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, 2013

Evan F. Bollig defended this dissertation on November 6, 2013.
The members of the supervisory committee were:

Gordon Erlebacher

Professor Directing Thesis

Mark Sussman

University Representative

Natasha Flyer

Committee Member

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 university requirements.

This work is dedicated to the one who taught me patience, compassion, bravery, and determination. Thank you, mom.

–E

ACKNOWLEDGMENTS

I feel compelled to start by thanking my major professor, Dr. Gordon Erlebacher, who shared with me a unique attitude and excitement in regard to research. After years of collaboration, I find myself echoing his own audacious style in questioning the work done by others; never quite satisfied by their responses. Without his supervision and constant prodding this dissertation would not have been possible.

Of no lesser significance is Dr. Natasha Flyer, who is not formally listed as a co-advisor due to restrictions at the university level, but who has in all capacities fulfilled that role. I am sincerely grateful for the time spent at NCAR under her tutelage. In addition to fortifying my understanding of RBF methods and geophysical PDEs, Natasha shared a number of unforgettable life-lessons that reflect in my choice of employment and pursuit of happiness today.

I also extend my deepest gratitude to the remainder of my committee for their patience and willingness to participate in this venture.

In no particular order, I am indebted to:

- Drs. David Yuen, Bengt Fornberg, Grady Wright, John Burkardt and Kiran Katta for their support and feedback on numerical modeling;
- the RBF research group at CU-Boulder for a number of ideas that led to interesting investigations;
- Dr. Geoff Womeldorff for insightful discussion and the high resolution Spherical CVT meshes tested herein;
- Steven Henke for a number of discussions regarding our concurrent investigations into space-filling curves, neighbor queries, and GPU computing;
- Brent Swartz, Yectli Huerta, Andrew Ring, and the rest of the University of Minnesota Supercomputing Institute for playing sounding board to my ideas and providing access to computational resources;
- the faculty and students in the FSU Department of Scientific Computing (DSC) who built a community of collaboration and interdisciplinary research that I am proud to be part of;
- the DSC staff for their tireless support and assistance;

- and past and present students in Gordon's research group (esp. Ian Johnson, Andrew Young, and Nathan Crock) for brainstorming sessions, weekend hack-a-thons and camaraderie.

This research was funded in part by NSF awards DMS-#0934331 (FSU), DMS-#0934317 (NCAR) and ATM-#0602100 (NCAR). It used resources of the University of Minnesota Supercomputing Institute, as well as the Keeneland Computing Facility at the Georgia Institute of Technology, which is supported by the National Science Foundation under Contract OCI-0910735.

I now close with a very special thank you to my loved ones (both friends and family) who have supported me during the course of this work. You have waited patiently as I devoted all time and energy into completing this milestone. I look forward to resuming life with all of you in it.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ALGORITHMS

ABSTRACT

Many numerical methods based on Radial Basis Functions (RBFs) are gaining popularity in the geosciences due to their competitive accuracy, functionality on unstructured meshes, and natural extension into higher dimensions. One method in particular, the Radial Basis Function-generated Finite Differences (RBF-FD), is drawing attention due to its comparatively low computational complexity versus other RBF methods, high-order accuracy (6th to 10th order is common), and parallel nature.

Similar to classical Finite Differences (FD), RBF-FD computes weighted differences of stencil node values to approximate derivatives at stencil centers. The method differs from classical FD in that the test functions used to calculate the differentiation weights are n-dimensional RBFs rather than one-dimensional polynomials. This allows for generalization to n-dimensional space on completely scattered node layouts.

Although RBF-FD was first proposed nearly a decade ago, it is only now gaining a critical mass to compete against well known competitors in modeling like FD, Finite Volume and Finite Element. To truly contend, RBF-FD must transition from single threaded MATLAB environments to large-scale parallel architectures.

Many HPC systems around the world have made the transition to Graphics Processing Unit (GPU) accelerators as a solution for added parallelism and higher throughput. Some systems offer significantly more GPUs than CPUs. As the problem size, N, grows larger, it behooves us to work on parallel architectures, be it CPUs or GPUs. In addition to demonstrating the ability to scale to hundreds or thousands of compute nodes, this work introduces parallelization strategies that span RBF-FD across multi-GPU clusters. The stability and accuracy of the parallel implementation is verified through the explicit solution of two PDEs. Additionally, a parallel implementation for implicit solutions is introduced as part of continued research efforts.

This work establishes RBF-FD as a contender in the arena of distributed HPC numerical methods.

CHAPTER 1

INTRODUCTION

A plethora of scientific problems 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 some discretization 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 have surfaced, called *meshfree methods*, which discard the requirement for connectivity and operate only on point clouds. Each class of methods offers numerous and, in many ways, complementary benefits. This work focuses on *Radial Basis Function-generated Finite Differences (RBF-FD)*; a method that draws inspiration from both meshed and meshfree methods.

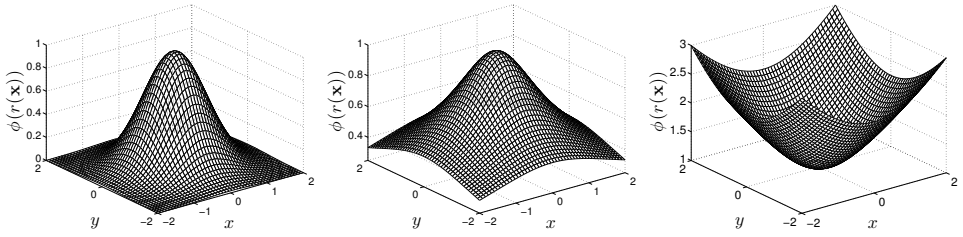
For decades meshed methods like Finite Difference, Finite Element and Finite Volume have been the powerhouses in computational modeling. The well-studied Eulerian schemes, with structured and stationary nodes, come backed by a vast literature on topics related to solvers, preconditioners, parallelization, etc. Unfortunately, those methods often come with many restrictions, be it limitations on node placement or other concerns like difficulties in scaling to higher dimensions. In the ideal case 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.

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:

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

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 to interpolate between nodes in n-dimensional space.



(a) Gaussian (b) Inverse Multiquadric (c) Multiquadric

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

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-FD) leverage the basis functions to compute stencil weights, which in turn are used in generalized 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 with minimal increase in programming complexity [52, 168]. In addition to competitive accuracy and convergence compared with other state-of-the-art methods [49, 50, 52, 53, 168], RBF methods boast stability for large time steps. Not all RBF methods are truly meshless. Most of them operate the same given an input set of nodes, whether or not the input includes an underlying mesh. However, some methods connect all nodes to all others (to be meshless), while methods like RBF-FD rely on stencils to connect points within small neighborhoods of one another. The generated stencils limit the scope of influence for nodes and complexity of the method, but also simulate a mesh.

To track the history of RBF methods, one must rewind to 1971 and R.L. Hardy's seminal research on interpolation with multi-quadric basis functions [78]. The transition from interpolation to RBF PDE methods dates back to 1990 [92, 93]. The RBF-FD method was first introduced in concept in 2000 [150], but it took another few years for the method to really get a start ([136], [151], [167] and [26]). Now a little over a decade old, the method is finally showing signs that it has the critical-mass following necessary for 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 into tens of thousands of nodes. Similar to most RBF methods, RBF-FD is predominantly implemented within small-scale, serial computing environments. The RBF community at-large continues to depend on MATLAB for investigation and extension development.

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. As part of this 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. Driven first by the gaming industry's insatiable desire for more realistic graphics, and now also by the scientific community's demand for massive parallelism, GPU performance in the recent years has tremendously grown compared to CPUs. Accelerators represent the latest trend in HPC, where compute nodes are commonly supplemented by one or more accessory boards for offloading compute intensive tasks. 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 exa-scale computing barrier [9].

Parallelization of RBF-FD is achieved at two levels. On the first level, the physical domain of the problem is partitioned into overlapping subdomains, each handled by different MPI processes. All processes operate independently to compute/load RBF-FD stencil weights, run diagnostic tests and perform other initialization tasks. A process only computes weights corresponding to stencils centered in the interior of its partition. After initialization, processes work together to concurrently solve the PDE. Communication barriers ensure that processes execute in lockstep and maintain consistent solution values across the domain.

The second level of parallelization offloads tasks to the GPU at each iteration of the PDE solver. The bulk of computation in RBF-FD relies on a *Sparse Matrix-Vector multiply (SpMV)* to evaluate derivatives. When mapped to the GPU, the SpMV operation is broken into two parts in order to overlap communication and computation, and amortize the cost of data transfer across multiple levels of hardware (see Chapter 7).

The layout of this document is as follows. This chapter continues with a survey of work related to 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 new algorithm for generating RBF-FD stencils as a faster alternative to the RBF community favorite, k-D Tree. 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 RBF-FD on a multi-GPU cluster. In Chapter 8 the parallel RBF-FD implementation is applied and verified on the solutions of both explicit and implicit geophysical PDEs. Finally, Chapter 10 concludes with a summary of novel contributions, results and a discussion on future directions.

1.1 Related Works: Parallel/Distributed PDE Methods

With the large following that RBF methods have gained since their inception in the 1970s, it is impossible to exhaustively list all applications, advances and related methods. Rather, this section concentrates specifically on RBFs applied to PDEs, and branches out to additional topics where necessary.

Table 1.1 provides a number of related works classified according to the type of parallelism they exhibit. The table presents investigations from both inside and outside the RBF community partitioned into one of three types: multi-CPU, single GPU, and multi-GPU. Initial classifications are refined by parallel language/standard employed. The multi-CPU class includes any investigations that are parallelized by OpenMP, MPI, or the combination of the two, but do not include details of GPU computing. If GPUs (one or more) are used, then we categorize based on the GPU language: CUDA, OpenCL, or "Other". The "Other" category lumps together all languages pre-dating CUDA, including shader-based languages (e.g., GLSL, HLSL, Cg), and early streaming languages (e.g., Brook+, Sh/RapidMind). Multi-GPU implementations match each GPU with a CPU which managed communication. Communication can be overlapped, implying that the GPU continues computing on a local problem while the CPU waits on data, or non-overlapped if both the GPU and CPU wait.

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 ([39]). 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. [10]. Their work

Table 1.1: Classification of references by numerical method and parallelization strategy.

	Multi-CPU (⋆:MPI+OpenMP)		Single GPU			Multi-GPU (†:Non-overlapped, ‡:Overlapped)	
	OpenMP	MPI	CUDA	OpenCL	Other	CUDA	OpenCL
RBF-FD							[21]† [This Work]‡
RBF PDE Methods	[100]	[39, 40, 84, 85, 173]	[75, 76, 132, 133]				
Non-PDE RBF Methods		[145]⋆	[33, 34]		[23, 25, 32, 158]		
Non-RBF PDE Methods		[134]⋆	[31]			[68, 69, 70, 71, 99, 122, 149]† [97, 98, 110, 121, 174]‡	

provided a solution to solve problem sizes into the millions of nodes.

This work leverages a domain decomposition; although, not for the purpose of conditioning. Instead, subdomains are distributed across multiple compute nodes and solved simultaneously. MPI communicates data dependencies across subdomain boundaries. Our implementation of RBF-FD is demonstrated to scale across more than a thousand CPU cores of an HPC cluster. In addition to high scalability for distributed computing, our novel implementation offloads computation to the GPU with an algorithm that overlaps communication and computation. This combination of multi-CPU and multi-GPU parallelism is unmatched in related work within the RBF-FD community. In fact, the multi-GPU aspect sets this work apart from the entire class of RBF methods.

1.1.1 RBF Methods on Multi-Core and Multi-CPU

Kosec and Šarler [100] have the only known (to our knowledge) OpenMP implementation for RBF PDE methods. 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.

The performance and scaling of a parallel RBF neural network is tested on a SunFire 6800 SMP by Strey [145]. Scaling benchmarks for OpenMP, MPI, and MPI+OpenMP versions of his code demonstrate the performance up to 12 of the 24 CPUs within the system.

Global RBF collocation method is parallelized in [84, 85] with a Schwarz Neumann-Neumann domain decomposition method to solve 3-D diffusion equations. In both cases the authors utilize MPI for communication and provide only coarse grained scaling data at 8, 27 and 64 nodes.

Divo and Kassab [40] develop a domain decomposition method with artificial subdomain boundaries and parallelize global RBF collocation across a small cluster of 10 compute nodes. Subdomains are processed independently, with derivative values at artificial boundary points averaged to maintain global consistency of physical values. The authors focus predominantly on verification of the decomposition method for heat transfer problems, and neglect thorough benchmarking and scalability testing for the MPI implementation. In [39], Divo and Kassab apply their domain decomposition to parallelize a local RBF collocation method. The method was run on 36 node cluster, but once again benchmarks and scalability tests are not provided.

Perhaps the most competitive parallel implementation of an RBF method is the PetRBF branch of PETSc [173]. The authors of PetRBF 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. 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. Strong scaling was also tested on a Cray XT4, where strong scaling tops out at 36% for 128 processes, a respectable number—and similar to observed results for our own code on for the same number of processes (see Chapter 6).

Stevens et al. [143] mention a parallel implementation under development, but no document is available at this time.

1.1.2 RBF Methods on GPUs

While a number of related works consider RBFs on GPUs, most of the investigations relate to visualization (e.g., [33, 158]), surface reconstruction (e.g., [25, 32]), and neural networks (e.g., [23]). Of these, only [33] uses a modern GPGPU langauge (CUDA), the rest rely on shader languages (i.e., GLSL, HLSL, etc) denoted as "Other" in Table 1.1.

The only known applications of GPUs to solve PDEs with RBFs are found in [75, 76, 132, 133].

In the case of [132, 133], Schmidt et al. compare implementations of a global RBF collocation scheme to a staggered leapfrog FD scheme. The authors solve the 2-D (heightfield) shallow water equations for Tsunami simulation with both methods. Time-stepping is controlled by a 4^{th} order Runge-Kutta scheme; the same scheme is adopted here. Results for a problem size of 32×32 RBF centers show similar error to a 185×185 staggered leapfrog discretization [133]. Their codes were written in MATLAB and leveraged the AccelerEyes Jacket [5] add-on to provide wrappers and data structures for internal calls to the NVidia CUDA API. The 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 (no GFLOP/sec provided) 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 promising, but no further contribution was made on the topic.

In [75, 76], Gumerov et al. designed a library of FORTRAN wrappers around CUDA that interface with existing code. The wrappers allow existing code bases to leverage GPUs with minimal modification. Wrapping the cuFFT (an FFTW implementation for the GPU [115]) and cuBLAS libraries the authors present accelerated implementations of both a pseudo-spectral plasma turbulence problem (nonlinear PDE), and a Fast Multipole Method that depends internally on an iterative solution to RBF interpolation. Their Fast Multipole Method is only distantly related to the RBF PDE methods discussed in Chapter 2. Moreover, the implementation again depends on dense matrix algebra.

While our method and [75, 76, 132, 133] are all based on RBFs, the problems are only distantly related on the GPU. Our work relies on sparse matrix operations, while the others operate on dense matrices. Dense matrix operations have a high computational complexity and are compute-bound problems that can fully utilize the hardware. They are considered ideal (or near to) by linear algebra libraries like BLAS [105] and LAPACK [6], and were demonstrated to fit well on GPUs in the early days of computer graphics, long before NVidia provided the efficient cuBLAS library [118]—a BLAS substitute on the GPU—in the initial public release of CUDA in 2006. In stark contrast to this, sparse problems have much lower computational complexity and are memory

bandwidth-bound. Getting good performance on sparse problems requires an entirely different set of rules.

Earlier this year (2013), Cuomo et al. [34] implemented RBF-interpolation on the GPU for surface reconstruction. Their work is based on an extension of PetRBF [173], and new built-in back-ends that allow GPU access within PETSc ([110]). PETSc now internally wraps both the CUSP [12] (CUDA) and ViennaCL [125, 126] (OpenCL) projects for sparse matrix algebra on the GPU. With the help of the CUSP backend, 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 accelerates their solutions between 6x and 25x. Unfortunately, the authors do not show any evidence of scaling the interpolation to multiple GPUs. Therefore, while evidence exists that PetRBF now has full GPU support, its scalability remains an open question.

Aside from our own investigations, we know of no other project that applies the GPU to the RBF-FD method.

1.1.3 Non-RBF Methods

A number of GPU implementations unrelated to RBF methods can provide context for the relevancy of our research.

For example, a single-GPU finite-volume implementation introduced by Corrigan et al. in [31] solves 3-D incompressible fluid problems on an unstructured grid. Within their GPU kernels, each element of the unstructured mesh is operated on by a single thread—this design is equivalent to our "one thread per stencil" test case in Chapter 5. We find that operating with a full warp per stencil is more efficient for stencils $n > 32$. The authors' effort to operate on unstructured grids also posed a challenge in that the unstructured elements with random ordering caused considerable slow-down due to non-coalesced memory loads. In response to this, Corrigan et al. reorder their meshes with a fixed-grid method similar to our approach in Chapter 4.

One of the earliest multi-GPU implementations, presented in a series of articles [68, 69, 70, 71] by Göddeke et al., was a GPU-enhanced version of the pre-existing distributed (with MPI) multigrid finite element package, FEAST. In [70] the authors integrated GPUs in a minimally invasive fashion (e.g., farming out local multigrid tasks to the GPU) and considered the weak scalability (i.e., increasing the number of co-processors while keeping the problem size fixed on each processor) on

up to 160 compute nodes with a single GPU each. Since the accelerated routines were implemented at a lower level than the original FEAST API, it was demonstrated in [71] that existing applications built on top of the original FEAST could benefit without code modifications when the new package, FEASTGPU, was substituted. [69] compared performance of single-, double- and mixed-precision calculations within FEASTGPU. The authors then applied FEASTGPU to nonlinear Navier-Stokes problems and found that the minimally invasive approach had limited impact on performance in the new setting where computational work was no longer concentrated within GPU-accelerated tasks [68].

Author's Note: Review from here to next section: At roughly the same time, a number of other efforts around the world started targeting multi-GPU clusters. Many (e.g., [99, 122, 149]) adopted non-overlapping communication and computation, only to realize the necessity for overlap in order to increase parallel efficiency. The challenges in multi-GPU computing are: a) to hide transfer latency for memory copies between CPU and GPU (and vice-versa), and b) to shrink the visible communication time, which suddenly appears larger against the accelerated GPU computation. Phillips et al. draw this conclusion based on scaling of their finite-volume method for compressible flow equations on an irregular structured grid on eight quad-core workstations with a total of 32 GPUs. Thibault et al. [149] also note the need for overlap when modeling 3-D incompressible fluids with finite differences on structured grids and a Jacobi iteration.

In another series of articles, Komatitsch et al. [97, 98, 99] ported a well-tuned spectral-element code named SPECFEM to the GPU. The code was initially offloaded in [99] and demonstrated up to 25x speedup over a serial CPU version. In [97, 98] the authors span computation across GPU clusters. Aided by non-blocking MPI communication and a sufficiently large problem, roughly 90% of communication time is demonstrated to be hidden during weak scaling tests.

Phillips and Fatica overlap communication and computation in [121] while solving the Himeno benchmark—a test specifically designed to compare performance of processors based on a memory-bound application (in this case the 3-D Poisson problem in generalized coordinates).

Most recently, the PETSc library gained GPU support [110, 174]. PETSc is a widely-used library with proven scalability on many supercomputing clusters that provides distributed sparse and dense matrix algebra, Krylov solvers, and preconditioners.

1.2 Major Contributions

Within this document readers should expect to find the following major contributions:

1. The first application of a fixed-grid neighbor query algorithm (popularized by distantly related particle methods) to generate stencils for RBF-FD.
    - Performance comparison with the RBF community favorite, k-D Tree, shows that the lower complexity fixed-grid method is able to achieve up to 2.5x speedup over the former method on randomly distributed nodes.
    - The fixed-grid method is also shown to accelerate RBF-FD time-step performance by up to 5x due spatial locality of node information in memory and improved cache effects.
2. Integer dilation and bit interleaving are introduced to the RBF community at large to construct a number of space-filling curve variants in 2-D and 3-D. The curves reorder cells of the fixed-grid algorithm on the prospect of additional cache hits for derivative calculations.
    - A comparison of the variants to the result of the well-known Reverse Cuthill-McKee (RCM) algorithm concludes that RCM remains the better option for reordering and matrix bandwidth minimization.
3. The design and tuning of the only known implementation of the RBF-FD method to scale across multiple compute nodes of a supercomputing cluster.
    - Our home-grown implementation distributes computation with a Restricted Additive Schwarz (RAS) domain decomposition in the first application to RBF-FD.
    - As part of the tuning process described herein, balanced workloads are achieved with the help of the METIS graph partitioning algorithm [94].
    - A number of steps are taken to reduce communication times in the CPU-only implementation. The resulting algorithm splits derivative calculations into two steps and overlaps communication with computation. We observe that up to 80% of the cost in communication can be hidden in some cases.
    - Scaling benchmarks up to 1024 processes (divided into 8 processes per node) and a grid resolution of $N = 160^3$ vertices (i.e., 4.1 million) prove that the implementation scales well in both a strong and weak sense.
4. The design and tuning of the only known single- and multi-GPU implementation of RBF-FD. Also, ours is the only known investigation in the broader RBF community to target multiple GPUs.

- Our first paper ([21]) introduced a set of custom OpenCL kernels for RBF-FD time-stepping of hyperbolic PDEs with an explicit RK4 scheme (up to 10 GPUs).
- Herein, the performance per GPU is tuned by choosing an alternative sparse matrix representation. The investigation reveals that the Ellpack (ELL) structure provided by the ViennaCL library [125] is a great fit for RBF-FD differentiation matrices, with over 4x faster performance than the original compressed-row storage (CSR).
- The multi-GPU implementation is extended to a novel overlapping algorithm that naturally derives from our distributed CPU optimizations. Non-blocking MPI communication plus two asynchronous OpenCL queues amortize the cost of data transfer between CPU and GPU, MPI communication, and in some cases a substantial amount of computation. Iterations of the resulting implementation achieve an average of 3x improvement in strong scaling compared to a non-overlapping equivalent.

Of lesser significance we also:

1. Verify our explicit solvers against two well studied hyperbolic PDEs (Vortex Roll-up [113, 114] and Cosine Bell Advection [88]).
   - Convergence studies confirm that hyperviscosity ([58]) stabilizes solutions.
   - Tuned parameters are provided for selecting the RBF support parameter and scaling hyperviscosity as a function of the problem size.
2. Implement a distributed multi-GPU preconditioned GMRES within ViennaCL for implicit RBF-FD solutions.
   - A divergence-free field is constructed as the manufactured solution for a simplified version of Stokes flow constrained to the unit sphere.
   - Contributed a CPU-only preconditioner (Incomplete LU with Zero Fill-in) to ViennaCL to test impact on GMRES convergence rates.
3. Provide preliminary benchmarks evaluating the performance of RBF-FD on the Intel Phi Architecture.

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 [92, 93]. 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 [78].

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 [52, 168]. In addition to competitive accuracy and convergence compared with other state-of-the-art methods [49, 50, 52, 53, 168], 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. The general approximation problem is then introduced, with a look at the core of all three method classificiations: 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 [46, 86]. The example 2-D 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 [131]. The conditioning of the system also dramatically

degrades as the number of nodes in the problem increases. Fortunately, recent algorithms such as Contour-Padé [63] and RBF-QR [57, 61] allow for numerically stable computation of interpolants in the nearly flat RBF regime (i.e., $\epsilon \to 0$) where high accuracy has been observed [62, 103].

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 [55].

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; 2-D 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:

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

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 [46, 62]. ε is the support parameter. All RBFs have global support. For compact support, enforce a cut-off radius (see Equation 2.1).

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 2-D using 15 Gaussians is shown in Figure 2.3.

RBFs have been shown in some cases to have exponential convergence for function approximation [46]. 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 [45]. 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 [46, 86] 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 in most cases, the accuracy of the interpolation, but worsens the conditioning of the RBF interpolation problem [131]. This inverse relationship is widely known as the *Uncertainty Relation* [86, 131]. Fortunately, recent algorithms such as Contour-Padé [63] and RBF-QR [57, 61] allow for numerically stable computation of interpolants in the nearly flat RBF regime (i.e., $\epsilon \to 0$) where high accuracy has been observed [62, 103].

RBF methods for interpolation first appeared in 1971 with Hardy's seminal research on multiquadrics [78]. In his 1982 survey of scattered data interpolation methods [65], Franke rated multiquadrics first-in-class against 28 other methods (3 of which were RBFs) [65]. Many other RBFs, including those presented in Table 2.1 have been applied in the literature, but for PDEs in particular, few can rival the attention received by multiquadrics. Recently, however, Gaussian RBFs

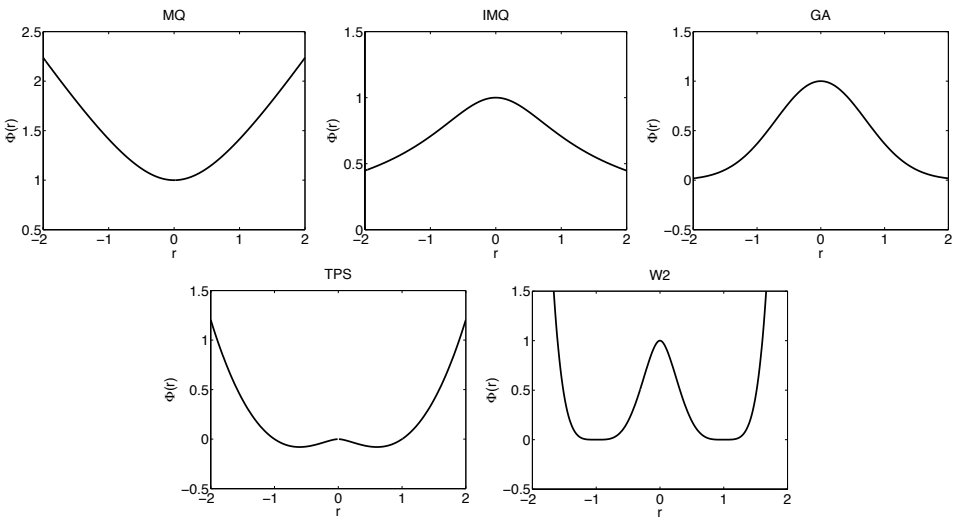


Figure 2.1: Example RBF shapes from Table 2.1 with parameter $\varepsilon = 1$.

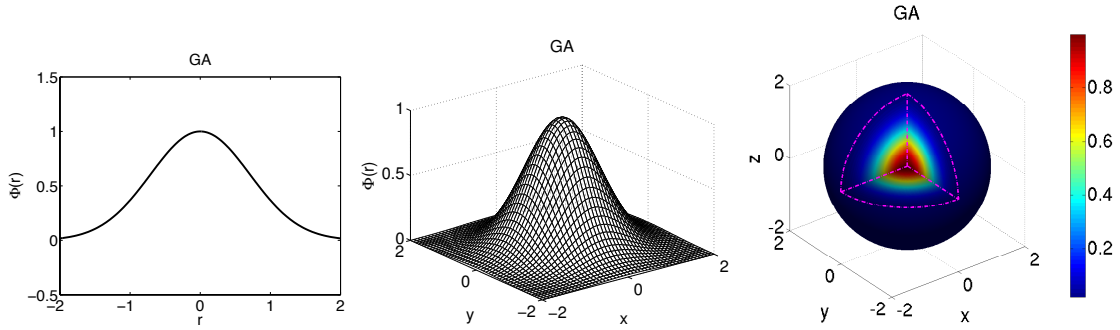


Figure 2.2: The Gaussian (GA) RBF (Table 2.1) with parameter $\varepsilon = 1$ and r in $D = 1$, 2 and 3.

are on the rise due to recent advances in eigenvalue stabilization and new methods for investigating the $\epsilon \to 0$ regime (see e.g., [57, 59]).

By 1990, the understanding of the scientific community regarding RBFs was sufficiently developed for collocating PDEs [92, 93]. 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:

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

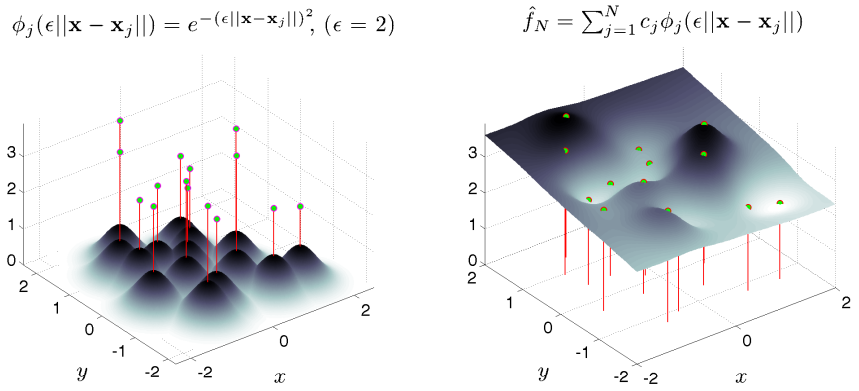


Figure 2.3: RBF interpolation using 15 translates of the Gaussian RBF with $\epsilon = 2$. One RBF is centered at each node in the domain. Linear combinations of these produce an interpolant over the domain passing through known function values.

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 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).

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

Three types of collocation occur throughout the RBF literature: Kansa's unsymmetric collocation method [92, 93], Fasshauer's symmetric collocation method [44], and the Direct collocation method [48].

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.

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

2.1.1 Global RBF Methods

Kansa's method [92, 93] (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 that 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 [44, 82] 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 [103].

The second alternative for RBF collocation, is based on Hermite scattered data interpolation (see [170]). The so-called *Fasshauer* or *Symmetric Collocation* method ([44]) first 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 [103, 142]. The resulting block structure of the linear system is symmetric and guaranteed to be non-singular [44]. 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 [46]. As [82] 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 [48] and satisfies the differential operator on the interior and the boundary. Larsson and Fornberg [103] 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 [103], no related work was found that applied, or investigated, this method further.

Both Kansa's method and Fasshauer's methods were shown in [45] 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 derivatives at the collocation points only [47]. 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 [46]. 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 compute 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 a LU-decomposition, with subsequent forward- and back-solves—could be reduced. While not in mainstream use by the RBF community, [111] correctly points out that iterative solvers with $O(N^2)$ complexity per time-step could be employed.

Hon et al. [81] employed Kansa's method to solve the shallow water equations for Typhoon simulation. In [53], Flyer and Wright employed RBF-PS (Kansa method) for the solution of shallow

	RBF Interpolation Type		
Method	Global (Dense)	Compact (Sparse Global)	Local
Kansa's Method	[45, 49, 52, 53, 62, 63, 65, 81, 82, 92, 93, 103, 112, 133, 168]	[62, 157]	[39, 100, 153, 155]
Fasshauer's Method	[44, 45, 103]	[108]	[141, 142, 143]
Direct Collocation	[48, 103]		
RBF-FD	N/A	N/A	[26, 27, 36, 50, 51, 57, 59, 136, 137, 166, 167, 169]

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.

water equations on a sphere. Their results show that RBFs allow for longer time steps with spectral accuracy. The survey [49] by Flyer and Fornberg showcases RBF-PS (Kansa's method) compared to some of the 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 [52], shallow water equations [53], and 3-D mantle convection [168], RBF-PS consistently required fewer time steps, and a fraction of the nodes for similar accuracy [49].

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:

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

where a cut-off radius, r, is defined past which the RBF (in this case $\varphi(r)$) has no influence on the interpolant. Note that the radius in Equation 2.1 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 [46]. The

actual complexity estimate of the CSRBF method depends on the sparsity of the problem and on the ordering of the assembled system. Assuming $n \ll N$ where n represents the number of nodes within the support region, [175] lists the complexity as dominated by $O(N)$ for properly structured systems, and the investigation in [111] found $O(N^{1.5})$ consistent—in the range of N tested—with the estimate provided by their choice of general sparse solver package (a sparse LU decomposition). A multi-level CSRBF method, introduced by Fasshauer [46], 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)* [157] and *radial point interpolation collocation method (RPICM)* [108], respectively. A more thorough survey of CSRBF history can be found in [46, 86].

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 that do not induce global deformation (see e.g., [30, 106, 171]).

2.1.3 Local RBF Methods

Around 2005, Šarler and Vertnik [153, 155] 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 [155], the authors consider 2-D diffusion problems. Divo and Kassab [39] employ the method for Poisson-like PDEs including fluid flow and heat transfer. Kosec and Šarler [100] apply the same technique to solve coupled heat transfer and fluid flow problems.

In similar fashion, Stevens et al. [143] introduced a local version of Fasshauer's method called *local Hermitian interpolation*. The authors have applied their method to 3-D soil problems based on transient Richards' equations [141, 142, 143].

2.2 Comparison of RBF Methods

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

Following [112], consider a PDE expressed in terms of a (linear) differential operator, $\mathcal{L}$:

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

where Ω is the interior of the physical domain, $\partial\Omega$ 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., [169]); 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$\{P_l(x)\}_{l=1}^M$:

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

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) [86]. 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 [46, 86], 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:

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

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 [46, 86]) that some choices of RBFs (e.g. multiquadrics and thin-plate splines [82]) 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 [46], 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., $x^0, x^1, \cdots, x^m$). In $\mathbb{R}^d$, $M = \binom{m+d}{d}$ [86], giving

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

where the second summation (referred to as *interpolation conditions* [86]) 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:

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

or

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

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

$$\underbrace{\begin{bmatrix} \phi & P \\ P^T & 0 \end{bmatrix}}_{A} \begin{pmatrix} c \\ d \end{pmatrix} = \begin{pmatrix} f \\ 0 \end{pmatrix} \tag{2.5}$$

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:

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

2.2.2 Reconstructing Solutions for PDEs

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

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

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., [24, 25]) capable of smoothing/de-noising a surface reconstructed from point clouds.

For all the methods that follow a linear system is generated:

$$A_\mathcal{L} \begin{pmatrix} c \\ d \end{pmatrix} = \begin{pmatrix} f \\ 0 \end{pmatrix}$$

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

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:

$$\begin{aligned} u(x) &\approx \begin{bmatrix} \phi & P \end{bmatrix} \begin{pmatrix} c \\ d \end{pmatrix} \Bigg|_{x=x_i} \\ &\approx BA_\mathcal{L}^{-1} \begin{pmatrix} f \\ 0 \end{pmatrix} \end{aligned} \tag{2.7}$$

Likewise, to obtain differential quantities,

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

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* [92, 93], 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 [112]; assuming that $\mathcal{L}$ is a linear operator,

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

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

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

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

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

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

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 $P^T = \begin{bmatrix} P_{\mathcal{L}}^T & P_{\mathcal{B}}^T \end{bmatrix}$ 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 [44]; however, singularities are rare in practice [103].

Fasshauer's Method. *Fasshauer's method* [44] 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:

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

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

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

which becomes the following:

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

Note that $\phi_{\mathcal{LL}}$ 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 [48, 103], 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:

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

Reformulating as a linear system provides:

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

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 [45]. 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., $\{x_i\} = \{x_c\}$) [45, 46]. Then, extending Equation 2.7, RBF-PS gives:

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

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:

$$D_{\mathcal{L}} \;=\; A_{\mathcal{L}}^{-T} B^T$$

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[46, 168].

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 [153, 155] for Kansa's method, [141, 142, 143] 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., [39, 142].

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:

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

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* [131]. Recall that as $\epsilon \to 0$, 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 [57], 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 [63] presented the *Contour–Padé* algorithm, which allows for numerically stable computation of highly accurate interpolants with nearly flat RBFs (i.e., $\epsilon \to 0$). Larsson and Fornberg [103] 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 2-D and slightly more in 3-D [60].

The *RBF-QR* method, was later introduced by Fornberg and Piret [61] in context of a sphere to let $\epsilon \to 0$ for a few thousand nodes. It was later extended to general 1-D, 2-D and 3-D problems in [57]. The RBF-QR method uses a truncated expansion of RBFs in terms of spherical harmonics or Chebyshev polynomials and leverages QR factorization to create 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 [57] show that double precision RBF-QR is between 3x-7x slower than RBF-Direct for the same values of ϵ. Fornberg, Larsson and Flyer [57] successfully implemented the 2-D method in less than 100 lines of MATLAB code and apply RBF-QR to problems with 6000 quasi-unifrom 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. [60] 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. Like RBF-QR, RBF-GA is also effective only for a small number of nodes: a few hundred in 2-D, and at least 500 in 3-D. 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 [60]

Benchmarks provided in [60] rank the stable methods for RBFs from fastest to slowest as: RBF-Direct, RBF-GA, and then RBF-QR. Keeping perspective on run-times, the authors note that RBF-GA is roughly 10x slower than RBF-Direct and RBF-QR is another 2x-4x slower than RBF-GA [60].

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 [150], but it was the simultaneous, yet independent, efforts in [136], [151], [167] and [26] 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:

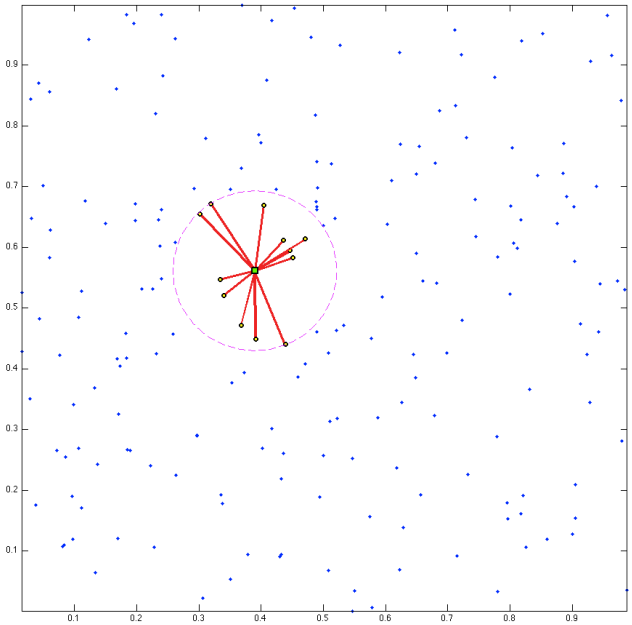
$$\mathcal{L}u(x) \mid_{x=x_c} \approx \sum_{j=1}^{n} c_j u(x_j) \tag{3.1}$$

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 ($||x - x_c||_2$).

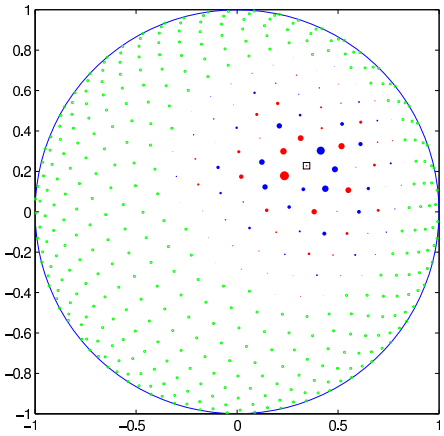
Figure 3.1 provides two examples of RBF-FD stencils. A single stencil of size $n = 13$ is depicted in Figure 3.1a within a domain of random points. The center, x_c, is represented by a green square, with 12 neighbors connected via red edges. The purple circle—the minimum covering circle for the stencil—illustrates 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 [59], weights for a 1-D classical-FD stencil can be obtained by solving a Vandermode system,

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



(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

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 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 $\{1, x, x^2, \cdots, x^{n-1}\}$ 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 [51, 54, 59, 169] show that better accuracy is achieved when the interpolant can exactly reproduce a constant, p_0, such that

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

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:

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

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

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 [46] 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 [167], RBF-QR [36, 57], or RBF-GA [60] 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., [52, 59]).

RBF-FD have been successfully employed for a variety of problems including Hamilton-Jacobi equations [26], convection-diffusion problems [27, 142], incompressible Navier-Stokes equations [28, 136], transport on the sphere [59], and the shallow water equations [51]. Shu et al. [137] compared

the RBF-FD method to Least Squares FD (LSFD) in context of 2-D 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 [166]. Chandhini and Sanyasiraju [27] studied it in context of 1-D and 2-D, 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 [26], and 2-D linear and non-linear PDEs including Navier-Stokes equations [136] have all been considered.

This work focuses on RBF-FD in the context of distributed environments, and on GPUs. Our implementation is verified Chapter 8 with a study of transport on the sphere following [59].

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

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

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") [6].

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, not all nodes in the domain must have an associated stencil, 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 differential 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 $\{x_j\}_{j=1}^n$:

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

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:

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

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

$$u' = D_{\mathcal{L}}u \tag{3.5}$$

whereas implicit solves for unknown u:

$$D_{\mathcal{L}}u = f \tag{3.6}$$

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.

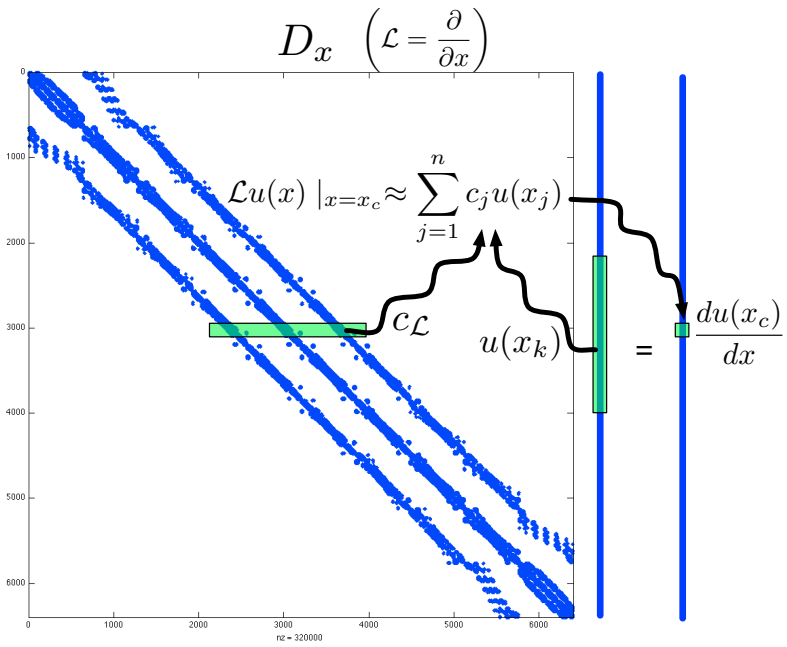


Figure 3.2: Differentiation matrix D_x is applied explicitly to calculate derivative approximations, $\frac{d}{dx}u(x)$.

Differentiation matrices are assembled at a cost of $O(n^3 N)$ 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:

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

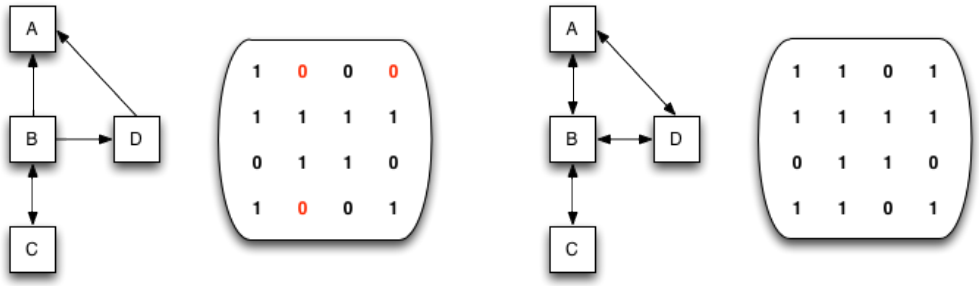
where either a single DM is composed by adding two lower order DMs, or the lower order DMs are directly multiplied against the vector u. Another option applies even lower order operators:

$$\nabla^2 u \approx D_x D_x u + D_y D_y u \ . \tag{3.7}$$

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 resident in memory. This process is sufficient assuming all DMs fit adequately. 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 capitalize on the 30% savings. Beyond 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 only decreases as N grows. 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 when dealing with 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.



(a) Directed (Unsymmetric)

(b) Undirected (Symmetric)

Figure 3.3: Simple adjacency graphs and their corresponding matrices. Edges connecting nodes of RBF-FD stencils produce a directed adjacency matrix.

3.2.1 Adjacency Matrix

In Chapters 4 and 6 the concept of an *adjacency matrix* is required. Adjacency matrices have the same structure as a DM, except all non-zeros elements are masked by the value 1. Non-zero elements indicate node connectivity an adjacency graph. The *degree* of a node is measured as the number of off-diagonal non-zeros in the adjacency matrix. Equivalently, the degree is the number of connections to neighboring nodes in the adjacency graph. For example, Figure 3.3a shows four nodes connected via directional arrows. Arrows originate at a stencil center and terminate at stencil nodes. Both the graph and matrix indicate that B depends on A, C and D, while C depends on B, and D depends on A. A has no dependencies. The degree for each node is given as zero for A, three for B, one for C and one for D. The graph in Figure 3.3a is directed with an associated unsymmetric adjacency matrix. Figure 3.3b demonstrates the case where edges are bijective resulting in an undirected graph and symmetric adjacency matrix.

Some operations like matrix decomposition and matrix reordering are easier applied to symmetric adjacency matrices. RBF-FD can benefit from some of those same operations by symmetrizing the adjacency matrix A with $A + A^T$. Note that all non-zeros in $A + A^T$ continue to be masked by 1s.

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 derviatives:

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

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

3.3.2 Cartesian Gradient ($\boldsymbol{\nabla}$)

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

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

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

3.3.3 Cartesian Laplacian (∇^2)

Fasshauer [46] provides the general form of ∇^2 in 2-D as:

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

which depends on Equation 3.8 and 3.9. For Gaussian RBFs in particular we have the following operators:

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

- 3-D:

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

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:

$$\nabla^2 \equiv \Delta_S.$$

Although this originated in the spherical coordinate system, [168] gives the Laplace-Beltrami operator as

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

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 ($P_x \cdot \boldsymbol{\nabla}$) on the Sphere

Following [51, 53], the gradient operator can be constrained to the sphere with this projection matrix:

$$P = I - \mathbf{x}\mathbf{x}^T = \begin{pmatrix} (1 - x_1^2) & -x_1x_2 & -x_1x_3 \\ -x_1x_2 & (1 - x_2^2) & -x_2x_3 \\ -x_1x_3 & -x_2x_3 & (1 - x_3^2) \end{pmatrix} = \begin{pmatrix} P_{x_1} \\ P_{x_2} \\ P_{x_3} \end{pmatrix} \tag{3.10}$$

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 [53]. First, let $\mathbf{x} = \left(x_1, x_2, x_3\right)$ be the stencil center Cartesian coordinates, and $\mathbf{x}_k = \left(x_{1,k}, x_{2,k}, x_{3,k}\right)$ 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 $P_x \cdot \boldsymbol{\nabla}$ 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 [59] and was included in our previous investigations in [21]. When explicitly solving hyperbolic equations, differentiation matrices encode convective operators of the form

$$D = \alpha\frac{\partial}{\partial\lambda} + \beta\frac{\partial}{\partial\theta} \tag{3.11}$$

The convective operator, discretized through RBF-FD, has eigenvalues in the right half-plane causing the method to be unstable [51, 59]. Stabilization of the RBF-FD method is achieved through the application of a hyperviscosity filter to Equation (3.11) [59]. By using Gaussian RBFs, $\phi(r) = e^{-(\epsilon r)^2}$, the hyperviscosity (a high order Laplacian operator) simplifies to

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

where k is the order of the Laplacian and $p_k(r)$ are multiples of generalized Laguerre polynomials that are generated recursively ([59]):

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

where d is the dimension. The application of hyperviscosity in Chapter 8, 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

$$\frac{\partial u}{\partial t} = -\mathcal{L}u + Hu, \tag{3.12}$$

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 $\mathcal{L}$). With the correct sign and careful choice of scaling, hyperviscosity is able to shift all eigenvalues 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 [51], 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 $N \to \infty$ [51]. 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 [51, 59].

Tuned parameters for hyperviscosity are provided in Chapter 8.

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 the discussion, consult Algorithm 3.1, which is split into two phases: Preprocessing and Application. The complexity for each phase can vary based on the algorithms utilized in each task.

The Preprocessing Phase encompasses tasks such as grid setup/generation, stencil generation and stencil weight calculations. As output this 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.

Algorithm 3.1 A High-Level View of RBF-FD

Preprocessing Phase:

$\{x\}_{j=1}^{N}$ = 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 Phase:

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

The algorithm loads/generates 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 local DMs. The assumption in this work is that grids do not

evolve in time, so DMs remain constant for the duration of phase 2 (Application). In the event of moving nodes, weight calculation and DM assembly would move into the loop in the second phase.

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 UpdateSolution step.

Generally the performance of RBF-FD hinges on the cost of SolvePDE. This is especially true for time-dependent PDEs where the CFL condition relates the number of required time-steps to the spacing between nodes. As the problem size grows, smaller time-steps must be taken in order to maintain stability. 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, coarser grid, 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 not only impact accuracy of the method, but stability as well. The only restriction from the method is that coincident nodes should not be connected. 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.

This work verifies and benchmarks RBF-FD on the following grids, which are available for public download (see [18]):

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

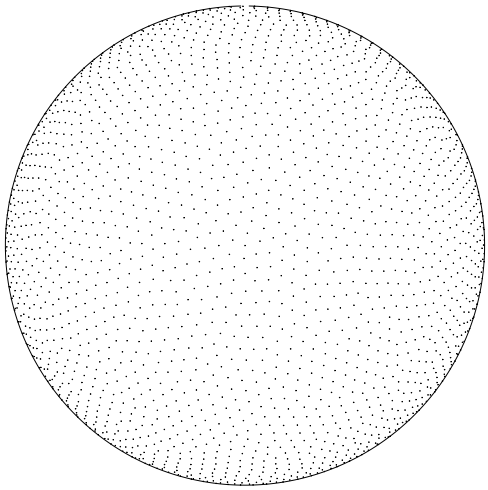


Figure 3.4: Quasi-regular nodes with $N = 4096$ maximum determinant (MD) node sets on the unit sphere.

to generate. Additionally, refinements—for scaling benchmarks and convergence tests—are direct subsets.

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 8 applies RBF-FD to solve PDEs on the unit sphere. For consistency with respect to related investigations (e.g., [56, 59, 61]), the Maximum Determinant (MD) node sets [139, 165] are chosen.

MD node sets were originally utilized by the RBF community due to their success in spherical harmonics interpolation [61]. 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) [165]. 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 $\epsilon \to 0$ [61].

The MD node files are available for download on the authors' web site [164], and range in size from $N = 4$ up to N=27,556 nodes on the sphere. Figure 3.4 plots the $N = 4096$ node set to illustrate the irregularity in distribution. Node sets greater than N=27,556 are not available.

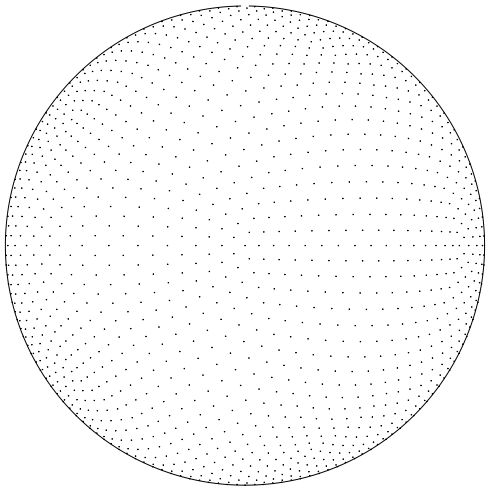


Figure 3.5: $N = 2562$ icosahedral nodes on the unit sphere.

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 3.5 shows Icosahedral nodes on the sphere. Icosahedral grids are nearly homogenous and isotropic, and have been in use since the 1960s [124]. 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). 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 [42, 163].

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 nearly coincide with the projected mass centroids, and the resulting distribution is a SCVT. SCVTs come with a sense

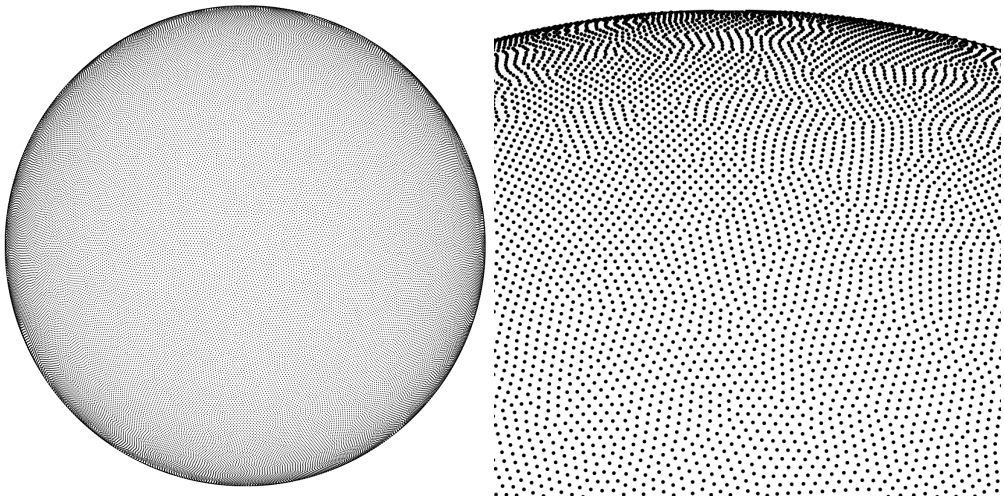


Figure 3.6: (Left) N=100,000 Spherical Centroidal Voronoi Tessellation nodes. (Right) Close-up of the same N=100,000 nodes to illustrate the irregularities in the grid.

of "optimality" in node locations due to energy minimizing properties (see [42]). In most large-scale applications, the iteration in SCVT generation is a probabilistic Lloyd's algorithm, with integrals computed through random sampling [42, 163]. 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 3.6 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 courtesy of Geoff Womeldorff and the SCVT library in [163].

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 against ill-conditioning to 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).

Unfortunately, no foolproof method exists for RBF-Direct—the "optimal" (in most cases this reads: "most acceptable") value of ϵ depends on the node distribution and varies by application.

Parameter sweeps to characterize the behavior of ϵ on large problems can be prohibitively expensive in terms of time and/or money. Ultimately, stable methods for obtaining weights (see § 2.3) are really necessary for general large-scale applications, even if such methods are 10x–40x slower than RBF-Direct [60].

This work utilizes a moderately reliable method, proposed in [51], to choose ϵ as a function of the grid resolution, N. The method performs a parameter sweep in ϵ and N to generate a set of mean condition numbers, $\bar{\kappa}_A$, defined as:

$$\bar{\kappa}_A = \frac{1}{N} \sum_{i=1}^{N} (\kappa_A)_i$$

where $(\kappa_A)_i$ is the condition number for the matrix, A_i, assembled by Equation 3.3 for the i-th stencil in the domain. MATLAB's COND command computes the 2-norm condition number defined as:

$$(\kappa_A)_i = ||A_i^{-1}||_2 \cdot ||A_i||_2.$$

The tiles of Figure 3.7 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 $\sqrt{N} = \{40, ..., 100\}$, in steps of 10, the code makes a sweep through $\epsilon = \{1, ..., 10\}$ in steps of 0.5 and assembles N RBF-FD interpolation matrices. The COND command evaluates $(\kappa_A)_i$ for each matrix. At the end of the sweep, the 2-D array of mean condition numbers are passed through the MATLAB CONTOUR command to identify and trace the line segments.

The resulting values of $\bar{\kappa}_A$ produce remarkably linear contours, with slopes that fan out from the ϵ axis. Note that the $(\kappa_A)_i$ depend on the stencil size, n, as evidenced by shifting contour fans for $n = 20, 40, 60, 80$, and 100. For large n we observe that the simultaneous increase in slope and decreasing separation between contours leaves little "wiggle" room in guessing ϵ under RBF-Direct.

As N grows larger, we assume that the condition number continues linearly. The cost of computing $\bar{\kappa}_A$ balloons as data points are added to the contour plot so values of N are not exhaustively tested. Also, be advised that the node distributions impact conditioning, so switching node distributions (e.g., from MD-nodes to a regular grid) would require constructing a new set of contours.

Regression slope and intercept parameters (c_1 and c_2) are superimposed in Figure 3.7 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 [51] find condition numbers on the order 10^{10} to 10^{12} to function well for competitive accuracy compared to other methods in the literature; the method begins to degrade on large grids (i.e., $N > 10^5$). Similar observations are made in Chapter 8.

Stable weight algorithms like Contour-Padé [167], RBF-QR [36, 57, 61], and RBF-GA [60] can allow RBF-FD to scale beyond the limits of RBF-Direct and will be included in a future investigation.

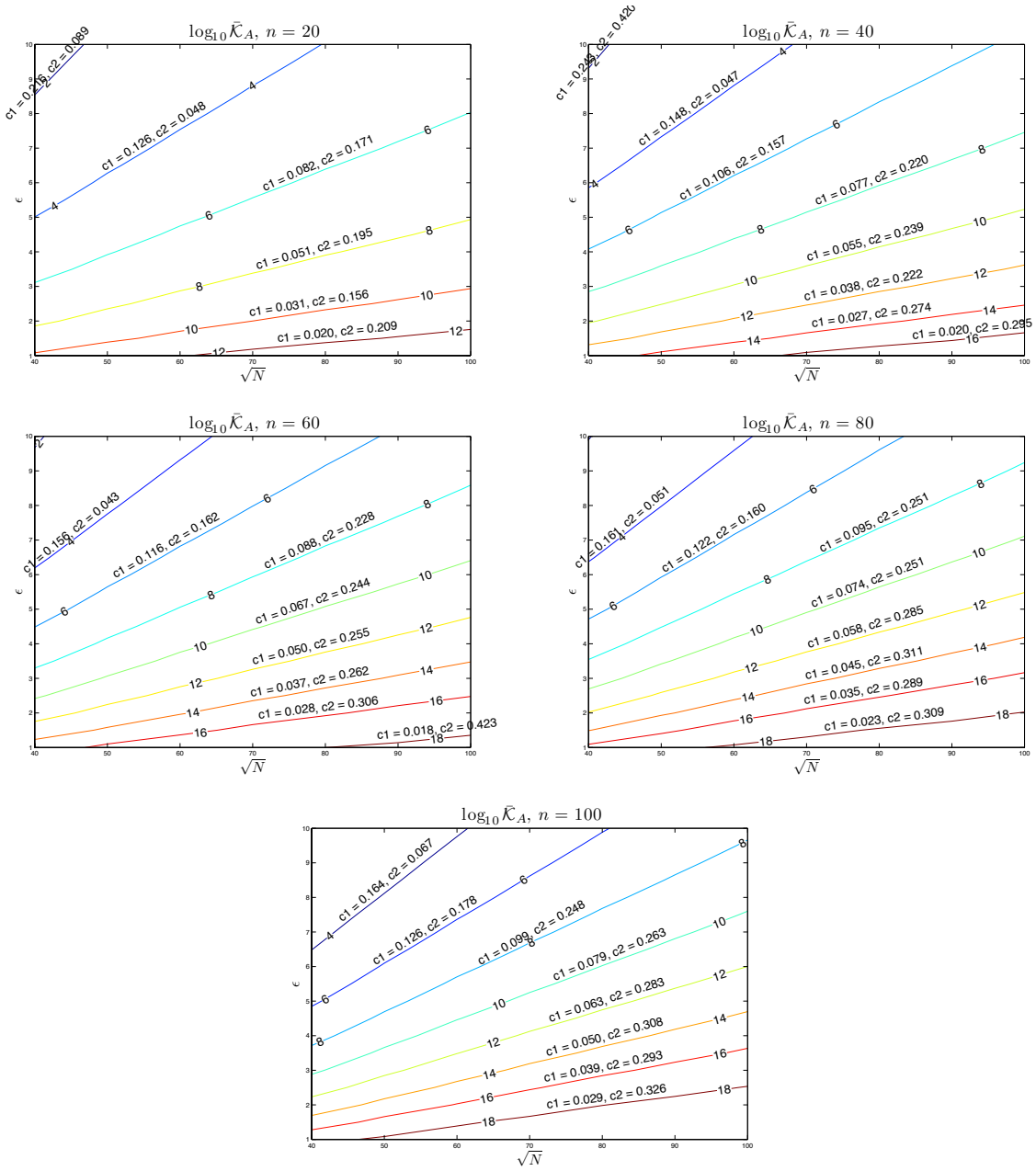


Figure 3.7: Contours for ϵ as a function of $\sqrt{N}$ for stencil sizes $n = 20$, 40, 60, 80 and 100 on the unit sphere. Contours assume near uniform distribution of nodes (e.g., maximum determinant (MD) nodes). Parameters superimposed above each contour provide coefficients for function $\epsilon(\sqrt{N}) = c_1\sqrt{N} - c_2$.

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 the 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 [147] (a.k.a. ℓ-nearest neighbor search [160]). Here "nearest" is defined with the Euclidean distance metric, although it is possible to generalize to other metrics (see e.g., [4]).

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 [160]) 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 is harder to solve. Consider, for example, the scenario in Figure 4.1. Two ball queries around a blue 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.

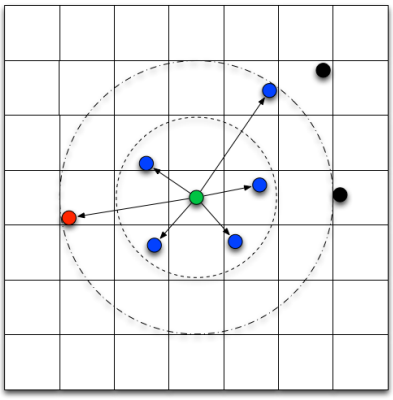


Figure 4.1: A stencil center in blue finds neighboring stencil nodes in blue. Two ball queries are shown as dashed and dash-dot circles to demonstrate the added difficulty of finding the right query radius to obtain the k-nearest neighbors.

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., [159, 160]), and particle methods like the Fast Multipole Method (e.g., [77, 172]) or Smoothed Particle Hydrodynamics (e.g., [102]). Examples of fast algorithms employed in these fields include the fixed-grid method [102, 160], k-D Trees [160], Range Trees [159, 160], and 2^d-Trees (i.e., Quad- and Octrees) [77, 172]. 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., [46, 51, 59]).

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 [73, 90, 102]. Samet[129] 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 ([147]). Benchmarks in § 4.3 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 partition 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* [138]. 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., [37, 129, 138] 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 [138, 172].

This work considers *Point k-D Trees* [129], 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 [37].

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 [129]. The first cut, $L1$, shown in blue, splits the nodes into two sets on either side of A. The corresponding tree in

Algorithm 4.1 BuildKDTree(P, *depth*)

Input: A set of d-dimensional points P and the current *depth*.
Output: The root of the k-D Tree for P.

if $size(P) = 1$ **then**
    return a new leaf storing P
end if
$L_i := median(coord(P, depth))$
$v_l :=$ BuildKDTree($coord(P, depth) \leq L_i$, $(depth + 1)$ modulo d)
$v_r :=$ BuildKDTree($coord(P, depth) > L_i$, $(depth + 1)$ modulo d)
return A new node $v := \begin{pmatrix} \text{value} := L_i \\ \text{left} := v_l \\ \text{right} := v_r \end{pmatrix}$

the center of Figure 4.2 shows $L1$ 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, $L2$ and $L3$ (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 $L1$ 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 has a complexity of $O(N \log N)$ with $O(N)$ storage [37, 129].

Frequently, the terms *k-D Tree* and *Point k-D Tree* are used synonymously by the RBF community (see e.g., [46, 51, 59]); the same convention is 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 [66], 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 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

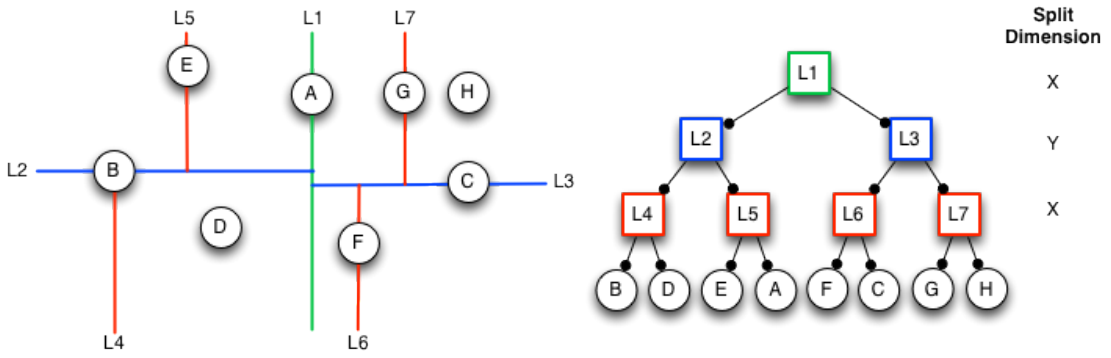


Figure 4.2: An example k-D Tree in 2-Dimensions. Nodes are partitioned with a cyclic dimension splitting rule (i.e., splits occur first in X, then Y, then X, etc.); all splits occur at the median node in each dimension.

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 [66, 148].

The authors of [66] 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 $O(n \log N)$ for one stencil.

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

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

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.
Output: A global priority queue, pq, containing the n-nearest neighbors to X_q sorted by distance from X_q in descending order.
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.

```
if root is leaf then
    Insert {root, dist(X_q, root)} into pq
    if pq.size > n then
        pq.pop                                  ▷ Keep only n-nearest neighbors
    end if
    return
end if

if coord(X_q, depth) <= root.value then
    KNNSearchKDTree(X_q, n, root.left, (depth + 1) % d)
else
    KNNSearchKDTree(X_q, n, root.right, (depth + 1) % d)
end if

if coord(X_q, depth) <= root.value then
    if BoundsOverlapBall(X_q) then
        KNNSearchKDTree(X_q, n, root.right, (depth + 1) % d)
    end if
else
    if BoundsOverlapBall(X_q) then
        KNNSearchKDTree(X_q, n, root.left, (depth + 1) % d)
    end if
end if
return
```

4.2 A Fixed-Grid Algorithm

While a k-D Tree is designed to accommodate high dimensional data, 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 [129, 159, 160] (a.k.a. uniform grid [73, 102]) assume stencils are generated based on their 2-D or 3-D spatial coordinates only. 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:

$$\begin{aligned}(dx, dy, dz) &= \left(\frac{(x_{max} - x_{min})}{h_n}, \frac{(y_{max} - y_{min})}{h_n}, \frac{(z_{max} - z_{min})}{h_n}\right) \\ (c_x, c_y, c_z) &= \left(\left\lfloor \frac{(p_x - x_{min})}{dx} \right\rfloor, \left\lfloor \frac{(p_y - y_{min})}{dy} \right\rfloor, \left\lfloor \frac{(p_z - z_{min})}{dz} \right\rfloor\right). \end{aligned} \tag{4.1}$$

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 [102]:

1. Subdivide the domain with the overlay grid.
2. For each node, identify the containing cell coordinates (Equation 4.1).

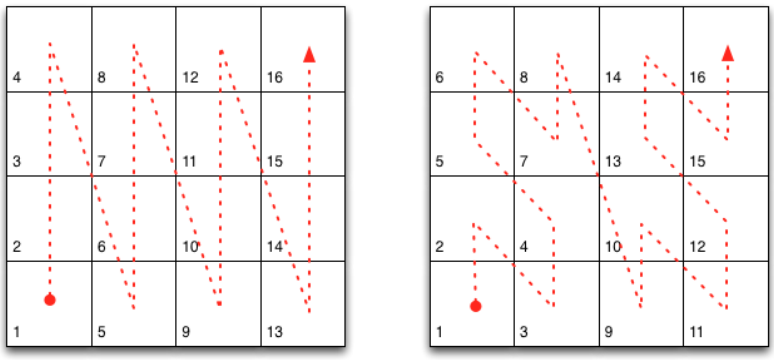


Figure 4.3: Two example space filling curves to linearize the same fixed-grid. Left: Raster-ordering (ijk); Right: Morton-/Z-ordering.

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 [129].

4.2.1 Fixed-grid Construction

The algorithm in this work is inspired by fixed-grid approaches for GPU particle simulations ([73, 90, 102]). 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 [72]. 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 [73, 90, 102] 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 halo 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 [72], but the authors opt for a Z-ordering of cells. As a point of difference in implementations, the authors of [72, 73, 102] leverage a fast radix sort algorithm to order nodes based on hash index, while [90] utilizes a slower bitonic sort algorithm. The fixed-grid in [159, 160] 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 ([159, 160]) is constructed as part of a single-pass bucket sort. Finally, in stark contrast to [73, 90, 102, 159, 160], querying neighbors is not restricted to a fixed radius, or number of cell halos. To satisfy the k-NN query, this implementation iteratively increases the query radius to include a new halo 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 presents the fixed-grid build process. The routine starts by allocating an array of empty buckets, Q, which 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 [129] 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

Algorithm 4.3 BuildFixedGrid(P, h_n)

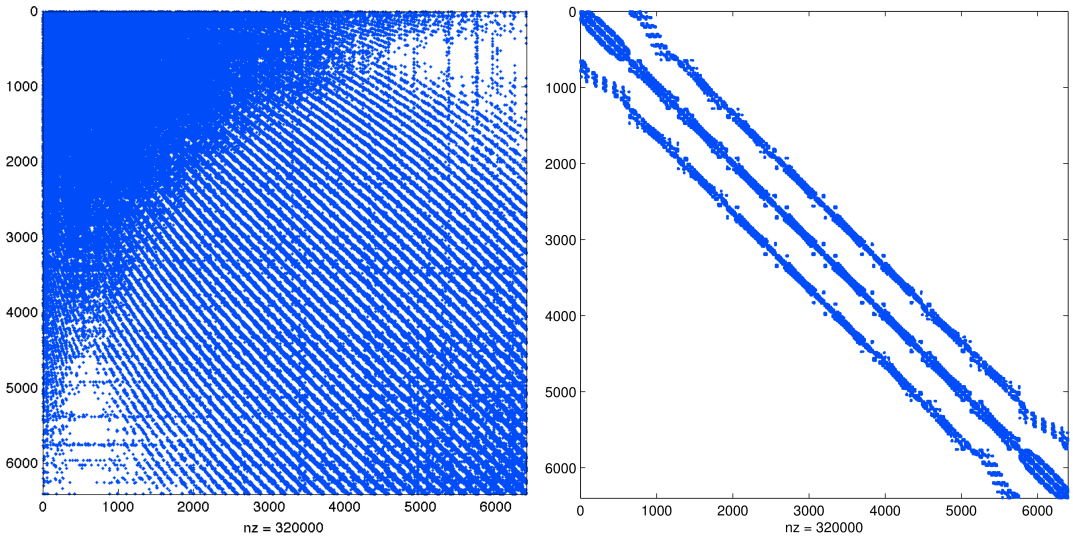
Input: A set points P, and the fixed-grid resolution, h_n.
Output: The reordered points in P, and corresponding cell buckets Q.

Create Q: an $(h_n)^d$ array of empty buckets.
for point p_i in P **do**
    $c :=$ CellCoords(p_i)
    $ind :=$ SpatialHash(c)
    Append index i onto $Q[ind]$
end for
for $j = 0, 1, ..., (h_n)^d$ **do**
    if $Q[j]$ is not empty **then**
        Append the set $P[Q[j]]$ onto $\hat{P}$
        Overwrite the set $Q[j]$ with new indices of $\hat{P}$
    end if
end for
$P := \hat{P}$
return

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 (blue dots). 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, but results from a fixed-grid sorted by raster-ordering with $h_n = 10$.

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.



(a) k-D Tree (b) Fixed-Grid

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).

4.2.2 Fixed-Grid Neighbor Query

Querying the k-nearest neighbors for a 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 halo 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. Also, for certain fixed-grid resolutions, a single halo may not satisfy the stencil size requirements, so the algorithm iterates outward. 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 $O(\log h_n)$ 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

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

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.
Output: The n-nearest neighbors list pq.

$halo := 1$
$c :=$ CellCoords(X_q)
$ind :=$ SpatialHash($cells$)
Append $P[Q[ind]]$ onto pq
while pq.size $< n$ OR $halo < 2$ **do**
    $cells :=$ NeighboringCellCoords($c, halo$)
    $inds :=$ SpatialHash($cells$)
    for each q in $Q[inds]$ **do**
        if q is not empty **then**
            Append node list $P[q]$ onto pq
        end if
    end for
    increment $halo$
end while
$dists :=$ ComputeDistances(pq)
Sort pq by $dists$
return the first n nodes in pq

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 blue 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 alternate black node is selected even though it is more distant. This happens because the more distant black node occupies the second halo of cells around the stencil center, whereas the true near neighbor is in the third. Algorithm 4.4 is able to truncate the search in the second iteration 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. Additionally, 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 [147] and now maintained as an independent Google Code project [148]. 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 is appealing for rapid-prototyping with MATLAB, and then porting applications to C++.

The pre-2012 implementation of *kd*-Tree Matlab followed the $O(N \log N)$ expected complexity for neighbor queries, but cost significantly more to build the tree. While build times for small and medium sized grids (i.e., less than $N = 50000$ nodes) were small enough to be inconspicuous, the implementation scaled as $O(N^2)$ making it prohibitively expensive for large problems.

The high cost ultimately led to work on a fixed-grid method to test the concept of neighbor queries with low build costs. The original prototype developed in pure MATLAB ([15]), outperformed the "efficient" MEX-compiled k-D Tree for problem sizes $N > 20000$, and lead to the development of a C++ implementation tested here. In the 2012 release of *kd*-Tree Matlab ([148]), 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 as presented above.

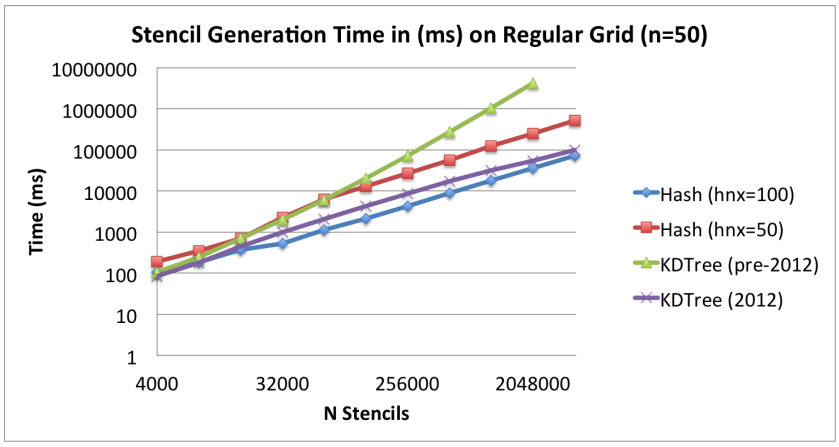
All benchmarks in this section were performed on the Itasca HPC cluster at the Minnesota Supercomputing Institute. Itasca is an HPLinux cluster with 1,134HP ProLiant blade servers, each with two-socket, quad-core 2.8 GHz Intel Xeon processors sharing at least 24GB of RAM [3]. Both k-D Tree and fixed-grid implementations are compiled with the Intel compiler toolchain (v13), and 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 in Figure **??** as evidence of 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 stencil generation with a fixed-grid—achieved over the more efficient release of *kd*-Tree Matlab. These results cover the total time to build the data-structure and query all stencils. The test grid is generated in raster-order, as is the fixed-grid.

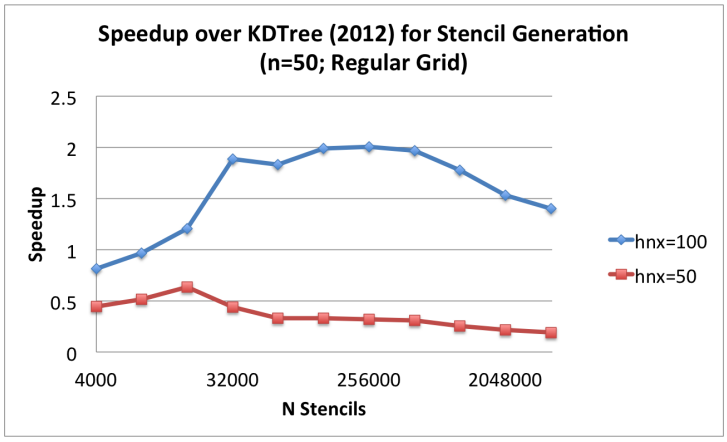
Prior to the 2012 improvements, the C++ implementation was over 130x faster for four million nodes. With the newer and more reasonable k-D Tree benchmarks, the fixed-grid method is only 2x faster in the best case shown here. Although 2x is not as impressive, it is a notable improvement. One issue in 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.

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 ([18]). 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 $h_n = 100$.

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,



(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.

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 [72, 73, 102].

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 it does eke out a small victory with the right choice of h_n. The reality is that stencil generation only

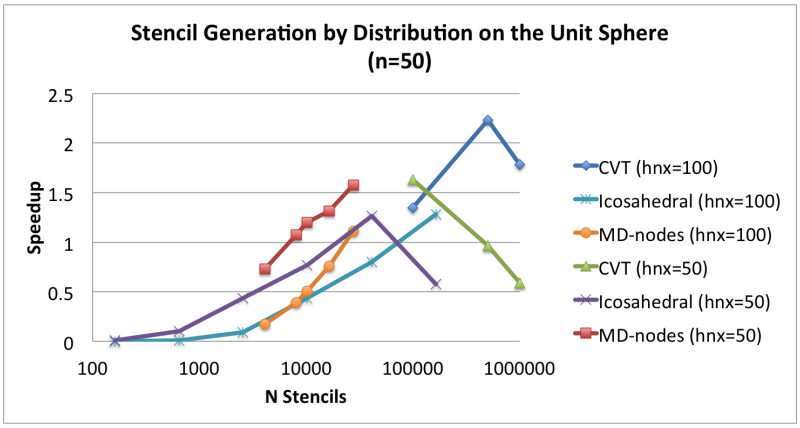


Figure 4.6: Fixed-grid speedup versus k-D Tree with stencil size $n = 50$.

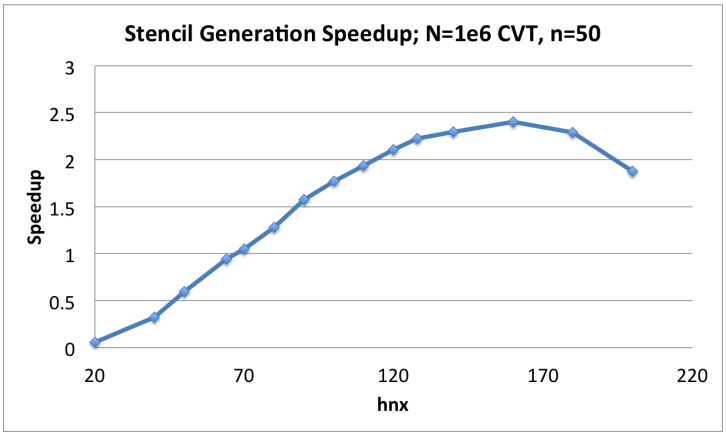
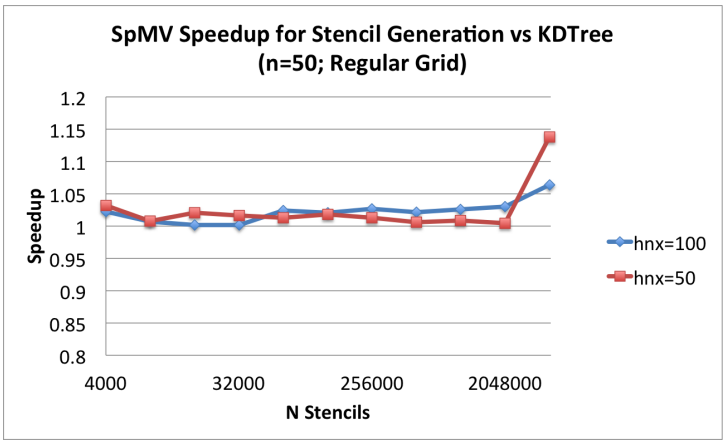


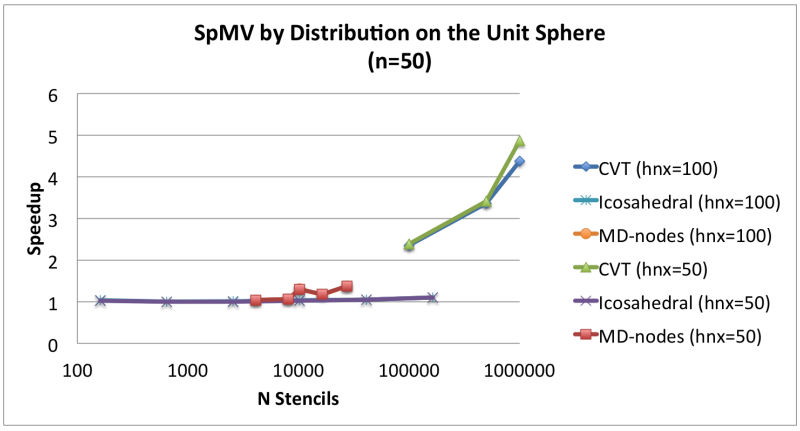
Figure 4.7: Fixed-grid speedup versus k-D Tree on a one-million node CVT (unit sphere), with stencil size $n = 50$.

occurs once, and the difference between k-D Tree and fixed-grid is easily amortized by iterations during the bulk of the RBF-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.

Regular grid and icosahedral sphere test cases in Figure 4.8 see no more than 5% improvement, which is unsurprising as their nodes are previously sorted. 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. Figure 4.9 demonstrates the impact on SpMV as a function of h_n for the $N = 10^6$ CVT



(a) 3-D Regular Grid



(b) Unit Sphere

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

test case. The results show that even a coarse fixed-grid ($h_n \geq 40$) is capable of achieving the spatial locality of data that results in 5x faster SpMV. The combined results of 2x faster stencils and 5x faster SpMV make a case for the use of a fixed-grid query as a general safety net to precondition RBF-FD when properties of the input grid are unknown.

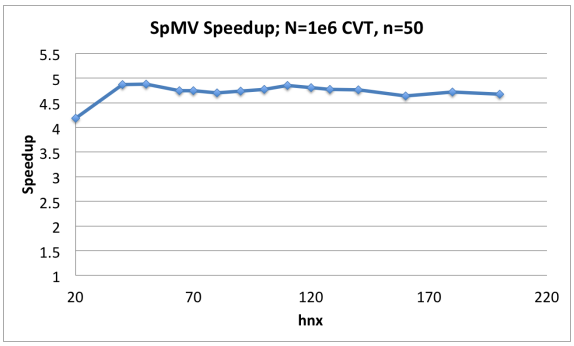


Figure 4.9: Impact on SpMV performance for $N = 10^6$ CVT of the unit sphere due to choice of h_n.

4.4 Alternative Orderings

As previously mentioned, the choice of space filling curve influences the sparsity pattern of differentiation matrices, which in turn impacts cache effects. The use of Z-ordering (Figure 4.3) for potential benefits on memory access is a common recommendation in fixed-grid methods (see e.g., the suggestions by [73, 90, 102] and implementations in [72, 109]). Under raster-ordering, cells are traversed with priority on one dimension. The resulting layout only has adjacent cells along that one dimension consecutive in memory. Other approaches, like Z-ordering, traverse cells by alternating dimensions/directions and increase locality of data in multiple dimensions.

This section investigates the potential benefits of Z-ordering and various other space filling curves implemented as part of the original MATLAB fixed-grid prototype ([15]). Examples of the tested curves are presented in Figure 4.10. Here each node correlates to a fixed-grid cell, and curves start at coordinates $(0, 0)$.

Of the six cases shown in Figure 4.10, five of them (Raster, X, Z, U, 4-node Z) are directly produced via *bit-interleaving* hash functions. Those same tiles are named based on the hierarchical pattern of connected nodes and groups of nodes.

The sixth tile in Figure 4.10 shows a Reverse Cuthill-McKee (RCM) ordering. The RCM method is distinct from the other orderings in that it is a *bandwidth reduction* algorithm that focuses on condensing the sparsity pattern of differentiation matrices around the diagonal rather than spatially sorting nodes. Figure 4.10 shows a special case that results when all nodes have stencils of size $n = 3$. It nicely illustrates the RCM as a breadth-first traversal through the grid. More discussion on RCM occurs later in this section.

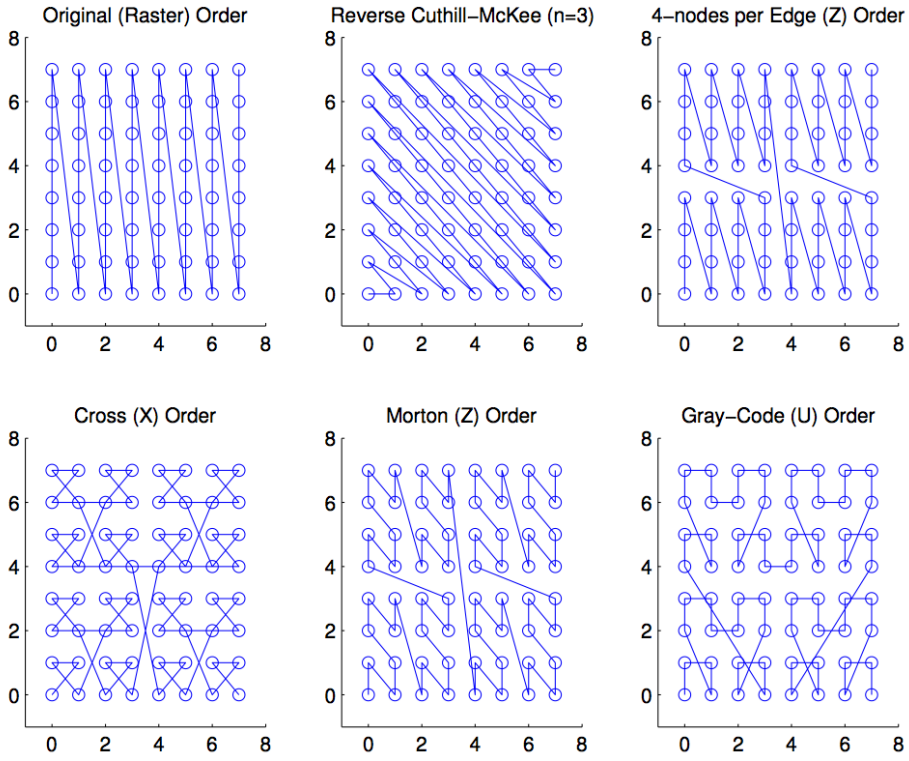


Figure 4.10: Example space filling curves used to reorder cells/nodes. The Raster, Z-, X-, U- and 4-nodes per Edge Z-orderings are space filling curves applied to reorder cells of the fixed-grid stencil queries. Reverse Cuthill-McKee (RCM) operates on output stencils and their associated adjacency graph. The RCM shown here is a special case with only 3 neighbors per node.

4.4.1 Bit-Interleaved Orderings

Bit-interleaving zips bits from two or more integers into one. For example, given two integers, X, and Y the bits are interleaved as:

$$\begin{aligned} X &= \{x_3 x_2 x_1 x_0\} \\ Y &= \{y_3 y_2 y_1 y_0\}, \\ interleave(X, Y) &= \{x_3 y_3 x_2 y_2 x_1 y_1 x_0 y_0\}. \end{aligned} \tag{4.2}$$

Including a third coordinate, Z, results in the following:

$$interleave(X, Y, Z) = \{x_3 y_3 z_3 x_2 y_2 z_3 x_1 y_1 z_3 x_0 y_0 z_3\}.$$

Table 4.1: 2-D Integer Dilation Masks for 32-Bit Integers

Level (i)	Mask (M_i)		S_i
0	$00000000000000001111111111111111_2$	0x0000$ffff$	32
1	$11111111000000000000000011111111_2$	0xff0000ff	16
2	$00001111000000001111000000001111_2$	0x0f00f00f	8
3	$11000011000011000011000011000011_2$	0xc30c30c3	4
4	$01001001001001001001001001001001_2$	0x49249249	2

Table 4.2: 3-D Integer Dilation Masks for 32-Bit Integers

Level (i)	Mask (M_i)		Shift (S_i)
0	$00000000000000001111111111111111_2$	0x0000$ffff$	48
1	$00000000000000000000000011111111_2$	0x000000ff	24
2	$00000000000011110000000000001111_2$	0x000f000f	12
3	$00000011000000110000001100000011_2$	0x03030303	6
4	$00010001000100010001000100010001_2$	0x11111111	3

Interleaving depends on *integer dilation* to modify an input integer value, so the output value has the original input bits spaced by new zeros. On a d-dimensional domain, bit-interleaving requires $d-1$ zeros between each bit. For example, dilating 1111_2 expands the four bit integer to 01010101_2 in 2-D, and 001001001001_2 in 3-D.

The process to dilate integers applies multiple bit-mask operations to spread the digits appropriately. For any b-bit integer, X_0, the dilation is given by the following iteration:

$$X_{i+1} = ((X_i \ll S_i) \mid X_i) \;\&\; M_i \qquad \text{for } i = 0, ..., \log_2(b) \tag{4.3}$$

where operators $\ll$, $|$, and & indicate left-shift, bit-wise OR, and bit-wise AND operators respectively. In general, a b-bit integer requires $\log_2(b)$ iterations through the dilation kernel. The masks (M_i) and shifts (S_i) necessary to dilate a 32-bit integer are provided in Table 4.1 for 2-D, and Table 4.2 for 3-D. Both binary and hexadecimal representations are provided for M_i. Observe that the shifts and bit-masks applied during each iteration depend on the size of the integer and the dimension (refer to [144] for more details).

It is possible to lose bits in translation when dilating values that require greater than $\lfloor {}^{b}/_{d} \rfloor$ bits. That means, in the case of 32-bits, any input to Equation 4.3 must be representable in 16-bits or less (i.e., $X_0 \leq 65535$) for 2-D, and 10-bits or less (i.e., $X_0 \leq 1023$) for 3-D. Note that these limits then become the maximum resolution possible for h_n in the fixed-grid method.

Table 4.3: Integer Dilation Interleaving Operators

Ordering	2-D	3-D
IJK	$(X * h_n) + Y$	$((X * h_n) + Y) * h_n + Z$
Z	$interleave(X, Y)$	$interleave(X, Z, Y)$
U	$interleave(X, X \oplus Y)$	$interleave(X, Z, Z \oplus Y)$
X	$interleave(X \oplus Y, X)$	$interleave(X \oplus Y, Z, Y)$
4-Node Z *	$(d_x \ll 2) \mid d_y$	$((d_x \ll 4) \mid (d_z \ll 2)) \mid d_y$

With dilation available, the process for 2-D bit-interleaving (Equation 4.2) reduces to:

$$\begin{aligned} (d_x, d_y) &= (dilate(X), dilate(Y)) \\ interleave(X, Y) &= (d_x \ll 1) \mid d_y. \end{aligned} \tag{4.4}$$

And in 3-D:

$$\begin{aligned} (d_x, d_y, d_z) &= (dilate(X), dilate(Y), dilate(Z)) \\ interleave(X, Y, Z) &= ((d_x \ll 2) \mid (d_y \ll 1)) \mid d_z. \end{aligned} \tag{4.5}$$

Equations 4.4 and 4.5 can be used as hash functions to produce a vast number of space filling curves (see [144]).

By default, Equation 4.4 results in the Z-ordering in Figures 4.3 and 4.10. Consider for example:

$$\begin{aligned} (c_x, c_y) &= (5, 3) = (0101_2, 0011_2) \\ (d_x, d_y) &= (00010001_2, 00000101_2) \\ interleave(c_x, c_y) &= (00100111_2) = 39. \end{aligned}$$

The mapped Z-index of 39 can be verified by counting the number of steps from $(0, 0)$ to $(5, 3)$ on the Z-order curve in Figure 4.10.

The X- and U- orders shown in Figure 4.10 also result from Equation 4.4 with only slight modifications to the input. Table 4.3 provides a comparison of orderings and the operators/inputs in 2-D and 3-D that produce them. The U and X orderings depend on a bit-wise XOR operator, $\oplus$, to combine coordinates before interleaving.

The last row of Table 4.3, the 4-node Z-curve, is a special case of bit interleaving. The 4-node Z is similar in theory to a standard Z-order, except that an extra zero per bit is requested when

coordinates are dilated as though the process were operating in the next higher dimension. Then, using the operators provided in Table 4.3 to interleave bits results in the following pattern:

$$interleave_{4node}(X, Y) = \{x_5x_4y_5y_4x_3x_2y_3y_2x_1x_0y_1y_0\}.$$

By reserving two bits for each dimension, the 4-node Z-curve is able to traverse up to four nodes in the Y direction before taking its first step in the X direction.

Often, interleaved cell coordinates near the boundaries of the domain do not produce contiguous hash indices and may balloon to values greater than the number of nodes/cells in the domain. In these situations, missing indices reference coordinates outside the domain that would complete the hierarchical power-of-2 curves before entering the domain again. When sorted and traversed least to greatest, the hashed indices give a proper ordering of the domain, and missing indices are skipped.

4.4.2 Bandwidth Reduction and Reverse Cuthill-McKee

The purpose of reordering nodes/cells is to improve memory access patterns. If condensing non-zeros of a sparse matrix achieves this, then the ideal case arises when n non-zeros occupy the first n diagonals of the matrix (including the main diagonal) for all rows. In terms of stencils on a grid, this ideal case corresponds to all nodes on a 1-D line connected to their left and right nearest neighbors. In higher dimensions, reproducing the ideal structure requires one to neglect spatially nearest neighbors, except in cases of very low resolution or small stencil sizes, and select neighbors by following a space filling curve left and right. Stencils in 2-D or 3-D constructed through k-NN (or k-ANN) queries result in a sparsity pattern with gaps between non-zeros. In the worst case, matrices would have non-zeros in each row spaced as far apart as possible while maintaining the appropriate connectedness of the adjacency graph.

As a measure of how well condensed non-zeros are, the *bandwidth* of a matrix, A, is defined as ([35, 107]):

$$bw(A) = \max\{|i - j| + 1 : A_{i,j} \neq 0\}.$$

The bandwidth simply counts the number of super- and sub-diagonals—i.e., diagonals above and below the main diagonal—that contain at least one non-zero. Incrementing by one accounts for a non-zero main diagonal. The lower the bandwidth, the better.

For any matrix, A, a permutation, P, can be constructed such that

$$PAP^T$$

reorders the matrix and reduces the bandwidth of A. A variety of *bandwidth reduction* algorithms exist with unique approaches for the construction of P (see e.g., [67, 107, 109]). In contrast to the space filling curves discussed above, bandwidth reduction algorithms are applied *after* stencils are generated in order to operate on the differentiation matrix and corresponding adjacency graph.

The effect pre- and post-multiplying A by P is equivalent to reindexing nodes and and updating all stencil connections to the new numbering. In practice, rather than construct a full permutation matrix P, which has exactly one 1 per row and per column, it is common to represent P as a vector where each entry indicates the column location of the 1. In that case the vector elements reference the original node ordering, and the indices of those elements provide the new ordering.

The most popular methods for bandwidth reduction are Cuthill-McKee (CM) algorithms [35, 107]. In particular this work is concerned with the *Reverse Cuthill-McKee* (RCM) variant. The authors of [107] show that, in comparison to regular CM, the RCM algorithm offers better conditioning of PAP^T for reduced fill-in and reduced computation during matrix decompositions. Although fill-in is not a concern in this work, the RCM variant is attained at no extra cost.

At a high level, CM methods start by selecting a node in the adjacency graph with the lowest degree (i.e., fewest non-zeros per row in the matrix). From that node, a breadth-first traversal begins, passing through all connected neighbors in order from least to greatest degree. If nodes have the same degree, as is the situation for RBF-FD with stencil size, n, then ties are broken arbitrarily. The original indices for visited nodes are pushed onto a queue, r. Another queue is maintained to catalogue previously traversed nodes and prevent the process from doubling back on itself. In the event that the adjacency graph has multiple disconnected sub-graphs, the algorithm exhausts the first sub-graph before beginning the next; as always, beginning with the lowest degree node available.

CM methods produce r as the vector representation of P. The RCM variant differs from CM by reversing the order of r so that rows corresponding to the last traversed nodes are at the top of PAP^T.

Most CM and RCM implementations assume the algorithms operate on square, symmetric adjacency matrices. In the case of RBF-FD the adjacency matrices are symmetrized with $A + A^T$.

For rectangular matrices with dimensions $N \times M$ (as are encountered in Chapter 6), RCM is applied only to the $N \times N$ square block with symmetry similarly constructed.

4.4.3 Impact of Orderings

To compare the impact of orderings in Figure 4.10, the MATLAB implementation of fixed-grid ([15]) generates stencils and writes the sorted nodes and stencils to disk. The nodes and stencils are accepted as input for the C++ SpMV benchmark used above to compare the k-D Tree and fixed-grid stencils. Visualization and bandwidth calculation are both managed in MATLAB. Bit interleaving occurs in a MATLAB script that leverages the internal *bitshift*, *bitand*, *bitor*, and *bitxor* routines for bitwise operations. Bitwise operators in MATLAB are restricted to 32-bits, but the maximum number of cells per dimension allowed by 32-bits (≤ 1023 in 3-D) is more than sufficient. The MATLAB internal *symrcm* computes the Reverse Cuthill-McKee reordering. Since the same number of non-zeros exist per row, RCM selects a different starting node based on the input ordering of nodes; this allows for some variance in the bandwidth. For consistency, results shown here apply RCM to the output of a raster-ordered fixed-grid query.

Figure 4.11 illustrates six sparsity patterns resulting from the curves in Figure 4.10. The test case is a regular grid of $N = 18^3$ nodes, each with stencil size $n = 31$. From left to right, the top row shows raster-ordering, Reverse Cuthill-McKee, and the 4-node Z order. The second row shows X-, Z-, and U-orders generated with bit-interleaving. The fixed-grid resolution in each dimension is $h_n = 6$. Bandwidth, bw, is included in the bottom label for each matrix, as well as the number of non-zeros, nz. Of all the orderings in Figure 4.11, RCM is by far the smallest bandwidth ($bw = 858$). The runner up, with a 30% wider bandwidth, is raster-ordering. The four remaining options are aesthetically pleasing, but their non-zero distributions have significantly less structure.

In similar fashion, Figure 4.12 provides the six orderings for $N = 4096$ MD-nodes on the unit sphere. Each row has $n = 31$ non-zeros, but in this case the fixed-grid resolution in each dimension is $h_n = 10$. Once again the top two orderings by bandwidth are RCM and raster-ordering.

While bandwidth is significant, the end-goal of reordering is to improve the benchmark for SpMV. Consider Table 4.4, which lists ordering impact on $N = 10^6$ CVT nodes on the unit sphere with $n = 50$ and $h_n = 160$ (i.e., the optimal h_n found in Figure 4.7). The first row of Table 4.4 provides bandwidths by order, and the second shows the speedup gained in the SpMV versus the benchmark for raster-ordering. In this case, the RCM has both the lowest bandwidth and highest

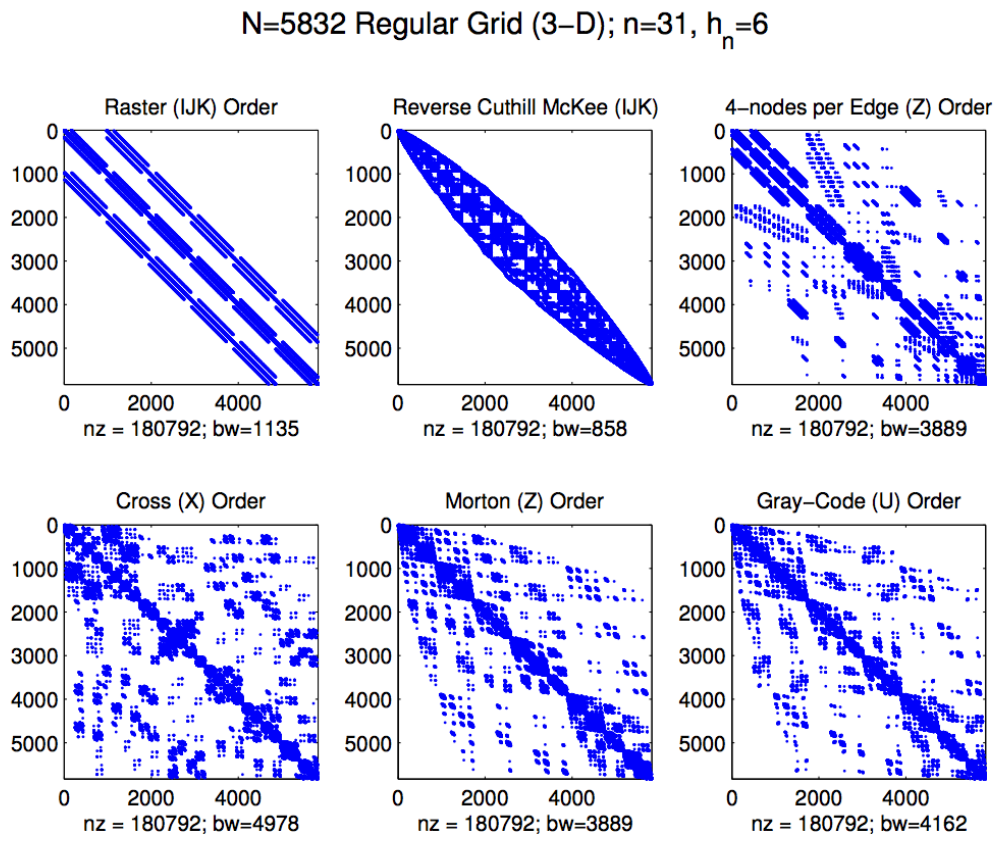


Figure 4.11: Impact of node orderings on an $N = 18^3$ regular grid in 3-D. Stencil size $n = 31$, fixed-grid resolution $(h_n)^d = 6^3$.

yield in the benchmark with a 9% improvement (5.06x faster than k-D Tree). Surprisingly the Z- and X-orderings are not far behind RCM even with 80x–120x wider bandwidths.

Table 4.4: Ordering Comparison for $N = 10^6$ CVT unit sphere. Input nodes are quasi-random and test the best-case scenario for reordering improvements. Stencil size $n = 50$, fixed-grid resolution $h_n = 160$.

	IJK	RCM	Z	X	U	4-node Z
Bandwidth	7885	6566	575606	819312	611513	513579
SpMV Speedup	1	1.09	1.06	1.06	1.04	0.94

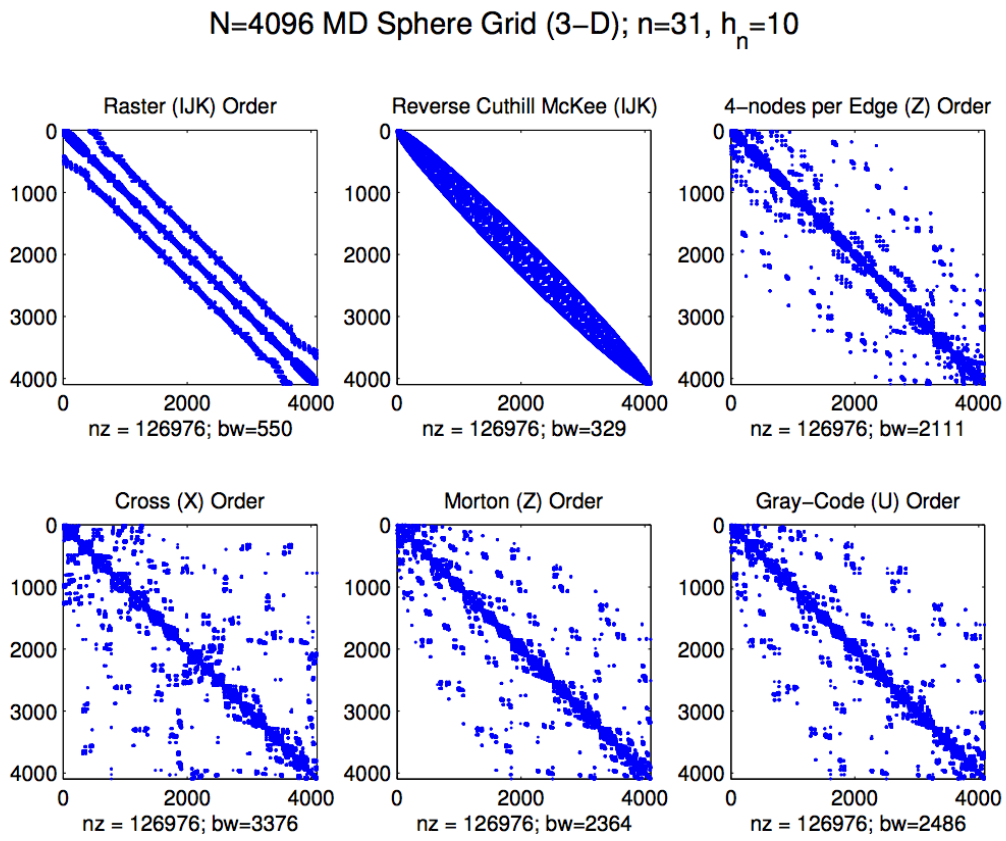


Figure 4.12: The impact of reordering on sparsity patterns for $N = 4096$ MD nodes on the sphere. Stencil size $n = 31$, fixed-grid resolution per dimension $h_n = 10$.

4.5 Conclusion and Future Work

This chapter introduced a fixed-grid method for RBF-FD stencil generation, and compared it to an efficient implementation of k-D Tree. The method proves itself capable of generating stencils twice as fast as k-D Tree. Reordering nodes during stencil generation improves later performance of some RBF-FD solutions by a factor of five, but with the help of bit-interleaving and space filling curves that performance can be boosted slightly more.

In addition to ordering nodes based on space filling curves, a bandwidth reduction algorithm named Reverse Cuthill-McKee was also employed to reorder the differentiation matrix. Reverse Cuthill-McKee results show that not only is the algorithm exceptional at reducing the bandwidth, it also results in the most improvement to SpMV benchmarks.

The implementation benchmarked above was developed as a pure CPU prototype with minimal

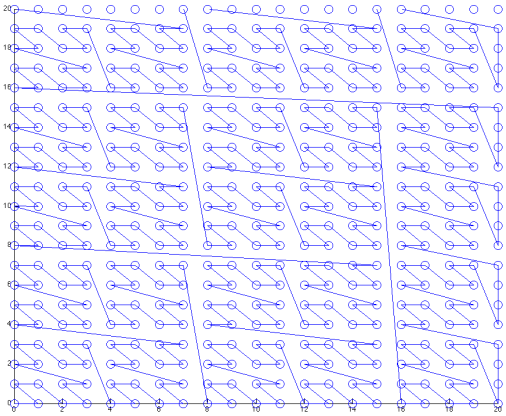
attention to optimization. Although RBF-FD only requires neighbor queries once, the fixed-grid method's long lasting positive impact on memory is sufficient to justify its continued use.

The C++ fixed-grid stencil generator defaults to the Raster-/IJK-order due to its ability to produce consistently small bandwidths. For situations when reordering is necessary for the best performance, the Reverse Cuthill-McKee algorithm is employed. In order to avoid MATLAB, RBF-FD applications in C++ rely on an implementation of the Cuthill-McKee algorithm provided by the BOOST library [1].

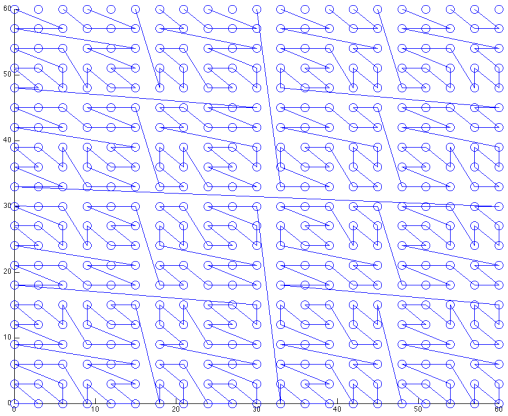
Due to the limited significance of stencil generation under RBF-FD, the overhead in implementing and debugging a GPU implementation of the fixed-grid method is hard to justify. No known investigations consider RBF-FD on moving nodes (e.g., for adaptive node refinement like [50]), although such research should find the fixed-grid method significantly relevant. Future work in this direction would easily justify a GPU implementation following [72, 73, 90, 102].

It also is worth noting that recent work by Connor and Kumar [29] developed operators to directly transform floating point coordinates into Z-orderings. The MATLAB fixed-grid prototype ([15]) includes a test of those operators, although the investigation is incomplete. Preliminary results are shown in Figure 4.13. A set of nodes are distributed in $[0, 20]^2$ with $dx = 1.0$ ($dy = dx$) in Figure 4.13a. Node coordinates are floating point representations of integer values, and the curve drawn shows the successful reproduction of a Z-ordering over the nodes. In Figure 4.13b, the same nodes are scaled to $dx = 3.0$ (i.e., still integer representations). A number of 3-node Z's appear at coordinates that are multiples of powers of two, but that is an anticipated flaw in the algorithm. The implementation fails when nodes are scaled to non-integer coordinates with $dx = 0.3$ (Figure 4.13c) and $dx = 0.05$ (Figure 4.13d).

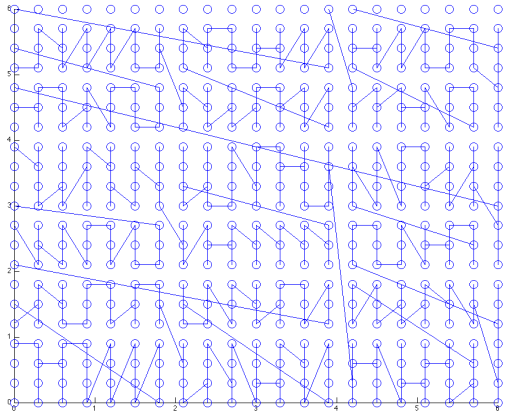
At this time, investigation into floating point Z-ordering is on hold as there is no urgent need for such an algorithm in RBF-FD. Direct communication with the lead author of [29] attributed the errors in Figures 4.13c and 4.13d to unexpected behavior/handling of doubles in MATLAB, or an unforeseen error in the design of their algorithm. The author indicated the possibility that these cases were not considered during testing.



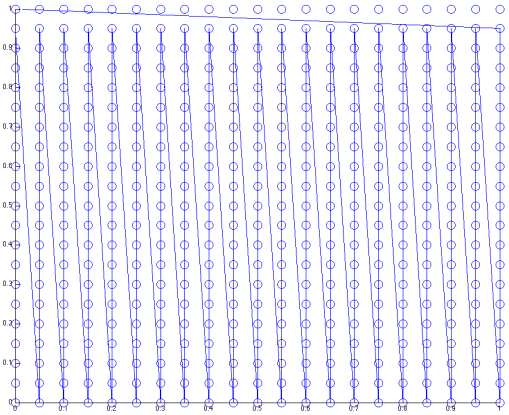
(a) Nodes in range $[0, 20]^2$, $dx = dy = 1$.



(b) Nodes in range $[0, 60]^2$, $dx = dy = 3$.



(c) Nodes in range $[0, 6]^2$, $dx = dy = 0.3$.



(d) Nodes in range $[0, 1]^2$, $dx = dy = 0.05$.

Figure 4.13: Preliminary results from Z-ordering nodes based on (double precision) floating point node coordinates (following algorithm in [29]). Our implementation functions well on integer coordinates when nodes are floating point representations of integers separated by $dx = 1$ (a). Increasing dx extends some edges to 3-node Z's, but is mostly correct (b). The implementation fails when nodes are not representations of integers (c) and (d).

CHAPTER 5

GPU SPMV (INCOMPLETE)

General Purpose GPU (GPGPU) computing is one of today's hottest trends in scientific computing. As problem sizes grow larger, it behooves us to work on parallel architectures, be it across multiple CPUs or one or more GPUs. With over 1 TFLOP/s peak possible throughput in double precision on a single GPU [2], a compelling argument is made for the latter.

In the last decade, CPU designers have been forced to transition to multi- and many-core chips in order to keep up with demand for continued growth in computational performance. Power constraints make further frequency scaling on CPUs almost impossible, which leaves increased core counts as the only viable solution for designers to continue satisfying Moore's Law [119].

GPUs originated as dedicated hardware for video games and computer graphics. Due to the inherent task parallelism in computer graphics (e.g., for rasterization), early GPU designers adopted a many-core strategy for hardware, which persists today. Computer graphics traditionally involved simple operations, so the majority of transistors on a GPU are dedicated to computation in vector pipelines with limited control for logic. The earliest GPUs had static rendering pipelines with fixed functions. In contrast to this, CPUs have always focused on fast serial and scalar operations. This means they allocate less transistors to computation in order to free space for the needs of advanced logic tasks like branching, caching, etc. [117, 119]. Figure 5.1 (Courtesy of NVidia, [117]) illustrates the difference between CPUs and GPUs at a high level. The high density of Arithmetic Logic Units

Figure 5.1: The GPU Devotes More Transistors to Data Processing (Image courtesy of NVidia [117])

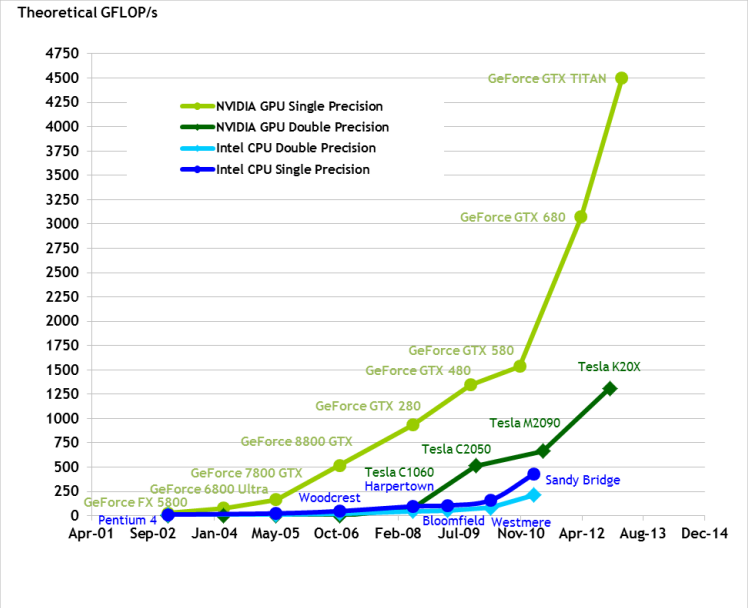
(ALUs) with simple control and caching units is signature of GPUs.

Thanks to the highly profitable and always demanding game industry, the static rendering pipeline of yore, was molded into a fully programmable and dynamic execution platform with a SIMD-like platform. Prior to 2006, the GPU had already evolved away from fixed-function pipelines and into limited programmable functionality with control over a subset of phases in the rendering process. At the time researchers were able—with substantial effort—to trick the GPU into solving scientific problems by encoding solutions within the graphics rendering process (see e.g., [79, 89, 119, 152]). GPUs, miles ahead in the many-core arena, only required a bit of flexibility in control and logic to compete against the CPU for general purpose computing.

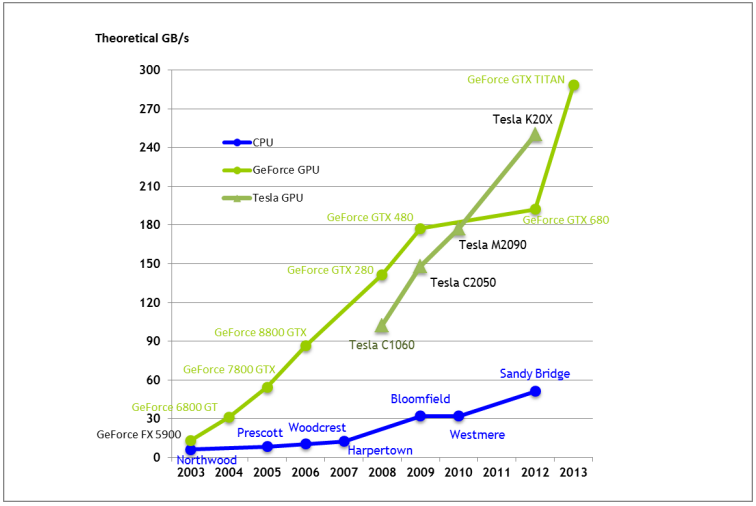
The release of NVidia's CUDA architecture and software stack [117] at the end of 2006, deprecated the need for tricks as developers could suddenly target the GPU with the ease of writing a generic C-like program. The monumental new hardware bypassed the graphics rendering pipeline and allowed programs to execute as though the GPU were just another compute device. The CUDA software stack included NVidia's compiler, *nvcc*, for developers to compile custom GPU codes. Since day one, developers who wanted to quickly transition existing code-bases to the GPU had access to the CUBLAS API, an implementation of the Basic Linear Algebra Subprograms (BLAS) suite on the GPU. CUFFT, a CUDA implementation of the Fast Fourier Transform (FFT), was also provided with the same justification. In many ways NVidia ensured that GPGPU computing was easily accessible to the general public.

The success of GPGPU is largely due to the incredible gap between compute capabilities of the GPU versus a CPU. Individual cores on a GPU operate at a lower clock frequency than CPU cores, but the sheer number of them leads to an astounding level of throughput in terms of floating point operations per second (FLOPs). Figure 5.2a (from [117]) illustrates the historical growth in terms of billions of FLOP/s (GFLOP/sec) achieved by NVidia GPUs versus Intel CPUs. Over the last decade the peak GFLOP/sec for a single GPU has sky-rocketed and consistently leads CPUs by a full order of magnitude. To supply much needed data to parallel cores, GPUs have also evolved a memory hierarchy with bandwidth up to five times higher than CPUs [117]. Figure 5.2b (courtesy of NVidia; [117]) compares the evolving memory bandwidth for NVidia and Intel.

In the years since CUDA was released a vast number of libraries and language extensions have surfaced to leverage the GPU architecture for scientific computing.



(a) Floating-Point Operations per Second for the CPU and GPU (Image courtesy of NVidia [117])



(b) Memory Bandwidth for the CPU and GPU (Image courtesy of NVidia [117])

Figure 5.2: Performance comparison for Graphics Processing Units (GPUs).

In 2012 NVidia open sourced CUDA to lower level virtual machine. This opened the possibility of directly compiling other languages like Python (Anaconda) to build kernels directly within the Python language for CUDA.

GPUs support single and double precision. RBF-FD certainly requires double precision at the

moment. Solving for the weights in single precision might work, but the lower precision would not be able to handle the ill-conditioned system. Projecting double precision weights into single precision results in weights that sum to $\approx 10^{-4}$, a poor approximation to zero.

5.1 Managing Expectations for the GPU

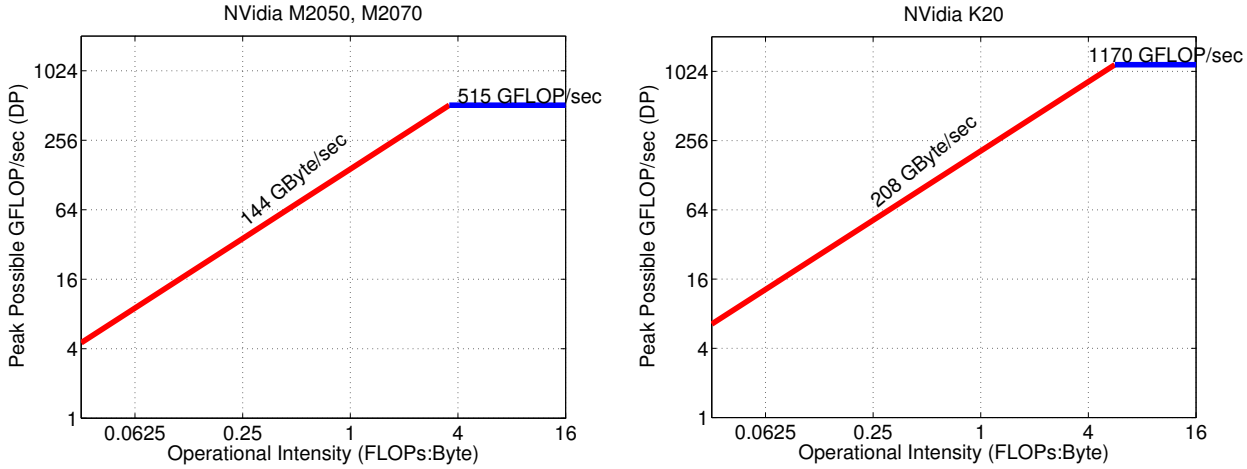
Success stories abound on the GPU citing massive gains. However, many like [122] appear to be impossible

RBF-FD is embarrassingly parallel with stencils applied independently. Unfortunately, perfect concurrency does not always imply perfect performance.

On the Fermi GPUs (Figure 5.3a) at 144 GB/s you need an OI of 4 to reach the peak of 515 GFLOP/sec per card. On the K20 (Figure 5.3b) you need 6 to hit the peak 1.17 TFLOP/s (208 GB/s). But with an OI of 1/16 you could get 9 GFLOP/sec on the Fermi and 13 on the K20 (compared to the 2 GFLOP/sec on Itasca).

CUDA is not the only solution for GPU computing.

In early 2009, the Khronos Group–the group responsible for maintaining OpenGL–announced a new specification for a general parallel programming lanugage referred to as the Open Compute



(a) Roofline Model for NVidia Fermi class GPUs, M2050 and M2070. SpMV Peak: 36 GFLOP/sec.

(b) Roofline Model for NVidia Kepler K20. SpMV Peak: 52 GFLOP/sec.

Figure 5.3: Roofline models manage expectations for maximum possible GFLOP/sec based on the Operational Intensity of kernels. The SpMV needed by RBF-FD has an operational intensity of 0.25.

Language (OpenCL) [96]. 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 already possible to see the CPU and GPU converging into 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.

5.2 Sparse Matrix Libraries

One thing missing from the original release of CUDA was sparse matrix algebra support. This led to investigations like [11, 146] and resulted in third party libraries like CUSP [12] and ViennaCL [125, 126], developed sparse matrix libraries for the GPU. The APIs of CUSP and ViennaCL are used like

Algorithms from [11] have since been included in the CUDA SDK as the cuSPARSE library.

NVidia eventually included a library named CUSPARSE as part of the CUDA SDK. Initial releases were Developers either implemented their own kernels (as we did in [21]) or

Performance of SpMV on GPU will not be optimal. Often one finds in literature examples of applications that have been accelerated by huge factors. In fact, speedup factors greater than 10x should never be possible for SpMV. In order to manage expectations of how much faster tasks will operate on the GPU we can turn to a *roofline model* [162]. The roofline model

[11] [156] etc.

5.3 Performance

5.3.1 GFLOP/sec

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 GFLOP/sec 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

In our initial implementation, published in [21], all bets were hedged in favor of OpenCL as the future of GPU computing. Custom kernels were developed to apply RBF-FD weights in the equivalent of a CSR SpMV.

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 hedged our bets on the functional portability.

Apple, Intel, AMD, NVidia all support OpenCL. Unfortunately Apple overrides vendor specific drivers and offers its own driver without support for concurrent kernel execution.

One serious limitation exists with the choice of OpenCL. Since the language is intended to function across a slew of platforms, the language itself is limited to a subset of hardware features common to the different vendors. Features like 2-D and 3-D textures, which are part of the CUDA standard, are vendor provided extensions in 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.

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

At the hardware level, a *thread* executes instructions on the GPU. On Fermi level GPUs, groups of 32 threads are referred to as *warps*. A warp is the unit of threads executed concurrently on a single *multi-processor*. In OpenCL (i.e., software), a collection of hardware threads performing the same instructions are referred to as a *work-group* of *work-items*. Work-groups execute as a collection of warps constrained to the same multiprocessor. Multiple work-groups of matching dimension are grouped into an *NDRange*. Figure 5.4 shows how threads/work-items are grouped into NDRanges. The *kernel* provides a master set of instructions for all threads in an NDRange [96].

Author's Note: Replace grid of thread blocks with ndrange of work-items

NVidia GPUs have a tiered memory hierarchy related to the grouping of threads described above. In multiprocessors, each computing core executes a thread with a limited set of registers. The number of registers varies with the generation of hardware, but always come in small quantities (e.g., 32K shared by all threads of a multiprocessor on the Fermi). Accessing registers is free, but keeping data in registers requires an effort to maintain balance between kernel complexity and the number of threads per block. Threads of a single work-group can share information within a multiprocessor through *shared memory*. With only 48 KB available per multiprocessor [115], shared memory is another fast but limited resource on the GPU. OpenCL refers to shared memory as *local memory*. Sharing information across multiprocessors is possible in *global device memory*—the largest and slowest memory space on the GPU. To improve seek times into global memory, Fermi level architectures include L1 on each multiprocessor and a shared L2 cache for all multiprocessors.

Under OpenCL, the equivalent to CUDA local memory is called *private memory* and is also reserved read-only per work-item in global device memory. CUDA's shared memory is labeled *local memory* and shared by threads of a work-group. The CUDA texture and constant caches are known as *Global/Constant Cache*, and similar to CUDA, exist outside of developer control. OpenCL refers to *global memory* and *constant memory* when referring to read/write and read-only sections of CUDA's global device memory (respectively). Finally, CUDA textures are equivalent to OpenCL *buffers* (1-D arrays) or *images* (2-D/3-D arrays) [96].

5.4.2 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.

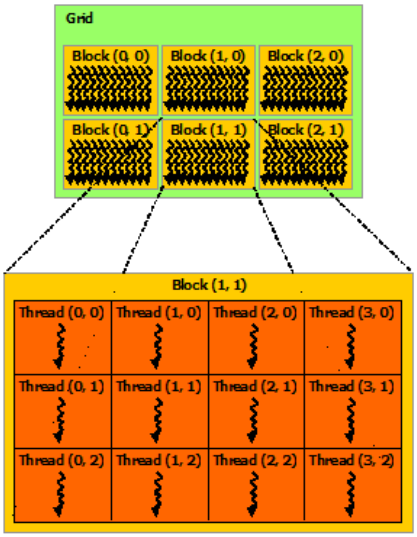


Figure 5.4: Grid of Thread Blocks (Image courtesy of NVidia [117])

5.4.3 Asynchronous Queuing

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

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

The two level parallelization can even be extended to three level parallelism with pThreads or OpenMP (see code samples attached to [118]). While OpenCL provides the means to target parallelism on either multi-core CPUs or many-core GPUs, it does not allow a parallel kernel on one hardware interact with a parallel kernel on the another. That is to say, an OpenCL kernel on the CPU cannot launch kernels on the GPU.

5.4.4 Custom Kernels

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.

Since our focus within this work is to lay the foundation for parallel computing with RBF-FD, we have made several simplifying assumptions in our code design. Libraries like PETsc, Hypre,

Trilinos and Deal.ii distribute sparse matrix operations in similar fashion to our approach. When work initially began on this dissertation, none of these competing libraries contained support for the GPU. PETsc is currently developing support for the GPU, but we have not had the chance to consider it yet.

Our codebase began as a prototype demonstrating the feasibility of RBF-FD operating on the GPU. The initial code, published in [21], was developed as N independent dot products of weights and solution values to approximate derivatives. Weights were stored linearly in memory, with the solution values read randomly. On the GPU, stencils were evaluated independently by threads, or shared by a warp of threads. Operating on stencils in this way implied that GPU kernels were to be hand written, tuned and optimized.

Much later, our perspective evolved to see derivative approximation and time-stepping as sparse matrix operations. This opened new possibilities for optimization and allowed us to forego hand optimization and fine-tuning of GPU kernels. With all of the effort put into optimizing sparse operations within libraries like CUSP [11], ViennaCL [125] and even the NVidia provided CUSPARSE [118], formulating the problem in terms of sparse matrix operations allows us to quickly prototype on the GPU and leverage all of the optimizations available within the third party libraries.

5.4.5 Explicit Solvers

Our initial implementation in [21] leverages the GPU for acceleration of the standard fourth order Runge-Kutta (RK4) scheme. Stencil weights are calculated once at the beginning of the simulation, and reused in every iteration. Subsequent runs load weights from disk. Ignoring code initialization, the cost of the algorithm is simply the explicit time advancement of the solution.

Figure 5.5 summarizes the time advancement steps for the multi-CPU/GPU implementation. The RK4 steps are:

$$\begin{aligned}
\mathbf{k}_1 &= \Delta t f(t_n, \mathbf{u}_n) \\
\mathbf{k}_2 &= \Delta t f(t_n + \frac{1}{2}\Delta t, \mathbf{u}_n + \frac{1}{2}\mathbf{k}_1) \\
\mathbf{k}_3 &= \Delta t f(t_n + \frac{1}{2}\Delta t, \mathbf{u}_n + \frac{1}{2}\mathbf{k}_2) \\
\mathbf{k}_4 &= \Delta t f(t_n + \Delta t, \mathbf{u}_n + \mathbf{k}_3) \\
\mathbf{u}_{n+1} &= \mathbf{u}_n + \frac{1}{6}(\mathbf{k}_1 + 2\mathbf{k}_2 + 2\mathbf{k}_3 + \mathbf{k}_4),
\end{aligned} \tag{5.1}$$

where each equation has a corresponding kernel launch. To handle a variety of Runga-Kutta implementations, steps $\mathbf{k}_{1\to4}$ correspond to calls to the same kernel with different arguments. The evaluation kernel returns two output vectors:

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

We choose $\alpha_i = 0, \frac{1}{2}, \frac{1}{2}, 1, 0$ and $\mathbf{k}_0 = \mathbf{u}_n$. The second output for each $\mathbf{k}_{i=1,2,3}$ serves as input to the next evaluation, $\mathbf{k}_{i+1}$. In an effort to avoid an extra kernel launch—and corresponding memory loads—the SAXPY that produces the second output uses the same evaluation kernel. Both outputs are stored in global device memory. When the computation spans multiple GPUs, steps $\mathbf{k}_{1\to3}$ are each followed by a communication barrier to synchronize the subsets $\mathcal{O}$ and $\mathcal{R}$ of the second output (this includes copying the subsets between GPU and CPU). An additional synchronization occurs on the updated solution, $\mathbf{u}_{n+1}$, to ensure that all GPUs share a consistent view of the solution going into the next time-step.

To evaluate $\mathbf{k}_{1\to4}$ in Equation 5.1, the discretized operators from Equation (3.12) are applied using sparse matrix-vector multiplication. If the operator D is composed of multiple derivatives, a differentiation matrix for each derivative is applied independently, including an additional multiplication for the discretized H operator. On the GPU, the kernel parallelizes across rows of the DMs, so all derivatives for stencils are computed in one kernel call.

For the GPU, the OpenCL language [96] assumes a lowest common denominator of hardware capabilities to provide functional portability. For example, all target architectures are assumed to support some level of SIMD (Single Instruction Multiple Data) execution for kernels. Multiple *work-items* execute a kernel in parallel. A collection of work-items performing the same task is called a *work-group*. While a user might think of work-groups as executing all work-items simultaneously, the work-items are divided at the hardware level into one or more SIMD *warps*, which are executed by a single multiprocessor. On the family of Fermi GPUs, a warp is 32 work-items [115]. OpenCL assumes a tiered memory hierarchy that provides fast but small *local memory* space that is shared within a work-group [96]. Local memory on Fermi GPUs is 48 KB per multiprocessor [115]. The *global device memory* allows sharing between work-groups and is the slowest but most abundant memory. In the GPU computing literature, the terms *thread* and *shared memory* are synonymous to *work-item* and *local memory* respectively, and are preferred below.

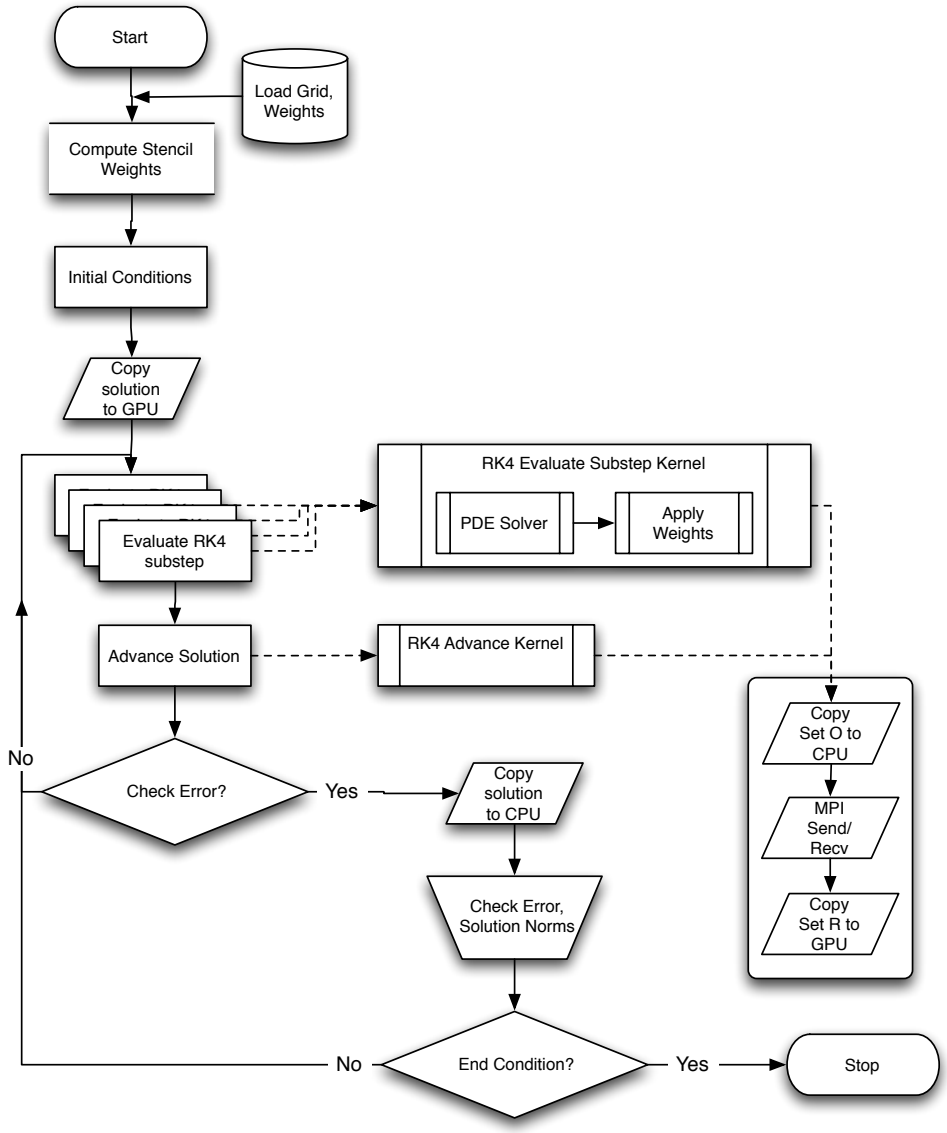


Figure 5.5: Workflow for RK4 on multiple GPUs.

Our initial implementation of RBF-FD on multiple GPUs ([21]) tested two approaches to computation of derivatives. In both cases, the stencil weights are stored in CSR format [11], a packed one-dimensional array in global memory with all the weights of a single stencil in consecutive memory addresses. Each operator is stored as an independent CSR matrix. The consecutive ordering on the weights implies that the solution vector is treated as random access.

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

Naive Approach: One thread per stencil. In this first implementation, each thread computes the derivative at one stencil center (Figure 5.6). The advantage of this approach is trivial concurrency. Since each stencil has the same number of neighbors, each derivative has an identical number of computations. As long as the number of stencils is a multiple of the warp size, there are no idle threads. Should the total number of stencils be less than a multiple of the warp size, the final warp would contain idle threads, but the impact on efficiency would be minimal assuming the stencil size is sufficiently large.

Perfect concurrency from a logical point of view does not imply perfect efficiency in practice. Unfortunately, the naive approach is memory bound. When threads access weights in global memory, a full warp accesses a 128-byte segment in a single memory operation [115]. Since each thread handles a single stencil, the various threads in a warp access data in very disparate areas of global memory, rather than the same segment. This leads to very large slowdowns as extra memory operations are added for each 128-byte segment that the threads of a warp must access. However, with stencils sharing many common nodes, and the Fermi hardware providing caching, some weights in the unused portions of the segments might remain in cache long enough to hide the cost of so many additional memory loads.

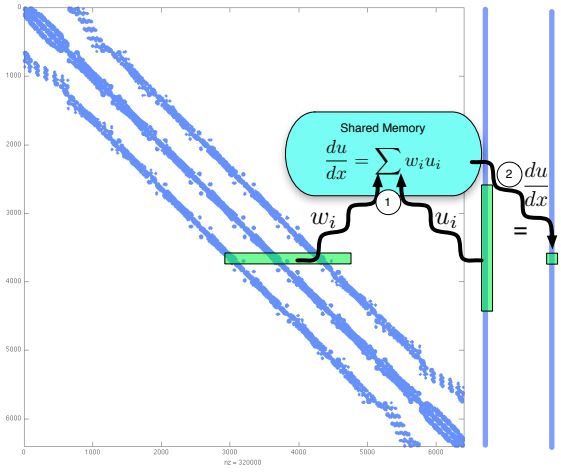


Figure 5.6: Naive approach to sparse matrix-vector multiply. Each thread is responsible for the sparse vector dot product of weights and solution values for derivatives at a single stencil.

Alternate Approach: One warp per stencil. An alternate approach, illustrated in Figure 5.7, dedicates a full warp of threads to a single stencil. Here, 32 threads load the weights of

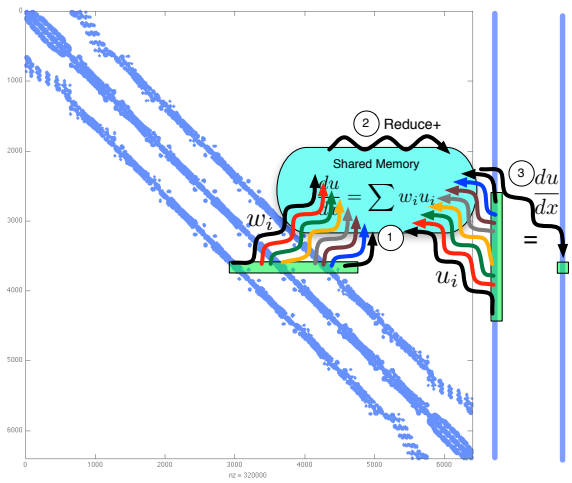


Figure 5.7: Alternative approach. A full warp (32 threads) collaborate to apply weights and compute the derivative at a stencil center.

a stencil and the corresponding elements of the solution vector. As the 32 threads each perform a subset of the dot product, their intermediate sums are accumulated in 32 elements of shared memory (one per thread). Should a stencil be larger than the warp size, the warp iterates over the stencil in increments of the warp size until the full dot product is complete. Finally, the first thread of the warp performs a sum reduction across the 32 (warp size) intermediate sums stored in shared memory and writes the derivative value to global memory.

By operating on a warp by warp basis, weights for a single stencil are loaded with a reduced number of memory accesses. Memory loads for the solution vector remain random access but see some benefit when solution values for a stencil are in a small neighborhood in the memory space. Proximity in memory can be controlled by node indexing (see Chapter 4).

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

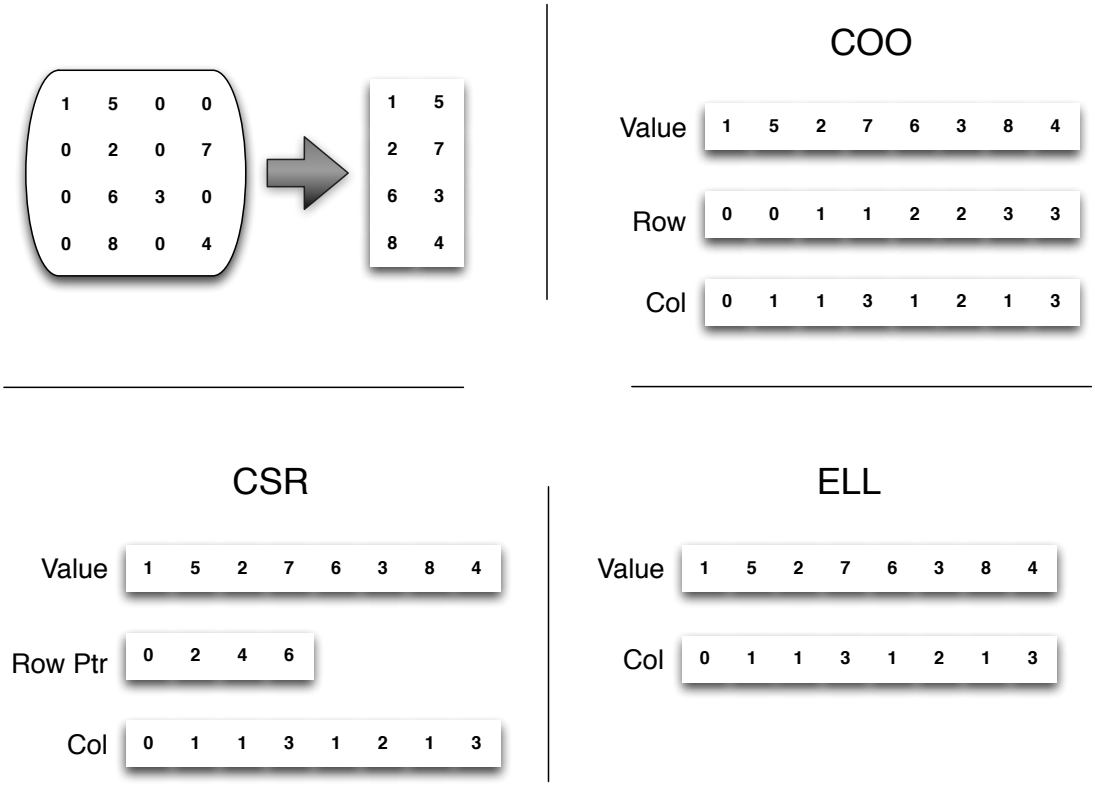


Figure 5.8: Sparse format example demonstrating a sparse matrix and corresponding storage data structures. ELL format assumes two non-zeros per row.

5.4.6 ViennaCL

In the time since the publication of [21] custom OpenCL kernels have been dropped in favor of a library called ViennaCL.

Sparse formats supported by ViennaCL are shown in Figure 5.8.

Other formats exist such as the Hybrid ELL plus CSR (HYB) format [11], Diagonal (DIAG) and Jagged Diagonal (JAD) variants, and a number of variants on the ELL format that operate in blocks (BELL), slices (SELL), or both (SBELL) [146]. The HYB format stores the bulk of a matrix in an ELL container and the excess non-zeros per row in a CSR format. HYB benefits from reduced storage and memory loads in the ELL, and maintains the generality of a CSR matrix.

GFLOP/sec calculated as:

$$GFLOP/sec = {}^{N*n*2}/_{t_ms} * 10^{-9}. \tag{5.2}$$

Paralution is another library that provides sparse matrix containers, iterative solvers, precon-

ditioners, etc. Paralution provides interfaces to plug into Deal.ii, OpenFOAM, and other packages, and runs on multi-core CPUs and GPUs.

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

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

The library includes a variety of sparse matrix formats, solvers, preconditioners, etc. Dense and sparse containers simplify targeting the GPU.

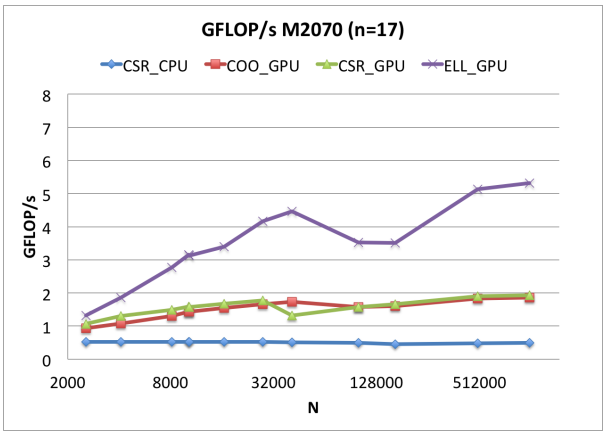
At the onset of work with ViennaCL two things were apparent: 1) the library was incredibly powerful but limited to square matrices; and 2) the library had no support for distributed computing. Our implementation of RBF-FD ([19]) required rectangular matrices resulting from domain decomposition. The enhancements to the API have since been added to the main branch of ViennaCL.

Due to the assumption that all stencils have equal size, the ELL format is preferred as the default.

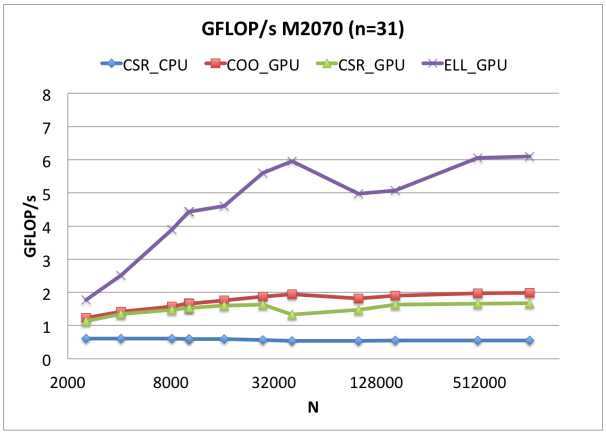
5.5 Cascade

The actual observed GFLOP/sec achieved on Cascade's M2070 GPUs with the ViennaCL package is shown in Figure 5.9. Figure 5.10 provides achieved GFLOP/sec for Cascade's K20 GPUs.

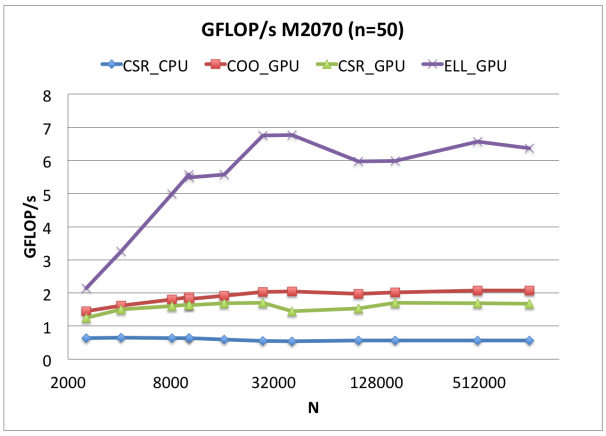
Figures 5.11a and 5.11b summarize the best achieved GFLOP/sec (in all cases the ELL format) for each stencil size on the M2070's and K20's respectively. Figures 5.11c and 5.11d show the achieved speedup each GPU gained over their respective Westmere and Sandy-Bridge CPUs. In the case of the K20, Figure 5.11d shows that the Sandy-Bridge architecture is closing the divide between CPUs and GPUs, although an order of magnitude difference in the GFLOP/sec still exists.



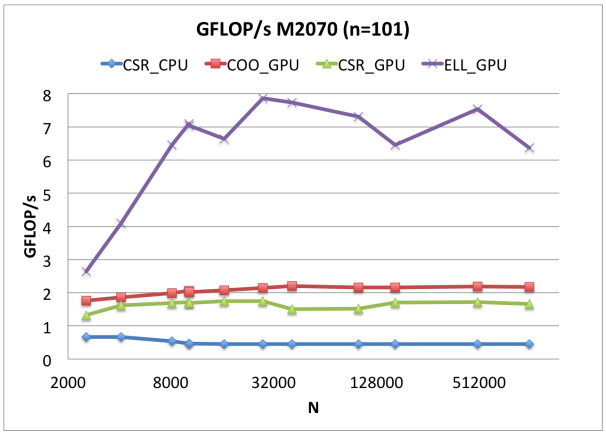
(a) Stencil size $n = 17$



(b) Stencil size $n = 31$



(c) Stencil size $n = 50$



(d) Stencil size $n = 101$

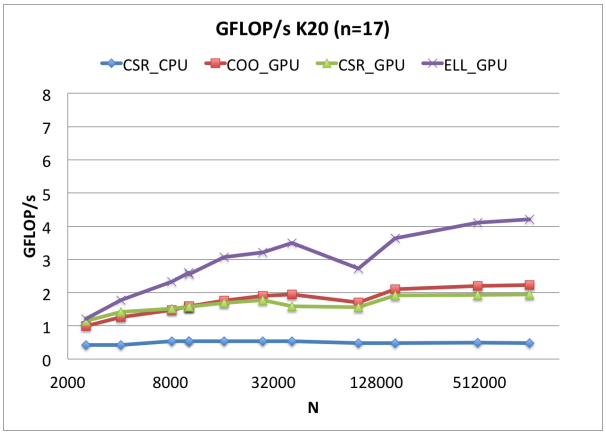
Figure 5.9: Performance comparison of CSR, COO and ELL matrix formats on Cascade's NVidia M2070 GPU and Intel Westmere CPU (Intel Xeon X5675).

5.5.1 Intel Phi

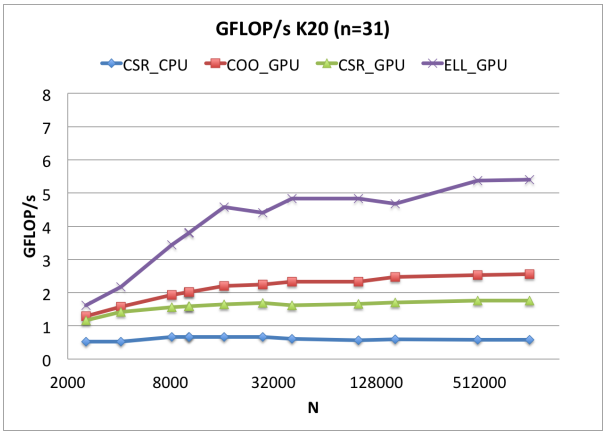
As part of future work into accelerating RBF methods, investigations are underway into the new Intel Phi architecture. The variety of hardware available on Cascade will help us establish a clear argument in the choice of accelerator type and resolve the dilemma between choosing Phi vs GPU for our method. 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.

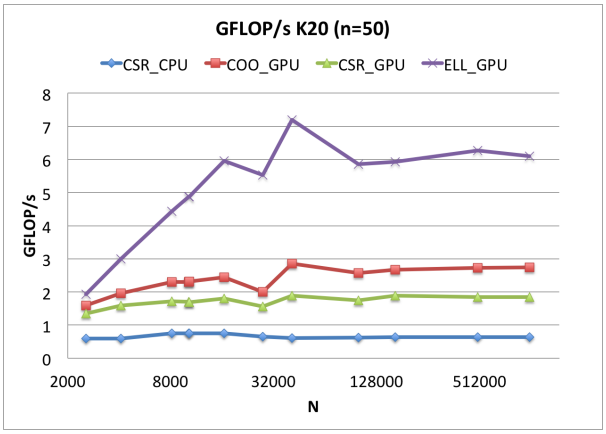
Figure 5.12 is provided for comparison with Figures 5.9 and 5.10. This stresses the point that



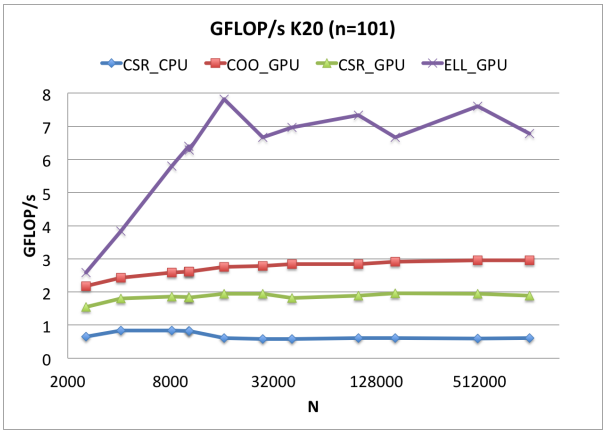
(a) Stencil size $n = 17$



(b) Stencil size $n = 31$



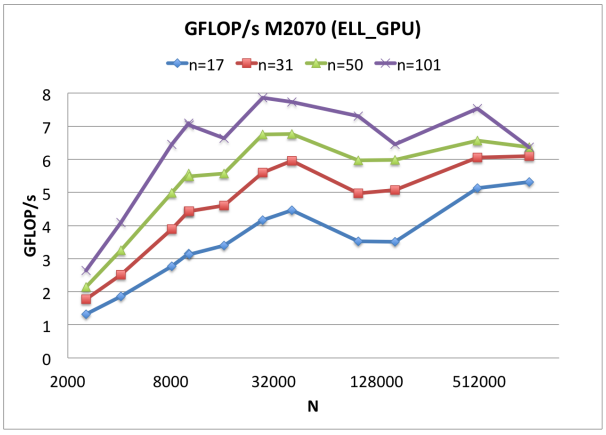
(c) Stencil size $n = 50$



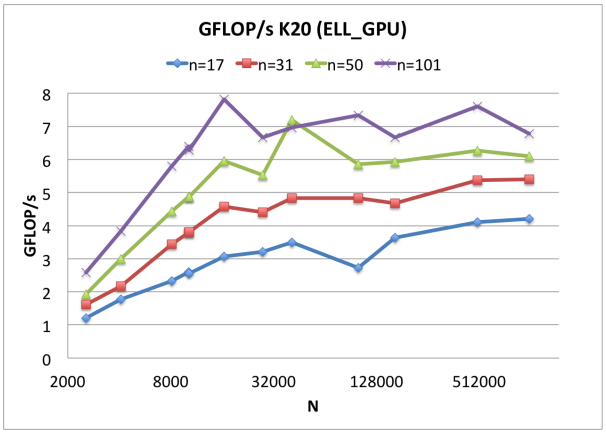
(d) Stencil size $n = 101$

Figure 5.10: Performance comparison of ViennaCL CSR, COO and ELL matrix formats on Cascade's NVidia K20 GPU versus the Boost::uBLAS CSR format on and Intel Sandy-Bridge CPU (Intel Xeon E5-2670).

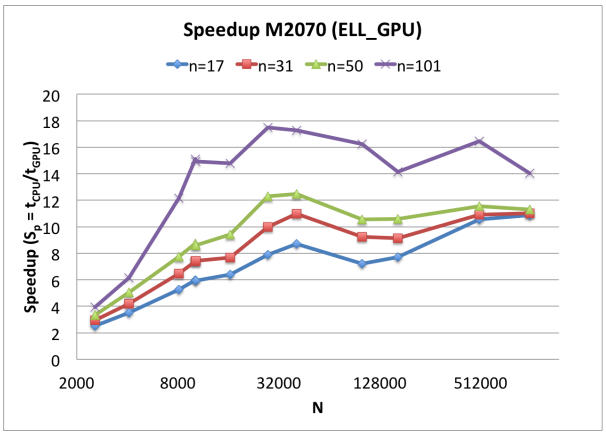
although OpenCL offers functional portability, performance portability is still limited.



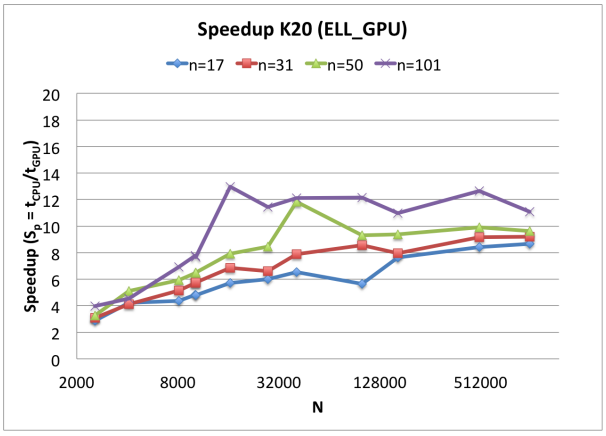
(a) GFLOP/sec, NVidia M2070 GPU (Cascade)



(b) GFLOP/sec, NVidia K20 GPU (Cascade)

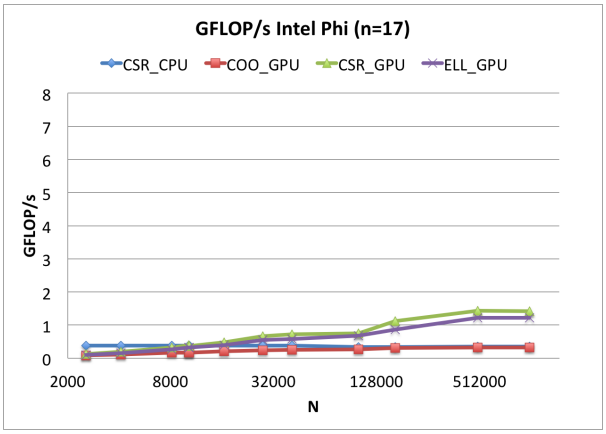


(c) Speedup, NVidia M2070 GPU (Cascade) vs. CSR format on Intel Westmere (Intel Xeon X5675)

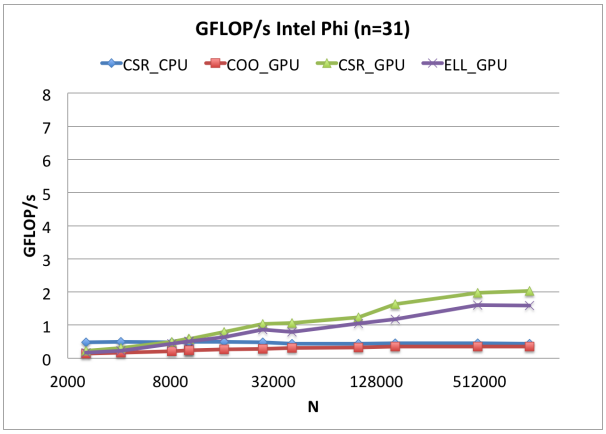


(d) Speedup, NVidia K20 GPU (Cascade) vs. CSR format on Sandy-Bridge (E5-2670)

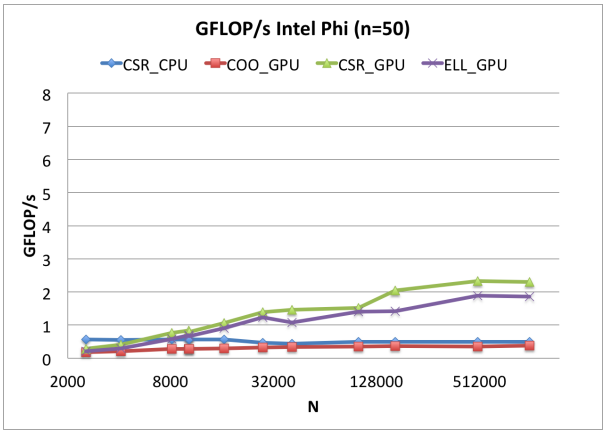
Figure 5.11: Comparison by stencil size for the ELL sparse matrix format.



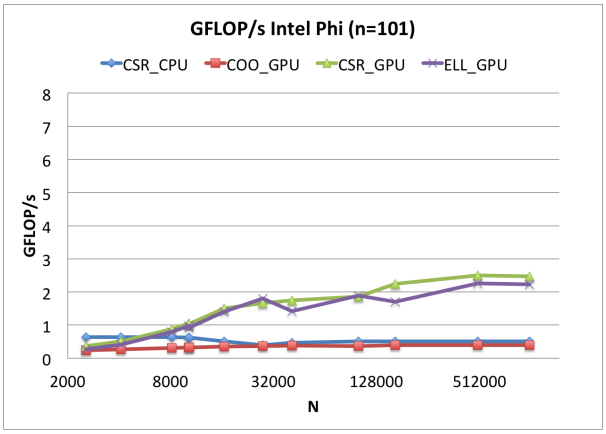
(a) Stencil size $n = 17$



(b) Stencil size $n = 31$



(c) Stencil size $n = 50$



(d) Stencil size $n = 101$

Figure 5.12: Performance comparison of ViennaCL CSR, COO and ELL matrix formats on Cascade's Intel Phi Accelerator, and the Boost::uBLAS CSR format on an Intel Sandy-Bridge CPU (Intel Xeon E5-2670). ViennaCL's kernels are optimized for NVidia GPUs, but function (albeit poorly) on the Intel Phi with the current release of Intel's OpenCL SDK (May, 2013).

5.6 Conclusions and Future Work

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

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

ViennaCL allows control of the number of work-items for each kernel.

ViennaCL includes a set of auto-tuning options.

The latest version of OpenMP (v4.0) introduces pragmas for offloading computation to accelerators. These pragmas will function similarly to pragmas from PGI, OpenACC, and the Intel MIC.

CHAPTER 6

DISTRIBUTED RBF-FD

Parallelizing applications in a distributed computing environment requires three design decisions [127]:

1. Partition the problem to distribute work across multiple processes. Intelligent partitioning impacts load balancing and communication latency.
2. Determine the subset of node information, solution values, etc. that are visible to each process, and establish index mappings to translate between a local context and the global problem.
3. Establish a local ordering of data. The right local order can improve solver performance and/or simplify data movement.

This chapter details the first implementation of RBF-FD designed for distributed computing environments and a few optimizations that help scale computation across a thousand processes. The discussion in this chapter applies to both multi-CPU and multi-GPU implementations of RBF-FD. Since each GPU is used as an accelerator, details of partitioning, local and global index mappings, and node ordering are managed on the CPU while the GPU computes only on a local context.

We operate under the assumption that inter-process communication is managed by the *Message Passing Interface (MPI)* [64]. Detail on how communication collectives are used is included below in Section 6.4. We use the term *MPI process* to refer to a process that occupies one CPU core and is associated with at most one GPU accelerator. MPI processes on multiple cores of the same CPU behave the same as processes on independent compute nodes in a cluster; the only difference being the fabric used for communication: cores of the same processor communicate via fast shared memory versus the slower QDR InfiniBand connecting cores of independent compute nodes. MPI always uses the fastest interconnect possible to connect processes.

In order to manage expectations of how well the code will run in a distributed environment, we turn to Amdahl's law, which states that the total speedup possible for p processors is limited by

the fraction of code that must run in serial [135]:

$$S_p = \frac{T_s + T_p}{T_s + \frac{T_p}{p}},$$

where T_s and T_p are the time spent in serial and parallelizable portions of code respectively. Amdahl's law reveals that scaling to a large number of processors, p, can make the time in parallelizable portions of code vanish, but T_s is constant. Communication is always included in T_s, so it is pertinent to minimize the cost of communication for maximum gain. Our initial implementation of distributed RBF-FD on multiple CPUs and multiple GPUs ([21]) had an extremely high cost of communication, which limited parallel execution to only a dozen or so processes. This chapter details the design and tuning of a distributed RBF-FD implementation capable of scaling across more than a thousand processors.

6.1 Partitioning

Parallelization of RBF-FD relies on *domain decomposition*. Domain decomposition methods simply subdivide/partition the computational domain of a PDE into *subdomains*, which are then solved independently. Subdomains can be solved sequentially by the same MPI process or in parallel across multiple processes. A number of methods exist for domain decomposition including Multiplicative Schwarz, Additive Schwarz, and Restricted Additive Schwarz (see related works in [140, 173]). The choice of method determines how the domain is subdivided, and how often data is passed between subdomains (i.e., how parallelizable the method is). Domain decomposition methods are well known as a form of preconditioning (see e.g., [10, 140]), but the two Additive variants of Schwarz decomposition are inherently parallel and naturally extend to distributed architectures [74, 173].

The parallelization strategy employed for RBF-FD in this work is equivalent to a Restricted Additive Schwarz (RAS), although our strategy was implemented prior to formal knowledge of the RAS method. In this case we assume that partitioning occurs within the spatial domain. At each time-step the PDE is first solved independently within subdomains, and then information is shared across subdomain boundaries to ensure a consistent global solution entering the next time-step. We also assume that subdomains are always assigned to independent processes, even though each process could operate on multiple subdomains in theory.

Following the notation in [140], consider domain, Ω, decomposed into overlapping subdomains Ω_1 and Ω_2, with a global boundary $\partial\Omega = \partial\Omega_1 \cup \partial\Omega_2$. At each time-step we seek the simultaneous solution to

$$\begin{aligned} \mathcal{L}u_1^{n+1} &= f \quad \text{in } \Omega_1, & \qquad \mathcal{L}u_2^{n+1} &= f \quad \text{in } \Omega_2 \\ \mathcal{B}u_1^{n+1} &= g \quad \text{on } \partial\Omega_1, & \qquad \mathcal{B}u_2^{n+1} &= g \quad \text{on } \partial\Omega_2, \\ u_1^{n+1} &= u_2^n \quad \text{on } \Gamma_{12}, & \qquad u_2^{n+1} &= u_1^n \quad \text{on } \Gamma_{21}, \end{aligned} \tag{6.1}$$

where Γ_{ij} indicates overlap between subdomains and is read as the subset of subdomain j required by subdomain i (i.e., $\Gamma_{ij} = \partial\Omega_i \cap \Omega_j$). A Dirichlet boundary condition connects subdomains at the interfaces. Although not considered here, future investigations may find Robin-type conditions of interest due to their convergence accelerating properties in the context of Schwarz methods for elliptic PDEs ([140]).

Equation 6.1 illustrates how nicely a PDE can be split in half, and then solved by independent processes. To enforce the Dirichlet conditions on Γ_{ij}, MPI transfers values u_j^n across interfaces at each time-step. Note that as the number of subdomains increases, additional constraints for Γ_{ij} must be imposed on each subdomain. The worst case a partitioning for p subdomains requires $p-1$ conditions (i.e., Γ_{ij}, for $i, j = 1, 2, ..., p$ and $i \neq j$). Ideally each Ω_i should only overlap with a small number of neighboring subdomains to minimize MPI communication.

The continuous form of decomposition in Equation 6.1 allows complete freedom in choosing how subdomains are actually partitioned. In this work we assume partitioning of the geometry on which the PDE is solved. Of particular interest here is how to partition the unit sphere, but the challenge of more general geometries is also kept in mind.

For ease of development and parallel debugging, partitioning is initially assumed to be linear along one spatial dimension (in this case the x-axis). Each partition/subdomain is then contiguous and overlaps with neighboring subdomains to the left and right. This approach is not uncommon when solving PDEs on rectangular domains (see e.g., [39, 149]).

Figure 6.1 illustrates the partitioning of $N = 10,201$ nodes on the unit sphere for four processes. Each partition, illustrated as a unique color, contains many stencil centers. A center's coordinates easily identify the containing partition. In cases where stencils in one partition depend on stencil nodes in another, we say that the dependencies (i.e., the nodes in Γ_{ij}) are *ghost nodes*. In many

ways ghost nodes are treated the same as any other stencil node. The MPI process in charge of a partition is fully aware of the ghost node coordinate, current solution value(s), etc. However, ghost nodes are managed by another subdomain/process, so changes must be explicitly synchronized in order to maintain consistency.

Alternating representations between node points and interpolated surfaces in Figure 6.1 illustrates the overlap regions where ghost nodes reside. Due to stencil dependencies, the representations of overlap regions are double-wide—i.e., they contain a set of ghost nodes for both the left and right partitions. The illustrated width of the overlap is between five and six nodes. As the stencil size increases, the overlap grows as $\sqrt{n}$. 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.

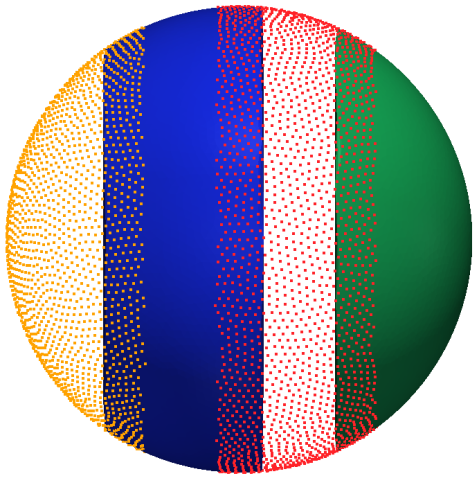


Figure 6.1: Partitioning of $N = 10,201$ nodes to span four processors with stencil size $n = 31$.

Linear partitioning is simple and easy to code. Each MPI process has at most two neighbors (left and right), so communication is straightforward. There are two concerns with this approach:

1. Scaling the number of processors can result in stencils spanning more than one narrow partitions in each direction. This introduces the need for complex communication collectives that go beyond just passing left and/or right.

2. In the case of irregular geometries, the partitioning of near uniform node distributions may result in imbalanced subdomains with unequal number of stencils.

Alternative sphere partitionings exist. In atmospheric and geophysical communities for example,

one often finds the cubed-sphere [87, 95], which transcribes a subdivided cube onto the sphere, and assigns projected rectangular subdivisions to individual processors. Another option is the icosahedral geodesic grid [124], which evenly balances the computational load by distributing equal sized geodesic triangles across processors. A complete list of options for partitioning the sphere is outside the scope of this work.

6.1.1 Graph Partitioning with METIS

One of the major concerns when partitioning for a distributed environment is load balancing. Consider, for example, the situation where $p-1$ processors have equal sized partitions, each with W_1 amount of work, and a p-th processors is allocated some larger amount of work, W_2. In that case, the maximum possible speedup on p processors obeys [74]:

$$S_p = \frac{(p-1)W_1 + W_2}{W_2} = 1 + (p-1)\frac{W_1}{W_2}. \tag{6.2}$$

The take-away from Equation 6.2 is that potential gains in parallelism are limited by the disproportionality of workloads.

To ensure load balancing, one commonly turns to *graph partitioning* algorithms. Formally, graph partitioning algorithms attempt to solve the k-way partitioning problem [94]:

> Given a graph $G = (V, E)$ with vertices (V) and any number of edges (E), partition V into k subsets, $V_1, V_2, ..., V_k$ such that
>
> - $V_i \cap V_j = \emptyset$ for $i \neq j$,
> - the size of each partition, $|V_i|$, is $|V_i| \approx \frac{N}{k}$,
> - $\bigcup_i V_i = V$,
> - and the number of edges whose incident vertices belong to different V_i is minimized.

In other words, we seek a partitioning that results in roughly equal sized subgraphs connected by as few edges as possible. In this case algorithms are applied to the adjacency graph produced by RBF-FD stencils (see § 3.2.1).

The output from graph partitioning algorithms are ideal for distributed applications for two reasons. First, partitions of roughly equal size ensure a balanced workload across processors. Second, edges crossing partitions correspond to ghost node dependencies, so minimizing those connections also ensures minimized data transfer via MPI.

A number of libraries exist for graph partitioning including Chaco [80], SCOTCH [120], and the METIS family of algorithms (e.g., METIS, ParMETIS, hMETIS) [94]. The algorithms vary by library, but in all cases vertex reordering and graph coloring capabilities are provided in addition to partitioning.

Our latest work (in [19]) partitions graphs via METIS using the library's *gpmetis* executable. During stencil generation, the associated adjacency graph is written to file. The METIS executable reads the file and performs a multi-level recursive bisection algorithm to partition the graph. At its core the algorithm first coarsens the graph into a sequence of smaller resolutions, applies a split to the smallest, and then pops back through the levels of recursion to project the coarse split onto finer graphs and smooth/correct the split at each level [94]. As input, METIS requires an undirected adjacency graph and the desired number of partitions. In return for the graph, METIS writes a file containing one integer per vertex, indicating the partition to which each stencil center is assigned. Note that METIS does not guarantee partitions will be contiguous. While this is often the case, METIS may find the alternative more fitting and assign disjoint subgraphs to the same MPI process.

The adjacency graph for RBF-FD is directed, so symmetry is induced in the associated matrix with $A + A^T$. The added connectivity is harmless to RBF-FD as symmetry is only meant for partitioning purposes and does not impact actual DMs. With respect to load balancing: cutting an edge connecting a stencil center to a stencil node is equivalent to cutting its transpose. METIS is equally resistant apply a cut whether the edge is directed or undirected.

Figure 6.2 provides an example of a METIS generated partitioning of $N = 10,201$ MD nodes for four processors. Three camera angles show the sphere viewed from the $(-x)$-axis (left), $(+x)$-axis (top-right), and $(-z)$-axis (bottom-right). Overlap regions are not illustrated.

6.1.2 A Load Balancing Comparison

Table 6.1 compares the load balance for linear and METIS-based partitionings of the unit sphere. The first two rows consider $p = 4$ processes as shown in Figures 6.1 and 6.2. Also shown in Table 6.1 is the case of $p = 16$ processors.

We assume each partition has $N_p = N_i + N_r$ nodes where N_i nodes are contained within each partition and N_r are ghost nodes. Although values at the N_r nodes are not computed within a partition, each process must still obtain their values. For the sake of argument we assume that the

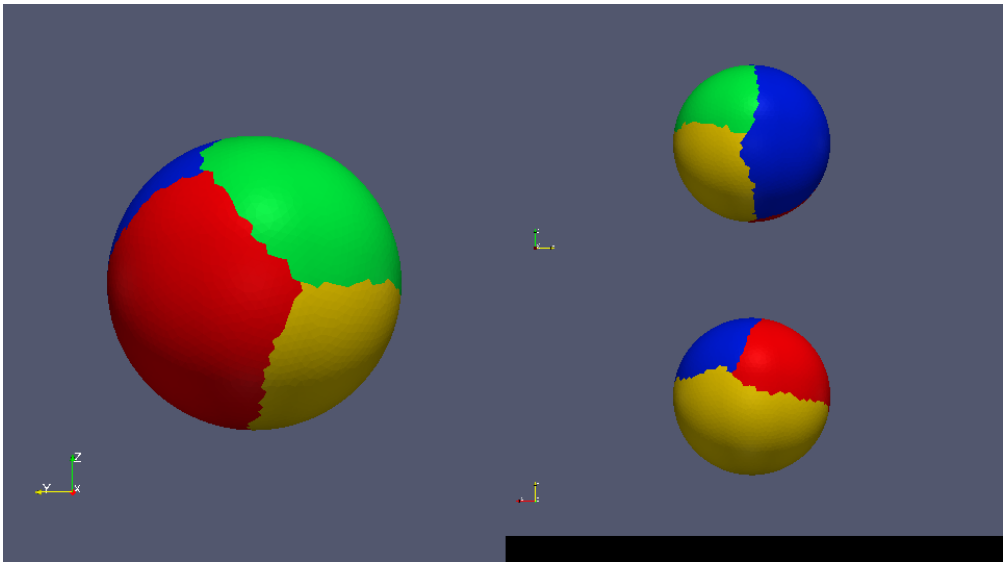


Figure 6.2: METIS partitioning of $N = 10,201$ nodes to span four processors with stencil size $n = 31$.

Table 6.1: Load balance comparison by partitioning for $N = 10201$ MD-nodes on the unit sphere with stencil size $n = 31$. A ratio of 1 is ideal.

p	Partitioning	Total ($\frac{\min N_p}{\max N_p}$)	Interior ($\frac{\min N_i}{\max N_i}$)	Ghost Nodes ($\frac{\min N_r}{\max N_r}$)
4	Linear	0.863	0.997	0.482
	METIS	0.986	0.971	0.913
16	Linear	0.551	0.980	0.264
	METIS	0.925	0.942	0.864

cost of communicating data for N_r nodes is equivalent to the cost of computing them—in practice communication is much more expensive. Based on these assumptions, we consider the ratio of work across partitions, $\frac{W_1}{W_2} \approx \frac{\min N_p}{\max N_p}$, in the column labeled "Total". Recall that ideal balancing is achieved at a ratio of 1. Columns labeled "Interior" and "Ghost Nodes" show the load balancing of only interior and overlap regions, respectively, in an effort to diagnose the source of imbalance across partitions.

Based on the per process N_p's, it is observed that that linear partitioning works moderately well for $p = 4$ but becomes horribly imbalanced at $p = 16$. In both cases METIS proves significantly better with ratios > 0.9. Substituting values from Table 6.1 into Equation 6.2 leads to the expectation that linear partitioning should be limited to less than 9.3x speedup for $p = 16$ processes, while

METIS is bounded by at most 14.9x. Data for the partition interiors shows that both linear and METIS partitionings distribute roughly equivalent number of nodes per interior—in fact, linear partitioning appears to be the more consistent of the two options across processors. The problem with linear partitioning, however, boils down to balancing ghost nodes. There are two obvious issues with linear partitioning:

- End-cap partitions only have dependencies in one direction, whereas interior partitions have left and right dependencies. This immediately drops the ratio to $\frac{\min N_r}{\max N_r} \leq 0.5$.
- Since each overlap region is a small circle of the sphere and approximately $\sqrt{n}$ ghost nodes thick, the number of nodes in each region is therefore a function of its circumference. As p grows, the ratio $\frac{\min N_r}{\max N_r}$ scales as the smallest to largest circumferences for overlap regions.

While useful for development and testing, the conclusion is that even at moderate p the situation is dire for linear partitions. METIS is preferred for consistent partition sizes and minimized overlap, but we cautiously note the increasing imbalance in ghost nodes for METIS. Scaling tests in § 6.5 demonstrate that even METIS can fail to find a partitioning of the graph with properly balanced ghost nodes when p is large.

6.2 Index Mappings

Once a partitioning is available, each MPI process 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 MPI process, which run from 1 to the maximum number of nodes on that process.

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

$\mathcal{G}$	: 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 6.2: Sets defined for stencil distribution to multiple CPUs

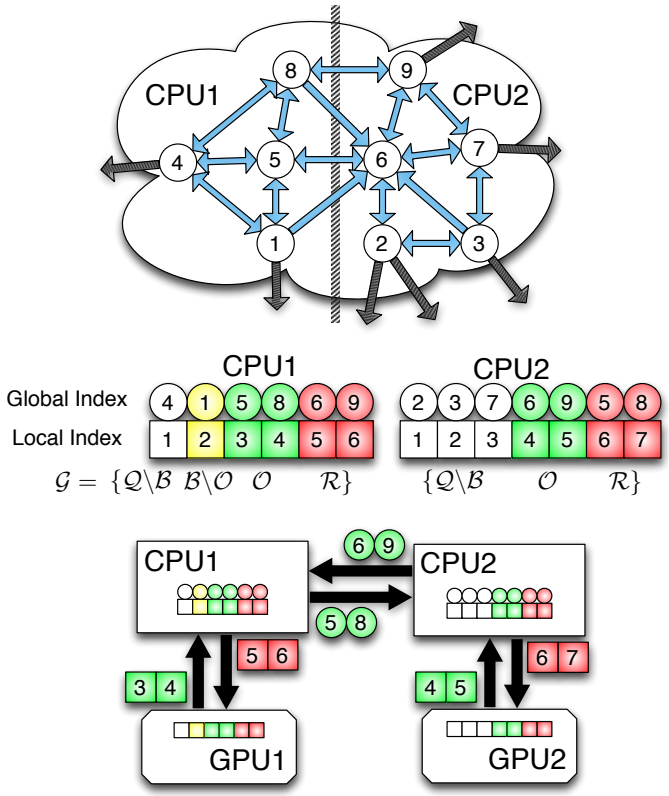


Figure 6.3: Partitioning, index mappings and memory transfers for nine stencils ($n = 5$) spanning two CPUs and two GPUs. Top: the directed graph created by stencil edges is partitioned for two CPUs. Middle: the partitioned stencil centers are reordered locally by each CPU to keep values sent to/received 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 6.2: $\mathcal{Q}\backslash\mathcal{B}$ is white, $\mathcal{B}\backslash\mathcal{O}$ is yellow, $\mathcal{O}$ is green and $\mathcal{R}$ is red.

Refer to Figure 6.3, which 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 $n = 5$. At the top of Figure 6.3, stencils are laid out with blue arrows pointing to stencil

neighbors and creating the edges of the directed adjacency graph. Note that the connections between nodes are not always bidirectional. For example, node 6 is in the stencil of node 3, but node 3 is excluded from the stencil around 6. Gray arrows point to stencil neighbors outside the small window and are irrelevant in the following discussion focused only on data flow between two processes on CPU1 and CPU2. Since each process is responsible for the derivative evaluation and solution updates for any stencil center, it is clear that edges intersecting the vertical line point to ghost node dependencies. 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 center of Figure 6.3 assigns the nine nodes to local sets for each process. The set of all nodes that a process interacts with is denoted by $\mathcal{G}$, which includes both the stencil centers within the partition (Ω_i), and all ghost nodes required to complete the calculations: $\bigcup_j \Gamma_{ij}$. We let $\mathcal{Q} \in \mathcal{G}$ contain indices for all nodes within the partition Ω_i. Then the set $\mathcal{R} = \mathcal{G} \backslash \mathcal{Q}$ contains indices of all ghost nodes. The set $\mathcal{O} \in \mathcal{Q}$ indexes nodes that are needed in other partitions (i.e., $\bigcup_j \Gamma_{ji}$). The set $\mathcal{B} \in \mathcal{Q}$ consists of nodes dependent on values from $\mathcal{R}$. Note that $\mathcal{O}$ and $\mathcal{B}$ overlap, but can differ in size due to lack of symmetry in the adjacency matrix. To capture the difference, set $\mathcal{B} \backslash \mathcal{O}$ are nodes that depend on $\mathcal{R}$ but are not sent to other processes, while $\mathcal{Q} \backslash \mathcal{B}$ are nodes that have no dependency on information from other processes.

Constructing these index mappings allows each process to ignore nodes/stencils outside their subdomain and focus only on their subset of the problem. Processes avoid both the cost of storing a complete RBF-FD differentiation matrix (DM) as well as full solution vectors. Instead, a local rectangular DM of size $|\mathcal{Q}| \times |\mathcal{G}|$ is constructed by each process with compressed solution vectors of size $|\mathcal{G}|$ to match. Our implementation follows a workflow that first generates stencils, and then partitions the adjacency graph in serial. In parallel, each processor loads its assigned partition, constructs index mappings, and solves for RBF-FD weights to generate a local DM.

6.3 Local Ordering

Figure 6.3 lists global node indices contained in $\mathcal{G}$ for each MPI process. Global indices are paired with a local mapping to indicate the internal node ordering for each process. The structure

of set $\mathcal{G}$,

$$\mathcal{G} = \{\mathcal{Q}\backslash\mathcal{B} \;\; \mathcal{B}\backslash\mathcal{O} \;\; \mathcal{O} \;\; \mathcal{R}\}, \tag{6.3}$$

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 $\mathcal{Q}\backslash\mathcal{B}$, yellow represents $\mathcal{B}\backslash\mathcal{O}$, 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 with computation, where distinguishing the set $\mathcal{B}$ allows the computation of $\mathcal{Q}\backslash\mathcal{B}$ without waiting on MPI communication to send and receive $\mathcal{O}$ and $\mathcal{R}$.

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

1. Transfer $\mathcal{O}$ from GPU to CPU
2. Encode $\mathcal{O}$ to send buffer
3. Distribute $\mathcal{O}$ to other CPUs and receive R from other CPUs
4. Decode $\mathcal{R}$ from received buffer
5. Transfer $\mathcal{R}$ to the GPU
6. Launch a GPU kernel to operate on $\mathcal{Q}$

The data transfers involved in this process are illustrated at the bottom of Figure 6.3. Each GPU is only aware of the local indexing in Equation 6.3. Benefiting from the local structure, set $\mathcal{O}$ is copied off the GPU and into CPU memory as one contiguous memory block. The CPU is then responsible for an *encode* process that maps local to global indices, copies elements of $\mathcal{O}$ into an MPI buffer with one entry for each receiving CPU (i.e., data is grouped by destination). MPI distributes the encoded buffer and receives $\mathcal{R}$ in a buffer that is sorted by origin. In general, Equation 6.3 need not have $\mathcal{R}$ structured the same as the receive buffer (i.e., data grouped by origin process). However, if the two do not match then received data must go through a *decode* process to shuffle elements into the proper local ordering. The reordered data is then copied as a contiguous block of memory

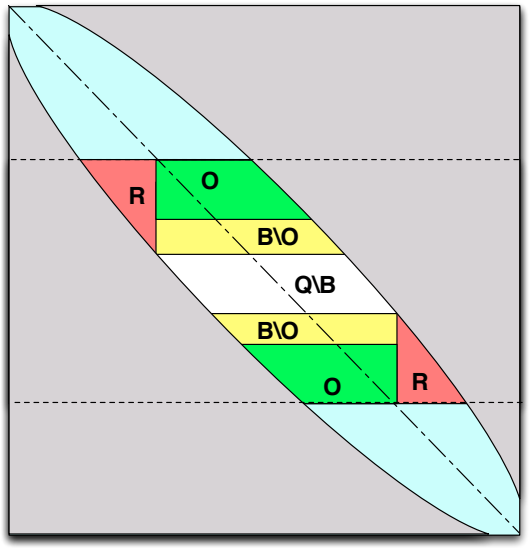


Figure 6.4: Decomposition for one processor selects a subset of rows from the DM. Blocks corresponding to node sets $\mathcal{Q}\backslash\mathcal{B}$, $\mathcal{B}\backslash\mathcal{O}$, $\mathcal{O}$, and $\mathcal{R}$ are labeled for clarity. The subdomain/partition is outlined by dashed lines.

to the GPU. Decoding data is avoided with a simple assumption on the ordering of $\mathcal{R}$. In § 6.5 bypassing the decode phase is one of several strategies employed for improved scalability.

6.3.1 Local Differentiation Matrix (DM) Structure

The local ordering of nodes leads to a unique DM on each MPI process. Consider Figure 6.4 which shows the index sets for a single partition in context of a global RBF-FD differentiation matrix. Here the gray area is zero, and non-zeros exist between bowed lines. The partition is illustrated as the contiguous set of rows between dashed lines. Set membership for each row is determined based on how the off-diagonal non-zeros in a row match vertically with the center diagonal. When an MPI process constructs the local DM for the partition, the two sets of $\mathcal{B}$ in Figure 6.4 are compressed into one contiguous set.

Figure 6.5 demonstrates the structure of a local DM on an MPI process. Labels denote the components of Equation 6.3. The local DM represents partition three of four from an original matrix with $N = 4096$ rows and $n = 31$ non-zeros per row. The RBF-FD stencils were generated on the $N = 4096$ MD node set, and partitioned with METIS.

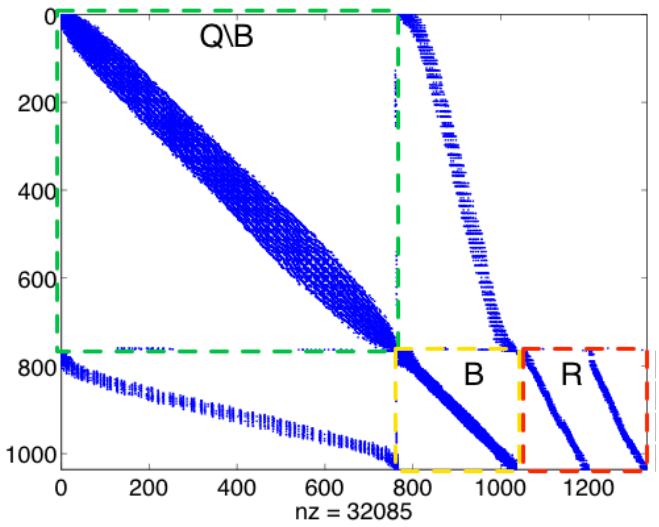


Figure 6.5: Spy of the local DM on processor 3 of 4 from a METIS partitioning of $N = 4096$ nodes with stencil size $n = 31$ and stencils generated with Algorithm 4.4 ($hnx = 100$). Blocks are highlighted to distinguish node sets $\mathcal{Q}\backslash\mathcal{B}$, $\mathcal{B}$, and $\mathcal{R}$. $\mathcal{R}$ 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.

6.4 Communication Collectives

With a partitioning and local ordering decided, communication collectives are established to transfer data between MPI processes. Two types of collectives are possible: all-to-all and all-to-subset.

All-to-all collectives pass data from every processor to all other processors. The most basic MPI implementation of an all-to-all, the MPI_Alltoall routine, is illustrated in Figure 6.6. For a cluster of p processes, MPI_Alltoall assumes that every process intends to send N_p bytes to all $p - 1$ processors. On the left, an output buffer with $p * N_p$ bytes is assembled locally on each process. Block sizes in Figure 6.6 are indicative of the number of bytes sent to each process. The subscripts on example data A, B, C indicate the destination process. When the collective executes, MPI_Alltoall scatters N_p bytes from all processes to effectively interchange/transpose data on the right of Figure 6.6.

Since RBF-FD partitioning via METIS restricts overlap to a subset of neighboring partitions, it makes sense to adopt an all-to-subset approach and avoid the overhead of unnecessary connec-

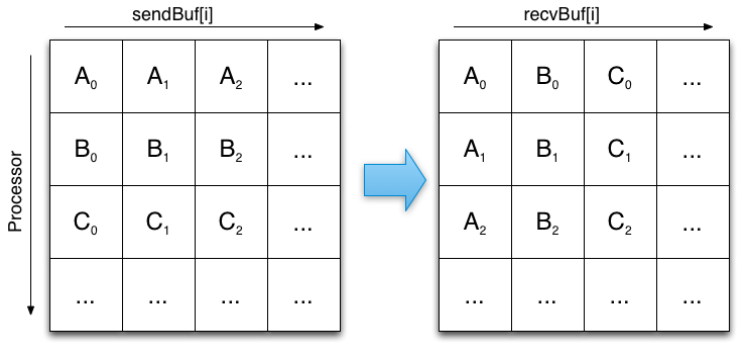


Figure 6.6: An "all-to-all" communication collective interchanges/transposes data across processes. All processors (e.g., 0, 1, 2) connect and transmit data subsets (e.g., A, B, C) to every other processor.

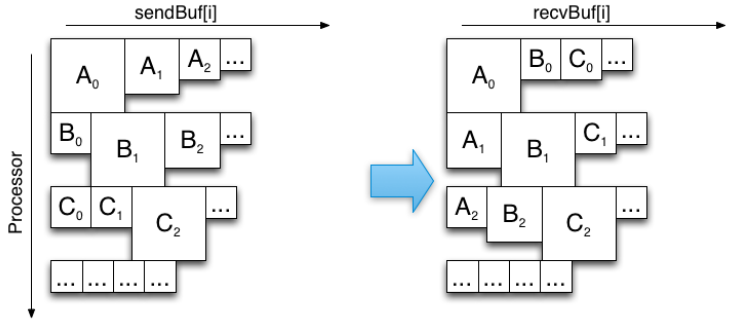


Figure 6.7: RBF-FD interchanges a variable number bytes between processes. An "all-to-subset" collective implemented via the MPI_Alltoallv collective compresses the interchange by allowing variable message sizes between processors. Zero-byte connections are skipped by the routine internally.

tions and memory padding required by MPI_Alltoall. Two types of all-to-subset are illustrated in Figures 6.7 and 6.8.

Figure 6.7 shows the behavior of an MPI_Alltoallv. MPI_Alltoallv improves on the all-to-all counterpart by allowing variable message sizes per connection. Unpadded messages transmit fewer bytes and reduce the overall communication time. Implementations of MPI_Alltoallv detect *zero-byte messages* (i.e., empty messages), and skip connections to their intended processes. This gives the routine an all-to-subset behavior. Note that MPI_Alltoallv should be used with caution: while zero-byte connections are avoided, the overhead in checking message sizes grows with the number of processors. For very large p the communication time can be dominated by those operations, defeating the gains in all-to-subset connection (see e.g., [8]).

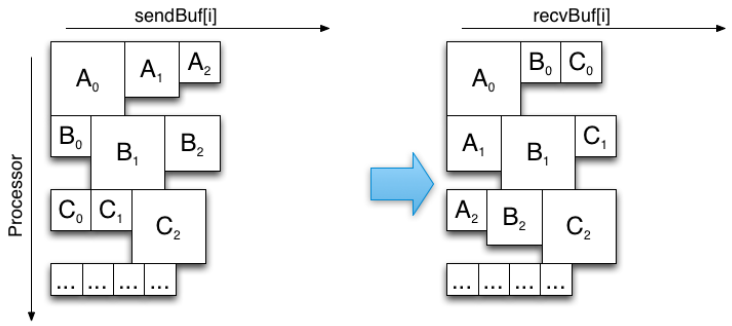


Figure 6.8: A true "all-to-subset" collective based on direct sends and receives allows for variable message sizes, and strictly truncates the number of connections between processors to only required connections. This collective is implemented with MPI_Send/MPI_Recv or MPI_Isend/MPI_Irecv.

MPI_Alltoallv is also a blocking collective: when the routine is called, all communication must complete before control is returned to computation. This blocking behavior prevents the use of MPI_Alltoallv in overlapping communication and computation within a distributed CPU environment. A distributed GPU environment can overlap MPI_Alltoallv communication due to the non-blocking behavior of GPU commands (this is tested in Chapter 7). As of this writing, a new version of the MPI standard (v3.0) introduces MPI_Ialltoallv, a non-blocking collective that would allow overlap on the CPU [64]. MPI_Ialltoallv is not considered here as popular MPI distributions did not yet offer MPI v3.0 compliant implementations.

Figure 6.8 shows a true all-to-subset collective implemented with direct sends and receives between processes that are either blocking (MPI_Send, MPI_Recv) or non-blocking (MPI_Isend, MPI_Irecv). Processes strictly communicate with required neighbors, so no zero-byte messages inflate communication time. A blocking version of this all-to-subset was employed in our first paper ([21]) for debugging purposes. MPI sends and receives were issued in a round-robin fashion. Unfortunately, the serialized connections made it infeasible to target $p > 10$.

In an effort to improve performance and scaling of our implementation an MPI_Alltoallv was employed. Appendix C presents benchmarks of the new collective running on Keeneland for comparison with our first paper and tests the scaling of the distributed SpMV up to 10 GPUs, one per compute node.

In what follows the MPI_Alltoallv implementation is taken as baseline for communication. An attempt is then made to further reduce communication times and improve the scaling of our implementation in a distributed CPU environment. By focusing on the CPU-only environment

we can test scaling on moderately large p (we go up to $p = 1024$). Scaling improvements for the CPU-only environment similarly improve the distributed GPU implementation (see Chapter 7).

6.5 Scaling Improvements

Two types of scaling are considered: weak and strong. Weak scaling monitors the growth in run-time as the number of processes (p) increases with the a constant workload per process, N_p. In other words the global problem size grows as $N = p * N_p$. Strong scaling keeps a fixed global problem size to consider $N_p = \frac{N}{p}$, where processes are issued diminishing workloads as p grows. The scalings reveal different properties:

- Weak scaling demonstrates the ability run problem sizes that are too large for a single CPU. Ideal weak scaling implies that the total run-time remains constant as additional parallelism is added.
- Strong scaling finds the point at which the cost of communication overcomes the cost of computation. Ideal strong scaling is linear in terms of speedup (i.e., $S_p = p$), but to achieve it is unrealistic due to the serial cost of communication.

For both scalings a simple idealized problem is tested where four derivatives are computed over a regular grid in 3-D and used as the intermediate vectors for an RK4 time-step. At the end of each SpMV a communication collective synchronizes the local derivative vectors. At the end of one thousand iterations, each process computes the local norm of the latest vector and an MPI_Reduce collects global norms for verification. Timings are reported for the distributed SpMV including MPI communication using *gettimeofday*.

Stencil sizes $n = 17, 31, 50$ and 101 are tested. Weak scaling is evaluated for $N_p = 4000$ per processor. Strong scaling considers $N = 4096000$ (i.e., $N = 160^3$) nodes. Timings are reported for total execution time in a distributed SpMV, as well as per-component times for computation and communication only.

Verification details are not reported here since they are only significant in ensuring that parallel results are consistent.

The benchmark is run on the Itasca HPC cluster at the University of Minnesota (Minnesota Supercomputing Institute). Itasca has 1,134HP ProLiant blade servers, each with two-socket, quad-core 2.8 GHz Intel Xeon processors. Each compute node has at least 24GB of RAM. Itasca has a

total of 8,744 cores available for computing with a total usable memory size of 31.3 TB [3]. Test cases are compiled with the Intel v2013.5 compiler ("icc") and the Intel provided MPI ("IMPI"). Optimization flag "-O3" is used to enable auto-vectorization, loop unrolling, blocking, etc.

Itasca's InfiniBand network topology is a 2-way fat-tree network. Each compute node, with 8 cores, is considered a *leaf* of the fat-tree. Processes within a leaf communicate via shared memory. Groups of 16 nodes connect via QDR InfiniBand to *leaf switches* at the second level of the tree. All leaf switches are in turn connected to two director nodes via 4x QDR InfiniBand. The topology exhibits 8:1 contention between nodes of the same leaf and additional 2:1 contention between leaf switches [123]. Our implementation of RBF-FD does not detect or adjust to network topology, so network contention may inflate communication times. Tuning for the network topology is reserved for future work.

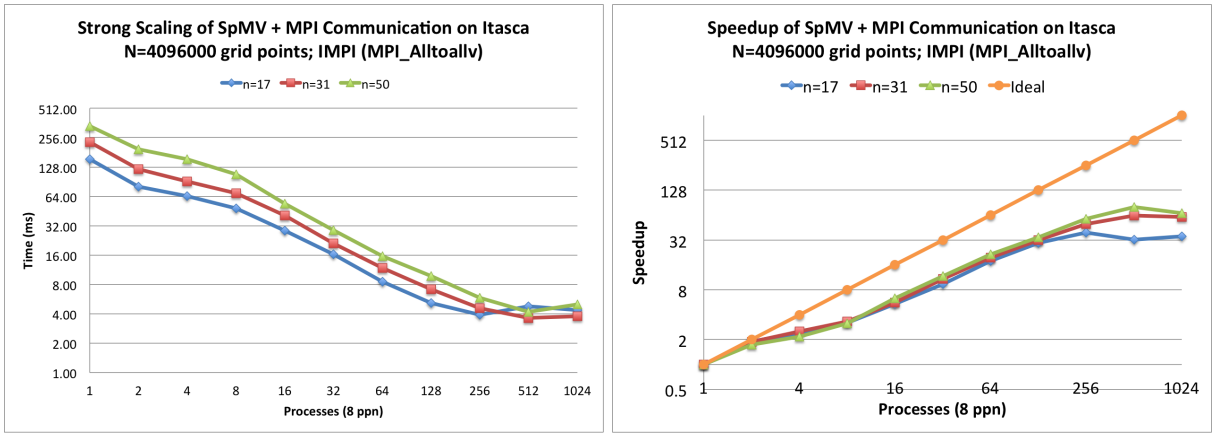
6.5.1 Baseline: MPI_Alltoallv

As a baseline for discussion, consider Figure 6.9 which shows the strong scaling achieved on Itasca with an MPI_Alltoallv collective. Strong scaling is assessed in terms of speedup: $S_p = {}^{t_1}/_{t_p}$ where t_1 is the serial time and t_p is parallel time. The average execution times for the distributed SpMV are provided in Figure 6.9a with corresponding speedups relative to the single process execution in Figure 6.9b. For $p <= 8$ the scaling is poor due to the growing number of processes on the same compute node contending for shared memory. The code scales well (i.e., nearly linear) between 8 and 128 processes, but the gains taper off beyond $p > 128$. Per-component data for the MPI_Alltoallv test case reveals the expected linear decrease in computation for $p > 8$ (see Figure 6.10a), and eventually the communication time exceeding computation time at $p \geq 128$ (see Figure 6.10b).

The data in Figure 6.11a shows MPI_Alltoallv weak scaling is poor. As p grows, the total time follows at a discouraging rate. Figure 6.11b, which presents the SpMV (computation) time only, verifies that beyond the initial contention on a single compute node, the SpMV is constant for all stencil sizes. That then implies the cost of communication could be reduced (or hidden).

6.5.2 Improvements

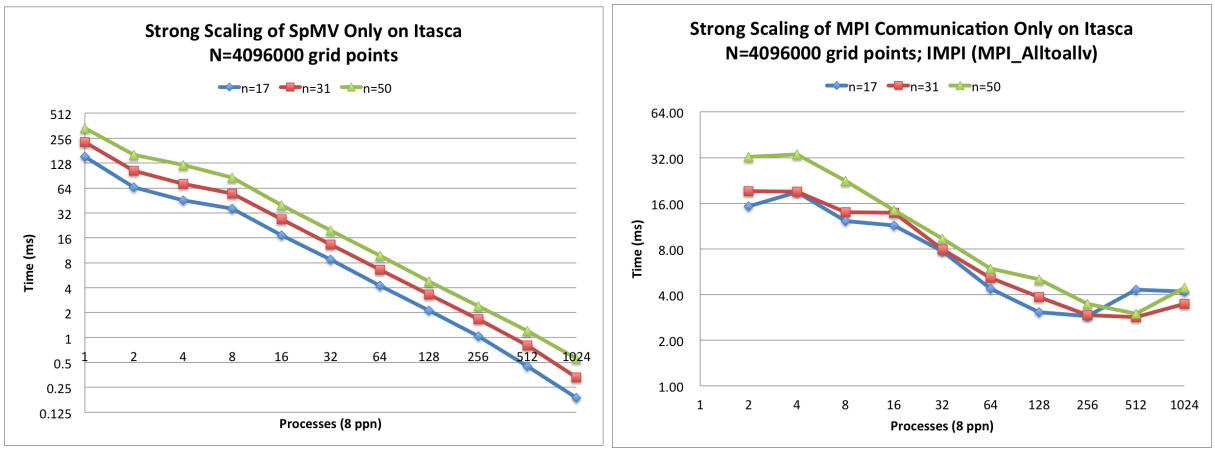
Figure 6.12 illustrates a progression of improvements made in effort to flatten the weak scaling curves from Figure 6.11a. Each tile of Figure 6.12 is a timeline for the distributed SpMV. Vertical



(a) Strong scaling

(b) Speedup ($S_p = {}^{t_1}/_{t_p}$)

Figure 6.9: Strong scaling of the distributed SpMV on $N = 4096000$ nodes (i.e., a 160^3 regular grid) and various stencil sizes. Communication handled by MPI_Alltoallv.



(a) SpMV only

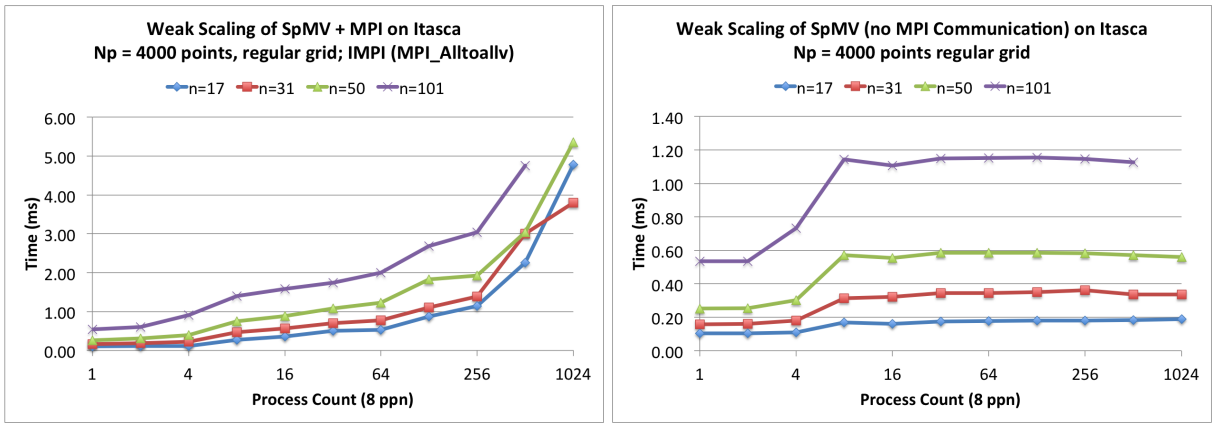
(b) Communication Only (MPI_Alltoallv)

Figure 6.10: Per-component benchmarks for strong scaling benchmarks on $N = 4096000$ nodes. Communication time outweighs computation time for $p > 128$ processes.

bars separate the SpMV in focus from execution of the next and last SpMV. Our implementation does not assume a global barrier before starting the next iteration, so load balancing is essential to limit wait times between processors. Operations are concurrent from top to bottom within each tile, and the top-most operations are launched first.

Tuning occurred as follows for the $n = 50$ node stencil case and $N_p = 4000$ nodes per process:

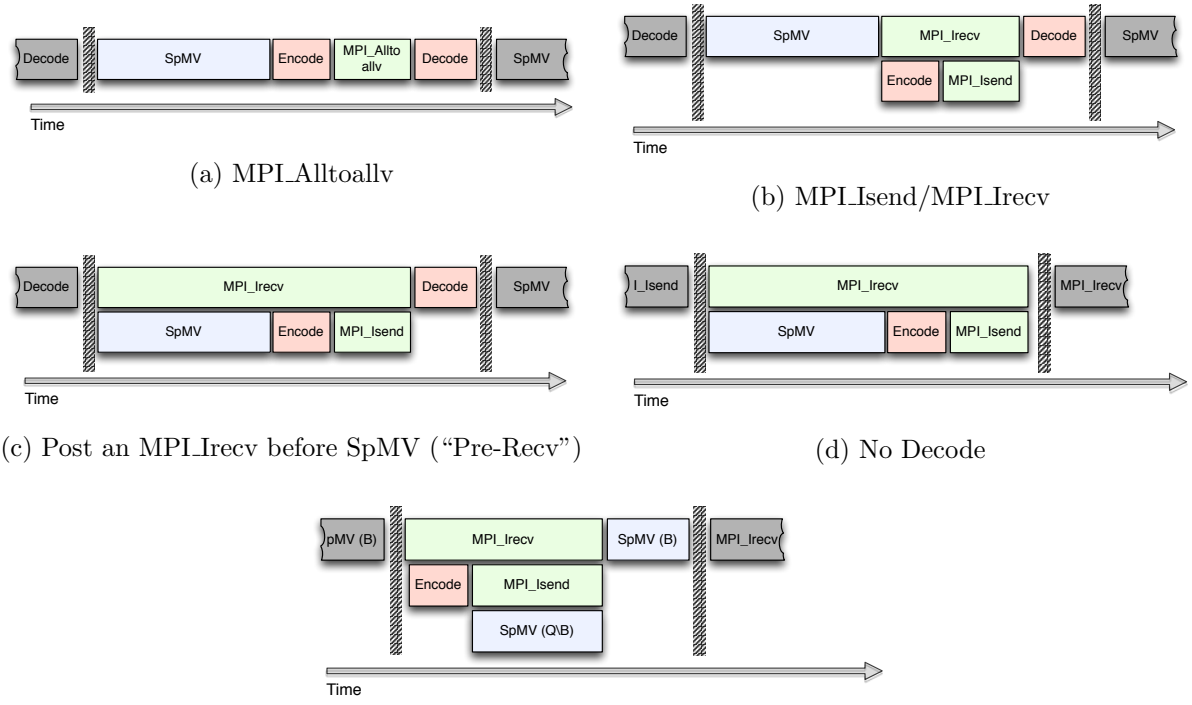
1. The entire SpMV for $\mathcal{Q}$ is executed, followed by a blocking MPI_Alltoallv collective (Fig-



(a) Weak Scaling

(b) SpMV only

Figure 6.11: Weak scaling of the SpMV. Increasing time for the SpMV reflects shared memory contention for processes on the same compute node (i.e., $p \leq 8$). Poor weak scaling is the result of growing communication times.



(a) MPI_Alltoallv

(b) MPI_Isend/MPI_Irecv

(c) Post an MPI_Irecv before SpMV ("Pre-Recv")

(d) No Decode

(e) Overlapping SpMV and Communication

Figure 6.12: MPI tuning steps to scale from 10 processors in [21] to 1024 processors here. Each horizontal timeline indicates the order of operations for a single distributed SpMV (i.e., between vertical bars). Task block sizes are not drawn to scale.

ure 6.12a). This is the baseline discussed above.

2. The MPI_Alltoallv is replaced by non-blocking MPI_Isend/MPI_Irecv (Figure 6.12b). This avoids unnecessary overhead in zero-byte message handling.

3. MPI_Irecv is issued *before* starting the SpMV to ensure MPI_Isends have pending connections when ready (Figure 6.12c). Posting the non-blocking receives early helps to hide some load balancing issues.

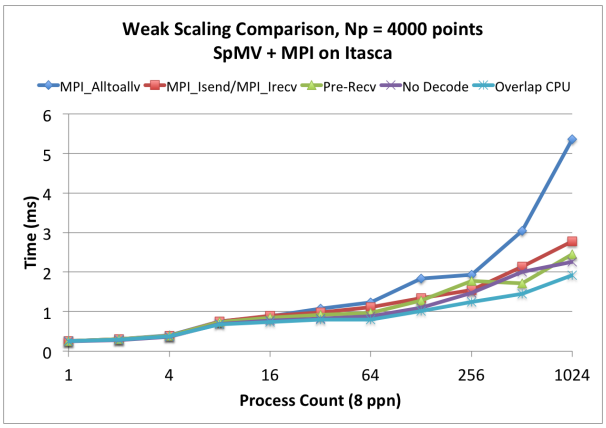
4. Based on the assumption that elements of $\mathcal{R}$ are sorted contiguous by neighboring process, the entire decode process is skipped (Figure 6.12d).

5. MPI_Irecv and MPI_Isend are issued before computing the SpMV for set $\mathcal{Q}\backslash\mathcal{B}$. MPI_Isend is non-blocking so the SpMV for $\mathcal{Q}\backslash\mathcal{B}$ starts before communication is complete. A barrier at the end of the first SpMV waits for communication to finish before launching the second SpMV on $\mathcal{B}$. This hides some cost of communication behind computation on the CPU (Figure 6.12e). This collective is the basis for distributed GPU computing approach in Chapter 7.

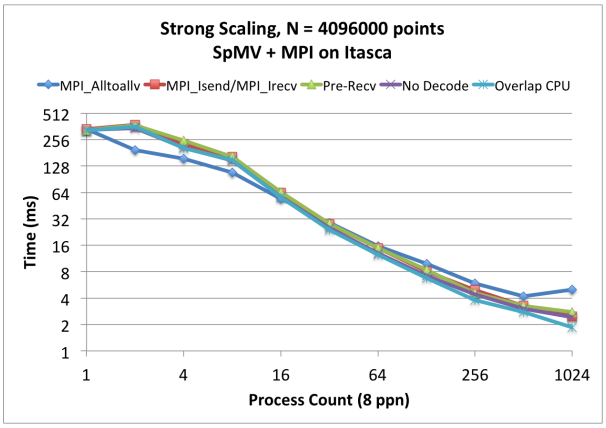
Weak scaling for each case is presented in Figure 6.13a. The gap between MPI_Alltoallv (blue) and the basic MPI_Isend/MPI_Irecv (red) in Figure 6.13a nicely captures the substantial loss of time due to zero-byte message tests. Posting MPI_Irecv before the SpMV (see the green line) shows moderate improvement for large p, and reflects that the load balancing of the test is getting worse. At $p = 1024$ processes the overlapping CPU approach is 2.8x faster than the baseline. Although its scaling is not ideal, overlapping the SpMV and communication does represent a significant improvement over the original MPI_Alltoallv.

Our overlapped SpMV approach is similar in concept to the "hybrid vector mode with naive overlap" adopted by Schubert et al. [134]. However, whereas Schubert et al. consider a two-level parallelism with each MPI process splitting into a number of OpenMP threads, our implementation is limited to a single CPU thread per MPI process. The comparison is bit surprising. Data provided in [134] suggest that the vector mode with naive overlap actually scales worse than the case when no overlap is used. Contrary to this, Figure 6.13a indicates that splitting the SpMV improves scaling compared to a non-overlapped approach. In the case of Schubert et al., their data shows better scaling for a non-overlapped

In similar fashion Figure 6.13b shows strong scaling for each of the collectives. Differences in curves are difficult to discern except for $p \leq 16$ and $p = 1024$. Starting with the latter case, observe



(a) Weak scaling comparison for $n = 50$ and $N_p = 4000$ on a 3-D regular grid with maximum resolution $N = 4096000$ on Itasca.



(b) Strong scaling comparison for $n = 50$ and $N = 4096000$ on a 3-D regular grid on Itasca.

Figure 6.13: Weak and Strong Scaling Improvements

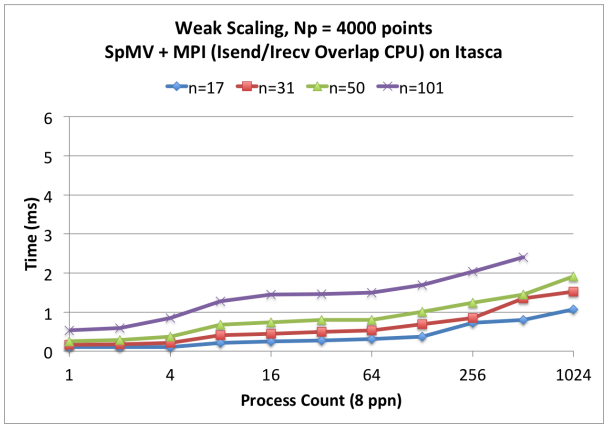
that the trend of near linear scaling for the overlapping collective persists at $p = 1024$ rather than bottoming-out in the fashion of MPI_Alltoallv.

For small p the MPI_Alltoallv is up to 2x faster than MPI_Isend/MPI_Irecv variants. For $p \leq 8$ it is safe to assume MPI_Alltoallv makes better use of local shared memory to communicate within the same node. Beyond $p = 8$, the collective most likely improves on MPI_Isend/MPI_Irecv by optimally packing messages or even breaking large messages into smaller pieces. Its also possible to some extent for the collective to adjust to the network topology. Duplicating those optimizations for the overlapped CPU collective is left for future work. Until then, RBF-FD in distributed CPU environments like Itasca should assume the use of MPI_Alltoallv collective for $p \leq 16$ and the overlapped MPI_Isend/MPI_Irecv in all other cases.

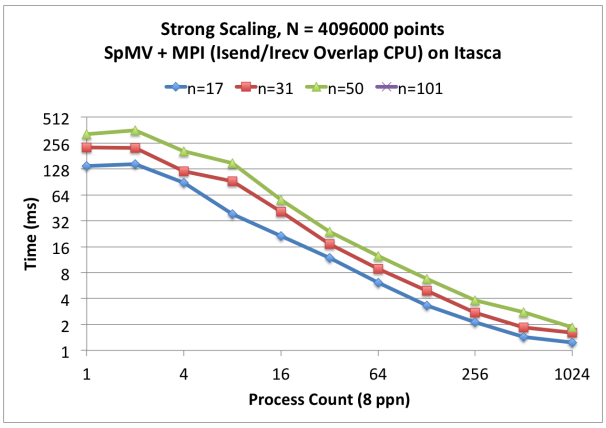
6.5.3 All Stencils Summary

Figures 6.14a and 6.14b demonstrate the scaling for the overlapped communication and computation and various stencil sizes ($n = 17, 31, 50, 101$).

Figure 6.14a shows that degradation in weak scaling starts at smaller p as the stencil size, n, grows. This is attributed—at least in part—to the proportional increase in the number of ghost node dependencies per partition sent/received. Based on the comparison in § 6.1.2, the ratio of ghost node counts was found to be $\frac{\min N_r}{\max N_r} = 0.39$ for $p = 1024$ and $n = 17$. The low ratio points

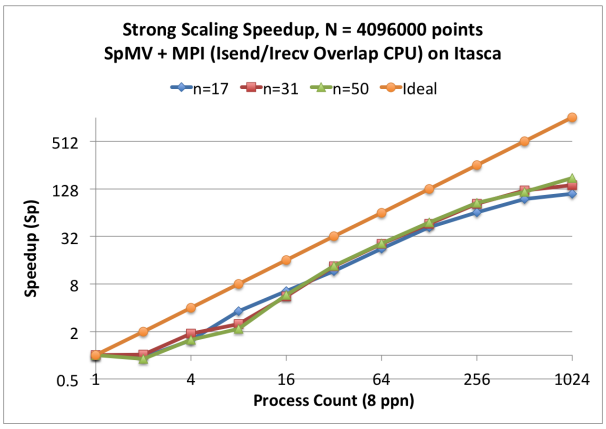


(a) Weak Scaling Time (All Stencils)

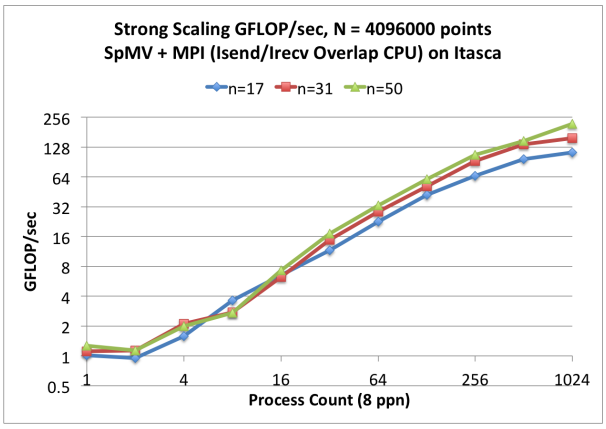


(b) Strong Scaling Time (All Stencils)

Figure 6.14: Weak and Strong scaling for various stencil sizes ($n = 17, 31, 50, 101$).



(a) Strong Scaling Speedup ($S_p = {}^{t_1}/_{t_p}$)

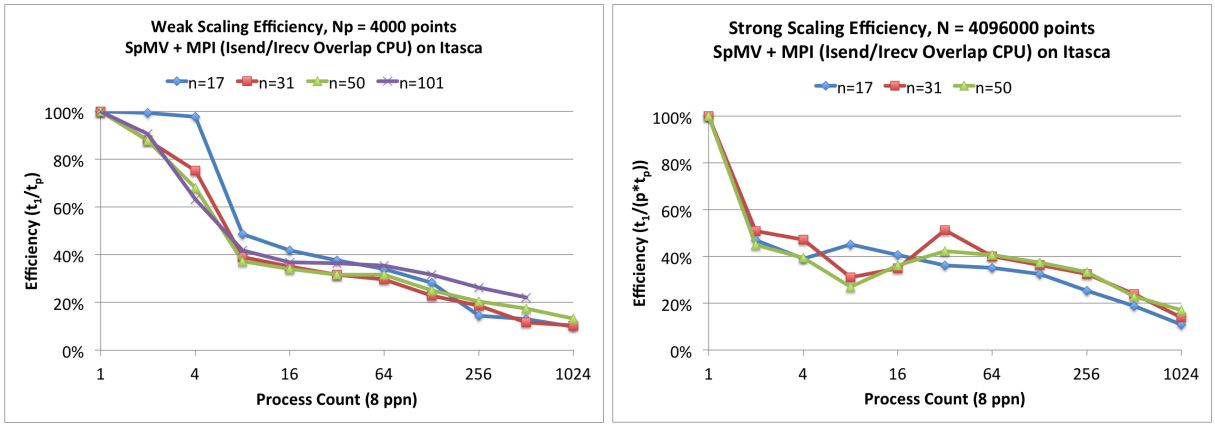


(b) Strong Scaling GFLOP/sec
($GFLOP/sec = ({}^{N*n*2}/_{t_{p(sec)}}) * 10^{-9}$)

Figure 6.15: Strong Scaling

to imbalanced communication loads as one cause for communication increase. This may be a limitation in METIS when partitioning domains for large p. If so, future work may be able to improve balancing with other graph partitioning methods.

Figure 6.14b demonstrates fair strong scaling for the implementation even though all three cases taper off toward the end. Figure 6.15a presents the strong scaling in turns of speedup. Here we see that, after an initial slow-down due to MPI_Isend/MPI_Irecv, the implementation scales linearly up to $p = 256$. For comparison against GPU performance, the measured GFLOP/sec for each strong



(a) Weak Scaling Efficiency $E_p = {}^{t_1}/_{t_p}$ (b) Strong Scaling Efficiency ($E_p = {}^{t_1}/_{p*t_p}$)

Figure 6.16: Scaling efficiency for various stencil sizes ($n = 17, 31, 50, 101$) on a regular grid with N_p maximum resolution $N = 4096000$ on Itasca.

scaling case is provided in 6.15b. The 110 GLFOP/sec (n=17) to 220 GFLOP/sec (n=50) achieved on 1024 cores of Itasca is a far cry from peta-scale performance, but still a respectable number for the memory bound SpMV.

For comparison with related work, and to assess how well the implementation utilizes hardware, we provide weak and strong scaling efficiencies in Figures 6.16a and 6.16b. Efficiency can have multiple definitions, but is always expressed as a percentage of linear scaling with 100% as ideal. Lower percentages indicate under-utilization of parallelism due to communication overhead or other issues. Weak Scaling Efficiency is calculated as:

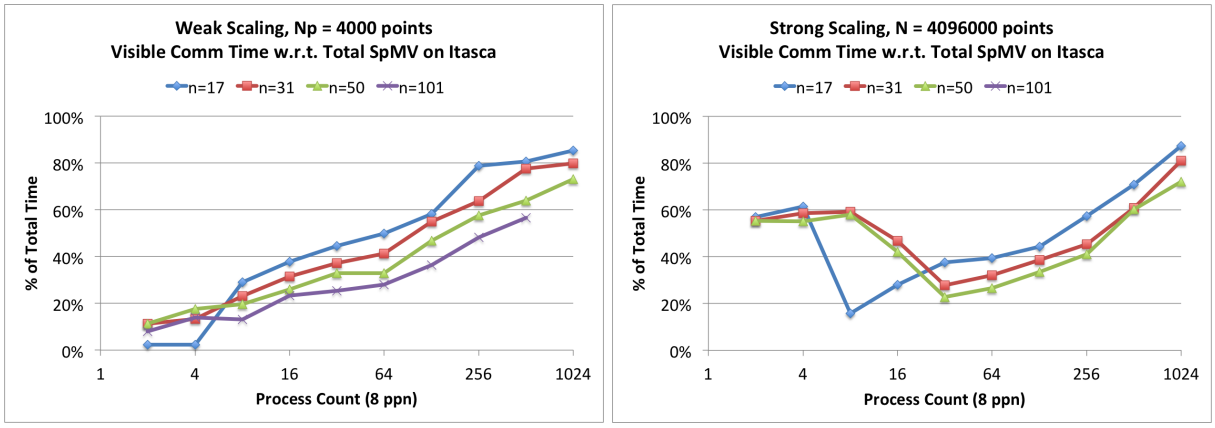
$$E_p = {}^{t_1}/_{t_p} * 100\%$$

where t_1 is the execution time on 1 processor and t_p is the time on p processors. Strong Scaling Efficiency is:

$$E_p = {}^{t_1}/_{(p*t_p)} * 100\%.$$

Figures 6.16a and 6.16b show our distributed RBF-FD achieves about 30% to 40% parallel efficiency in strong and weak cases for $p \leq 128$. At $p = 1024$ strong and weak scaling converge to a little over 15% efficiency.

Figures 6.17a and 6.17b quantify the overhead in communication for all weak and strong scaling test cases. Both figures show the percentage of *visible communication time*, t_{Vis}, during an iteration



(a) Weak Scaling

(b) Strong Scaling

Figure 6.17: Visible communication with respect to total SpMV Time. Lower is better.

of the distributed SpMV. The visible time is defined as:

$$t_{Vis} = {}^{(t_{Comm} - t_{SpMV1})}/_{t_{Total}}$$

where t_{Vis} is the subset of time between the first MPI_Irecv post and the start of the first SpMV (for $\mathcal{Q}\backslash\mathcal{B}$). Any communication hidden by the first SpMV is subtracted off t_{Comm} with t_{SpMV1}, the time spent computing.

The local minima in Figure 6.17b reveal an optimal choice of p by stencil size for overlapping communication and computation. Note that the minima at $p = 8$ for $n = 17$, and $p = 32$ for $n = 31$ and 50 correspond to strong scaling efficiency peaks in Figure 6.16b. Remarkably, we observe at those minima up to 80% of the communication time is hidden.

For weak scaling, the constant SpMV computation time (and constant overlap) means the linear trend in Figure 6.17a is purely growth in communication time with respect to p. Possible causes include (but are not limited to):

- *Worsening load balance.* As previously mentioned, METIS may be incapable of partitioning the domain optimally.
- *Network contention.* As the number of processes scale, the increasing network traffic is expected to slow MPI globally.
- *Improperly padded or aligned messages.* MPI, like any other software, operates most efficiently when data is properly aligned and messages have the right size.

All three of these concerns will be investigated as part of future work.

6.6 Conclusion

This chapter covered the design and tuning of the first distributed RBF-FD implementation for CPU and GPU clusters. The implementation was demonstrated to be scalable with 30% to 40% parallel efficiency on 128 processors of the Itasca HPC (CPU-only) cluster at the University of Minnesota given a problem size of $N = 4096000$ nodes and stencil sizes between $n = 17$ and $n = 50$.

A number of optimizations reduce communication times and increase scalability beyond the results in our first paper, [21]. Starting with an MPI_Alltoallv collective, the tuning effort ultimately leads to a scheme which overlaps communication and computation on the CPU. Chapter 7 extends this collective to overlap with computation on GPUs.

Our implementation leverages METIS for general domain decomposition and load balancing. Individual processes do not require a full mapping between RBF nodes and CPUs. Instead, processes assemble and operate on local linear systems for their subdomains. Communication between processes is limited to nearest-neighbor or all-to-subset collectives based on the intersection of subdomains.

CHAPTER 7

DISTRIBUTED GPU SPMV (INCOMPLETE)

Recent advances in distributed (multi-)GPU computing involve NVidia's GPU Direct technology. GPU Direct allows hardware like the PCI-e bus to directly access memory on the GPU device. CUDA V5.5 adds full support for MPI applications

Lawlor [104] wraps a subset of MPI collectives, memory copies between GPU and CPU, and—to some extent—callbacks to GPU kernels behind a simplified API called cudaMPI. Although the concept of cudaMPI is good for minimally invasive design, the library only provides a handful of a limited subset of MPI a for overlapping communication and computation.

OpenCL does not support must be explicitly copied to the CPU

Distributing SpMV across multiple GPUs poses a new problem: as previous 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.

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 [154].

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.

Similar approaches to overlapping communication and computation can be found in [134] and [149].

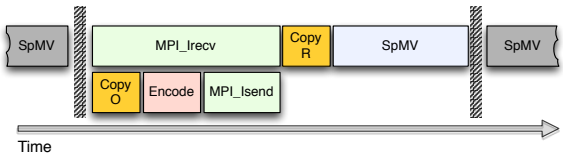
When operating on multiple GPUs we avoid the copy-out or decode phase by requiring that the local ordering of nodes sort the set $\mathcal{R}$ by the rank of the process sending each node. This

way, when the MPI collective finishes and all values arrive contiguous by provider, the data can be copied directly to the GPU without reordering.

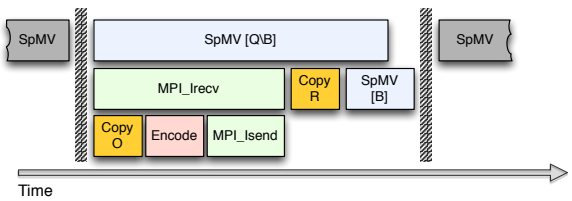
7.1 Overlapping with the GPU

Overlapping communication and GPU computation is enabled by two levels of asynchronous commands:

- *Non-blocking MPI.* The MPI_Isend/MPI_Irecv routines are used to post communication and immediately return the main thread to computation.
- *Asynchronous GPU Queues.* Two OpenCL queues are utilized to send commands to the GPU.



(a) GPU and MPI_Isend/MPI_Irecv (Non-Overlapping)



(b) GPU and MPI_Isend/MPI_Irecv (Overlapping)

Figure 7.1: Overlapping GPU kernels with CPU managed communication. Task block sizes are not drawn to scale.

7.2 Distributed GPU Performance

7.2.1 Non-Overlapping Multi-GPU Results

See Appendices C and **??** for scaling benchmarks from the non-ViennaCL multi-GPU implementation. Appendix C demonstrates the scalability of an iteration of RK4 as applied in § 8.2 on the Keeneland GPU cluster—the benchmark assumes three SpMV operations per intermediate evaluation of the RK4 time-step. Appendix **??** is similar, but considers only two SpMV operations per evaluation corresponding to the test case in § 8.1. In both cases, the performance is based on a non-overlapping communication collective (MPI_Alltoallv).

Benchmarks

7.2.2 Overlapping Results

We scale the SpMV across the GPUs on Cascade.

By comparing benchmarks from the non-overlapped and overlapped GPU cases, we get the speedup:

$$S_p = {}^{t_{non-overlapped}}/_{t_{overlapped}}$$

in using the overlapped solution. Any reduction in time is evidence of overlap.

If speedup greater than $p = (ComputeNodes * PPN)$ is achieved

The total time required to compute the distributed SpMV for 1 node and 4 PPN is observed in the neighborhood of 80 ms for $n = 50$ and the non-overlapping algorithm (i.e. without splitting $\mathcal{Q}\backslash\mathcal{B}$ and $\mathcal{B}$). Of those 80 ms, approximately 60 ms is spent performing non-computation tasks such as transferring from GPU down to CPU, encoding and sending data via MPI and then transferring back up to the GPU. The remaining 20 ms is all dedicated to performing the SpMV.

By introducing the overlapped GPU kernels, the total time for the distributed SpMV for $n = 50$ (1 node, 4 PPN) drops to approximately 15 ms—25% less than the SpMV-only portion of the non-overlapped approach. Since the non-computation tasks amount to only 12 ms in the overlapped case, our first conclusion is that overlapping hides 100% of communication for the cases tested in Table 7.1, as well as the entire SpMV for $\mathcal{B}$.

Within each block of Table 7.1 we expect the GFLOP/sec to double by row. Obviously the strong scaling in the case of the GPU is not ideal when executing more than one PPN. However, for

Table 7.1: GFLOP/sec achieved by the distributed ELL SpMV on Cascade's M2070 GPUs, with MPI communication overlapping two GPU kernels. The SpMV computes derivatives over a 3-D regular grid of size $N = 4096000$ nodes (i.e., 160^3). Data reflects various combinations of stencil sizes, number of compute nodes and number of processes per node (PPN). Quantities in parentheses denote the speedup factors achieved by the overlapping algorithm over the non-overlapping approach for identical combinations of compute nodes, PPN, stencil size, etc.

		Observed GFLOP/sec (Speedup over Non-overlapped)			
Compute Nodes	PPN	n=17	n=31	n=50	n=101
1	1	8.5 (1.0x)	8.4 (1.0x)	9.0 (1.0x)	–
2	1	13.8 (2.3x)	13.0 (1.9x)	13.1 (2.0x)	13.5 (1.8x)
4	1	13.1 (1.9x)	25.1 (2.8x)	24.6 (2.4x)	25.2 (1.9x)
8	1	24.5 (2.0x)	33.2 (2.3x)	41.2 (3.4x)	53.6 (2.3x)
1	2	11.3 (3.4x)	12.2 (3.0x)	12.1 (3.2x)	12.7 (3.0x)
2	2	13.0 (3.0x)	22.9 (4.2x)	23.1 (3.6x)	24.5 (3.0x)
4	2	25.1 (2.3x)	37.8 (4.1x)	50.1 (5.0x)	53.5 (3.8x)
8	2	35.8 (2.2x)	38.3 (2.5x)	59.6 (2.7x)	87.6 (3.7x)
1	4	14.1 (**4.3x**)	22.5 (**5.1x**)	24.4 (**5.4x**)	24.4 (**4.4x**)
2	4	19.6 (2.4x)	32.0 (4.2x)	37.2 (3.9x)	50.8 (4.6x)
4	4	27.5 (2.2x)	38.6 (3.0x)	57.0 (3.0x)	81.3 (4.5x)
8	4	50.9 (2.9x)	61.6 (2.2x)	88.8 (3.2x)	130.8 (3.5x)

$n = 50$ and 1 PPN we see super but on only 32 GPUs, and impeded by the I/O hub bottleneck, we manage to achieve 130 GFLOP/sec for $n = 101$. Compare this to the GFLOP/sec scaling results on Itasca in Figure 6.15b where 130 GFLOP/sec requires approximately

Greater than 4x speedup (bold-faced in Table 7.1) is possible due to a design flaw on Cascade: each compute node has four attached GPUs connected to the host via a shared I/O hub. Concurrent data transfers from multiple GPUs to host, or vice versa—as would occur while executing a synchronized, distributed SpMV—result in contention on the hub and serialization of transfers. Serialization effectively quadruples the total run-time for 4 PPN. By overlapping communication and computation we notice two effects: a) contention disappears (or is at least hidden by computation), and b)

7.3 Multiple Kernel Scheduling

describe fermi's ability to schedule multiple kernels, what it means for our queues.

7.4 Future Work

One of the problems with choosing to work in OpenCL is the fact that the standard offers the lowest common denominator of features from the various hardware vendors that support it. Many vendor specific features never make it into the language.

Take for example GPUDirect, a technology introduced first CUDA v3.1 for NVidia hardware. GPUDirect allows direct access to GPU memory addresses from various sources including other GPUs. The technology allows GPUs to bypass copies to host memory en-route to another GPU on the same compute node. Combine GPUDirect with the new MPI aware features in CUDA v5.0 and data can pass directly from a GPU onto the infiniband fabric and up to another GPU [101]. This type of feature may never be available in OpenCL.

The new Kepler K20s allow both concurrent kernel execution and dynamic parallelism. Dynamic parallelism is a means of scheduling new kernel launches from within an exist kernel. In order to support this feature,

K20s also support CUDA 5.5 which introduces MPI aware CUDA. The nvcc compiler now detects MPI calls and routes data movement directly from InfiniBand to the GPU memory rather than making a stop on the host memory. This is only possible with GPUDirect (direct memory addressing) and dynamic parallelism (to spawn an MPI process from a kernel).

Features like MPI-CUDA will are unlikely to be available in OpenCL in the future. NVidia is no longer leading or distributing the OpenCL libraries with their driver. NVidia only plans to support the OpenCL spec v1.1.

CHAPTER 8

NUMERICAL VALIDATION

Here, we present the first results in the literature for parallelizing RBF-FDs on multi-CPU and multi-GPU architectures for solving PDEs. A less verbose version of this chapter was included in [21].

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 [113, 114] and 2) solid body rotation [88]. 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 [50, 52, 57, 62]. 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 [139] for details) in order to be consistent with previously published results (see e.g., [52] and [59]). 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 [130].

We assume the spherical coordinate system is used in terms of latitude λ and longitude θ:

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

8.1 Vortex Rollup

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

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

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

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 Equation 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

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

where $\rho(\theta) = \rho_0 \cos\theta$ is the radial distance of the vortex with $\rho_0 = 3$. The exact solution to Equation 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 Equation 8.1 is

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

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 Equation 8.2 in the form:

$$\begin{aligned} \frac{d\mathbf{h}}{dt} &= -\text{diag}(\omega(\theta))D_\lambda \mathbf{h} + \gamma_c N^{-k} D_{\Delta^k}\mathbf{h}, \\ &= \left(-\text{diag}(\omega(\theta))D_\lambda + \gamma_c N^{-k} D_{\Delta^k}\right)\mathbf{h}, \end{aligned} \tag{8.3}$$

where D_{Δ^k} is the differentiation matrix for the k-th power Laplacian, Δ^k, described in Section 3.3.6, and the expression between parentheses in Equation 8.3 is the stabilized DM applied at each time-step.

The scaling parameter γ_c and the order of hyperviscosity k are given in Table 8.1. The parameters are tuned for the MD node sets. Since the values of γ_c and k vary based on the eigenvalues/conditioning of D_λ, switching to an icosahedral or spherical CVT requires additional tuning. Changes to the stencil size or the function for ϵ also require tuning.

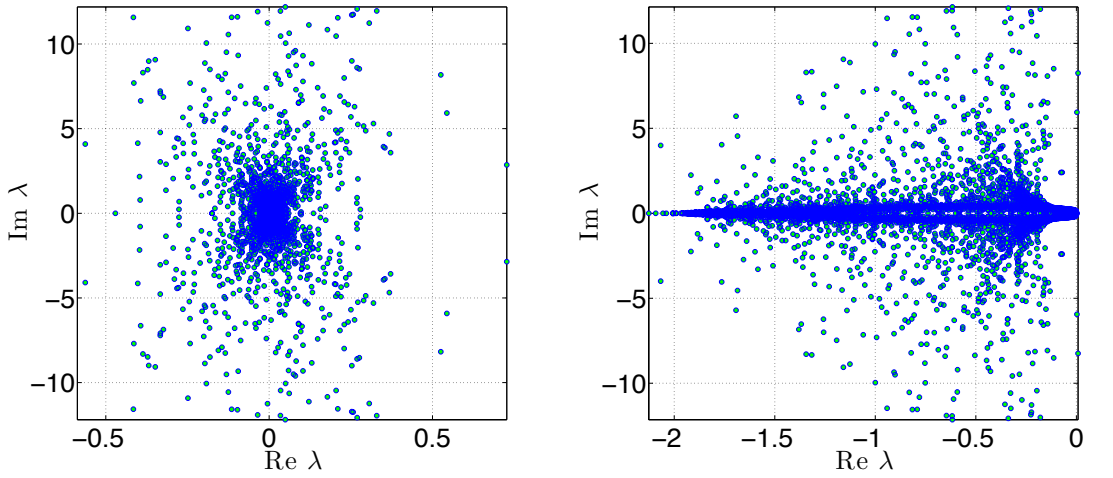
Table 8.1: Values for hyperviscosity and the RBF shape parameter ϵ for vortex roll-up test.

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

The parameters γ_c and k are 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 first choose k to damp the higher spurious eigenmodes of $\text{diag}(\omega(\theta))D_\lambda$ while leaving the lower physical modes that can be resolved by the stencil intact. This effectively pushes eigenvalues to the left half of the complex plane. We then try to adjust γ_c to get eigenvalues condensed as near to the imaginary axis as possible. As a rule of thumb we try to keep the range of filtered real parts within twice the width of the unfiltered range, so hyperviscosity does not cause excessive diffusion in the solution. Figure 8.1 shows the effect of hyperviscosity on the eigenvalues of the DM, $-diag(\omega(\theta))D_\lambda$, from no hyperviscosity in Equation 8.2 (Figure 8.1a) to the hyperviscosity modified DM in Equation 8.3 (Figure 8.1b).

The RBF shape parameter, ϵ, is expressed as a function of N based on work in [51]. If we let $(\kappa_A)_j$ be the condition number of the interpolation matrix in Equation 3.3 for the j^{th} stencil, then we seek an ϵ 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 grows. For a constant mean condition number, ϵ varies linearly with $\sqrt{N}$ (see [51] Figure 4a and b, and the examples in Section 3.6). 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 [51].

Figure 8.2 shows the initial condition for Equation 8.1 (Figure 8.2a) and the solution at $t = 10$ (Figure 8.2b), on $N = 10201$ nodes, with stencil size $n = 50$. This resolution is sufficient to properly capture the vortices at $t = 10$ (Figure 8.2c), but lower resolutions would suffer approximation errors associated with insufficient grid resolution. For this reason, the solution at $t = 3$ is considered in



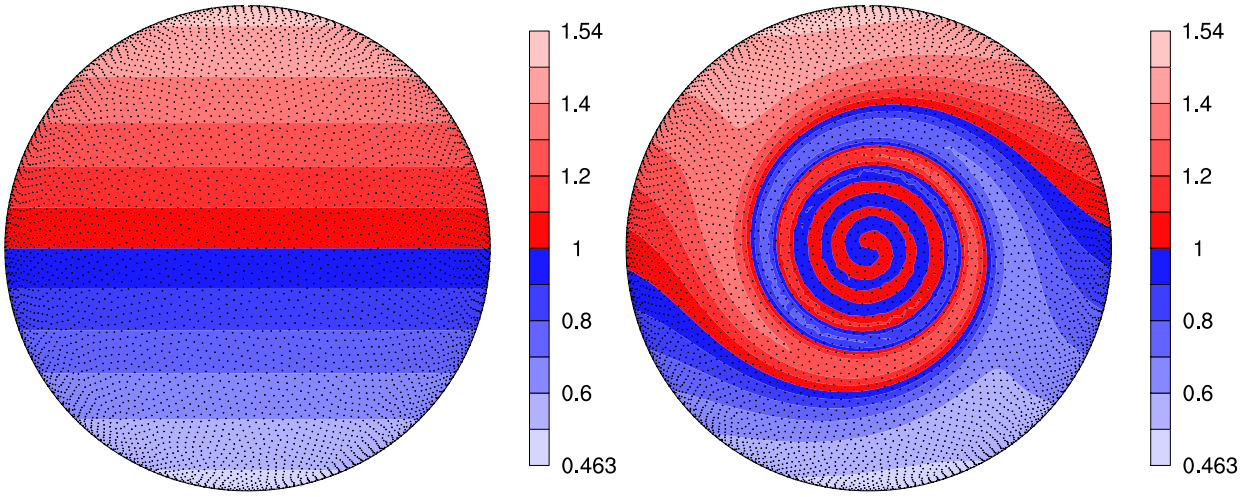
(a) No Hypervsiscosity

(b) With Hyperviscosity ($k = 4$ and $\gamma_c = 40$)

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 $\epsilon = 3.5$.

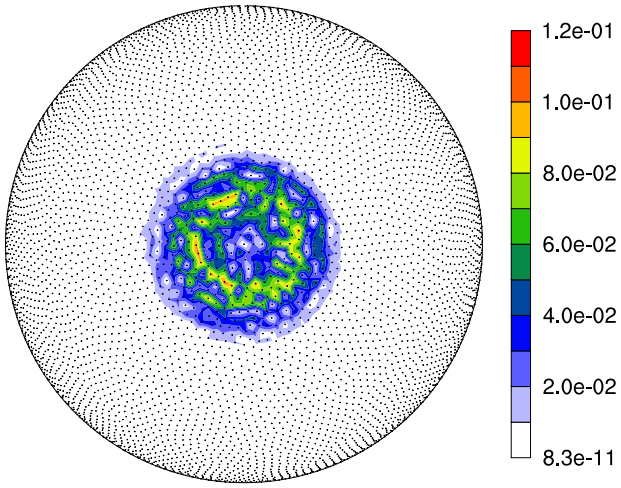
the normalized ℓ_2 error convergence study presented in Figure 8.3. The time step $\Delta t = 0.05$ for all resolutions.

Based on Figure 8.3a, which shows the convergence without the hyperviscosity filter on eigenvalues, we see that only one test case reflects the unstable eigenvalues present in the DM. In a stroke of luck the unstable modes are small enough that the solution at $t = 3$ is not impacted, but as t increases the instabilities begin to dominate. Figure 8.3b demonstrates the convergence of vortex roll-up with hyperviscosity included. The solution errors are slightly higher than Figure 8.3b, but the instabilities have vanished.



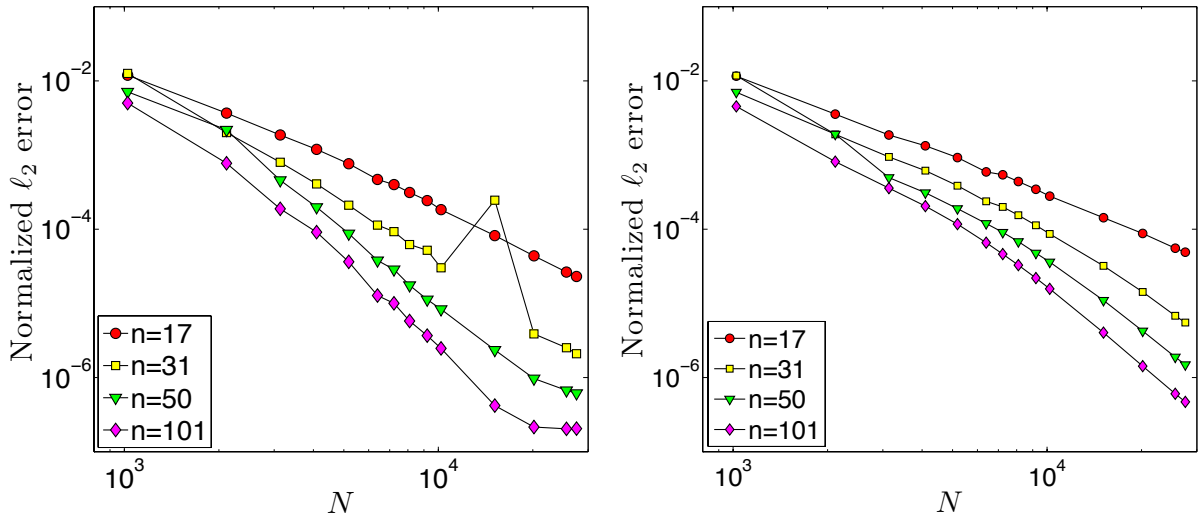
(a) Initial Condition

(b) Computed Solution ($t = 10$)



(c) Relative ℓ_2 Error

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 $1.25(10^{-2})$.



(a) No Hyperviscosity

(b) With Hyperviscosity

Figure 8.3: Convergence plot for vortex roll-up at $t = 3$.

8.2 Solid body rotation

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

$$\frac{\partial h}{\partial t} + \frac{u}{\cos\theta}\frac{\partial h}{\partial \lambda} + v\frac{\partial h}{\partial \theta} = 0, \tag{8.4}$$

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 [52, 113].

The discretized form of Equation 8.4 is

$$\frac{d\mathbf{h}}{dt} = -\text{diag}\left(\frac{u}{\cos\theta}\right) D_\lambda \mathbf{h} - \text{diag}(v) D_\theta \mathbf{h} \tag{8.5}$$

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 Equation 8.5 into one operator, our implementation evaluates them as two sparse matrix-vector multiplies. The separate operations 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 ($1/\cos\theta \to \infty$ as $\theta \to \pm\frac{\pi}{2}$) in Equation 8.4 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 [52, 59], applying the entire spatial operator to the right hand side of Equation 3.3 generates a single DM that analytically removes the singularities at poles (see Appendix A for details).

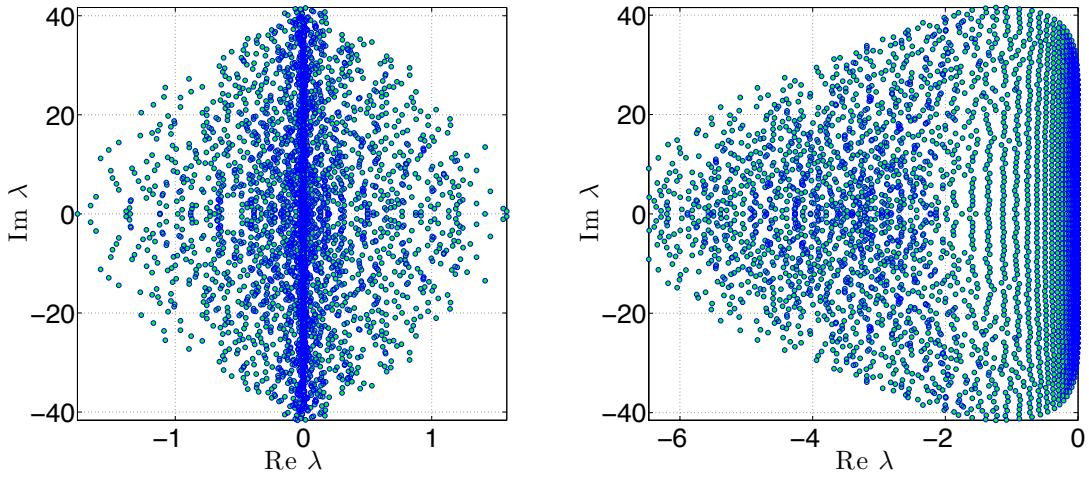
We will advect a C^1 cosine bell height-field given by

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

having a maximum height of $h_0 = 1$, a radius $R = \frac{1}{3}$ and centered at $(\lambda_c, \theta_c) = (3\pi/2, 0)$, with $\rho = \arccos(\sin\theta_c\sin\theta + \cos\theta_c\cos\theta\cos(\lambda - \lambda_c))$. The angle of rotation, $\alpha = \pi/2$, 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	$5 * 10^{-2}$



(a) No Hyperviscosity

(b) With Hyperviscosity ($k = 8$ and $\gamma_c = 5 * 10^{-2}$)

Figure 8.4: Eigenvalues of Equation 8.5 for the cosine bell test case with $N = 4096$ nodes, stencil size $n = 101$, and $\epsilon = 3.5$. Eigenvalues are divided by u_0 to remove scaling effects of velocity.

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. In this case, the eigenvalues are significantly more sensitive to hyperviscosity and require fine tuning of γ_c. Figure 8.5 demonstrates the impact on eigenvalues from Figure 8.4 when γ_c is increased from $5 * 10^{-2}$ to $5 * 10^{-1}$. The higher γ_c excessively damps modes, effectively wiping out those that are needed for the solution.

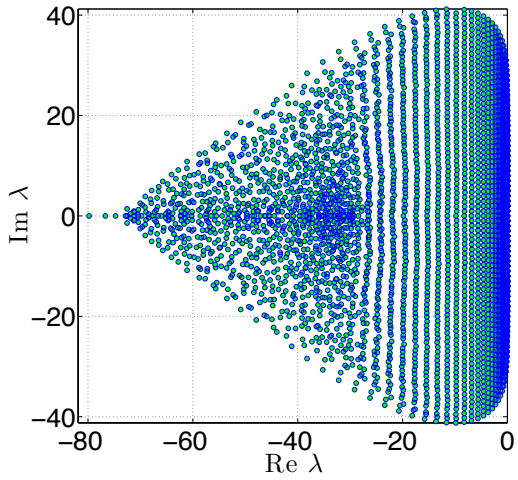
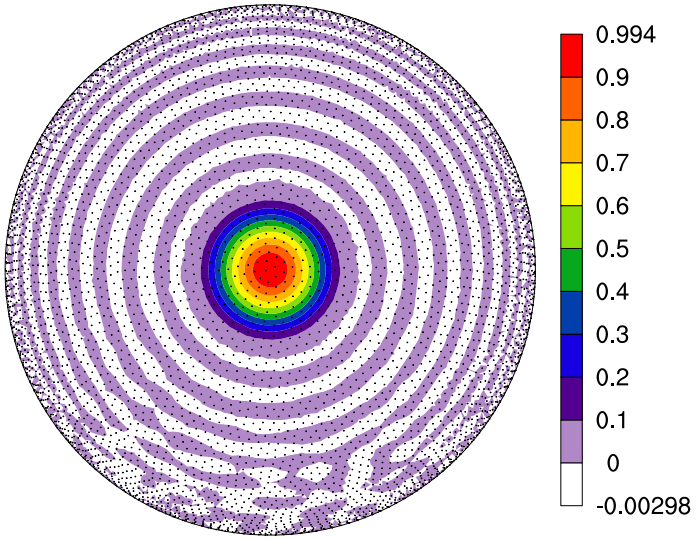


Figure 8.5: Example of excessive hyperviscosity with parameters $k = 8$ and $\gamma_c = 5 * 10^{-1}$.

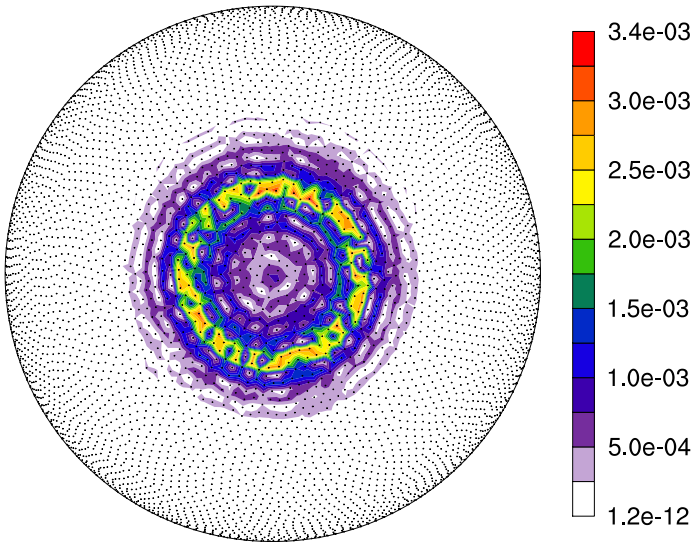
Figure 8.6 shows the cosine bell transported ten full revolutions around the sphere. Without hyperviscosity, RBF-FD cannot complete a single revolution of the bell before instability takes over. However, adding hyperviscosity allows computation to extend to dozens or even thousands of revolutions and maintain stability (e.g., see [59]).

After ten revolutions, the cosine bell is still intact with only 0.6% lost from the original peak ($h_0 = 1$). The white and purple rippling effect shown in Figure 8.6a is not severe. The absolute error in Figure 8.6b clarifies that the majority of the absolute error appears at the base of the C^1 bell where the discontinuity appears in the derivative. Beyond the region at the base of the bell, the absolute error drops well below 10^{-5}.

Figure 8.7 illustrates the convergence of the RBF-FD method for the cosine bell advection at 10 full revolutions. All tests in Figure 8.7 assume 1000 time-steps per revolution (i.e., $\Delta t = 1036.8$ seconds). We find that 1000 time-steps is a conservative estimate to stably advect the bell even up to the highest resolution MD node set, $N = 27556$. Our choice of Δt is half as large as the 30 minute step taken for $N = 4096$ nodes in [52], but still twice as large as the competing Discontinuous Galerkin implementation and 8x larger than both Spherical Harmonics and Double Fourier methods also considered therein. For $N = 27556$ and $n = 101$ a minimum of 650 time-steps are required per revolution.



(a) 10 Revolutions



(b) Absolute Error at 10 Revolutions

Figure 8.6: Cosine bell solution after 10 revolutions with $N = 10201$ nodes and stencil size $n = 101$. Hyperviscosity parameters are $k = 8$, $\gamma_c = 5(10^{-2})$.

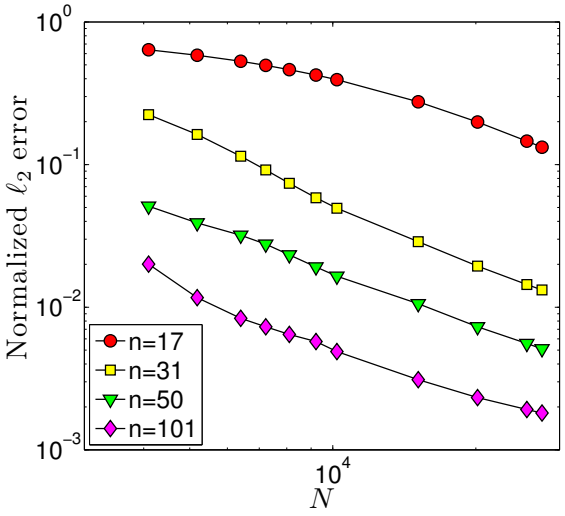


Figure 8.7: Convergence plot for cosine bell advection. Normalized ℓ_2 error at 10 revolutions with hyperviscosity enabled.

CHAPTER 9

STOKES (INCOMPLETE)

This whole chapter is incomplete work. We started working toward stoke solutions on the unit sphere but it has yet to be completed.

9.1 Introduction

We consider herein the solution to steady-state viscous Stokes flow on the surface of a sphere, governed by:

$$\boldsymbol{\nabla} \cdot [\eta(\boldsymbol{\nabla}\mathbf{u} + (\boldsymbol{\nabla}\mathbf{u})^T)] + RaT\hat{r} = \boldsymbol{\nabla} p \tag{9.1}$$

$$\boldsymbol{\nabla} \cdot \mathbf{u} = 0 \tag{9.2}$$

where the unknowns $\mathbf{u}$ and p represent the vector velocity- and scalar pressure-field respectively, η is the viscosity tensor, Ra is the non-dimensional Rayleigh number, and T is an initial temperature profile. Many practical applications in sciences such as geophysics, climate modeling, and computational fluid dynamics must solve variations of the Navier-Stokes equations. The focus of this paper is on the implicit solve component for viscous (Stokes) flow, which amounts to the steady-state problem described by Equations 9.1 and 9.2.

This article introduces the first (to our knowledge) parallel approach to solve the steady-state equations on the surface of the unit sphere with the Radial Basis Function-generated Finite Differences (RBF-FD) method. Building on our work in [21], which parallelized explicit RBF-FD advection, our goal is to integrate both explicit and implicit components within a larger transient flow model.

For decades, the demand for fast and accurate numerical solutions in fluid flow has lead to a plethora of computational methods for various geometries, discretizations and dimensions. On the sphere in $\mathbb{R}^3$ popular discretizations include the standard latitude-longitude grid, cubed-sphere [113], Ying-Yang overlapping grid [91], icosahedral grid [124] and centroidal voronoi tessellations

[41]. Associate with each discretization is a mesh—specific to the choice of numerical method—that indicates connectivity of nodes for differentiation.

Until now, most of the focus in RBF-FD has been on explicit methods. However, many practical applications in sciences such as geophysics, climate modeling, and computational fluid dynamics must solve variations of the Navier-Stokes equations, and depend on an implicit solve component. This paper develops multi-GPU algorithms for implicit RBF-FD systems toward the goal of integration within transient flow problems.

Our goal is to demonstrate to the geosciences that RBF-FD can function well on both hyperbolic and elliptic problems. In [21] we introduced a multi-GPU implementation of RBF-FD and demonstrated the method's strong ability to stably advect solid bodies on the sphere. In this paper we continue toward the goal of RBF-FD solutions for fluid flow problems with a multi-GPU Poisson solver for steady-state Stokes flow. In this context, speed is not a paramount issue.

Related work on RBF methods targeting the GPU is quite limited. Schmidt et al. [133] solve an implicit system for a global RBF method using Accelerys Jacket in Matlab. Our work in [21] introduced the first parallel implementation of RBF-FD for explicit advection capable of spanning multiple CPUs as well as multiple GPUs.

Related work on multi-CPU or multi-GPU RBFs

- CPU [173] [161]
- single-GPU [133]
- multi-GPU
  - Preconditioned BiCGStab for Navier Stokes, Finite Element method [68]

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

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

Parallel GMRES

- CPU only: PETSc [173], Hypre [161]
- Parallel GMRES on single GPU available in ViennaCL [125] and CUSP [12]
- Parallel GMRES on Multiple GPUs [7]
- Reduced Communication with increased computation [38]

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

9.2 Multi-GPU GMRES

Our implementation leverages ViennaCL to directly benefit from improvements to the performance of underlying sparse matrix-vector product, vector dot vector and other linear algebra primitives. Also, ViennaCL provides seamless interoperability with the Boost::UBLAS, EIGEN and MTL libraries via C++ templates. We test the performance of our algorithm on one or more CPUs with the Boost::UBLAS library.

The GMRES algorithm was introduced in 1986 by Saad and Schultz [128]. The iterative solver support general matrix structures, whereas methods like Conjugate Gradient require symmetric positive definiteness.

At the core of the GMRES algorithm is an Arnoldi (orthogonalization) process.

Variants of GMRES exist that utilize unique orthogonalization steps. The motivation behind alternative Arnoldi processes is to save both memory and operation counts. Saad [128] introduced a practical implementation of the GMRES method based on Given's rotations to compute an implicit QR factorization. The Given's based algorithm is part of libraries like CUSP [12] and CULA Sparse [83]; ViennaCL implements the Householder reflection algorithm.

The Given's rotation algorithm is easier to parallelize than Householder, which led to the addition of an alternate GMRES algorithm with restarts under ViennaCL.

The authors of [7] do not describe their orthogonalization process, but based on the description of their parallelization strategy it is safe to assume a Given's rotation is used.

Algorithm 9.1 presents the basic Left-preconditioned GMRES algorithm with restarts and Given's rotations. The comments in blue denote portions of the algorithm that depend on MPI collectives and the MPI routine used .

Algorithm 9.1 Left-preconditioned GMRES(k) with Given's Rotations

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

Note that the application of a preconditioner such as ILU0 introduces an additional call to MPI_Alltoallv before everywhere M^{-1} is present in Algorithm 9.1.

Spanning GMRES across multiple GPUs uses the same collectives as those developed in Chapter 6.

9.2.1 Interleaved Solution

In a distributed environment values for each U, V, W and P must be obtained from ghost nodes. Rather than execute four collectives (one per component), we interleave components to group all components by node. Figure 9.1 demonstrates the concept of interleaving the solution. This allows us to directly copy a double4 containing the $\{u_i, v_i, w_i, p_i\}$ for node x_i

The DM assembly depends on all dimensions of solution values for a single node to be consecutive

in memory. While Equation 9.5 has all components of U, V, W and P grouped together, the values of u_1, v_1, w_1 and p_1 correspond to node (x_1, y_1, z_1).

Figure 9.1 demonstrates the effect of interleaving our solution. The sparsity pattern of the original DM with solutions grouped by component is shown in Figure 9.1a. Well defined blocks of non-zeros are filled with RBF-FD weights from Equation 9.3. Figure 9.1b presents the sparsity pattern for interleaved solution components. The pattern is similar to a single block of Figure 9.1a, but the sub-matrix $(10:50) \times (10:50)$ of each solution ordering, shown in Figures 9.1c and 9.1d, illustrate that non-zeros in Figure 9.1b are small 4×4 blocks with the structure of Equation 9.3.

The distributed GMRES implementation depends on two MPI collectives: MPI_Alltoallv and MPI_Allreduce.

We solve the PDE on the surface of the sphere as an example for one and multiple GPUs.

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

$$\begin{pmatrix} -\eta\nabla^2 & 0 & 0 & \frac{\partial}{\partial x_1} \\ 0 & -\eta\nabla^2 & 0 & \frac{\partial}{\partial x_2} \\ 0 & 0 & -\eta\nabla^2 & \frac{\partial}{\partial x_3} \\ \frac{\partial}{\partial x_1} & \frac{\partial}{\partial x_2} & \frac{\partial}{\partial x_3} & 0 \end{pmatrix} \begin{pmatrix} u_1 \\ u_2 \\ u_3 \\ p \end{pmatrix} = \frac{RaT}{\sqrt{x_1^2 + x_2^2 + x_3^2}} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ 0 \end{pmatrix}. \tag{9.3}$$

where we recall from Chapter 3 that the ∇^2 operator in spherical polar coordinates for $\mathbb{R}^3$ is:

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

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

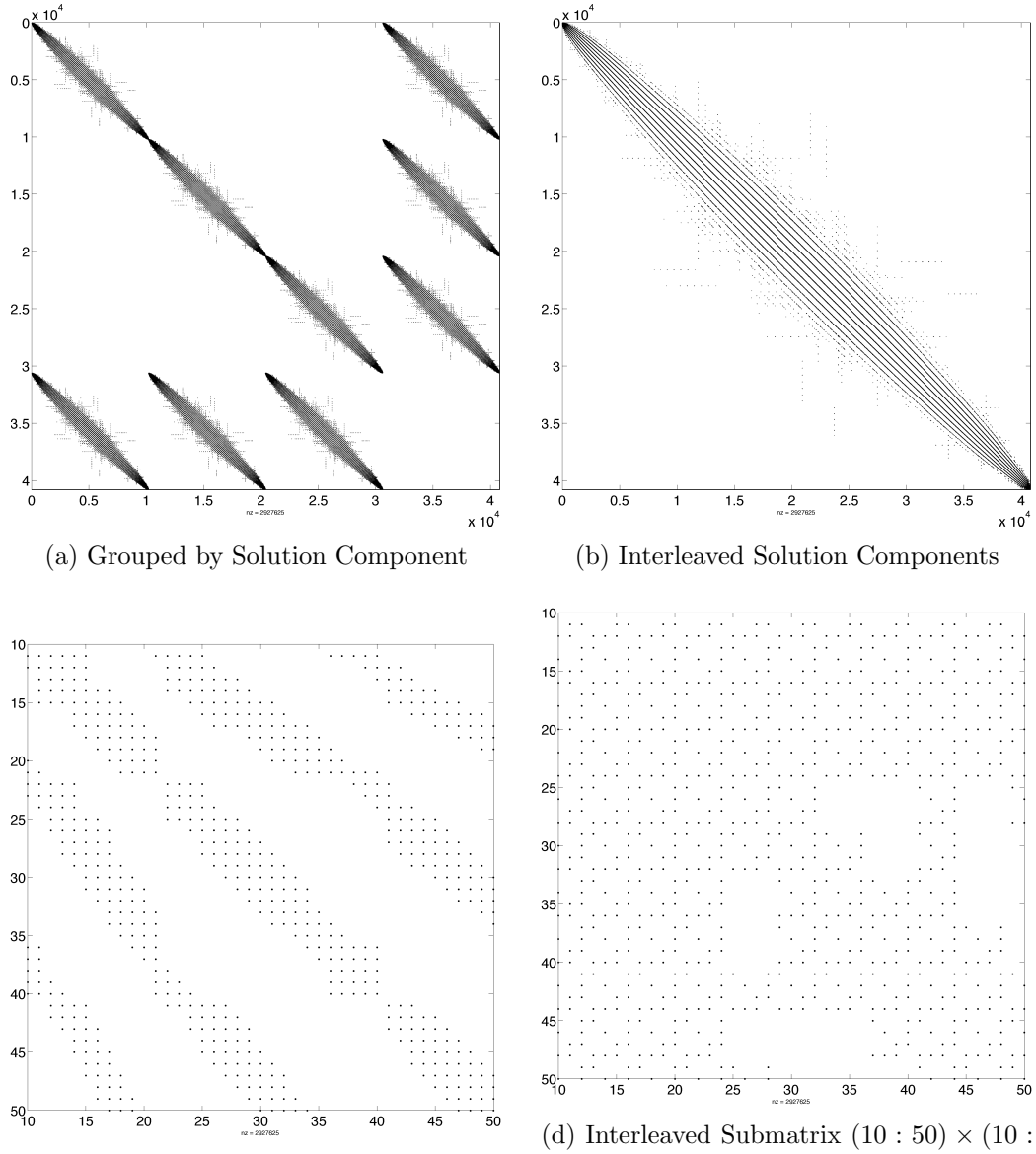
On the surface of the unit sphere the radial term vanishes, so we are left with:

$$\nabla^2 \equiv \Delta_S.$$

The RBF operator from [168]—Equation (20)—is applied to the RHS of Equation 3.3 to generate Laplace-Beltrami RBF-FD weights:

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

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



(a) Grouped by Solution Component

(b) Interleaved Solution Components

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

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

Figure 9.1: Sparsity pattern of linear system in Equation 9.3. Solution values are either ordered by component (e.g., $(u_1, \cdots, u_N, v_1, \cdots, v_N, w_1, \cdots, w_N, p_1, \cdots, p_N)^T$) or interleaved (e.g., $(u_1, v_1, w_1, p_1, \cdots, u_N, v_N, w_N, p_N)^T$).

Additionally following [51, 53], the off-diagonal blocks in Equation 9.3 must be constrained to the sphere via the projection matrix:

$$P = I - \mathbf{x}\mathbf{x}^T = \begin{pmatrix} (1-x_1^2) & -x_1x_2 & -x_1x_3 \\ -x_1x_2 & (1-x_2^2) & -x_2x_3 \\ -x_1x_3 & -x_2x_3 & (1-x_3^2) \end{pmatrix} = \begin{pmatrix} P_{x_1} \\ P_{x_2} \\ P_{x_3} \end{pmatrix} \tag{9.4}$$

where $\mathbf{x}$ is the unit normal at the stencil center, and [51, 53] show that with a little manipulation, the weights can be directly computed with these operators:

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

9.2.2 Constraints

Due to the lack of boundary conditions on the sphere, the family of solutions that satisfy the PDE in Equation 9.3 includes four free constants (one for each u_1, u_2, u_3 and p).

One way to close the null-space of the solution is to augment Equation 9.3 with the following constraints:

$$\left(\begin{array}{cccc:cccc} -\eta\nabla^2 & 0 & 0 & \frac{\partial}{\partial x_1} & 1_{N\times 1} & 0 & 0 & 0 \\ 0 & -\eta\nabla^2 & 0 & \frac{\partial}{\partial x_2} & 0 & 1_{N\times 1} & 0 & 0 \\ 0 & 0 & -\eta\nabla^2 & \frac{\partial}{\partial x_3} & 0 & 0 & 1_{N\times 1} & 0 \\ \frac{\partial}{\partial x_1} & \frac{\partial}{\partial x_2} & \frac{\partial}{\partial x_3} & 0 & 0 & 0 & 0 & 1_{N\times 1} \\ \hdashline 1_{1\times N} & 0 & 0 & 0 & \multicolumn{4}{c}{\multirow{4}{*}{$0_{4\times 4}$}} \\ 0 & 1_{1\times N} & 0 & 0 & & & & \\ 0 & 0 & 1_{1\times N} & 0 & & & & \\ 0 & 0 & 0 & 1_{1\times N} & & & & \end{array}\right)\left(\begin{array}{c} u_1 \\ u_2 \\ u_3 \\ p \\ \hdashline c_1 \\ c_2 \\ c_3 \\ c_4 \end{array}\right)$$
$$= \frac{RaT}{\sqrt{x_1^2+x_2^2+x_3^2}}\left(\begin{array}{c} x_1 \\ x_2 \\ x_3 \\ 0 \\ \hdashline \int_\Omega u_1\partial\Omega \\ \int_\Omega u_2\partial\Omega \\ \int_\Omega u_3\partial\Omega \\ \int_\Omega p\partial\Omega \end{array}\right). \quad (9.5)$$

where the subscript on $1_{N\times 1}$ indicates a $N\times 1$ vector of ones. These constraints add the unknowns (c_1, c_2, c_3, c_4), which should solve to be zero. The four rows on the bottom require that the solution satisfy the integral over the domain for each solution component. In combination with the four added columns, the constraints indicate that the solution components must satisfy integrals using the same constant value. This is only possible if the constants are zero. The added constraints are not physically significant, but are chosen to satisfy the system algebraicly. Of course when solving Equation 9.5 with GMRES, the constraints help increase the rate of convergence.

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

9.2.3 Manufactured Solution

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

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

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

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

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

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

Typically, on the surface of the sphere, the projection operator from Equation 9.4 must be applied to an arbitrary $g(\mathbf{x})$. Here, we choose the components of $g(\mathbf{x})$ to be various spherical harmonics in Cartesian coordinates. Spherical harmonics nicely satisfy $P_x Y_l^m = Y_l^m$. Consequently, the entire application of P_x is neglected.

We select $g(x)$ to be:

$$g(x) = 8Y_3^2 - 3Y_{10}^5 + Y_{20}^{20} \tag{9.6}$$

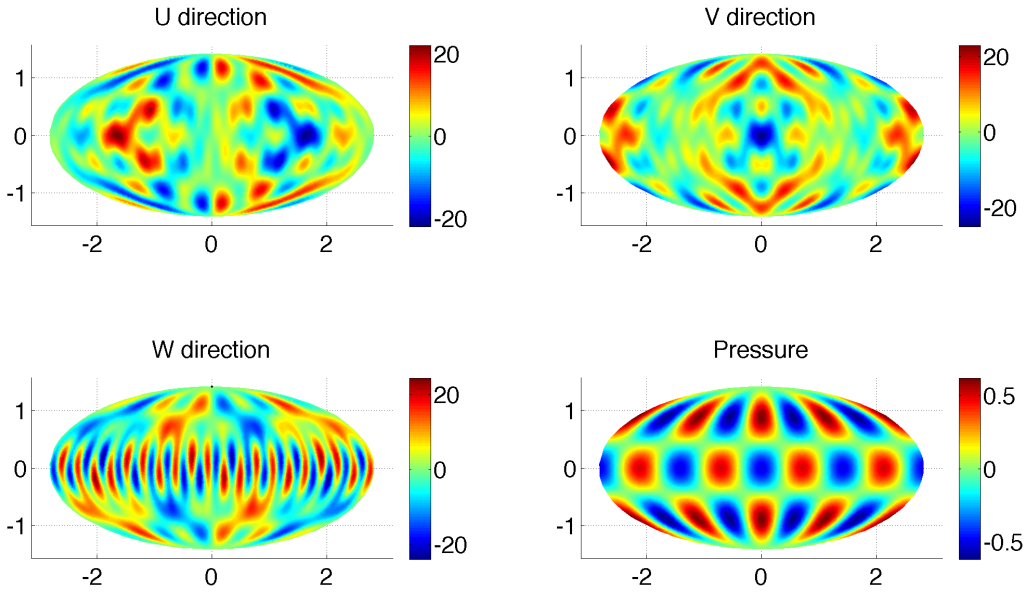
and the pressure function:

$$P = Y_6^4 \tag{9.7}$$

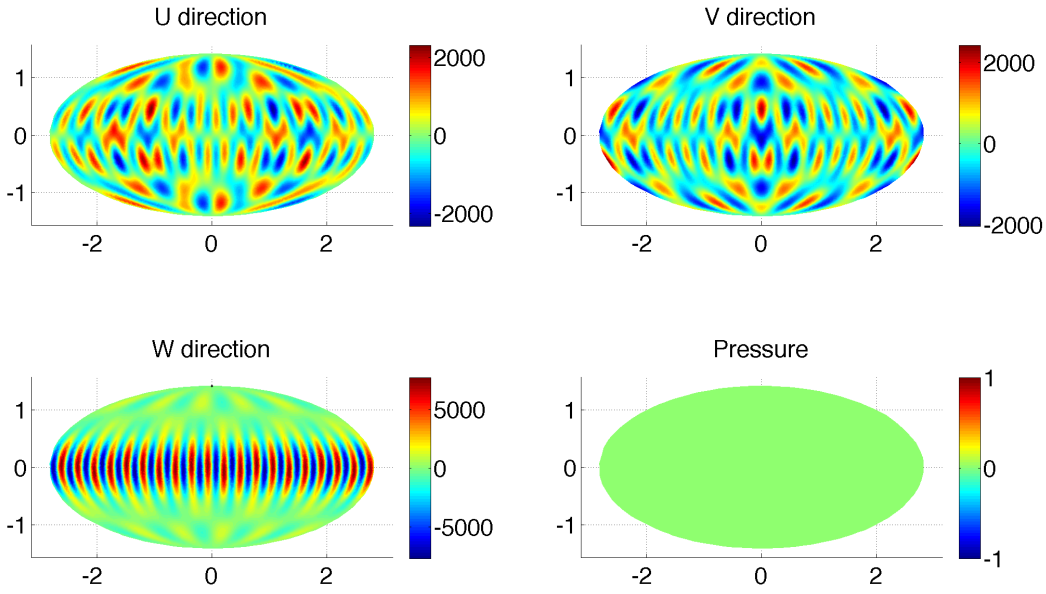
The manufactured solution is shown in Figure 9.2. A Mollweide projection given as:

$$x = \frac{2\sqrt{2}}{\pi}\lambda \cos\theta$$
$$y = \sqrt{2}\sin\theta$$

maps the sphere onto the plane.



(a) Solution $Q_x(g(x))$ with $g(x) = 8Y_3^2 - 3Y_{10}^5 + Y_{20}^{20}$



(b) Right Hand Side (Constant Viscosity)

Figure 9.2: A divergence free field is manufactured for the sphere.

Convergence. As the problem size N increases, we expect the approximation to the solution to converge on the order of $\sqrt{n}$ where n is the choice of stencil size.

Author's Note: Finish convergence plot.

9.3 Preconditioning

GMRES is slow to converge when used without a preconditioner.

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

Our tests show that an incomplete LU factorization with zero fill-in [127] functions well.

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

9.4 GMRES Results

CHAPTER 10

FINAL DISCUSSION, CONCLUSIONS AND FUTURE WORK

Chapter 6 introduced the design and tuning of the first distributed RBF-FD implementation for CPU and GPU clusters. The implementation achieves 30% to 40% parallel efficiency on 128 processors of the Itasca HPC cluster (CPU-only) at the University of Minnesota when given a problem size of $N = 4096000$ nodes and stencil sizes between $n = 17$ and $n = 50$.

Our implementation leverages METIS for general domain decomposition and load balancing. Individual processes do not require a full mapping between RBF nodes and CPUs. Instead, processes assemble and operate on local linear systems for their subdomains. Communication between processes is limited to nearest-neighbor or all-to-subset collectives based on the intersection of subdomains.

A number of tuning steps reduce communication times and increase scalability beyond the results in our first paper, [21]. Starting with an MPI_Alltoallv collective, the tuning effort ultimately leads to a scheme which overlaps communication and computation on the CPU. Chapter 7 extends this collective to overlap with computation on GPUs.

Within this document readers should expect to find the following major contributions:

1. The first ever application of a fixed-grid neighbor query algorithm (popularized by distantly related particle methods) to generate stencils for RBF-FD.
    - Performance comparison with the RBF community favorite, k-D Tree, shows that the lower complexity fixed-grid method is able to achieve up to 2.5x speedup over the former method on randomly distributed nodes.
    - The fixed-grid method is also shown to accelerate RBF-FD time-step performance by up to 5x due spatial locality of node information in memory and improved cache effects.
2. Integer dilation and bit interleaving are introduced to the RBF community at large to construct a number of space-filling curve variants in 2-D and 3-D. The curves reorder cells of the fixed-grid algorithm on the prospect of additional cache hits for derivative calculations.

- A comparison of the variants to the result of the well-known Reverse Cuthill-McKee (RCM) algorithm concludes that RCM remains the better option for reordering and matrix bandwidth minimization.

3. The design and tuning of the only known implementation of the RBF-FD method to scale across multiple compute nodes of a supercomputing cluster.

    - Our home-grown implementation divides computation with a Restricted Additive Schwarz (RAS) domain decomposition in a first-of-its-kind application to RBF-FD.
    - As part of the tuning process described herein, balanced workloads are achieved through the use of the METIS graph partitioning algorithm [94].
    - A number of steps are taken to improve communication collectives in the CPU-only implementation. The resulting algorithm splits derivative calculation into two steps and overlaps communication with computation. We observe that up to 80% of the cost in communication can be hidden in some cases.
    - Scaling benchmarks up to 1024 processes (divided into 8 processes per node) and a grid resolution of $N = 160^3$ vertices (i.e., 4.1 million) prove that the implementation scales well in both a strong and weak sense.

4. The design and tuning of the only known single- and multi-GPU implementation of RBF-FD. Also, ours is the only known investigation in the broader RBF community to target multiple GPUs ([34, 132] are single-GPU).

    - Our first paper ([21]) introduced a set of custom OpenCL kernels for RBF-FD time-stepping of hyperbolic PDEs with an explicit RK4 scheme (up to 10 GPUs).
    - Here, the performance per GPU is tuned by choosing an alternative sparse matrix representation. The investigation reveals that the Ellpack (ELL) structure provided by the ViennaCL library [125] is a great fit for RBF-FD differentiation matrices, with over 4x faster performance than the original compressed-row storage (CSR).
    - The multi-GPU implementation is extended to a novel overlapping algorithm that naturally derives from our distributed CPU optimizations. Non-blocking MPI collectives plus two asynchronous OpenCL queues amortize the cost of data transfer between CPU and GPU, MPI communication, and in some cases a substantial amount of computation. Iterations of the resulting implementation achieve an average of 3x better strong scaling compared to a non-overlapping equivalent.

Of lesser significance we also:

1. Verify our explicit solvers against two well studied hyperbolic PDEs (Vortex Roll-up [113, 114] and Cosine Bell Advection [88]).

- Convergence studies confirm that hyperviscosity ([58]) stabilizes solutions.
- Tuned parameters are provided for selecting the RBF support parameter and scaling hyperviscosity as a function of the problem size

2. Implement a distributed multi-GPU preconditioned GMRES within ViennaCL for implicit RBF-FD solutions.
   - A divergence-free field is constructed as the manufactured solution for a simplified version of Stokes flow constrained to the unit sphere.
   - A CPU-only preconditioner (Incomplete LU with Zero Fill-in) is implemented to test impact on GMRES convergence rates
3. Provide preliminary benchmarks evaluating the performance of RBF-FD on the Intel Phi Architecture.

10.1 Future Work

This work led to a number of additional investigations:

- There are a number of alternatives for SpMV that target various matrix features (e.g., BELL, SELL, SELL-T, etc.). New investigation will determine the which, if any, alternatives can benefit RBF-FD.
- ViennaCL has support for multiple backends, including the Intel Phi. Initial experiments result in performance $\frac{1}{10}$th as fast as the K20 GPU. Those results are: a) limited by a beta driver for OpenCL on Intel Phi; and b) using OpenCL kernels intended for the GPU architecture. Continued investigations are targeting features (language, vector instructions, etc.) of the Phi.
- To improve the accuracy of RBF-FD weights,
- New features on NVidia Kepler level GPUs and CUDA v5 allow dynamic parallelism. Dynamic parallelism allows GPU kernels to spawn new kernels directly on the GPU without returning to the CPU. It provides stream priority features which allow kernels to preemptively execute out of order based on priority and dependencies. Such a feature is powerful for MPI support where the GPU can continue processing kernels while waiting on MPI communication to complete.
- In Section 3.6 we mention the need for stable methods to solve for weights. We have partially implemented the RBF-GA method from [60]. Initial exposure to the algorithm leaves the impression that the change of basis on the RHS is a nuisance. In effort to bypass the difficulty

in deriving new RHS expressions, we plan to assemble complex DMs using DMs of lower (i.e., first-) order operators.

APPENDIX A

AVOIDING POLE SINGULARITIES WITH RBF-FD

This content follows [52, 53].

Chapter 8 introduces cosine bell advection as

$$\frac{\partial h}{\partial t} = \frac{u}{\cos\theta}\frac{\partial h}{\partial \lambda} + v\frac{\partial h}{\partial \theta} \tag{A.1}$$

in the spherical coordinate system defined by

$$x = \cos\theta\cos\lambda$$

$$y = \cos\theta\sin\lambda$$

$$z = \sin\theta$$

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

One of the many selling points for RBF-FD and other RBF methods is their ability analytically avoid pole singularities, which arise from the choice of coordinate system and not from the methods themselves. Since RBFs are based on Euclidean distance between nodes, and not the geodesic distance, it is said that they do not "feel" the effects of the geometry or recognize singularities naturally inherent in the coordinate system [52]. Here we demonstrate how pole singularities are analytically avoided with RBF-FD for cosine bell advection.

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

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

In spherical coordinates,

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

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

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

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

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

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

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

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

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

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

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

and solve the explicit method of lines problem:

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

where now the system is completely free of singularities at the poles [53].

We note that the expression $\cos(\frac{\pi}{2})$ evaluates on some systems to a very small value rather than zero (e.g., $6.1(10^{-17})$ on the Keeneland system with the GNU gcc compiler). The small value in turn causes $\frac{1}{\cos\theta}$ to result in a large value (e.g., $1.6(10^{16})$) rather than "inf" or "NaN". An "inf" or "NaN" would corrupt the numerics, but the large value allows terms to cancel in double precision. Rather than avoid placing nodes at the poles, or assume singularities cancel numerically, it is preferred to use operators A.4 and A.5 to analytically cancel singularities when computing RBF-FD weights.

We observe that the latter approach causes the conditioning of the system to change slightly. The hyperviscosity parameters provided in Chapter 8 function well in both cases. However, their impact on the eigenvalue distributions in the case of analytic cancellation is noticeably stronger and results in a larger spread of eigenvalues in the left half-plane.

APPENDIX B

PROJECTED WEIGHTS ON THE SPHERE

Operating on the sphere requires constrained operators described in Section 3.3.5:

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

where

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

Here we investigate the difference between DMs constructed using Equation B.1 on the RHS of the RBF-FD weight system (Equation 3.3) versus DMs composed from combinations of the standard Cartesian $\boldsymbol{\nabla}$ operator following Equation B.2. We label the former case as *direct weights* and the latter as *indirect weights*.

Direct Weights. Following [51], B.1 takes the following form when adapted to RBF-FD:

$$[\mathbf{p}_x\cdot\nabla f(\mathbf{x})]|_{\mathbf{x}=\mathbf{x}_c} = \sum_{k=1}^{n} c_k \underbrace{\left[x_c\mathbf{x}_c^T\mathbf{x}_k - x_k\right]\frac{1}{r}\frac{d\phi(r(x_c))}{dr}}_{B_{c,k}^{\mathbf{p}_x}}. \tag{B.3}$$

and so forth for the $\mathbf{p}_y\cdot\nabla, \mathbf{p}_z\cdot\nabla$ operators, where $\mathbf{x}_c$ is the stencil center and $\mathbf{x}_k$ are stencil nodes. To compute RBF-FD weights for the $\mathbf{p}_x\cdot\nabla$ operator, the RHS of Equation 3.3 is filled with elements $B_{c,k}^{\mathbf{p}_x}$. We will refer to this method of obtaining the weights as the *direct* method due to the ability to directly compute RBF-FD weights for the operators $\mathbf{P}\cdot\nabla$, and assemble the differentiation matrices $\mathbf{D}_{\mathbf{p_x}\cdot\nabla}, \mathbf{D}_{\mathbf{p_y}\cdot\nabla}, \mathbf{D}_{\mathbf{p_z}\cdot\nabla}$ without the need to compute and/or store other (unneeded) weights.

Indirect Weights. Alternatively, weights can be computed *indirectly* as a weighted combination of existing RBF-FD DMs for the unprojected ∇ operator. Here we assume that differentiation matrices to compute the components of ∇ are readily available in memory:

$$\mathbf{D}_{\nabla} = \begin{pmatrix} \mathbf{D}_x \\ \mathbf{D}_y \\ \mathbf{D}_z \end{pmatrix},$$

where each matrix contains weights computed with the operators from Section 3.3.2 applied to the RHS of Equation 3.3.

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

$$\mathbf{D}_{\mathbf{P}\cdot\nabla} = \begin{pmatrix} \mathbf{D}_{\mathbf{p_x}\cdot\nabla} \\ \mathbf{D}_{\mathbf{p_y}\cdot\nabla} \\ \mathbf{D}_{\mathbf{p_z}\cdot\nabla} \end{pmatrix} = \begin{pmatrix} diag(1-X^2)\mathbf{D}_x - diag(XY)\mathbf{D}_y - diag(XZ)\mathbf{D}_z \\ -diag(XY)\mathbf{D}_x + diag(1-Y^2)\mathbf{D}_y - diag(YZ)\mathbf{D}_z \\ -diag(XZ)\mathbf{D}_x - diag(YZ)\mathbf{D}_y + diag(1-Y^2)\mathbf{D}_z \end{pmatrix} \tag{B.4}$$

where $X = \{x_{c,i}\}_{i=1}^{N}$, $Y = \{y_{c,i}\}_{i=1}^{N}$, $Z = \{z_{c,i}\}_{i=1}^{N}$ are the individual component values of the stencil centers $\{\mathbf{x}_{c,i}\}_{i=1}^{N}$ respectively (i.e., $x_c = (x_c, y_c, z_c)$).

This concept equates to classical Finite Differences where for example, the standard 5-point finite difference formula in 2-D is expressed a weighted combination of coefficients for 1-D differences.

Indirectly computing weights is of interest for a few reasons:

- *The potential for memory conservation.* For example, consider a coupled PDE that requires four operators: $\mathbf{D}_x$, $\mathbf{D}_y$, $\mathbf{D}_z$, $\mathbf{D}_{\nabla^2}$. A single DM on $N = 10^6$ nodes with stencil size $n = 101$ requires roughly 1.6 GB of memory in double precision. Indirectly computing $\mathbf{D}_{\nabla^2}$ based on the three other DMs can save a large chunk of memory. For a GPU or other accelerator (e.g., Intel Phi) with only 6 GB of global memory, the savings can be compelling.

- *The ability to compose an operator with weights loaded from disk.* Possible use cases include distributing a generated grid online with precomputed RBF-FD weights for the Cartesian gradient. New investigations can reuse the grid and indirectly compose consistent operators for their problem.

- *Simplification in operator construction.* One often finds it difficult to directly/analytically apply high powered differential operators to RBFs. In those cases, the discretized operators can be easily composed indirectly via one or more matrix additions and/or multiplications.

B.1 Comparison of Direct and Indirect Weights

To verify functionality of direct versus indirect approaches, DMs are assembled with each method for the MD-node sets ranging between $N = 121$ and $N = 27556$. A manufactured solution is constructed on the sphere as shown in Figure B.1. Starting with the function $f(x) = Y_3^2 \sin(20x)$ in Figure B.1a, we seek approximations to the constrained derivative in x, $\mathbf{p}_x \cdot \nabla f(x)$ (Figure B.1b), and the constrained Laplacian, $\Delta_S f(x)$ (Figure B.1c).

The test case, $f(x)$, checks convergence of approximation on a scalar field with oscillations, which are intentionally difficult to capture without sufficient grid resolution. It also exercises RBF-FD's ability to capture derivatives constrained to the sphere while operating in Cartesian coordinates.

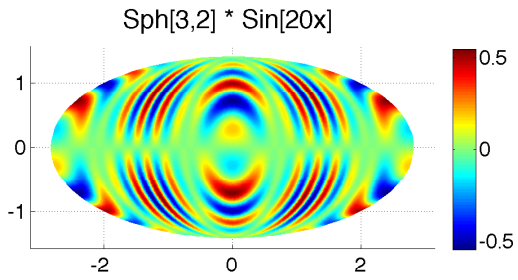
Since each DM results in approximation error, the expectation is that errors will compound when indirectly assembling DMs. Indirect weights for $\mathbf{p}_x \cdot \nabla$ are assembled following the first row of the RHS of Equation B.4 with $\mathbf{D}_x$, $\mathbf{D}_y$, and $\mathbf{D}_z$ given as the components of the Cartesian gradient. In the case of the Δ_S operator, the indirect weights are constructed as $(\mathbf{P} \cdot \nabla) \cdot (\mathbf{P} \cdot \nabla)$ to consider the worst case when errors multiply. The second order constrained operator applies Equation B.4 to the components of the Cartesian gradient, squares the resulting components of $\mathbf{D}_{\mathbf{P}\cdot\nabla}$ and adds components to get the final matrix with 9 matrix-vector multiplies, 8 matrix additions and 3 matrix multiplies. We arrive at the same results if Equation B.4 is applied both to the Cartesian gradient and then again to the resulting DMs for $\mathbf{D}_{\mathbf{P}\cdot\nabla}$, or if intermediate DMs are multiplied against function values and the resulting derivative approximations combined in place of the DMs (i.e., the final DM is not explicitly assembled).

The accuracy of each approximation is measured as the ℓ_2 relative error,

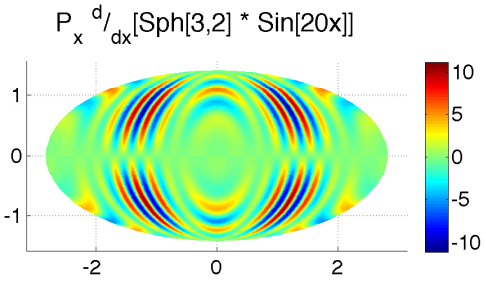
$$\text{relative } \ell_2 \text{ error} = \frac{||f_{approx} - f_{exact}||_2}{||f_{exact}||_2},$$

where f_{approx} is the approximate derivatives and f_{exact} is the manufactured solution. Figure B.2 provides the relative ℓ_2 errors for RBF-FD stencils of size $n = 17, 31, 50$, and 101. The choice of ϵ in each case follows the parameters provided in Table 8.1. RBF-FD weights are calculated using the RBF-Direct method, and no hyperviscosity.

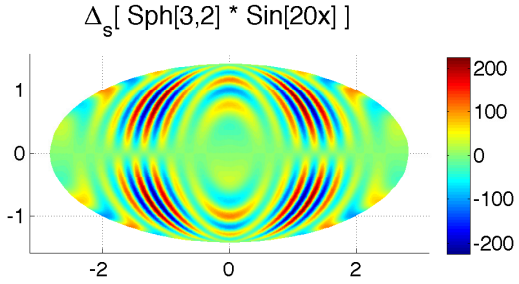
The relative error for both direct (Figure B.2a) and indirect approximations (Figure B.2b) of $\mathbf{p}_x \cdot \nabla$ are nearly identical. Although the grid is under-resolved and incapable of capturing oscillations for the first few test cases, the results converge well past $N > 400$. In the case of Δ_S,



(a) Test function: $Y_3^2 \sin(20x)$.



(b) $\mathbf{p}_x \cdot \nabla(Y_3^2 \sin(20x))$



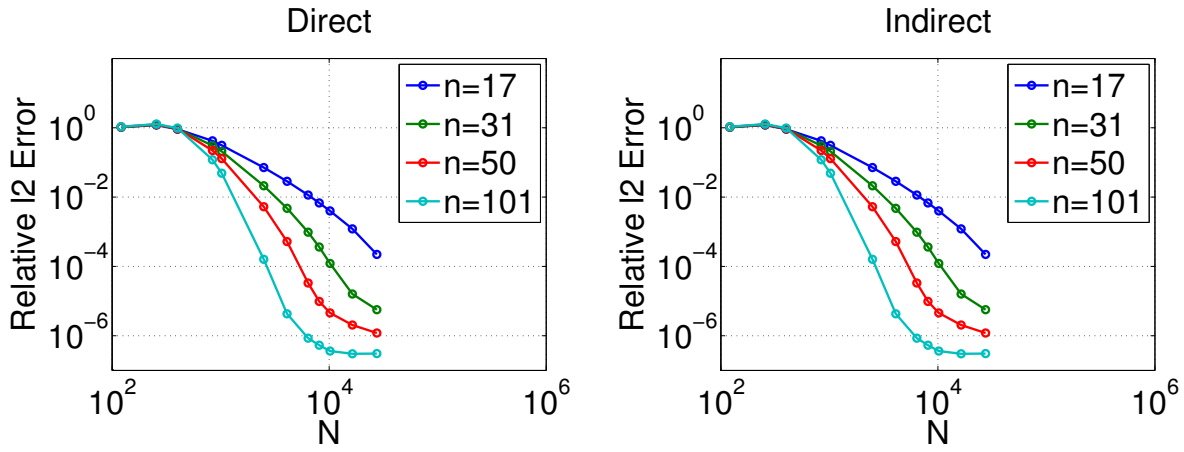
(c) Surface Laplacian: $\Delta_S(Y_3^2 \sin(20x))$

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

the relative ℓ_2 error for direct (Figure B.2c) versus indirect weights (Figure B.2d) behaves similarly for both cases when $N > 800$ nodes. However, the direct method starts converging earlier than the indirect method and maintains one or two digits higher precision overall.

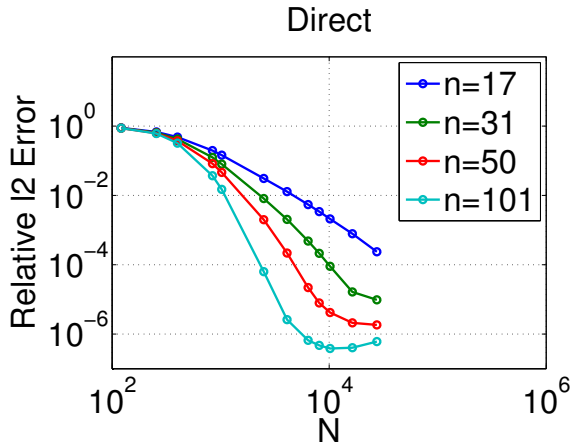
The plateau on convergence in Figure B.2 is an indicator of limitations in the RBF-Direct method and the choice of ϵ (see discussion of *saturation error* in [46, 49]). Future investigation may consider the use of an alternative weight method like RBF-GA ([60]) to maintain the rate of convergence. RBF-GA utilizes a change of basis on both the LHS and RHS of Equation 3.3, which implies the need to apply derivatives to both gaussian RBFs and incomplete gamma functions. With accurate RBF-GA weights, indirectly assembled DMs could be a viable alternative to the pain of deriving the modified RHS.

Figure B.3 considers the unsigned difference in relative ℓ_2 errors: $|(\ell_2)_{\text{direct}} - (\ell_2)_{\text{indirect}}|$. The results confirm that indirect weight approximations for the constrained first derivative are as good (or bad) as using direct weights. The absolute value of the differences shows over six digits of

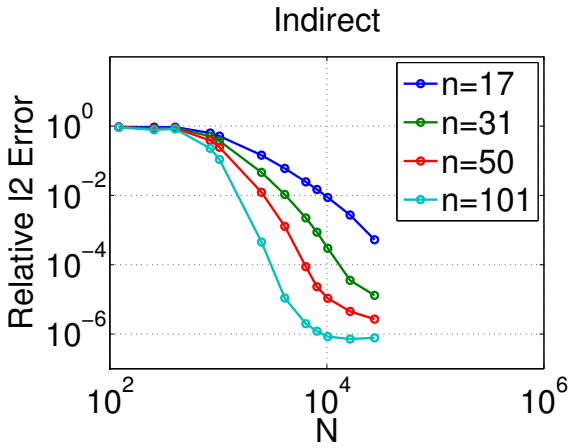


(a) $\mathbf{p}_x \cdot \nabla(Y_3^2 \sin(20x))$, direct weights

(b) $\mathbf{p}_x \cdot \nabla(Y_3^2 \sin(20x))$, indirect weights



(c) $\Delta_S(Y_3^2 \sin(20x))$, direct weights

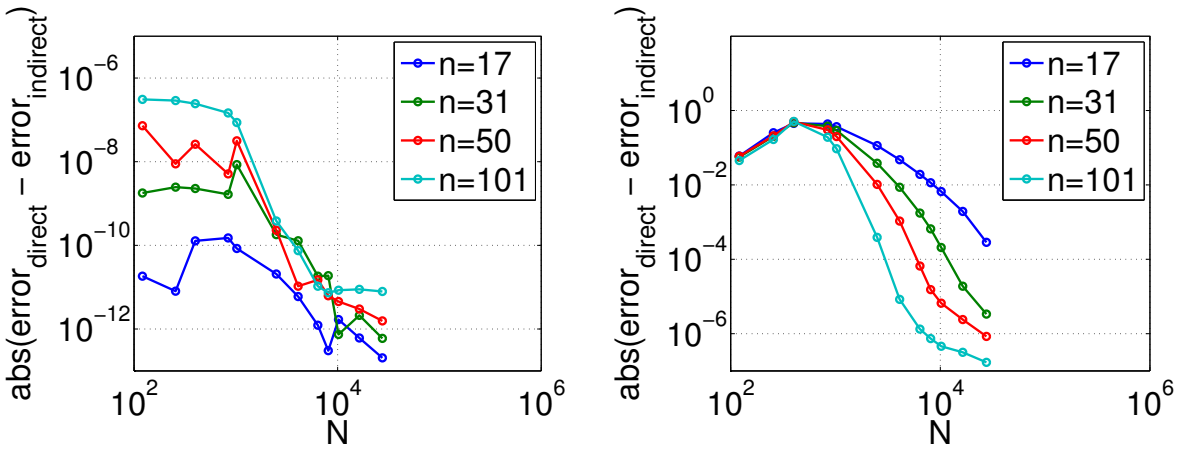


(d) $\Delta_S(Y_3^2 \sin(20x))$, indirect weights

Figure B.2: Relative ℓ_2 error in differentiation.

agreement in accuracy for the first derivative. In the case of Δ_S, the indirect approach is significantly less accurate with only one or two digits of agreement in many cases. Fortunately, as N grows the indirect and direct approximations converge.

In conclusion, the results in Figures B.2 and B.3 confirm that indirectly assembling RBF-FD weights can function well in terms of convergence, with similar rates as directly computed weights. Test results for the Δ_S operator show that higher order operators assembled from simple Cartesian gradient DMs may lose an order of magnitude accuracy in approximation, and require twice as many grid points to recover the solution produced by directly computed RBF-FD weights.



(a) $\mathbf{p}_x \cdot \nabla(Y_3^2 \sin(20x))$

(b) $\Delta_S(Y_3^2 \sin(20x))$

Figure B.3: Unsigned differences of relative ℓ_2 differentiation errors for Direct and Indirect weights.

APPENDIX C

KEENELAND BENCHMARKS

This appendix provides new data from Keeneland, a supercomputing cluster designed with 240 six-core Intel Xeon 5660 (2.80GHz) CPUs with HyperThreading (HT) enabled (x2 threads per core), 12MB cache per core shared on chip, 24GB system memory, and 360 NVidia M2070 GPUs with 6GB of device memory and 14 multiprocessors each [154]. A high level layout of the Keeneland system is presented in Figure C.1. The system was designed with its 240 CPUs partitioned into sets of two CPUs per compute node, four nodes per chassis, and six chassis per rack. An InfiniBand QDR Network handles inter-node communication. In addition to the dual CPUs, each node has three NVidia M2070 Fermi class GPUs.

At the onset of this dissertation, the KID system provided the initial development environment for our multi-CPU and multi-GPU implementation. In Spring 2012, the KID system was upgraded to the Keeneland Full Scale (KFS) system and the M2070 GPUs (150 Gbyte/sec bandwidth) were replaced by M2090 GPUs (177 Gbyte/sec [116]).

All data in this Appendix was acquired prior to the 2012 upgrade and is provided for direct comparison with our first paper ([21]), which demonstrated scaling of the test cases in § 8.2 and

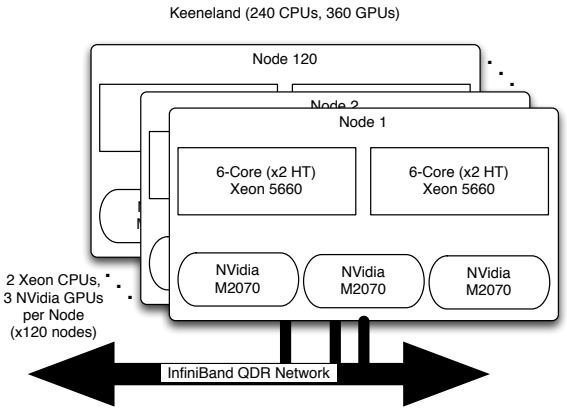


Figure C.1: The Keeneland Initial Delivery System (KIDS), a multi-GPU cluster with 360 GPUs and 240 CPUs capable of 201 TFLOP/sec. Each of the six-core Xeon chips has HyperThreading (HT) enabled for two software threads per core.

§ 8.1. The distributed RK4 iteration utilized the primitive MPI_Send/MPI_Recv communication routines to simulate a round-robin, all-to-subset collective where each processor took a turn to send data (0 or more bytes) or otherwise waited to receive messages. The choice of blocking routines were useful for code verification.

The implementation demonstrated poor scaling behavior as blocking collectives serialized transmissions, and every processor had to wait until the end of the collective to proceed. A linear-in-x partitioning was also employed on the sphere, which only worsens scaling (as shown in Section 6.1.2). However, the objective in [21], was not to present a well tuned implementation of RBF-FD for distributed GPU environments. Rather, we focused on the design decisions for partitioning, index mapping, etc. and verification of both single and distributed GPU implementations—as inefficient as they were.

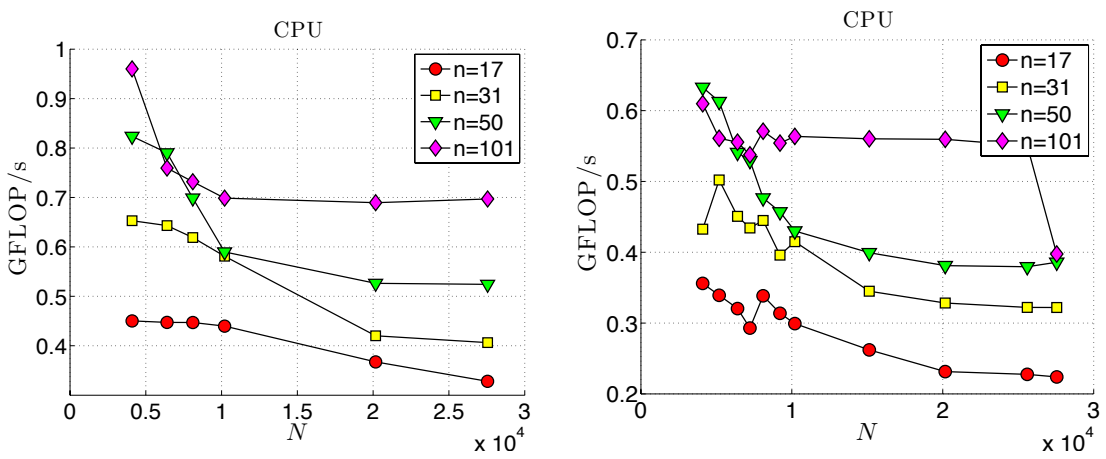
The benchmarks provided here replace the serialized sends and receives with an MPI_Alltoallv. Communication times have dropped significantly and improved scaling of the implementation. The linear partitioning remains, although the effects of the imbalanced workloads are not apparent because benchmarks only scale up to 10 GPUs with one process (i.e., one GPU) per node. All benchmarks on the GPU measure runtime of the custom OpenCL kernels described in Section **??**.

Unfortunately, shortly after the transition from KIDs to KFS our research allocation expired leaving scaling benchmarks for greater than 10 GPUs untested. The tuning efforts in Chapters 6 and 7 that provided load balancing and added overlapping communication and computation were completed too late to test on Keeneland.

C.1 Single GPU Performance

In [21] only speedup plots are provided to demonstrate improvement by the GPU over the CPU. For better analysis, data is presented here in both GFLOP/sec and speedup form.

Consider the performance of a full RK4 iteration for the Cosine Bell Advection test case in § 8.2. As discussed in Chapter 5, a regular SpMV involves $(2Nn)$ FLOPs. Intermediate evaluations in the RK4 time-step (i.e., the steps $k_{1...4}$ in Equation 5.1) perform three SpMV operations: one for each D_θ, D_λ and D_{hv}, plus another seven vector operations to scale and join derivatives $(5N)$ and prep the input for the subsequent evaluation $(2N)$. The cost of evaluating an intermediate step is therefore estimated at $(6Nn + 7N)$ FLOPs. In that case, the four evaluations of an RK4 Cosine



(a) Cosine Bell Advection

(b) Vortex Roll-Up

Figure C.2: GFLOP/sec on one CPU (1 core) of the Keeneland GPU cluster.

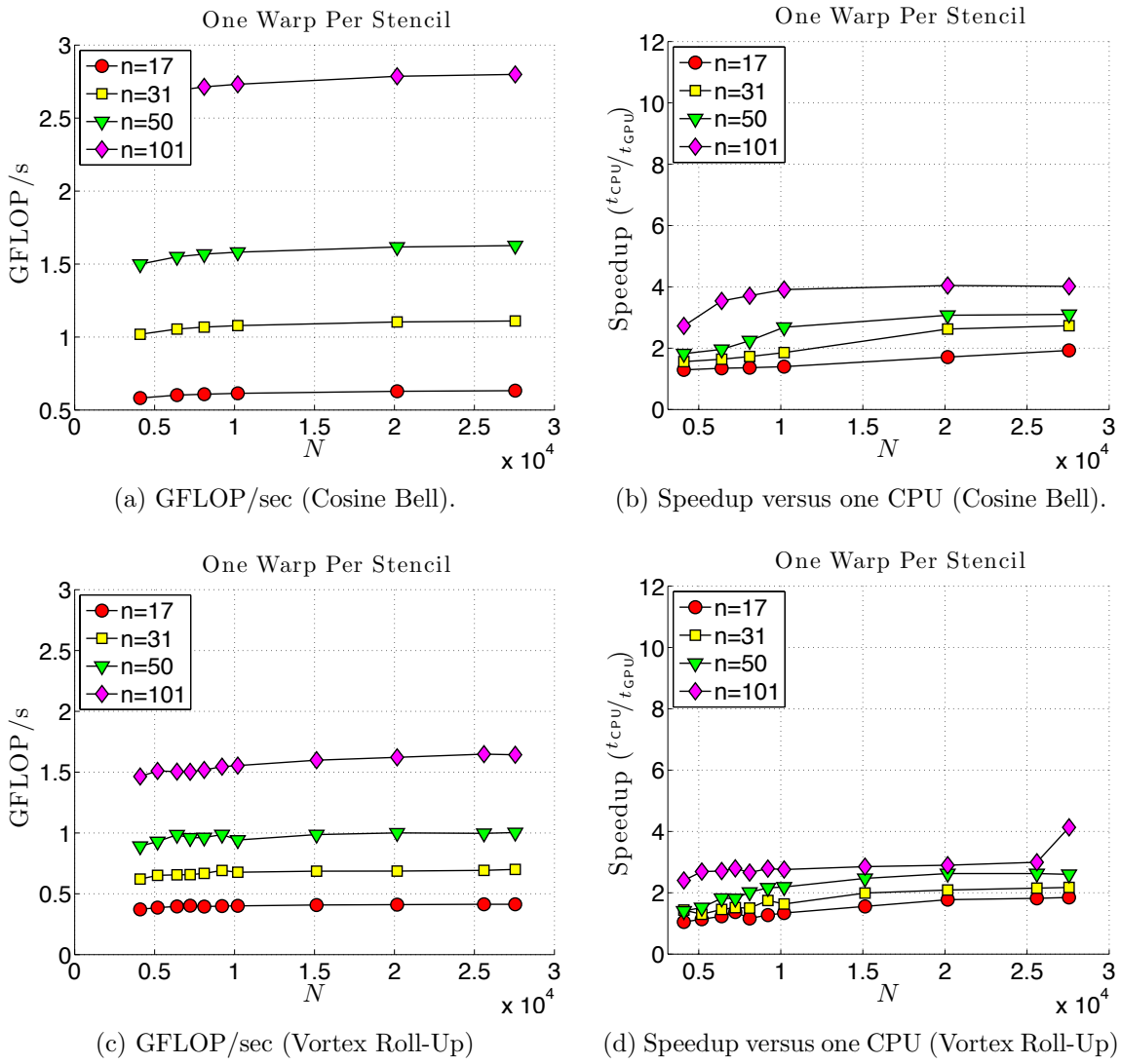
Bell iteration, plus an additional $5N$ FLOPs overhead to combine intermediate solutions totals to: $(24Nn + 33N)$ FLOPs.

The Vortex Roll-Up from § 8.2 computes only two SpMVs per intermediate evaluation of the RK4 time-step for a cost of $(4Nn + 5N)$. Following the same logic for the RK4 iteration we arrive at the total estimate: $(16Nn + 25N)$ FLOPs.

Using the FLOP estimate for each case, Figure C.2 provides the average GFLOP/sec observed for a single time-step of Cosine Bell Advection (Figure C.2a) and Vortex Roll-Up (Figure C.2b) on a single CPU in Keeneland. The average time is calculated over 10000 time-steps. In both cases the CPU performs the SpMV operation by running a simple nested loop on data packed in CSR format. The Intel compiler applies auto-vectorization, auto-blocking and other strategies to the nested loop based on the "-O2" compiler flag.

Figure C.3a and C.3c show the GFLOP/sec achieved by operating one warp per stencil in our custom GPU kernels on the M2070 GPUs. Once again the data is packed in a CSR format. The corresponding speedups achieved over the CPU is provided in Figures C.3b and C.3d. Due to its higher operation count, the Cosine Bell Advection is able to achieve over 2.75 GFLOP/sec for $n = 101$, nearly double the Vortex Roll-Up case. Also, by operating as warps dedicated to each stencil, the OpenCL kernel nicely maintains a consistent GFLOP/sec in all cases.

In similar fashion the GFLOP/sec achieved by operating one thread per stencil is demonstrated in Figure C.3a (Cosine Bell) and C.4c (Vortex Roll-Up). Speedups over the CPU for the thread



(a) GFLOP/sec (Cosine Bell).

(b) Speedup versus one CPU (Cosine Bell).

(c) GFLOP/sec (Vortex Roll-Up)

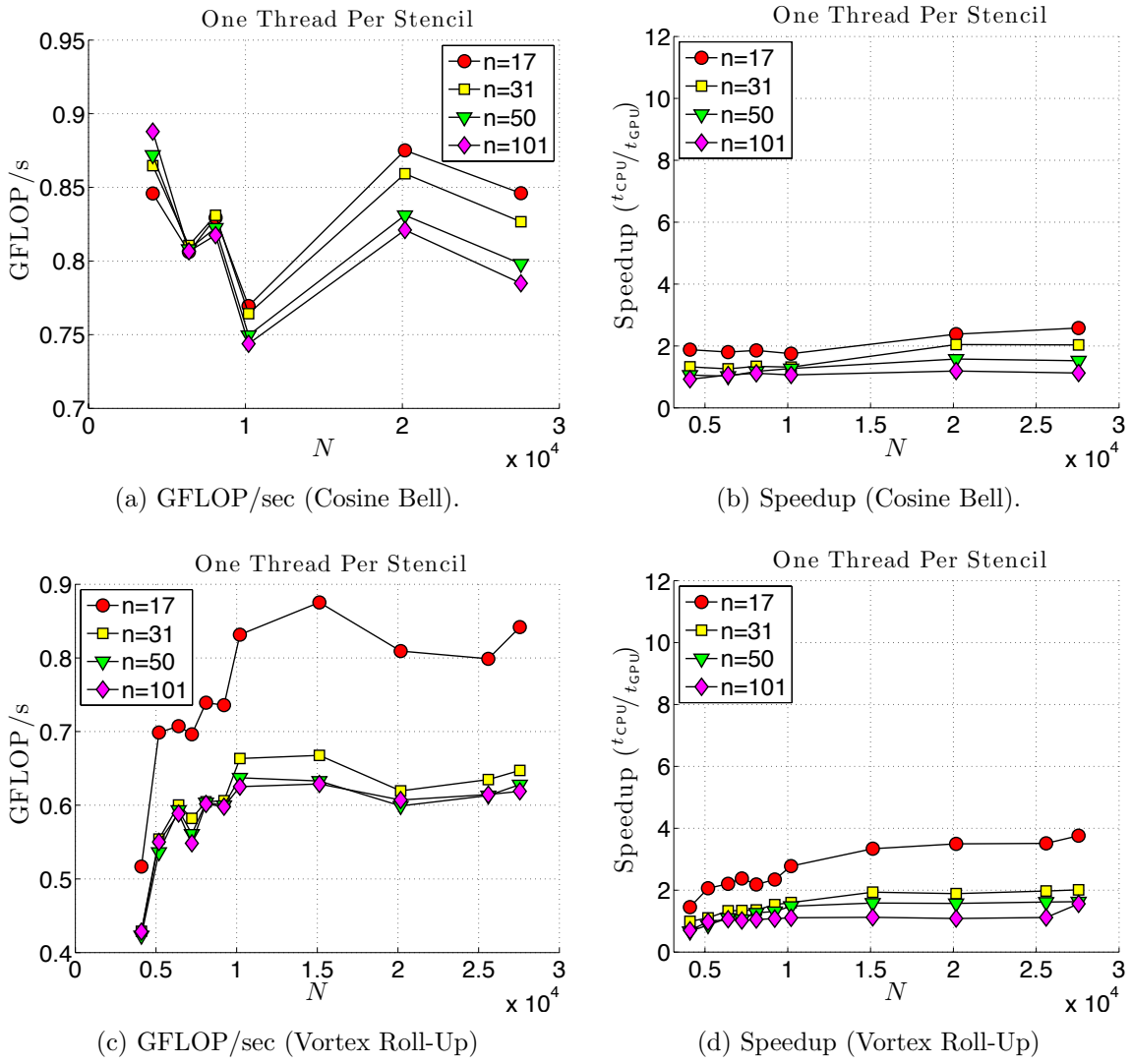
(d) Speedup versus one CPU (Vortex Roll-Up)

Figure C.3: GFLOP/sec and Speedup observed on one GPU (M2070) of the Keeneland GPU cluster for Cosine Bell Advection and Vortex Roll-Up when operating by "One Warp Per Stencil" within the RK4 kernel.

approach are given in Figures C.4b and C.4b.

Based on data in Figures C.3 and C.4 we observe:

1. For stencils of size $n < 32$, operating with perfect concurrency across stencils (i.e., operating one thread per stencil) allows for the highest GFLOP/sec by stencil size. Unfortunately, this approach achieves nearly equivalent GFLOP/sec across all stencil sizes, which emphasizes its limited applicability and diminishing returns. The achieved GFLOP/sec is also less predictable than the warp counterpart.



(a) GFLOP/sec (Cosine Bell).

(b) Speedup (Cosine Bell).

(c) GFLOP/sec (Vortex Roll-Up)

(d) Speedup (Vortex Roll-Up)

Figure C.4: GFLOP/sec and Speedup observed on one GPU (M2070) of the Keeneland GPU cluster when operating by "One Thread Per Stencil" within the RK4 kernel.

2. Operating by warp has surprisingly predictable performance. Each curve grows slightly as N scales, but the most significant factor impacting GFLOP/sec is the stencil size.

3. The additional operations for the third SpMV in cosine bell advection (Figure C.3b) also contribute significantly to the achieved throughput. This is to be expected as the RBF-FD (SpMV) is a memory bound problem, and the GPU work-items sit idle waiting for data to reach them. The added complexity of another SpMV makes use of cached data for the vector and is easily computed long before the idle time finishes.

These observations lead to the conclusion, that although operating one thread per stencil is an

option, under most circumstances operating by a warp on stencils is preferred. We also note that the demonstrated performance of the ELL matrix format in Chapter 5 is far superior to the data shown here. As such, effort to further tune these two kernels was diverted to continuing investigations with the help of the ViennaCL ELL matrix.

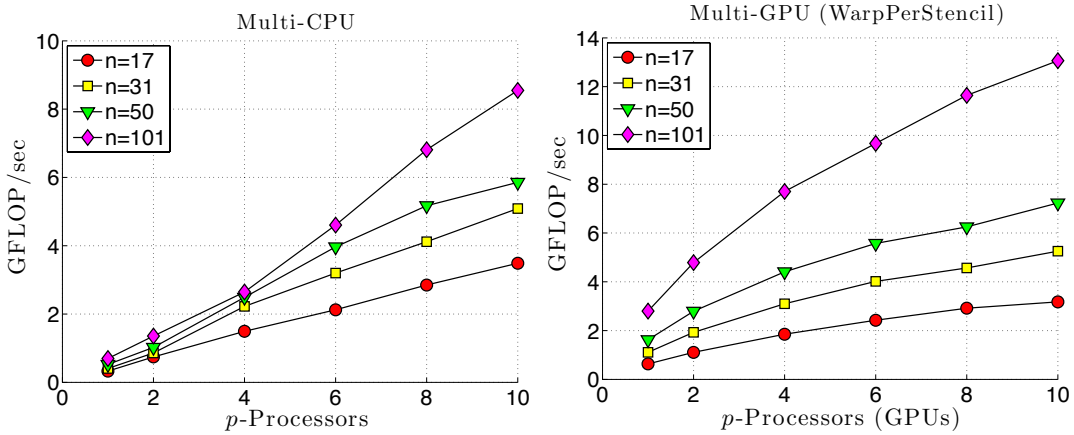
C.2 Keeneland Scaling with MPI_Alltoallv

The following data demonstrates performance of the implementation with an MPI_alltoallv collective. These benchmarks consider the cosine bell advection from Section 8.2 with $N = 27556$ nodes.

For comparison to results in Chapter 7 we provide actual GFLOP/sec achieved on Keeneland in Figure C.5.

Figure C.6a shows the strong scalability of our method on multiple CPUs. This figure demonstrates that our method does scale linearly (almost super-linearly) as the number of CPUs increases. The nice speedup is due to improved caching on processors that results as individual problem sizes decrease and the processors are able to keep a larger percentage of the problem within fast cache memory.

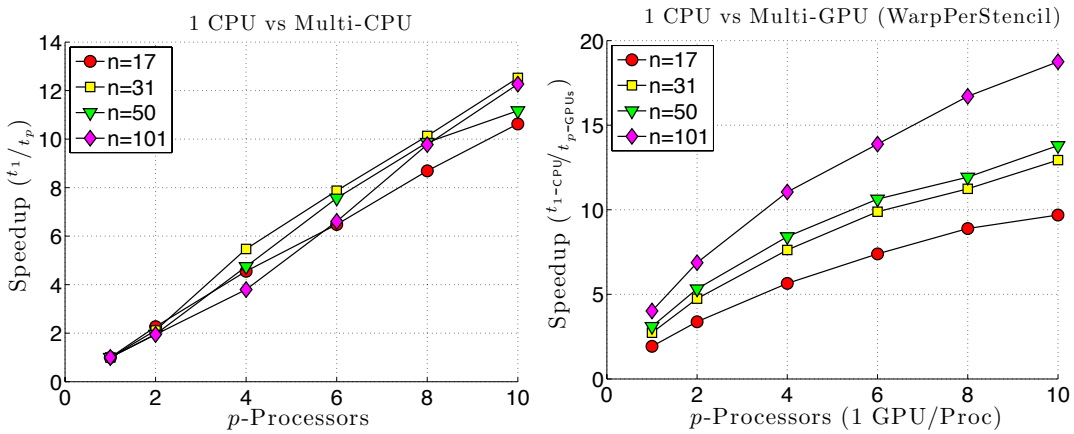
Figure C.6b shows the scaling of multiple GPUs vs 1 CPU. Ideally, this figure would be the product of Figures C.6a and C.3c (at $N = 27556$) since the GPUs are attached to CPUs in a one



(a) Multi-CPU strong scaling GFLOP/sec.

(b) Multi-GPU GFLOP/sec

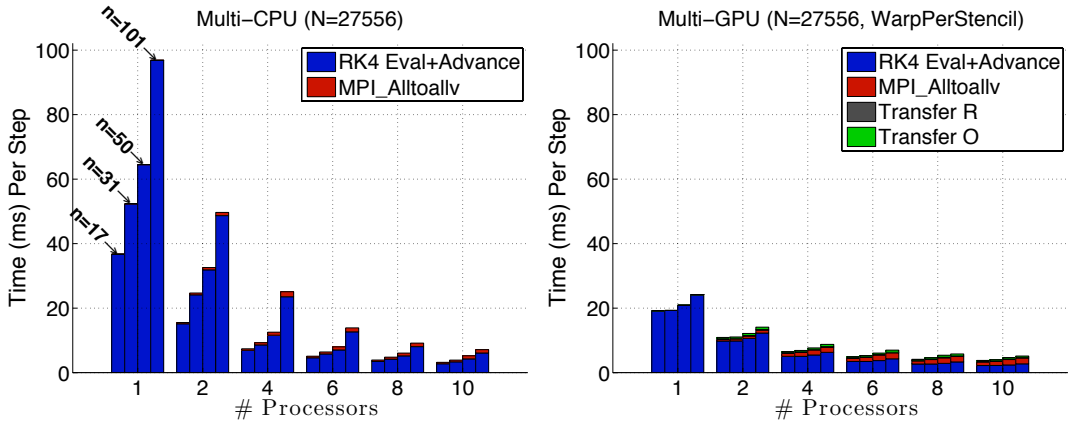
Figure C.5: GFLOP/sec achieved on Keeneland with a non-overlapping MPI_Alltoallv communication



(a) Multi-CPU strong scaling. The problem size is sufficiently large to hide latency in MPI communication.

(b) Multi-GPU strong scaling vs one CPU

Figure C.6: Multi-CPU and Multi-GPU strong scaling on Keeneland for the "One Warp per Stencil" RK4 implementation applied to the $N = 27556$ MD node set.



(a) Multi-CPU

(b) Multi-GPU

Figure C.7: Cost comparison of benchmark components on Keeneland. When the GPU accelerates computation, the non-parallelizable time for MPI_Alltoallv and data transfers between GPU and CPU quickly become the dominant factors.

to one correspondence. Unfortunately, gains quickly diminish. Operating on the GPU impacts scaling in two ways: a) the fraction of time spent in computation during an iteration is reduced, which means MPI communication dominates faster; and b) the GPU adds the cost of data transfers between host and device, which further exacerbates the situation.

Figure C.7a and Figure C.7b show the per-component time spent in communication, transfer and computation. The data reflects a significant reduction in the communication cost compared to the results published in [21], Figure 18. However, Figure C.7b shows that even with the more efficient MPI_Alltoallv collective, the time spent outside of computation (i.e., outside "RK4 Eval+Advance") is amplified to about 25% of the total iteration time for $p = 4$ GPUs. The transfer time to the GPU nearly doubles the cost of communication.

Based on this data from Keeneland it is clear that the multi-GPU implementation with the MPI_Alltoallv collective offers very little gain in comparison to the multi-CPU implementation. To scale to more GPUs, a larger problem size is absolutely necessary, but that alone is not sufficient: communication and transfer times need to be cut. This is answered (to a limited extent) by the work in Chapter 7.

APPENDIX D

COMMUNITY OUTREACH

- Erlebacher, G., Saule, E., Flyer, N., and Bollig, E. Sparse Matrix Vector Multiplication with Multiple vectors and Multiple Matrices on the MIC Architecture. Submitted to IEEE Parallel & Distributed Processing Symposium (IPDPS), 2014, October 2013

- Bollig, E. F., Flyer, N., and Erlebacher, G. Solution to PDEs using radial basis function finite-differences (rbf-fd) on multiple GPUs. *Journal of Computational Physics 231*, 21 (2012), 7133 – 7151

- Bollig, E. F., Flyer, N., and Erlebacher, G. Solving Elliptic PDEs with RBF-generated Finite Differences on Multiple GPUs. *in preparation* (2013)

- Bollig, E. F. Multi-GPU Solutions of Hyperbolic and Elliptic PDEs with RBF-FD. Contributed Talk. SIAMSEAS Annual Meeting. Huntsville, AL, March 2012

- Bollig, E. F. RBF-generated Finite Differences for Elliptic PDEs on Multiple GPUs. Poster. FSU Dept. of Sci. Comp. Computational Xpo. Tallahassee, FL, March 2013. Also presented at: the Minnesota Supercomputing Institute Research Exhibition, Minneapolis, MN, April 2013

- Bollig, E. F. Parallel Algorithms for RBF-FD Solutions to PDEs on the Sphere. Poster. FSU Dept. of Sci. Comp. Computational Xpo. Tallahassee, FL, March 2012

- Bollig, E. F. A Multi-CPU/GPU implementation of RBF-generated finite differences for PDEs on a Sphere. Poster. American Geophys. Union. San Francisco, CA, December 2011

- Bollig, E. F. Accelerating RBF-FD in Parallel Environments. Student Presentation. NSF CBMS Workshop: Radial Basis Functions Mathematical Developments and Applications. Darmouth, MA, June 2011

BIBLIOGRAPHY

[1] *BOOST C++ Libraries*. http://www.boost.org. 78

[2] Tesla kepler gpu accelerators. http://www.nvidia.com/content/tesla/pdf/Tesla-KSeries-Overview-LR.pdf, Oct 2012. 80

[3] Itasca (hp proliant bl280c g6 linux cluster). https://www.msi.umn.edu/hpc/itasca, June 2013. 64, 117

[4] Kdtreesearcher class. MATLAB R2013a Documentation (http://www.mathworks.com/help/stats/kdtreesearcherclass.html), Aug 2013. 52

[5] AccelerEyes. *Jacket User Guide - The GPU Engine for MATLAB*, 1.2.1 ed., November 2009. 7

[6] Anderson, E., Bai, Z., Bischof, C., Blackford, S., Demmel, J., Dongarra, J., Du Croz, J., Greenbaum, A., Hammarling, S., McKenney, A., and Sorensen, D. *LAPACK Users' Guide*, third ed. Society for Industrial and Applied Mathematics, Philadelphia, PA, 1999. 7, 35

[7] Bahi, J. M., Couturier, R., and Khodja, L. Z. Parallel gmres implementation for solving sparse linear systems on gpu clusters. In *SpringSim (HPC)* (2011), L. T. Watson, G. W. Howell, W. I. Thacker, and S. Seidel, Eds., SCS/ACM, pp. 12–19. 144

[8] Balaji, P., Chan, A., Gropp, W., Thakur, R., and Lusk, E. The importance of non-data-communication overheads in mpi. *Int. J. High Perform. Comput. Appl. 24*, 1 (Feb. 2010), 5–15. 114

[9] Barak, S. Gpu technology key to exascale says nvidia. http://www.eetimes.com/electronics-news/4230659/GPU-technology-key-to-exascale-says-Nvidia, November 2011. 3

[10] Beatson, R. K., Light, W. A., and Billings, S. Fast Solution of the Radial Basis Function Interpolation Equations: Domain Decomposition Methods. *SIAM J. Sci. Comput. 22*, 5 (2000), 1717–1740. 4, 102

[11] Bell, N., and Garland, M. Implementing sparse matrix-vector multiplication on throughput-oriented processors. *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis - SC '09*, 1 (2009), 1. 84, 89, 91, 94

[12] Bell, N., and Garland, M. Cusp: Generic parallel algorithms for sparse matrix and graph computations, 2012. Version 0.3.0. 8, 84, 144

[13] Bollig, E. F. A Multi-CPU/GPU implementation of RBF-generated finite differences for PDEs on a Sphere. Poster. American Geophys. Union. San Francisco, CA, December 2011.

[14] Bollig, E. F. Accelerating RBF-FD in Parallel Environments. Student Presentation. NSF CBMS Workshop: Radial Basis Functions Mathematical Developments and Applications. Darmouth, MA, June 2011.

[15] Bollig, E. F. A fixed-grid prototype for rbf stencil generation with various space filling curves. `https://github.com/bollig/rbf_hash_query`, 2012. 64, 69, 75, 78

[16] Bollig, E. F. Multi-GPU Solutions of Hyperbolic and Elliptic PDEs with RBF-FD. Contributed Talk. SIAMSEAS Annual Meeting. Huntsville, AL, March 2012.

[17] Bollig, E. F. Parallel Algorithms for RBF-FD Solutions to PDEs on the Sphere. Poster. FSU Dept. of Sci. Comp. Computational Xpo. Tallahassee, FL, March 2012.

[18] Bollig, E. F. Sphere grids. `https://github.com/bollig/sphere_grids`, Aug 2012. 45, 65

[19] Bollig, E. F. Radial basis function finite differences on the gpu. `https://github.com/bollig/rbffd_gpu`, Sep 2013. 95, 106

[20] Bollig, E. F. RBF-generated Finite Differences for Elliptic PDEs on Multiple GPUs. Poster. FSU Dept. of Sci. Comp. Computational Xpo. Tallahassee, FL, March 2013.

[21] Bollig, E. F., Flyer, N., and Erlebacher, G. Solution to PDEs using radial basis function finite-differences (rbf-fd) on multiple GPUs. *Journal of Computational Physics 231*, 21 (2012), 7133 – 7151. xiii, 5, 11, 42, 84, 85, 89, 91, 94, 102, 115, 119, 125, 131, 142, 143, 153, 154, 165, 166, 172

[22] Bollig, E. F., Flyer, N., and Erlebacher, G. Solving Elliptic PDEs with RBF-generated Finite Differences on Multiple GPUs. *in preparation* (2013).

[23] Brandstetter, A., and Artusi, A. Radial Basis Function Networks GPU Based Implementation. *IEEE Transaction on Neural Network 19*, 12 (December 2008), 2150–2161. 5, 6

[24] Carr, J. C., Beatson, R. K., Cherrie, J. B., Mitchell, T. J., Fright, W. R., McCallum, B. C., and Evans, T. R. Reconstruction and Representation of 3D Objects with Radial Basis Functions. In *SIGGRAPH '01: Proceedings of the 28th annual conference on Computer graphics and interactive techniques* (New York, NY, USA, 2001), ACM, pp. 67–76. 24

[25] Carr, J. C., Beatson, R. K., McCallum, B. C., Fright, W. R., McLennan, T. J., and Mitchell, T. J. Smooth Surface Reconstruction from Noisy Range Data.

In *GRAPHITE '03: Proceedings of the 1st international conference on Computer graphics and interactive techniques in Australasia and South East Asia* (New York, NY, USA, 2003), ACM, pp. 119–ff. 5, 6, 24

[26] Cecil, T., Qian, J., and Osher, S. Numerical Methods for High Dimensional Hamilton-Jacobi Equations Using Radial Basis Functions. *JOURNAL OF COMPUTATIONAL PHYSICS 196* (2004), 327–347. 2, 20, 32, 34, 35

[27] Chandhini, G., and Sanyasiraju, Y. Local RBF-FD Solutions for Steady Convection-Diffusion Problems. *International Journal for Numerical Methods in Engineering 72*, 3 (2007). 20, 34, 35

[28] Chinchapatnam, P. P., Djidjeli, K., Nair, P. B., and Tan, M. A compact RBF-FD based meshless method for the incompressible Navier–Stokes equations. *Proceedings of the Institution of Mechanical Engineers, Part M: Journal of Engineering for the Maritime Environment 223*, 3 (Mar. 2009), 275–290. 34

[29] Connor, M., and Kumar, P. Fast construction of k-nearest neighbor graphs for point clouds. *IEEE Transactions on Visualization and Computer Graphics 16* (2010), 599–608. xi, 78, 79

[30] Correa, C., Silver, D., and Chen, M. Volume Deformation via Scattered Data Interpolation. *Proceedings of Eurographics/IEEE VGTC Workshop on Volume Graphics* (2007), 91–98. 21

[31] Corrigan, A., Camelli, F., Löhner, R., and Wallin, J. Running Unstructured Grid CFD Solvers on Modern Graphics Hardware. In *19th AIAA Computational Fluid Dynamics Conference* (June 2009), no. AIAA 2009-4001. 5, 8

[32] Corrigan, A., and Dinh, H. Computing and Rendering Implicit Surfaces Composed of Radial Basis Functions on the GPU. *International Workshop on Volume Graphics* (2005). 5, 6

[33] Cuntz, N., Leidl, M., Darmstadt, T., Kolb, G., Salama, C., Böttinger, M., Klimarechenzentrum, D., and Hamburg, G. GPU-based Dynamic Flow Visualization for Climate Research Applications. *Proc. SimVis* (2007), 371–384. 5, 6

[34] Cuomo, S., Gallettiy, A., Giuntay, G., and Staracey, A. Surface reconstruction from scattered point via rbf interpolation on gpu. *CoRR abs/1305.5179* (2013). 5, 8, 154

[35] Cuthill, E., and McKee, J. Reducing the bandwidth of sparse symmetric matrices. In *Proceedings of the 1969 24th national conference* (New York, NY, USA, 1969), ACM '69, ACM, pp. 157–172. 73, 74

[36] Davydov, O., and Oanh, D. T. On the optimal shape parameter for gaussian radial basis

function finite difference approximation of the poisson equation. *Computers & Mathematics with Applications 62*, 5 (2011), 2143 – 2161. 20, 34, 50

[37] de Berg, M., Cheong, O., van Kreveld, M., and Overmars, M. *Computational Geometry: Algorithms and Applications*, 3rd ed. Springer-Verlag, 2008. 54, 55

[38] Dekker, K. Parallel GMRES and Domain Decomposition. Tech. rep., Delft University of Technology, Delft, The Netherlands, 2000. 144

[39] Divo, E., and Kassab, A. An Efficient Localized Radial Basis Function Meshless Method for Fluid Flow and Conjugate Heat Transfer. *Journal of Heat Transfer 129* (2007), 124. 4, 5, 6, 20, 21, 28, 103

[40] Divo, E., and Kassab, A. J. A meshless method for conjugate heat transfer problems. *Engineering Analysis with Boundary Elements 29*, 2 (2005), 136 – 149. 5, 6

[41] Du, Q., Gunzburger, M., Ju, L., and Wang, X. Centroidal Voronoi Tessellation Algorithms for Image Compression, Segmentation, and Multichannel Restoration. *Journal of Mathematical Imaging and Vision 24*, 2 (2006), 177–194. 143

[42] Du, Q., Gunzburger, M. D., and Ju, L. Voronoi-based Finite Volume Methods, Optimal Voronoi Meshes, and PDEs on the Sphere. *Computer Methods in Applied Mechanics and Engineering 192* (August 2003), 3933–3957. 47, 48

[43] Erlebacher, G., Saule, E., Flyer, N., and Bollig, E. Sparse Matrix Vector Multiplication with Multiple vectors and Multiple Matrices on the MIC Architecture. Submitted to IEEE Parallel & Distributed Processing Symposium (IPDPS), 2014, October 2013.

[44] Fasshauer, G. E. Solving Partial Differential Equations by Collocation with Radial Basis Functions. In *In: Surface Fitting and Multiresolution Methods A. Le M'ehaut'e, C. Rabut and L.L. Schumaker (eds.), Vanderbilt* (1997), University Press, pp. 131–138. 17, 18, 20, 26

[45] Fasshauer, G. E. RBF Collocation Methods and Pseudospectral Methods. Tech. rep., 2006. 15, 19, 20, 27

[46] Fasshauer, G. E. *Meshfree Approximation Methods with MATLAB*, vol. 6 of *Interdisciplinary Mathematical Sciences*. World Scientific Publishing Co. Pte. Ltd., 5 Toh Tuck Link, Singapore 596224, 2007. ix, 13, 15, 18, 19, 20, 21, 22, 23, 25, 27, 28, 34, 40, 53, 55, 162

[47] Fasshauer, G. E., and Zhang, J. G. On choosing "optimal" shape parameters for rbf approximation. *Numerical Algorithms 45*, 1-4 (2007), 345–368. 19

[48] Fedoseyev, A. I., Friedman, M. J., and Kansa, E. J. Improved Multiquadric Method for Elliptic Partial Differential Equations via PDE Collocation on the Boundary. *Computers & Mathematics with Applications 43*, 3-5 (2002), 439 – 455. 17, 19, 20, 27

[49] Flyer, N., and Fornberg, B. Radial basis functions: Developments and applications to planetary scale flows. *Computers & Fluids 46*, 1 (July 2011), 23–32. 2, 13, 20, 162

[50] Flyer, N., and Lehto, E. Rotational transport on a sphere: Local node refinement with radial basis functions. *Journal of Computational Physics 229*, 6 (Mar. 2010), 1954–1969. 2, 13, 20, 78, 131

[51] Flyer, N., Lehto, E., Blaise, S., Wright, G. B., and St-Cyr, A. Rbf-generated finite differences for nonlinear transport on a sphere: shallow water simulations. *Submitted to Elsevier* (2011), 1–29. 20, 33, 34, 41, 42, 43, 49, 50, 53, 55, 133, 147, 148, 159

[52] Flyer, N., and Wright, G. B. Transport schemes on a sphere using radial basis functions. *Journal of Computational Physics 226*, 1 (2007), 1059 – 1084. 2, 13, 20, 34, 131, 137, 139, 157

[53] Flyer, N., and Wright, G. B. A Radial Basis Function Method for the Shallow Water Equations on a Sphere. In *Proc. R. Soc. A* (December 2009), vol. 465, pp. 1949–1976. 2, 13, 19, 20, 41, 147, 148, 157, 158

[54] Fornberg, B., Driscoll, T., Wright, G., and Charles, R. Observations on the behavior of radial basis function approximations near boundaries. *Computers & Mathematics with Applications 43*, 3-5 (Feb. 2002), 473–490. 33

[55] Fornberg, B., and Flyer, N. Accuracy of Radial Basis Function Interpolation and Derivative Approximations on 1-D Infinite Grids. *Adv. Comput. Math 23* (2005), 5–20. 14

[56] Fornberg, B., Flyer, N., and Russell, J. Comparisons Between Pseudospectral and Radial Basis Function Derivative Approximations. *IMA Journal of Numerical Analysis* (2009). 46

[57] Fornberg, B., Larsson, E., and Flyer, N. Stable Computations with Gaussian Radial Basis Functions. *SIAM J. on Scientific Computing 33*, 2 (2011), 869—-892. 14, 15, 16, 20, 29, 30, 34, 50, 131

[58] Fornberg, B., and Lehto, E. Stabilization of rbf-generated finite difference methods for convective pdes. *J. Comput. Physics 230*, 6 (2011), 2270–2285. 11, 155

[59] Fornberg, B., and Lehto, E. Stabilization of RBF-generated finite difference methods for convective PDEs. *Journal of Computational Physics 230*, 6 (Mar. 2011), 2270–2285. 16, 20, 32, 33, 34, 35, 42, 43, 46, 53, 55, 131, 137, 139

[60] Fornberg, B., Lehto, E., and Powell, C. Stable calculation of gaussian-based rbf-fd stencils. Tech. Rep. 2012-018, Uppsala University, Numerical Analysis, 2012. 30, 34, 49, 50, 155, 162

[61] Fornberg, B., and Piret, C. A Stable Algorithm for Flat Radial Basis Functions on a Sphere. *SIAM Journal on Scientific Computing 30*, 1 (2007), 60–80. 14, 15, 30, 46, 50

[62] Fornberg, B., and Piret, C. On Choosing a Radial Basis Function and a Shape Parameter when Solving a Convective PDE on a Sphere. *Journal of Computational Physics 227*, 5 (2008), 2758 – 2780. ix, 14, 15, 20, 131

[63] Fornberg, B., and Wright, G. Stable computation of multiquadric interpolants for all values of the shape parameter. *Computers & Mathematics with Applications 48*, 5-6 (2004), 853 – 867. 14, 15, 20, 29

[64] Forum, T. M. Mpi: A message passing interface, 1993. 101, 115

[65] Franke, R. Scattered Data Interpolation: Tests of Some Method. *Mathematics of Computation 38*, 157 (1982), 181–200. 15, 20

[66] Friedman, J. H., Bentley, J. L., and Finkel, R. A. An algorithm for finding best matches in logarithmic expected time. *ACM Trans. Math. Softw. 3*, 3 (Sept. 1977), 209–226. 55, 56

[67] Gibbs, N. E., Poole, William G., J., and Stockmeyer, P. K. An algorithm for reducing the bandwidth and profile of a sparse matrix. *SIAM Journal on Numerical Analysis 13*, 2 (1976), pp. 236–250. 74

[68] Göddeke, D., Buijssen, S., Wobker, H., and Turek, S. GPU Acceleration of an Unmodified Parallel Finite Element Navier–Stokes Solver. *High Performance Computing and Simulation* (2009). 5, 8, 9, 143

[69] Göddeke, D., and Strzodka, R. Performance and accuracy of hardware-oriented native-, emulated- and mixed-precision solvers in FEM simulations (Part 2: Double Precision GPUs). Tech. rep., Technical University Dortmund, 2008. 5, 8, 9

[70] Göddeke, D., Strzodka, R., Mohd-Yusof, J., McCormick, P., Buijssen, S., Grajewski, M., and Turek, S. Exploring weak scalability for FEM calculations on a GPU-enhanced cluster. *Parallel Computing* (2007). 5, 8

[71] Göddeke, D., Wobker, H., Strzodka, R., Mohd-Yusof, J., McCormick, P., and Turek, S. Co-Processor Acceleration of an Unmodified Parallel Solid Mechanics Code with FEASTGPU. *International Journal of Computational Science and Engineering* (2008). 5, 8, 9

[72] Goswami, P., Schlegel, P., Solenthaler, B., and Pajarola, R. Interactive sph simulation and rendering on the gpu. In *Proceedings of the 2010 ACM SIGGRAPH/Eurographics Symposium on Computer Animation* (Aire-la-Ville, Switzerland, Switzerland, 2010), SCA '10, Eurographics Association, pp. 55–64. 59, 60, 66, 69, 78

[73] Green, S. Cuda particles. NVidia Whitepaper, 2010. 53, 58, 59, 60, 66, 69, 78

[74] Gropp, W. D., and Keyes, D. E. A domain decomposition method with locally uniform mesh refinement. In *Third International Symposium on Domain Decomposition Methods for Partial Differential Equations* (Philadelphia, 1990), T. F. Chan, R. Glowinski, J. Périaux, and O. B. Widlund, Eds., SIAM, pp. 115–129. 102, 105

[75] Gumerov, N., Duraiswami, R., and Dorland, W. Middleware for programming NVIDIA GPUs from Fortran 9x. 5, 7

[76] Gumerov, N., Duraiswami, R., Fantalgo, L., Elkridge, M., and Dorland, W. Efficient Personal Supercomputing in Fortran 9x on CPU-GPU Systems. 5, 7

[77] Gumerov, N. A., Duraiswami, R., and Borovikov, E. A. 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. 53

[78] Hardy, R. Multiquadratic Equations of Topography and Other Irregular Surfaces. *J. Geophysical Research*, 76 (1971), 1–905. 2, 13, 15

[79] Harris, M. *GPU Gems 2 : Programming Techniques for High-Performance Graphics and General-Purpose Computation*. Addison-Wesley, 2005, ch. 31 Mapping Computational Concepts to GPUs, pp. 493–508. 81

[80] Hendrickson, B., and Leland, R. The chaco user's guide: Version 2.0. Tech. rep., 1995. 106

[81] Hon, Y.-C., Cheung, K. F., Mao, X.-Z., and Kansa, E. J. A Multiquadric Solution for the Shallow Water Equations. *ASCE J. Hydraulic Engineering 125* (1999), 524–533. 19, 20

[82] Hon, Y. C., and Schaback, R. On unsymmetric collocation by radial basis functions. *Appl. Math. Comput. 119*, 2-3 (2001), 177–186. 18, 20, 23

[83] Humphrey, J. R., Price, D. K., Spagnoli, K. E., Paolini, A. L., and Kelmelis, E. J. Cula: hybrid gpu accelerated linear algebra routines, 2010. 144

[84] Ingber, M. S., Chen, C. S., and Tanski, J. A. A mesh free approach using radial basis functions and parallel domain decomposition for solving three-dimensional diffusion equations. *International Journal for Numerical Methods in Engineering 60*, 13 (2004), 2183–2201. 5, 6

[85] Ingber, M. S., Schmidt, C. C., Tanski, J. A., and Phillips, J. Boundary-element

analysis of 3-d diffusion problems using a parallel domain decomposition method. *Numerical Heat Transfer, Part B: Fundamentals 44*, 2 (2003), 145–164. 5, 6

[86] Iske, A. *Multiresolution Methods in Scattered Data Modeling.* Springer, 2004. 13, 15, 21, 22, 23

[87] Ivan, L., Sterck, H. D., Northrup, S. A., and Groth, C. P. T. Three-Dimensional MHD on Cubed-Sphere Grids: Parallel Solution-Adaptive Simulation Framework. In *20th AIAA CFD Conference* (2011), no. 3382, pp. 1325–1342. 105

[88] Jakob-Chien, R., Hack, J., and Williamson, D. Spectral transform solutions to the shallow water test set. *Journal of Computational Physics 119*, 1 (1995), 164–187. 11, 131, 154

[89] Jansen, T. C. *GPU++: An Embedded GPU Development System for General-Purpose Computations.* PhD thesis, Technische Universität München Forschungsintitut caesar in Bonn, 2007. 81

[90] Johnson, I. Real-time particle systems in the blender game engine. Master's thesis, Florida State University, November 2011. 53, 59, 60, 69, 78

[91] Kameyama, M., Kageyama, A., and Sato, T. Multigrid-based simulation code for mantle convection in spherical shell using Yin–Yang grid. *Physics of the Earth and Planetary Interiors 171*, 1-4 (Dec. 2008), 19–32. 142

[92] Kansa, E. J. Multiquadrics–A scattered data approximation scheme with applications to computational fluid-dynamics. I. Surface approximations and partial derivative estimates. *Computers Math. Applic*, 19 (1990), 127–145. 2, 13, 16, 17, 18, 20, 25

[93] Kansa, E. J. Multiquadrics–A scattered data approximation scheme with applications to computational fluid-dynamics. II. Solutions to parabolic, hyperbolic and elliptic partial differential equations. *Computers Math. Applic*, 19 (1990), 147–161. 2, 13, 16, 17, 18, 20, 25

[94] Karypis, G., and Kumar, V. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM Journal on Scientific Computing 20*, 1 (1999), 359–392. 10, 105, 106, 154

[95] Katta, K. K. *High-order central finite-volume schemes for atmospheric modeling.* PhD thesis, The University of Texas at El Paso, 2012. 105

[96] Khronos OpenCL Working Group. *The OpenCL Specification (Version: 1.0.48)*, October 2009. 84, 86, 87, 90

[97] Komatitsch, D., Erlebacher, G., Göddeke, D., and Michéa, D. High-order finite-

element seismic wave propagation modeling with MPI on a large GPU cluster. *J. Comput. Phys. 229*, 20 (2010), 7692–7714. 5, 9

[98] Komatitsch, D., Göddeke, D., Erlebacher, G., and Michéa, D. Modeling the propagation of elastic waves using spectral elements on a cluster of 192 GPUs. *Computer Science Research and Development 25*, 1-2 (2010), 75–82. 5, 9

[99] Komatitsch, D., Michéa, D., and Erlebacher, G. Porting a high-order finite-element earthquake modeling application to NVIDIA graphics cards using CUDA. *Journal of Parallel and Distributed Computing 69*, 5 (2009), 451–460. 5, 9

[100] Kosec, G., and Šarler, B. Solution of thermo-fluid problems by collocation with local pressure correction. *International Journal of Numerical Methods for Heat & Fluid Flow 18* (2008). 5, 20, 21

[101] Kraus, J. An introduction to cuda-aware mpi. `https://developer.nvidia.com/content/introduction-cuda-aware-mpi`, March 2013. 130

[102] Krog, Ø. E. GPU-based Real-Time Snow Avalanche Simulations. Master's thesis, Norwegian University of Science and Technology, June 2010. 53, 58, 59, 60, 66, 69, 78

[103] Larsson, E., and Fornberg, B. A Numerical Study of some Radial Basis Function based Solution Methods for Elliptic PDEs. *Comput. Math. Appl 46* (2003), 891–902. 14, 15, 18, 19, 20, 26, 27, 29

[104] Lawlor, O. S. Message Passing for GPGPU Clusters: cudaMPI. In *IEEE Cluster PPAC Workshop* (2009). 126

[105] Lawson, C. L., Hanson, R. J., Kincaid, D. R., and Krogh, F. T. Basic linear algebra subprograms for fortran usage. *ACM Trans. Math. Softw. 5*, 3 (Sept. 1979), 308–323. 7

[106] Lin, Y., Chen, C., Song, M., and Liu, Z. Dual-RBF based surface reconstruction. *Vis Comput 25*, 5-7 (May 2009), 599–607. 21

[107] Liu, W.-H., and Sherman, A. H. Comparative analysis of the cuthill-mckee and the reverse cuthill-mckee ordering algorithms for sparse matrices. *SIAM Journal on Numerical Analysis 13*, 2 (Apr 1976), pp. 198–213. 73, 74

[108] Liu, X., Liu, G., Tai, K., and Lam, K. Radial point interpolation collocation method (RPICM) for partial differential equations. *Computers & Mathematics with Applications 50*, 8-9 (2005), 1425 – 1442. 20, 21

[109] Mellor-crummey, J., Whalley, D., and Kennedy, K. Improving memory hierarchy performance for irregular applications using data and computation reorderings. In *International Journal of Parallel Programming* (2001), pp. 425–433. 69, 74

[110] Minden, V., Smith, B., and Knepley, M. Preliminary implementation of petsc using gpus. *Proceedings of the 2010 International Workshop of GPU Solutions to Multiscale Problems in Science and Engineering* (2010). 5, 8, 9

[111] Morse, B. S., Yoo, T. S., Rheingans, P., Chen, D. T., and Subramanian, K. R. Interpolating implicit surfaces from scattered surface data using compactly supported radial basis functions. In *ACM SIGGRAPH 2005 Courses* (New York, NY, USA, 2005), SIGGRAPH '05, ACM. 19, 21

[112] Mouat, C., and Beatson, R. RBF Collocation. Tech. Rep. UCDMS2002/3, Department of Mathematics & Statistics, University of Canterbury, New Zealand, February 2002. 20, 22, 25

[113] Nair, R., Thomas, S., and Loft, R. A discontinuous Galerkin transport scheme on the cubed sphere. *Monthly Weather Review 133*, 4 (Apr. 2005), 814–828. 11, 131, 137, 142, 154

[114] Nair, R. D., and Jablonowski, C. Moving Vortices on the Sphere: A Test Case for Horizontal Advection Problems. *Monthly Weather Review 136*, 2 (Feb. 2008), 699–711. 11, 131, 154

[115] NVidia. *NVIDIA CUDA - NVIDIA CUDA C - Programming Guide version 4.0*, March 2011. 7, 87, 90, 92

[116] NVIDIA. Tesla m-class gpu computing modules. Datasheet, May 2011. 165

[117] NVidia. *CUDA C Programming Guide*, July 2013. xi, 80, 81, 82, 88

[118] NVidia. *CUDA Toolkit Documentation*, Oct 2013. 7, 88, 89

[119] Owens, J. D., Luebke, D., Govindaraju, N., Harris, M., Krüger, J., Lefohn, A. E., and Purcell, T. J. A Survey of General-Purpose Computation on Graphics Hardware. *Computer Graphics Forum 26*, 1 (2007), 80–113. 80, 81

[120] Pellegrini, F. Scotch 3.1 user's guide. Tech. rep., 1996. 106

[121] Phillips, E., and Fatica, M. Implementing the himeno benchmark with cuda on gpu clusters. In *Parallel Distributed Processing (IPDPS), 2010 IEEE International Symposium on* (2010), pp. 1–10. 5, 9

[122] Phillips, E., Zhang, Y., Davis, R., and Owens, J. Rapid Aerodynamic Performance Prediction on a Cluster of Graphics Processing Units. *Proceedings of the 47th AIAA Aerospace Sciences Meeting, number AIAA 565* (2009). 5, 9, 83

[123] Porter, D. Itasca's IB Fabric. Tutorial Slides, `http://msi.umn.edu/~porter/tutorials/old1/Itasca_fin2.pdf`, 2009. 117

[124] Randall, D., Ringler, T., and Heikes, R. Climate modeling with spherical geodesic grids. *Computing in Science & Engineering* (2002), 32–41. 47, 105, 142

[125] Rupp, K., Rudolf, F., and Weinbub, J. ViennaCL-A High Level Linear Algebra Library for GPUs and Multi-Core CPUs. In *Proc. GPUScA* (2010), pp. 51—56. 8, 11, 84, 89, 144, 154

[126] Rupp, K., Weinbub, J., and Rudolf, F. Automatic Performance Optimization in ViennaCL for GPUs Categories and Subject Descriptors. In *Proceedings of the 9th Workshop on Parallel/High-Performance Object-Oriented Scientific Computing* (2010), ACM, pp. 6:1–6:6. 8, 84

[127] Saad, Y. *Iterative Methods for Sparse Linear Systems*, second ed. Society for Industrial Mathematics, 2003. 101, 152

[128] Saad, Y., and Schultz, M. H. GMRES: A generalized minimal residual algorithm for solving nonsymmetric linear systems. *SIAM J. Sci. Stat. Comput. 7*, 3 (1986), 856–869. 144

[129] Samet, H. *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. 53, 54, 55, 58, 59, 60

[130] Sanderson, C. Armadillo: An open source c++ linear algebra library for fast prototyping and computationally intensive experiments. Tech. rep., NICTA, 2010. 131

[131] Schaback, R. Multivariate Interpolation and Approximation by Translates of a Basis Function. In *Approximaton Theory VIII–Vol. 1: Approximation and Interpolation*, C. Chui and L. Schumaker, Eds. World Scientific Publishing Co., Inc, 1995, pp. 491–514. 13, 15, 29

[132] Schmidt, J., Piret, C., Kadlec, B., Yuen, D., Sevre, E., Zhang, N., and Liu, Y. Simulating Tsunami Shallow-Water Equations with Graphics Accelerated Hardware (GPU) and Radial Basis Functions (RBF). In *South China Sea Tsunami Workshop* (2008). 5, 7, 154

[133] Schmidt, J., Piret, C., Zhang, N., Kadlec, B., Yuen, D., Liu, Y., Wright, G., and Sevre, E. Modeling of Tsunami Waves and Atmospheric Swirling Flows with Graphics Processing Unit (GPU) and Radial Basis Functions (RBF). *Concurrency and Computat.: Pract. Exper.* (2009). 5, 7, 20, 143

[134] Schubert, G., Hager, G., Fehske, H., and Wellein, G. Parallel sparse matrix-vector multiplication as a test case for hybrid mpi+openmp programming. *CoRR abs/1101.0091* (2011). 5, 120, 126

[135] Shi, J. Y., Taifi, M., Pradeep, A., Khreishah, A., and Antony, V. Program scalability analysis for hpc cloud: Applying amdahl's law to nas benchmarks. *High Performance Computing, Networking Storage and Analysis, SC Companion: 0* (2012), 1215–1225. 102

[136] Shu, C., Ding, H., and Yeo, K. S. Local radial basis function-based differential quadrature method and its application to solve two-dimensional incompressible Navier-Stokes equations. *Computer Methods in Applied Mechanics and Engineering 192*, 7-8 (2003), 941 – 954. 2, 20, 32, 34, 35

[137] Shu, C., Ding, H., and Zhao, N. Numerical Comparison of Least Square-Based Finite-Difference (LSFD) and Radial Basis Function-Based Finite-Difference (RBFFD) Methods. *Computers and Mathematics with Applications 51*, 8 (2006), 1297–1310. 20, 34

[138] Skiena, S. *The Algorithm Design Manual (2. ed.)*. Springer, 2008. 54

[139] Sloan, I. H., and Womersley, R. S. Extremal systems of points and numerical integration on the sphere. *Adv. Comput. Math 21* (2003), 107–125. 46, 131

[140] St.-Cyr, A., Gander, M. J., and Thomas, S. J. Optimized multiplicative, additive, and restricted additive schwarz preconditioning. *SIAM J. Scientific Computing 29*, 6 (2007), 2402–2425. 102, 103

[141] Stevens, D., Power, H., Lees, M., and Morvan, H. A Meshless Solution Technique for the Solution of 3D Unsaturated Zone Problems, Based on Local Hermitian Interpolation with Radial Basis Functions. *Transp Porous Med 79*, 2 (Sep 2008), 149–169. 20, 21, 28

[142] Stevens, D., Power, H., Lees, M., and Morvan, H. The use of PDE centres in the local RBF Hermitian method for 3D convective-diffusion problems. *Journal of Computational Physics* (2009). 18, 20, 21, 28, 34

[143] Stevens, D., Power, H., and Morvan, H. An order-N complexity meshless algorithm for transport-type PDEs, based on local Hermitian interpolation. *Engineering Analysis with Boundary Elements 33*, 4 (2008), 425 – 441. 6, 20, 21, 28

[144] Stocco, L., and Schrack, G. On spatial orders and location codes. *Computers, IEEE Transactions on 58*, 3 (2009), 424–432. 71, 72

[145] Strey, A. A comparison of openmp and mpi for neural network simulations on a sunfire 6800. In *PARCO* (2003), G. R. Joubert, W. E. Nagel, F. J. Peters, and W. V. Walter, Eds., vol. 13 of *Advances in Parallel Computing*, Elsevier, pp. 201–208. 5, 6

[146] Su, B.-Y., and Keutzer, K. clspmv: A cross-platform opencl spmv framework on gpus. In *ICS* (2012), U. Banerjee, K. A. Gallivan, G. Bilardi, and M. Katevenis, Eds., ACM, pp. 353–364. 84, 94

[147] Tagliasacchi, A. kd-tree for matlab. `http://www.mathworks.com/matlabcentral/fileexchange/21512-kd-tree-for-matlab`, Sep 2010. 52, 53, 64

[148] Tagliasacchi, A. kd-tree matlab. `https://code.google.com/p/kdtree-matlab`, Jun

2012. 56, 64

[149] Thibault, J., and Senocak, I. CUDA implementation of a Navier-Stokes solver on multi-GPU desktop platforms for incompressible flows. *47th AIAA Aerospace Sciences Meeting* (2009), 2009–758. 5, 9, 103, 126

[150] Tolstykh, A. On using RBF-based differencing formulas for unstructured and mixed structured-unstructured grid calculations. In *Proceedings of the 16 IMACS World Congress, Lausanne* (2000), pp. 1–6. 2, 32

[151] Tolstykh, A. I., and Shirobokov, D. A. On using radial basis functions in a "finite difference mode" with applications to elasticity problems. In *Computational Mechanics*, vol. 33. Springer, December 2003, pp. 68 – 79. 2, 32

[152] Trendall, C., and Stewart, A. J. General Calculations using Graphics Hardware with Applications to Interactive Caustics. In *Proceedings of the Eurographics Workshop on Rendering Techniques 2000* (London, UK, 2000), Springer-Verlag, pp. 287–298. 81

[153] Vertnik, R., and Šarler, B. Meshless local radial basis function collocation method for convective-diffusive solid-liquid phase change problems. *International Journal of Numerical Methods for Heat & Fluid Flow 16*, 5 (2006), 617–640. 20, 21, 28

[154] Vetter, J., Glassbrook, R., Dongarra, J., Schwan, K., Loftis, B., McNally, S., Meredith, J., Rogers, J., Roth, P., Spafford, K., and Yalamanchili, S. Keeneland: Bringing heterogeneous GPU computing to the computational science community. *IEEE Computing in Science and Engineering 13*, 5 (2011), 90–95. 126, 165

[155] Šarler, B., and Vertnik, R. Meshfree Explicit Local Radial Basis Function Collocation Method for Diffusion Problems. *Computers and Mathematics with Applications 51*, 8 (2006), 1269–1282. 20, 21, 28

[156] Vuduc, R., Demmel, J. W., and Yelick, K. A. Oski: A library of automatically tuned sparse matrix kernels. In *Institute of Physics Publishing* (2005). 84

[157] Wang, J. G., and Liu, G. R. A point interpolation meshless method based on radial basis functions. *Int. J. Numer. Methods Eng. 54* (2002). 20, 21

[158] Weiler, M., Botchen, R., Stegmaier, S., Ertl, T., Huang, J., Jang, Y., Ebert, D., and Gaither, K. Hardware-Assisted Feature Analysis and Visualization of Procedurally Encoded Multifield Volumetric Data. *IEEE Computer Graphics and Applications 25*, 5 (2005), 72–81. 5, 6

[159] Wendland, H. Fast evaluation of radial basis functions: Methods based on partition of unity. In *Approximation Theory X: Wavelets, Splines, and Applications* (2002), Vanderbilt University Press, pp. 473–483. 53, 58, 60

[160] Wendland, H. *Scattered Data Approximation.* Cambridge Monographs on Applied and Computational Mathematics. Cambridge University Press, 2005. 52, 53, 58, 60

[161] Wildeman, A. Parallel preconditioners for stokes flow, August 2009. 143, 144

[162] Williams, S., Waterman, A., and Patterson, D. Roofline: an insightful visual performance model for multicore architectures. *Commun. ACM 52*, 4 (Apr. 2009), 65–76. 84

[163] Womeldorff, G. Spherical Centroidal Voronoi Tessellations: Point Generation and Density Functions Via Images. Master's thesis, Florida State University, 2008. 47, 48

[164] Womersley, R. Extremal (maximum determinant) points on the sphere S^2. `http://web.maths.unsw.edu.au/~rsw/Sphere/Extremal/New/index.html`, Oct 2007. 46

[165] Womersley, R. S., and Sloan, I. H. How good can polynomial interpolation on the sphere be?, 2001. 46

[166] Wright, G., and Fornberg, B. Scattered node mehrstellenverfahren-type formulas generated from radial basis functions. In *The International Conference on Computational Methods* (December 15-17 2004). 20, 35

[167] Wright, G. B. *Radial Basis Function Interpolation: Numerical and Analytical Developments.* PhD thesis, University of Colorado, 2003. 2, 20, 32, 34, 50

[168] Wright, G. B., Flyer, N., and Yuen, D. A. A hybrid radial basis function–pseudospectral method for thermal convection in a 3-d spherical shell. *Geochem. Geophys. Geosyst. 11*, Q07003 (2010), 18 pp. 2, 13, 20, 28, 41, 146

[169] Wright, G. B., and Fornberg, B. Scattered node compact finite difference-type formulas generated from radial basis functions. *J. Comput. Phys. 212*, 1 (2006), 99–123. 20, 22, 33

[170] Wu, Z. M. Hermite-Birkhoff interpolation of scattered data by radial basis functions. *Approx. Theory Appl*, 8 (1992), 1–10. 18

[171] Yang, X., Zhang, Z., and Zhou, P. Local Elastic Registration of Multimodal Medical Image Using Robust Point Matching and Compact Support RBF. In *BMEI '08: Proceedings of the 2008 International Conference on BioMedical Engineering and Informatics* (Washington, DC, USA, 2008), IEEE Computer Society, pp. 113–117. 21

[172] Ying, L. A kernel independent fast multipole algorithm for radial basis functions. *Journal of Computational Physics 213*, 2 (2006), 451 – 457. 53, 54

[173] Yokota, R., Barba, L., and Knepley, M. G. PetRBF — A parallel O(N) algorithm for radial basis function interpolation with Gaussians. *Computer Methods in Applied Mechanics and Engineering 199*, 25-28 (May 2010), 1793–1804. 5, 6, 8, 102, 143, 144

[174] Yokota, R., Hamada, T., Bardhan, J. P., Knepley, M. G., and Barba, L. A. Biomolecular electrostatics simulation by an fmm-based bem on 512 gpus. *CoRR abs/1007.4591* (2010). 5, 9

[175] Zhang, H., and Genton, M. G. Compactly supported radial basis function kernels, 2004. 21

BIOGRAPHICAL SKETCH

Evan Bollig was born on the 6th of June, 1983, and grew up in the sleepy town of Sun Prairie, WI. He completed his Bachelor studies at the University of Minnesota–Twin Cities in 2006 with dual degrees in Computer Science and German Studies.

In 2009, Evan was awarded a Master's Degree in Computational Science from the Department of Scientific Computing at Florida State University with a thesis titled, "Centroidal Voronoi Tessellation of Manifolds using the GPU". As a Master's student, Evan worked as both a graduate research and teaching assistant in the Department of Scientific Computing.

During the summer of 2009, Evan was one of 11 participants in the Summer Internship in Parallel Computational Science (SIParCS) program at the National Center for Atmospheric Research (NCAR). The following year he returned to NCAR for a second internship; this time with Dr. Natasha Flyer in the Institute for Mathematics Applied to Geosciences (IMAGe) division. In 2011, he was invited to return a third time as a graduate student visitor for an extended collaboration with Dr. Flyer between June 2011 and February 2012.

Evan was briefly employed as a Software Engineer by the Tallahassee, FL branch of Homes.com from August to December 2012.

He presently lives in Saint Paul, MN, and works as a research associate for the University of Minnesota Supercomputing Institute (since December 2012).