THE FLORIDA STATE UNIVERSITY

COLLEGE OF ARTS AND SCIENCE

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

By

EVAN F. BOLLIG

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

Degree Awarded:
Fall Semester, 2012

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

The members of the supervisory committee were:

Gordon Erlebacher
Professor Directing Thesis

Natasha Flyer
Outside University Representative

Mark Sussman
University Representative

Dennis Slice
Committee Member

Ming Ye
Committee Member

Janet Peterson
Committe Member

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

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ALGORITHMS

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 a discretized version of the domain. Traditionally, numerical methods have been *meshed methods*, in that they rely on some underlying grid/lattice to connect discretized points in a well-defined manner. More recently a new class of methods 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.

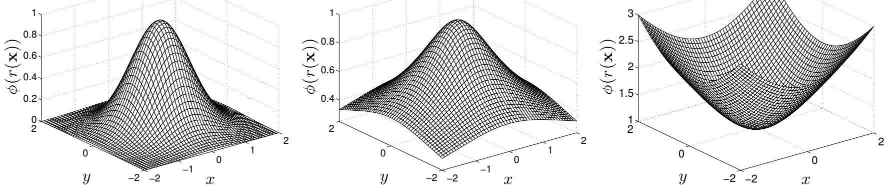
For decades meshed methods like Finite Difference, Finite Element and Finite Volume have been the powerhouses in computational modeling. The well-studied Eulerian schemes, with structured and stationary nodes, come backed by a vast literature on topics related to solvers, preconditioners, parallelization, 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:

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

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

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



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

Figure 1.1: Commonly Used Radial Basis Functions (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 [17, 40]. In addition to competitive accuracy and convergence compared with other state-of-the-art methods [15, 16, 17, 18, 40], 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 [19]. The transition from interpolation to RBF PDE methods dates back to 1990 [21, 22]. RBF-FD was first introduced in concept in 2000 [35], but it took another few years for the method to really get a start ([32], [34], [39] and [10]). 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 2, are many-core accelerators capable of general purpose, embarrassingly parallel computations. Accelerators represent the latest trend in HPC, where compute nodes are commonly supplemented by one or more accessory boards for 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 [5].

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 4).

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 **??** provides a historical survey of RBF methods as a backdrop to present RBF-FD in Chapter **??**. Chapter **??** introduces a new algorithm for generating RBF-FD stencils as a faster alternative to the RBF community favorite, k-D Tree. In Chapter 3, the first scalable implementation of RBF-FD to span one thousand processors is described in detail. Chapter 2 continues with the challenge of offloading computation to GPUs, and Chapter 4 expands the discussion to RBF-FD on a multi-GPU cluster. In Chapter **??** the parallel RBF-FD implementation is applied to solve both explicit and implicit geophysical PDEs. Finally, Chapter **??** concludes with a summary of novel contributions, results and a discussion on future directions.

1.1 On Parallel/Distributed RBF-FD

Parallel implementations of RBF methods rely on domain decomposition. Depending on the implementation, domain decomposition not only accelerates solution procedures, but can decrease the ill-conditioning that plague all global RBF methods [14]. 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. [6] in the year 2000 as a way to increase problem sizes into the millions of nodes.

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

In 2007, Divo and Kassab [14] used a domain decomposition method with artificial subdomain boundaries for their implementation of a local collocation method [14]. The subdomains are processed independently, with derivative values at artificial boundary points averaged to maintain global consistency of physical values. Their implementation was designed for a 36 node cluster, but benchmarks and scalability tests are not provided.

However, research on the parallelization of RBF algorithms to solve PDEs on multiple CPU/GPU architectures is essentially non-existent. We have found three studies that have addressed this topic, none of which implement RBF-FD but rather take the avenue of domain decomposition for global RBFs (similar to a spectral element approach). In [14], Divo and Kassab introduce subdomains with artificial boundaries that are processed independently. Their implementation was designed for a 36 node cluster, but benchmarks and scalability tests are not provided. Kosec and Šarler [25] parallelize coupled heat transfer and fluid flow models using OpenMP on a single workstation with one dual-core processor. They achieved a speedup factor of 1.85x over serial execution, although there were no results from scaling tests. Yokota, Barba and Knepley [41] apply a restrictive additive Schwarz domain decomposition to parallelize global RBF interpolation of more then 50 million nodes on 1024 CPU processors.

Kosec and Šarler [25] have the only known (to our knowledge) OpenMP implementation for RBFs. The authors parallelize coupled heat transfer and fluid flow problems on a single workstation. The application involves the local RBF collocation method, explicit time-stepping and Neumann boundary conditions. A speedup factor of 1.85x over serial execution was achieved by executing on two CPU cores; no further results from scaling tests were provided.

Stevens et al. [33] mention a parallel implementation under development, but no document is available at this time.

Perhaps the most competitive parallel implementation of RBFs is the PetRBF branch of PETSc [41]. 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 Author's Note: review: 128 processes.

1.2 On GPU RBF Methods

With regard to the latter, there is some research on leveraging RBFs on GPUs in the fields of visualization [12, 38], surface reconstruction [9, 11], and neural networks [8].

Only Schmidt et al. [31] have accelerated a global RBF method for PDEs on the GPU. Their MATLAB implementation applies global RBFs to solve the linearized shallow water equations utilizing the AccelerEyes Jacket [3] library to target a single GPU.

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

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

Gradually new and complex logic was added to the GPU to produce the shader languages that allowed developers to customize specific parts of the rendering pipeline. This allowed scientific problems such as the diffusion equation cite Lore and others to be solved in process of rendering. In other words, the GPU was tricked into computing.

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

Scientific Computing has seen a widespread adoption of GPGPU because of the goal to get to "exa-scale" computing, which may only be possible in the near future with the help of GPU accelerators [5].

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

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

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

Related work on RBFs and GPUs is sparse. In 2009, Schmidt et al. [30, 31] implemented a global RBF method for Tsunami simulation on the GPU using the AccelerEyes Jacket [3] add-on for MATLAB. Jacket provides a MATLAB interface to data structures and routines that internally call to the NVidia CUDA API. Their model was based on a single large dense matrix solve, and with the help of Jacket the authors were able to achieve approximately 7x speedup over the standard MATLAB solution on the then current generation of the MacBook Pro laptop. The authors compared the laptop CPU (processor details not specified) to the built-in NVidia GeForce 8600M GT GPU. Schmidt et al.'s implementation was the first contribution to the RBF community to leverage accelerators. The results were significant and promising, but no further contributions were made on the topic.

While both Schmidt et al.'s method and the method presented here are based on RBFs, the two problems are only distantly related when it comes to implementation on the GPU. Dense matrix operations have a high computational complexity, are considered ideal (or near to) by linear algebra libraries like BLAS [**?**] and LAPACK [4], and were demonstrated to fit well on GPUs from the onset of General Purpose GPU (GPGPU) Computing. In fact, NVidia included CUBLAS [**?**] (a GPU based BLAS library for their hardware) with their initial public release of the game-changing CUDA development kit in 2006. In stark contrast to this, sparse matrix operations have minimal computational complexity and are less than ideal for the GPU.

Earlier this year (2013), Cuomo et al. [13] implemented RBF-interpolation on the GPU for surface reconstruction. Their implementation utilizes PetRBF [41], and new built-in extensions that allow GPU access within PETSc. PETSc internally wraps the CUSP project [**?**] for sparse matrix algebra on the GPU. With the help of these libraries, Cuomo et al. solve and apply sparse interpolation systems on the GPU for up to three million nodes on an NVidia Fermi C1060 GPU (4GB). They compare results to a single core CPU implementation on an Intel i7-940 CPU and demonstrate that the GPU accelerate their solutions between 6x and 25x. Unfortunately, the authors do not show evidence of scaling the interpolation across multiple GPUs; so while evidence exists that PetRBF now has full GPU support, it remains to be seen how well the code can scale in GPU mode.

1.3 On Multi-GPU Methods

Multi-GPU Jacobi iteration for Navier stokes flow in cavity http://scholarworks.boisestate.edu/cgi/viewcontent.cgi?article=1003&context=mecheng_facpubs

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

CHAPTER 2

GPU SPMV

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, and with over 1 TFLOP/s peak throughput possible in double precision on a single GPU[2], a strong argument is made for the latter.

GPUs originated as dedicated hardware for video games and computer graphics (e.g., for rasterization). 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 programming model.

For many years leading up to 2006, researchers were able—with significant effort—to trick the GPU into solving scientific problems by encoding solutions within the graphics rendering process. The release of NVidia's CUDA architecture and software stack at the end of 2006, deprecated those tricks as developers could suddenly target the GPU with the ease of writing a general C-program. The CUDA hardware and software allowed for non-graphics applications to offload computation. GPUs were finally accessible for use by the general public, not limited to game designers and.

Today, CPUs are designed for scalar arithmetic and logic. GPUs emphasize vector operations with SIMD-like architecture. Due to limitations on how CPUs

CPU growth is limited by power and size. In response, CPU designers for years have been scaling the number of cores on a die rather than the frequency.

To meet the demand for computer graphics GPUs are designed with emphasis on fast memory bandwidth, thousands of simple compute cores and features like 2-D and 3-D texture caching.

In recent years the

CPU benefits versus GPU benefits. CPUs are designed for complex branching and advanced logic. GPUs are designed for high throughput. GPU cores are much simpler

marked both a new generation of GPU architecture and a programming model accessible

the addition of a new software layer that finally made GPGPU accessible to the general public. The CUDA API includes routines for memory control, interoperability with graphics contexts (i.e., OpenGL programs), and provides GPU implementation subsets of BLAS and FFTW libraries [26]. After the undeniable success of CUDA for C, new projects emerged to encourage GPU programming in languages like FORTRAN (see e.g., HMPP [1] and

Portland Group Inc.'s CUDA-FORTRAN [27]).

Modern GPUs boast thousands of compute cores.

This transition was followed closely by evolving programming languages. Today, GPUs can be leveraged from C/C++, FORTRAN, Java, Python, MATLAB, and more. The list seems endless, with new developments appearing every day.

into multi-core co-processors for high performance scientific 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 Language (OpenCL) [24]. 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.

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 [36].

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.

[7] [37] etc.

2.1 GPGPU

GPGPU evolution

2.1.1 OpenCL

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

2.1.2 Hardware Layout

Modern GPUs have a memory hierarchy and hardware layout.

2.2 Performance

2.2.1 GFLOP Throughput

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

The second quantification is to check the throughput of the process. By quantifying the GFLOP throughput we have a measure that tells us two things: first, a concrete number quantifying the amount of work performed per second by either hardware, and second because we can calculate the peak throughput possible on each hardware, we also have a measure of how occupied our CPU/GPU units are. With the GFLOPs we can also determine the cost per watt for computation and conclude on what problem sizes the GPU is cost effective to target and use.

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

2.2.2 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.

2.3 Targeting the GPU

2.3.1 OpenCL

2.3.2 Naive Kernels

2.3.3 SpMV Formats/Kernels

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

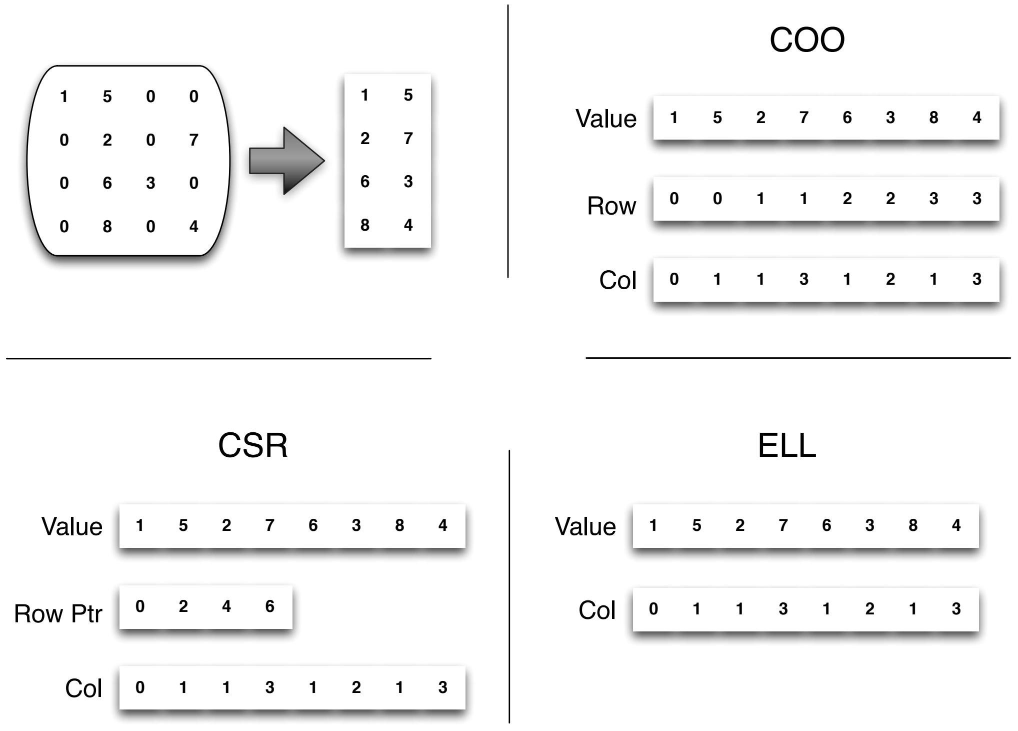


Figure 2.1: Sparse format example demonstrating a sparse matrix and corresponding storage data structures.

2.4 Performance Comparison

2.4.1 Performance of Cosine CL vs VCL

2.4.2 VCL Formats Comparison

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

As part of future work into accelerating RBF methods, investigations are underway into the new Intel Phi architecture.

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

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

Compare formats:

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

How is communication overlap handled with each format?

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

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

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

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

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

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

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

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

The library can tune itself to find the optimal number of work-items based on the device.

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

CHAPTER 3

DISTRIBUTED RBF-FD

Parallelizing any applications in a distributed computing environment requires three design decisions [29]. First, the problem must be partitioned in some fashion to distribute work across multiple processes. Intelligent partitioning impacts load balancing and the ratio of computation versus communication. Imbalanced computation can result in excessive delay per iteration as some processors tackle larger problem sizes with others sitting idle. Second, a decision must be made on what subset of node information, solution values, etc. are visible to each process, and then establish index mappings that translate between a local context and the global problem. Third, each process must establish a local ordering of data. Choosing the right local order can improve solver performance and/or simplify data movement. Of particular interest here is how the local ordering can minimize data transfer between CPU and GPU.

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 design decisions made here will tie into the resulting performance of the distributed multi-GPU implementations.

3.1 Partitioning

For ease of development and parallel debugging, partitioning is initially assumed to be linear within one physical direction (typically the x-direction). Figure 3.1 illustrates a partitioning of $N = 10,201$ nodes on the unit sphere onto four CPUs.

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

In Figure 3.1, alternating representations between node points and interpolated surfaces illustrates the overlap regions where ghost nodes reside. Due to stencil dependencies in each partition, the overlap region representations are double-wide—i.e., they contain a set of ghost nodes for both the left and right partitions.

As the stencil size increases, the width of the overlap regions relative to total number of nodes on the sphere proportionally increases. In the case of the unit sphere from Figure 3.1, the width of the overlap is roughly $\sqrt{n}$ for stencil size n. Figure 3.1 shows the case of $n = 31$ nodes per stencil. Higher order RBF-FD stencils (i.e., larger stencil sizes) exacerbate the situation by further increasing the number of bytes that must be sent via MPI. Observe that since stencils need not have symmetric dependencies (i.e., if stencil s_1 depends on s_2, s_2 need not depend on s_1), the number of ghost nodes for each partition can vary.

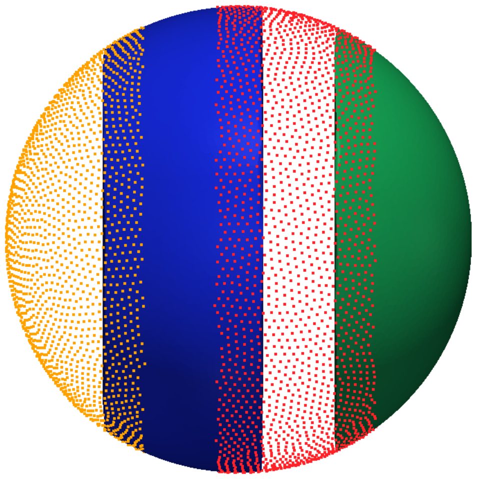


Figure 3.1: Partitioning of $N = 10,201$ nodes to span four processors with stencil size $n = 31$.

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

Many other options for partitioning the sphere exist. In atmospheric and geophysical communities for example, one often finds the cubed-sphere [20**?**], which transcribes a subdivided cube onto the sphere and assigns projected rectangular elements to individual processors. Another option is the icosahedral geodesic grid [28], which evenly balances the computational load by distributing equal sized geodesic triangles across processors. The options for partitioning the sphere are endless, and are outside the scope of this work.

Other interesting partitionings can be generated with software libraries such as the METIS [23] family of algorithms, capable of partitioning and reordering directed graphs produced by RBF-FD stencils.

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

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

METIS divides stencils into contiguous partitions of nearly equivalent size.

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

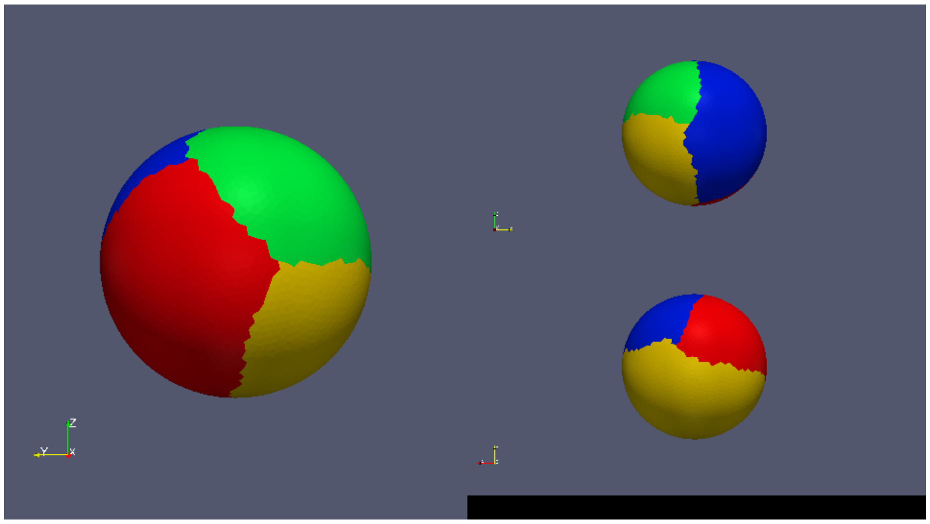


Figure 3.2: METIS partitioning of $N = 10,201$ nodes to span four processors with stencil size $n = 31$.

3.2 Index Mappings and Local Node Ordering

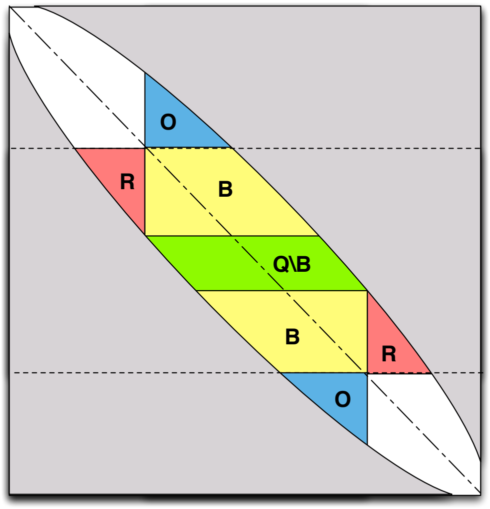


Figure 3.3: Decomposition for one processor selects a subset of rows from the DM. Blocks corresponding to node sets $\mathcal{Q}\backslash\mathcal{B}$, $\mathcal{O}$, and $\mathcal{R}$ are labeled for clarity. The subdomain for the processor is outlined by dashed lines.

3.3 Local node ordering

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

It is convenient to break up the nodes on a given CPU into various sets according to whether they are sent to other processors, are retrieved from other processors, are permanently on the processor, etc. Note as well, that each node has a home processor since the RBF nodes are partitioned into multiple domains without overlap. Table 3.1, defines the collection of index lists that each CPU must maintain for both multi-CPU and multi-GPU implementations.

Figure 3.5 illustrates a configuration with two CPUs and two GPUs, and 9 stencils, four on CPU1, and five on CPU2, separated by a vertical line in the figure. Each stencil has size $n = 5$. In the top part of the figures, the stencils are laid out with blue arrows pointing to stencil neighbors and creating the edges of a directed adjacency graph. Note that the connection between two nodes is not always bidirectional. For example, node 6 is in the stencil of node 3, but node 3 *is not* a member of the stencil of node 6. Gray arrows point to stencil neighbors outside the small window and are not relevant to the following discussion, which focuses only on data flow between CPU1 and CPU2. Since each CPU is responsible for the derivative evaluation and solution updates for any stencil center, it is clear that some nodes have a stencil with nodes that are on a different CPU. For example,

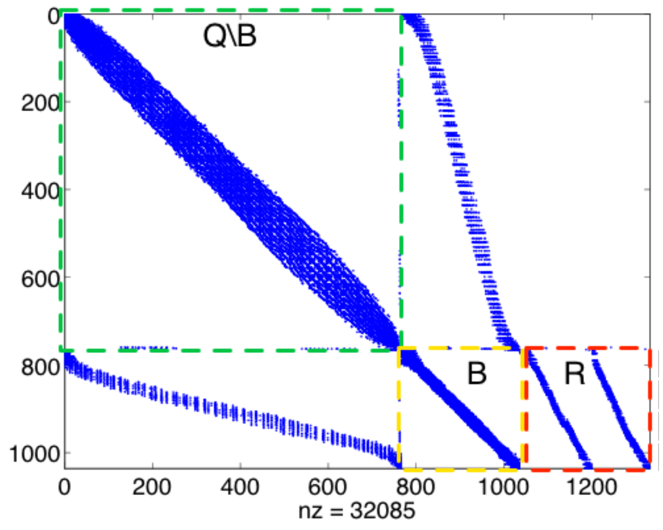


Figure 3.4: Spy of the sub-DM view on processor 3 of 4 from a METIS partitioning of $N = 4096$ nodes with stencil size $n = 31$ and stencils generated with Algorithm **??** ($hnx = 100$). Blocks are highlighted to distinguish node sets $\mathcal{Q}\backslash\mathcal{B}$, $\mathcal{B}$, and $\mathcal{R}$. $\mathcal{R}$ 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.

node 8 on CPU1 has a stencil comprised of nodes 4,5,6,9, and itself. The data associated with node 6 must be retrieved from CPU2. Similarly, the data from node 5 must be sent to CPU2 to complete calculations at the center of node 6.

The set of all nodes that a CPU interacts with is denoted by $\mathcal{G}$, which includes not only the nodes stored on the CPU, but the nodes required from other CPUs to complete the calculations. The set $\mathcal{Q} \in \mathcal{G}$ contains the nodes at which the CPU will compute derivatives and apply solution updates. The set $\mathcal{R} = \mathcal{G}\backslash\mathcal{Q}$ is formed from the set of nodes whose values must be retrieved from another CPU. For each CPU, the set $\mathcal{O} \in \mathcal{Q}$ is sent to other CPUs. The set $\mathcal{B} \in \mathcal{Q}$ consists of nodes that depend on values from $\mathcal{R}$ in order to evaluate derivatives. Note that $\mathcal{O}$ and $\mathcal{B}$ can overlap, but differ in size, since the directed adjacency graph produced by stencil edges is not necessarily symmetric. The set $\mathcal{B}\backslash\mathcal{O}$ represents nodes that depend on $\mathcal{R}$ but are not sent to other CPUs, while $\mathcal{Q}\backslash\mathcal{B}$ are nodes that have no dependency on information from other CPUs. The middle section Figure 3.5 lists global node indices contained in $\mathcal{G}$ for each CPU. Global indices are paired with local indices to indicate the node ordering internal to each CPU. The structure of set $\mathcal{G}$,

$$\mathcal{G} = \{\mathcal{Q}\backslash\mathcal{B} \quad \mathcal{B}\backslash\mathcal{O} \quad \mathcal{O} \quad \mathcal{R}\}, \tag{3.1}$$

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

$\mathcal{G}$	: 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 3.1: Sets defined for stencil distribution to multiple CPUs

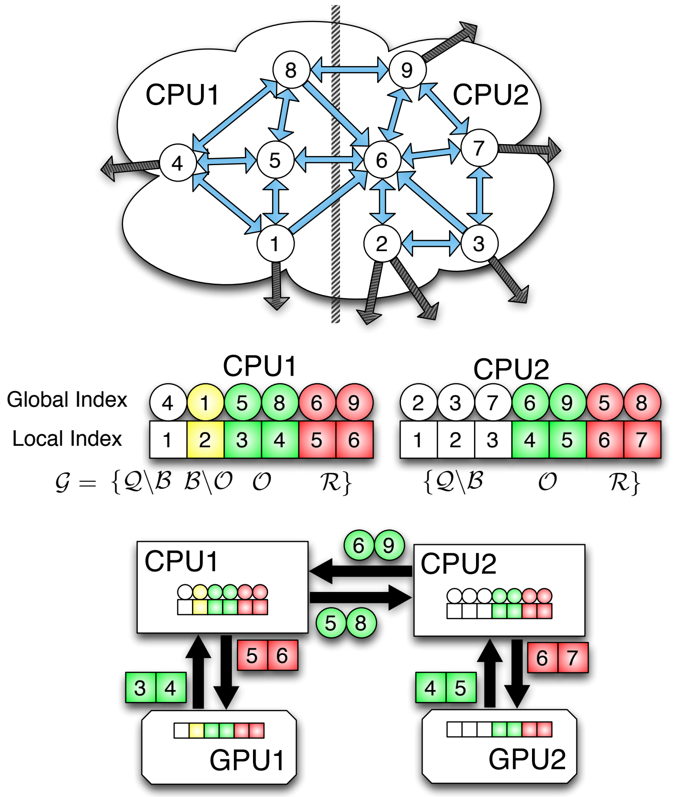

CPU1
CPU2
Global Index
Local Index
$\mathcal{G} = \{\mathcal{Q}\backslash\mathcal{B}$ $\mathcal{B}\backslash\mathcal{O}$ $\mathcal{O}$ $\mathcal{R}\}$
$\{\mathcal{Q}\backslash\mathcal{B}$ $\mathcal{O}$ $\mathcal{R}\}$
GPU1
GPU2

Figure 3.5: 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 3.1: $\mathcal{Q}\backslash\mathcal{B}$ is white, $\mathcal{B}\backslash\mathcal{O}$ is yellow, $\mathcal{O}$ is green and $\mathcal{R}$ is red.

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

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

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

Figure 3.1 illustrates a partitioning of $N = 10,201$ nodes on the unit sphere onto four CPUs. Each partition, illustrated as a unique color, represents set $\mathcal{G}$ for a single CPU. Alternating representations between node points and interpolated surfaces illustrates the overlap regions where nodes in sets $\mathcal{O}$ and $\mathcal{R}$ (i.e., nodes requiring MPI communication) reside. As stencil size increases, the width of the overlap regions relative to total number of nodes on the sphere also increases.

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

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

2. Distribute $\mathcal{O}$ to other CPUs, receive R from other CPUs

3. Transfer $\mathcal{R}$ to the GPU

4. Launch a GPU kernel to operate on $\mathcal{Q}$

The data transfers involved in this process are illustrated at the bottom of Figure 3.5. Each GPU operates on the local indices ordered according to Equation (3.1). The set $\mathcal{O}$ is copied off the GPU and into CPU memory as one contiguous memory block. The CPU then maps local to global indices and transfers $\mathcal{O}$ to other CPUs. CPUs send only the subset of node values from $\mathcal{O}$ that is required by the destination processors, but it is important to note that node information might be sent to several destinations. As the set $\mathcal{R}$ is received, the CPU converts back from global to local indices before copying a contiguous block of memory to the GPU. Author's Note: remember distributed case: no decode

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

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

3.4 Test Case

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

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

3.5 Communication Collectives

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

The collective operation is essentially transposing information as seen in Figure 3.7.

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

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

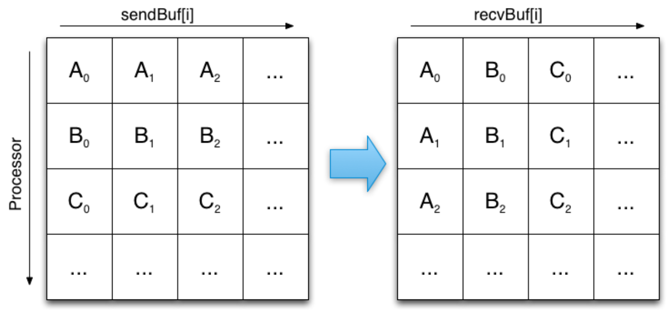


Figure 3.6: The MPI_Alltoall collective allows processors to interchange/transpose data by passing an equivalent number of bytes to every other processor.

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

3.5.1 Alltoallv

As a baseline for scaling we start with MPI_Alltoallv.

Author's Note: figure: alltoall visual

3.5.2 Isend/Irecv

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

Author's Note: figure: isend visual

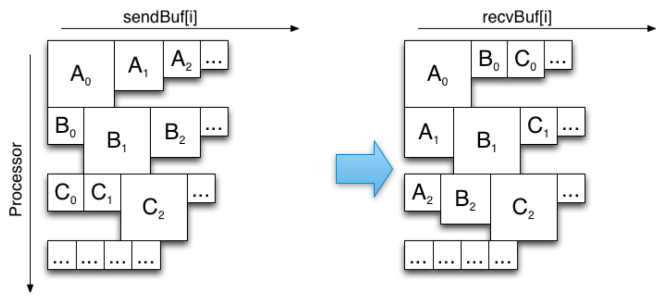


Figure 3.7: The MPI_Alltoallv collective compresses the interchange from MPI_Alltoall by allowing for variable message sizes between all processors. Assume message sizes are proportional to square size in figure. When packet sizes are null MPI_Alltoallv has undefined behavior.

Figure 3.7: `http://www.mcs.anl.gov/papers/P1699.pdf` observes that the zero-byte messages still add up due to processing required to analyze send-buffer and determine when connection is required.

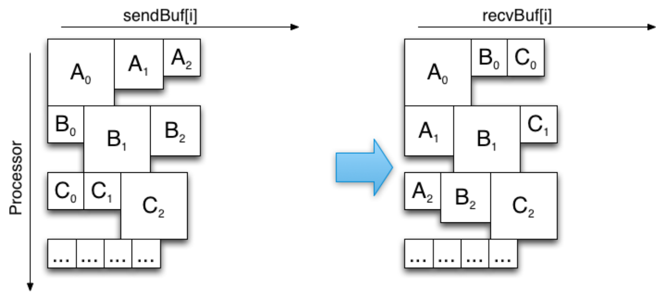


Figure 3.8: The "subset-to-subset" (MPI_Isend/MPI_Irecv) collective allows for variable message sizes, and truncates the number of connections between processors to only required connections.

3.5.3 No Decode

Author's Note: figure: per iteration stacked bar for n=50 and 16 processes to show cost of decode Author's Note: figure: algorithm for collective

Author's Note: figure: alltoall to isend improvement. justify comm_combo for up to 16 procs. Author's Note: figure: comm_combo gains Author's Note: figure: algorithm for collective

3.5.4 Immediate Isend on Encode

Author's Note: figure: algorithm for collective
Author's Note: back to section: figure: improvement on all CPU collectives (n=50)
Author's Note: table: show percentage of comm time for actual mpi time. busy network

can cause slower comm times. but the decode cost is gone. it can also be an issue if we have saturated comm pipes

3.6 CPU Scaling

Author's Note: Show the strong and weak scaling here

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

3.6.1 Strong Scaling

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

3.6.2 Weak Scaling

Weak scaling considers the amount of time for a fixed problem size per process and variable number of processors. That is to say, each processor has roughly the same amount of work, so as we scale to a large number of processors, changes in time will be the result of increased communication overhead.

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

Author's Note: Here and strong scaling: table showing the min and max Osize,Rsize

3.6.3 Bandwidth

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

The aggregate bandwidth reveals when processes saturate the interconnects.

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

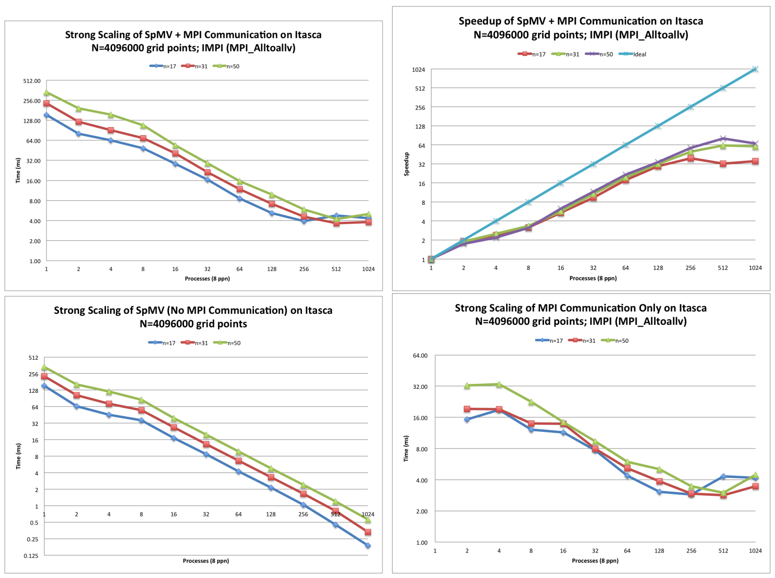


Figure 3.9: Strong scaling the distributed SpMV for $N = 4096000$ nodes (i.e., a 160^3 regular grid) and various stencil sizes. Here the MPI_Alltoallv collective operation is used. (Left) Strong scaling of SpMV (including cost of communication). (Center) Strong scaling of computation only. (Right) Strong scaling of communication only.

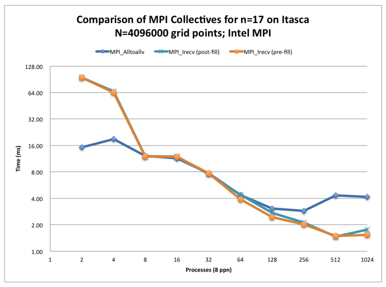


Figure 3.10: Scaling comparison of MPI_Alltoallv and two types of MPI_Isend/MPI_Irecv collectives: one with MPI_Irecv issued after filling the MPI_Isend send buffer (post-fill), and the other issued before filling the MPI_Isend buffer (pre-fill).

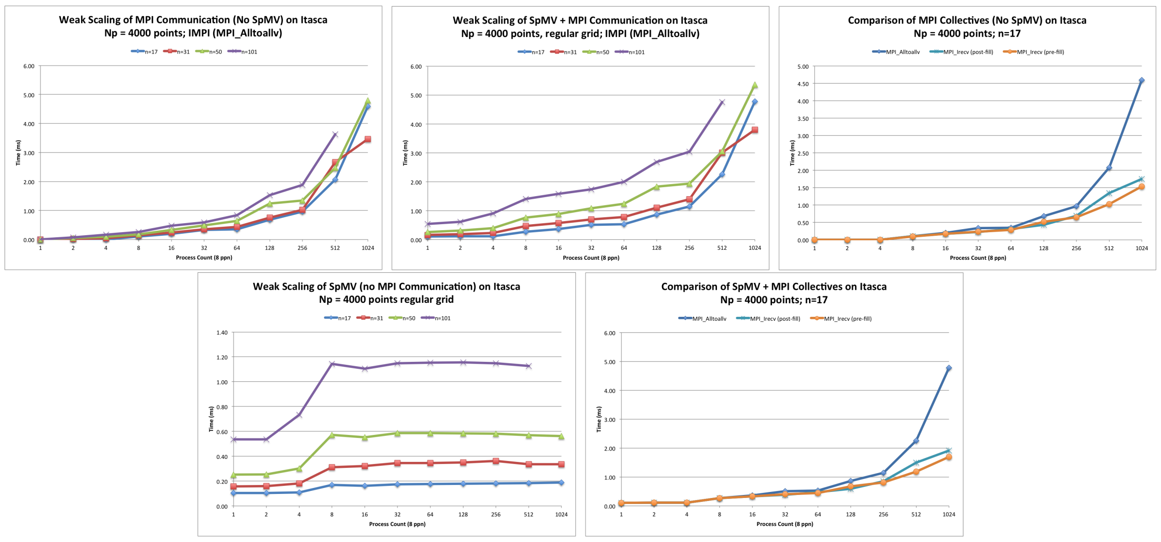
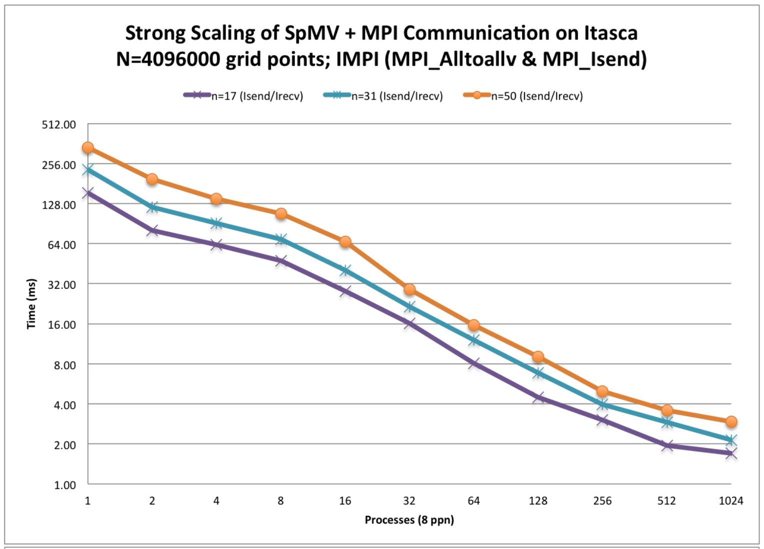


Figure 3.11: Weak scaling of the SpMV



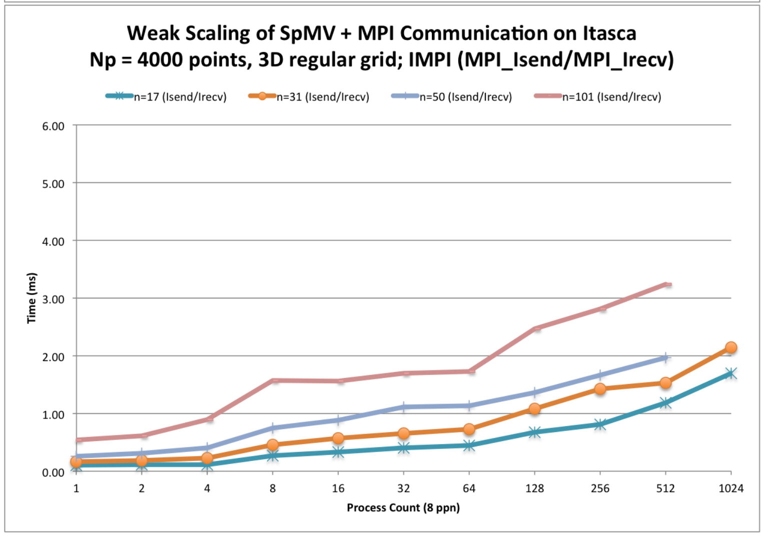


Figure 3.12: Scaling of SpMV with MPI_Isend/MPI_Irecv

CHAPTER 4

DISTRIBUTED GPU SPMV

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.

4.1 Overlapped Queues

4.2 Avoiding Copy Out

We can 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.

4.2.1 Avoiding Copy-Out on CPU

4.3 Scaling

We scale the SpMV across the GPUs on Cascade.

4.3.1 Fermi

4.3.2 Kepler

4.3.3 Shared K20s

BIBLIOGRAPHY

[1] HMPP Data Sheet. http://www.caps-entreprise.com/upload/article/fichier/64fichier1.pdf, 2009. 7

[2] Tesla kepler gpu accelerators. http://www.nvidia.com/content/tesla/pdf/Tesla-KSeries-Overview-LR.pdf, Oct 2012.

[3] AccelerEyes. *Jacket User Guide - The GPU Engine for MATLAB*, 1.2.1 edition, November 2009. 4, 5

[4] E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen. *LAPACK Users' Guide*. Society for Industrial and Applied Mathematics, Philadelphia, PA, third edition, 1999. 5

[5] Sylvie Barak. Gpu technology key to exascale says nvidia. http://www.eetimes.com/electronics-news/4230659/GPU-technology-key-to-exascale-says-Nvidia, November 2011. 2, 5

[6] R. K. Beatson, W. A. Light, and S. Billings. Fast Solution of the Radial Basis Function Interpolation Equations: Domain Decomposition Methods. *SIAM J. Sci. Comput.*, 22(5):1717–1740, 2000. 3

[7] Nathan Bell and Michael Garland. Implementing sparse matrix-vector multiplication on throughput-oriented processors. *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis - SC '09*, (1):1, 2009. 8

[8] Andreas Brandstetter and Alessandro Artusi. Radial Basis Function Networks GPU Based Implementation. *IEEE Transaction on Neural Network*, 19(12):2150–2161, December 2008. 4

[9] J. C. Carr, R. K. Beatson, B. C. McCallum, W. R. Fright, T. J. McLennan, and T. J. Mitchell. 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*, pages 119–ff, New York, NY, USA, 2003. ACM. 4

[10] Tom Cecil, Jianliang Qian, and Stanley Osher. Numerical Methods for High Dimensional Hamilton-Jacobi Equations Using Radial Basis Functions. *JOURNAL OF COMPUTATIONAL PHYSICS*, 196:327–347, 2004. 2

[11] A Corrigan and HQ Dinh. Computing and Rendering Implicit Surfaces Composed of Radial Basis Functions on the GPU. *International Workshop on Volume Graphics*, 2005. 4

[12] N Cuntz, M Leidl, TU Darmstadt, GA Kolb, CR Salama, M Böttinger, D Klimarechenzentrum, and G Hamburg. GPU-based Dynamic Flow Visualization for Climate Research Applications. *Proc. SimVis*, pages 371–384, 2007. 4

[13] Salvatore Cuomo, Ardelio Gallettiy, Giulio Giuntay, and Alfredo Staracey. Surface reconstruction from scattered point via rbf interpolation on gpu. *CoRR*, abs/1305.5179, 2013. 5

[14] E Divo and AJ Kassab. An Efficient Localized Radial Basis Function Meshless Method for Fluid Flow and Conjugate Heat Transfer. *Journal of Heat Transfer*, 129:124, 2007. 3, 4

[15] Natasha Flyer and Bengt Fornberg. Radial basis functions: Developments and applications to planetary scale flows. *Computers & Fluids*, 46(1):23–32, July 2011. 2

[16] Natasha Flyer and Erik Lehto. Rotational transport on a sphere: Local node refinement with radial basis functions. *Journal of Computational Physics*, 229(6):1954–1969, March 2010. 2

[17] Natasha Flyer and Grady B. Wright. Transport schemes on a sphere using radial basis functions. *Journal of Computational Physics*, 226(1):1059 – 1084, 2007. 2

[18] Natasha Flyer and Grady B. Wright. A Radial Basis Function Method for the Shallow Water Equations on a Sphere. In *Proc. R. Soc. A*, volume 465, pages 1949–1976, December 2009. 2

[19] R Hardy. Multiquadratic Equations of Topography and Other Irregular Surfaces. *J. Geophysical Research*, (76):1–905, 1971. 2

[20] L. Ivan, H. De Sterck, S. A. Northrup, and C. P. T. Groth. Three-Dimensional MHD on Cubed-Sphere Grids: Parallel Solution-Adaptive Simulation Framework. In *20th AIAA CFD Conference*, number 3382, pages 1325–1342, 2011. 13

[21] E J Kansa. Multiquadrics–A scattered data approximation scheme with applications to computational fluid-dynamics. I. Surface approximations and partial derivative estimates. *Computers Math. Applic*, (19):127–145, 1990. 2

[22] E J Kansa. 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):147–161, 1990. 2

[23] George Karypis and Vipin Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM Journal on Scientific Computing*, 20(1):359–392, 1999. 13

[24] Khronos OpenCL Working Group. *The OpenCL Specification (Version: 1.0.48)*, October 2009. 8

[25] G Kosec and B Šarler. Solution of thermo-fluid problems by collocation with local pressure correction. *International Journal of Numerical Methods for Heat & Fluid Flow*, 18, 2008. 4

[26] NVidia. *NVIDIA CUDA - NVIDIA CUDA C - Programming Guide version 4.0*, March 2011. 7

[27] Portland Group Inc. *CUDA Fortran Programming Guide and Reference*, 1.0 edition, November 2009. 8

[28] DA Randall, TD Ringler, and RP Heikes. Climate modeling with spherical geodesic grids. *Computing in Science & Engineering*, pages 32–41, 2002. 13

[29] Yousef Saad. *Iterative Methods for Sparse Linear Systems.* Society for Industrial Mathematics, second edition, 2003. 12

[30] J. Schmidt, C. Piret, B.J. Kadlec, D.A. Yuen, E. Sevre, N. Zhang, and Y. Liu. Simulating Tsunami Shallow-Water Equations with Graphics Accelerated Hardware (GPU) and Radial Basis Functions (RBF). In *South China Sea Tsunami Workshop*, 2008. 5

[31] J. Schmidt, C. Piret, N. Zhang, B.J. Kadlec, D.A. Yuen, Y. Liu, G.B. Wright, and E. Sevre. Modeling of Tsunami Waves and Atmospheric Swirling Flows with Graphics Processing Unit (GPU) and Radial Basis Functions (RBF). *Concurrency and Computat.: Pract. Exper.*, 2009. 4, 5

[32] C. Shu, H. Ding, and K. S. Yeo. 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):941 – 954, 2003. 2

[33] David Stevens, Henry Power, and Herve Morvan. An order-N complexity meshless algorithm for transport-type PDEs, based on local Hermitian interpolation. *Engineering Analysis with Boundary Elements*, 33(4):425 – 441, 2008. 4

[34] A. I. Tolstykh and D. A. Shirobokov. On using radial basis functions in a "finite difference mode" with applications to elasticity problems. In *Computational Mechanics*, volume 33, pages 68 – 79. Springer, December 2003. 2

[35] A.I. Tolstykh. On using RBF-based differencing formulas for unstructured and mixed structured-unstructured grid calculations. In *Proceedings of the 16 IMACS World Congress, Lausanne*, pages 1–6, 2000. 2

[36] J.S. Vetter, R. Glassbrook, J. Dongarra, K. Schwan, B. Loftis, S. McNally, J. Meredith, J. Rogers, P. Roth, K. Spafford, and S. Yalamanchili. Keeneland: Bringing heterogeneous GPU computing to the computational science community. *IEEE Computing in Science and Engineering*, 13(5):90–95, 2011. 8

[37] Richard Vuduc, James W Demmel, and Katherine A Yelick. Oski: A library of automatically tuned sparse matrix kernels. In *Institute of Physics Publishing*, 2005. 8

[38] M Weiler, R Botchen, S Stegmaier, T Ertl, J Huang, Y Jang, DS Ebert, and KP Gaither. Hardware-Assisted Feature Analysis and Visualization of Procedurally Encoded Multifield Volumetric Data. *IEEE Computer Graphics and Applications*, 25(5):72–81, 2005. 4

[39] Grady B. Wright. *Radial Basis Function Interpolation: Numerical and Analytical Developments.* PhD thesis, University of Colorado, 2003. 2

[40] Grady B. Wright, Natasha Flyer, and David A. Yuen. A hybrid radial basis function–pseudospectral method for thermal convection in a 3-d spherical shell. *Geochem. Geophys. Geosyst.*, 11(Q07003):18 pp., 2010. 2

[41] Rio Yokota, L.A. Barba, and Matthew G. Knepley. PetRBF — A parallel O(N) algorithm for radial basis function interpolation with Gaussians. *Computer Methods in Applied Mechanics and Engineering*, 199(25-28):1793–1804, May 2010. 4, 5