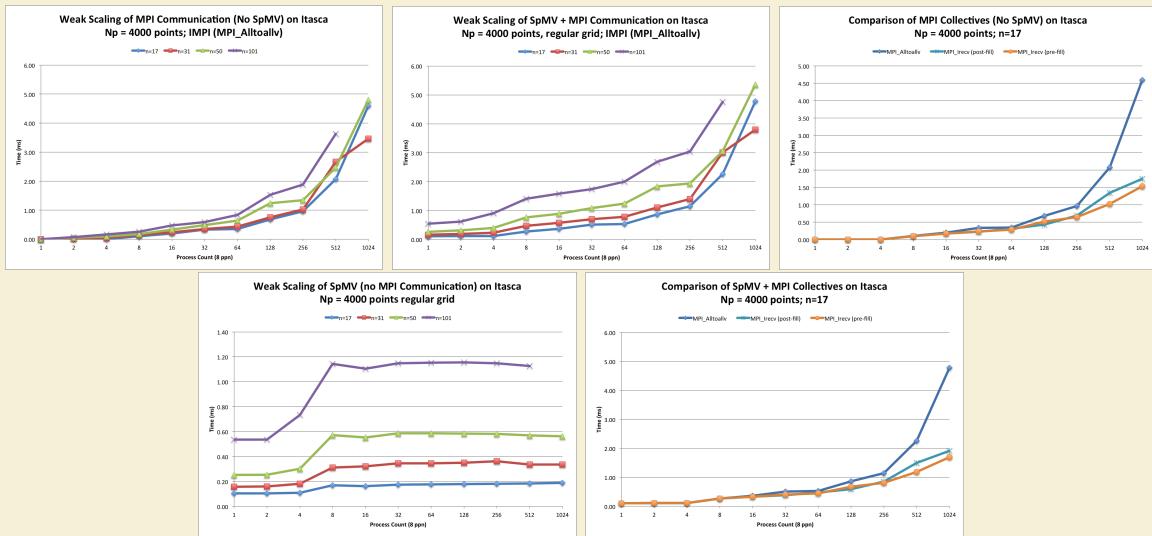


Figure 6.12: Weak scaling of the SpMV

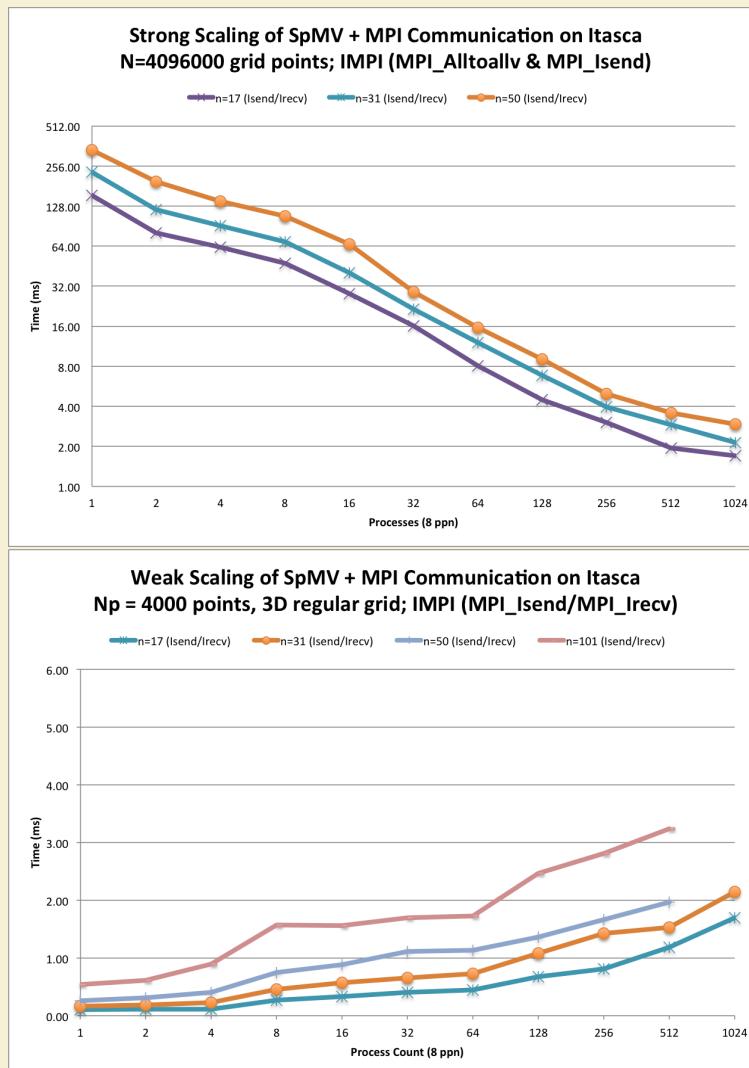


Figure 6.13: Scaling of SpMV with MPI_Isend/MPI_Irecv

CHAPTER 7

DISTRIBUTED GPU SPMV

Distributing SpMV across multiple GPUs poses a new problem: as previously mentioned, the data sent and received via MPI collectives must be copied from device to host and vice-versa. To amortize this cost we introduce a novel overlapping algorithm to hide the cost of communication behind the cost of a concurrent SpMV on the GPU.

7.1 Overlapped Queues

7.2 Avoiding Copy Out

7.2.1 Avoiding Copy-Out on CPU

7.3 Scaling

We scale the SpMV across the GPUs on Cascade.

7.3.1 Fermi

7.3.2 Kepler

7.3.3 Shared K20s

CHAPTER 8

NUMERICAL VALIDATION

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. :

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 of RBFs in solving PDEs on the sphere [6, 8? ?]. 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., [8] and [10]). 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 [?].

8.0.4 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

$$\frac{\partial h}{\partial t} + \frac{u}{\cos \theta} \frac{\partial h}{\partial \lambda} = 0 \quad (8.1)$$

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 (8.1) 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 (8.1) 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 (8.1) is

$$\frac{d\mathbf{h}}{dt} = -\operatorname{diag}(\omega(\theta)) D_\lambda \mathbf{h}. \quad (8.2)$$

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 (8.2) in the form given in (3.14). The scaling parameter γ_c and the order of hyperviscosity k are given in Table 8.1. 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 8.1b shows the effect of hyperviscosity on the eigenvalues of the DM, $-\operatorname{diag}(\omega(\theta)) D_\lambda$, in (8.2).

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 ($(\kappa_A)_j$ is the condition number of the interpolation matrix in (??), representing the j^{th} stencil). For a constant mean condition number, ϵ varies linearly with \sqrt{N} (see [7] 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 [7].

Figure 8.2 shows the solution to Equation (8.1) 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 8.3. The time step $\Delta t = 0.05$ for all resolutions.

Author's Note: Include older figures of convergence without stabilization. Mention that CVT nodes require independent tuning of HV param. Its useful but not incredibly convenient at the moment.

Table 8.1: 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

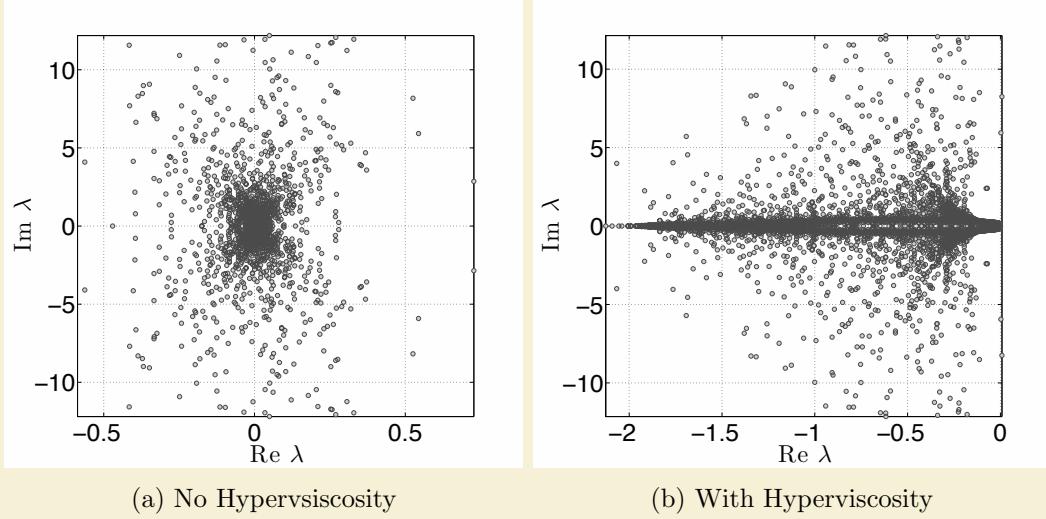


Figure 8.1: Eigenvalues of $\text{diag}(\omega(\theta))D_\lambda$ for the vortex roll-up test case for $N = 4096$ nodes, stencil size $n = 101$ and $\epsilon = 3.5$. Left: no hyperviscosity. Right: hyperviscosity enabled with $k = 4$ and $\gamma_c = 40$. [Author's Note: color](#)

8.0.5 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

$$\frac{\partial h}{\partial t} + \frac{u}{\cos \theta} \frac{\partial h}{\partial \lambda} + v \frac{\partial h}{\partial \theta} = 0, \quad (8.3)$$

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 [8?].

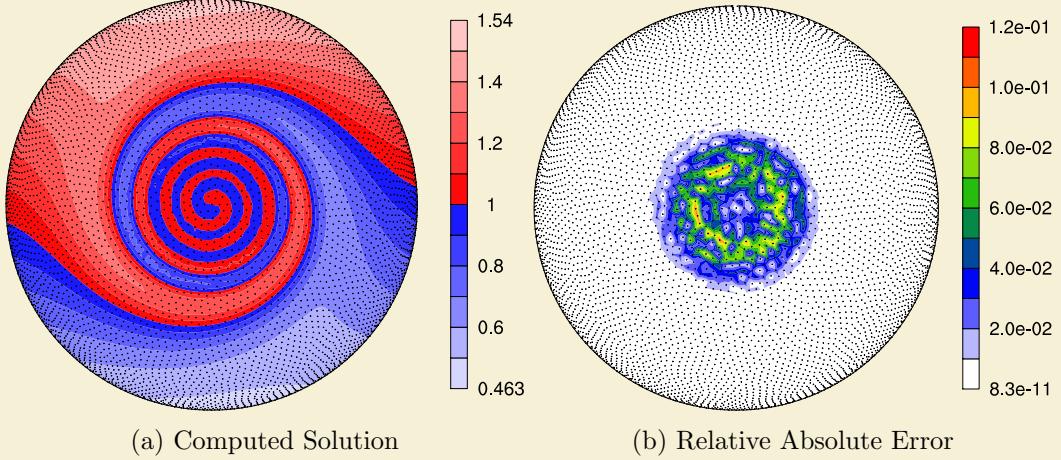


Figure 8.2: Vortex roll-up solution at time $t = 10$ using RBF-FD with $N = 10, 201$ and $n = 50$ point stencil. Normalized ℓ_2 error of solution at $t = 10$ is $1.25(10^{-2})$ [Author's Note: add initial condition figure](#)

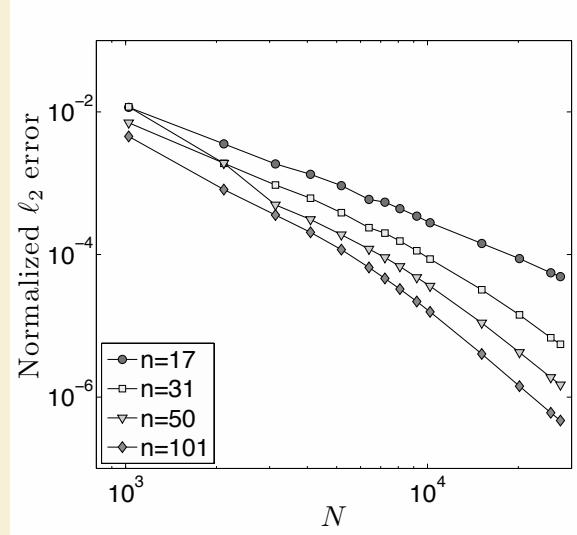


Figure 8.3: Convergence plot for vortex roll-up at $t = 3$. [Author's Note: color](#)

The discretized form of (8.3) is

$$\frac{d\mathbf{h}}{dt} = -\text{diag}\left(\frac{u}{\cos \theta}\right) D_\lambda \mathbf{h} - \text{diag}(v) D_\theta \mathbf{h} \quad (8.4)$$

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 (8.4) into one operator, our implementation evaluates them as two sparse matrix-vector multi-

plies. 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 (8.3) 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 [8, 10], 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 8.2: 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 8.4 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 8.2.

Figure 8.5 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 [10]). After ten revolutions, the cosine bell is still intact. The majority of the absolute error (Figure 8.5b) appears at the base of the C^1 bell where the discontinuity appears in the derivative. At ten revolutions, Figure 8.6 illustrates the convergence of the RBF-FD method. All tests in Figure 8.6 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 [8], 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](#)

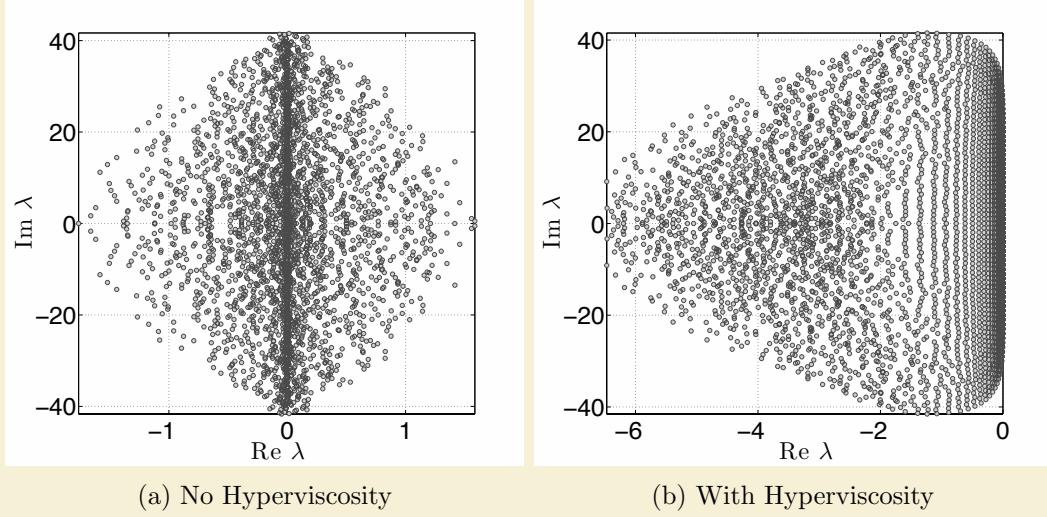


Figure 8.4: Eigenvalues of (8.4) for the cosine bell test case with $N = 4096$ nodes, stencil size $n = 101$, and $\epsilon = 3.5$. Left: no hyperviscosity. Right: hyperviscosity enabled with $k = 8$ and $\gamma_c = 5 * 10^{-2}$. Eigenvalues are divided by u_0 to remove scaling effects of velocity.

Author's Note: [color](#)

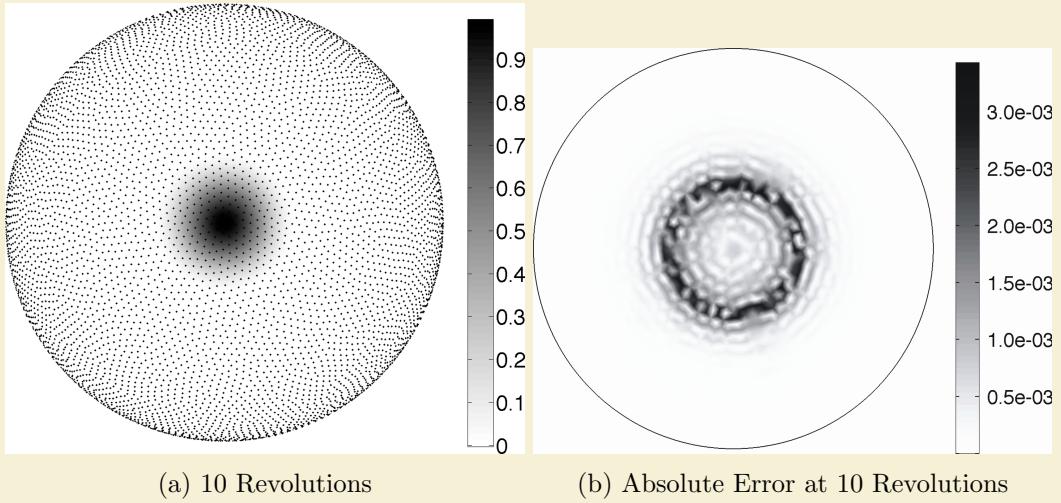


Figure 8.5: Cosine bell solution after 10 revolutions with $N = 10201$ nodes and stencil size $n = 101$. Hyperviscosity parameters are $k = 8$, $\gamma_c = 5(10^{-2})$.

Author's Note: [Insert color figures](#)

[to quantify the appropriate \$dt\$ that would compare RBF-FD to their \$N = 4096\$ case with global RBFs.](#)

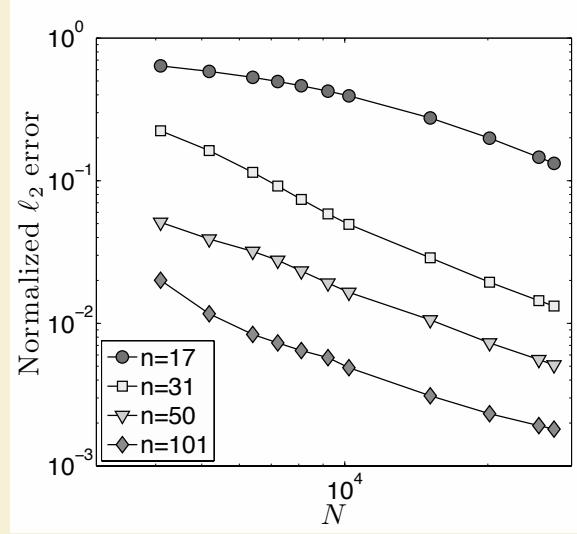


Figure 8.6: Convergence plot for cosine bell advection. Normalized ℓ_2 error at 10 revolutions with hyperviscosity enabled.

8.1 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 8.1. 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 $(\lambda_c, \theta_c) = (3\pi/2, 0)$. The angle $\alpha = \pi/2$ is chosen to transport the bell over the poles of the coordinate system, and $u_0 = 2\pi a / 1036800$.

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

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

8.1.1 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).

CHAPTER 9

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.

9.1 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:

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

9.2 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.

9.2.1 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.

9.2.2 Asynchronous Queueing

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.

9.3 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 9.1 considers some of the more monumental differences between the Fermi and GT200 architectures.

9.3.1 Double Precision

Double precision operations take XXX cycles

9.3.2 Local Caching

Local caching allows us to bypass the need for

9.3.3 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?

9.3.4 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 9.1: Comparison of NVidia’s new Fermi architecture to the GT200 architecture used for GTX 280, Tesla C1060 and other GPUs in use today.

9.4 HPC Spear Cluster

9.5 Keeneland

9.6 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

9.7 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 9.1a 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.

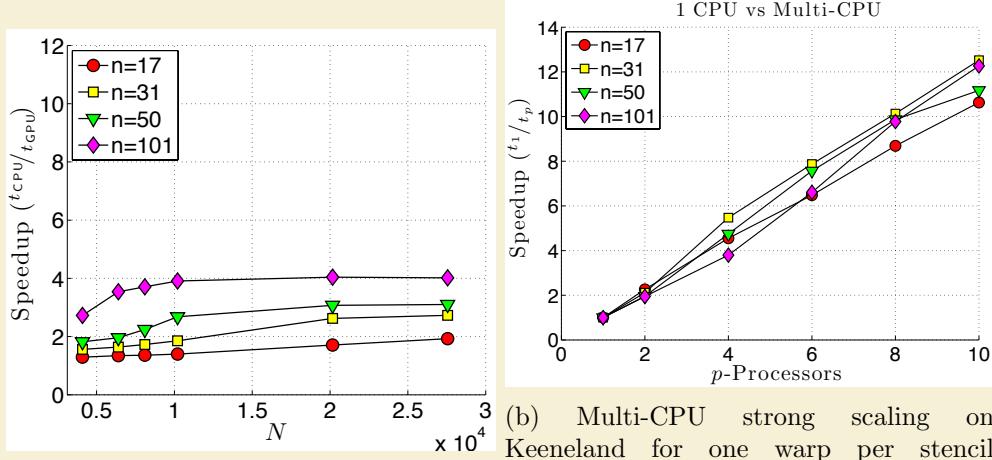
Figure 9.1b shows the strong scalability of our method on Multiple CPUs. In distributed computing, ideal scaling is linear. This figure demonstrates that our method does scale linearly (almost super-linearly) as the number of CPUs increases, so our prospect for spanning all CPUs on Keeneland is within 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 9.1c 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.

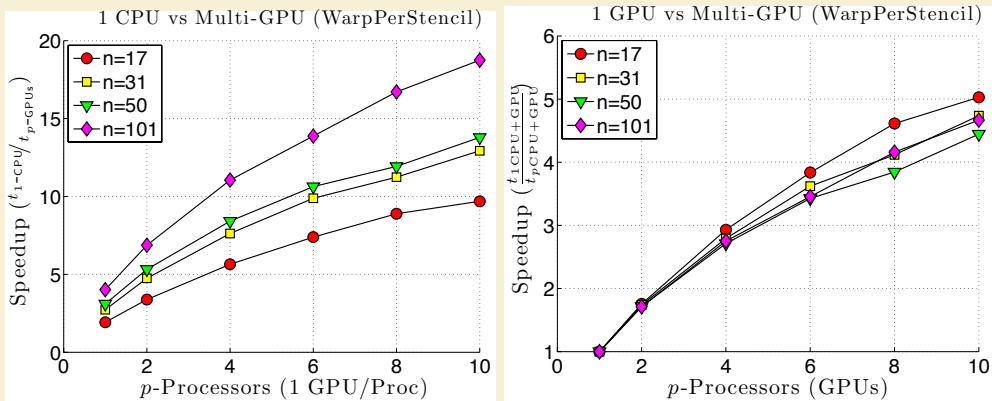
Figure 9.1d shows the scalability of multiple GPUs vs 1 GPU. Here we see a sub-linear behavior for all cases. This is attributed to both the cost of transfer between CPUs and GPUs and the decreasing problem size as number of processors increases, which underutilizes the GPUs.

Figure 9.1e and Figure 9.1f show the smaller percentage of time per iteration dedicated to communication compared to the figures in the paper. In the Figure 9.1f, 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.

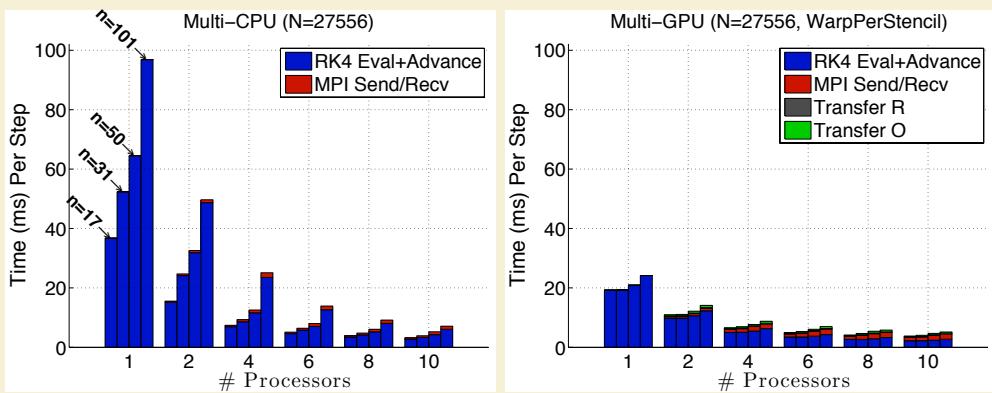
I am generating another set of figures that demonstrate the scaling when we overlap communication and computation. MPI collectives do not allow overlap, but the asynchronous GPU kernel launches do. Therefore, I expect: - the scaling on multiple CPUs vs 1 CPU to be the same as it is now - the scaling 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



(a) One warp per stencil kernel on one GPU in Keeneland
 Author's Note: [Check if this is SpMV only or with comm...](#)



(c) Multi-GPU strong scaling vs one CPU (d) Multi-GPU strong scaling vs one GPU on Keeneland for one warp per stencil on Keeneland for one warp per stencil



(e) Multi-CPU benchmarks by component (f) Multi-GPU benchmarks by component on Keeneland on Keeneland

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

9.8 Asynchronous OpenCL

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

9.9 Multi-Queue OpenCL

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

9.10 GPU Kernel Optimizations

9.10.1 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?

9.10.2 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?

9.10.3 Comparison: custom SpMV for explicit schemes vs ViennaCL

BIBLIOGRAPHY

- [1] Kdtreesearcher class. MATLAB R2013a Documentation (<http://www.mathworks.com/help/stats/kdtreesearcherclass.html>), Aug 2013. 41
- [2] Evan F. Bollig, Natasha Flyer, and Gordon Erlebacher. Solution to PDEs using radial basis function finite-differences (rbf-fd) on multiple GPUs. *Journal of Computational Physics*, 231(21):7133 – 7151, 2012. 33
- [3] Michael Connor and Piyush Kumar. Fast construction of k-nearest neighbor graphs for point clouds. *IEEE Transactions on Visualization and Computer Graphics*, 16:599–608, 2010. 59, 60
- [4] Mark de Berg, Otfried Cheong, Marc van Kreveld, and Mark Overmars. *Computational Geometry: Algorithms and Applications*. Springer-Verlag, 3rd edition, 2008. 43, 44
- [5] Gregory E. Fasshauer. *Meshfree Approximation Methods with MATLAB*, volume 6 of *Interdisciplinary Mathematical Sciences*. World Scientific Publishing Co. Pte. Ltd., 5 Toh Tuck Link, Singapore 596224, 2007. vi, 1, 9, 10, 11, 14, 15, 17, 19, 21, 27, 31, 42, 44
- [6] Natasha Flyer and Erik Lehto. Rotational transport on a sphere: Local node refinement with radial basis functions. *Journal of Computational Physics*, 229(6):1954–1969, March 2010. 2, 9, 15, 59, 79
- [7] Natasha Flyer, Erik Lehto, Sebastien Blaise, Grady B. Wright, and Amik St-Cyr. Rbf-generated finite differences for nonlinear transport on a sphere: shallow water simulations. *Submitted to Elsevier*, pages 1–29, 2011. 15, 26, 27, 32, 33, 34, 38, 39, 42, 44, 80
- [8] Natasha Flyer and Grady B. Wright. Transport schemes on a sphere using radial basis functions. *Journal of Computational Physics*, 226(1):1059 – 1084, 2007. 2, 9, 14, 15, 27, 79, 81, 83
- [9] Natasha Flyer and Grady B. Wright. A Radial Basis Function Method for the Shallow Water Equations on a Sphere. In *Proc. R. Soc. A*, volume 465, pages 1949–1976, December 2009. 2, 9, 14, 15, 32
- [10] Bengt Fornberg and Erik Lehto. Stabilization of RBF-generated finite difference methods for convective PDEs. *Journal of Computational Physics*, 230(6):2270–2285, March 2011. 3, 11, 15, 25, 26, 27, 33, 34, 36, 42, 44, 79, 83

- [11] Jerome H. Friedman, Jon Louis Bentley, and Raphael Ari Finkel. An algorithm for finding best matches in logarithmic expected time. *ACM Trans. Math. Softw.*, 3(3):209–226, September 1977. [44](#)
- [12] Prashant Goswami, Philipp Schlegel, Barbara Solenthaler, and Renato Pajarola. Interactive sph simulation and rendering on the gpu. In *Proceedings of the 2010 ACM SIGGRAPH/Eurographics Symposium on Computer Animation*, SCA ’10, pages 55–64, Aire-la-Ville, Switzerland, Switzerland, 2010. Eurographics Association. [47](#), [53](#), [55](#), [57](#), [60](#)
- [13] S. Green. Cuda particles. NVidia Whitepaper, 2010. [42](#), [45](#), [47](#), [53](#), [55](#)
- [14] N. A. Gumerov, R. Duraiswami, and E. A. Borovikov. Data structures, optimal choice of parameters, and complexity results for generalized multilevel fast multipole methods in d dimensions. Technical Report UMIACS-TR-2003-28, University of Maryland (College Park, Md.), Apr 2003. [41](#), [42](#)
- [15] I. Johnson. Real-time particle systems in the blender game engine. Master’s thesis, Florida State University, November 2011. [42](#), [47](#), [55](#)
- [16] Øystein E. Krog. GPU-based Real-Time Snow Avalanche Simulations. Master’s thesis, Norwegian University of Science and Technology, June 2010. [42](#), [45](#), [46](#), [47](#), [53](#), [55](#)
- [17] Wai-Hung Liu and Andrew H. Sherman. Comparative analysis of the cuthill-mckee and the reverse cuthill-mckee ordering algorithms for sparse matrices. *SIAM Journal on Numerical Analysis*, 13(2):pp. 198–213, Apr 1976. [57](#)
- [18] John Mellor-crummey, David Whalley, and Ken Kennedy. Improving memory hierarchy performance for irregular applications using data and computation reorderings. In *International Journal of Parallel Programming*, pages 425–433, 2001. [56](#)
- [19] Jia Pan and Dinesh Manocha. Fast GPU-based Locality Sensitive Hashing for K-Nearest Neighbor Computation. *Proceedings of the 19th ACM SIGSPATIAL GIS ’11*, 2011. [60](#)
- [20] Timothy J. Purcell, Craig Donner, Mike Cammarano, Henrik Wann Jensen, and Pat Hanrahan. Photon mapping on programmable graphics hardware. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware*, pages 41–50. Eurographics Association, 2003.
- [21] Yousef Saad. *Iterative Methods for Sparse Linear Systems*. Society for Industrial Mathematics, second edition, 2003. [60](#), [65](#)
- [22] Hanan Samet. *Foundations of Multidimensional and Metric Data Structures (The Morgan Kaufmann Series in Computer Graphics and Geometric Modeling)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2005. [42](#), [43](#), [44](#), [45](#), [47](#), [48](#)
- [23] Steven Skiena. *The Algorithm Design Manual (2. ed.)*. Springer, 2008. [42](#), [43](#)

- [24] Andrea Tagliasacchi. kd-tree for matlab. <http://www.mathworks.com/matlabcentral/fileexchange/21512-kd-tree-for-matlab>, Sep 2010. 41, 42, 51
- [25] Andrea Tagliasacchi. kd-tree matlab. <https://code.google.com/p/kdtree-matlab>, Jun 2012. 44, 51
- [26] Holger Wendland. Fast evaluation of radial basis functions: Methods based on partition of unity. In *Approximation Theory X: Wavelets, Splines, and Applications*, pages 473–483. Vanderbilt University Press, 2002. 41, 42, 45, 47
- [27] Holger Wendland. *Scattered Data Approximation*. Cambridge Monographs on Applied and Computational Mathematics. Cambridge University Press, 2005. 41, 42, 45, 47
- [28] Lexing Ying. A kernel independent fast multipole algorithm for radial basis functions. *Journal of Computational Physics*, 213(2):451 – 457, 2006. 41, 42, 43