Chapter 1

GPU SpMV

1.1 Related Work

[**?**] [**?**] [**?**] etc.

1.2 GPGPU

GPGPU evolution

1.2.1 OpenCL

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

1.2.2 Hardware Layout

Modern GPUs have a memory hierarchy and hardware layout.

1.2.3 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.

1.3 Targeting the GPU

1.3.1 OpenCL

1.3.2 Naive Kernels

1.3.3 SpMV Formats/Kernels

1.4 Performance Comparison

1.4.1 Performance of Cosine CL vs VCL

1.4.2 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.

Chapter 2

Distributed Solver

Distributed application of RBF-FD requires three design decisions [**?**]. First, the problem is partitioned in some fashion to distribute work across multiple processes. Intelligent partitioning impacts load balancing of processors and the ratio of computation versus communication; imbalanced computation can result in excessive delay per iteration as some processors tackle larger problem sizes while others sit idle, waiting on information. Second, one must determine whether processes have access to all or a subset of node information, solution values, etc. and establish index mappings that translate between a local context and the global problem. Last but not least, the local ordering processor can re-order nodes locally in an effort to improve solver efficiency and local system conditioning. Node ordering is also significant in the future discussion of offloading computation to GPUs as it can help to minimize data transfer between CPU and GPU.

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

2.1 Partitioning

For ease of development and parallel debugging, partitioning is initially assumed to be linear within one physical direction (typically the x-direction). Figure 2.2 illustrates a partitioning of $N = 10,201$ nodes on the unit sphere onto four CPUs. Each partition, illustrated as a unique color, represents set $\mathcal{G}$ for a single CPU. Alternating representations between node points and interpolated surfaces illustrates the overlap regions where nodes in sets $\mathcal{O}$ and $\mathcal{R}$ (i.e., nodes requiring MPI communication) reside. As stencil size increases, the width of the overlap regions relative to total number of nodes on the sphere also increases.

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

The linear partitioning in Figure 2.2 was chosen for ease of implementation. Communication is limited for each processor to left and right neighbors only, which simplifies parallel debugging. This partitioning, however, does not guarantee properly balanced computational work-loads. Author's Note: Update w/ ParMETIS Other partitionings of the sphere exist but are not studied here because this paper's focus is neither on efficiency nor on selecting a partitioning strategy for maximum accuracy. Examples of alternative approaches include a cubed-sphere [**?**] or icosahedral geodesic grid [**?**], which can evenly balance the computational load across partitions. Other interesting partitionings can be generated with software libraries such as the METIS [**?**] family of algorithms, capable of partitioning and reordering directed graphs produced by RBF-FD stencils.

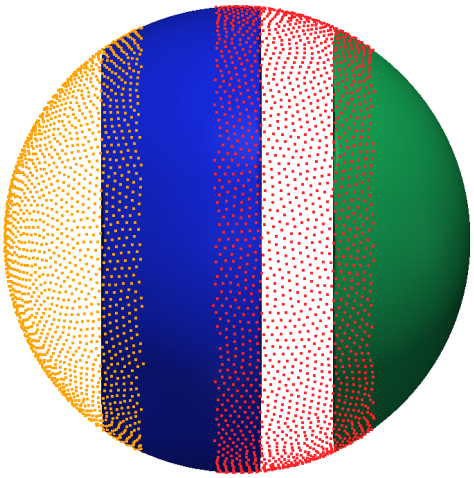

Figure 2.1: Partitioning of $N = 10,201$ nodes to span four processors with stencil size $n = 31$.

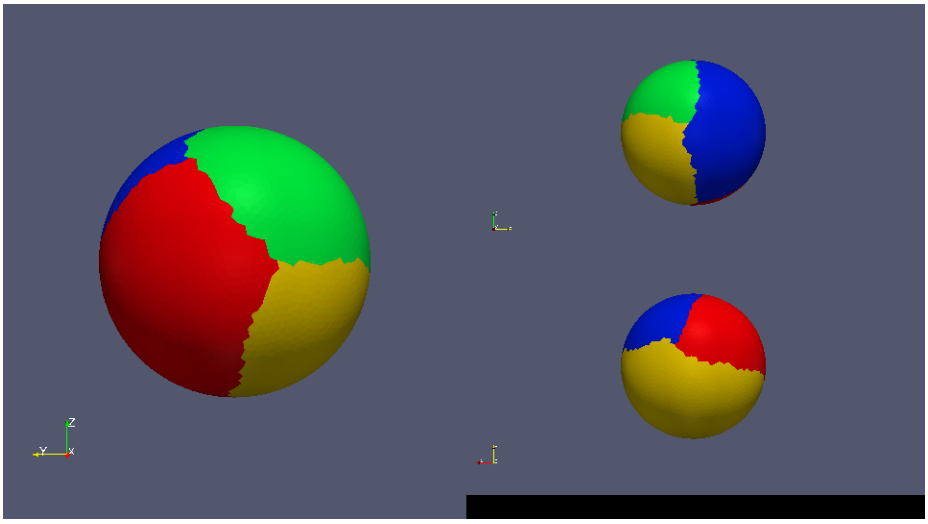

Figure 2.2: METIS partitioning of $N = 10,201$ nodes to span four processors with stencil size $n = 31$.

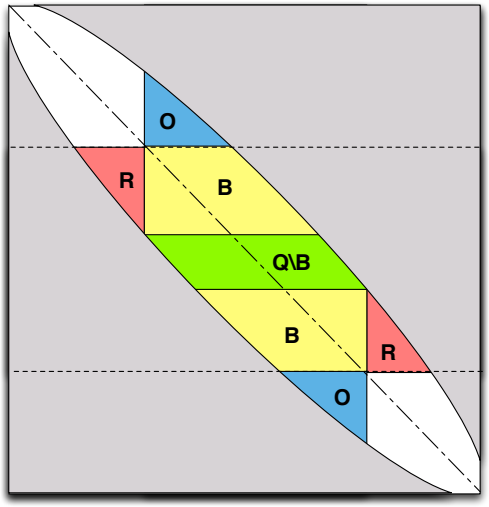

Figure 2.3: Decomposition for one processor selects a subset of rows from the DM. Blocks corresponding to node sets $\mathcal{Q}\backslash\mathcal{B}$, $\mathcal{O}$, and $\mathcal{R}$ are labeled for clarity.

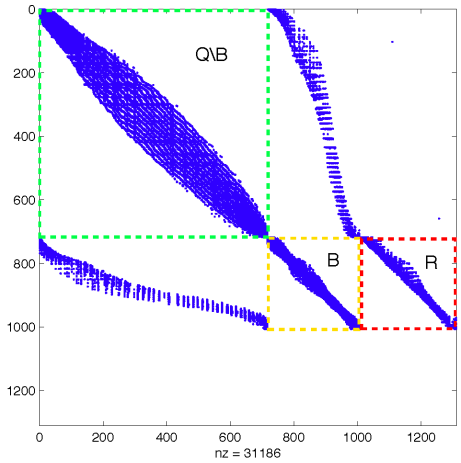

Figure 2.4: Spy of the sub-DM view on processor 2 of 4 from a METIS partitioning of $N = 10,201$ nodes with stencil size $n = 31$. Blocks are highlighted to distinguish node sets $\mathcal{Q}\backslash\mathcal{B}$, $\mathcal{O}$, and $\mathcal{R}$. Stencils involved in MPI communications have been permuted to the bottom of the matrix.

2.2 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

$\mathcal{G}$	: 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 2.1: Sets defined for stencil distribution to multiple CPUs

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

Figure 2.5 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, 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 2.5 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 $\mathcal{G}$,

$$\mathcal{G} = \{\mathcal{Q} \backslash \mathcal{B} \quad \mathcal{B} \backslash \mathcal{O} \quad \mathcal{O} \quad \mathcal{R}\}, \tag{2.1}$$

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 $\mathcal{R}$.

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

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

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

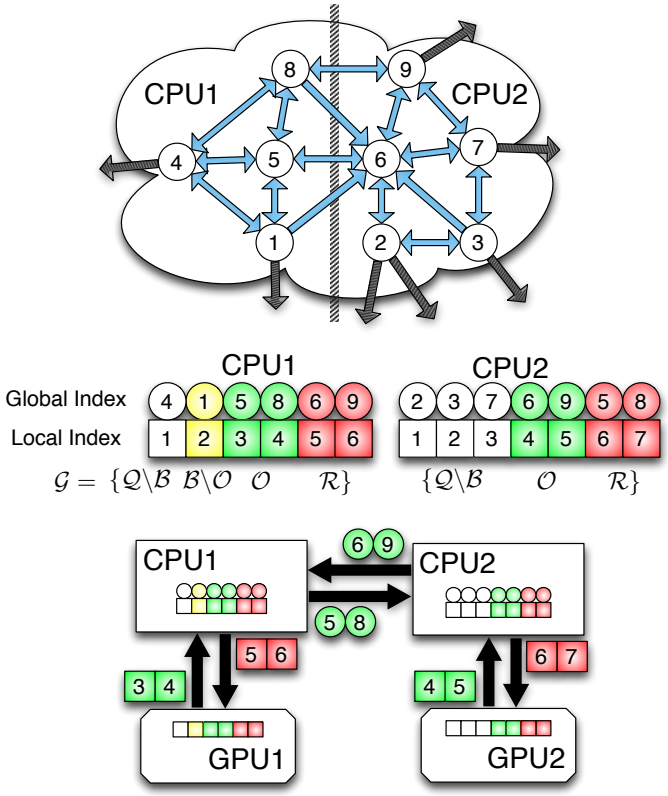

Figure 2.5: Partitioning, index mappings and memory transfers for nine stencils ($n = 5$) spanning two CPUs and two GPUs. Top: the directed graph created by stencil edges is partitioned for two CPUs. Middle: the partitioned stencil centers are reordered locally by each CPU to keep values sent to/received 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 2.1: $\mathcal{Q}\backslash\mathcal{B}$ is white, $\mathcal{B}\backslash\mathcal{O}$ is yellow, $\mathcal{O}$ is green and $\mathcal{R}$ is red.

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 2.5. Each GPU operates on the local indices ordered according to Equation (2.1). 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.

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.

2.3 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 2.7.

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.

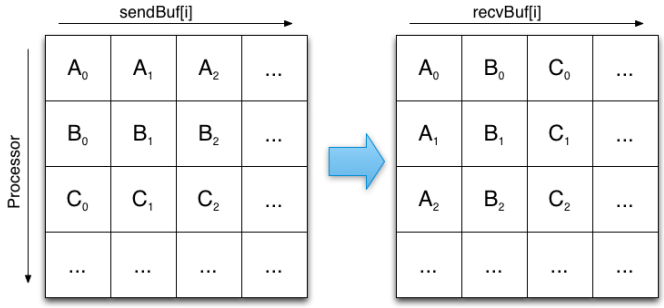

Figure 2.6: The MPI_Alltoall collective allows processors to interchange/transpose data by passing an equivalent number of bytes to every other processor.

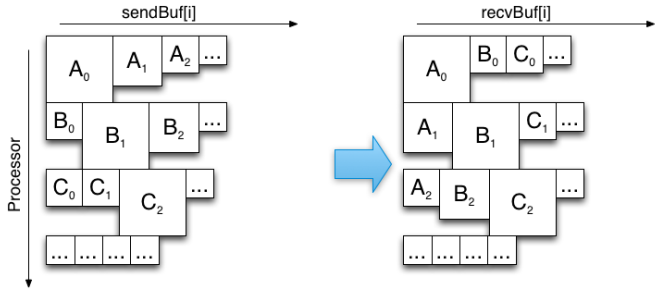

Figure 2.7: The MPI_Alltoallv collective compresses the interchange from MPI_Alltoall by allowing for variable message sizes between all processors. Assume message sizes are proportional to square size in figure.

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

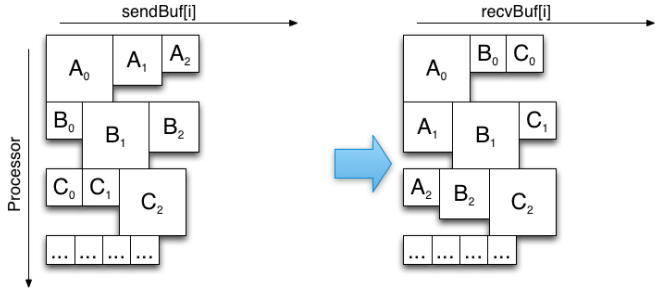

Figure 2.8: The MPI_Isend/MPI_Irecv collective allows for variable message sizes, and truncates the number of connections between processors to only required connections.

2.4 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.

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

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.

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

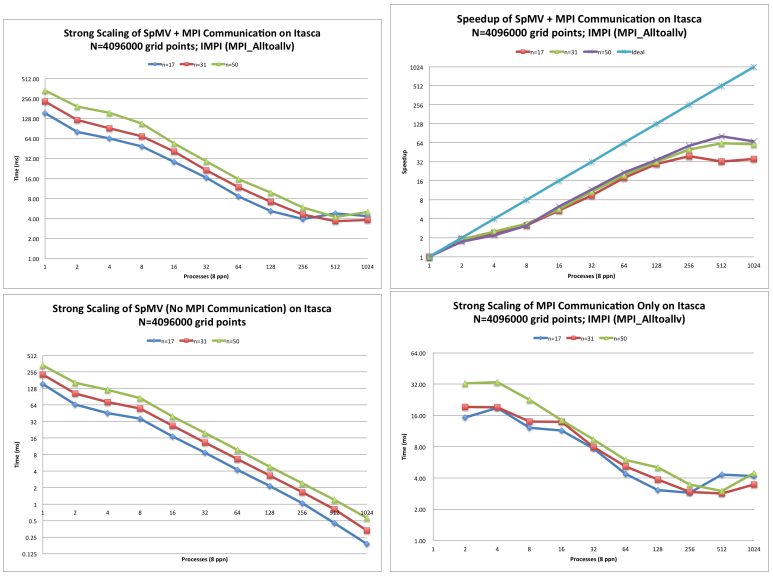

Figure 2.9: Strong scaling the distributed SpMV for $N = 4096000$ nodes (i.e., a 160^3 regular grid) and various stencil sizes. Here the MPI_Alltoallv collective operation is used. (Left) Strong scaling of SpMV (including cost of communication). (Center) Strong scaling of computation only. (Right) Strong scaling of communication only.

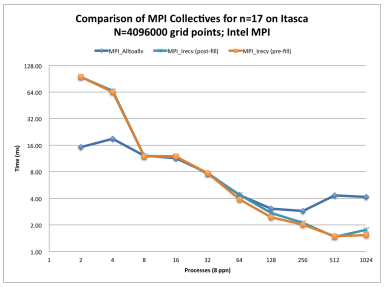

Figure 2.10: Scaling comparison of MPI_Alltoallv and two types of MPI_Isend/MPI_Irecv collectives: one with MPI_Irecv issued after filling the MPI_Isend send buffer (post-fill), and the other issued before filling the MPI_Isend buffer (pre-fill).

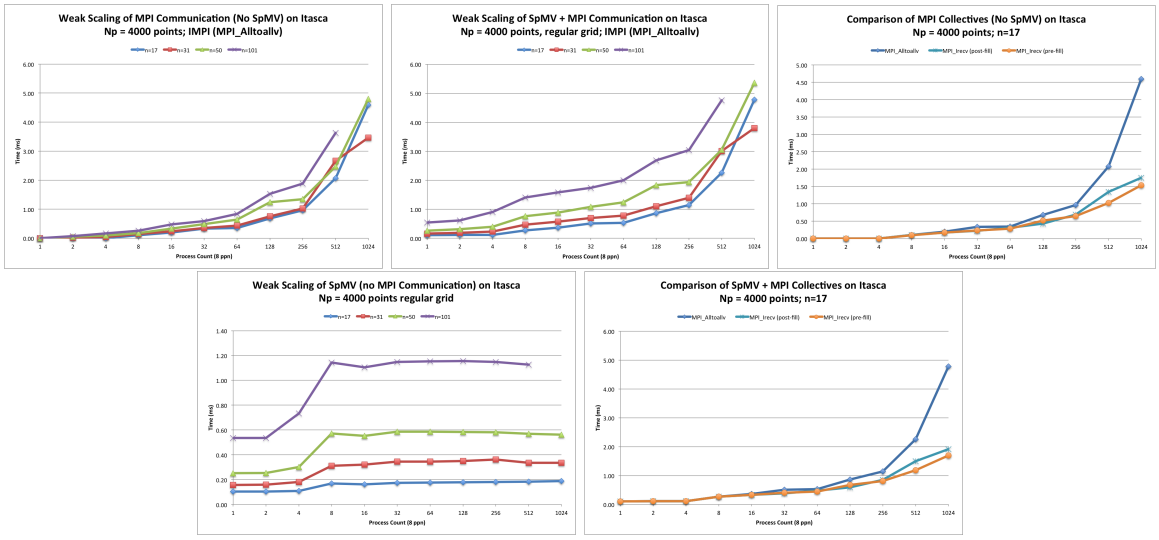

Figure 2.11: Weak scaling of the SpMV

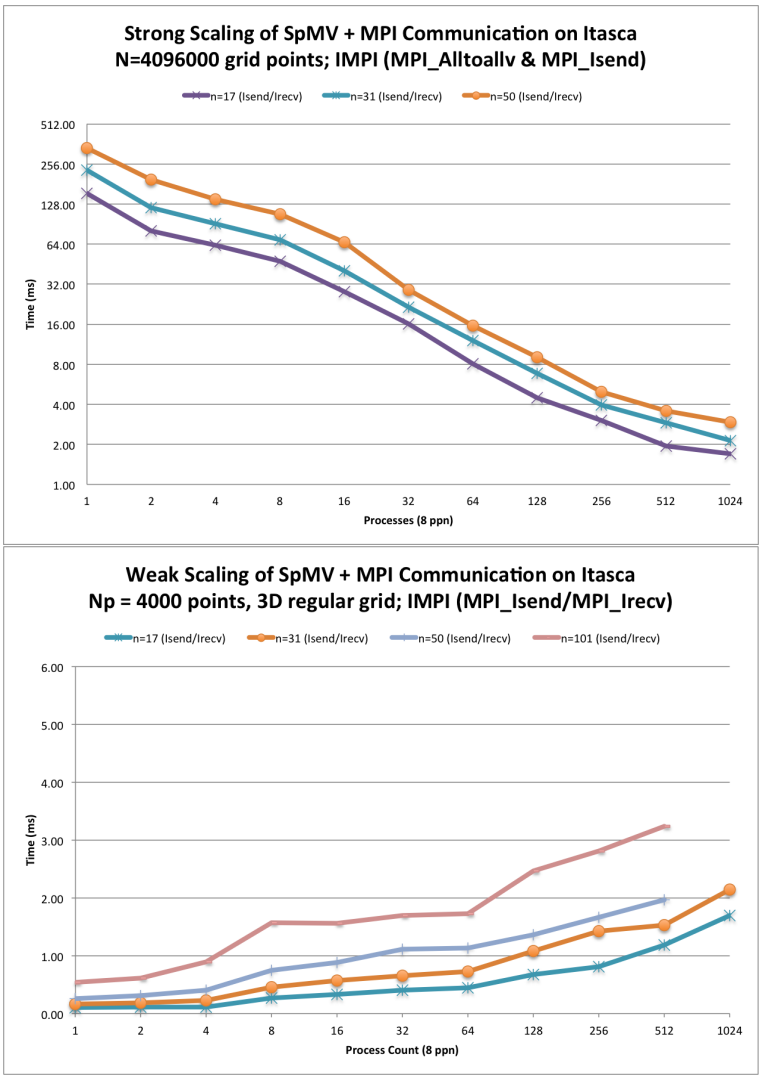

Figure 2.12: Scaling of SpMV with MPI_Isend/MPI_Irecv

Chapter 3

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.

3.1 Overlapped Queues

3.2 Avoiding Copy Out

3.2.1 Avoiding Copy-Out on CPU

3.3 Scaling

We scale the SpMV across the GPUs on Cascade.

3.3.1 Fermi

3.3.2 Kepler

3.3.3 Shared K20s