REDUCING COMMUNICATION BOTTLENECKS IN ITERATIVE SOLVERS

BY

SHELBY LOCKHART

DISSERTATION

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois Urbana-Champaign, 2023

Urbana, Illinois

Doctoral Committee:

Professor Luke N. Olson, Chair and Director of Research
Professor William D. Gropp
Associate Professor Edgar Solomonik
Dr. Katarzyna Świrydowicz, Pacific Northwest National Laboratory

Abstract

This work focuses on reducing communication bottlenecks in iterative solvers, specifically, communication bottlenecks associated with the advent of modern high performance computing architectures. Communication bottlenecks can be the result of irregular point-to-point communication or high synchronization costs across all participating processes. There are two ways to address these communication bottlenecks, namely, alterations to the parallel implementation of communication within a given operation or algorithmic changes to reduce communication in an algorithm by performing mathematically equivalent operations that require less communication.

A novel node-aware communication technique that takes advantage of the high-bandwidth interconnects and high CPU core counts on modern supercomputers is presented, which demonstrates improved performance for irregular point-to-point communication for both distributed inter-CPU and inter-GPU communication when communicating large data volumes. The technique is used in the design of a communication efficient *enlarged* conjugate gradient method implemented within a CPU-based parallel solver. Additionally, the technique is used to reduce communication costs in the unstructured mesh boundary exchanges in *MIRGE-Com*, a GPU-based codebase for solving the compressible Navier-Stokes equations for viscous flows, and the Euler equations for inviscid flows of reactive fluid mixtures within the Center for Exascale Enabled Scramjet Design.

Recent research introduced low synchronization orthogonalization routines within the context of generalized minimum residual methods, demonstrating the power of re-writing solvers to take advantage of mathematically equivalent operations that decompose into more distributed-computing friendly kernels. A performance study of these low synchronization orthogonalization routines extended to Anderson acceleration is presented, demonstrating a reduction in synchronization cost for the method in both distributed CPU and GPU computing environments. Furthermore, an analysis of recent Anderson acceleration variants, alternating Anderson acceleration and composite Anderson acceleration, and the extension of low synchronization orthogonalization techniques to composite Anderson acceleration is included. Importantly, considerations for the development of performant Anderson acceleration solvers on emerging supercomputer architectures are discussed.

In sum, this work contributes strategies for reducing communication bottlenecks in iterative solvers which take into consideration the underlying architecture of high performance computing systems. Techniques utilizing restructured message passing and updated mathematical algorithms which require less costly communication are presented.

To my family, my forever champions.

Acknowledgments

This dissertation would not have been possible without the guidance, feedback, and support of many people. Foremost, I would like to thank my advisor, Luke Olson, who provided me with research guidance and valuable feedback on scientific communication. My publications were greatly improved by his suggestions on plot formatting and figure design. Secondly, I would like to thank Bill Gropp for serving on my committee and providing helpful advice throughout my research. I would also like to thank my committee members, Edgar Solomonik and Katarzyna Świrydowicz, whose own research influenced this work, and whose feedback improved the quality of this dissertation.

I owe special thanks to the amazing women in STEM that have inspired me throughout my life. Notable amongst them is Amanda Bienz, who I would like to thank for being my officemate for so many years of my graduate school experience and being an unofficial advisor and mentor. Equally important is Carol Woodward, who I would like to thank for not only providing me with amazing internship experiences, but for being an amazing role model. This work would not have been possible without the support of these strong and intelligent women, and for that I am forever grateful.

Most importantly, I would like to thank my family and friends. I would like to thank Emma Skeels, for always being there for me, even from afar. I owe special thanks to my partner, Alexey Voronin, for being the most caring partner I could ever ask for. I was always met with his support and a hug during the most stressful times, which I know was not an easy feat for a fellow graduate student, and for which I do not think I will ever be able to adequately thank him.

I would like to thank my sister, Jessica Lockhart, for being my life-long role model. Throughout my entire life, she served as a source of inspiration. Since I was young, I wanted to be just like her: smart and driven. As an adult, I continue to watch her thrive within her professional journey, always inspiring me to push harder within my own work.

Last, but certainly not least, I would like to thank my parents, Celena and Jesse Lockhart, whose love knows no bounds. There was never a point in my life where I felt my dreams were too big or there was something I could not achieve, and that is due to their unwavering support. To them, I owe everything.

Portions of this research are part of the Blue Waters sustained-petascale computing project, which is supported by the National Science Foundation (awards OCI0725070 and ACI-1238993) and the state of Illinois. Blue Waters is a joint effort of the University of Illinois at Urbana-Champaign and its National Center for Supercomputing Applications. This material is based in part upon work supported by the Department of Energy, National Nuclear Security Administration, under award number DE-NA0002374.

Portions of this material are based in part upon work supported by the Department of Energy, National Nuclear Security Administration, under Award Number DE-NA0003963.

Portions of this work were supported by the Scientific Discovery through Advanced Computing (SciDAC) project "Frameworks, Algorithms and Scalable Technologies for Mathematics (FASTMath)," funded by the U.S. Department of Energy Office of Advanced Scientific Computing Research.

Portions of this work were performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under contract DE-AC52-07NA27344. Lawrence Livermore National Security, LLC. LLNL-PROC-827159.

Table of Contents

Chapter 1: Introduction

Iterative solvers play a critical role in everyday society, forming the core of all large-scale physics-based simulations such as the modeling of subsurface flows, climate change phenomena, flight simulations, hydrodynamics, pandemic-related research, nuclear energy, etc.... The iterative solvers used for these simulations span multiple classes of numerical methods which demonstrate a variety of performance bottlenecks on large-scale parallel systems. Notably, most of the solvers experience significant performance limitations due to communication bottlenecks, or the overhead of moving data between distinct compute units (processes). These bottlenecks on current supercomputer architectures create a limitation on reaching theoretical peak performance, but more importantly, they are an impediment on the time-to-solution for simulations.

As we enter the era of exascale computing, supercomputers are exhibiting increasingly higher computational throughput due to the inclusion of multiple GPUs per node, a trend extending to future leadership-class supercomputers. These GPUs can operate concurrently on much higher data volumes than the previous generations of CPU-only machines. As a result, iterative solvers that can be redesigned to operate on larger blocks of data are being favored over more classical numerical solvers. However, redesigning solvers to take advantage of the high computational throughput of GPUs does not reduce or eliminate existing communication overhead. In fact, it increases the number of potential data flow paths for performing inter-process communication. Moreover, as the speed at which computation can be performed goes down, the relative cost of communicating data goes up.

1.1 Communication in Iterative Solvers

A significant performance limitation for iterative solvers on large-scale, distributed parallel computers can be attributed to communication bottlenecks, often due to a lack of computational work compared to communication overhead or the performance of mathematical operations that require synchronization amongst all participating processes [1, 2]. These communication bottlenecks are attributed to two types of communication on distributed systems, namely:

Irregular point-to-point : inter-process communication occurring via point-to-point messages, for which communication patterns are dependent upon input data, and

Global reductions : inter-process communication requiring the synchronization of all processes.

Bottlenecks stemming from irregular point-to-point communication typically occur within the context of sparse data operations, such as sparse matrix-vector multiplication, sparse matrix-matrix multiplication, sparse matrix-dense matrix multiplication, etc..., and unstructured mesh boundary exchanges. An example of this communication structure is given in Figure 1.1, where processes spanning four nodes need to communicate with processes both on-node and off-node in a non-structured pattern. Global reduction bottlenecks occur as the result of mathematical

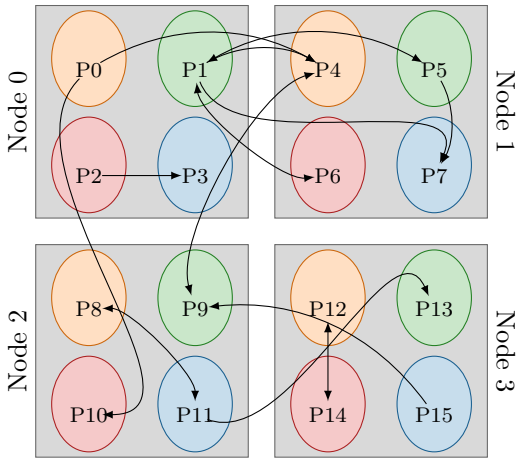


Figure 1.1: Irregular point-to-point communication.

operations that require values from all participating processes, e.g. inner products and norm calculations. An example of the resulting communication structure is given in Figure 1.2 where every process is contributing to a computation, but then the result of that computation needs to be passed back to every process. Hence, this creates a synchronization amongst all processes.

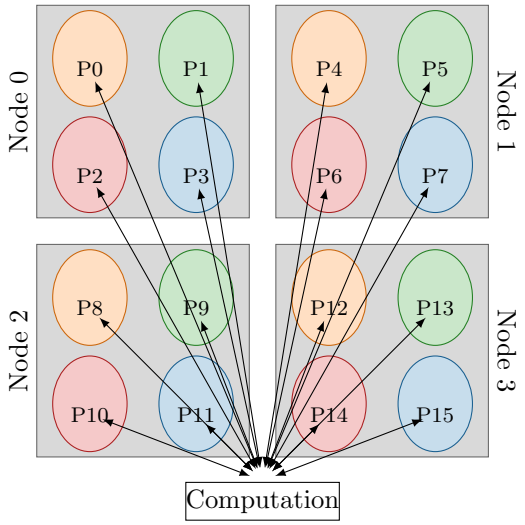


Figure 1.2: Global reduction communication.

Reducing and removing communication bottlenecks can be done in two ways: message passage restructuring or altering the parallel implementation of communication within a given operation and redesigning algorithms such that they are mathematically equivalent yet require less communication. Message passage restructuring typically results in communication-avoiding algorithms, such as communication-avoiding Krylov methods [3, 4], communication-avoiding sparse matrix-matrix multiplication [5], pipelined methods which overlap communication and computation [6], and most recently node-aware communication within sparse matrix operations [7]. Additionally, various partitioning techniques can result in message passage restructuring that

reduces the overall volume and or time required for point-to-point communication, such as hypergraph partitionings [8, 9] and 2.5D or 3D matrix partitionings for distributed general sparse matrix-matrix multiplication (SpGEMM) [10, 11, 12, 13]. Redesigning algorithms to be mathematically equivalent yet requiring less communication overall typically takes the form of algorithms that delay communication at the cost of performing more computation. This is true for s-step Krylov methods [14, 15], for recent low synchronization orthogonalization routines [16, 17, 18], and for the iterated Gauss-Seidel generalized minimum residual method [19].

1.2 Modern Supercomputer Architectures

Many current large-scale supercomputers consist of heterogeneous nodes containing multiple GPUs connected to a single CPU per socket with two sockets per node. In the case of the Lassen

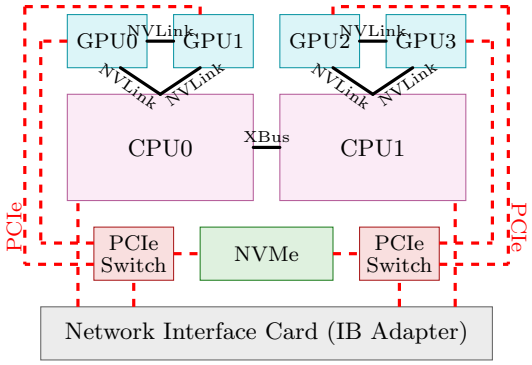


Figure 1.3: Lassen compute node.

supercomputer, each socket consists of a single IBM Power9 CPU connected to two NVIDIA V100 GPUs [20] (see Figure 1.3), while the Summit supercomputer has a single IBM Power9 CPU connected to three NVIDIA V100 GPUs [21]. For both machines, each CPU has 20 available cores, and CPUs and GPUs are connected via NVLink. Furthermore, CPUs are connected directly to the Infiniband (IB) adapter through PCIe lanes, while GPUs are connected to the IB adapter via a PCIe switch system connecting the GPUs to NVMe and the adapter. Nodes are connected via Mellanox EDR 100G InfiniBand in a non-blocking fat tree topology. Upcoming Department of Energy exascale machines, Frontier [22] and El Capitan[1], will have nodes with a similar structure to those found in Lassen and Summit. However, these compute nodes will consist of a single socket housing an AMD EPYC CPU connected to four AMD Instinct 250X GPUs via AMD Infinity Fabric with a Slingshot network.

Both current and future supercomputers boast heterogeneous architectures with multiple paths for data movement between two GPUs. Two connected GPUs either exchange data directly or stage through the host CPU by first copying data to CPU memory, then transferring data from the

[1] https://www.llnl.gov/news/el-capitan-testbed-systems-rank-among-top-200-worlds-most-powerful-computers

local CPU to the host CPU of the receiving GPU, and finally copying received data to the destination GPU. The process of staging data through the host CPU can be used for any set of communicating GPUs independent of their relative locations. However, device-aware data movement paradigms, such as CUDA-aware MPI using GPUDirect [23] on Lassen, remove the necessity of copying data to the host CPU and allow data to be pulled directly from device memory, even in the case of inter-node data transfers. The addition of device-aware technologies increases the number of potential data movement paths necessitating the use of robust performance modeling to determine communication bottlenecks, as well as, design optimal communication strategies.

1.3 Overview of Contributions

Despite hardware advancements and the computational power of supercomputers reaching an all-time high, communication bottlenecks remain one of the key hurdles in achieving peak performance of iterative solvers within large-scale scientific simulations. This dissertation investigates methods for reducing the communication bottlenecks in iterative solvers associated with irregular point-to-point communication, within sparse matrix operations and unstructured mesh boundary exchanges, and global reductions within factorization updating routines. The main contributions of this thesis include the following:

Split node-aware communication

In Chapter 3, a novel node-aware communication scheme is presented which takes into consideration the total data volume being communicated by individual nodes, allowing each node to decide the most efficient way to communicate its inter-node data. Furthermore, modeling and profiling results extend the method from inter-CPU communication to inter-GPU communication.

Communication efficient *enlarged* conjugate gradients

Chapter 4 includes a detailed performance study and analysis of the *enlarged* conjugate gradients method, noting the performance bottleneck at scale of point-to-point communication within the block vector operations. Introducing Split communication into these operations reduces the performance bottleneck of communication in the *enlarged* conjugate gradient algorithm and results in large speedups over standard communication for various test cases.

Efficient large-scale unstructured-mesh boundary exchanges

Predictive science simulations, such as those conducted within the Center for Exascale-enabled Scramjet Design, typically require computing on large-scale partitions of leadership-class supercomputers (up to and exceeding 200 nodes) due to the large unstructured meshes on which the simulations are run. Boundary exchanges for unstructured meshes can create performance bottlenecks for large-scale simulations due to the irregular point-to-point structure of the communication. In Chapter 5, we use Split communication as the foundation for an optimized unstructured-mesh boundary exchange communication

scheme that effecitvely collapses the communication costs of boundary exchanges at scale within the Center for Exascale-enabled Scramjet Design's codebase.

Scalable Anderson acceleration solvers

Anderson acceleration, while an effective method for improving the convergence of fixed point and Picard iterations, requires solving a least squares problem at each iteration. This results in poor performance at scale due to the necessity of using a stable orthogonalization routine, such as modified Gram-Schmidt which requires a high number of global reductions.
In Chapter 6, low synchronization orthogonalization routines are introduced into Anderson acceleration, effectively reducing the number of costly global reductions required per iteration. In addition, a thorough performance study and analysis of multiple Anderson acceleration variants is conducted in the development of a scalable Anderson acceleration solver for emerging architectures.

Chapter 2: Background

In this chapter, we summarize the necessary background for reducing communication bottlenecks in iterative solvers through message passage restructuring and algorithm redesigning. We present the max-rate model for modeling data movement, an overview of existing node-aware communication strategies which reduce performance overheads in irregular point-to-point communication, and low synchronization QR-updating routines which reduce performance overheads related to synchronizations in global communication.

2.1 Modeling Data Movement

Throughout the thesis, the *max-rate* model is used as the basis for communication modeling [24]. The max-rate model is an improvement to the standard postal model of communication, accounting for injection limits into the network. The traditional postal model estimates the cost of communicating a message between two symmetric multiprocessing (SMP) nodes as

$$T = \alpha + \beta \cdot s \tag{2.1}$$

where α is the latency, β is the per-byte transfer cost, and s is the number of bytes being communicated. The max-rate model adds parameters for injection-bandwidth limits and the number of actively communicating processes, resulting in the following time estimation,

$$T = \alpha \cdot m + \max\left(\frac{\texttt{ppn} \cdot s}{R_N}, \frac{s}{R_b}\right) \tag{2.2}$$

where α is again the latency, m is the maximum number of messages sent by a single process on a given node, s is the maximum number of bytes sent by a single process on a given SMP, `ppn` is the number of processes per node, R_N is the rate at which a network interface card (NIC) can inject data into the network, and R_b is the rate at which a process can transport data. When $\texttt{ppn} \cdot R_b < R_N$, this model reduces to the postal model.

For inter-CPU communication, additional improvements are available to the max-rate model within the context of irregular point-to-point communication. Additional hardware and software overhead penalties are represented in the LogP model [25], which is extended to include long message costs in the LogGP model [26]. Additionally, models for queue search times and network contention have been shown to be important for accurately predicting performance of point-to-point communication [27]. These models provide penalty parameters that increase the accuracy of standard communication models and motivate design decisions within the context of node-aware communication techniques, discussed in more detail in Section 2.2 and Chapter 3.

The max-rate model also applies to inter-GPU communication [28]. Here, the noted difficulty in reaching injection bandwidth limits with inter-GPU communication is due to the low number of

communicating GPUs per node. Additionally, for large message counts, performance benefits are observed [28] when staging communication between GPUs through host CPUs.

2.2 Node-Aware Communication

Sparse matrix-vector multiplication (SpMV), defined as

$$A * v \rightarrow w \tag{2.3}$$

with $A \in \mathbb{R}^{m \times n}$ and v, $w \in \mathbb{R}^n$, is a common kernel in sparse iterative methods. It is known to lack strong scalability in a distributed memory parallel environment, a problem stemming from low computational requirements and the communication overhead associated with applying standard communication techniques to sparse matrix operations.

There have been a number of techniques designed to reduce the communication costs of the parallel SpMV. Graph partitioning algorithms produce efficient data layouts that often lead to improved parallel partitions and system loads which reduce time spent in communication [29, 30, 31, 32]. Additionally, topology-aware task mapping addresses communication overhead via accurately mapping parallel partitions to supercomputer nodes [33, 34, 35]. While these methods can result in reduced communication times, they are often accompanied by costly set-up times or more complex matrix distributions. In the case of node-aware communication techniques, A, v, and w are generally considered to be partitioned row-wise across p processes with contiguous rows stored on each process (see Figure 2.1). In addition, the rows of A on each process are split into 2 blocks, namely on-process and off-process. The on-process block is the diagonal block of columns corresponding to the on-process portion of rows in v and w, and the off-process block contains the nonzeros of matrix A associated with the non-local rows of v and w stored off-process. This splitting is common practice, as it differentiates between the portions of a SpMV that require communication with other processes.

A common approach to a SpMV is to compute the local portion of the SpMV with the on-process block of A while messages are exchanged to attain the off-process portions of v necessary for the local update of w. While this allows for the overlap of some communication and computation, it requires the exchange of many point-to-point messages, which still creates a large communication overhead (see Figure 2.2). MPI + X, or hierarchical parallelism, strategies can help mitigate this communication overhead by pairing a single MPI rank with multiple threads for local computation. These strategies allow for efficient overlap of communication and computation, yet they limit the number of available communicating processes per node [36, 37, 38]. While this can reduce the overall amount of time required for the SpMV over methods that use flat-MPI standard communication techniques, it typically does not reduce communication overhead as much as node-aware communication techniques with flat-MPI, as detailed in [7].

The inefficiency of the standard communication approach is attributed to two redundancies shown in Figure 2.3. First, many messages are injected into the network from each node. Some

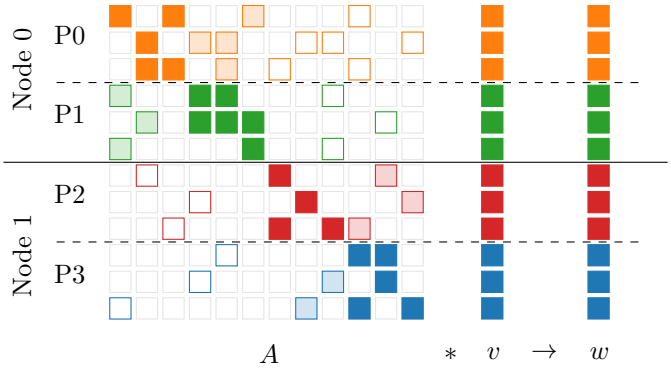


Figure 2.1: Communication and partitioning of a SpMV, $A * v \to w$. With $n = 12$, matrix A and vectors v and w are partitioned across two nodes and four processors (P0, P1, P2, and P3), indicated by the colors orange, green, red, and blue. A solid block, ■, represents the portion of the SpMV requiring only on-process values from v. A shaded block, ▢, represents the portion of the SpMV requiring on-node but off-process communication of values from v, and the outlined blocks, □, require values of v from processors off-node.

nodes are sending multiple messages to a single destination process on a separate node, creating a redundancy of messages. Secondly, processes send the necessary values from their local portion of v to any other process requiring that information for its local computation of w. However, the same information may already be available and received by a separate process on the same node, creating a redundancy of data being sent through the network.

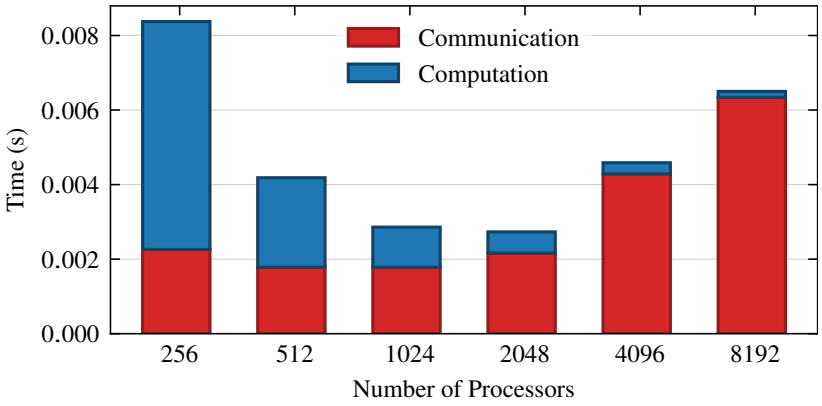


Figure 2.2: The time required for a single SpMV, split into communication and computation, for Example 4.1 run on Blue Waters with 16 processes per node.

Node-aware communication techniques [7, 39] mitigate these issues in both SpMVs and sparse matrix-matrix multiplication (SpGEMMs) by considering node topology to further break down the off-process block into vector values of v that require on- or off-node communication; this decomposition is shown in Figure 2.1. As a result, costly redundant messages are traded for faster, on-node communication, resulting in two different multi-step schemes, namely 3-step and 2-step.

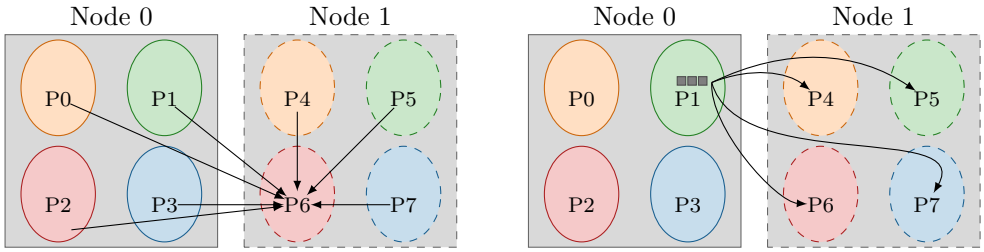


Figure 2.3: Standard communication. On the left, Node 0 injects multiple messages into the network, all to P6 on Node 1. On the right, P1 sends all highlighted data to multiple processes on Node 1, leading to redundant messages.

2.2.1 3-Step

3-Step node-aware communication, first introduced in [39], eliminates both redundancies in standard communication by gathering all necessary data to be sent off-node in a single buffer. Pairing all processes with a receiving process on distinct nodes ensures efficiency of the method by making sure every process remains active throughout the communication scheme. First, all messages sent to a separate node are gathered in a buffer by the single process associated with the node. Secondly, this process sends the data buffer to the paired process on the receiving node. Thirdly, the paired process on the receiving node redistributes the data to the correct destination processes on-node. An example of these steps is outlined in Figure 2.4.

As noted in [39], the method can be extended to include further breakdown of data exchanges to include intra-socket data communication before the intra-node communication phase. However, there are minimal performance benefits in extending the communication strategy throughout the entire node hierarchy for CPU to CPU communication. Instead, this strategy was adopted for GPU to GPU communication in [40], where the full hierarchy of the node is utilized to achieve optimal performance due to the fast data transfer rates of socket-level GPU interconnects on Summit [21].

2.2.2 2-Step

When communicating high data volumes between nodes, 3-Step communication can see limitations as the single buffer communicating data grows extremely large, thus motivating the design of a 2-Step node-aware technique as in [7]. The 2-Step technique eliminates the redundancy of sending duplicate data through the network, but does not reduce the redundancy of multiple messages being sent between nodes. In 2-Step, *each* process exchanges information needed by the receiving node with their paired process directly, followed by the receiving node redistributing the messages on-node, as shown in Figure 2.5. Overall, the total number of bytes communicated with 3-Step and 2-Step communication techniques is the same, but the number and size of inter-node messages differs.

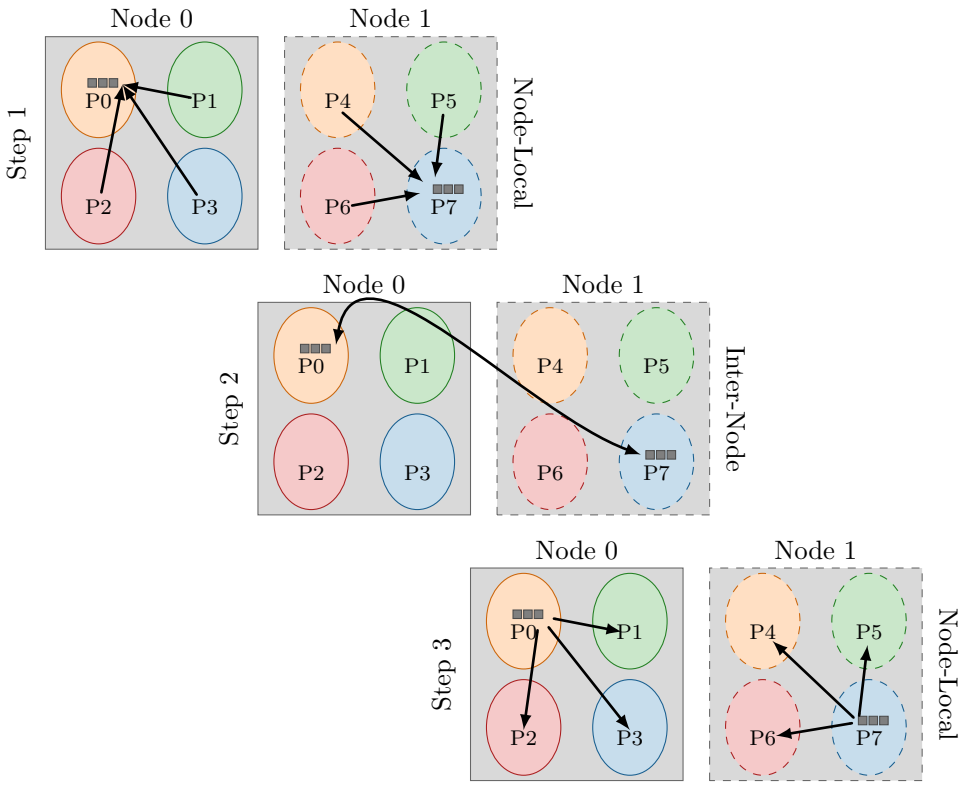


Figure 2.4: 3-Step node-aware. In Step 1, all data on Node 0 that needs to be sent to Node 1 is collected in a buffer on P0, the process paired to send and receive from Node 1. In Step 2, P0 sends this buffer from Node 0 to P7, the receiving process on Node 1. In Step 3, P7 redistributes the data to the correct receiving processes on Node 1.

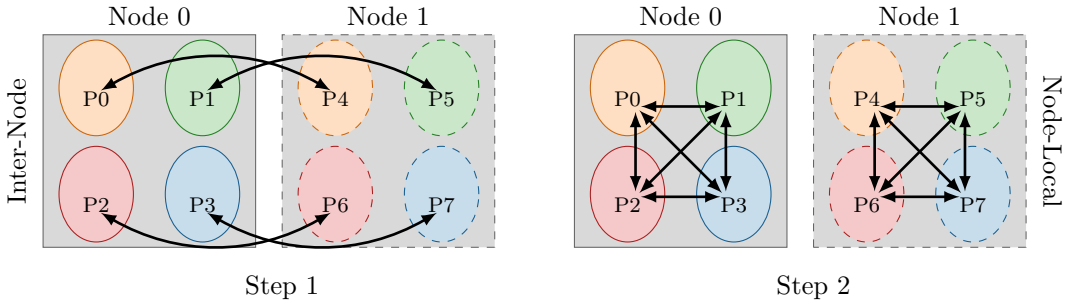


Figure 2.5: 2-Step node-aware. Each process on Node 0 is paired with a receiving process on Node 1. In Step 1, each process on Node 0 sends the data needed by any process on Node 1 to its paired process on Node 1. Here, P0 is sending to P4, P1 to P5, P2 to P6, and P3 to P7. In Step 2, each process on Node 1 redistributes the data received from Node 0 to the destination on Node 1.

2.2.3 Node-Aware Communication Models

The *max-rate* model presented in (2.2) is used to quantify the efficiency of node-aware communication throughout the remainder of the thesis. For clarity, all modeling parameters referenced throughout the remainder of the thesis are defined in Table 2.1.

parameter	description	first use
p	number of processes	§4.3
$\texttt{nnz}$	number of nonzeros in A	§4.3.3
α	network latency	(2.2)
s	maximum number of bytes sent by a process	(2.2)
m	maximum number of messages sent by a process	(2.2)
$\texttt{ppn}$	processes per node	(2.2)
$\texttt{gpn}$	GPUs per node	(3.3)
R_N	network injection rate (B/s)	(2.2)
R_b	network rate (B/s)	(2.2)
α_ℓ	on-node latency	(2.4)
$R_{b,\ell}$	on-node rate (B/s)	(2.4)
$s_{\texttt{proc}}$	maximum number of bytes sent by a process	(2.6)
$s_{\texttt{node}}$	maximum number of bytes injected by a node	(2.6)
$m_{\texttt{proc}\to\texttt{node}}$	maximum number of nodes to which a processor sends	(2.6)
$m_{\texttt{node}\to\texttt{node}}$	maximum number of messages between two nodes	(2.5)
$s_{\texttt{node}\to\texttt{node}}$	maximum size of a message between two nodes	(2.5)

Table 2.1: Modeling parameters. Standard communication modeling parameters are listed in the top portion of the table, with node-aware communication specific modeling parameters listed in the bottom portion.

In the case of on-node messages, the injection rate is not present and the max-rate model reduces to the standard postal model for communication

$$T = \alpha_\ell \cdot m + \frac{s}{R_{b,\ell}}, \tag{2.4}$$

where α_ℓ is the *local* or on-node latency and $R_{b,\ell}$ is the rate of sending a message on-node.

In [7], the max-rate model is extended to 2-step and 3-step communication by splitting the model into inter-node and intra-node components. For 3-step, the communication model becomes

$$T_{\text{total}} = \underbrace{\alpha \cdot \frac{m_{\texttt{node}\to\texttt{node}}}{\texttt{ppn}} + \max\left(\frac{s_{\texttt{node}}}{R_N}, \frac{s_{\texttt{proc}}}{R_b}\right)}_{\text{inter-node}} + \underbrace{2 \cdot \left(\alpha_\ell \cdot (\texttt{ppn} - 1) + \frac{s_{\texttt{node}\to\texttt{node}}}{R_{b,\ell}}\right)}_{\text{intra-node}} \tag{2.5}$$

where $m_{\texttt{node}\to\texttt{node}}$ is the maximum number of messages communicated between any two nodes and $s_{\texttt{node}\to\texttt{node}}$ is the size of messages communicated between any two nodes.

For 2-step, this results in

$$T_{\text{total}} = \underbrace{\alpha \cdot m_{\texttt{proc}\to\texttt{node}} + \max\left(\frac{s_{\texttt{node}}}{R_N}, \frac{s_{\texttt{proc}}}{R_b}\right)}_{\text{inter-node}} + \underbrace{\alpha_\ell \cdot (\texttt{ppn} - 1) + \frac{s_{\texttt{proc}}}{R_{b,\ell}}}_{\text{intra-node}} \tag{2.6}$$

where $s_{\texttt{node}}$ and $s_{\texttt{proc}}$ represent the maximum number of bytes injected by a single NIC and

communicated by a single process from an SMP node, respectively, and $m_{\texttt{proc}\to\texttt{node}}$ is the maximum number of nodes with which any process communicates.

The latency to communicate between nodes, α, is often much higher than the intra-node latency, α_ℓ, thus motivating a multi-step communication approach. In a 2-step method, having every process on-node communicate data minimizes the constant factor $\max\left(\frac{s_{\texttt{node}}}{R_N}, \frac{s_{\texttt{proc}}}{R_b}\right)$, which depends on the maximum amount of data being communicated to a separate node by a single process. In practice, a 3-step method often yields the best performance for a parallel SpMV since the amount of data being communicated by a single process is often small. As a result moving the data to be communicated off-node into a single buffer minimizes the first term in (2.5). These multi-step communication techniques minimize the amount of time spent in inter-node communication. We extend this idea to develop a new node-aware communication technique in Chapter 3.

2.3 Low Synchronization Gram-Schmidt QR Factorizations

This section provides the necessary background for the portion of the thesis addressing communication bottlenecks resulting from high numbers of synchronizations in Gram-Schmidt QR updating routines (Chapter 6). There exist several methods for computing the QR factorization of a set of vectors,

$$X = QR, \tag{2.7}$$

where X is an $n \times k$ matrix with linearly independent columns, Q is a set of k orthogonal vectors, and R is an upper triangular $k \times k$ matrix.

Householder transformations, Givens rotations, and Gram-Schmidt procedures are the most common strategies used for performing QR factorizations [41, Chapter 3]. In distributed computing environments, Householder-based methods are typically favored for situations where an entire factorization needs to be computed, due to their superior stability over Gram-Schmidt procedures [42, 43]. However, as noted in [16] and demonstrated in [44, 45], when updating a given QR factorization, Householder-based methods require a higher number of synchronizations. In this work, we consider methods derived from the Gram-Schmidt procedure as they form the basis for low synchronization orthogonalization routines [16]. Specifically, we focus on the process of updating a given QR factorization to include a vector q, illustrated in Figure 2.6, where $[X, q] = \hat{Q}\hat{R}$. Column-oriented algorithms are presented as opposed to those that proceed row-wise, as these are better suited to a low-synchronization formulation and promote data-locality for cache-memory access both for parallel computing on CPUs and GPUs [46]. Additionally, throughout this section and the remainder of the thesis, matrix entries are referenced via subscripts with 0-based indexing. For example, $M_{0,0}$ refers to the first row, first column of a matrix M, and slices of a matrix are inclusive, i.e. $M_{:,0:k-1}$ refers to all rows of the matrix and columns $0, \ldots, k-1$.

The classical Gram-Schmidt (CGS) process updates a given QR factorization with the vector q by applying the projection, $P = I - QQ^T$, to q then normalizing q — summarized in Algorithm 2.1. While this process is attractive in a parallel computing environment, due to the fact that P can be

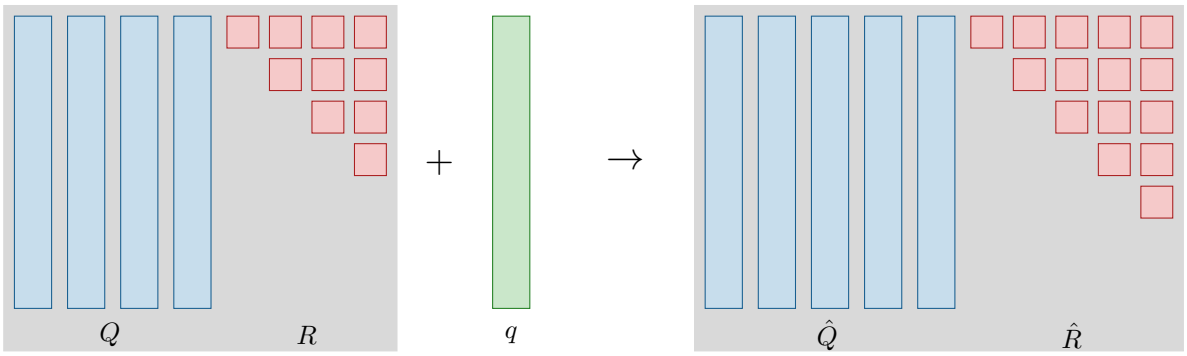


Figure 2.6: Provided a QR factorization with matrices Q and R and a vector, q, update the QR factorization to inclue q, producing a new QR factorization comprised of $\hat{Q}$ and $\hat{R}$.

Algorithm 2.1: Update QR using CGS

Input: Q, R, q and k
Output: Q, R

1 $R_{:k-1,k} \leftarrow Q^T q$ → Global Reduction
2 $q \leftarrow q - QR_{:k-1,k}$
3 $R_{k,k} \leftarrow \|q\|_2$ → Global Reduction
4 $Q_{:,k} \leftarrow q/R_{k,k}$

applied with a single reduction on Line 1 of Algorithm 2.1, this algorithm is highly unstable and exhibits a loss of orthogonality dependent on $\mathcal{O}(\varepsilon)\kappa^2(X)$ where ε is machine precision, or the relative error associated with representing a number within a floating-point system, and $\kappa^2(X)$ is the square of the condition number of X [47, 48]. A common solution for the instability of CGS is to apply modified Gram-Schmidt (MGS) instead. This process applies the rank–1 elementary projection matrices, $I - Q_{:,j}Q_{:,j}^T$, $j = 0, \ldots, k-1$, one for each of the k linearly independent columns of Q, to the vector q then normalizes q — summarized in Algorithm 2.2. Performing the

Algorithm 2.2: Update QR using MGS

Input: Q, R, q and k
Output: Q, R

1 **for** $j = 0, \ldots, k-1$
2 $\quad R_{j,k} \leftarrow Q_{:,j}^T q$ → Global Reduction
3 $\quad q \leftarrow q - R_{j,k}Q_{:,j}$
4 $R_{k,k} \leftarrow \|q\|_2$ → Global Reduction
5 $Q_{:,k} \leftarrow q/R_{k,k}$

Gram-Schmidt process in this way effectively reduces the loss of orthogonality to $\mathcal{O}(\varepsilon)\kappa(X)$ [49]. However, applying the projections sequentially requires k dot products, one for each of the columns, resulting in communication bottlenecks when performed in a parallel distributed environment [50]. New variants of CGS and MGS have been introduced to improve instability and parallel performance issues caused by these methods. For both CGS and MGS, these low synchronization

methods result from introducing a correction matrix, T into the projection operators, resulting in new projection operators of the form $P = I - QTQ^T$ — application and construction of these new projectors is discussed below.

2.3.1 ICWY Modified Gram Schmidt

The inverse compact WY MGS representation was recently introduced in [16]. The compact WY representation relies on a triangular correction matrix T, which contains a strictly lower triangular matrix L. One row or block of rows of L is computed at a time in a single global reduction. Each row

$$L_{k-2,0:k-2} = (Q^T_{:,0:k-2} Q_{k-2})^T \tag{2.8}$$

is obtained within a single step. The associated orthogonal projector is based on developments in [51], presented in [16], and can be written as the following

$$P = I - Q_{0:k-2} T_{0:k-2,0:k-2} Q^T_{:,0:k-2}. \tag{2.9}$$

Here, the correction matrix is defined as

$$T_{0:k-2,0:k-2} = (\, I + L_{k-1} \,)^{-1}. \tag{2.10}$$

The implied triangular solve requires an additional $(k-1)^2$ flops at iteration $k-1$ and thus leads to a slightly higher operation count compared to the original MGS algorithm. The operation $Q^T_{:,0:k-2} Q_{:,k-2}$ increases ICWY-MGS complexity by kn^2 for an overall complexity of $\mathcal{O}(kn^2)$, and reduces synchronizations from $k-1$ at update k to 1 when also lagging normalization.

Algorithm 2.3: Update QR using ICWY MGS

Input: Q, R, T, q and k
Output: Q, R, T

1 $T_{k-2,0:k-2} \leftarrow Q^T_{:,0:k-2} Q_{:,k-2}$ → Delayed Reduction
2 $R_{0:k-2,k-1} \leftarrow Q^T_{:,0:k-2} q$ → Global Reduction
3 $T_{k-2,k-2} \leftarrow 1$
4 $R_{0:k-2,k-1} \leftarrow T^{-1}_{0:k-2,0:k-2} R_{0:k-2,k-1}$
5 $q \leftarrow q - Q_{:,0:k-2} R_{0:k-2,k-1}$
6 $R_{k-1,k-1} \leftarrow \|q\|_2$ → Global Reduction
7 $Q_{:,k-1} \leftarrow q / R_{k-1,k-1}$

Here, we present a two-reduction variant of the ICWY-MGS algorithm. The formation of the correction operator and the matrix R can be merged on Line 2. With the inclusion of the normalization at the end of the algorithm, this results in two synchronizations per application of the QR updating algorithm.

In the original presentation of ICWY MGS in [16], the normalization step is lagged and merged

into a single reduction. Only one global reduction is required per iteration, and the amount of inter-process communication does not depend on the number of rank–1 projections $I - Q_{:,j}\, Q_{:,j}^T$ applied at each iteration. The required number of dot products per iteration is effectively reduced to one for GMRES and s–step Krylov solvers [16, 18]. The inverse compact WY algorithm maintains $\mathcal{O}(\varepsilon)\kappa(X)$ loss of orthogonality.

2.3.2 CGS-2

In the case of CGS, CGS-2 (classical Gram-Schmidt with reorthogonalization) corrects the projection by reorthogonalizing the vectors of Q and thereby reduces the loss of orthogonality to $\mathcal{O}(\varepsilon)$ [48, 52]. The form of the correction matrix for this algorithm was derived in [16].

The CGS-2 algorithm is simply CGS with reorthogonalization, which, unlike CGS, is known to improve large cancellation errors and maintain numerical stability [51, 53, 54, 55]. Reorthogonalizing the vectors in Q updates the associated orthogonal projector to include a correction matrix, T, although this matrix may not be explicitly formed in practice.

In Appendix 1 of [16], the form of the projection and correction matrices is derived within the context of DCGS-2 (CGS-2 with delayed reorthogonalization), however, the matrices remain the same and are given by

$$P = I - Q_{0:k-2} T_{0:k-2,0:k-2} Q_{:,0:k-2}^T, \tag{2.11}$$

$$T_{0:k-2,0:k-2} = I - L_{0:k-2,0:k-2} - L_{0:k-2,0:k-2}^T. \tag{2.12}$$

Here, L contains the same information as in the ICWY-MGS case, namely each row of L is generated by the reorthogonalization of the vectors within Q before updating the factorization with an additional vector.

Algorithm 2.4 details the process of updating a QR factorization with an additional vector without explicitly forming the correction matrix. Reorthogonalization happens explicitly on Lines 3 and 4 before being added back into the matrix R on Line 5. Overall, this process requires three reductions to be performed per iteration—two for the orthogonalization and reorthogonalization of q against each vector in Q and one for the final normalization of q. The amount of computation per iteration has increased to approximately $2kn$ or $\mathcal{O}(kn)$. While the amount of computation is still on the same order as MGS, the workload has approximately doubled, and, fortunately, there are no additional storage requirements.

Furthermore, the normalization of the added vector can be lagged, depending upon which solver or application the orthogonalization routine is being used within. DCGS-2, or CGS with delayed reorthogonalization, is based on the delayed CGS-2 algorithm introduced in [56]. In DCGS-2, the reorthogonalization of the vectors in CGS-2 is delayed to the subsequent iteration. In the original derivation, it is noted that this process is tantamount to updating the QR factorization with a CGS vector, and a deteriorated loss of orthogonality (between $\mathcal{O}(\varepsilon)$–$\mathcal{O}(\varepsilon)\kappa^2(X)$, that of CGS-2 and CGS) is often observed.

Algorithm 2.4: Update QR using CGS2

Input: Q, R, q and k
Output: Q, R

1 $s \leftarrow Q_{:,0:k-2}^T q$ → Global Reduction
2 $y \leftarrow q - Q_{:,0:k-2} s$
3 $z \leftarrow Q_{:,0:k-2}^T y$ → Global Reduction
4 $q \leftarrow y - Q_{:,0:k-2} z$
5 $R_{0:k-2,k-1} \leftarrow s + z$
6 $R_{k-1,k-1} \leftarrow \|q\|_2$ → Global Reduction
7 $Q_{:,k-1} \leftarrow q / R_{k-1,k-1}$

A stable variant of DCGS-2 was derived in [17] by exploiting the form of the correction matrix T and introducing a normalization lag and delayed reorthogonalization for use within GMRES. The symmetric correction matrix $T_{0:k-2,0:k-2}$ is the same as in the context of CGS-2, namely

$$T_{0:k-2,0:k-2} = I - L_{0:k-2,0:k-2} - L_{0:k-2,0:k-2}^T. \tag{2.13}$$

When the matrix $T_{0:k-2,0:k-2}$ is split into the pieces $I - L_{0:k-2,0:k-2}$ and $L_{0:k-2,0:k-2}^T$, then applied across two iterations of the DCGS-2 algorithm coupled with lagging the normalization of the added vector, the resulting loss of orthogonality is $\mathcal{O}(\varepsilon)$ in practice. Performing DCGS-2 in this way decreases the number of reductions to one per iteration because the reorthogonalization is performed "on-the-fly" and essentially operates a single iteration behind. It is also noted in [17] that the final iteration of GMRES requires an additional synchronization due to the required final normalization. Lagging the reorthogonalization and introducing the operation $Q_{:,0:k-3}^T Q_{:,k-2}$, makes this method's computational complexity $\mathcal{O}(kn^2)$, the same as ICWY-MGS.

Chapter 3: Split Node-Aware Communication

Portions of this chapter appear in the papers "Characterizing the Performance of Node-Aware Strategies for Irregular Point-to-Point Communication on Heterogeneous Architectures", to appear in Parallel Computing [57] and "Performance Analysis and Optimal Node-Aware Communication for Enlarged Conjugate Gradient Methods" published in ACM Transactions on Parallel Computing [58].

Throughout this chapter and the remainder of the thesis, results are presented for the system Cray MPI implementation on the now retired Blue Waters system. Blue Waters was a Cray XE/XK machine at the National Center for Supercomputing Applications (NCSA) at University of Illinois. Blue Waters [59, 60] contained a 3D torus Gemini interconnect; each Gemini consisted of two nodes. The complete system contained 22 636 XE compute nodes, each with two AMD 6276 Interlagos processors, and additional XK compute nodes unused in my results.

Results are additionally presented for the Spectrum MPI implementation on the Lassen supercomputer [20]. In [28], it is shown that Lassen and Summit [21] demonstrate similar performance using Spectrum MPI (there, the MPI implementation is optimized for use on the two DOE machines), therefore results for a single machine are provided. Lassen, a 23-petaflop IBM system at Lawrence Livermore National Laboratory, consists of 795 nodes connected via a Mellanox 100 Gb/s Enhanced Data Rate (EDR) InfiniBand network. Each node on Lassen is dual-socket with 44 IBM Power9 cores and 4 NVIDIA Volta GPUs. All machine specifications relevant to presented results are summarized in Table 3.1.

	Blue Waters	Lassen
Interconnect	3D Torus i9.6 GB/s Gemini	Mellanox 100 Gb/s EDR Infiniband
CPU Architecture	AMD 6276 Interlagos	IBM Power9
CPU Cores per Node (Available)	16	40
GPU Architecture	–	NVIDIA Volta
Available Nodes	22 636	795
Number of Sockets per Node	2	2
Number of CPUs per Socket	1	1
Number of GPUs per Socket	–	2

Table 3.1: Machine specifications.

Moreover, each of the presented model parameters presented in this section and modelling sections further in the thesis is the result of ping-pong and node-pong timings collected through BenchPress[2], a node architecture-aware library used for benchmarking data movement performance on large-scale systems. The ping-pong and node-pong tests are performed for 1000 iterations and averaged; each model parameter is then given by a linear least-squares fit to the collected data.

[2] https://github.com/bienz2/BenchPress

3.1 Introduction

Modern parallel supercomputers exhibit increasingly higher computational throughput due to the inclusion of multiple GPUs per node — refer back to Section 1.2. These GPUs operate on much higher data volumes concurrently than previous CPU-only clusters, yet the issue of communication bottlenecks persists and is exacerbated in a multi-node–multi-GPU setting. While the high computational intensity of modern supercomputers is driving a new era of applications, the volume of data communicated between compute units has also increased, creating new obstacles for data movement performance.

Here, we focus on irregular point-to-point communication, which generates performance bottlenecks in parallel solvers and graph algorithms due to the prevalence of sparse matrix operations and unstructured mesh computations [1, 61]. We characterize the performance of various irregular point-to-point communication strategies using MPI within homogeneous and heterogeneous compute environments via performance modeling, leading to the development of a novel node-aware communication strategy and suggests the extension of node-aware communication strategies for inter-CPU communication onto heterogeneous architectures.

Node-aware communication schemes utilize the relative location of communicating processes and exchange costly data flow paths for lower cost alternatives [39]. While there are many potential paths for data movement on heterogeneous architectures, we consider the communication paths available via the MPI API and only consider device specific optimizations, such as utilizing CUDA Multi-Process Service (MPS) to allow multiple MPI ranks to copy data from a single GPU, for the purpose of comparison.

In Section 3.5.1, we present modeling parameters for all potential data flow paths between CPUs and GPUs, which are then used within performance models to predict the cost of various node-aware communication schemes when used within inter-CPU and inter-GPU communication in Section 3.5.2. Models are first validated via comparison against the performance of communication within a sparse matrix-vector product, then further modeling results are shown suggesting that for large message counts, optimal performance is achieved when GPU data is staged through a host process and split across multiple processes before communicating through the network.

Furthermore, Section 3.6 provides a study of the techniques modeled in Section 3.5.2 when applied to the irregular point-to-point communication patterns in distributed sparse matrix-vector multiplication (SpMV), further validating model predictions.

3.2 Benchmarking Point-to-Point Communication on Supercomputers

Node-aware communication strategies exchange some number of inter-node messages for a higher on-node message volume due to the higher performance costs associated with sending inter-node messages [39], particularly on more traditional networks, e.g., the now retired BlueWaters system [62]. Yet this is not always the case for more recent interconnects, such as on Lassen, which

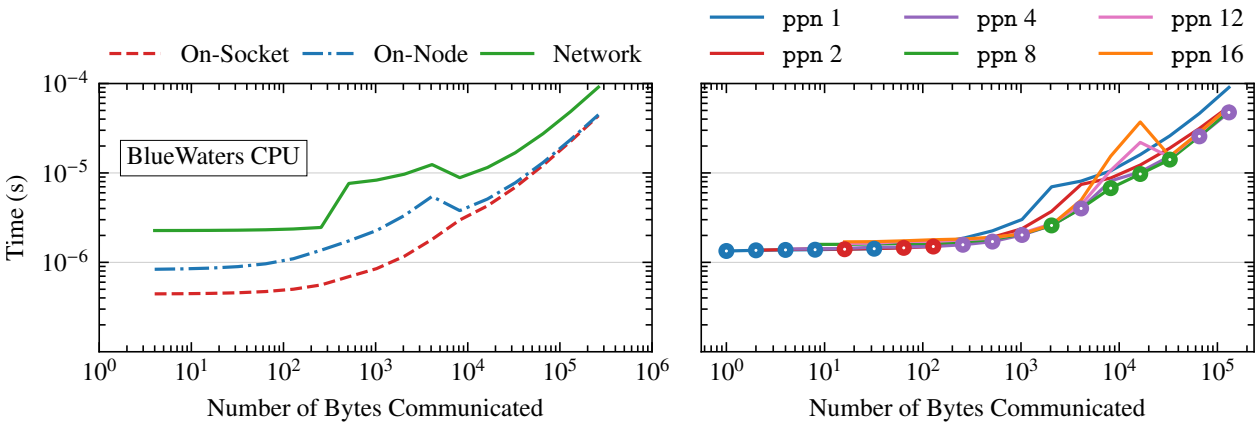


Figure 3.1: Blue Waters inter-CPU ping-pong (left) and inter-CPU node-pong (right), minimum times circled.

shows varying performance for inter-node versus intra-node communication depending on the amount of data being communicated. Hence, in this section, we view the effects placement of data and amount of data being communicated has on performance for Blue Waters and Lassen inter-CPU communication, as well as, Lassen inter-GPU communication.

The left plot of Figure 3.1 depicts the results of an inter-CPU ping-pong test on Blue Waters, i.e. the amount of time required to send data between two processes distinguishing between where the two processes are physically located: on the same socket, the same node and separate sockets, and separate nodes requiring network communication. We see that as the number of bytes communicated between two processes increases, it becomes increasingly important whether those two processes are located on the same socket, node, or require communication through the network. Inter-node communication is fastest when message sizes are small, but intra-node communication is clearly faster with the time being dependent on how physically close the processes are located. For instance, when two processes are on the same socket, communication is faster than when they are on the same node, but different sockets. In addition to the importance of the placement of two communicating processes, the total message volume and number of actively communicating processes plays a key role in communication performance. While it is costly for every process on a node to send 10^5 bytes, there are performance beneifts when splitting a large communication volume across all processes on a node, depicted in the right plot of Figure 3.1. Here, Blue Waters sees only modest performance improvements when splitting large messages across multiple processes due to the low number of available processes per node.

Similar to Blue Waters ping-pong performance, Lassen demonstrates the fastest communication performance for intra-socket messages, but cross-socket on-node communication and inter-node communication performance differs — as seen in the left plot of Figure 3.2 which shows inter-CPU ping-pong test results for Lassen. In addition to network communication being faster than on-node communication for large message sizes, the CPUs used in current supercomputers have high numbers of cores (for example, the IBM Power9 has 40 available cores on Lassen), making splitting

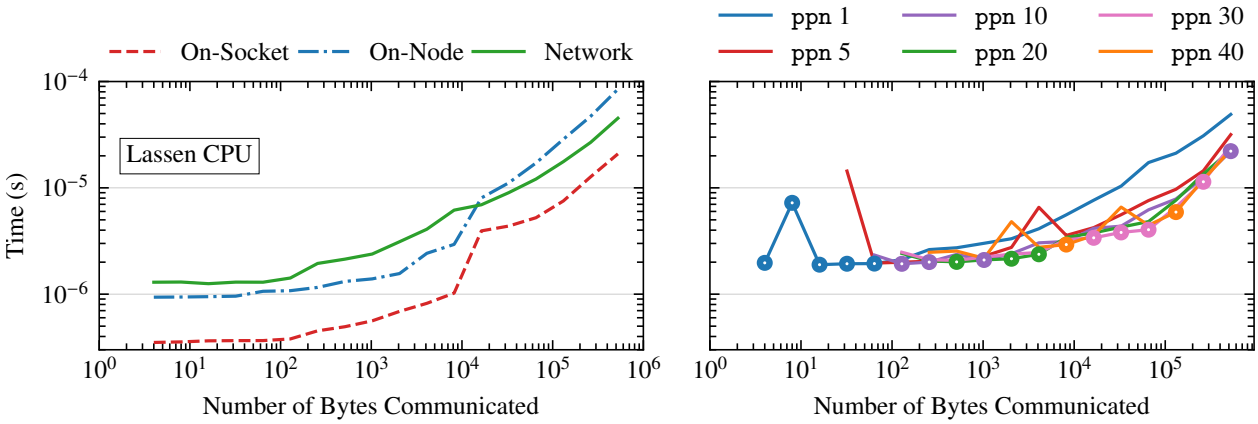


Figure 3.2: Lassen inter-CPU ping-pong (left) and inter-CPU node-pong (right), minimum times circled.

large data volumes across all available cores more performant than when the entire data volume is sent by a single process. This can be seen in the inter-CPU node-pong results–the right plot of Figure 3.2–which presents the amount of time required to send various amounts of data between two distinct nodes when splitting the data across `ppn` processes per node.

Additionally, using the device-aware technologies available through the Spectrum MPI implementation on Lassen, we can benchmark inter-GPU communication. Inter-GPU communication is much more costly than inter-CPU communication, but exhibits the inverse relationship of inter-node and intra-node communication observed for inter-CPU performance. In Figure 3.3, the left plot depicts the results of an inter-GPU ping-pong.

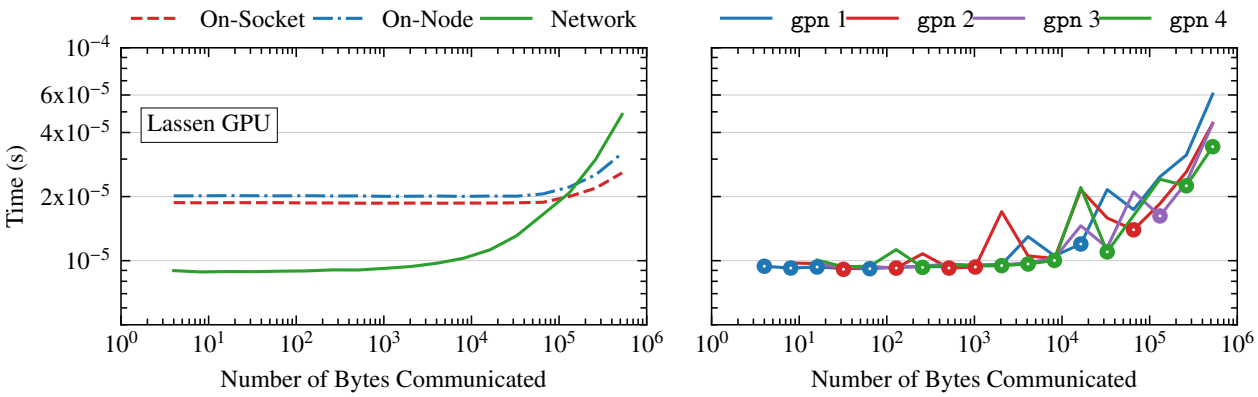


Figure 3.3: Lassen inter-GPU ping-pong (left) and inter-GPU node-pong (right), minimum times circled.

Here, network communication times are much faster than when the two communicating GPUs are on the same socket or same node and separate sockets, until message sizes grow extremely large. Unlike inter-CPU communication, there is not a large benefit to splitting data volumes across multiple GPUs per node, due to the minimal amount of GPUs per node (4 on Lassen). The right plot of Figure 3.3 depicts this in the performance of the inter-GPU node pong, where `gpn` is the

number of active GPUs per node. However, it could be efficient performance-wise to copy data to a host CPU process, then split the inter-node data volume across multiple processes for communication. Motivated by the benchmarks above, we introduce an optimized multi-step communication process that combines aspects of the 3-step and 2-step communication techniques in Section 3.3.

3.3 Split Node-Aware Communication

Leveraging the benefits of communicating large data volumes with all available on-node processes and taking into consideration the variable performance of sending a message depending on the relative locations of processes, we introduce a multi-step communication technique in which each node determines the best way to communicate its inter-node data, named *Split*. We reduce the number of inter-node messages and conglomerate messages to be sent off-node for certain cases when the message sizes to be sent off-node are below a given threshold (`message_cap`), and we split the messages to be sent off-node across multiple processes when the message size is larger than the threshold. Hence, each node determines the most efficient path to inter-node communication. As a result, this Split communication eliminates the redundancies of data being injected into the network, as with 3-step and 2-step (see Section 2.2), but in some cases, it does not reduce the number of inter-node messages as much as 3-step, and can increase the number of inter-node messages injected by a single node to be larger than those injected by 2-step but never exceeding the total number of active processes per node.

Pseudo-code of the setup for the proposed process, excluding reducing the global communication strategy to 3-step or 2-step communication, is provided in Algorithm 3.1 with commmunication parameters defined in Table 3.2. Here, we detail the operations summarized in Algorithm 3.1.

Line 8 The algorithm begins by splitting inter-node messages by their origin node (on-node or off-node).

Line 9 A *local* communicator is created for exchanging all messages with origin on-node.

Line 10 All messages with origin off-node are split into lists according to their origin node.

Lines 11 Parameters, such as the number of nodes from which this node receives, the *maximum* amount of data being received from a single other node, and the *total* amount of data being received from any node by this node, are determined.

Lines 12–17 In this block of the algorithm, the appropriate `message_cap` is determined.

Lines 12–13 First, the maximum amount of data being received from any node is checked to determine if it is smaller than the user provided `message_cap`. When this occurs, the data of each node should be sent in a single message.

Algorithm 3.1: Setup for Split communication.

Input: `l_recv` [list of messages to receive]
`comm` [world communicator]
`message_cap` [user-defined message cap size]
Output: `local_comm` [on-node subcommunicator]
`local_Rcomm` [redistribution subcommunicator]
`global_comm` [off-node subcommunicator]
`local_Scomm` [on-node sending subcommunicator]

1 **Split** messages by origin, off-node and on-node
2 `local_comm` ← **Create** on-node communicator
3 **Split** off-node messages by node
4 **Set** parameters in Table 3.2
5 **if** `max_IN_recv_size` < `message_cap`
6 **Conglomerate** all inter-node receives by node
7 **else**
8 **if** $\frac{\texttt{total\_IN\_recv\_vol}}{\texttt{message\_cap}} > \texttt{PPN}$ & `num_IN_nodes` < `PPN`
9 **Set** $\texttt{message\_cap} = \lceil \frac{\texttt{total\_IN\_recv\_vol}}{\texttt{PPN}} \rceil$
10 **Split** inter-node receives to max size `message_cap`
11 **Set** on-node receive order (descending by size)
12 `local_Rcomm` ← **Create** redistribution communicator (receive)
13 `global_comm` ← **Create** inter-node communicator
14 `local_Scomm` ← **Create** redistribution communicator (send)

Algorithm 3.2: Split communication.

1 Perform `local_comm` communication.
2 Perform `local_Scomm` data redistribution.
3 Perform `global_comm` inter-node communication.
4 Perform `local_Rcomm` data redistribution.

Parameter	Description
`message_cap`	maximum message size when splitting communicated inter-nodal data volumes
`total_IN_recv_vol`	total amount of data being received by this node from any other node in Bytes
`max_IN_recv_size`	maximum amount of data being received from a single other node in Bytes
`num_IN_nodes`	number of nodes from which this node is receiving any messages
`PPN`	processes per node

Table 3.2: Split communication parameters.

Lines 14–17 Otherwise, if the total inter-node data volume being communicated divided by the provided `message_cap` is greater than the active number of processes per node, then the `message_cap` is increased to be the total inter-node data volume divided by the number of on-node processes.

Line 18 On-node processes are assigned inter-node messages to receive in descending order of size, starting with local rank 0. Inter-node messages to be sent are assigned in the reverse order starting with local rank `ppn`-1. This in combination with the message splitting ensures that all processes are active during communication.

Line 19 A *local* communicator is created for redistributing all received inter-node data to its final destination processes on-node.

Line 20 A *global* communicator is created for exchanging inter-node messages based on send and receive message assignment in **Line 18**.

Line 21 A *local* communicator is created for redistributing all inter-node data to be sent by this node to the local processes responsible for sending the inter-node messages.

Algorithm 3.2 provides the steps for performing Split communication once the relevant communicators have been created. While Algorithm 3.1 and the four stages of node-aware communication in Algorithm 3.2 would ultimately be hidden from the user, Algorithm 3.2 demonstrates the flexibility of the communication technique. Depending on the computation in which Split communication is being used, Lines 2 to 4 of Algorithm 3.2 can be overlapped with various pieces of the computation — details of performance gains when overlapping computation with node-aware communication can be found in [39], and for Split are discussed within the context of specific applications in Chapters 4 and 5. Furthermore, the chosen inter-node message size cutoff is determined by the rendezvous protocol based on the communication modeling for Lassen, as it is the observed point at which cross-socket intra-node communication becomes slower than inter-node communication (see Figure 3.2), but it is worth noting that the message size cutoff can be determined via tuning or any other chosen criteria. Similarly, we use a message size cutoff of three in Figure 3.4 to visually demonstrate the multi-step technique when communicating between two nodes with four processes each.

The proposed process is summarized in Figure 3.4, with steps detailed in Description 3.1. This figure outlines the Split communication process of sending data between two nodes, each with four local processes.

Performance modeling and tests of the Split method within the context of sparse matrix-vector multiplication (SpMV) on CPUs and GPUs are presented in Sections 3.5 and 3.6 to demonstrate validity and use cases for the method.

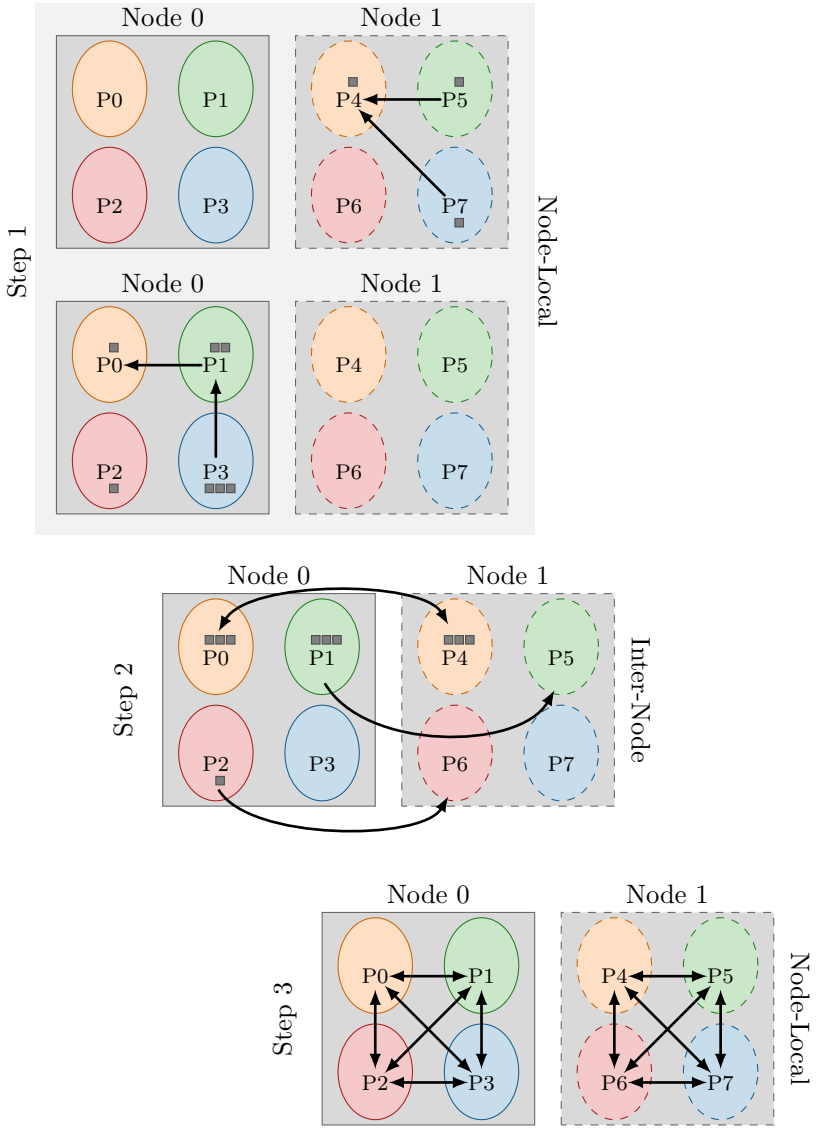


Figure 3.4: Split node-aware. Here, data is communicated between two distinct nodes: Node 0 and Node 1, each with 4 local processes, denoted P#. In Step 1, each node conglomerates small messages to be sent off-node, splits messages based on a message cap of 3, and retains messages approximately the size of the message cap. In Step 2, the buffers prepared in Step 1 are sent to their destination node, specifically to the paired process on that node with the same local rank. P0 exchanges data with P4, while P1 sends data to P5, and P2 sends data to P6. For Step 3, all processes redistribute their received data to the correct destination processes on-node.

Step 1: Each node conglomerates small messages to be sent off-node while retaining larger messages. Messages are assigned to processes in descending order of size to the first available process on node. This is done for every node simultaneously.

Step 2: Buffers prepared in Step 1 are sent to their destination node, specifically to the paired process on that node with the same local rank. P0 exchanges data with P4, while P1 sends data to P5, and P2 sends data to P6.

Step 3: All processes on each individual node redistribute their received data to the correct destination processes on-node. In this step, all communication is node-local.

Description 3.1: Detailed steps for the Split communication process.

3.4 Node-Aware Communication on Heterogeneous Architectures

In this section, we provide context for the modeling and benchmarking in Sections 3.5 and 3.6. Namely, we clarify how the 3-step (Section 2.2.1), 2-step (Section 2.2.2), and Split (Section 3.3) node-aware communication schemes are used to perform inter-GPU communication. When data is moved between two GPUs on separate nodes using MPI, the data can be moved in one of two ways:

Device-aware: data is sent directly from the sending GPU through the NIC and the network to the receiving GPU without being copied to the host CPU; and

Staged-through-host: data is copied to the host CPU before being sent through the NIC and the network to the receiving GPU's host CPU then copied to the receiving GPU.

In the case of 3-step and 2-step communication, both device-aware communication and staged-through-host communication can be used.

Figure 3.5 depicts exchanging inter-node data for 3-step communication. Either device-aware communication can be used to perform the same steps as in Figure 2.4, or data can be staged through the GPU's host processes, represented by a single shaded blue square beneath each GPU. Unused CPU cores are gray.

Similarly, 2-step communication for inter-GPU communication (Figure 3.6) is performed the same as in Figure 2.5 with data either staged through a single host process (blue square beneath each GPU) or communicated directly between GPUs through device-aware technologies.

Split communication is the only node-aware communication strategy which explicitly requires staged-through-host communication, depicted in Figure 3.7. The split communication strategy is designed on the principle of splitting all data to be communicated inter-node across all available on-node cores if necessary. Here, inter-node data to be communicated is copied to some number of host processes (one for each GPU in Figure 3.7, denoted by a blue square beneath the GPU), then scattered across the other on-node cores before inter-node communication is performed. Non-host process cores that are still active for inter-node communication are highlighted green in Figure 3.7.

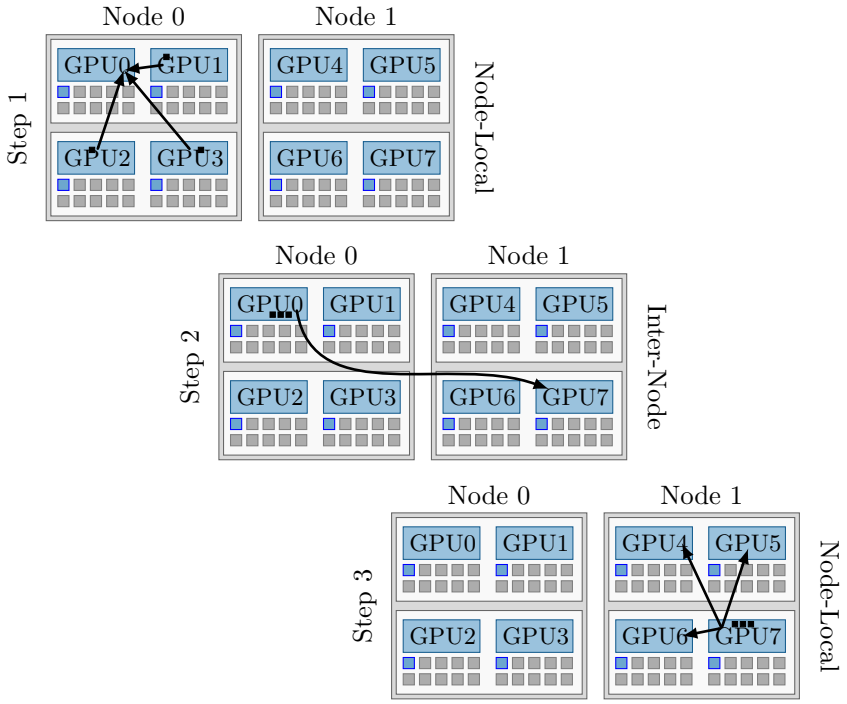


Figure 3.5: 3-Step communication.

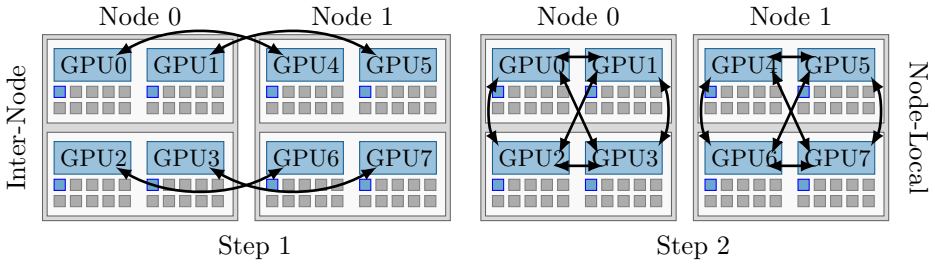


Figure 3.6: 2-Step communication.

In step 3 of the process, all data is gathered back to the host processes before being copied back to the GPU.

3.5 Node-Aware Communication Models

In the following section, we present communication models for node-aware communication for both inter-CPU communication on both Blue Waters and Lassen, as well as, inter-GPU communication in the case of Lassen. Device-aware and staged-through-host communication models are presented in the case of inter-GPU communication. Because both of these involve moving data through the GPU and possibly the CPU, it is important to consider the cost of transmitting data through all possible data flow paths involving the CPU or GPU.

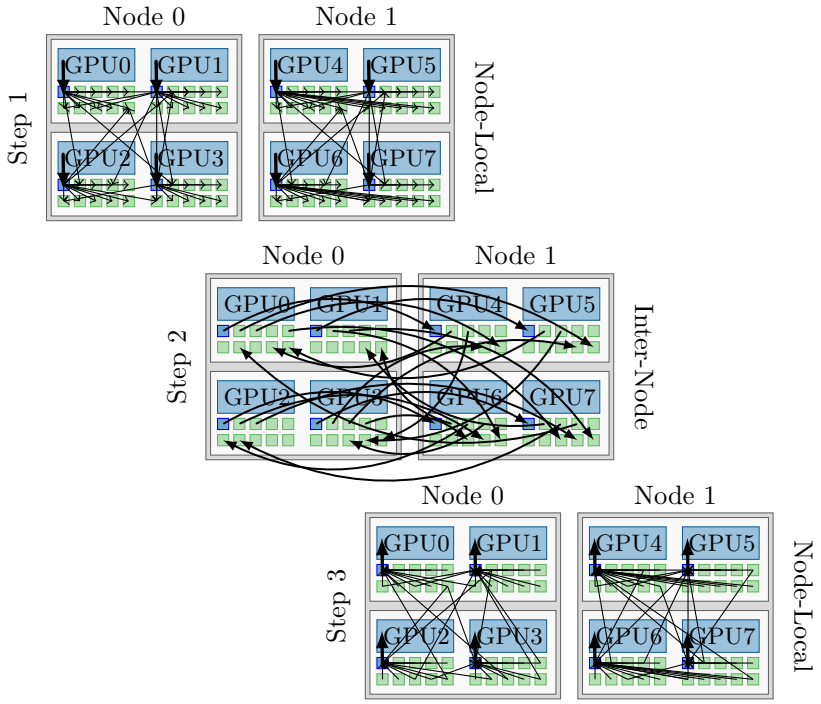


Figure 3.7: Split communication.

3.5.1 Data Movement Modeling Parameters

The postal model presented in (2.1) is used to model the time required for sending a single message between two CPUs or two GPUs, with the measured parameters for Blue Waters and Lassen in Table 3.3, respectively. The α and β parameters are separated based on where the two processes are physically located with respect to one another, namely on the same socket, on different sockets but the same node, or separate nodes. In addition, the parameters are split further based on messaging protocol:

short fits in the envelope so the message is sent directly to the receiving process;

eager assumes adequate buffer space is already allocated by the receiving process; or

rendezvous requires the receiving process to allocate buffer space for the message before the data is sent.

The short protocol has been excluded from the GPU messaging parameter portion of Table 3.3 because this protocol is not used in device-aware communication on Lassen. Furthermore, the inter-GPU parameters are split further into whether GPUDirect technologies were enabled (GDR) or disabled (No GDR), demonstrating the benefits of utilizing GPUDirect technologies for device-aware communication.

Because staging data through a host process requires copying data to the sending host CPU and from the receiving GPU's host process, measured parameters for `cudaMemcpyAsync` are included

			Blue Waters			Lassen		
			on-socket	on-node	off-node	on-socket	on-node	off-node
inter-CPU	Short	α	4.42e-07	8.28e-07	2.27e-06	3.67e-07	9.25e-07	1.89e-06
		β	2.56e-09	4.76e-08	1.42e-09	1.32e-10	1.19e-09	6.88e-10
	Eager	α	5.32e-07	1.21e-06	6.95e-06	4.61e-07	1.17e-06	2.44e-06
		β	3.12e-09	9.54e-08	7.33e-08	7.12e-11	2.18e-10	3.79e-10
	Rend	α	1.67e-06	2.46e-06	4.04e-06	3.15e-06	6.77e-06	7.76e-06
		β	6.10e-09	6.16e-09	2.88e-09	3.40e-11	1.49e-10	7.97e-11
inter-GPU GDR	Eager	α	-	-	-	1.87e-06	2.02e-05	8.95e-06
		β	-	-	-	5.79e-11	2.15e-10	1.72e-10
	Rend	α	-	-	-	1.82e-05	1.93e-05	1.10e-05
		β	-	-	-	1.46e-11	2.39e-11	1.72e-10
inter-GPU No GDR	Eager	α	-	-	-	4.15e-05	4.27e-05	4.56e-05
		β	-	-	-	6.08e-09	6.24e-09	6.03e-10
	Rend	α	-	-	-	5.54e-05	5.75e-05	8.72e-05
		β	-	-	-	8.96e-11	8.11e-11	7.96e-11

α [sec] β [sec/byte]

Table 3.3: Measured parameters for communication on Blue Waters and Lassen.

in Table 3.4 with distinction between whether the copy is using a single process or four processes to move data from the device. We assume that all data copies will occur on-socket, and we do not consider cases with more than four processes pulling data from a single GPU at a time as there was no observed benefit in splitting data copies further across multiple processes — see Figure 3.8.

		1 proc	4 proc	8 proc	10 proc
HostToDevice	α	1.30e-05	1.52e-05	3.10e-05	3.85e-05
	β	1.85e-11	5.52e-10	7.88e-11	9.43e-11
DeviceToHost	α	1.27e-05	1.47e-05	3.03e-05	3.81e-05
	β	1.96e-11	1.50e-10	6.21e-11	1.05e-10

α [sec] β [sec/byte]

Table 3.4: Measured parameters for `cudaMemcpyAsync` on Lassen.

In addition to considering the postal model for inter-CPU and inter-GPU communication, the max-rate model presented in (2.2) is required for accurately predicting the performance of staging GPU data through a host process when using more than a single process per node. Therefore, the measured injection bandwidth limits for inter-CPU communication are presented in Table 3.5. The inter-GPU injection bandwidth limit is excluded, as these limits are not reached with the four available GPUs per node on Lassen. Using the measured modeling parameters, we now model the performance of various communication strategies based on the node-aware techniques discussed

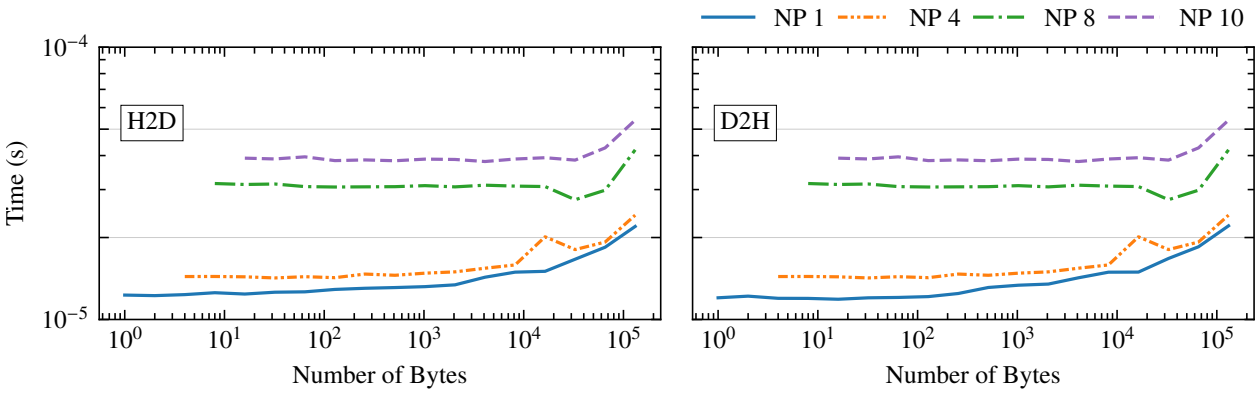


Figure 3.8: The time required to copy various amounts of data from a single GPU using `cudaMemcpyAsync` when splitting the copy across NP processes. HostToDevice (H2D) and DeviceToHost (D2H) timings presented.

	R_N^{-1} [bytes/sec]
Blue Waters	6.57e-09
Lassen	4.19e-11

Table 3.5: Measured parameter for inter-CPU injection bandwidth limits.

in Sections 2.2 and 3.3.

3.5.2 Communication Models

In this section, we present performance models for all node-aware strategies including Split for inter-CPU and inter-GPU inter-node communication. Inter-GPU communication models include both device-aware and staged-through-host communication for data exchanges on Lassen, though these models do extend to any machine with two sockets per node. Inter-CPU models apply to both Blue Waters and Lassen, but also extend to any machine with two sockets per node. For each node-aware case, the models are divided into the time spent in on-node communication (Sections 3.5.2.1 and 3.5.2.2), off-node communication (Section 3.5.2.3), and data copies, in the case of staged-through-host communication (see Section 3.5.2.4). The models themselves do not consider the removal of duplicate data discussed in Section 2.2, as the amount of duplicate data injected into the network is operation and problem dependent. However, adapting the input parameters for the models to reflect the removal of duplicate data is straightforward.

Performance is modeled for standard communication and all node-aware communication strategies discussed in Sections 2.2, 3.3 and 3.4. We consider staged-through-host and device-aware communication for all of the strategies except for the split strategies, for which device-aware communication does not apply. "Split + MD" first copies data to a single host process, then splits the inter-node data to be communicated across multiple processes via extra on-node inter-CPU

messages. "Split + DD" uses duplicate device pointers to copy data from a GPU to multiple host process, reducing the number of on-node messages required to split the inter-node data volume being communicated. Each GPU is assumed to have a single host process except in the case of "Split + DD". For reference, the inter-GPU modeled communication strategies are listed in Table 3.6. Furthermore, discussion of the node aware communication strategies is done within

	Staged-through-host	**Device-aware**
Standard	✓	✓
3-Step	✓	✓
2-Step	✓	✓
Split + MD	✓	
Split + DD	✓	

Table 3.6: Modeled inter-GPU communication strategies.

the context or processing units (PUs) for instances that apply to both inter-CPU and inter-GPU communication. When it is neccessary to differentiate between whether the communication strategy is being used for inter-CPU or inter-GPU communication, it is stated explicitly.

3.5.2.1 Modeling On-Node Communication for 3-Step and 2-Step

For 3-Step communication, all data originating on any PU on node k with a destination of any PU on node l is first gathered locally. In the worst-case scenario, all PUs on node k must contribute data for node l, requiring communication among all PUs per node. For inter-CPU communication, this is modeled as

$$T_{\texttt{on-cpu}}(s) = (\texttt{pps} - 1)(\alpha_{\texttt{on-socket}} + \beta_{\texttt{on-socket}} \cdot s) + \texttt{pps} \cdot (\alpha_{\texttt{on-node}} + \beta_{\texttt{on-node}} \cdot s) \tag{3.1}$$

where pps is the CPU processes per socket and s is the maximum message size sent by any single process. For inter-GPU communication, the model is similarly,

$$T_{\texttt{on-gpu}}(s) = (\texttt{gps} - 1)(\alpha_{\texttt{on-socket}} + \beta_{\texttt{on-socket}} \cdot s) + \texttt{gps} \cdot (\alpha_{\texttt{on-node}} + \beta_{\texttt{on-node}} \cdot s) \tag{3.2}$$

where gps is the GPUs per socket and s is the maximum message size sent by any single GPU.

The last step of both 2-Step and 3-Step communication involves redistributing data received via inter-node communication to its final destination PU on-node. The worst case scenario for both strategies occurs when all of the data received via inter-node communication needs to be redistributed to every other PU on-node. This is also modeled with (3.2), with s representing the maximum received inter-node message size.

3.5.2.2 Modeling On-Node Communication for Split

For inter-CPU communication, the worst-case scenario for split communication occurs when a single CPU process contains all data to be sent off-node with a data size large enough that it is split across all on-node processes. This equates to the same worst-case scenario for 3-Step and 2-Step communication in (3.1), hence

$$T_{\texttt{on-cpu-split}}(s) = T_{\texttt{on-cpu}}(s) \tag{3.3}$$

where s is the total inter-node data volume to be communicated. Additionally, this also represents the worst-case redistribution scenario.

For inter-GPU communication, the split strategies require copying all data on node k with destination of any GPU on node $l \neq k$ to the host processes before distributing the data across some number of on-node processes. Finally, each process sends data through the network. For large inter-node message sizes, the worst-case scenario occurs when a single GPU contains all data to be sent off-node with a data size large enough that it is split across all on-node processes. In the case of Lassen, there are a maximum of 40 on-node processes, therefore distributing the data would require an additional 19 on-socket messages and 20 off-socket/on-node messages if a single host process per GPU were being used. Generalizing the split strategy to any architecture using multiple host processes with duplicate device pointers yields

$$T_{\texttt{on-gpu-split}}(s, \texttt{ppg}) = \left(\frac{\texttt{pps}}{\texttt{ppg}} - 1\right) \cdot (\alpha_{\texttt{on-socket}} + \beta_{\texttt{on-socket}} \cdot s) + \left(\frac{\texttt{pps}}{\texttt{ppg}}\right) \cdot (\alpha_{\texttt{on-node}} + \beta_{\texttt{on-node}} \cdot s) \tag{3.4}$$

as the modeled time, where ppg is the number of host processes per GPU, and pps is the processes per socket, and s is the total data volume to be communicated inter-node split across ppg.

Similar to the worst-case scenario for 3-Step and 2-Step on-node communication, the worst-case redistribution scenario for the split strategies is equivalent to (3.4). In this case, a single GPU must redistribute all received inter-node data to every other GPU on-node; here, s represents the total data volume received via inter-node communication split across ppg.

3.5.2.3 Modeling Off-Node Communication

For the off-node communication portion of each of the multi-step communication strategies, the max-rate model (2.2) is used for inter-CPU communication and inter-GPU routines that are staged-through-host, and the postal model (2.1) is used for device-aware routines. For the max-rate model, the time spent in off-node communication is given by

$$T_{\texttt{off}}(m, s) = \alpha_{\texttt{off-node}} \cdot m + \max\left(\frac{s_{\texttt{node}}}{R_N}, s \cdot \beta_{\texttt{off-node}}\right) \tag{3.5}$$

for a number of messages to be communicated, m, and a maximum number of bytes sent by a single process, $s_{\texttt{proc}}$ where $s_{\texttt{node}}$ is the maximum number of bytes injected into the network by any single

Communication Strategy	Model
Standard	Max-rate model (2.2)
3-Step	$T_{\texttt{off}}(m_{\texttt{node}\to\texttt{node}}, s_{\texttt{node}\to\texttt{node}}) + 2 \cdot T_{\texttt{on-cpu}}(s_{\texttt{node}\to\texttt{node}})$
2-Step	$T_{\texttt{off}}(m_{\texttt{proc}\to\texttt{node}}, s_{\texttt{proc}}) + T_{\texttt{on-cpu}}(s_{\texttt{proc}})$
Split	$T_{\texttt{off}}(m_{\texttt{proc}\to\texttt{node}}, s_{\texttt{node}}/_{\texttt{ppn}}) + 2 \cdot T_{\texttt{on-cpu-split}}(s_{\texttt{node}})$

Table 3.7: Inter-CPU communication models. (Recall parameters defined in Table 2.1)

Communication Strategy		Model
Standard	Staged-through-host	Max-rate model (2.2)
	Device-aware	Postal model (2.1)
3-Step	Staged-through-host	$T_{\texttt{off}}(m_{\texttt{node}\to\texttt{node}}, s_{\texttt{node}\to\texttt{node}}) + 2 \cdot T_{\texttt{on-gpu}}(s_{\texttt{node}\to\texttt{node}}) + T_{\texttt{copy}}(s_{\texttt{proc}}, s_{\texttt{node}\to\texttt{node}})$
	Device-aware	$T_{\texttt{off-DA}}(m_{\texttt{node}\to\texttt{node}}, s_{\texttt{node}\to\texttt{node}}) + 2 \cdot T_{\texttt{on-gpu}}(s_{\texttt{node}\to\texttt{node}})$
2-Step	Staged-through-host	$T_{\texttt{off}}(m_{\texttt{proc}\to\texttt{node}}, s_{\texttt{proc}}) + T_{\texttt{on-gpu}}(s_{\texttt{proc}}) + T_{\texttt{copy}}(s_{\texttt{proc}}, s_{\texttt{node}\to\texttt{node}})$
	Device-aware	$T_{\texttt{off-DA}}(m_{\texttt{proc}\to\texttt{node}}, s_{\texttt{proc}}) + T_{\texttt{on}}(s_{\texttt{proc}})$
Split	Staged-through-host + MD	$T_{\texttt{off}}(m_{\texttt{proc}\to\texttt{node}}, s_{\texttt{node}}/_{\texttt{ppn}}) + 2 \cdot T_{\texttt{on-gpu-split}}(s_{\texttt{node}}, 1) + T_{\texttt{copy}}(s_{\texttt{proc}}, s_{\texttt{node}\to\texttt{node}})$
	Staged-through-host + DD	$T_{\texttt{off}}(m_{\texttt{proc}\to\texttt{node}}, s_{\texttt{node}}/_{\texttt{ppn}}) + 2 \cdot T_{\texttt{on-gpu-split}}(s_{\texttt{node}}, 4) + T_{\texttt{copy}}(s_{\texttt{proc}}, s_{\texttt{node}\to\texttt{node}})$

Table 3.8: Inter-GPU communication models. (Recall parameters defined in Table 2.1)

node. For device-aware communication, this reduces to the postal model

$$T_{\texttt{off-DA}}(m, s) = \alpha_{\texttt{off-node}} \cdot m + s \cdot \beta_{\texttt{off-node}}. \tag{3.6}$$

3.5.2.4 Copy Parameter for Staged-through-Host Communication

The time required to copy data between the CPU and GPU is given by

$$T_{\texttt{copy}}(s_{\texttt{send}}, s_{\texttt{recv}}) = \alpha_{\texttt{H2D}} + \beta_{\texttt{H2D}} \cdot s_{\texttt{send}} + \alpha_{\texttt{D2H}} + \beta_{\texttt{D2H}} \cdot s_{\texttt{recv}}. \tag{3.7}$$

where $s_{\texttt{send}}$ is the initial data copied from the source GPU, and $s_{\texttt{recv}}$ is the final data copied to the destination GPU.

For all communication strategies except splitting with duplicate device pointers, a single process copies all data from a corresponding GPU. In the case of splitting with duplicate device pointers, we set the number of processes copying data simultaneously to four in our model. Parameters for both a single host process copying data and four host processes copying data simultaneously are presented in Table 3.4.

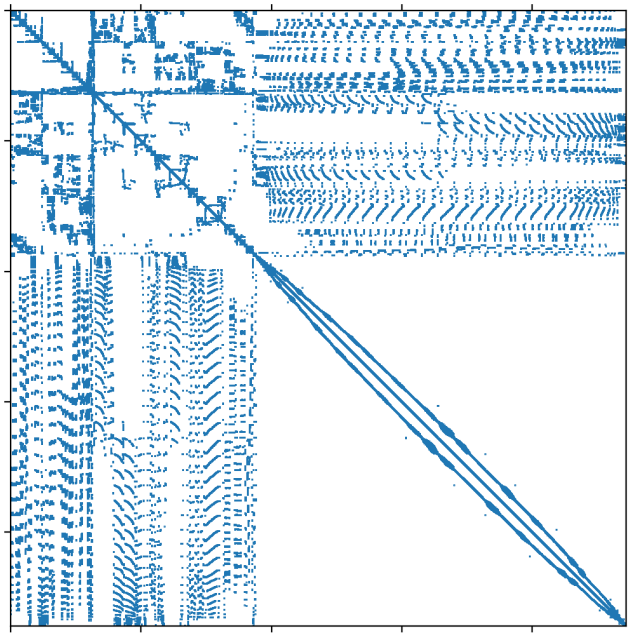
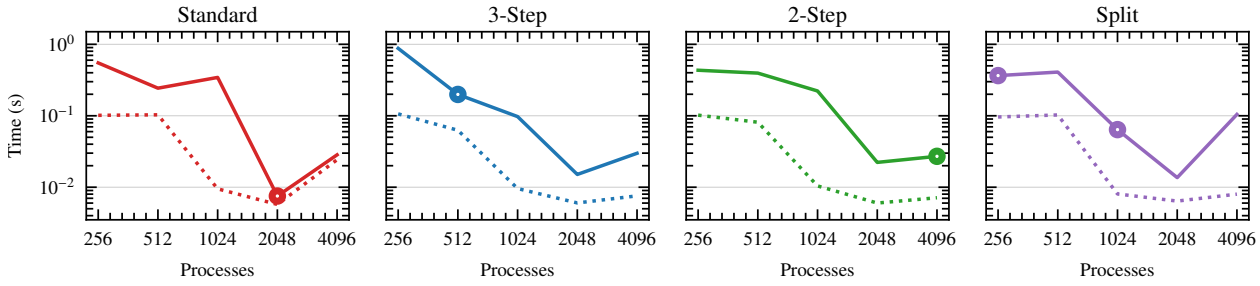


Figure 3.9: Sparsity pattern for the `audikw_1` matrix.

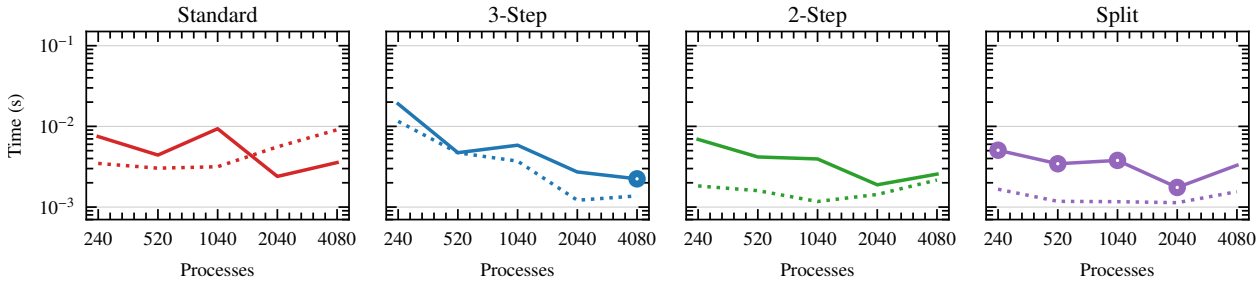
3.5.2.5 Model Validation

Tables 3.7 and 3.8 present the full models for the various inter-CPU and inter-GPU communication strategies given in Table 3.6, which combine the preceding sub-models. We provide a brief validation of the models via performance of the communication pattern induced by sparse matrix-vector multiplication (SpMV) with the `audikw_1` matrix from the SuiteSparse matrix collection [63]. The matrix has 943 695 rows and columns, and a nonzero density of 8.72e-05 with the associated sparsity pattern in Figure 3.9. Due to the high number of nonzero entries in the top rows and first columns of the matrix, the communication pattern associated with a SpMV for `audikw_1` incurs high numbers of on-node and inter-node communication, therefore it is a perfect test case for validating the models which model the worst-case on-node communication scenarios for each of the communication strategies. It is worth noting that these models are not designed to provide a fine-grained prediction. Their purpose is to predict which communication scheme will perform fastest, a task at which they succeed. It is worth noting that the models are easily adaptable to modeling the *exact* on-node communication that would occur for a given application, should a more fine-grained model be desired. This would require knowledge of problem partitioning, as well as, the communication load of every participating process.

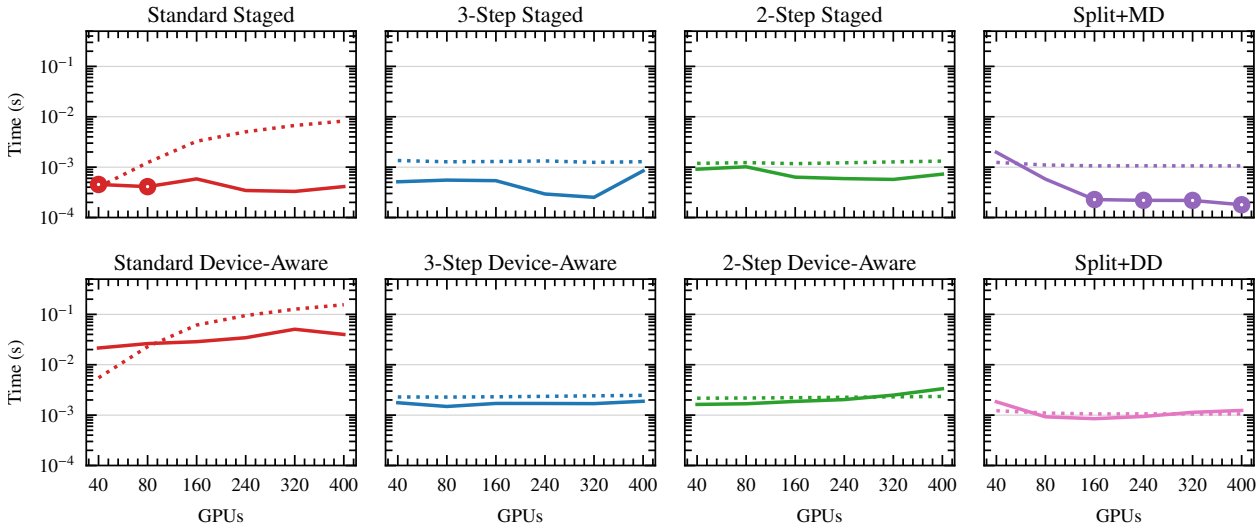
Figure 3.10 depicts the measured times (solid lines) for SpMV communication alongside model predictions (dotted lines) with minimum performing *and minimum model-predicted* times circled. For inter-CPU communication, the models typically lie beneath the measured performance results, consistent with previous literature on inter-CPU node-aware communication, however, they do accurately predict the minimum performing communication strategy for both Blue Waters (Figure 3.10a) and Lassen(Figure 3.10b). In the inter-GPU standard communication cases, the



(a) Blue Waters CPU.



(b) Lassen CPU.



(c) Lassen GPU.

Figure 3.10: Model validation. Solid lines, ——, depict measured times, and dotted lines, ……, depict model predictions. Circles are used to highlight the minimum performing communication strategy, accurately predicted by the models.

modeled times are an order of magnitude higher than actual measured times, but for the inter-GPU node-aware commmunication models, the predicted times provide a tight upperbound, generally on the same order of magnitude as the measured performance (Figure 3.10c). In Section 3.5.3, we use these models to predict the performance of common irregular point-to-point communication scenarios.

3.5.3 Modeled Performance

In this section, we present the modeled performance for common scenarios with irregular point-to-point communication. We consider inter-CPU communication strategies for both Blue Waters and Lassen (Section 3.5.3.1), and inter-GPU communication for Lassen (Section 3.5.3.2). We model a node sending a *modest* number of inter-node messages and a *large* number of inter-node messages, with messages distributed evenly across all on-node processes or GPUs. Because the node-aware communication performance models are dependent upon the number of destination nodes, the models are split further, modelling if the data was being sent to 4 nodes or 16 nodes. Note that the number of processes or GPUs to which data is being communicated does not reflect overall problem partititioning. It simply models the cases where the maximum number of processes or GPUs with which any one node would need to communicate is 4 nodes or 16 nodes.

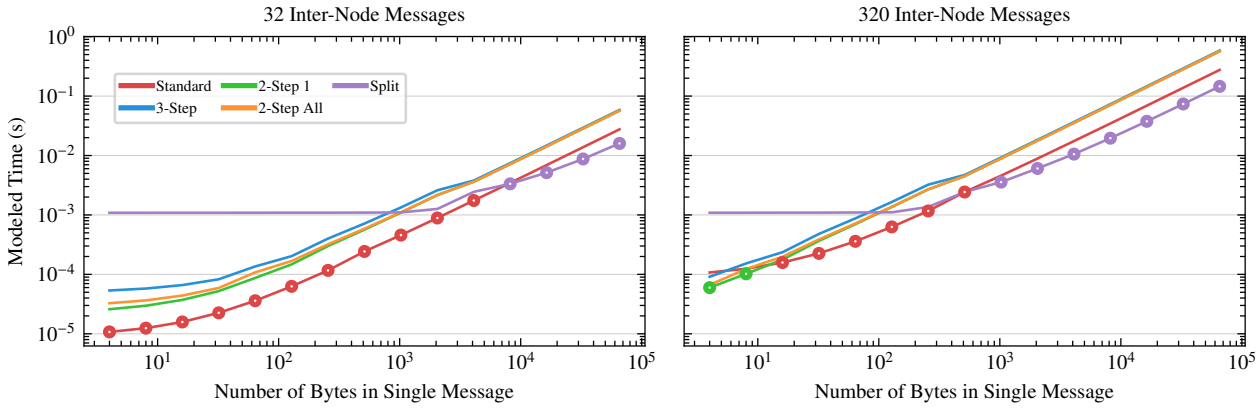
For each of these scenarios, we model the amount of time required for each node to send their messages to the destination nodes using standard communication. This modeled performance is compared against that of the various node-aware strategies where the messages are split and/or agglomerated accordingly. There are two cases presented for the 2-Step strategy, considering if every process or GPU on the source node is sending data to every process or GPU on the destination node (2-Step All), or if all the messages being sent to the destination node are from a single active process or GPU on the source node (2-Step 1). The message size cap for the split strategies is taken to be the message cap used for switching to the rendezvous protocol on Lassen, as discussed in Section 3.3.

3.5.3.1 Inter-CPU Communication

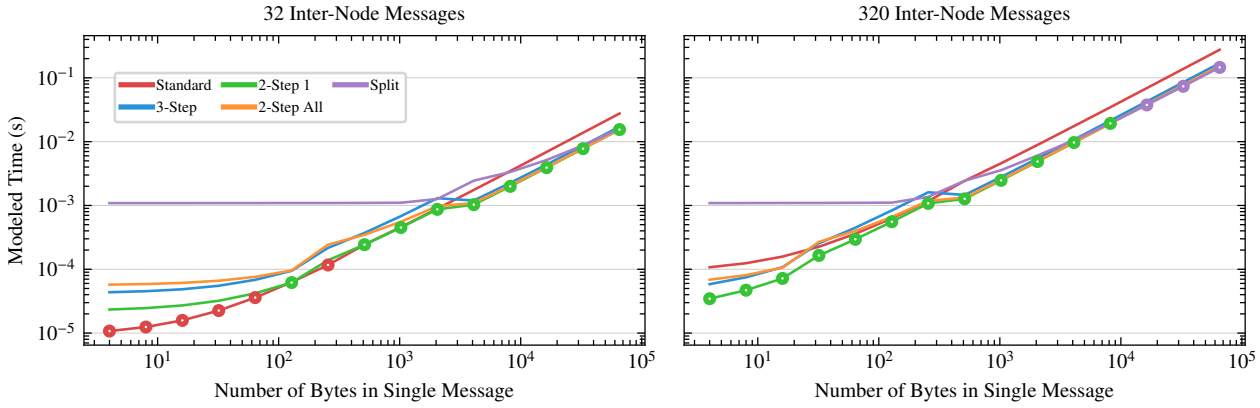
Figures 3.11 and 3.12 present the inter-CPU modeled performance of communication for Blue Waters and Lassen, respectively. For Blue Waters, we model sending 32 and 320 inter-node messages, with messages distributed evenly across 16 on-node processes. For Lassen, we increase the number of inter-node messages to 40 and 400, evenly distributed across 40 on-node processes.

In the case of Blue Waters, Figure 3.11, we see that the inclusion of split communication extends the performance benefits of node-aware communication. When sending to a small number of nodes (Figure 3.11a), standard communication is modeled as being the most performant, except in the case of sending high data volumes, where split communication is predicted as the best communication strategy. Sending to a larger number of nodes (Figure 3.11b), we see the best-case scenario for 2-Step (2-Step 1) indicated as the most performant strategy except for small numbers of small messages (standard) or large numbers of large messages (split). Overall, this indicates that for high data volumes, split is the most optimal communication strategy on Blue Waters.

In the case of Lassen, Figure 3.12, standard and split communication are almost split evenly as the most performant strategeis. When a small number of messages is being sent, independent of the number of receiving nodes, standard communication models as the fastest, but with split communication modeling a very close performance time. For high numbers of messages,



(a) 4 Nodes



(b) 16 Nodes

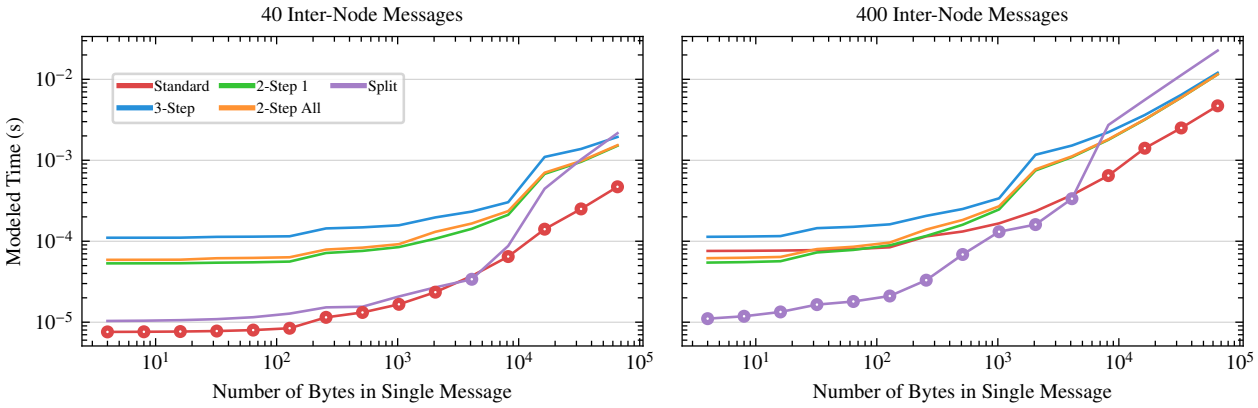
Figure 3.11: Blue Waters models. The modeled time to send data from a single node to **4 nodes (top)** and **16 nodes (bottom)**, where data from the sending node is sent via 32 or 320 messages distributed evenly across all on-node processes when using Standard communication. Minimum modeled times are circled.

independent of the number of receiving nodes, split communication models as the most performant for message sizes up 10^4 bytes, after which either standard communication (Figure 3.12a) or the best-case 2-Step 1 communication (Figure 3.12b) models as the most performant. It is worth noting that 2-Step All and 3-Step communication are close the performant of 2-Step 1 for message sizes larger than 10^4 when sending to 16 nodes, indicating that node-aware communication is still predicted to be more performant than standard communication for high data volumes.

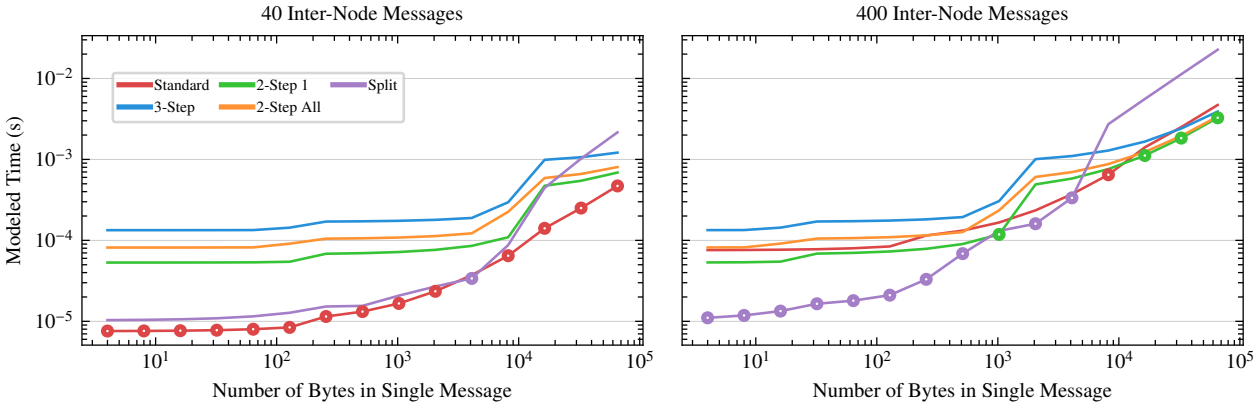
3.5.3.2 Inter-GPU Communication

Figure 3.13 displays the inter-GPU modeled performance on Lassen for the cases of sending 32 or 256 inter-node messages, distributed evenly across 4 on-node GPUs.

In Figure 3.13, we present the minimum modeled times on the top rows, excluding the 2-Step 1 approaches, as they present the *best-case* scenario for 2-Step communication, which does not often



(a) 4 Nodes

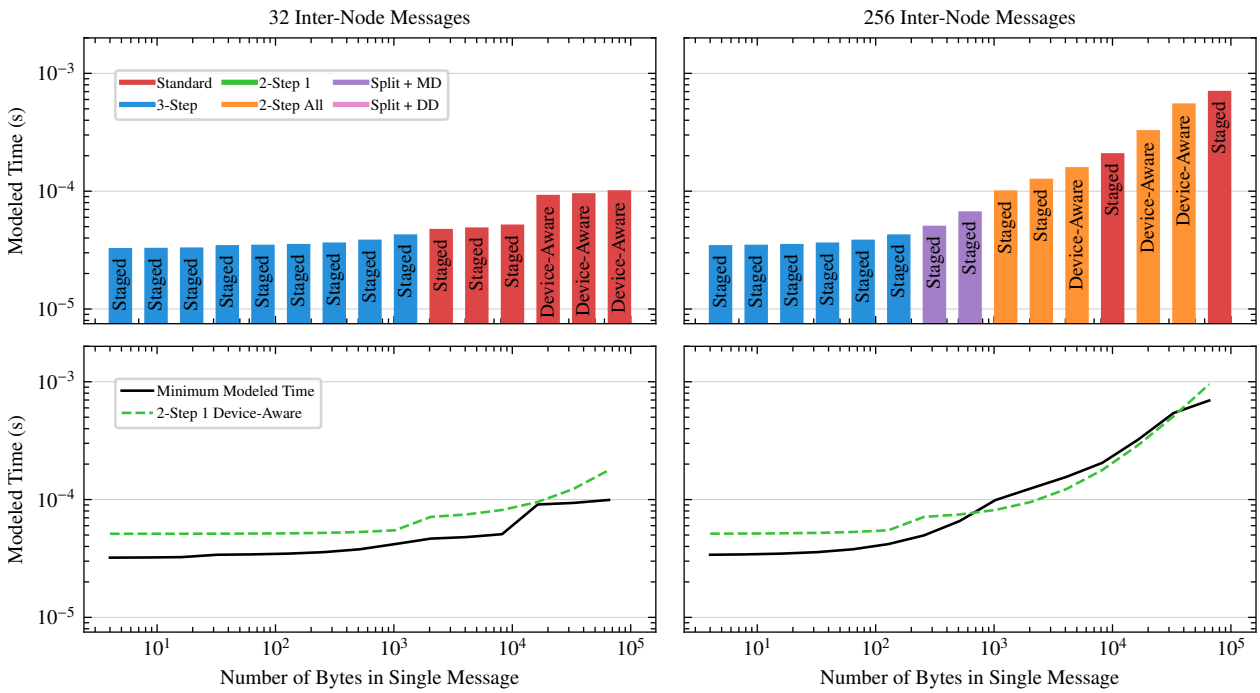


(b) 16 Nodes

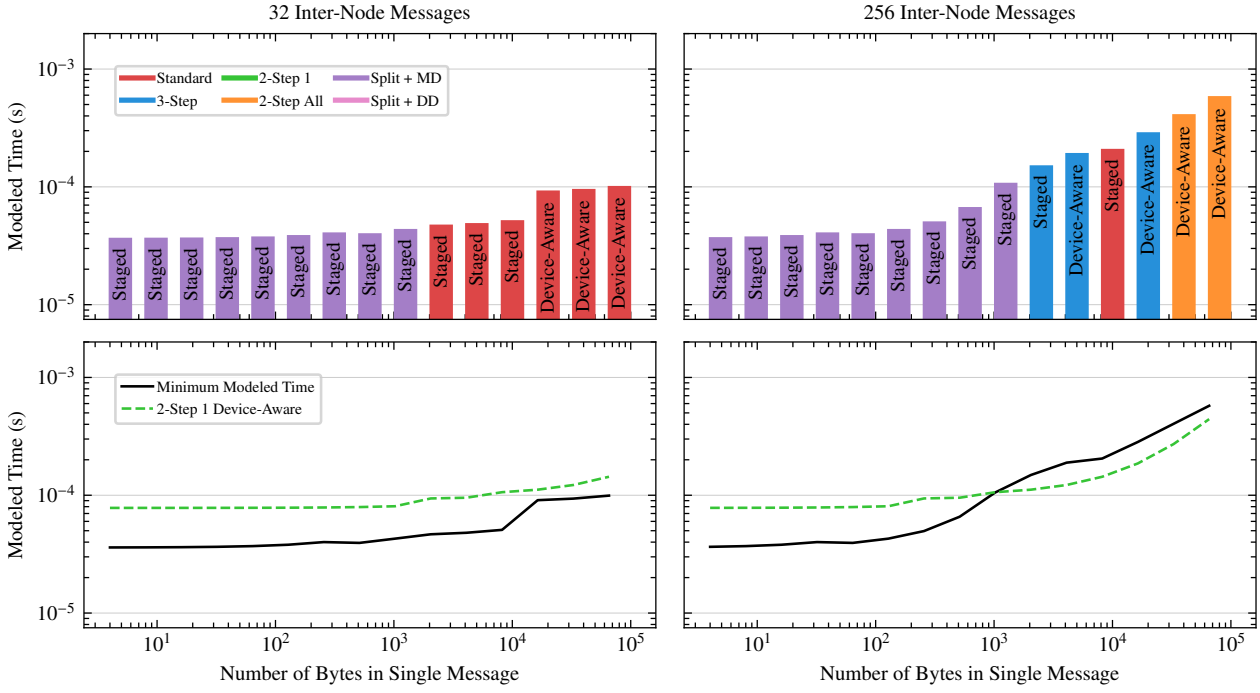
Figure 3.12: Lassen models. The modeled time to send data from a single node to **4 nodes (top)** and **16 nodes (bottom)**, where data from the sending node is sent via 40 or 400 messages distributed evenly across all on-node processes when using Standard communication. Minimum modeled times are circled.

occur in practice. However, we do think it is important to present these modeled times in order to depict a comprehensive picture of node-aware communication's potential, hence they are included in the bottom rows of the plots. For large message counts (256 Inter-Node Messages plots in Figure 3.13) and for message sizes greater than 10^3 Bytes, device-aware 2-Step 1 is predicted to perform best, indicating that for high inter-node data volumes, if the on-node data was distributed such that every GPU on a given node k was communicating with a distinct GPU on destination node l, 2-Step communication would be best. This is consistent with the observed performance of the application-specific hierarchical communication in [40]. Now, we include discussion of the circled minimum times excluding the 2-Step 1 performance predictions.

In the case of a small number of messages injected into the network to a small number of nodes (Figure 3.13a), 3-Step and standard communication are observed as the most performant with Split+MD communication replacing 3-Step as the most performant for 16 nodes (Figure 3.13b). In both cases, the staged-through-host strategies predict the best performance until message sizes



(a) 4 Nodes



(b) 16 Nodes

Figure 3.13: The modeled time to send data from a single node to **4 nodes (top)** and **16 nodes (bottom)**, where data from the sending node is sent via 32 or 256 messages distributed evenly across all on-node GPUs when using Standard communication. Minimum modeled times (excluding the 2-Step *best-case* scenario, 2-Step 1) are presented with bars on the top rows. A comparison of the minimum modeled times to the 2-Step 1 case is presented on the bottom rows.

grow extremely large ($> 10^4$ Bytes), where standard device-aware communication is modeled to be best. Device-aware communication is also modeled to be best for large message sizes when a node is injecting a large number of messages into the network. However, due to the high message volume, 3-Step and 2-Step device-aware strategies are predicted to have the optimal performance, due to their reduction in messages sent compared to standard communication.

Staged-through-host node-aware communication techniques model the best performance independent of number of destination nodes for all message sizes up to 10^4 Bytes. When communicating with a small number of nodes (Figure 3.13a), 3-Step and 2-Step communication are often predicted to be the most performant, while Split+MD communication is predicted to be the most performant when communicating with a larger number of nodes (Figure 3.13b). This can be attributed to the use of all available processes on-node (40 in the case of Lassen), so that each individual process is injecting fewer messages into the network than in the case of 3-Step or 2-Step communication, where there is only a single process paired with each GPU (4 in the case of Lassen).

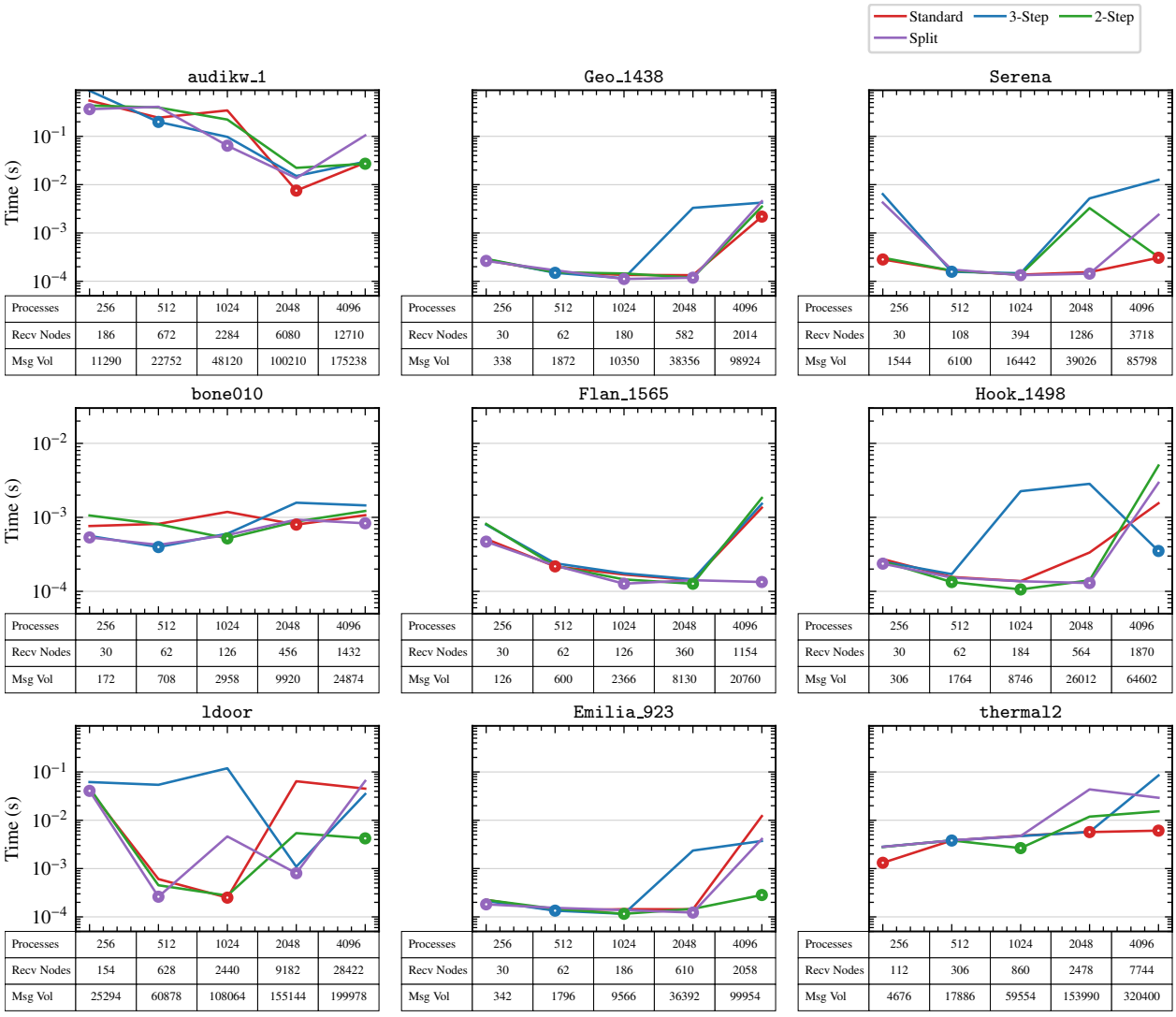
The device-aware node-aware strategies models present relatively large costs. However, this is unsurprising, considering the high overhead for inter-GPU communication on-socket and on-node (as indicated by the measured parameters in Table 3.3). The only cases for which device-aware node-aware strategies have improved performance over staged-through-host techniques is when the communicated inter-node data volume is *extremely* large, or assumed to have an optimal communication pattern (as in 2-Step 1).

Concerning the removal of duplicate data, there should be no impact on performance for small numbers of inter-node messages. A performance impact is noticed primarily when their is communication of a high inter-node data volume via a high number of messages. Once message sizes grow past 10^3 Bytes in standard communication for all high message count models, removing duplicate data impacts which node-aware communication strategy could be most performant. Seeing as there is very little different in modeled performance for removing duplicate data, these modeled times are excluded for brevity.

Overall, the staged-through-host node-aware communication strategies model the best predicted performance for communication patterns requiring a high number of inter-node message exchanges. In Section 3.6, we benchmark the performance of the communication strategies within the context of sparse matrix-vector multiplication, verifying model predictions.

3.6 Split Node-Aware Communication in Parallel SpMV

In this section, we present performance results for the various communication strategies discussed throughout Sections 2.2 and 3.5 when applied to the communication patterns of a single distributed SpMV – see Figure 2.1. The test matrices are a subset of the largest matrices in the SuiteSparse matrix collection [63]. For each benchmark, we perform 1000 test runs and present the *maximum* average time required for communication by any *single* process. The results presented reflect *actual* measurements, not model predictions. Results are split into inter-CPU results on Blue Waters and



audikw_1					
Processes	256	512	1024	2048	4096
Recv Nodes	186	672	2284	6080	12710
Msg Vol	11290	22752	48120	100210	175238

Geo_1438					
Processes	256	512	1024	2048	4096
Recv Nodes	30	62	180	582	2014
Msg Vol	338	1872	10350	38356	98924

Serena					
Processes	256	512	1024	2048	4096
Recv Nodes	30	108	394	1286	3718
Msg Vol	1544	6100	16442	39026	85798

bone010					
Processes	256	512	1024	2048	4096
Recv Nodes	30	62	126	456	1432
Msg Vol	172	708	2958	9920	24874

Flan_1565					
Processes	256	512	1024	2048	4096
Recv Nodes	30	62	126	360	1154
Msg Vol	126	600	2366	8130	20760

Hook_1498					
Processes	256	512	1024	2048	4096
Recv Nodes	30	62	184	564	1870
Msg Vol	306	1764	8746	26012	64602

ldoor					
Processes	256	512	1024	2048	4096
Recv Nodes	154	628	2440	9182	28422
Msg Vol	25294	60878	108064	155144	199978

Emilia_923					
Processes	256	512	1024	2048	4096
Recv Nodes	30	62	186	610	2058
Msg Vol	342	1796	9566	36392	99954

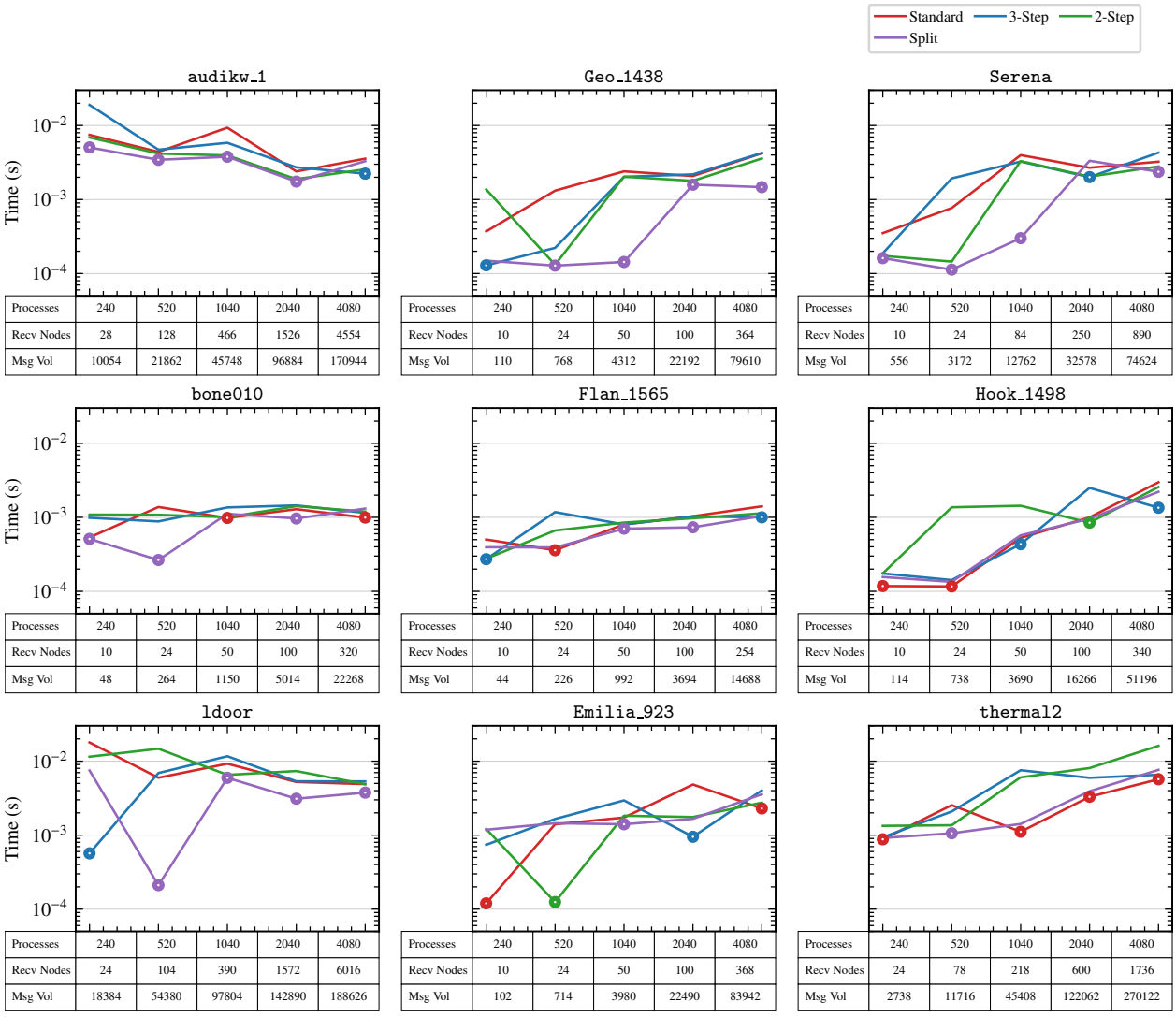
thermal2					
Processes	256	512	1024	2048	4096
Recv Nodes	112	306	860	2478	7744
Msg Vol	4676	17886	59554	153990	320400

Figure 3.14: Blue Waters inter-CPU results. The measured time spent in irregular point-to-point communication for a distributed SpMV for various SuiteSparse matrices. Number of processes across which the problem was partitioned and standard communication maximum number of connected nodes for any single node (Recv Nodes) and message volume included beneath plots. Minimum times circled for convenience.

Lassen (Section 3.6.1) and inter-GPU results on Lassen (Section 3.6.2) for clarity.

3.6.1 Inter-CPU Communication

Figures 3.14 and 3.15 display the distributed SpMV communication benchmark performance times for each communication strategy for each SuiteSparse matrix. Presented beneath each plot is the number of processes on which the SpMV is partitioned, the maximum number of nodes to which any single node is communicating (Recv Nodes), and the communicated inter-node message volume for standard communication.



audikw_1					
Processes	240	520	1040	2040	4080
Recv Nodes	28	128	466	1526	4554
Msg Vol	10054	21862	45748	96884	170944

Geo_1438					
Processes	240	520	1040	2040	4080
Recv Nodes	10	24	50	100	364
Msg Vol	110	768	4312	22192	79610

Serena					
Processes	240	520	1040	2040	4080
Recv Nodes	10	24	84	250	890
Msg Vol	556	3172	12762	32578	74624

bone010					
Processes	240	520	1040	2040	4080
Recv Nodes	10	24	50	100	320
Msg Vol	48	264	1150	5014	22268

Flan_1565					
Processes	240	520	1040	2040	4080
Recv Nodes	10	24	50	100	254
Msg Vol	44	226	992	3694	14688

Hook_1498					
Processes	240	520	1040	2040	4080
Recv Nodes	10	24	50	100	340
Msg Vol	114	738	3690	16266	51196

ldoor					
Processes	240	520	1040	2040	4080
Recv Nodes	24	104	390	1572	6016
Msg Vol	18384	54380	97804	142890	188626

Emilia_923					
Processes	240	520	1040	2040	4080
Recv Nodes	10	24	50	100	368
Msg Vol	102	714	3980	22490	83942

thermal2					
Processes	240	520	1040	2040	4080
Recv Nodes	24	78	218	600	1736
Msg Vol	2738	11716	45408	122062	270122

Figure 3.15: Lassen inter-CPU results. The measured time spent in irregular point-to-point communication for a distributed SpMV for various SuiteSparse matrices. Number of processes across which the problem was partitioned and standard communication maximum number of connected nodes for any single node (Recv Nodes) and message volume included beneath plots. Minimum times circled for convenience.

For Blue Waters (Figure 3.14), Split is the fastest communication strategy more often than any other communication techniques — 19 tests out of 45 test cases, contributing to the node-aware communication strategies performing best in most cases (35 tests). Importantly, for a subset of the cases in which Split was the fastest communication strategy, both 3-Step and 2-Step communication are slower than Standard communication, in-line with model predictions; for example, see `Serena` 2048 processes or `Flan_1565` 256 and 4096 processes. Additionally, there are cases when 3-Step and 2-Step communication exhibits disparate performance, but Split communication still performs the fastest.

Inter-CPU communication performance on Lassen (Figure 3.15) are similar to Blue Waters. Split is often the fastest communication strategy — 23 out of 45 test cases, and node-aware communication strategies out-perform standard communication with a similar frequency (34 out of 45 test cases). However, unlike in the Blue Waters results, we observe no cases in which 3-Step and 2-Step communication both perform worse than Standard communication while Split was the fastest strategy. For most cases, Split is the fastest communication strategy or shows similar performance to the fastest strategy, including Standard communication. Interestingly, there are multiple cases where 3-Step communication is slower than 2-Step communication, but Split is still faster than 2-Step communication — see the top row of Figure 3.15. Overall, the inclusion of Split communication in kernels that require irregular point-to-point communication can greatly reduce the amount of time spent in communication. It is also worth noting that these results are for a single distributed SpMV which, when partitioned across multiple CPU cores, does not result in large message sizes. Further analysis of the benefits of Split communication within the context of high data volumes is discussed in Chapter 4.

3.6.2 Inter-GPU Communication

For each of the communication strategies benchmarked, each GPU is assumed to have a *single host process*, except in the case of "Split + DD" where *four host processes* are used to copy data from each GPU. The number of host processes used to copy data from any GPU is distinct from the number of processes used to communicate inter-node data, in the case of the split strategies. For the split strategies, after data is copied from each GPU via some number of host processes, inter-node communicated data is potentially partitioned across up to 40 processes on-node (the maximum number of on-node processes/cores for Lassen.)

Figure 3.16 displays the distributed SpMV communication benchmark performance times for each communication strategy presented in Section 2.2 for each SuiteSparse matrix. Presented beneath each plot is the number of GPUs on which the SpMV is partitioned, the maximum number of nodes to which any single node is communicating (Recv Nodes), and the communicated inter-node message volume for standard communication.

Consistent with the majority of model predictions for large inter-node message volumes, the staged-through-host communication strategies exhibit far faster performance than the device-aware communication strategies. However, it is worth noting that device-aware 3-Step and device-aware 2-Step are typically much faster than standard device-aware communication. In the case of the `thermal2` matrix, which exhibits a high inter-node message volume for standard communication, the gap between the device-aware node-aware strategies and staged-through-host communication strategies is smaller than for other matrices. Additionally, "Split + DD" consistently performs worse than "Split + MD", consistent with modeled predictions. This is unsurprising considering the latency associated with using duplicate device pointers (∼1.5e-05 in Table 3.4) is much higher than the latency of sending on-socket messages (∼3.7e-07–∼3.2e-06 in Table 3.3) to distribute data being

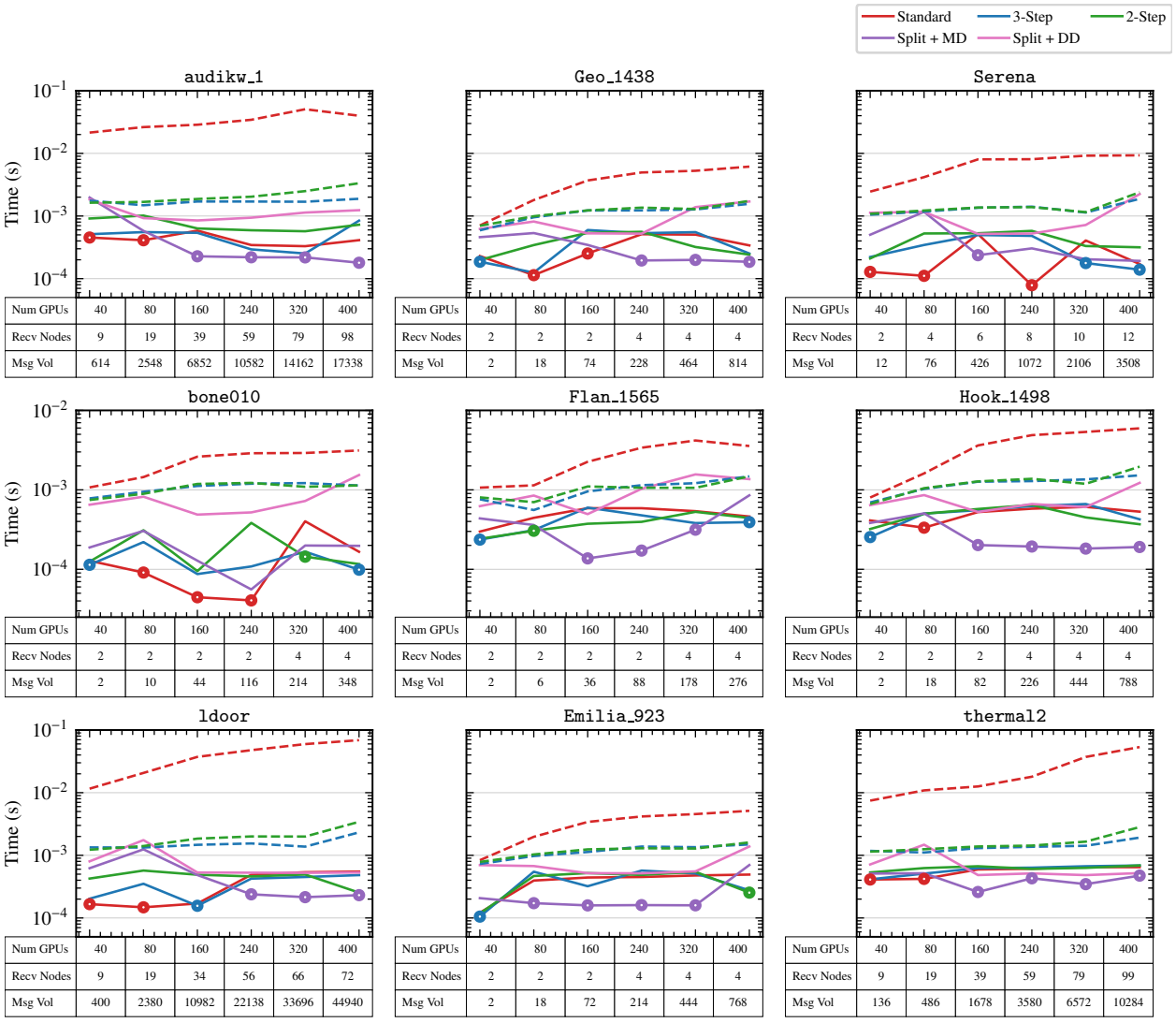


Figure 3.16: Lassen inter-GPU results. The measured time spent in irregular point-to-point communication for a distributed SpMV for various SuiteSparse matrices. Number of GPUs across which the problem was partitioned and standard communication maximum number of connected nodes for any single node (Recv Nodes) and message volume included beneath plots. Solid lines, ——, depict staged-through-host communication, and dashed lines, - - -, depict device-aware communication. Minimum times circled for convenience.

sent from a single GPU across multiple ranks.

The majority of the presented results are similar to the model prediction plots (Figure 3.13), where the fastest communication strategy was typically predicted to be one of the staged-through-host strategies: "Split + MD", Standard, or "3-Step". "Split + MD" exhibits the minimal performing time in most cases, except for smaller counts of participating GPUs (40 or 80 in the case of `audikw_1`, `Serena`, `ldoor`, `thermal2`), or for low inter-node message counts (`bone010`, `Geo_1438`) in which standard communication becomes more optimal.

Overall, staged-through-host node-aware communication strategies demonstrate the best performance for the majority of benchmarks, with "Split + MD" typically being the fastest, consistent with model predictions in Figure 3.13b.

3.7 Conclusions

The advancement of parallel computers has introduced the design of supercomputers with heterogeneous compute nodes due to the inclusion of multiple GPUs per node. For distributed applications, this typically results in larger communicated data volumes, as each compute unit can operate on a larger partition of the problem. In addition to increased data volumes, the inclusion of multiple GPUs per node has increased the complexity of determining optimal data movement paths, particularly in the case of inter-node irregular point-to-point communication.

In this chapter, we characterized the performance of irregular point-to-point communication between CPUs and GPUs via modeling and introduced node-aware communication strategies to inter-node communication on heterogeneous architectures. Our models suggested the use of staged-through-host node-aware communication strategies, specifically split methods were indicated as potential top performers, for inter-GPU communication. These results were confirmed by a performance study on distributed SpMVs which saw split node-aware communication performing best in most cases, and typically much faster than standard device-aware communication. Additionally, the inter-CPU communication results demonstrated that choosing between only Standard, 2-Step, and 3-Step communication is limiting for performance. The inclusion of Split communication greatly extended the ability of node-aware communication schemes to improve irregular point-to-point communication performance.

Importantly, this work also provides groundwork on designing efficient communication strategies for the next generation of supercomputers. Future exascale machine architectures will include higher CPU core counts per node, alongside higher bandwidth interconnects (e.g., on Frontier or El Capitan), two parameters that largely affect the performance of node-aware communication strategies. Based on the presented models, Split communication strategies will likely be the most efficient communication techniques to take advantage of the high bandwidth interconnects, but distributing data to be communicated across a larger number of on-node CPU cores could pose performance constraints. Additionally, the models presented in Section 2.2 naturally extend to architectures with single socket nodes.

Chapter 4: Communication Efficient Enlarged Conjugate Gradients

Portions of this chapter appear in the paper "Performance Analysis and Optimal Node-Aware Communication for Enlarged Conjugate Gradient Methods" published in ACM Transactions on Parallel Computing [58].

4.1 Introduction

The iterative solution to large, sparse linear systems of the form $Ax = b$ often requires *many* sparse matrix-vector multiplications and costly collective communication in the form of inner products; this is the case with the conjugate gradient (CG) method, and with Krylov methods in general. In this chapter, we consider so-called *enlarged* Krylov methods [64], wherein block vectors are introduced to improve convergence, thereby reducing the amount of collective communication in exchange for denser point-to-point communication in the sparse matrix-block vector multiplication. We analyze the associated performance expectations and introduce efficient communication methods that render this class of methods more efficient at scale.

There have been a number of suggested algorithms for addressing the imbalance in computation and communication within Krylov methods, including communication avoidance [3, 4], overlapping communication and computation [6], and delaying communication at the cost of performing more computation [14]. Most recently, there has been work on reducing iterations to convergence via increasing the amount of computation per iteration, and ultimately, the amount of data communicated [15, 65, 66]. These approaches have been successful in reducing the number of global synchronization points; the current work is considered complementary in that the goal is reduction of the total *amount* of communication, which is achieved via message passage restructuring utilizing the MPI API.

In addition to reducing synchronization points, Enlarged Krylov methods such as enlarged conjugate gradient (ECG) reduce the number of sparse matrix-vector multiplications by improving the convergence of the method through an increase in the amount of computation per iteration. This is accomplished by using block vectors, which results in both an increase in (local) computational work, but also an increase in inter-process communication per iteration. Consequently, the focus of this chapter is on analyzing the effects of block vectors on the performance of ECG and proposing optimal strategies to address the communication imbalances they introduce.

There are two key contributions made in this chapter.

1. A performance study and analysis of an enlarged Krylov method based on ECG, with an emphasis on the communication and computation of block vectors. Specifically, we note how they re-balance the point-to-point and collective communication within a single iteration of ECG, shifting the performance bottleneck to the point-to-point communication.

2. The inclusion of Split communication (see Chapter 3) for the blocked data structures in ECG. Introducing Split into the algorithm resluts in speedups as high as 60x for various large-scale test matrices on two different supercomputer systems, as well as, reduces the point-to-point communication bottleneck in ECG.

4.2 Background

The conjugate gradient (CG) method for solving a symmetric and positive definite system of equations, $Ax = b$, exhibits poor parallel scalability in many situations [3, 4, 67, 68]. In particular, the *strong* scalability is limited due to the high volume of collective communication relative to the low computational requirements of the method. The enlarged conjugate gradient (ECG) method has a lower volume of collective communication and higher computational requirements per iteration compared to CG, thus exhibiting better strong scalability. In this section, we detail the basic structure of ECG, briefly outlining the method in terms of mathematical operations and highlighting the key differences from standard CG. A key computation kernel in both Krylov and enlarged Krylov methods is that of a sparse matrix-vector multiplication; node-aware communication techniques for this operation are discussed in Section 2.2 and explored further in Section 4.4. Additionally, Appendix A provides the details on the code written to produce the results within this chapter.

Throughout this section and the remainder of the chapter, ECG performance is analyzed with respect to the problem described in Example 4.1.

Example 4.1. *In this example, we consider a discontinuous Galerkin finite element discretization of the Laplace equation, $-\Delta u = 1$ on a unit square, with homogeneous Dirichlet boundary conditions. The problem is generated using MFEM [69] and the resulting sparse matrix consists of* 1 310 720 *rows and* 104 529 920 *nonzero entries. Graph partitioning is not used to reorder the entries, unless stated.*

4.2.1 Enlarged Conjugate Gradient Method

Similar to CG, ECG begins with an initial guess x_0 and seeks an update as a solution to the problem $Ax = b$, with the initial residual given by $r_0 = b - Ax_0$. Unlike CG, which considers updates of the form $x_k \in x_0 + \mathcal{K}_k$, where $\mathcal{K}_k$ is the Krylov subspace defined as

$$\mathcal{K}_k = \text{span}\{r_0, Ar_0, A^2r_0, \ldots, A^{k-1}r_0\}, \tag{4.1}$$

ECG targets $x_k \in x_0 + \mathcal{K}_{k,t}$ where $\mathcal{K}_{k,t}$ is the *enlarged* Krylov space defined as

$$\mathcal{K}_{k,t} = \text{span}\{T_{r_0,t}, AT_{r_0,t}, A^2T_{r_0,t}, \ldots, A^{k-1}T_{r_0,t}\}, \tag{4.2}$$

with $T_{r,t}$ representing a projection of the residual r (defined next). Notably, the enlarged Krylov subspace contains the traditional Krylov subspace: $\mathcal{K}_k \subset \mathcal{K}_{k,t}$ [66].

In (4.2), $T_{r,t}$ defines a projection of the residual r (normally the initial residual r_0) from $\mathbb{R}^n \to \mathbb{R}^{n \times t}$, by splitting r across t subdomains. The projection may be defined in a number of ways, with the caveat that the resulting columns of $T_{r,t}$ are linearly independent and preserve the row-sum:

$$r = \sum_{i=1}^{t} (T_{r,t})_i, \tag{4.3}$$

where we denote the i^{th} column of T as $(T)_i$. An illustration of multiple permissible splittings is shown in Figure 4.1.

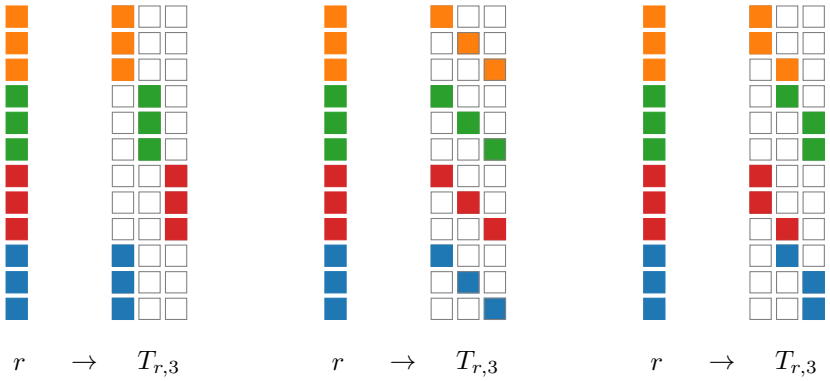


Figure 4.1: Three examples of $T_{r,t}$, with $t = 3$. In each case, r is a vector of length 12, decomposed into a 12×3 block vector. The colors represent a case of four processors: orange, green, red, and blue.

Increasing the number of subdomains, t, increases computation from single vector updates to block vector updates of size $n \times t$. Additional uses of block vectors within ECG are outlined in detail in Algorithm 4.1. On Line 5 a (small) linear system is solved to generate the t search directions. In addition, the sparse matrix-block vector product (SpMBV) AP_k is performed at each iteration. The number of iterations to convergence is generally reduced from that required by CG, but the algorithm does not eliminate the communication overhead when the algorithm is performed at scale. Unlike CG, where the performance bottleneck is caused by the load imbalance incurred from each inner product in the iteration, ECG sees communication overhead at scale due to the communication associated with the SpMBV kernel (see Figures 4.4 and 4.6 in Section 4.3.2). We introduce Split communication method into the kernel in Section 4.4 to improve this performance.

Additionally, Algorithm 4.1 can easily be updated to include preconditioning, represented by application of a preconditioning matrix M^{-1}. Here we summarize the necessary changes to Algorithm 4.1; for a full discussion of preconditioned ECG, see [64]. Initialization of Z is updated to include application of the preconditioner to the split residual vector, $Z = M^{-1}R$, and a single step is added to each iteration in which the preconditioner is applied to the block of vectors AP before Line 11, $M^{-1}AP$. Consequently, Lines 11 to 13 are then updated to include

Algorithm 4.1: Enlarged Conjugate Gradient

Input: A, b, x_0
Output: x

$r := b - Ax_0$
$P := 0$, $X := T_{x,t}$, $R := T_{r,t}$, $Z := R$
while *not converged*
  $P_{\text{old}} := P$
  $P := Z(Z^TAZ)^{-\frac{1}{2}}$
  $c := P^TR$
  $X := X + Pc$
  $R := R - APc$
  if $\|\sum_{i=1}^{t}(R)_i\| < tolerance$
    `break`
  $d := (AP)^T(AP)$
  $d_{\text{old}} := (AP_{\text{old}})^T(AP)$
  $Z := AP - Pd - P_{\text{old}}d_{\text{old}}$
$x := \sum_{i=1}^{t}(X)_i$

Algorithm 4.2: Preconditioning Updates to Enlarged Conjugate Gradient Iteration

$d := (AP)^T(M^{-1}AP)$
$d_{\text{old}} := (AP_{\text{old}})^T(M^{-1}AP)$
$Z := M^{-1}AP - Pd - P_{\text{old}}d_{\text{old}}$

computations involving $M^{-1}AP$ as shown in Algorithm 4.2. As noted by the authors in [64], any preconditioner that can be applied to CG can be applied to ECG, but due to the block structure of the algorithm and the reduction in iterations to convergence, it may not be efficient to use preconditioners that introduce additional communication when performed in parallel, though they may be more robust. Hence, discussion of preconditioner choice and performance is excluded from this work as it is often problem-specific, and the focus in this work is on the characterization and optimization of the performance of the ECG algorithm itself.

4.3 Performance Study and Analysis of ECG

In this section we detail the per-iteration performance and performance modeling of ECG. A communication efficient version of Algorithm 4.1 is implemented in RAPtor [70] and is based on the work in [64]. Throughout this section and the remainder of the thesis, we assume an $n \times n$ matrix A with `nnz` nonzeros is partitioned row-wise across a set of p processes. Each process contains at most $\frac{n}{p}$ contiguous rows of the matrix A. In the modeling that follows, we assume an equal number of nonzeros per partition. In addition, each block vector in Algorithm 4.1 — R, X, Z, AZ, P, and AP — is partitioned row-wise, and with the same row distribution as A. The variables c, d, and d_{old} are size $t \times t$ and a copy of each is stored locally on each process. Tests were performed on the Blue

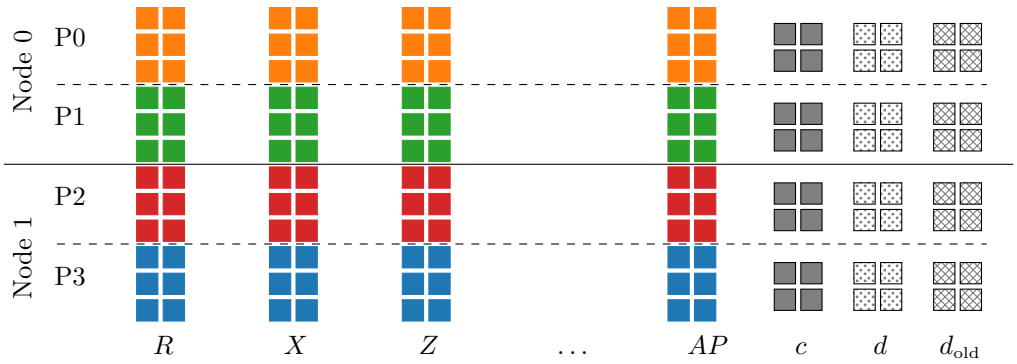


Figure 4.2: The above figure displays the partitioning of all the vectors and intermediary Partition of vectors R, X, Z, AZ, P, and AP along with $t \times t$ working arrays c, d, and d_{old}, as in Algorithm 4.1, for $t = 2$.

Waters Cray XE/XK machine at the National Center for Supercomputing Applications (NCSA) at University of Illinois. Relevant machine specifications for Blue Waters are provided in Table 3.1.

4.3.1 Parallel Implementation

The scalability of a direct implementation of Algorithm 4.1 is limited [64], however, this is improved by fusing communication and by executing the system solve in Line 5 in Algorithm 4.1 on each process. This is accomplished in [64] by decomposing the computation of P into several steps as described in Algorithm 4.3.

Algorithm 4.3: Calculating $P := Z(Z^T AZ)^{-1/2}$

1	$AZ \leftarrow A * Z$	[sparse matrix-block vector multiplication]
2	$Z^T AZ \leftarrow Z^T * AZ$	[block inner product]
3	$C^T C \leftarrow Z^T AZ$	[Cholesky factorization]
4	$P \leftarrow \text{solve } P * C = Z$	[Triangular solve with multiple right sides]

The $t \times t$ product $Z^T(AZ)$ is stored locally on every process in the storage space of c, as shown in Figure 4.2. The Cholesky factorization on Line 3 of Algorithm 4.3 is performed simultaneously on every process, yielding a (local) copy of C. Then each process performs a local triangular system solve using the local vector values of Z to construct the local portion of P (see Line 4). Similarly, an additional sparse matrix block vector product $AP = A * P$ is avoided by noting that AP is constructed using

$$AP \leftarrow \texttt{Triangular Solve with Multiple Right Sides of } AP * C = AZ$$

since the product AZ and the previous iteration's $AP = A * P$ are already stored.

Algorithm 4.4 summarizes our implementation in terms of computational kernels, with the on process computation in terms of floating point operations along with the associated type of

communication. The remainder of the calculations within ECG consist of local block vector updates, as well as block vector inner products for the values c, d, and d_{old}. A straightforward approach is to compute these independently within the algorithm, resulting in four `MPI_Allreduce` global communications per iteration, as in [64]. However, since the input data required to calculate c, d, and d_{old} are available on Line 6 in Algorithm 4.1 when c is computed, a single global reduction is possible. The implementation described in Algorithm 4.4 highlights a single call to `MPI_Allreduce` for all of these values and reducing them in the same buffer. This reduces the number of `MPI_Allreduce` calls to two per iteration. Additionally, the point-to-point communication required for the SpMBV is performed using the standard communication approach described in Section 2.2 and used in standard industry codes, such as PETSc [71].

Algorithm 4.4: Enlarged Conjugate Gradient by Kernel

```
1  SpMV
2  Vector Initialization
3  for k = 1, ...
4  |  SpMBV                    → Flops: 2 · (nnz/p) · t      Comm: point-to-point
5  |  Block inner product      → Flops: 2 · (n/p) · t^2      Comm: global all reduce
6  |  Cholesky decomposition   → Flops: (1/3) · t^3
7  |  Triangular solves        → Flops: 2 · (1/2) · t^2
8  |  Block inner product      → Flops: 2 · (n/p) · t^2      Comm: global all reduce
9  |  Block vector addition    → Flops: 2 · (n/p) · t
10 |  Block vector axpy        → Flops: 2 · (n/p) · t
```

From Algorithm 4.4, we note that computation and communication per iteration costs of ECG have increased over that of parallel CG, with computation in terms of floating point operations and the type of communication incurred next to each kernel. For our implementation, the number of collective communication calls to `MPI_Allreduce` has remained the same as CG (at two), but the number of values in the global reductions has increased from a single float in each of CG's global reductions to t^2 and $3t^2$. The singular SpMV from CG has increased to a sparse matrix block vector product (SpMBV), which does not increase the number of point-to-point messages, but does increase the size of the messages being communicated by the enlarging factor t. This SpMBV is performed the same as the SpMV presented in Figure 2.1 with the local portion of the SpMBV computed while messages are exchanged to attain off-process data. Additionally, the local computation for *each kernel* has increased by a factor of t. ECG uses these extra per iteration requirements to reduce the total number of iterations to convergence, resulting in fewer iterations than CG as seen in Figure 4.3.

4.3.2 Per Iteration Performance

In Figure 4.4 we decompose the performance of a single iteration of ECG for Example 4.1 into (local) computation, point-to-point communication, and collective communication. Performance

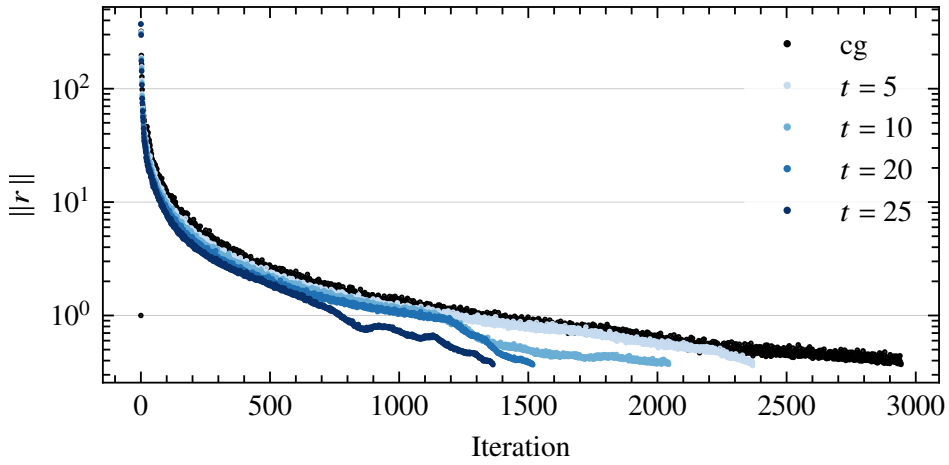


Figure 4.3: Residual history for CG and ECG with various enlarging factors t for Example 4.1.

tests were executed on Blue Waters [59, 60]. Each test is the average of 20 iterations of ECG; reported times are the maximum average time recorded for any single process. At small scales, local computation dominates performance, while at larger scales, the point-to-point communication in the single SpMBV kernel and the collective communication in the block vector inner products become the bottleneck in ECG. Figure 4.4 also shows the time spent in a single inner product. While we observe growth with the number of processes, as expected, the relative cost (and growth) within ECG remains low. Importantly, increasing t at high processor counts does not significantly contribute to cost. This is shown in Figure 4.5, where the mean runtime for various block vector inner products all fall within each other's confidence intervals. This suggests that increasing t to drive down the iteration count will have little effect on the per iteration cost of the two calls to `MPI_Allreduce`, and in fact, will result in fewer total calls to it due to the reduction in iterations.

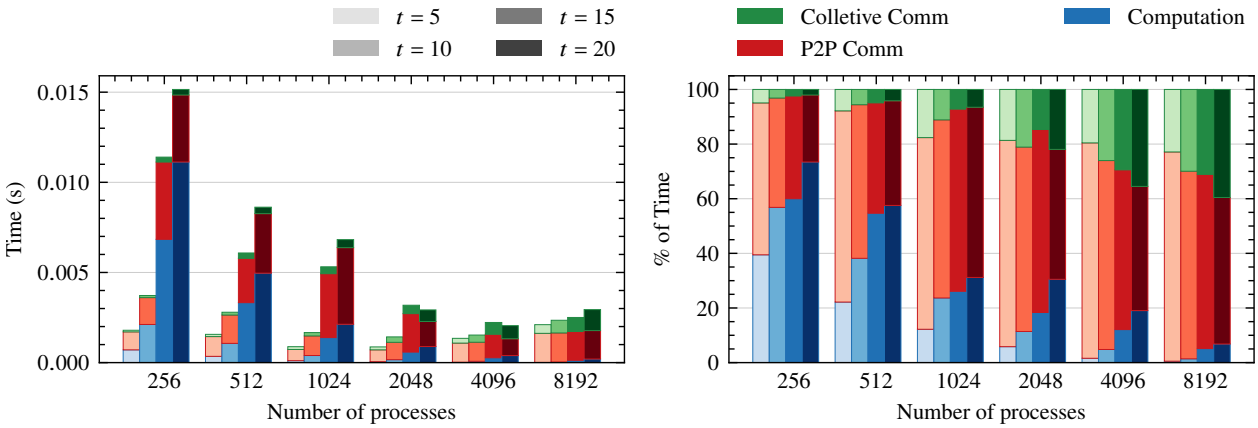


Figure 4.4: Time (left) and percentage of time (right) for a single iteration of ECG for various block vector sizes, t, and processor counts for Example 4.1.

We would also like to note that on newer systems, such as Lassen, a much larger percentage of time is attributed to the point-to-point communication in the SpMBV kernel than for Blue

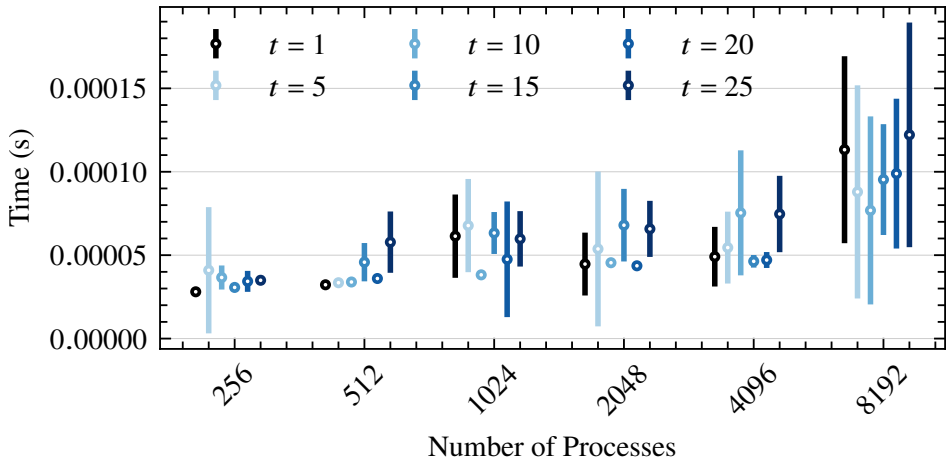


Figure 4.5: Time for a single inner product (right) for various block vector sizes, t, and processor counts for Example 4.1. Vertical lines denote the confidence intervals.

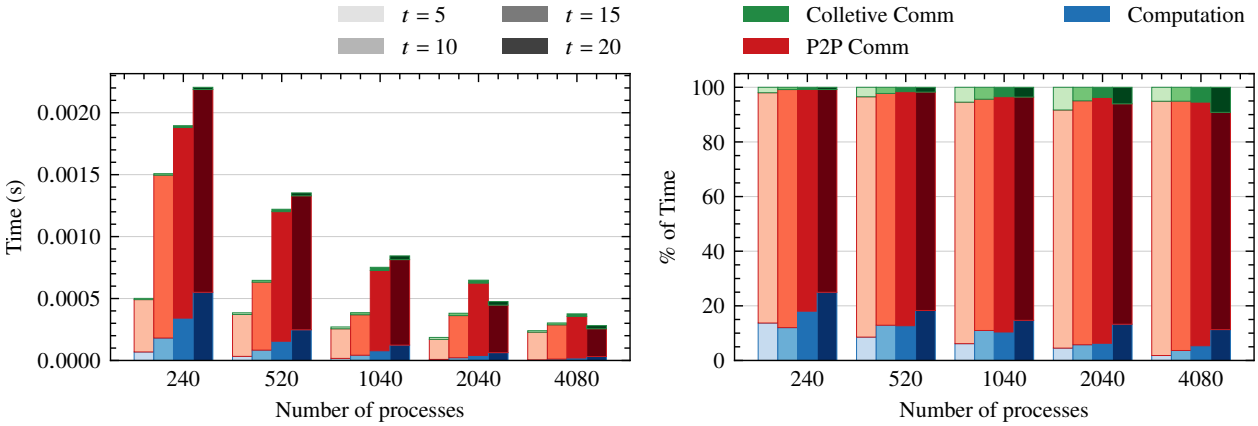


Figure 4.6: Time (left) and percentage of time (right) for a single iteration of ECG for various block vector sizes, t, and processor counts for Example 4.1 on Lassen.

Waters — see Figure 4.6.

The remainder of this section focuses on accurately predicting the performance of a single iteration of ECG through robust communication performance models. In particular, SpMBV communication is addressed in detail in Section 4.4 where we discuss the use of Split communication for blocked data.

4.3.3 Performance Modeling

To better understand the timing profiles in Figures 4.4 and 4.6, we develop performance models. Below, we present two different models for the performance of communication within a single iteration of ECG. First, consider the standard postal model for communication, which represents the maximum amount of time required for communication by an individual process in a single

iteration of ECG as

$$T_{\text{postal}} = \underbrace{\alpha \cdot m + \frac{s \cdot t}{R_b}}_{\text{point to point}} + \underbrace{2\alpha \cdot \log(p) + \frac{f \cdot 4t^2}{R_b}}_{\text{collective}} \tag{4.4}$$

where f is the number of bytes for a floating point number — e.g. $f = 8$. See Table 2.1 for a complete description of model parameters. As discussed in Section 2.1, this model presents a misleading picture on the performance of ECG at scale, particularly on current supercomputer architectures where SMP nodes encounter injection bandwidth limits when sending inter-node messages.

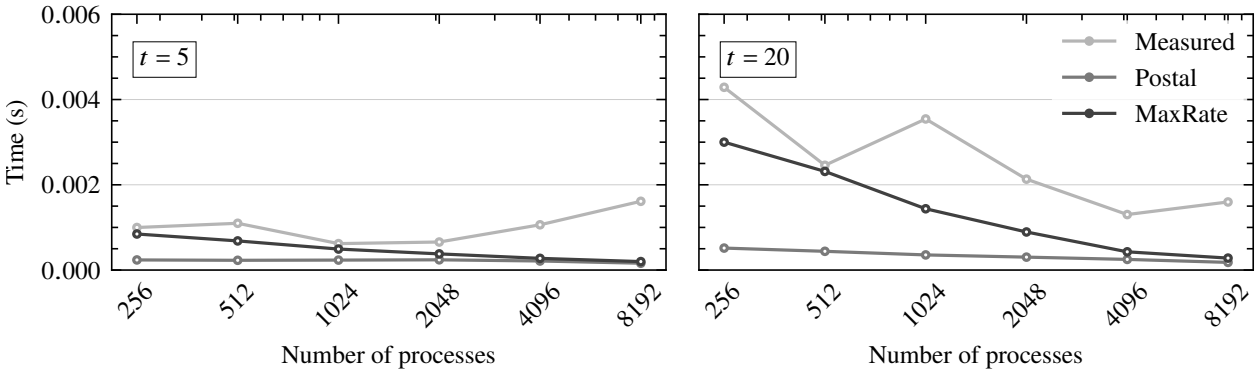


Figure 4.7: Max-rate model versus the postal model for the point to point communication in one iteration of ECG for Example 4.1 using various t on Blue Waters. Measured runtimes are also included. (*note:* $t = 5$ and $t = 20$ shown for brevity.)

To improve the model we drop in the max-rate model for the point-to-point communication, resulting in

$$T_{MR} = \underbrace{\alpha \cdot m + \max\left(\frac{\texttt{ppn} \cdot s \cdot t}{R_N}, \frac{s \cdot t}{R_b}\right)}_{\text{point to point}} + \underbrace{2\alpha \cdot \log(p) + \frac{f \cdot 4t^2}{R_b}}_{\text{collective}}. \tag{4.5}$$

Figure 4.7 shows that the max-rate model provides a more accurate upper bound on the time spent in point to point communication within ECG. The term $2 \cdot \alpha \cdot \log(p) + \frac{f \cdot 4 \cdot t^2}{R_b}$ remains unchanged in Equations (4.4) and (4.5) to represent the collective communication required for the two block vector inner products. Each block vector inner product incurs latency from requiring $\log(p)$ messages in an optimal implementation of the `MPI_Allreduce`. More accurate models for modeling the communication of the `MPI_Allreduce` in the inner product exist, such as the logP model [25] and logGP model [26], but optimization of the reduction is outside the scope of this thesis, so we leave the postal model for representing the performance of the inner products.

Modeling the computation within an iteration of ECG is straightforward. The computation for a single iteration of ECG is written as the sum of the kernel floating point operations

procs →	256		512		1024		2048		4096		8192	
t	m-s	std	m-s	std	m-s	std	m-s	std	m-s	std	m-s	std
5	42.2	55.6	57.9	70.0	71.7	70.2	80.1	75.5	83.5	78.9	81.9	76.6
10	21.2	40.0	34.5	56.2	51.2	65.2	65.9	67.5	75.9	69.1	77.7	68.7
15	13.0	37.6	22.2	40.4	35.2	66.7	48.6	67.0	60.0	58.5	60.5	63.8
20	9.3	24.5	16.5	38.3	27.4	62.3	40.6	47.6	54.4	45.5	57.2	53.6

(a) 2-Step Communication

procs →	256		512		1024		2048		4096		8192	
t	m-s	std	m-s	std	m-s	std	m-s	std	m-s	std	m-s	std
5	29.6	55.6	43.0	70.0	54.8	70.2	64.2	75.5	72.3	78.9	71.4	76.6
10	14.6	40.0	24.4	56.2	35.9	65.2	49.5	67.5	65.6	69.1	68.5	68.7
15	9.2	37.6	15.8	40.4	23.9	66.7	34.4	67.0	49.9	58.5	50.8	63.8
20	6.7	24.5	11.9	38.3	18.6	62.3	28.7	47.6	45.9	45.5	48.8	53.6

(b) 3-Step Communication

Table 4.1: Modeled percentage of time to be spent in point-to-point communication for multistep ("m-s") compared against the measured percentage of time spent in point-to-point communication for standard ("std") in a single iteration of ECG for Example 4.1 with varying t and processor counts on Blue Waters. Blue values correspond to values for which the percentage of time decreased by $5\% - 20\%$, green values are a decrease of over 20%, and red values predict an increase over the original percentage.

in Algorithm 4.4, which results in the following

$$T_{comp} = \gamma \cdot \left((2t)\frac{\texttt{nnz}}{p} + (4t + 4t^2)\frac{n}{p} + \frac{1}{2}t^2 + \frac{1}{3}t^3 \right) \tag{4.6}$$

where γ is the time required to compute a single floating point operation. More accurate models, such as the roofline model, exist for predicting peak computational performance, as well as, identifying computational bottlenecks [72]. However, because point-to-point communication is the overwhelmingly dominate cost in strong-scaling performance of ECG, we maintain a simple computational model. In total, we arrive at the following model for a single iteration of ECG

$$T_{ECG} = \alpha \cdot m + \max\left(\frac{\texttt{ppn} \cdot s \cdot t}{R_N}, \frac{s \cdot t}{R_b} \right) + 2\alpha \cdot \log(p) + \frac{f \cdot 4t^2}{R_b} + \gamma \cdot \left((2t)\frac{\texttt{nnz}}{p} + (4t + 4t^2)\frac{n}{p} + \frac{1}{2}t^2 + \frac{1}{3}t^3 \right). \tag{4.7}$$

Using this model, we can predict the reduction in the amount of time spent in point-to-point communication using the multi-step communication techniques presented in Section 2.2 for a single iteration of ECG for Example 4.1 — see Table 4.1.

We see that ECG is still limited at large processor counts even when substituting the node-aware communication techniques to replace the costly point-to-point communication observed in Figure 4.4. The models do predict a large amount of speedup for most cases, however, when

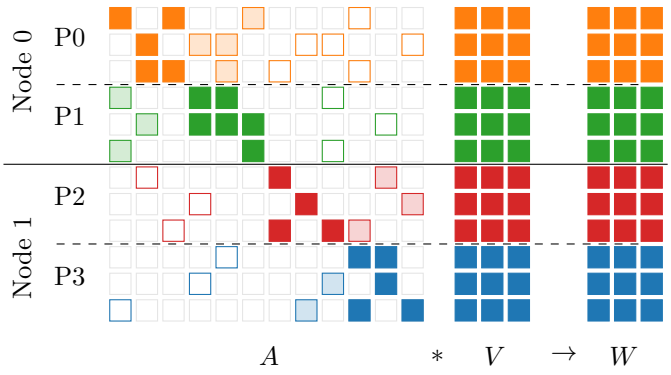


Figure 4.8: Sparse matrix-block vector multiplication (cf. Figure 2.1).

using 3-step communication, suggesting that node-aware communication techniques can reduce the large point-to-point bottleneck observed in the performance study. While a large communication cost stems equally from the collective communication the `MPI_Allreduce` operations, their performance is dependent upon underlying `MPI` implementation and outside the scope of this thesis. We address the point-to-point communication performance further in Section 4.4 by analyzing it through the lens of node-aware communication techniques, optimizing them to achieve the best possible performance at scale.

4.4 Efficient Communication in Parallel SpMBV using Split Node-Aware

As discussed in Section 4.3.2, scalability for ECG is limited by the sparse matrix-block vector multiplication (SpMBV) kernel defined as

$$A * V \rightarrow W, \tag{4.8}$$

with $A \in \mathbb{R}^{n \times n}$ and V, $W \in \mathbb{R}^{n \times t}$, where $1 < t \ll n$. Due to the block vector structure of V, each message in a SpMV is increased by a factor of t (see Figure 4.8). The larger messages associated with $t > 1$ increase the amount of time spent in the point-to-point communication at larger scales, making the SpMBV operation an ideal candidate for node-aware messaging approaches.

Additionally, the SpMBV kernel is a key computational kernel within block Krylov subspace methods [65]. While block Krylov methods are slightly different from *enlarged* Krylov methods, in that they solve a linear system with multiple right hand sides, and *enlarged* Krylov methods project an initial residual into a larger subspace to solve a single system, they both require a SpMBV computational kernel. It is worth noting that the techniques presented in the following sections impact the parallel performance of block Krylov subspace methods, via reduction of time spent in point-to-point communication. However, we do not analyze per iteration performance impacts outside the context of ECG.

Equally important to mention is work done on improving the parallel performance of distributed SpGEMMs and SpMMs, since a SpMBV is technically a form of SpMM. A number of partitioning

techniques have been developed to alleviate the overhead of point-to-point communication within the SpGEMM and SpMM. Here, we highlight relevant work in communication reduction. Optimal hypergraph partitioning has shown to decrease time spent in communication, when the sparsity patterns of input and output matrices are known [8, 9]. Additionally, 2.5D and 3D matrix partitionings can reduce communication times even further by balancing communication costs with memory usage [10, 11, 12, 13]. In [73], various communication optimization techniques for SpGEMMs and SpMVs were evaluated for use within SpMMs, demonstrating the importance the size of the dense matrix has on communication performance. Here, the approach is to reduce the communication overhead by applying node-aware communication techniques.

4.4.1 Performance Modeling

Recalling the node-aware communication models from Section 2.2.3, we augment the 2-step and 3-step models with block vector size, t. As a result, the 2-step model in (2.6) becomes

$$T_{total} = \underbrace{\alpha \cdot m_{\texttt{proc}\to\texttt{node}} + \max\left(\frac{t \cdot s_{\texttt{node}}}{R_N}, \frac{t \cdot s_{\texttt{proc}}}{R_b}\right)}_{\text{inter-node}} + \underbrace{\alpha_\ell \cdot (\texttt{ppn} - 1) + t \cdot \frac{s_{\texttt{proc}}}{R_{b,\ell}}}_{\text{intra-node}} \tag{4.9}$$

while the 3-step model in (2.5) becomes

$$T_{total} = \underbrace{\alpha \cdot \frac{m_{\texttt{node}\to\texttt{node}}}{\texttt{ppn}} + \max\left(\frac{t \cdot s_{\texttt{node}}}{R_N}, \frac{t \cdot s_{\texttt{proc}}}{R_b}\right)}_{\text{inter-node}} + \underbrace{2 \cdot \left(\alpha_\ell \cdot (\texttt{ppn} - 1) + t \cdot \frac{s_{\texttt{node}\to\texttt{node}}}{R_{b,\ell}}\right)}_{\text{intra-node}}. \tag{4.10}$$

In both models, t impacts maximum rate, a term that is relatively small for $t = 1$ and dominates the inter-node portion of the communication as t grows. Since messages are increased by a factor of t, the single buffer used in 3-step communication quickly reaches the network injection bandwidth limits. Using multiple buffers, as in 2-step communication, helps mitigate the issue, however more severe imbalance persists since the amount of data sent to different nodes is often widely varying.

Figure 4.9 shows every inter-node message sent by a single process alongside the message size for Example 4.1 when performing the SpMBV kernel with 4096 processes and 16 processes per node. We present the message sizes for 3-step and 2-step communication, noting that the overall number of inter-node messages decreases when using 3-step communication, but the average message size sent by a single process increases. For $t = 20$, the maximum message size nears 10^6 for 3-step communication, while the maximum message size barely reaches 10^5 for 2-step communication. Additionally, there is clear imbalance in the inter-node message sizes for 2-step communication with messages ranging in size from 10^3–10^5 Bytes.

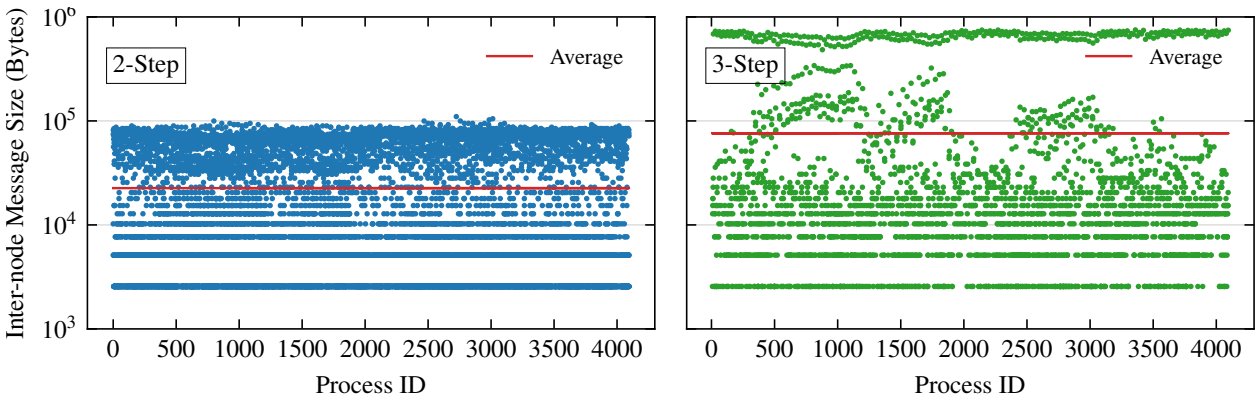


Figure 4.9: The inter-node message sizes across all processes for 2-Step and 3-step communication when performing the SpMBV for Example 4.1 when $t = 20$ across 4096 processes on Blue Waters. The average message size across all processes is marked by a red line.

4.4.2 Performance Profiling

We next apply the node-aware communication strategies presented for SpMVs and SpGEMMs in Section 2.2 to the SpMBV kernel within ECG.

Figure 4.10 displays the performance of standard, 2-step, and 3-step communication when applied to the SpMBV kernel for Example 4.1 with $t = 5$ and $t = 20$. The 2-step communication appears to outperform the others in most cases. This is due to the large amount of data to be sent off-node that is split across many processes. We see 2-step communication performing better due to the term $\alpha \cdot m_{\texttt{proc}\to\texttt{node}}$ in (4.9) being smaller than the $\alpha \cdot m_{\texttt{node}\to\texttt{node}}$ term in the 3-step communication model (4.10) due to multiplication by the factor t. In fact, all counts measured in the ECG profiling section are now multiplied by the enlarging factor t. While the traditional SpMV shows speedup with 3-step communication, we now see that 2-step is generally the best fit for our methods as message size, and thus t, increases.

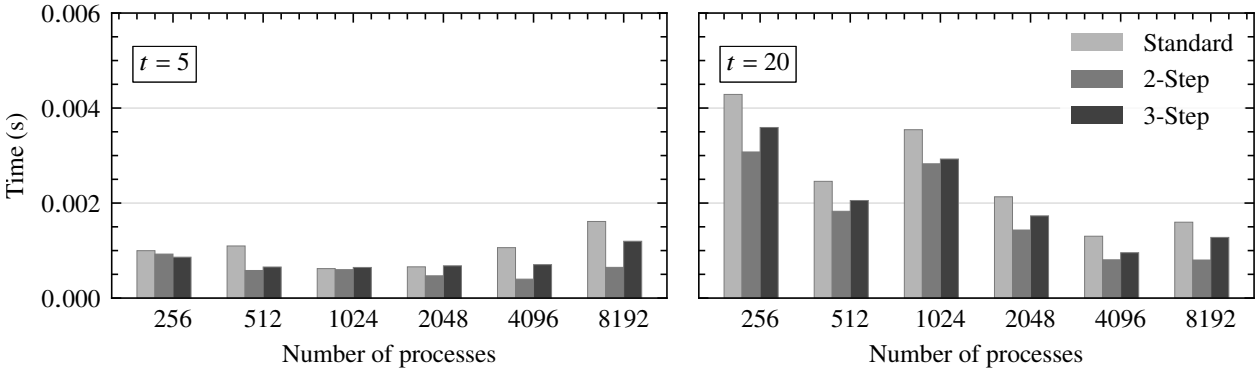


Figure 4.10: Strong scaling results for Example 4.1 using standard, 2-step, and 3-step communication in the SpMBV (*note:* $t = 5$ and $t = 20$ shown for brevity).

matrix	rows/cols	nnz	nnz/row	density
audikw_1	943 695	77 651 847	82.3	8.72e-05
Geo_1438	1 437 960	60 236 322	41.9	2.91e-05
bone010	986 703	47 851 783	48.5	4.92e-05
Emilia_923	923 136	40 373 538	43.7	4.74e-05
Flan_1565	1 565 794	114 165 372	72.9	4.66e-05
Hook_1498	1 498 023	59 374 451	39.6	2.65e-05
ldoor	952 203	42 493 817	44.6	4.69e-05
Serena	1 391 349	64 131 971	46.1	3.31e-05
thermal2	1 228 045	8 580 313	7.0	5.69e-06

Table 4.2: Test Matrices.

Next we consider node-aware communication performance results for a subset of the largest matrices in the SuiteSparse matrix collection [63] (matrix details can be found in Table 4.2). These were selected based on size and density to provide a variety of scenarios.

While 2-step communication is effective in many instances, it is not always the most optimal communication strategy, as depicted in Figure 4.11. Unlike the results for Example 4.1, 3-step and 2-step communication do not always outperform standard communication, and in some instances (4096 processes), for most values of t there is performance degradation. This is seen more clearly in Figure 4.12.

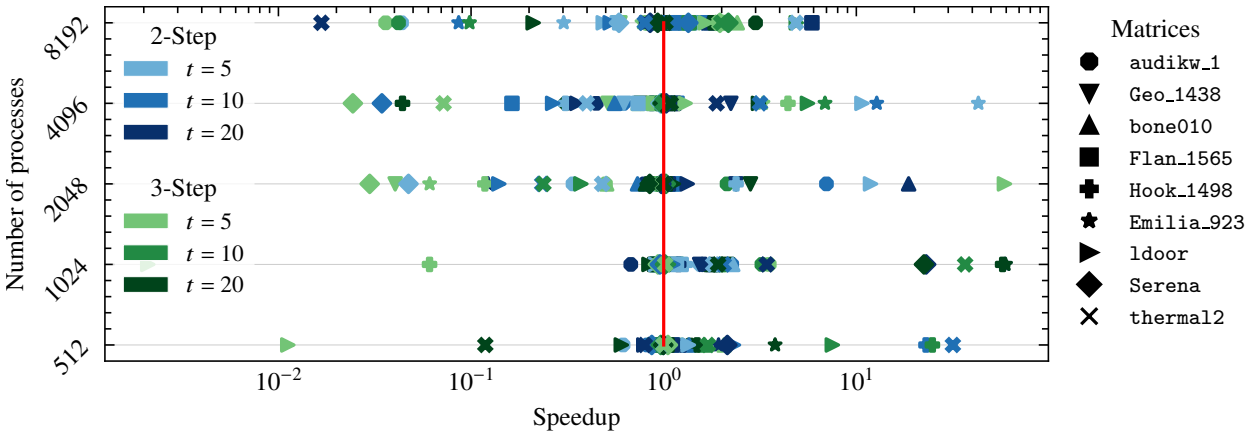


Figure 4.11: Speedup of 2-step and 3-step communication over standard communication in the SpMBV for various matrices from the SuiteSparse matrix collection on Blue Waters. The red line marks 1.0, or no speedup.

It is important to highlight cases where only a single node-aware communication technique results in performance deterioration over standard communication. Distinct examples include Geo_1438 and thermal2 on 4096 processes. Both of these matrices benefit from 2-step communication for $t = 10$ and 20, but performance degrades when using 3-step communication. Another example is the performance of ldoor on 8192 processes (see Figure 4.12). For $t = 5$ and 10, ldoor results in performance degradation using 2-step communication, but up to 5× speedup

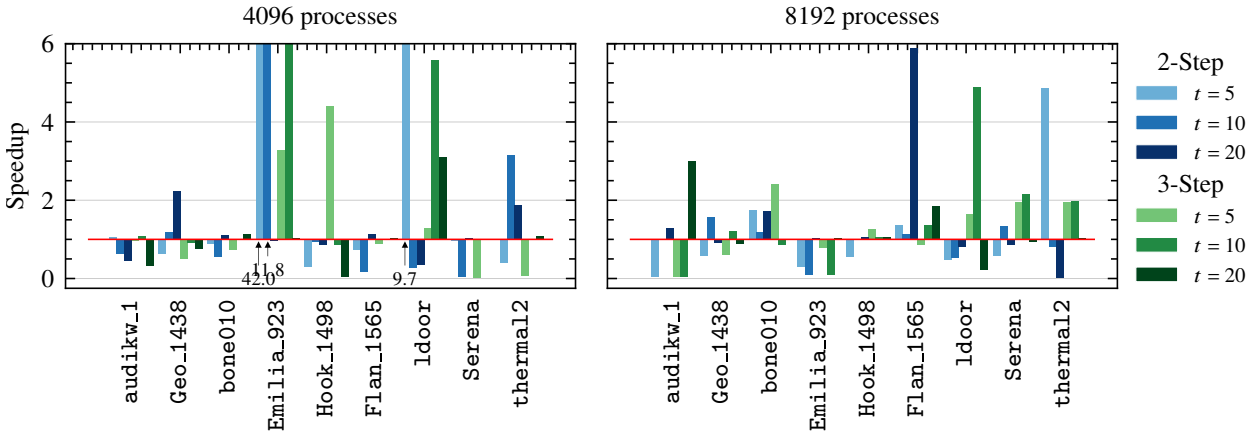


Figure 4.12: Speedup of 2-step and 3-step communication over standard communication in the SpMBV for various matrices on Blue Waters for 4096 and 8192 processes. The red line marks 1.0, or no speedup. Speedups greater than 6.0 are given via annotations on the plot.

over standard communication when using 3-step communication. These cases highlight that while one node-aware communication technique underperforms in comparison to standard communication, the other node aware technique is still much faster. Using this as the key motivating factor, we discuss using a Split node-aware communication technique for blocked data.

4.4.3 Optimal Communication with Split Node-Aware

When designing an optimal communication scheme for the blocked data format, the main consideration is the impact the number of vectors within the block have on the size of messages communicated. The effects message size and message number have on performance can vary based on machine, hence we present results for Blue Waters alongside Lassen.

In Section 3.2, we saw the effects placement of data and amount of data being communicated has on performance times. We demonstrated that as the number of bytes communicated between two processes increases, it becomes increasingly important whether those two processes are located on the same socket, node, or require communication through the network. For both machines, inter-node communication is fastest when message sizes are small, and there are few messages being injected into the network. On Blue Waters, intra-node communication is the fastest, with the time being dependent on how physically close the processes are located. For instance, when two processes are on the same socket, communication is faster than when they are on the same node, but different sockets. This is true for Lassen, as well, but cross-socket intra-node communication is not always faster than communicating through the network. Once message sizes exceed 10^4 bytes, and there are fewer than five processes actively communicating, inter-node communication is faster than two cross-socket intra-node processes communicating. For both machines, however, we see that once a large enough communication volume is reached, it becomes faster to split the inter-node data being sent across a subset of the processes on a single node due to network contention as observed in [62].

In addition to the importance of the placement of two communicating processes, the total message volume and number of actively communicating processes plays a key role in communication performance. While it is extremely costly for every process on a node to send 10^5 bytes as seen in the left plots of Figures 3.1 and 3.2, there are performance benefits when splitting a large communication volume across all processes on a node, depicted in the right plots of Figures 3.1 and 3.2. Blue Waters sees modest performance benefits when splitting large messages across multiple processes, whereas Lassen sees much greater performance benefits.

These observations help justify why 3-step communication would outperform 2-step communication, and vice versa in certain cases of the SpMBV profiling presented in Figure 4.7. Sending all messages in a single buffer becomes impractical when the block size, t, is very large, but having each process communicate with a paired process also poses problems when some of the inter-node messages being sent are still very large, as seen in Figure 4.9.

Motivated by the results above, we introduce the Split communication process to the SpMBV. We reduce the number of inter-node messages and conglomerate messages to be sent off node for certain cases when the message sizes to be sent off-node are below the rendezvous protocol cutoff, and we split the messages to be sent off-node across multiple processes when the message size is larger than the rendezvous protocol cutoff. Hence, each node is determining the most optimal way to perform its inter-node communication.

The resulting speedups for the two systems Blue Waters and Lassen when using Split communication are presented in Figure 4.13.

The method sees speedup for *some* test matrices and performance degradation for most on Blue Waters, which is consistent with the minimal speedup seen by splitting large messages across multiple processes in Figure 3.1. Additionally, it is likely that network contention is playing a large role in the Blue Waters results as the message sizes become large based on the findings in [62], and due to each node determining independently how to send its own data without consideration of the size or number of messages injected by other nodes.

The Split communication performs better on Lassen than Blue Waters and aligns with expectations based on the combination of using 3-step for nodes with small inter-node messages to inject into the network where on-node communication is faster than network communication and splitting large messages where the benefits are much greater. While Blue Waters achieves higher speedups in some cases than Lassen, both systems see speedups as large as 60x. These results only present part of the overall communication picture, however. There are still cases where the global communication strategy should be reduced to 3-step communication, 2-step communication, or standard communication. This differs based on the two machines and the specific test matrix, implying the use of tuning between the techniques will result in use of the most optimal communication strategy. Overall results including tuning between node-aware communication strategies and their affect on a single iteration of ECG are presented in Section 4.5.

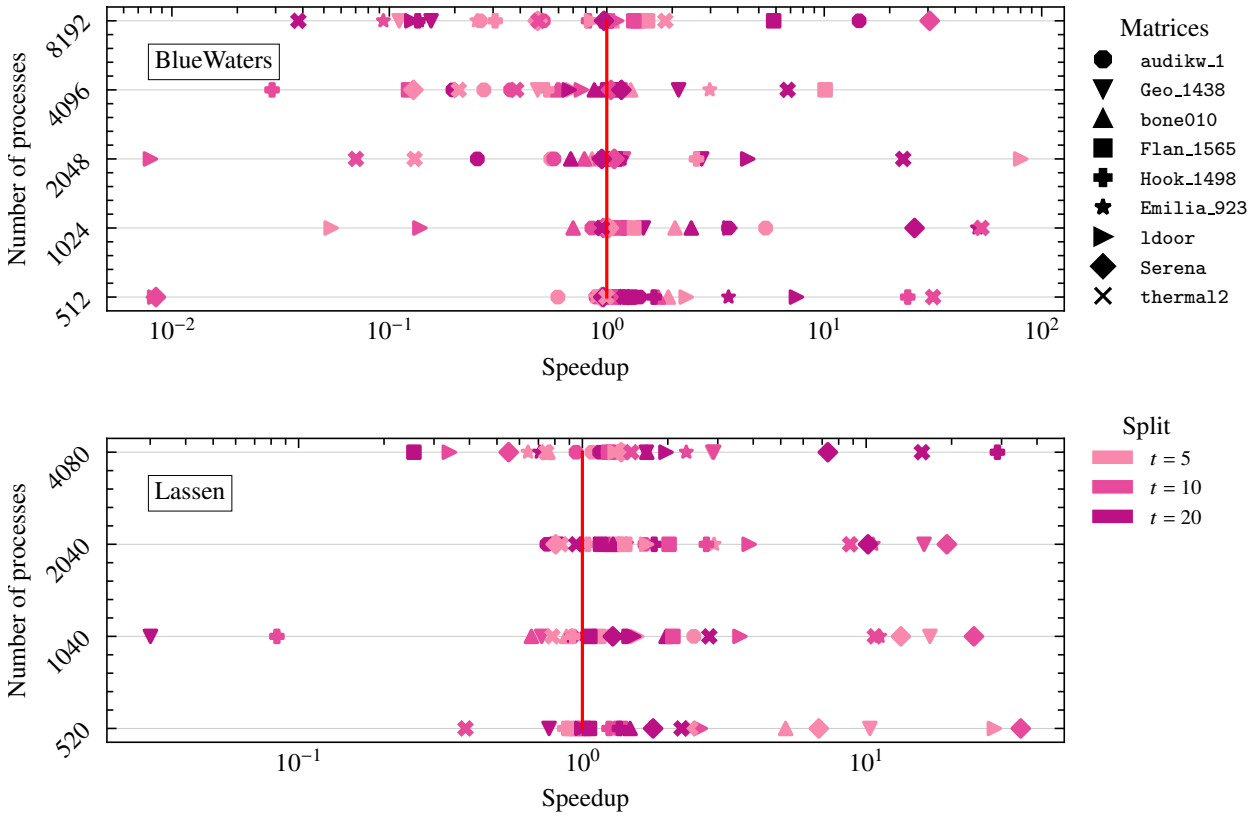


Figure 4.13: Speedup of Split communication over standard communication (without reducing to a global communication strategy) in the SpMBV for various matrices from the SuiteSparse matrix collection on Blue Waters and Lassen. The red line marks 1, or no speedup.

4.5 Overall Results

Tuning between standard and the node-aware communication strategies (including Split) comes at the minimal cost of performing four different SpMBVs during setup of the SpMBV communicator. Speedups over standard communication when using tuning to use the fastest communication technique are presented in Figure 4.14.

In the top plot of Figure 4.14, Blue Waters benefits in 33% of the cases from including the Split communication strategy. These results are expected based on Figure 4.13 where the benefits of Split communication were less than ideal. In fact, for 20% of the cases on Blue Waters, standard communication is the most performant, consistent with matrices of low density. The matrices `Geo_1438` and `ldoor`, which have the smallest density of the test matrices (Table 4.2), see minimal benefits from the multi-step communication techniques as the number of processes is scaled up due to the minimal amount of data being communicated.

We expected to see 2-step and Split communication perform the best on Lassen due to the faster inter-node communication for smaller sized messages and the benefits of splitting messages across many processes on node (Figure 3.2). These expectations are consistent with the results presented in the bottom plot of Figure 4.14; most test matrices saw the best SpMBV performance

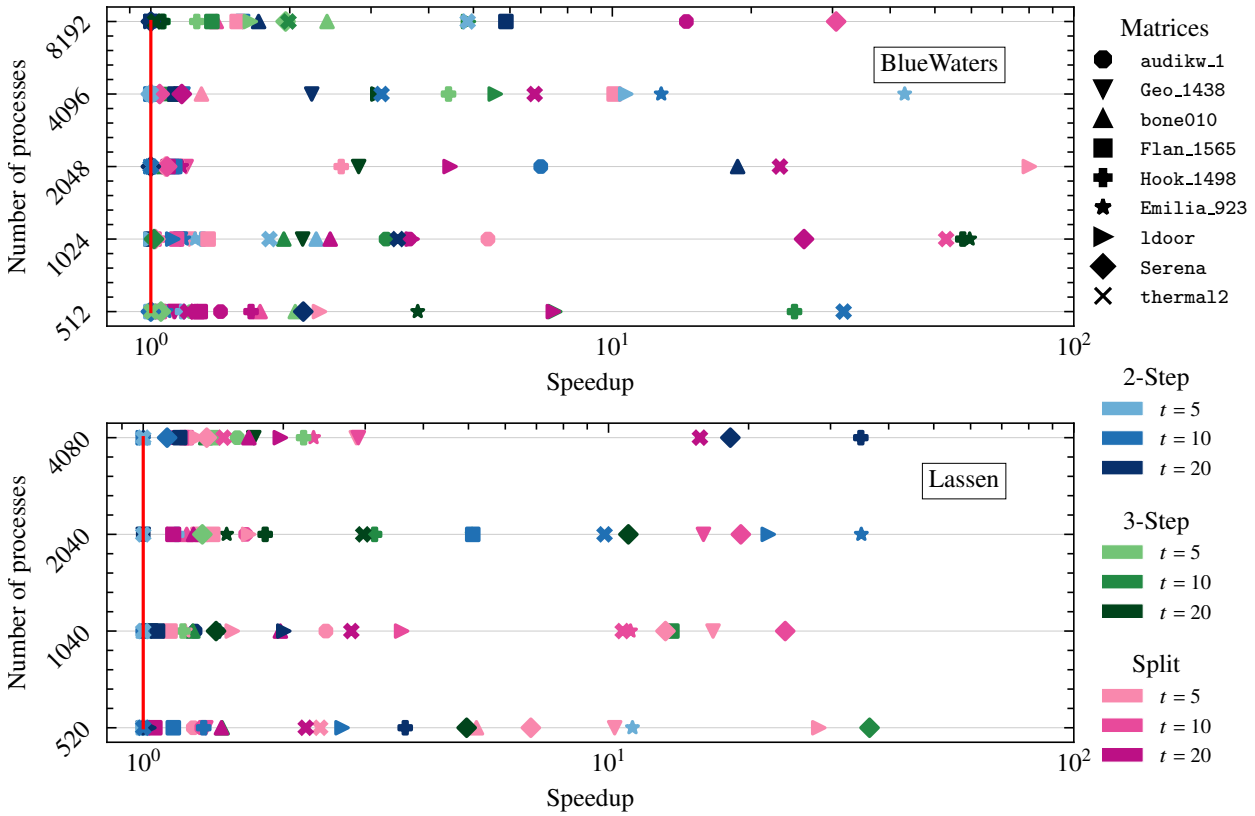


Figure 4.14: Speedup of tuned communication over standard communication in the SpMBV for various matrices from the SuiteSparse matrix collection on Blue Waters and Lassen. The red line marks 1.0, or no speedup.

with Split communication (44% of the cases). The remainder of the test cases were divided almost equally between 2-step, 3-step, and standard communication for which technique was the most performant (approximately 18% of the cases, each).

Table 4.3 shows the benefits of using the tuned point-to-point communication over the standard communication in a single iteration of ECG for Example 4.1. Tuned communication reduces the percentage of time spent in point-to-point communication independent of system, though the performance benefits are typically best when more data is being communicated ($t = 20$ in Table 4.3a and Table 4.3b.) For Blue Waters, the new communication technique results in point-to-point communication taking $20\% - 40\%$ less of the total time compared to the percentage of time when using standard communication (corresponding to the blue highlighted values in Table 4.3a). Performance benefits are much greater on Lassen where the tuned communication results in a decrease of more than 40% of the total iteration time compared to an iteration time with standard communication in most cases.

procs →	256		512		1024		2048		4096		8192	
t	m-s	std	m-s	std	m-s	std	m-s	std	m-s	std	m-s	std
5	54.7	55.6	34.1	70.0	66.2	70.2	70.6	75.5	70.2	78.9	41.7	76.6
10	31.8	40.0	39.3	56.2	34.6	65.2	48.4	67.5	59.7	69.1	32.2	68.7
15	9.3	37.6	18.2	40.4	30.4	66.7	31.9	67.0	38.4	58.5	26.0	63.8
20	7.5	24.5	15.0	38.3	26.3	62.3	46.0	47.6	30.8	45.5	29.7	53.6

(a) Blue Waters

procs →	240		520		1040		2040		4080	
t	m-s	std	m-s	std	m-s	std	m-s	std	m-s	std
5	49.4	84.3	43.6	88.0	66.6	88.4	82.2	87.2	82.7	93.1
10	26.8	87.2	33.7	84.9	52.1	84.7	58.4	89.3	78.3	91.3
15	9.9	81.2	26.4	85.7	33.6	86.1	55.3	90.0	61.2	89.2
20	12.7	74.2	17.8	79.9	29.6	81.7	39.0	80.7	57.2	79.5

(b) Lassen

Table 4.3: Measured percentage of time spent in point-to-point communication for the tuned multistep ("m-s") compared against the measured percentage of time spent in point-to-point communication for standard ("std") in a single iteration of ECG for Example 4.1 with varying t and processor counts on Blue Waters and Lassen. Blue values correspond to values for which the percentage of time decreased by $5\% - 20\%$, green values are a decrease of $20\% - 40\%$, and yellow values decreased by more than 40% from the original percentage.

4.6 Conclusions

The enlarged conjugate gradient method (ECG) is an efficient method for solving large systems of equations designed to reduce the collective communication bottlenecks of the classical conjugate gradient method (CG.) Within ECG, block vector updates replace the single vector updates of CG, thereby reducing the overall number of iterations required for convergence and hence the overall amount of collective communication.

In this chapter, we performed a performance study and analysis of the effects of block vectors on the balance of collective communication, point-to-point communication, and computation within the iterations of ECG. We noted the increased volume of data communicated and its disproportionate affects on the performance of the point-to-point communication; the communication bottleneck of ECG shifted to be the point-to-point communication within the sparse matrix-block vector kernel (SpMBV). To address the new SpMBV bottleneck, we utilized the Split communication technique that builds on existing node-aware communication techniques and improves them for the emerging supercomputer architectures with greater numbers of processes per node and faster inter-node networks.

Overall, this chapter provides a comprehensive study of the performance of ECG in a distributed parallel environment, and introduces a novel point-to-point multi-step communication technique into the SpMBV that provides consistent speedup over standard communication

techniques independent of machine. Notably, the novel communication technique naturally extends to any iterative method in which there is a sparse matrix-block vector product kernel, such as block Krylov methods.

The results are generated and methods implemented in the open source library RAPtor [70].

Chapter 5: Efficient Large-Scale Unstructured-Mesh Boundary Exchanges in Exascale Enabled Scramjet Design

5.1 Introduction

Predictive simulation of supersonic mixing and combustion within scramjets requires a complex combination of field experiments and numerical models that form the basis of high-performance computing simulations. There are multiple ways to perform scramjet combustion simulations, with the three most common approaches being direct numerical simulation (DNS), large-eddy simulation (LES), and Reynolds-averaged Navier-Stokes equations (RANS) [74]. While each of these approaches is accompanied by its own drawbacks, LES simulations have been shown to be more accurate than RANS simulations, while RANS simulations have been shown to be more performant than LES simulations in terms of time-to-solution [75, 76].

A multitude of research has been performed on advancing computational fluid dynamics (CFD) codebases on supercomputing architectures. Namely, there is research on characterizing the performance of high performance parallel implicit CFD codes for unstructured grids [77] and discussion on the impact of unstructured mesh partitioning on parallel performance and reduction of time spent in communication of CFD codes [78]. Recently, research has shifted to advance the performance of CFD codebases on heterogeneous architectures. In [79], the nuances of porting CFD software to GPUs is discussed, and [80] presents a hierarchical approach to CFD simulations on heterogeneous supercomputers that auto-balances partitionings distributed across compute units to reduce communicated data volumes.

In this chapter, we focus on *MIRGE-Com*[3] (detailed in Section 5.2), a simulation framework developed within the Center for Exascale-enabled Scramjet Design (CEESD), which is a discontinous Galerkin (DG) code used to perform simulations of scramjet combustion. Importantly, *MIRGE-Com* takes a novel approach to large-scale simulation design by employing the use of code generation and a "lazy" evaluation compute model to address the difficulties of designing such large-scale CFD simulations for heterogeneous architectures. Details of the *MIRGE-Com* framework are given in Section 5.2, with details on lazy evaluation provided in Section 5.2.4. Our work focuses on optimizing the resulting parallel communication for the unstructured mesh boundary exchanges, detailed in Section 5.3. We propose a communication strategy with the foundation of Split node-aware communication for heterogeneous architectures (Chapter 3) adapted to the structure of the generated code within *MIRGE-Com* that aims to collapse the time spent in communication in order to drive down the overall required time-to-solution. This strategy is presented in Section 5.4. Discussion of on-going development and implementation plans for *MIRGE-Com* are provided in Section 5.4.2 and Section 5.5.

[3] https://github.com/illinois-ceesd/mirgecom

Component	Description
ρ	fluid density
v	velocity
ρv	momentum density
ρE	total energy
Y_α	species mass fractions
p	thermodynamic pressure of the fluid
τ	viscous stress tensor
q	total heat flux vector
$\mathbf{J}_\alpha$	species diffusive flux vector
$E^{\texttt{chem}}$	chemical reaction source term in energy conservation equation
$W_\alpha^{\texttt{chem}}$	chemical reaction source term in species conservation equation

Table 5.1: *MIRGE-Com* mathematical model vector components.

5.2 *MIRGE-Com* Framework

MIRGE-Com is designed to solve the compressible Navier-Stokes equations for viscous flows, and the Euler equations for inviscid flows of reactive fluid mixtures. Given a reactive fluid mixture with N_s mixture species on an n-dimensional unstructured mesh Ω, $(n = 1, 2, 3)$, the equations are discretized using a DG method similar to the BR1 algorithm introduced by Bassi et al. in [81] with the thermal terms and chemical reaction sources defined as in [82, 83], respectively. Additionally, Einstein summation convention is used throughout the following section when applicable.

5.2.1 Mathematical Models

The governing equations can be written in the following compact form,

$$\partial_t \mathbf{Q} + \partial \mathbf{F}^I(\mathbf{Q}) = \partial \mathbf{F}^V(\mathbf{Q}, \nabla \mathbf{Q}) + \mathbf{S}, \tag{5.1}$$

where $\mathbf{Q}$ is the vector of conserved variables, $\mathbf{F}^I$ is the vector of inviscid fluxes, $\mathbf{F}^V$ is the vector of viscous fluxes, and the vector of sources for each scalar equation is $\mathbf{S}$. The individual components of the vectors are given by

$$\mathbf{Q} = \begin{bmatrix} \rho \\ \rho E \\ \rho v \\ (\rho Y)_\alpha \end{bmatrix}, \quad \mathbf{F}^I = \begin{bmatrix} \rho v \\ (\rho E + p)v \\ ((\rho v)v + p\delta) \\ (\rho Y)_\alpha v \end{bmatrix}, \quad \mathbf{F}^V = \begin{bmatrix} 0 \\ \tau v - q \\ \tau \\ -\mathbf{J}_\alpha \end{bmatrix}, \quad \mathbf{S} = \begin{bmatrix} 0 \\ E^{\texttt{chem}} \\ 0 \\ W_\alpha^{\texttt{chem}} \end{bmatrix}.$$

with individual vector components defined in Table 5.1, and mixtures have N_s components with $1 \leq \alpha \leq N_s$. The various fluxes are further disected as follows. Species diffusive fluxes are

$$\mathbf{J}_\alpha = \rho d_\alpha \nabla Y_\alpha, \tag{5.2}$$

where d_α gives the species diffusivities, and Y_α are the species mass fractions. The heat flux is given by

$$\mathbf{q} = \underbrace{-\kappa \nabla T}_{\text{conductive heat flux}} + \underbrace{h_\alpha \mathbf{J}_\alpha}_{\text{diffusive heat flux}} \tag{5.3}$$

where κ is the thermal conductivity, T is the gas temperature, and h_α is the species specific enthalpy. These individual quantities within the fluxes become relavant in Section 5.3, as they are pertinent to parallel communication within the codebase.

5.2.2 Finite Element Method Discretization

Returning to the compact form of the governing equations, (5.1), and applying DG, the gradient of the conserved variables is written as auxiliary unknowns, $\mathbf{\Sigma} = \nabla \mathbf{Q}$, and the compressible Navier-Stokes equations become the following coupled system

$$\mathbf{\Sigma} - \nabla \mathbf{Q} = 0 \tag{5.4}$$

$$\partial_t \mathbf{Q} + \partial \mathbf{F}^I(\mathbf{Q}) - \partial \mathbf{F}^V(\mathbf{Q}, \mathbf{\Sigma}) = \mathbf{S}. \tag{5.5}$$

The DG method then requires constructing approximations $\mathbf{Q}_h$ and $\mathbf{\Sigma}_h$ to $\mathbf{Q}$ and $\mathbf{\Sigma}$ in discontinuous finite element spaces, where Ω_h is a collection of disjoint simplices E. For any integer k,

$$\mathbf{V}_h^k = \{\mathbf{v} \in L^2(\Omega_h)^N : \mathbf{v}|_E \in [P^k(E)]^N, \quad \forall E \in \Omega_h\}, \tag{5.6}$$

$$\mathbf{W}_h^k = \{\mathbf{w} \in L^2(\Omega_h)^{N \times d} : \mathbf{w}|_E \in [P^k(E)]^{N \times d}, \quad \forall E \in \Omega_h\}, \tag{5.7}$$

define the discontinuous spaces of piecewise polynomial functions used, where $N = d + 2 + N_s$, d is the spatial dimension, N_s is the total number of mixture species, and $P^k(E)$ is a polynomial space on E containing functions of degree $\leq k$. The final form of the DG formulation is achieved by multiplying by test functions in the defined discontinuous spaces, $\mathbf{v}_h \in \mathbf{V}_h^k$ and $\mathbf{w}_h \in \mathbf{W}_h^k$ and integrating over each element. The resulting weak form of the problem reads, find $(\mathbf{Q}_h, \mathbf{\Sigma}_h) \in \mathbf{V}_h^k \times \mathbf{W}_h^k$ such that $\forall \quad (\mathbf{v}_h, \mathbf{w}_h) \in \mathbf{V}_h^k \times \mathbf{W}_h^k$,

$$\sum_{E \in \Omega_h} \left[\int_E \mathbf{v}_h \cdot \partial_t \mathbf{Q}_h d\Omega + \oint_{\partial E} \mathbf{v}_h \mathbf{h} d\sigma - \int_E \nabla \mathbf{v}_h \cdot \mathbf{F}(\mathbf{Q}_h, \mathbf{\Sigma}_h) d\Omega \right] = \sum_{E \in \Omega_h} \int_E \mathbf{v}_h \cdot \mathbf{S}_h d\Omega, \tag{5.8}$$

$$\sum_{E \in \Omega_h} \left[\int_E \mathbf{w}_h \cdot \mathbf{\Sigma}_h d\Omega - \oint_{\partial E} \mathbf{w}_h \cdot \mathbf{H}_s d\sigma + \int_E \nabla \mathbf{w}_h \cdot \mathbf{Q}_h d\Omega \right] = 0, \tag{5.9}$$

where $\mathbf{h} = \mathbf{h}_e - \mathbf{h}_v$ is the numerical flux with the inviscid and viscous contributions to the physical flux, respectively, and $\mathbf{H}_s$ is the gradient numerical flux for the auxillary equation. For element

boundaries, numerical fluxes are used and account for the discontinuities at element faces. A single valued flux is calculated, for which both elements agree and adheres to all consistency relations imposed by the conservation laws.

5.2.3 Parallel Partitioning of Elements

MIRGE-Com constructs an unstructured mesh a-priori or generates it in-line with built-in, serial mesh generators, then at runtime, METIS [84] is used to partition the mesh. The elements of the discretized unstructured mesh (see Figure 5.1 for an example of the nozzle mesh) are distributed across a fixed number of GPUs. While the group of elements that each partition contains are connected, the elements corresponding to the parallel partition boundary require exchanging information with other GPUs for calculating the element boundary fluxes. As an artifact of the problem size and parallel partitioning, the number of GPUs with which data needs to be exchanged is often ≥ 4 and are not guaranteed to be neighboring, i.e. GPUs may be located on separate compute nodes.

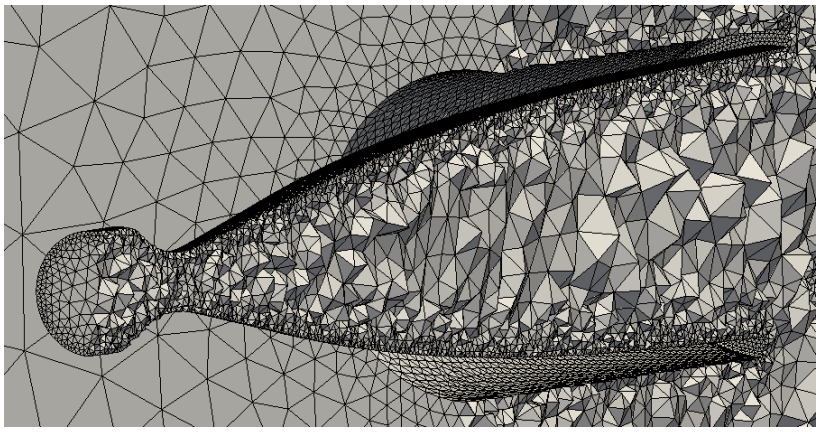


Figure 5.1: Nozzle mesh[4].

The target scramjet simulation for CEESD requires a discretized mesh of approximately 160 million elements partitioned across 200 nodes, or 800 GPUs, of Lassen. For a standard compute model, exact message sizes and counts are determined based on the problem discretization – discussed in detail in Section 5.3.

5.2.4 Lazy Evaluation

MIRGE-Com utilizes a "lazy" compute model through use of the open source, in-development Python package Pytato[5]. Pytato employs a two stage compilation pipline based on a Loopy [85]/Pytato internal representation intended for expressing domain-specific

[4]Credit: M. Smith, CEESD.
[5]https://github.com/kaushikcfd/pytato

transformations [86]. Notably, Pytato includes kernel fusion code transformations that demonstrate approximately 50% of roofline performance on a GPU-based DG-FEM benchmark suite, resulting in approximately 3x speedup over state-of-the-art implementations, such as JAX [6] [87]. While Pytato can be used to generate lazy evaluation array transformations on CPU or GPU-based code, here we consider the GPU-based code generation that *MIRGE-Com* utilizes.

Overall, Pytato produces a directed acyclic graph (DAG) describing the flow of computation being performed in the program, where the edges of the graph correspond to data arrays and the nodes correspond to the fused computational kernels. In the distributed setting of *MIRGE-Com*, this DAG is partitioned across some number of GPUs. A simple example of what a distributed lazy evaluation DAG might look like on four GPUs is given in Figure 5.2.

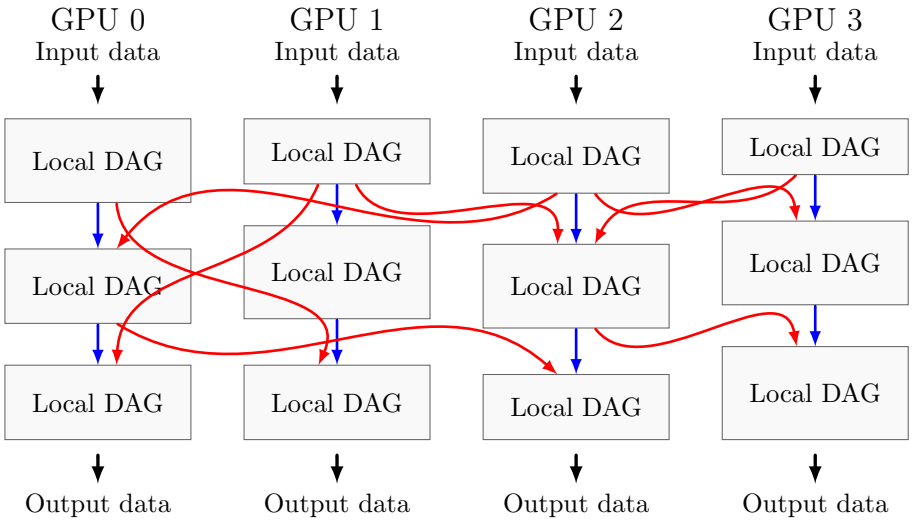


Figure 5.2: Distributed DAG.

Here, the boxes represent partitions of each GPU's local DAG, or fused computational kernels that output data required by another partition of the DAG. The edges/arrows correspond to the required data for local DAG partitions. Blue arrows represent on GPU data, while red arrows represent the movement of data from a separate GPU, or data that requires communication. It is important to note that local DAG partitions are not necessarily the same size, therefore not all GPUs will be communicating their data simultaneously. Local DAG partitions begin to execute once the necessary data for that partition has been received, allowing the overlap of communication and computation. Section 5.4 discusses optimization strategies for communication within this distributed DAG paradigm of the *MIRGE-Com* framework.

5.2.5 Codebase Dependencies

Here, we summarize the relevant information about *MIRGE-Com* pertaining to code implementation. *MIRGE-Com* utilizes the OpenCL programming model through the usage of

[6] https://github.com/google/jax

various packages that rely on PyOpenCL as a computational framework [88]. To run the generated PyOpenCL code on Lassen's NVIDIA GPUs, *MIRGE-Com* relies on Portable OpenCL (PoCL), an OpenCL implementation which has a kernel compiler that works on the LLVM internal representation [89]. Importantly, *MIRGE-Com* uses NVIDIA's Unified Memory (UM) on Lassen through PoCL's Shared Virtual Memory (SVM). This avoids costly migrations of PoCL memory buffers from device to host, and vice-versa, but can affect communication performance, discussed more thoroughly in Section 5.3.1.

5.3 Communication in *MIRGE-Com*

Communication in *MIRGE-Com* happens in multiple stages due to computation dependencies within the right hand side evaluation, as well as, the structure of the distributed DAG. It is the calculation of the fluxes, $\mathbf{h}_e$, $\mathbf{h}_v$, and $\mathbf{H}_s$ (recall Section 5.2.2), that generates the inter-process communication within *MIRGE-Com*, specifically, the communication of the conserved variables of the fluid solution ($\mathbf{Q}$), their gradients ($\nabla\mathbf{Q}$), and the gradient of the fluid temperature (∇T) that lie on parallel partition boundaries. We calculate the expected number of messages communicated and message sizes for each right hand side evluation, as they are dependent upon the dimension of the mesh, parallel partitioning of the mesh, spatial discretization of the problem being solved, as well as, the number of mixture species within the chemical reactions. The associated parameters are summarized in Table 5.2.

Parameter	Description
N_d	dimension of discretized mesh
N_s	number of mixture species
p_o	order of the spatial discretization
p_{el}	points per mesh element
p_{bel}	number of points per boundary element
e_{pb}	number of elements on a parallel partition boundary
n_{pb}	number of parallel partition boundaries

Table 5.2: *MIRGE-Com* communication parameters.

Communication of the conserved variables at the partition boundaries requires sending $N_{cv} = N_d + 2 + N_s$ scalar fields for each point on the boundary. The gradients of the conserved variables then require sending $N_d \cdot N_{cv}$ scalar fields for each point on the boundary, and the temperature gradient requires communication of only N_d scalar fields for each boundary point. This results in a total of

$$n_{\text{sf}} = (N_d + 1)(N_d + 2 + N_s) + N_d \tag{5.10}$$

scalar fields exchanged for each right hand side evaluation.

With each right hand side evluation, every GPU sends messages containing n_{sf} double precision scalar fields to every other GPU with whom it shares a partition boundary. The number of

partition boundaries on a given GPU is not uniform across the simulation. Partitioning of unstructured meshes with METIS produces partitions which have variable numbers of, and sizes of partition boundaries. For this discussion, the number of partition boundaries is given by the local number, n_{pb}, and it is understood that this number is not necessarily the same for every GPU. Moreover, each of the communicated variables is sent in different stages of communication due to data dependencies, therefore the total number of messages sent per right hand side evaluation is

$$m_{\text{RHS}} = 3n_{\text{pb}}, \tag{5.11}$$

assuming that all scalar fields per partition boundary are being sent in the same message. Furthermore, the number of right hand side evaluations performed per timestep is dependent upon the time integration method used. For example, 4^{th}-order Runge-Kutta (RK4), the current default integration scheme, requires four right hand side evaluations per timestep. Hence, the total number of messages per simulation step for a given GPU is $4m_{\text{RHS}}$.

Additionally, the scalar fields are communicated at every point on the partition boundary on every GPU. To determine the size of each message, we need to know the number of points on the partition boundary. For modeling the message sizes, it is useful to know how to calculate the number of points per mesh element, and how it relates to the number of points on each partition boundary. For a tetrahedral mesh, the number of points per element, p_{el}, is related to the spatial discretization order, p_o, and the mesh dimension by

$$p_{el} = \frac{(N_d + p_o)!}{(N_d! p_o!)}. \tag{5.12}$$

In *MIRGE-Com*, partition boundaries include only shared element *faces*, and not lower dimensional constructs. Hence, the partition boundaries have all elements of dimension $N_d - 1$, with a corresponding number of points per boundary element,

$$p_{\text{bel}} = \frac{N_d(N_d + 1)\dots(N_d + (p_o - 1))}{p_o!}. \tag{5.13}$$

Note that the scalar fields sent in the messages are still of full mesh dimension, but the number of points per boundary element are computed for sub-discretizations of dimension $N_d - 1$ and order p_o. Using this information, we calculate the message sizes for each communicated variable, given in Table 5.3, where e_{pb} is the number of elements per parallel partition boundary.

Equally relevant to the message sizes communicated is the number of parallel partition boundaries, or the number of other GPUs with which any GPU needs to communicate, as this affects the communication performance. As noted in Section 5.2.3, this number is often greater than 4, and communicating GPUs are not necessarily located on the same node. In building to the production case, CEESD has developed test cases for *MIRGE-Com* that simulate pieces of the production physics, using smaller scale yet similar unstructured meshes. One such example is the `isolator-injection` test case which simulates a 2D single species combustion initialization with

Communicated Quantity	Message Sizes
$\mathbf{Q}$	$e_{\text{pb}} \cdot p_{\text{bel}} \cdot (N_d + 2 + N_s)$
$\nabla\mathbf{Q}$	$e_{\text{pb}} \cdot p_{\text{bel}} \cdot N_d \cdot (N_d + 2 + N_s)$
∇T	$e_{\text{pb}} \cdot p_{\text{bel}} \cdot N_d$

Table 5.3: *MIRGE-Com* message sizes.

fuel injection. While this test case does not simulate all of the necessary physics, it does give a view at the amount of communication to expect in test runs.

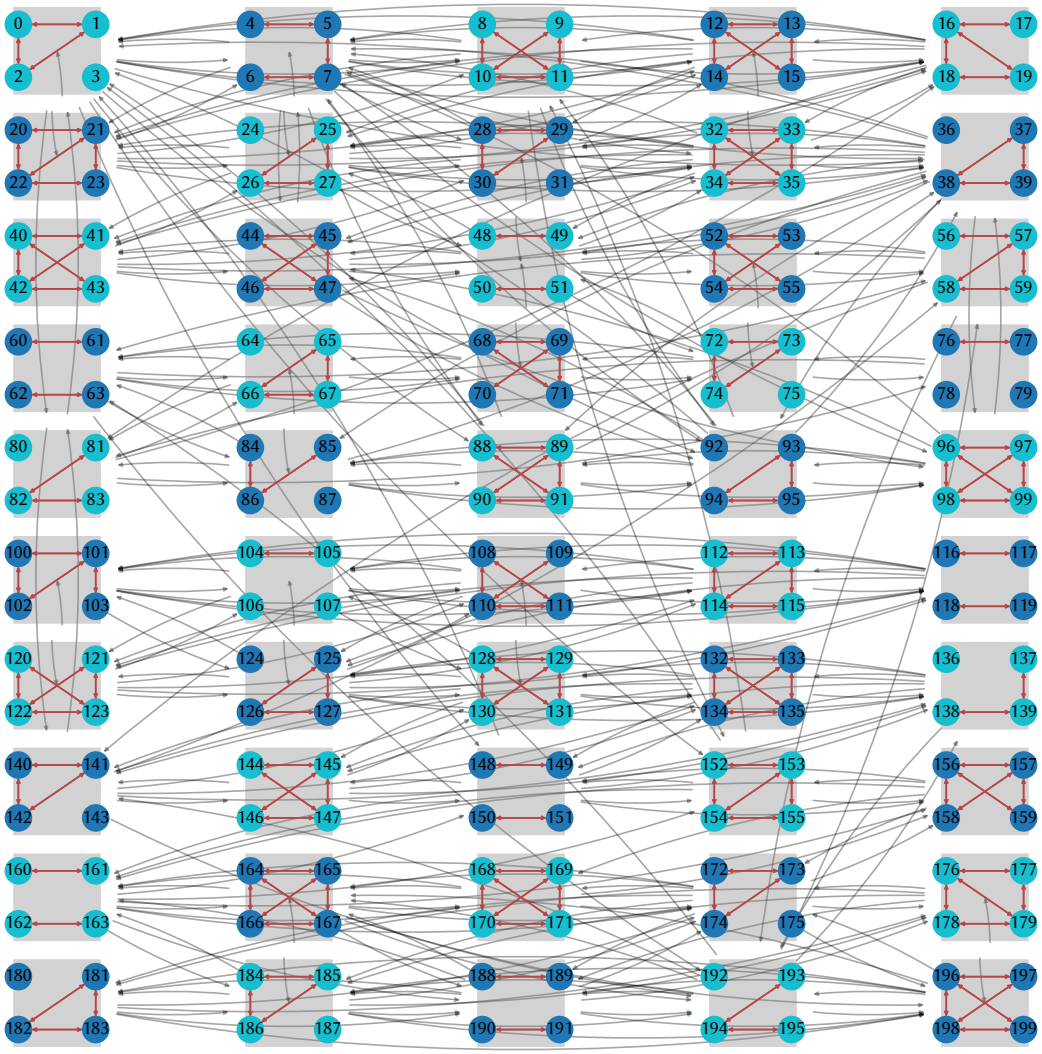


Figure 5.3: Data movement in `isolator-injection` test case.

Figure 5.3 depicts the communication connections for each node in a 50 node/ 200 GPU run of

the `isolator-injection` test case on Lassen. GPUs are represented by the numbered circles. On-node communication connections are given by red arrows, while inter-node connections are simplified to just be the connected nodes given by gray arrows. In most cases, each of the gray lines represents two or three GPUs on each of the nodes communicating. For this test case, GPUs communicated data with $4-8$ other GPUs, often sending data to $1-4$ other nodes with each node typically sending data to 4 other nodes.

Further impacting the potential performance of communication in *MIRGE-Com* is the structure in which communication is performed. Similar to the standard communication presented in Figure 2.3 all messages are sent via a single host process of the GPU while the remaining available cores on a Lassen node are idle, resultng in multiple on-node messages and inter-node messages per single rank. This is depicted in Figure 5.4 where the host processes for each GPU are highlighted blue, while idle cores are shaded gray. Furthermore, on-node messages are red arrows, while inter-node data exchanges are represented by black arrows. Combining the formulas for

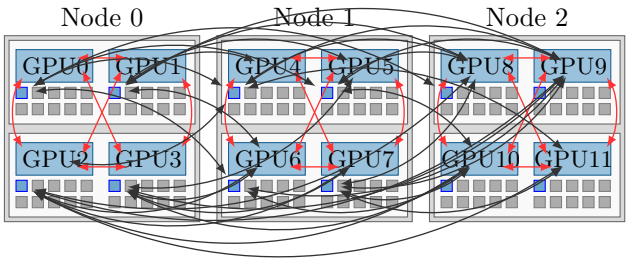


Figure 5.4: Communication in *MIRGE-Com*.

message sizes with predicted numbers of communicating processes and problem partitioning, we model the expected communication performance for *MIRGE-Com* in Section 5.3.1.

5.3.1 Performance Modeling

In this section, we provide models for various *MIRGE-Com* scenarios, taking note of the effects the mathematics (discretization order, dimension of discretized mesh, number of mixture species), as well as, the parallel partitioning (number of elements on a parallel partition boundary, number of parallel partition boundaries) have on expected communication performance.

First, we present modeling paramaters for inter-GPU communication on Lassen utilizing SVM, as this is the memory management model utilized by *MIRGE-Com* — see Section 5.2.5. Modeling parameters are gathered in the same fashion as in Chapter 3. We perform ping-pong and node-pong benchmarks on Lassen, then use a linear least-squares fit to the data to obtain the parameters. In Figure 5.5, the left plot depicts the results of an inter-GPU ping-pong benchmark when using SVM. When using SVM as the memory management model, communication performance is similar to the device-aware communication with Spectrum MPI presented in Figure 3.3 where on-socket and on-node communication is slower than data movement through

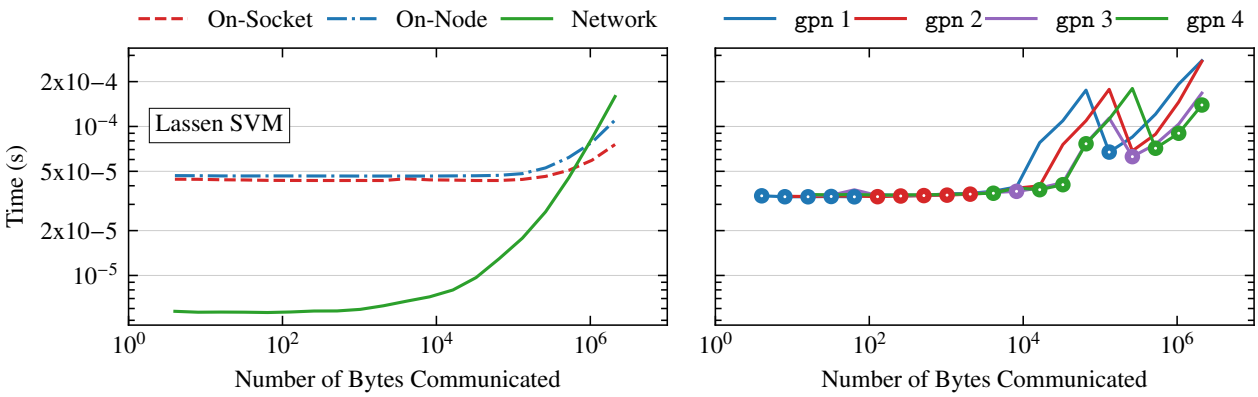


Figure 5.5: Lassen inter-GPU ping-pong (left) and inter-GPU node-pong (right) when utilizing SVM, minimum times circled.

the network. However, on-socket and on-node performance are slower when utilizing SVM. It is worth noting that it is common to experience performance degradation using MPI with SVM, and research is currently being done to improve this performance on emerging architectures [90, 91, 92]. The resulting postal model α and β parameters for the eager and rendezvous protocols are presented in Table 5.4. Note that utilizing SVM results in slower performance than the

		on-socket	on-node	off-node
Eager	α	4.37e-05	4.66e-05	5.71e-06
	β	3.98e-11	3.36e-11	1.98e-10
Rend	α	4.15e-05	4.43e-05	6.80e-06
	β	1.78e-11	3.33e-11	7.29e-11

α [sec] β [sec/byte]

Table 5.4: Measured parameters for inter-GPU communication on Lassen utilizing SVM.

device-aware MPI for on-socket and on-node communication, but similar performance to device-aware communication for off-node communication.

Message sizes are calculated based on the target prediction simulation for CEESD, namely the case where there are 200,000 elements of a 3D mesh ($N_d = 3$) is on each GPU. Using the formulas in Table 5.3, we model all cases pertaining to

$$p_o \in [2, 3, 4] \qquad n_{\text{pb}} \in [4, 10, 20] \qquad N_s \in [1, 2, \ldots, 7].$$

The number of elements on a partition boundary (e_{pb}) is calculated from one of the following scenarios:

1. 20% of the on-GPU elements lie on a parallel partition boundary (Figure 5.6) or
2. 40% of the on-GPU elements lie on a parallel partition boundary (Figure 5.7).

For each of these scenarios, the boundary elements are assumed to be partitioned evenly across the number of parallel partition boundaries (n_{pb}).

Figures 5.6 and 5.7 present modeled times for each stage of communication within a right hand side evaluation for *MIRGE-Com*, namely communication of Q (Figures 5.6a and 5.7a), ∇Q (Figures 5.6b and 5.7b), and ∇T (Figures 5.6c and 5.7b). Subplots correspond to the number of parllel partition boundaries (n_{pb}) that the boundary elements are equally divided between. Modeled times (top rows) and message sizes (bottom rows) are given with respect to the number of species in the chemcial reaction (N_s). We present models for three different spatial orderings with $p_o = 2$ given by a solid red line, $p_o = 3$ by a dashed blue line, and $p_o = 4$ by a green dashed and dotted line. As a reminder, these models represent the standard communication strategy currently in-place within *MIRGE-Com*, utilizing only the four GPUs per node on Lassen with each GPU paired to a *single* MPI host rank.

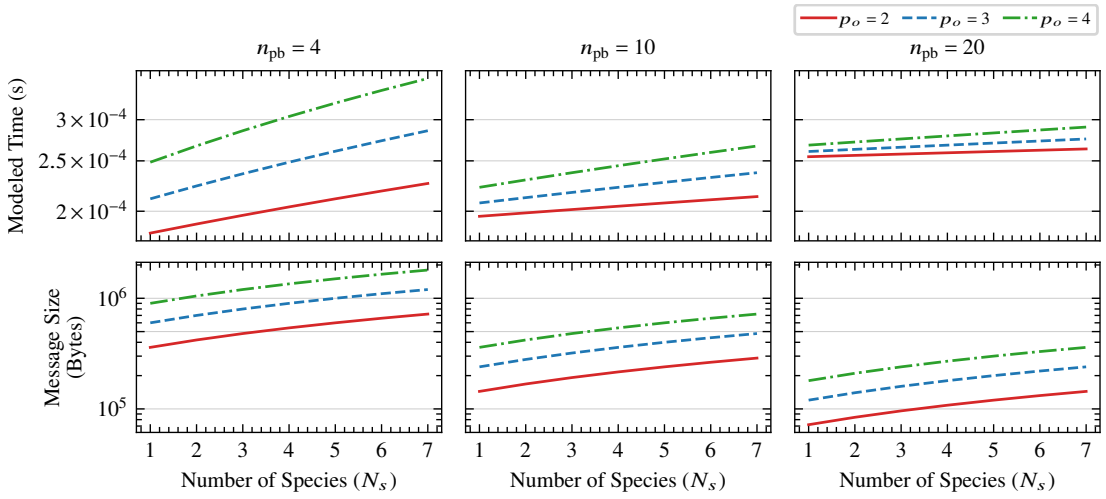
Figure 5.6 contains the models for the case if 20% of the 200,000 on-GPU elements were on parallel partition boundaries. Here, we note that the message sizes and predicted performance of ∇T communication does not change with the number of species, which was first shown in Table 5.3, but is included here for comparison against Figures 5.6a and 5.6b. Across all communicated quantities in Figure 5.6, if the entire boundary data volume is being communicated to only 4 other GPUs ($n_{\text{pb}} = 4$), then there are gaps in the predicted performance when increasing the spatial discretization order (p_o) due to the increased message sizes.

For Q communication models (Figure 5.6a), we see the expected amount of time required to communicate goes down when increasing the number of parallel partition boundaries from four to ten, but then increases again when the number of parallel partition boundaries increases to 20. Predicted performance for ∇Q communication does not exhibit this same trend, most likely due to the increased message sizes. In Figure 5.6b, we see the predicted amount of time required to communicate going down as the number of parallel partition boundaries increases, correlated with the message sizes decreasing. Finally, for communication of ∇T (Figure 5.6c), we see the expected time to communicate go up as we increase the number of parallel partition boundaries, even though the individual message sizes are going down.

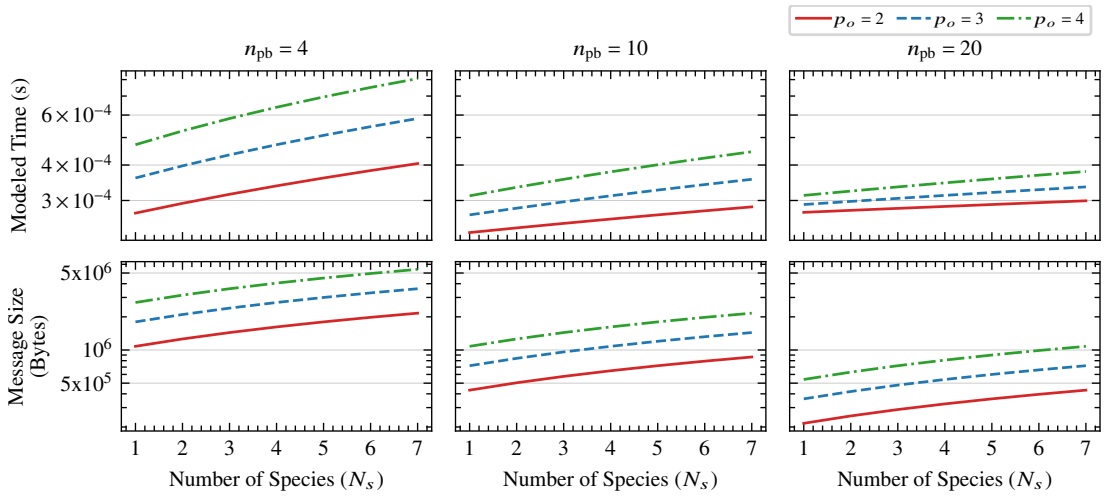
Figure 5.7 presents the models for the case if 40% of the 200,000 on-GPU elements were on parallel partition boundaries. Exhibiting the same trend as in Figure 5.6, the expected amount of time to communicate each of the quantities increases with the spatial discretization order due to the increasing message sizes. The expected time to communicate Q and ∇Q doubled over the case of $n_{\text{pb}} = 4$ when only 20% of the on-node elements were parallel partition boundary elements. Additionally, due to the increase in the total data volumes being communicated, $n_{\text{pb}} = 20$ is expected to perform much faster than $n_{\text{pb}} = 4$ for both Q and ∇Q. This is unsurprising considering the drastic slope increase in the communication benchmarks once the number of bytes communicated exceeds 10^6 Bytes (see Figure 5.5). ∇T communication models predict similar performance for communication to Figure 5.6c, but it is worth noting that for both $n_{\text{pb}} = 4$ and $n_{\text{pb}} = 10$ there is a *slight* increase in the expected amount of time to communicate.

While the individual predicted communication times appear minimal, the upper bound on the cumulative estimated time required to communicate is approximately 0.004 seconds per right hand side evaluation. Seeing as the RK4 time integrator requires the fewest right hand side evaluations of all the potential time integrators to be used in *MIRGE-Com*, this results in a minimum of approximately 0.016 seconds spend in communication for a single time step. This is very high considering current small-scale test runs within CEESD have exhibited approximately 0.2s required to perform all computations within a single time-step for 200,000 elements on a single GPU. Though 12% of time being spent in communication is small compared to the percentages seen in sparse matrix operations (see Chapter 4), the percentage of time spent in communication is only expected to increase as the cost of computation is driven down with developments in lazy evaluation. Currently, CEESD aims to reduce the amount of time required to compute each timestep down to milliseconds, resulting in a predictive simulation runtime of one to two weeks. Hence, these modeled communication costs are high, relative to the target time per timestep.

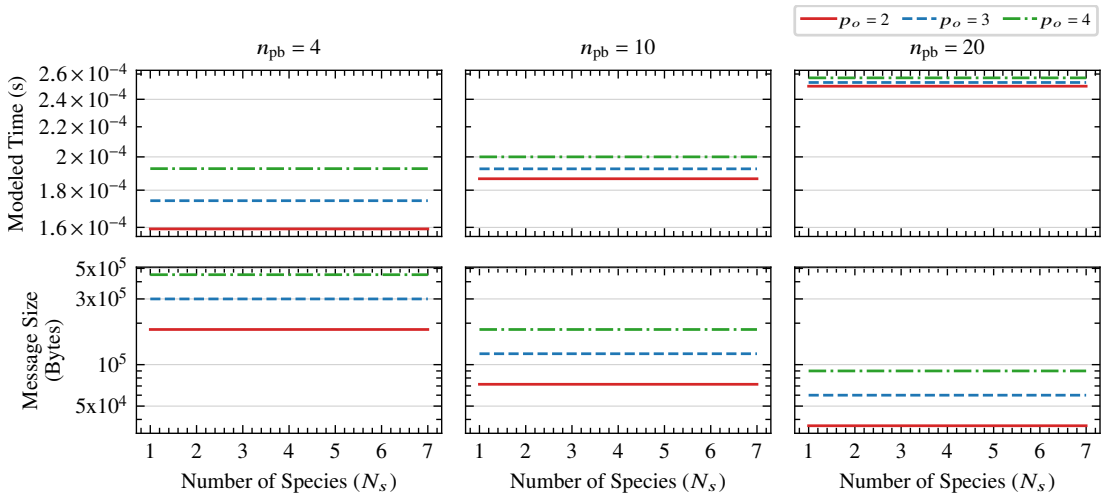
To address these relatively high communication costs, we propose a communication strategy based on Split (Chapter 3) that takes advantage of all on-node CPU cores to perform the communication. This communication strategy is the most relevant, considering that the models show it can be more efficient to communicate data in smaller message sizes, assuming the overall communicated data volume is extremely large. (For Figure 5.7, the modeled time to communicate decreased as n_{pb} increased which was not true for Figure 5.6 where the total communicated data volume was smaller.) Overall, we target a communication strategy that reduces the amount of time required to communicate for all communicated quantities. We observe some communicated quantities go down in performance time as the number of parallel partition boundaries is increased, while others consistently rise (∇T). Taking advantage of all available CPU cores to perform communication can help achieve this, as demonstrated in Section 5.4.



(a) Q

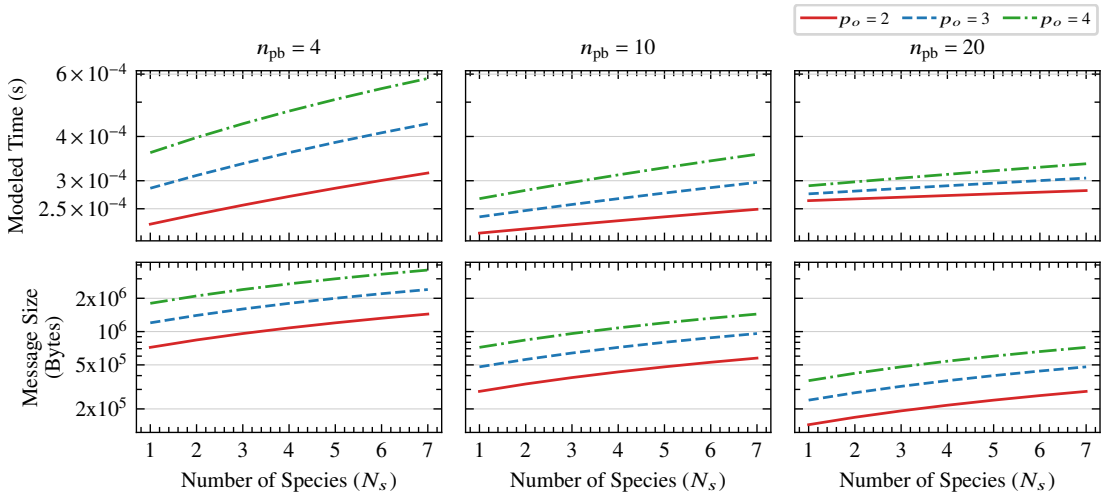


(b) ∇Q

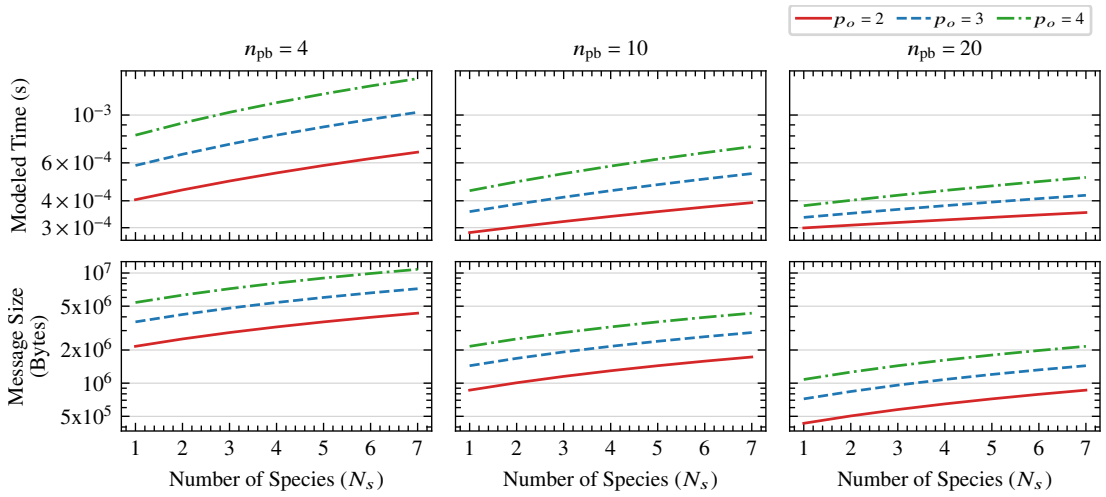


(c) ∇T

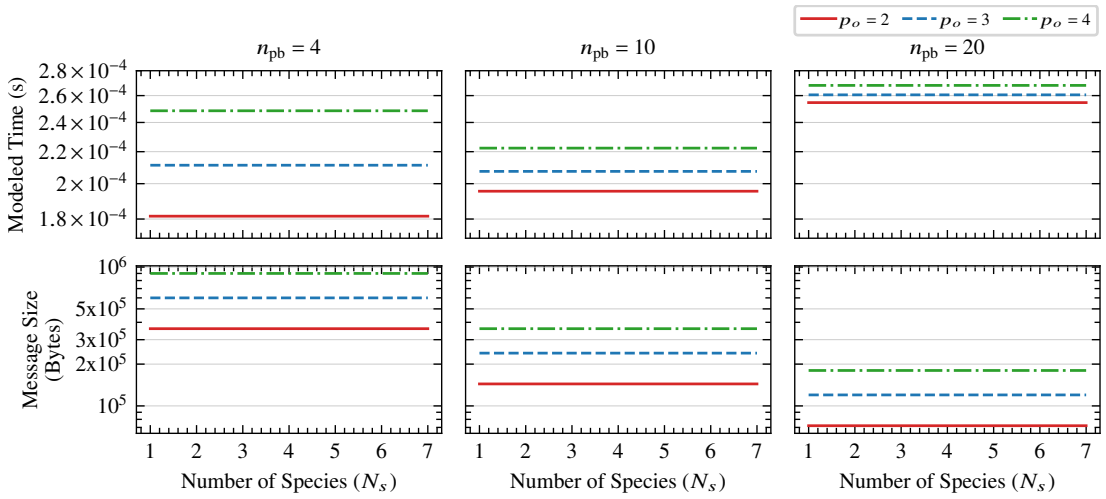
Figure 5.6: Modeled times and message sizes for the communicated quantities in *MIRGE-Com* if 20% of the on-GPU elements were on parallel partition boundaries.



(a) Q



(b) ∇Q



(c) ∇T

Figure 5.7: Modeled times and message sizes for the communicated quantities in *MIRGE-Com* if 40% of the on-GPU elements were on parallel partition boundaries.

5.4 Efficient Communication in Boundary Exchanges

Motivated by Split node-aware communication demonstrating effectiveness at reducing the amount of time required to communicate large data volumes on heterogeneous architectures (Chapter 3), we extend similar communication techniques to the boundary exchanges in *MIRGE-Com*. First, we note the differences between the unstructured mesh boundary exchanges in *MIRGE-Com* and the irregular point-to-point communication in sparse data operations on which the method was initially tested. The unstructured mesh boundary exchanges do *not* suffer from data redundancy — the data being exchanged between GPUs is distinct. However, there is technically a message redundancy, as many GPUs are sending data to multiple other GPUs, some of them on the same destination node. Additionally, the messages being exchanged are fairly large, as seen in Figures 5.6 and 5.7.

The original node-aware communication techniques were designed within the context of sparse-matrix operations, and execute by splitting all inter-nodal messages across the available on-node MPI ranks. Because duplicate data was being removed via message agglomeration and because the sparsity pattern of matrices dictate the number of on-node processes which will require any data received inter-nodally, it is not feasible to pin sending and receiving processes close to the origin or final destination process of the data. However, in the case of *MIRGE-Com*, the data being sent and received by any single GPU is unique. This allows us to distribute inter-nodal sends and receives to processes physically close to the origin and destination GPU, thus reducing the amount of time required for the on-node distribution and on-node gathering of data in Steps 1 and Steps 3 of a multi-step communication scheme. In fact, we can reduce on-node distribution to on-socket distribution of messages. Additionally, because exchanged data within the boundary exchanges is required for continuation of execution, communication costs can be further reduced by utilizing separate MPI ranks for local communication and the inter-node message exchanges. This allows for overlap of node-local communication and inter-node communication. This efficient communication strategy is summarized in Figure 5.8 and detailed in Description 5.1. In Section 5.4.1, we provide models for the potential performance of this communication strategy within *MIRGE-Com*.

5.4.1 Performance Modeling

To model the amount of time required to perform the efficient boundary exchanges described in Figure 5.8, we model the *worst-case* scenarios for each of the individual steps ((5.14), (5.15), (5.20)) and sum them. All of the models assume a constant message size, s, across all communicating processes for simplification, the same as the standard performance modeling presented in Section 5.3.1. Furthermore, each stage of communication dependends upon the number of parallel partition boundaries, n_{pb}, and it is assumed that $\texttt{gpn} - 1$ of the n_{pb} boundary exchanges are being sent to the other on-node GPUs, the same as the standard models presented in Section 5.3.1.

We rely on consistent notation with Chapter 3. Namely, `gpn` is the number of GPUs per node, and `gps` is the number of GPUs per socket. We use the variable `ppg` to mean the number of

Step 1 *Perform socket-local distribution of inter-node send data.*
First, inter-node send data is copied to the host-rank of the GPU. Next, send data is split by prioritizing splitting the data over the physically closest cores to the host-rank of the GPU. If one GPU on a socket is sending more data than another, then the GPU sending the most data distributes its data to be sent across more of the on-socket cores. This is seen in the Socket-Local data distribution for Node 1 in Figure 5.8 where GPU4 and GPU6 are sending more data than GPUs: GPU5 and GPU7, respectively.

Step 2a *Exchange node-local messages.*
Messages are exchanged through device-aware communication via the GPU host-ranks — row1 of **Step 2** in Figure 5.8.

Step 2b *Exchange inter-node messages.*
Messages are exchanged through non-host ranks of the GPUs — row2 of **Step 2** in Figure 5.8.

Node-local message exchanges (**Step 2a**) and inter-node message exchanges (**Step 2b**) happen simultaneously.

Step 3 *Perform socket-local gathering of inter-node received data.*
This operation is the reverse of **Step 1** and therefore mirrors it in the number of socket-local cores that the inter-node received data is split across. However, inter-node receives are assigned to socket-local ranks in reverse order so that all local ranks are active in communication. As a result, socket-local ranks receiving data can be different than the socket-local ranks sending inter-node data.

Description 5.1: Detailed steps for efficient communication in *MIRGE-Com* boundary exchanges.

processes per GPU, but its context has changed slightly. Here, the host rank is the only rank with access to on-device GPU buffers, and ppg refers to the number of cores which the GPU was assigned to split its inter-node communicated data across. For our models, we assume an even split of on-node cores between GPUs, hence $\texttt{ppg} = 10$. The α and β parameters for inter-CPU or inter-GPU communication are differentiated by descriptive subscripts.

For Step 1, we assume that the inter-node data to be communicated is still within GPU device memory and the CUDA system will have to migrate the data to the host before distributing the messages to on-socket processes. This is modeled by (5.14),

$$T_{\text{st1}}(s, n_{\text{pb}}) = \underbrace{\alpha_{\texttt{D2H}} + \beta_{\texttt{D2H}} \cdot (s \cdot (n_{\text{pb}} - (\texttt{gpn} - 1)))}_{\text{Copy to host}} + \underbrace{\alpha_{\texttt{on-socket}} \cdot (n_{\text{pb}} - (\texttt{gpn} - 1)) + \beta_{\texttt{on-socket}} \cdot s}_{\text{On-socket distribute}} \tag{5.14}$$

where we model the time to copy the data to the host process, then distribute the inter-node data to be communicated across the appropriate number of on-socket processes.

For Step 2, the model considers whether the time to send node-local messages (5.16) or the time to send inter-node messages (5.17) will take longer to execute. This is given by

$$T_{\text{st2}}(s, n_{\text{pb}}) = \max\left(T_{\text{st2-on}}(s, n_{\text{pb}}),\ T_{\text{st2-off}}(s, n_{\text{pb}})\right), \tag{5.15}$$

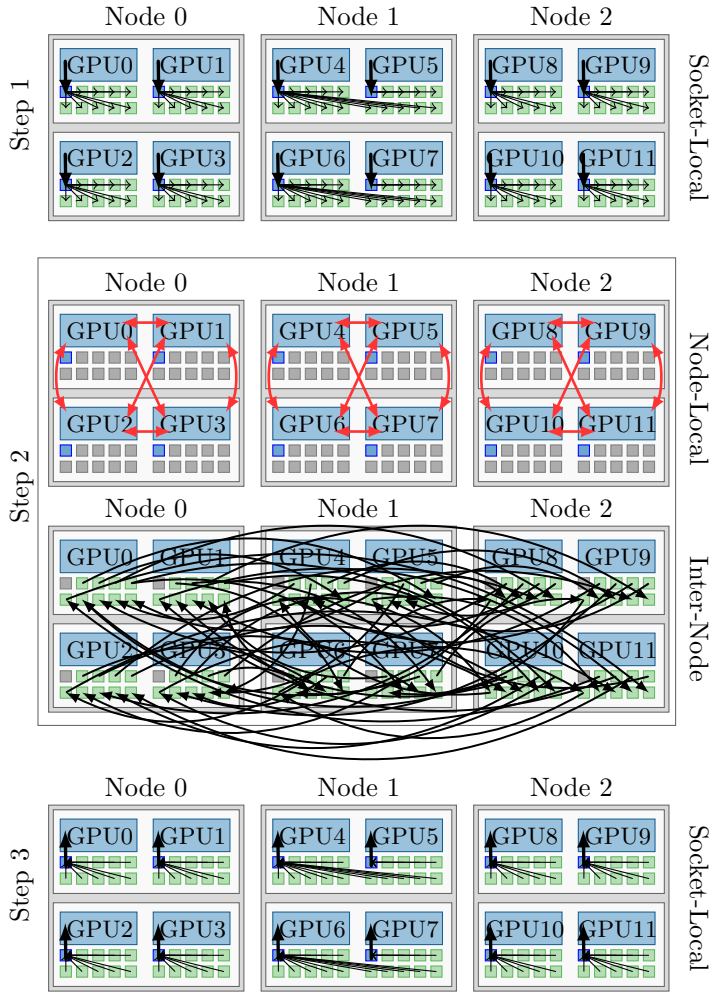


Figure 5.8: Split node-aware in *MIRGE-Com*.

where the node-local communication model is given by

$$\begin{aligned} T_{\text{st2-on}}(s, n_{\text{pb}}) = &\underbrace{\alpha_{\texttt{SVM-on-socket}} \cdot (\texttt{gps} - 1) + \beta_{\texttt{SVM-on-socket}} \cdot s}_{\text{On-socket boundary exchanges}} \\ &+ \underbrace{\alpha_{\texttt{SVM-on-node}} \cdot (\texttt{gpn} - \texttt{gps}) + \beta_{\texttt{SVM-on-node}} \cdot s}_{\text{On-node boundary exchanges}}, \end{aligned} \tag{5.16}$$

which assumes that data is being sent to every other GPU on-node. The inter-node communication performance is modeled by

$$T_{\text{st2-off}}(s, n_{\text{pb}}) = \alpha_{\texttt{off-node}} \cdot m_{\max} + \max\left(\frac{\texttt{ppn}_{\texttt{active}} \cdot s}{R_N}, \frac{s}{R_b}\right) \tag{5.17}$$

where $m_{\max}$ is the maximum number of messages sent by a single process, given by

$$m_{\max} = \left\lceil \frac{n_{\text{pb}} - (\texttt{gpn} - 1)}{\texttt{ppg} - 1} \right\rceil \tag{5.18}$$

and $\texttt{ppn}_{\texttt{active}}$ is the number of active processes per node injecting data into the network, given by

$$\texttt{ppn}_{\texttt{active}} = (\texttt{ppg} - 1) \cdot \texttt{gpn}. \tag{5.19}$$

Both the maximum number of messages sent by a single process ($m_{\max}$) and the active number of processes injecting data into the network ($\texttt{ppn}_{\texttt{active}}$) are based on the assumption that the number of boundary exchanges requiring network communication is $n_{\text{pb}} - (\texttt{gpn} - 1)$.

Step 3 is the reverse of Step 1, and is given by

$$T_{\text{st3}}(s, n_{\text{pb}}) = \underbrace{\alpha_{\texttt{on-socket}} \cdot (n_{\text{pb}} - (\texttt{gpn} - 1)) + \beta_{\texttt{on-socket}} \cdot s}_{\text{On-socket gather}} + \underbrace{\alpha_{\texttt{H2D}} + \beta_{\texttt{H2D}} \cdot (s \cdot (n_{\text{pb}} - (\texttt{gpn} - 1)))}_{\text{Copy from host}}, \tag{5.20}$$

and thus the entire modeled time to perform an efficient boundary exchange for one of the variables in *MIRGE-Com*, is given by

$$T_{\text{eff}}(s, n_{\text{pb}}) = T_{\text{st1}}(s, n_{\text{pb}}) + T_{\text{st2}}(s, n_{\text{pb}}) + T_{\text{st3}}(s, n_{\text{pb}}). \tag{5.21}$$

For these models, the predicted speedup over the standard communication models is presented in Figures 5.9 and 5.10 which models the same cases presented for standard communication models in Section 5.3.1. Notedly, these models do not take into consideration the possibility of message agglomeration or the number of distinct nodes to which a given node is sending data, which could result in further speedup. For these cases, we see that there are potentially more performance benefits for Q and ∇Q communication for the $n_{\text{pb}} = 4$ case for both 20% and 40% of the elements lying on parallel partition boundaries. As n_{pb} grows, the speedups become less the higher the order (p_o). The only performance improvements indicated for ∇T are when $n_{\text{pb}} = 20$, which is likely due to the smaller message sizes being communicated for this boundary exchange (see Table 5.3). Additionally, this suggests the usage of different communication strategies for communicating the different quantities.

While it may be most performant to use Split communication for ∇Q, standard communication will most likely be the most performant strategy for ∇T, and potentially Q, depending on the problem partitioning. This suggests the usage of a quick performance test while setting up communication strategies to test out standard and Split, then choose the most efficient communication strategy for each quantity's boundary exchanges. Additionally, it is worth noting that there will likely be higher speedups in practice, since it is unlikely that every GPU on a single node will be performing the same number of boundary exchanges. Furthermore, more efficient partitioning strategies are currently being implemented within *MIRGE-Com*, to reduce the number of GPUs being communicated with, 4–8.

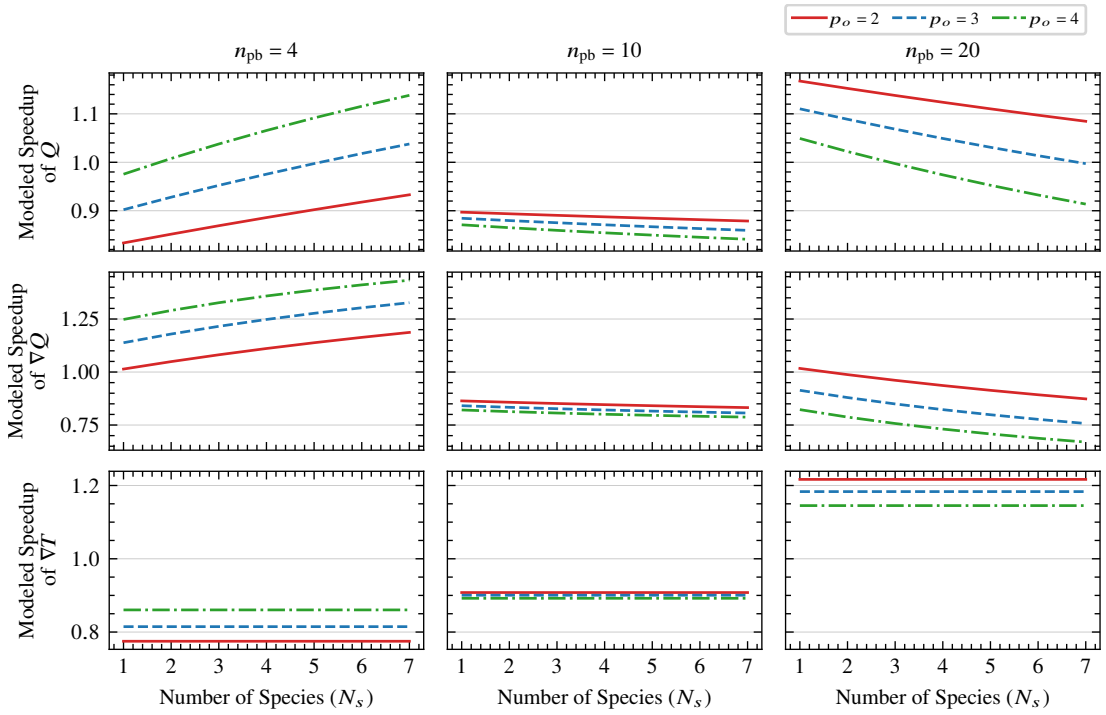


Figure 5.9: Modeled speedup for the communicated quantities in *MIRGE-Com* if using message splitting across on-socket cores for inter-node communication if 20% of the on-GPU elements were on parallel partition boundaries.

The presented models predict the worst-case scenario of having to migrate the data from device memory to host memory. If we do not consider the need to copy the data, as if the data had already been migrated by the CUDA system, then the speedups improve, depicted in Figures 5.11 and 5.12. Now, there is speedup predicted for communication of both Q and ∇Q in every case. Additionally, in Figure 5.12, for $p_0 = 4$, even ∇T shows a slight speedup in the $n_{\text{pb}} = 4$ case.

5.4.2 Implementation

The implementation of the presented boundary exchange communication strategy is designed to be close to a "drop-in replacement" for the existing MPI communication layer within the *MIRGE-Com* framework and interface solely with the Pytato package. Current communication within *MIRGE-Com* simply uses MPI through the mpi4py interface, and a communicator is created as seen in Listing 5.1.

```
1 from mpi4py import MPI
2 comm = MPI.COMM_WORLD
```

Listing 5.1: *MIRGECom* standard communicator creation.

In this case, it is assumed that the global number of processes in `MPI.COMM_WORLD` is equal to the participating number of GPUs, i.e. a single MPI rank per GPU. Additionally, this communicator is

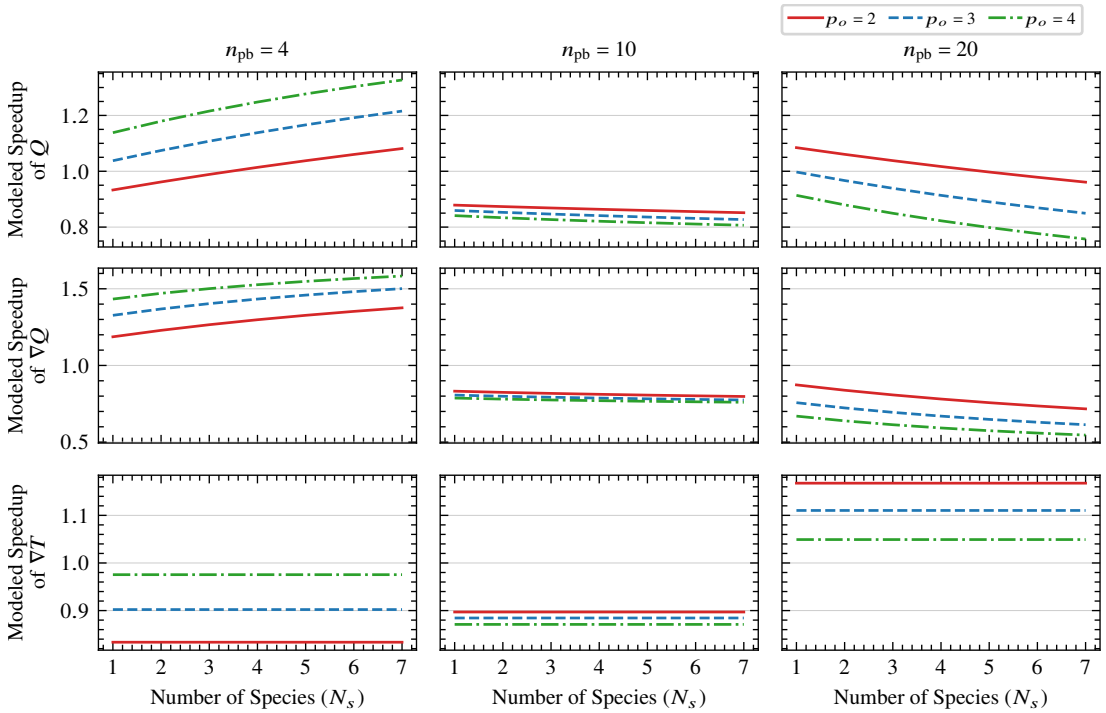


Figure 5.10: Modeled speedup for the communicated quantities in *MIRGE-Com* if using message splitting across on-socket cores for inter-node communication if 40% of the on-GPU elements were on parallel partition boundaries.

passed to every function within a given test case, so replacing it with a new communicator is straightforward, assuming the new communicator is consistent with the existing mpi4py API.

For the proposed efficient communication strategy, it is necessary to ensure that the new communicator functions in a similar fashion to that created in Listing 5.1, and does not require the user to differentiate between the host ranks of each GPU and the non-host ranks. Listing 5.2 shows the creation of a communicator through the in-development package pynabe.

```
from mpi4py import MPI
from pynabe import Comm
mpi_comm = MPI.COMM_WORLD
comm = Comm(mpi_comm)
```

Listing 5.2: *MIRGECom* efficient communicator creation.

Here, the generated communicator can be passed to any function within *MIRGE-Com* and still utilize the mpi4py API. The entire number of MPI ranks within `MPI.COMM_WORLD` is the total number of GPU host ranks and the non-host ranks, hence when a standard MPI API function call is made through the proposed communicator, it is called through a virtual communicator of the host ranks which maintains the same rank ordering as the standard communicator in Listing 5.2. Therefore, no updates are required to any functions in *MIRGE-Com* that utilize MPI communication for setup.

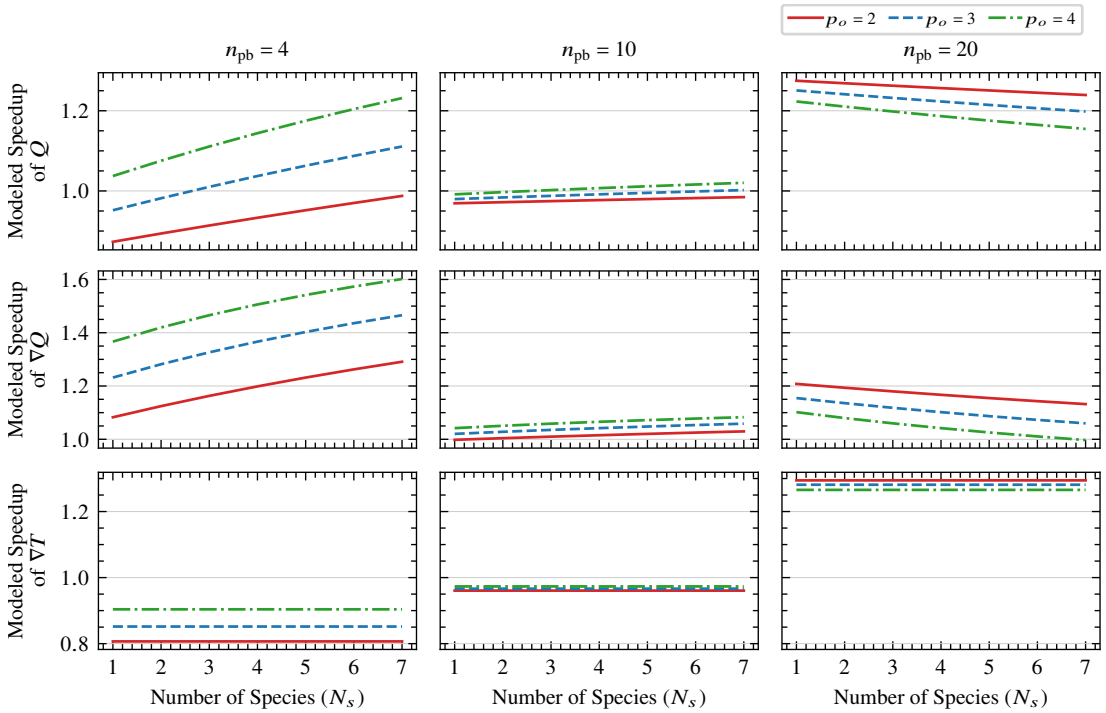


Figure 5.11: Modeled speedup for the communicated quantities in *MIRGE-Com* if using message splitting across on-socket cores for inter-node communication if 20% of the on-GPU elements were on parallel partition boundaries. Data does not require migrating between memories via SVM.

Utilizing the new efficient boundary exchange requires interfacing the pynabe generated communicator with Pytato. This is done through the `setup_communicators(comm, compiled_rhs)` function in Listing 5.3. The `compiled_rhs` variable contains the computation DAG for lazy evaluation as discussed in Section 5.2.4. Setting up the efficient boundary exchange requires knowledge of the input and output data for each computational kernel, hence the `setup_communicators` function compiles lists of the sends and receives for each of the computational kernels in the rhs DAG (`compiled_rhs`), which is then given to the communicator to generate the appropriate subcommunicators as described in Algorithm 3.1, then communication is performed similar to Algorithm 3.2.

```
...
compiled_rhs = actx.compile(rhs)  # Generate DAG of rhs computation
setup_communicators(comm, compiled_rhs)  # Create efficient boundary exchange
    communicators
...
```

Listing 5.3: Setup efficient boundary exchange in *MIRGECom*.

After the communicators for the efficient boundary exchange are setup, they are called through functions, `send_data(..)` and `recv_data(..)` which are passed to Pytato as user-defined functions, instructing Pytato on how to perform the communication between DAG partitions.

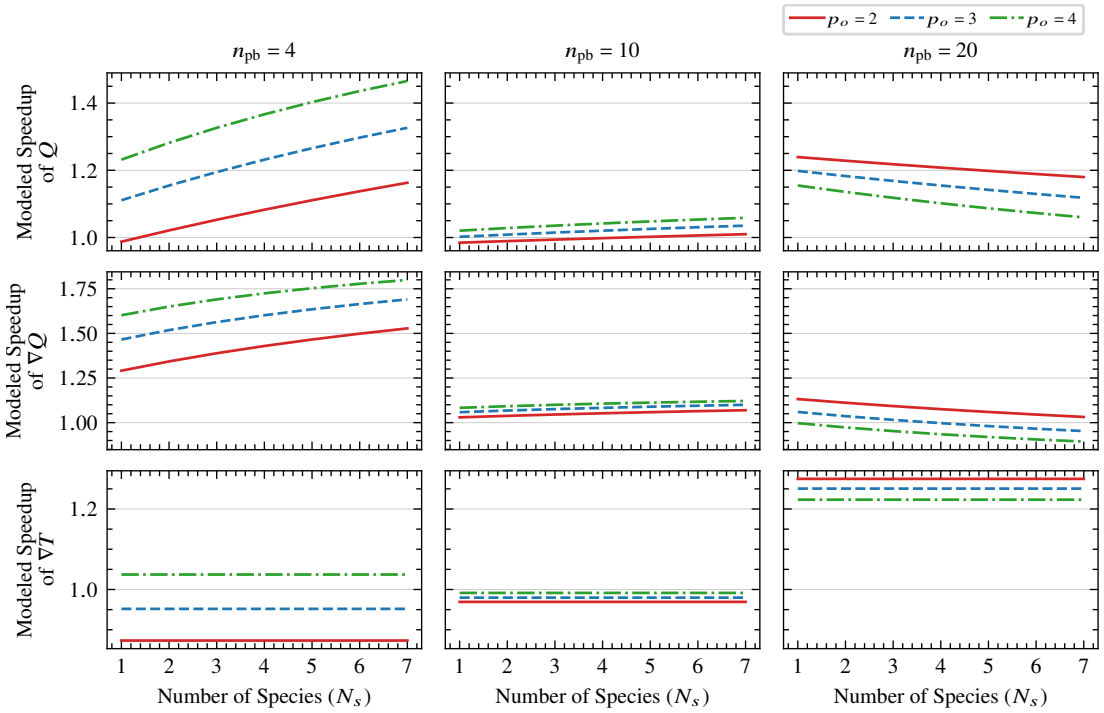


Figure 5.12: Modeled speedup for the communicated quantities in *MIRGE-Com* if using message splitting across on-socket cores for inter-node communication if 40% of the on-GPU elements were on parallel partition boundaries. Data does not require migrating between memories via SVM.

5.5 Conclusions

Predictive simulation of supersonic mixing and compustion within scramjets is a complex problem made tractable through the usage of large-scale LES. The simulation framework *MIRGE-Com* tackles designing large-scale simulations for heterogeneous architectures through the usage of code generation and "lazy" evaluation.

In this chapter, we performed a performance model-based analysis of the expected communication for large-scale simulation runs of the *MIRGE-Com* framework. We related the communication performance to the mathematical equations being solved and the employed discretizaion strategy, noting an increase in expected communication time for increased order of the spatial discretization and number of mixture species in the chemical reaction. Importantly, we noted varying performance predictions for different flux quantities communicated based on the message sizes and total communicated data volume for each. We introduced a strategy to collapse the required communication time of these boundary exchanges based on Split node-aware communication that utilizes all of the available CPU cores instead of a single host rank per GPU to perform all of the boundary exchanges.

Overall, this chapter provides a comprehensive analysis of the expected communication performance of full-scale *MIRGE-Com* simulation runs, and introduces a novel communication strategy for large-scale unstructured-mesh boundary exchanges based on multi-step communication

techniques. Notably, the novel communication technique naturally extends to any large-scale codebase requiring boundary exchanges or irregular point-to-point communication between GPUs.

Chapter 6: Scalable Anderson Acceleration Solvers

Portions of this chapter appear in the paper "Performance of Low Synchronization Orthogonalization Methods in Anderson Accelerated Fixed Point Solvers" published in Proceedings of the 2022 SIAM Conference on Parallel Processing for Scientific Computing [93].

6.1 Introduction

Given a nonlinear function $G : \mathbb{R}^n \to \mathbb{R}^n$ and a vector $x \in \mathbb{R}^n$, x is a fixed point of the function G if x is unchanged after application of G, as in (6.1).

$$x = G(x) \tag{6.1}$$

A common method for solving fixed point problems is the fixed point (FP) method given in Algorithm 6.1 where the function G is applied repeatedly to an initial starting guess for x, x_0, until convergence. In practice, convergence of this method can be quite slow.

Algorithm 6.1: Fixed Point (FP)

Input: x_0, tol, and maxIters
Output: x^*
1 **for** $i = 0, 1, \ldots,$ maxIters
2 $x_{i+1} = G(x_i)$ **if** $\|x_{i+1} - x_i\| < tol$
3 **Return** x_{i+1} as x^*

Anderson acceleration (AA) is a method employed to accelerate the convergence of the fixed point method for solving systems of nonlinear equations [94, 95, 96]. The general form of this method is given in Algorithm 6.2 where m prior function evaluations are used to accelerate the convergence of the FP iteration. An example of the convergence acceleration is given in Figure 6.1. Generalizing the algorithm further, a damping parameter $\beta > 0$ can be introduced resulting in the following modification to Line 6 in Algorithm 6.2,

$$x_{i+1} = (1 - \beta_i) \sum_{k=0}^{m_i} \alpha_k^{(i)} x_{i-m_i+k} + \beta_i \sum_{k=0}^{m_i} \alpha_k^{(i)} G(x_{i-m_i+k}). \tag{6.2}$$

For now, we only consider the case of $\beta = 1$ which is equivalent to Line 6 in Algorithm 6.2, but include the possibility of damping for further discussion later. Furthermore, the formulation of this method leading to the most efficient implementation recasts the constrained linear least-squares problem on Line 5 as an unconstrained minimization problem. Letting $\mathcal{F}_i = [\Delta f_{i-m_i}, \ldots, \Delta f_{i-1}]$ with $\Delta f_k = f_{k+1} - f_k$, then the least-squares problem becomes

$$\min_{\gamma^{(i)}} \|f_i - \mathcal{F}_i \gamma^{(i)}\|_2, \tag{6.3}$$

Algorithm 6.2: General Anderson Acceleration

Input: x_0, $m \geq 1$, tol, and maxIters
Output: x^*

$x_1 = G(x_0)$, $f_0 = G(x_0) - x_0$
for $i = 1, 2, \ldots,$ maxIters
  Set $m_i = \min\{m, i\}$
  Set $F_i = [f_{i-m_i}, \ldots, f_i]$ with $f_i = G(x_i) - x_i$
  Determine $\alpha^{(i)} = [\alpha_0^{(i)}, \ldots, \alpha_{m_i-1}^{(i)}]^T$ that solves

$$\min_{\alpha^{(i)}} \|F_i \alpha^{(i)}\|_2 \quad s.t. \quad \sum_{k=0}^{m_i} \alpha_k^{(i)} = 1$$

  Set $x_{i+1} = \sum_{k=0}^{m_i} \alpha_k^{(i)} G(x_{i-m_i+k})$
  if $\|x_{i+1} - x_i\| < tol$
    Return x_{i+1} as x^*

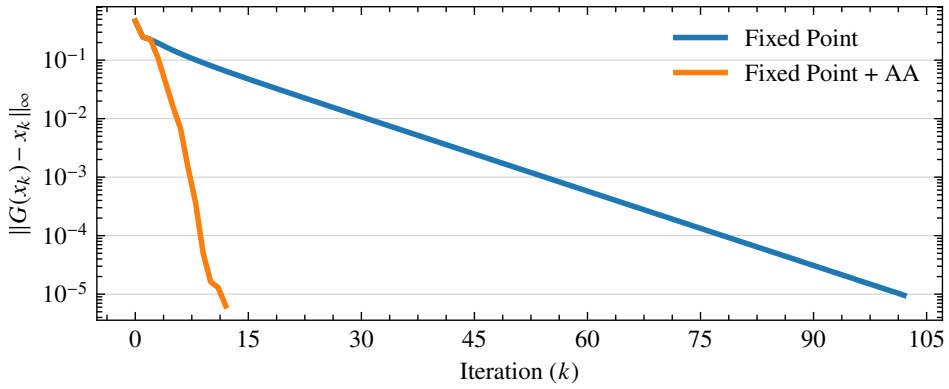


Figure 6.1: Residual history for the 2D Bratu problem (Section 6.4.3.2) for fixed point iteration and Anderson acceleration.

with α easily recoverable from γ via the given relation, $\alpha_k = \gamma_k$, $\alpha_k = \gamma_k - \gamma_{k-1}$,... for $0 \leq k < m_{i-1}$. This results in the most common form of Anderson acceleration, summarized in Algorithm 6.3 and discussed further in Section 6.2.1.

Solving the unconstrined minimization problem is generally done through solving a least squares problem (LSP) via QR factorization [95, 97]. Because the LSP is highly sensitive, it is imperative to employ a stable algorithm, such as modified Gram-Schmidt, to update the factorization at each iteration. The application of modified Gram-Schmidt constitutes the main bottleneck of the algorithm when performed in a distributed memory parallel environment, due to the associated high communication costs of performing multiple dot products in sequence [50].

A solution to the costly QR updates within Anderson acceleration is to apply an algorithm that requires fewer synchronization points per iteration, i.e. performs multiple dot products

simultaneously or computes an equivalent QR update at a lower parallel cost. Recent research in [16, 17] introduced low synchronization algorithms for MGS, CGS-2 and GMRES [98], and because Anderson acceleration is equivalent to GMRES when solving systems of linear equations [95], it is natural to consider leveraging these advances in Anderson acceleration.

We have implemented and tested three low synchronization algorithms within the SUNDIALS KINSOL [99] implementation of Anderson acceleration. These methods reduce the number of global reductions to a constant number per iteration independent of the size of the Anderson acceleration iteration space. While these methods can require as little as a single synchronization within Krylov methods, a single reduction is not possible within Anderson acceleration due to the requirement for normalization before the LSP solve. Nevertheless, these new methods are able to reduce required global communications to 2 or 3 per iteration.

In addition to demonstrating how to extend these ideas to Anderson acceleration, this chapter also includes a problem independent performance study that illustrates the improved strong scalability of the new kernels, particularly for large iteration spaces and processor counts. The relative cost of the new methods is problem dependent, thus, we present convergence and timing data for multiple examples to demonstrate the reliability of the algorithms for Anderson acceleration and improved performance at scale.

Additionally, we present convergence comparisons for the example problems with new Anderson acceleration variants and introduce the most performant low synchronization orthogonalization strategy, as determined by the performance study, into the methods for consideration. We demonstrate the importance of orthogonalization scheme choice in achieving best performance of Anderson acceleration on large-scale systems.

The rest of the chapter is organized as follows. In Section 6.2, we describe the standard Anderson acceleration algorithm, as well as, recent variants, and relevant components in the QR factorization process. In Section 6.3, we present and compare the low synchronization QR factorization variants. In Section 6.4 we give performance results for the different methods employed within the Anderson acceleration algorithm itself and from several test problems respectively. Section 6.5 presents convergence results for recently developed Anderson acceleration variants, Alternating Anderson acceleration and Composite Anderson acceleration, including a discussion of performance considerations for these methods. Additionally, Section 6.5 includes the introduction of low synchronization orthogonalization routines into Composite Anderson acceleration. Finally, Section 6.6 provides conclusions based on our findings.

6.2 Background

In this section, we present the necessary background for designing scalable Anderson acceleration solvers. A thorough history of Anderson acceleration, as well as, recently developed variants of the method are provided (Section 6.2.1). Additionally, a full discussion of the low synchronization orthogonalization routines introduced into Anderson acceleration in Section 6.4 is presented

in Section 6.3.

6.2.1 Anderson Acceleration

Anderson acceleration (Algorithm 6.3) utilizes a linear combination of m prior function evaluations to accelerate the convergence of the FP iteration. The weights $\gamma_k^{(i)}$ minimize the FP residual norm in the linear case and are computed by solving a LSP (Line 6 in Algorithm 6.3) via a QR factorization of $\mathcal{F}_i$ where Q is a matrix with linearly independent, normalized columns, and R is an upper triangular matrix. Solving the LSP (Algorithm 6.4) requires updating the QR decomposition

Algorithm 6.3: Anderson Acceleration (AA)

Input: x_0, $m \geq 1$, tol, and maxIters
Output: x^*

1 $x_1 = G(x_0)$, $f_0 = G(x_0) - x_0$
2 **for** $i = 1, 2, \ldots,$ maxIters
3 Set $m_i = \min\{m, i\}$ and $f_i = G(x_i) - x_i$
4 Set $\mathcal{G}_i = [\Delta g_{i-m_i}, \ldots, \Delta g_{i-1}]$ with $\Delta g_k = G(x_{k+1}) - G(x_k)$
5 Set $\mathcal{F}_i = [\Delta f_{i-m_i}, \ldots, \Delta f_{i-1}]$ with $\Delta f_k = f_{k+1} - f_k$
6 Determine $\gamma^{(i)} = [\gamma_0^{(i)}, \ldots, \gamma_{m_{i-1}}^{(i)}]^T$ that solves

$$\min_{\gamma^{(i)}} \|f_i - \mathcal{F}_i \gamma^{(i)}\|_2$$

7 Set $x_{i+1} = G(x_i) - \mathcal{G}_i \gamma^{(i)}$
8 **if** $\|x_{i+1} - x_i\| < tol$
9 **Return** x_{i+1} as x^*

to incorporate the vector Δf_{i-1}. This update is comprised of two subroutines: `QRAdd` appends a vector to the QR factorization while performing the necessary orthogonalization and `QRDelete` removes the oldest vector from the factorization. A key factor in the performance and convergence

Algorithm 6.4: Least Squares Problem (LSP) Solve

Input: i, m, m_i, f_i, Δf_{i-1}, $\gamma^{(i)}$, Q, and R
Output: Q, R, and $\gamma^{(i)}$

1 **if** $i = 1$
2 Set $Q_{:,0} = \Delta f_{i-1} / \|\Delta f_{i-1}\|_2$ and $R_{0,0} = \|\Delta f_{i-1}\|_2$
3 **else**
4 **if** $i > m$
5 `QRDelete`(Q, R, m_i)
6 `QRAdd`$(Q, R, \Delta f_{i-1}, m_i)$
7 Solve $R\gamma^{(i)} = Q^T * f_i$

of Anderson acceleration is the number of residual history vectors or depth, m. The depth determines the maximum size of the QR factorization and thus the number of vectors that must be orthogonalized each iteration in `QRAdd`. The start-up phase ($i < m$), the number of vectors increases by one each iteration until m vectors are retained. Once the history is full, the subsequent iterations require orthogonalization against $m - 1$ vectors. As discussed below, the number of synchronizations required by `QRAdd` depends on m and the method for updating the QR factorization. Generally, `QRDelete` does not require any communication, as initially noted in [100]. In practice the depth is often kept small ($m < 5$) however, there are potential convergence benefits to be gained if it were economical, performance-wise, to run with larger m.

There has been an abundance of research performed on proving the convergence and performance benefits of using Anderson acceleration to accelerate the convergence of fixed point iterations, for both linear and non-linear systems of equations.

For linear systems of equations, it is proved that stationary Anderson acceleration without damping is locally r-linearly convergent if the fixed point function map is a contraction and the coefficients in the linear combination remain bounded [101]. This has been extended this analysis of linear problems to Anderson acceleration with damping and proved the exact convergence rate [102]. In [103], it is proved that Anderson acceleration improves the convergence rate of the Picard iteration for solving the steady incompressible Navier-Stokes equations. Recently, [104] derives optimal asymptotic convergence factors for stationary Anderson acceleration and designed tests for estimating the asymptotic convergence speed of nonstationary Anderson acceleration, as well as nonlinear GMRES. These results were extended in [105] via application of stationary Anderson acceleration to an alternating direction method of multipliers method. Finally, proof of convergence for non-smooth fixed point problems in which the nonlinearities can be split into a smooth contractive part and non-smooth contractive part with small Lipschitz constant was given in [106].

Initial work on implementation and performance considerations for Anderson acceleration has been done. Originally, implementation suggestions were made, such as recasting the constrained linear least-squares problem to an unconstrained form and using Givens rotations when deleting a column from the QR factorization [107]. These implementation suggestions were later expanded; [100] provides a detailed discussion and analysis of implementing Anderson acceleration in a distributed parallel environment, employing some of the techniques originally suggested in [107]. Furthermore, we provide a thorough performance analysis of various orthogonalization strategies for updating the QR factorization within Anderson acceleration, suggesting the use of inverse compact WY Gram-Schmidt orthogonalization methods Section 6.4. These methods are discussed in more detail in Section 6.3.

Additionally, it has been shown that Anderson acceleration is similar or equivalent to numerous iterative methods. For example, there has been much work done showing that Anderson acceleration is analagous to nonlinear GMRES methods [108, 109, 110, 111]. Additionally, it has been shown that it is equivalent to the direct inversion of the iterative subspace method (DIIS) [112, 113, 114]. In [95], it is shown that Anderson acceleration without truncation (i.e. m is

equal to the number of iterations required to converge) is equivalent to the linear GMRES method when applied to a linear fixed point problem. Finally, it is noted that Anderson acceleration is similar to quasi-Newton methods, but without the requirement of approximating Jacobians [97].

Recently, there has been work on developing nonstationary forms of Anderson acceleration. Historically, dynamic damping within the method was studied [115], which is built upon with the suggestion of a procedure for adaptively choosing the damping paramater each iteration in [116]. Further work in developing a heuristic strategy for calculating each iteration's damping factor, resulting in faster convergence is shown in [102]. Furthermore, an optimized damping at *each* iteration is introduced in [117]. Alternatively, instead of dynamically determining damping factors, it is proposed to dynamically alternate subspace sizes to potentially further accelerate convergence of the method [118]. Considering convergence benefits alongside performance concerns, alternating iterations of Anderson acceleration with some number of Picard iterations to reduce the overhead of computing the LSP solve was proposed, and showed that this is effective at accelerating the convergence of both linear and nonlinear fixed point iterations [119, 120]. (We discuss this method in more detail in Section 6.2.1.1.) Finally, optimized damping in combination with dynamic subspace sizes is introduced in [121] in the development of a fully composite, nonstationary Anderson acceleration method.

6.2.1.1 Nonstationary Anderson Acceleration

Nonstationary variants of Anderson acceleration are nonstationary because the structure of each iteration can change. Namely, these methods use varying damping parameters (β) or varying subspace sizes (m) at each iteration. Here, we focus on the latter, reviewing the recently developed alternating Anderson acceleration and composite Anderson acceleration methods.

Alternating Anderson Acceleration Alternating Anderson acceleration simply alternates between applying a fixed number of fixed point updates to the iterates with Anderson updates. This process is summarized in Algorithm 6.5. The algorithm only requires a single additional parameter, p, to determine the update required at each iteration, and can ultimately result in far fewer costly QR updates depending on convergence. In practice, this method often yields improved convergence over standalone Anderson acceleration and reduces to standard Anderson acceleration when $p = 1$ [121].

Composite Anderson Acceleration Composite Anderson acceleration is a variant of Anderson acceleration based on the GMRES recursive algorithm and consisting of an inner and outer loop [121]. The outer loop performs the standard Anderson acceleration in Algorithm 6.3, but instead of looping back to the top of the algorithm after calculating the current iterate, x_{i+1}, this value is instead used as the starting point for the inner loop. The inner loop can be any variant of Anderson acceleration run for some number of iterations, and the inner m, m_N, need not be the same as that of the outer loop. This process is summarized in Algorithm 6.6. In [121], it is

Algorithm 6.5: Alternating Anderson Acceleration (A-AA)

Input: x_0, $m \geq 1$, p, tol, and maxIters. [AA: $p = 1$, A-AA: $p > 1$]

Output: x^*

$x_1 = G(x_0)$ and $f_0 = G(x_0) - x_0$

for $i = 1, 2, \ldots,$ maxIters

  Set $m_i = \min\{m, i\}$ and $f_i = G(x_i) - x_i$

  Set $\mathcal{G}_i = [\Delta g_{i-m_i}, \ldots, \Delta g_{i-1}]$ with $\Delta g_k = G(x_{k+1}) - G(x_k)$

  Set $\mathcal{F}_i = [\Delta f_{i-m_i}, \ldots, \Delta f_{i-1}]$ with $\Delta f_k = f_{k+1} - f_k$

  if $i \bmod p \neq 0$

    Set $x_{i+1} = G(x_i)$

  else

    Determine $\gamma^{(i)} = [\gamma_0^{(i)}, \ldots, \gamma_{m_{i-1}}^{(i)}]^T$ that solves

$$\min_{\gamma^{(i)}} \|f_i - \mathcal{F}_i \gamma^{(i)}\|_2$$

    Set $x_{i+1} = G(x_i) - \mathcal{G}_i \gamma^{(i)}$

  if $\|x_{i+1} - x_i\| < tol$

    return x_{i+1} as x^*

Algorithm 6.6: Composite Anderson Acceleration (C-AA)

Input: x_0, $m \geq 1$, m_N, tol, maxIters, and maxIters$_N$. [AA:maxIters$_N = 0$ C-AA:maxIters$_N > 0$]

Output: x^*

$x_1 = G(x_0)$ and $f_0 = G(x_0) - x_0$

for $i = 1, 2, \ldots,$ maxIters

  Set $m_i = \min\{m, i\}$ and $f_i = G(x_i) - x_i$

  Set $\mathcal{G}_i = [\Delta g_{i-m_i}, \ldots, \Delta g_{i-1}]$ with $\Delta g_k = G(x_{k+1}) - G(x_k)$

  Set $\mathcal{F}_i = [\Delta f_{i-m_i}, \ldots, \Delta f_{i-1}]$ with $\Delta f_k = f_{k+1} - f_k$

  Determine $\gamma^{(i)} = [\gamma_0^{(i)}, \ldots, \gamma_{m_{i-1}}^{(i)}]^T$ that solves

$$\min_{\gamma^{(i)}} \|f_i - \mathcal{F}_i \gamma^{(i)}\|_2$$

  Set $x_{i+1/2} = G(x_i) - \mathcal{G}_i \gamma^{(i)}$

  Set $\hat{x}_0 = x_{i+1/2}$

  Call AA($\hat{x}_0$, m_N, maxIters$_N$)

  Set $x_{i+1} = \hat{x}_N$

  if $\|x_{i+1} - x_i\| < tol$

    return x_{i+1} as x^*

demonstrated that composite Anderson acceleration often has much faster convergence than standard Anderson acceleration, particularly for larger m (20) and small m_N (1-2). Additionally, it is worth noting that in Algorithm 6.6, when the number of inner iterations is greater than zero,

$\text{maxIters}_N > 0$, but $m_N = 0$, then the algorithm reduces to Algorithm 6.5 where $p = \text{maxIters}_N$.

While both alternating Anderson acceleration and composite Anderson acceleration reduce the number of LSP updates by driving down the required iterations for convergence, they do not eliminate or reduce the main cost in the LSP updates: the orthogonalization of the new Δf_i vector against all previous vectors in the Anderson acceleration subspace. Because many more applications are utilizing Anderson acceleration with the requirement of $m \in [20, 100]$, it remains imperative to reduce the cost of synchronization for the LSP, even if a method such as alternating or composite Anderson acceleration is being used. In Section 6.3, we review various orthogonalization routines for use within Anderson acceleration, in particular, we review various low synchronization orthogonalization routines and how to incorporate them into the LSP solve of Anderson acceleration.

Remark. *Throughout this chapter, all presented performance and convergence results were obtained through use of the SUNDIALS library. New orthogonalization strategies and Anderson acceleration variants were implemented therein. SUNDIALS is a solver library of time integrators for ordinary differential equations and differential-algebraic equations, as well as, solvers for nonlinear algebraic equations [122]. In this work, we focus on the KINSOL package of SUNDIALS, which comprises the nonlinear solvers, specifically we make use of the KINFP fixed point solver [99].*

The entire SUNDIALS software stack is built upon vector abstractions, called N_Vectors. These abstractions provide various interfaces for running SUNDIALS on both CPUs and GPUs for serial and distributed settings. In this work, the Parallel N_Vector is used for distributed CPU performance tests, while the MPIPlusX N_Vector with a local CUDA N_Vector is used as the local vector for each MPI rank [123]. As a result of the abstractions, inter-GPU communication for the GPU performance tests is still staged through a single host process.

6.3 QR Update Methods in Anderson Acceleration

In this section, we discuss the `QRAdd` kernel, Line 6 of Algorithm 6.4, employed to update the Anderson acceleration iteration space with the Δf_{i-1} vector. We present the baseline `QRAdd_MGS` (modified Gram Schmidt) algorithm alongside three low synchronization variants of the kernel implemented within Anderson acceleration in SUNDIALS. For each of the `QRAdd` kernels, we discuss the form of the projectors applied to orthogonalize the given set of vectors, $\mathcal{F}_i$, as well as, their predicted parallel performance.

Throughout this section and the remainder of the chapter, matrix entries are referenced via subscripts with 0-based indexing. For example, $M_{0,0}$ refers to the first row, first column of a matrix M, and slices of a matrix are inclusive, i.e. $M_{:,0:k-1}$ refers to all rows of the matrix and columns $0, \ldots, k-1$. Additionally, all matrices and vectors are assumed to be allocated with the maximum space requirements at the time of solver setup.

6.3.1 Modified Gram Schmidt

The standard approach for updating the QR factorization within Anderson acceleration is to apply MGS, outlined in Algorithm 6.7. In each Anderson acceleration iteration, applying MGS requires m_i dot products; $m_i - 1$ for the orthogonalization against all previous vectors in the Anderson acceleration iteration space, and one for the normalization at the end of the algorithm.

Algorithm 6.7: `QRAdd_MGS`

Input: Q, R, Δf_{i-1} and m_i
Output: Q, R

1 **for** $j = 0, \ldots, m_i - 2$
2 $\quad R_{j,m_i-1} \leftarrow Q_{:,j}^T * \Delta f_{i-1}$ → Global Reduction
3 $\quad \Delta f_{i-1} \leftarrow \Delta f_{i-1} - R_{j,m_i-1} * Q_{:,j}$
4 $R_{m_i-1,m_i-1} \leftarrow \|\Delta f_{i-1}\|_2$ → Global Reduction
5 $Q_{:,m_i-1} \leftarrow \Delta f_{i-1} / R_{m_i-1,m_i-1}$

This algorithm results in m_i synchronizations across all processes, as the reductions form a chain of dependencies and can not be performed in tandem. The high costs of these synchronizations are exacerbated by the lack of computational workload between each reduction, namely the entire algorithm only requires $\mathcal{O}(m_i n)$ flops.

6.3.2 ICWY Modified Gram Schmidt

As stated in Section 2.3.1, the complexity of ICWY-MGS is $\mathcal{O}(m_i n^2)$. When implementing ICWY-MGS in the context of Anderson acceleration, lagging the normalization until the subsequent iteration is not an option like in GMRES, as the factorization is applied immediately after updating in the LSP solve on Line 6 of Algorithm 6.4. The resulting QR update algorithm within Anderson acceleration is detailed in Algorithm 6.8.

Algorithm 6.8: `QRAdd_ICWY`

Input: Q, R, T, Δf_{i-1} and m_i
Output: Q, R, T

1 $T_{m_i-2,0:m_i-2} \leftarrow Q_{:,0:m_i-2}^T * Q_{:,m_i-2}$ → Delayed Reduction
2 $R_{0:m_i-2,m_i-1} \leftarrow Q_{:,0:m_i-2}^T * \Delta f_{i-1}$ → Global Reduction
3 $T_{m_i-2,m_i-2} \leftarrow 1$
4 $R_{0:m_i-2,m_i-1} \leftarrow T_{0:m_i-2,0:m_i-2}^{-1} R_{0:m_i-2,m_i-1}$
5 $\Delta f_{i-1} \leftarrow \Delta f_{i-1} - Q_{:,0:m_i-2} * R_{0:m_i-2,m_i-1}$
6 $R_{m_i-1,m_i-1} \leftarrow \|\Delta f_{i-1}\|_2$ → Global Reduction
7 $Q_{:,m_i-1} \leftarrow \Delta f_{i-1} / R_{m_i-1,m_i-1}$

Here, we present a two reduction variant of the ICWY-MGS algorithm. The formation of the correction operator and the matrix R can be merged on Line 2. With the inclusion of the

normalization at the end of the algorithm, this results in two synchronizations per iteration until the Anderson acceleration iteration space is filled. Once Anderson acceleration reaches Line 5 in Algorithm 6.4, the oldest vector in the factorization is deleted. While this can be performed with Givens rotations in cases where only Q and R are stored, the explicit storage and application of T requires that this correction matrix be updated by introducing a single reduction. Overall, this process results in two global reduction steps per iteration until m is reached, after which, there are three global reductions per iteration.

6.3.3 CGS-2

As presented in Section 2.3.2, the CGS-2 algorithm involves application of a projector with an embedded correction matrix. Algorithm 6.9 details the process of updating the Anderson acceleration QR factorization with an additional vector without explicitly forming the correction matrix. Reorthogonalization happens explicitly on Lines 3 and 4 before being added back into the matrix R on Line 5.

Algorithm 6.9: `QRAdd_CGS2`

Input: Q, R, Δf_{i-1} and m_i
Output: Q, R

1 $s \leftarrow Q^T_{:,0:m_i-2} * \Delta f_{i-1}$ → Global Reduction
2 $y \leftarrow \Delta f_{i-1} - Q_{:,0:m_i-2} * s$
3 $z \leftarrow Q^T_{:,0:m_i-2} * y$ → Global Reduction
4 $\Delta f_{i-1} \leftarrow y - Q_{:,0:m_i-2} * z$
5 $R_{0:m_i-2,m_i-1} \leftarrow s + z$
6 $R_{m_i-1,m_i-1} \leftarrow \|\Delta f_{i-1}\|_2$ → Global Reduction
7 $Q_{:,m_i-1} \leftarrow \Delta f_{i-1}/R_{m_i-1,m_i-1}$

Overall, this process requires three reductions to be performed per iteration—two for the orthogonalization and reorthogonalization of Δf_{i-1} against all the vectors in Q and one for the final normalization of Δf_{i-1}. The amount of computation per iteration has increased to approximately $2m_in$ or $\mathcal{O}(m_in)$. While the amount of computation is still on the same order as MGS, the workload has approximately doubled, and, fortunately, there are no additional storage requirements.

6.3.3.1 DCGS-2

Algorithm 6.10 details our `QRAdd_DCGS2` implementation in which there are two synchronizations per iteration. Because Anderson acceleration requires using the QR-factorization to solve the LSP at the end of *every* iteration, it is not possible to lag the normalization of the vector Δf_{i-1} to the next iteration as in the stable variant discussed in Section 2.3.2. The global reduction for the reorthogonalization of the vectors can be still be performed at the same time as the orthogonalization of Δf_{i-1} against Q. The synchronization on Line 1 is lagged to coincide

Algorithm 6.10: `QRAdd_DCGS2`

Input: Q, R, Δf_{i-1} and m_i
Output: Q, R

1 $R_{0:m_i-2,m_i-1} \leftarrow Q^T_{:,0:m_i-2} * \Delta f_{i-1}$ → Delayed Reduction
2 **if** $m_i > 3$
3 $\quad s \leftarrow Q^T_{:,0:m_i-3} Q_{:,m_i-2}$ → Global Reduction
4 $\quad Q_{:,m_i-2} \leftarrow Q_{:,m_i-2} - Q_{:,m_i-3} * s$
5 $\quad R_{0:m_i-3,m_i-2} \leftarrow R_{0:m_i-3,m_i-2} + s$
6 $\Delta f_{i-1} \leftarrow \Delta f_{i-1} - Q_{:,0:m_i-2} * R_{0:m_i-2,m_i-1}$
7 $R_{m_i-1,m_i-1} \leftarrow \|\Delta f_{i-1}\|_2$ → Global Reduction
8 $Q_{:,m_i-1} \leftarrow \Delta f_{i-1} / R_{m_i-1,m_i-1}$

with Line 3. The amount of computation for DCGS-2 increases over that of CGS-2 except on the first iteration where it performs the same number of flops as MGS. Lagging the reorthogonalization and introducing the operation $Q^T_{:,0:m_i-3} Q_{:,m_i-2}$, makes this method $\mathcal{O}(m_i n^2)$, the same as ICWY-MGS.

6.4 Low Synchronization QR Update Methods in Anderson Acceleration

In this section, we detail the expected performance for each of the `QRAdd` algorithms implemented within SUNDIALS as determined by the total number of synchronizations to be performed, as well as a performance study demonstrating their measured performance.

6.4.1 Performance Expectations

For each of the `QRAdd` algorithms, the total number of synchronizations performed is dependent upon the number of iterations required for convergence. In the case of `QRAdd_MGS`, this is given by

$$\text{Syncs}_{\text{MGS}} = \sum_{j=1}^{m-1} j + \sum_{j=m}^{k} m = \frac{(m-1)\cdot m}{2} + m\cdot(k-m) + 1 \tag{6.4}$$

where k is the iterations required to converge. This summation is a result of MGS requiring an additional inner product at every single iteration, up until the Anderson acceleration subspace is filled. The size of each inner product, however, remains constant with each contributing process summing over their local values, then performing a reduction across all processes to a single value.

For the low synchronization orthogonalization update methods, the number of synchronizations performed each iterations remains constant, but the size of the inner products changes at each iteration. For `QRAdd_ICWY`, the total number of synchronizations performed is

$$\text{Syncs}_{\text{ICWY}} = 2\cdot k + 1, \tag{6.5}$$

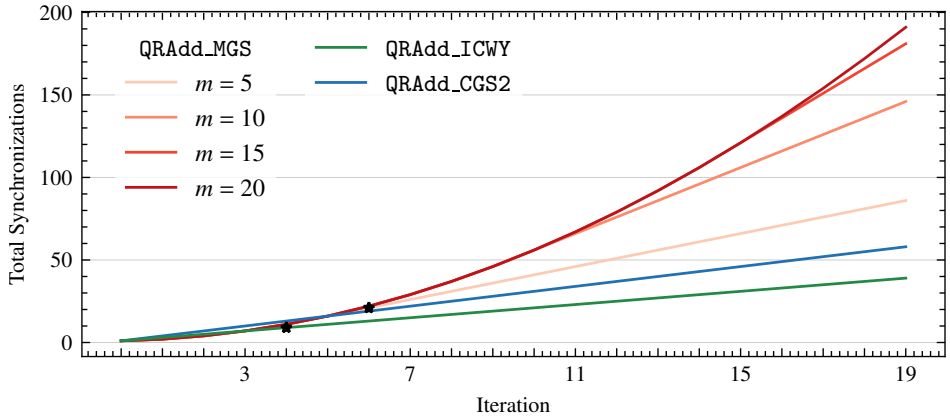


Figure 6.2: Total number of synchronizations performed up to a given iteration for each of the QRAdd strategies when implemented within SUNDIALS. Shown here is the difference between `QRAdd_MGS`, `QRAdd_ICWY`, and the `QRAdd_CGS2` routines, assuming optimal implementation of the algorithms. The number of iterations at which `QRAdd_MGS` has no longer performed fewer inner products is denoted by the black dots on the plots.

and the size of the synchronization increases each iteration to include double the number of previous vectors orthogonalized against, up until the Anderson acceleration space is full. The final synchronization occurs for normalization of the newly appended vector to the QR factorization.

`QRAdd_CGS2` requires 3 synchronizations per iteration. Unlike `QRAdd_ICWY`, the first two synchronizations can not be performed in the same buffer because the second synchronization is a reorthogonalization which has dependencies on computations after the first synchronization. The third and final inner product is the normalization of the vector being added to the QR factorization, thus the total number of synchronizations is given by

$$\text{Syncs}_{\text{CGS2}} \quad = \quad 3 \cdot k + 1. \tag{6.6}$$

The dependencies in `QRAdd_CGS2` are removed in `QRAdd_DCGS2` due to the lagged reorthogonalization. Therefore, `QRAdd_DCGS2` results in the same number of synchronizations as `QRAdd_ICWY`,

$$\text{Syncs}_{\text{DCGS2}} \quad = \quad 2 \cdot k + 1. \tag{6.7}$$

Predicting which `QRAdd` kernel will be most performant is done by simply determining the total number of synchronizations required for each algorithm. In Figure 6.2, we show the total number of synchronizations to be performed by each of the given `QRAdd` kernels for a given number of iterations. At six iterations, independent of m, one of the low synchronization `QRAdd` routines will result in fewer total synchronizations — crossover points are depicted by black stars in Figure 6.2. Note, this is under the assumption that the inner products within the `QRAdd` routines are the dominating cost. Figure 6.2 also depicts the crossover point for when `QRAdd_ICWY` or `QRAdd_DCGS2` results in fewer overall synchronizations which happens as early as four iterations. Section 6.4.2 confirms these performance expectations.

6.4.2 Performance Study

In this section, we detail the GPU and CPU performance of the `QRAdd` variants from Section 6.3. We differentiate between two performance costs: the time required to fill the Anderson acceleration iteration space, defined as "Start-Up Iterations" and the time required to add an additional vector to the space once it is filled, termed "Recycle Iteration". In the case of CPU performance, a parallel strong-scaling study is performed. With GPUs, we perform a weak-scaling study, because a common goal is to saturate the devices, and thus avoid under-utilization.

Matrices and vectors containing n rows are partitioned row-wise across p processes when performing strong-scaling studies on CPUs. That is, each process (or CPU core) contains n/p contiguous rows of a given matrix or vector. For GPU performance studies, we provide the local vector size per GPU participating in the computation. All tests are performed on the LLNL Lassen supercomputer, for GPU performance tests each GPU is paired with a single MPI rank as the host process. Each test is performed 20 times and for 10 iterations (for a total of 200 timings); for each individual test the maximum time required by any single process or GPU is recorded, and the minimum time across all tests is presented.

The results presented are based on the Anderson acceleration implementation within SUNDIALS. Parallel CPU tests are implemented with a so-called node-local vector abstraction, called the Parallel N_Vector. Similarly, the GPU tests are implemented with the MPIPlusX N_Vector where the CUDA N_Vector is used as the local vector for each MPI rank [123]. Notably, while we use the CUDA N_Vector as the local vector portion, this can be switched with any N_Vector implementation in SUNDIALS. Due to this abstraction, communication between GPUs is still performed by staging data through the host process and not via CUDA-aware technologies.

The low synchronization methods added to SUNDIALS leverage the fused dot product operation to perform matrix-vector products with Q^T. This operation enables computing the dot product of a single vector with multiple vectors (the columns of Q) as a single operation requiring only one MPI call. For ICWY and DCGS-2 a new N_Vector operation was introduced in SUNDIALS enabling delayed synchronization by separating the local reduction computation and final global reduction into separate operations. Both of these fused operations perform a one-time copy of the independent vector data pointers into a device buffer accessed by the fused operations. In addition to combining multiple reductions into a single call, these operations reduce the number of kernel launches in the GPU case. We discuss further in Section 6.4.2.1.

6.4.2.1 Start-Up Iterations

We first consider the start-up iteration times, displayed in Figures 6.3 and 6.4. On the CPU we use a global vector of size 1 048 576, and for the GPU a local vector size of 1 500 000. The number of iterations performed is equivalent to the number of vectors, m, in the Anderson acceleration iteration space.

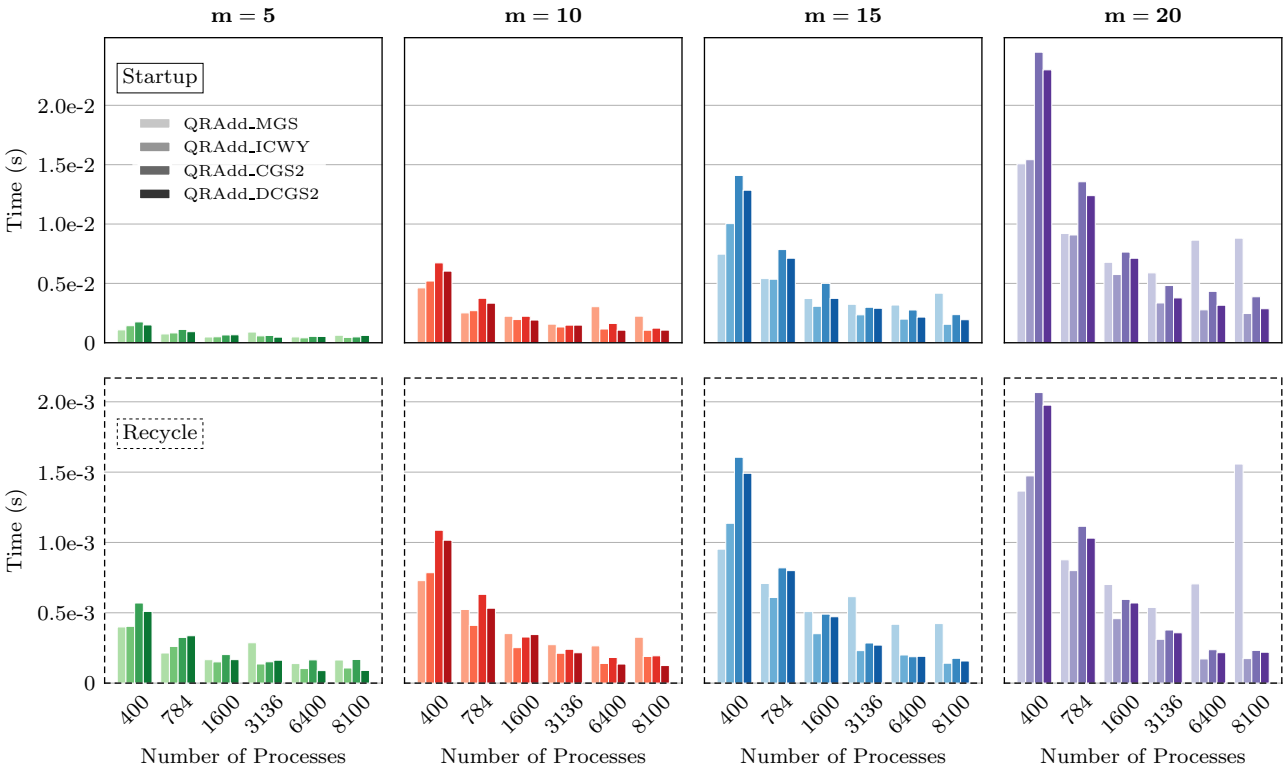


Figure 6.3: The cumulative time for the **start-up iterations (top)** and **single recycle iteration (bottom)** of the four different `QRAdd` kernels within AA for a vector of size 1 048 576 on a varying number of Lassen **CPU** cores.

The top rows of Figure 6.3 (CPU) and Figure 6.4 (GPU) display the time required to fill the Anderson acceleration iteration space. As part of this operation, `QRAdd_MGS` requires $\sum_{k=1}^{m} k = \frac{m^2+m}{2}$ total synchronizations, while `QRAdd_ICWY`, `QRAdd_CGS2`, and `QRAdd_DCGS` require $2m-1$, $3m-2$, and $2m-1$, respectively.

Predicting performance based solely on the number of dot products performed, we expect that `QRAdd_ICWY` and `QRAdd_DCGS2` will become faster than `QRAdd_MGS` after $m = 3$ and `QRAdd_CGS2` will outperform `QRAdd_MGS` after $m = 6$, for processor counts where the synchronization imposed by dot products is the dominate cost. This is consistent with the CPU performance seen in Figure 6.3. The benefits of applying low synchronization `QRAdd` algorithms is not observed until the processor count reaches 1600 or higher when the global reduction costs lead to larger bottlenecks. As m increases, the disparity between the time required for `QRAdd_MGS` and the low synchronization variants greatly increases.

Our observed GPU performance depends on the number of kernel calls for each `QRAdd` subroutine in addition to the number of global reductions. Compared to `QRAdd_MGS`, `QRAdd_ICWY` performs fewer CUDA kernels each iteration for $m > 3$, while `QRAdd_CGS2` and `QRAdd_DCGS2` perform four additional CUDA kernel calls for $m > 1$. Also, all three low synchronization variants perform sequential on-CPU updates to R, unlike `QRAdd_MGS`. Finally, in addition to the kernel launches and sequential updates, the low synchronization routines increase the amount of data

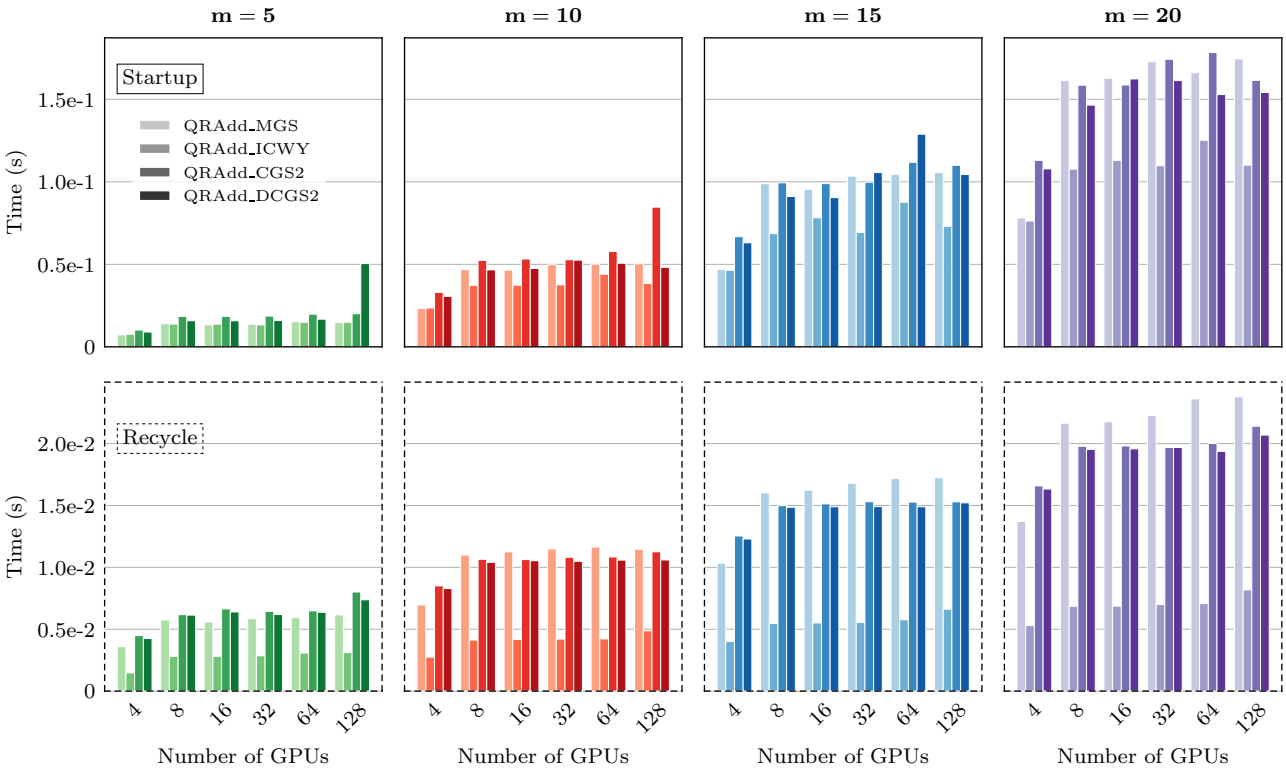


Figure 6.4: The cumulative time for the **start-up iterations (top)** and **single recycle iteration (bottom)** of the four different `QRAdd` kernels within AA for a local vector of size 1 500 000 on a varying number of Lassen **GPUs**.

being moved between the host and the GPU. Specifically, the fused dot products employed by these routines require that at most $m_i - 1$ values be copied to and from the GPU multiple times per iteration while `QRAdd_MGS` copies only a single value per individual dot product.

All three low synchronization variants incur the cost of two kernel launches that require data transfers to and from the host process. However, in the case of `QRAdd_ICWY`, the lower computational requirements result in fewer overall CUDA kernel calls than for `QRAdd_CGS2` and `QRAdd_DCGS2`. Specifically, Line 2 in Algorithm 2.4 and Line 4 in Algorithm 6.10 correspond to additional vector updates that are not required by `QRAdd_ICWY`; hence their performance times remain higher than `QRAdd_ICWY` independent of GPU count or m. There are minimal performance gains when using `QRAdd_CGS2` or `QRAdd_DCGS2` for filling the Anderson acceleration iteration space, as `QRAdd_DCGS2` is only slightly faster than `QRAdd_MGS` when more than 8 GPUs are used and only for $m = 15$ and larger.

6.4.2.2 Recycle Iterations

We next consider the cost of the `QRAdd` kernel once the Anderson acceleration iteration space is filled, namely the time required to orthogonalize a single vector against $m - 1$ vectors. Assuming a large iteration count is required for convergence, this cost reflects the expected run-time of the

QRAdd kernel for the majority of a given solve. We label these times as the "Recycle Iteration" time in Figures 6.3 and 6.4. The CPU and GPU timings presented are for the same global and local vector size as presented in Section 6.4.2.1, and the results are for a *single* iteration. Note that this time does not include the additional synchronization introduced by ICWY into QRDelete, first mentioned in Section 6.3 and discussed further in Section 6.4.3.

Once the Anderson acceleration iteration space is filled, the per iteration cost is greatly decreased by using one of the low synchronization orthogonalization algorithms when operating on CPUs (see the bottom row of Figure 6.3). While the "Startup Iterations" did not result in performance gains on CPUs until after 1600 processes, for $m > 5$, we observe performance gains with QRAdd_ICWY with process counts as low as 784. QRAdd_MGS is still faster at smaller scales for all values of m due to the reduced synchronization costs of performing an MPIAll_Reduce on a small, closed-set number of processes. The performance gains of the low synchronization algorithms at larger scales ranges from 2–8× speedup for $m = 10$, 15, and 20 at 8100 processes. With this drastic speedup for each iteration, we expect much larger performance gains for test problems that run to convergence.

Unlike the CPU performance for Anderson acceleration, QRAdd_ICWY demonstrates performance gains independent of the number of GPUs participating in the computation or the size of the Anderson acceleration iteration space, m. This is seen in the bottom row of Figure 6.4, which displays the performance for a single "Recycle Iteration" on multiple GPUs. While QRAdd_CGS2 and QRAdd_DCGS2 do not see the same performance improvements as QRAdd_ICWY, they do begin exhibiting faster performance than QRAdd_MGS beginning at 8 GPUs and for $m = 10$ and larger, though gains are modest.

6.4.3 Numerical Experiments

In this section, we highlight the strong-scaling parallel efficiency of standard Anderson acceleration compared with Anderson acceleration with low synchronization orthogonalization for test problems run to convergence. The example problems provided are not exhaustive, but the selected tests do stress the performance of low synchronization Anderson acceleration. In addition, application specific tests are used to demonstrate the benefits of low synchronization Anderson acceleration for both distributed CPU and GPU computing environments. All experiments are performed on the LLNL Lassen supercomputer, with the same setup as described in Section 6.4.2. To account for machine variability, each run is executed 10 times and we report the minimum.

6.4.3.1 Anisotropic 2D Heat Equation + Nonlinear Term

This test problem highlights the performance of all Anderson acceleration variants for various m and iterations to convergence. We consider a steady-state 2D heat equation with an additional

nonlinear term $c(u)$,

$$u_{xx} + u_{yy} + c(u) = f \quad \text{in} \quad \mathcal{D} = [0,1] \times [0,1] \tag{6.8}$$

$$u = 0 \quad \text{on} \quad \partial\mathcal{D}. \tag{6.9}$$

The chosen analytical solution is

$$u_{\text{exact}} = u(x,y) = \sin^2(\pi x)\sin^2(\pi y), \tag{6.10}$$

hence, the static term f is defined as follows

$$\begin{aligned} f(x,y) = &2\pi^2(\cos^2(\pi x) - \sin^2(\pi x))\sin^2(\pi y) \\ &+ 2\pi^2(\cos^2(\pi y) - \sin^2(\pi y))\sin^2(\pi x) \\ &+ c(u_{\text{exact}}). \end{aligned} \tag{6.11}$$

The spatial derivatives are computed using second-order centered differences, with the data distributed over 1024×1024 points on a uniform spatial grid, resulting in a system of equations of size $1\,048\,576 \times 1\,048\,576$. The Laplacian term is implemented as a matrix-vector product giving the the algebraic system as

$$A\mathbf{u} + c(\mathbf{u}) = \mathbf{b}. \tag{6.12}$$

where $\mathbf{u}$ denotes the discrete vector of unknowns. Solving for $\mathbf{u}$ results in the following FP formulation

$$\mathbf{u} = G(\mathbf{u}) = A^{-1}(\mathbf{b} - c(\mathbf{u})). \tag{6.13}$$

We use the SUNDIALS PCG solver to solve the linear system with the hypre PFMG preconditioner performing two relaxation sweeps per iteration. Both the FP nonlinear solver and the PCG linear solver set a stopping criteria with a tolerance of 10^{-10}. A zero vector is used as the starting guess in all cases.

Nonlinear Term 1 As a first example, consider the nonlinear reaction term

$$c(u) = u + ue^u + ue^{-u} + (u - e^u)^2. \tag{6.14}$$

In this case, Anderson acceleration exhibits rapid convergence when $m = 5$, requiring 11 iterations to converge for all variants. Figure 6.5 displays the overall time to convergence, split into $G(\mathbf{u})$ evaluation time and time spent in Anderson acceleration. In general, the $G(\mathbf{u})$ performance is volatile, most likely due to the sparse matrix operations required for the linear solve; thus we focus on the Anderson acceleration performance in the right of Figure 6.5.

The time spent in Anderson acceleration meets expectations, based on the results in Section 6.4.2. For such a small m, we observe minimal performance improvements over Anderson acceleration with MGS, particularly at small process counts. However, the observed improvements

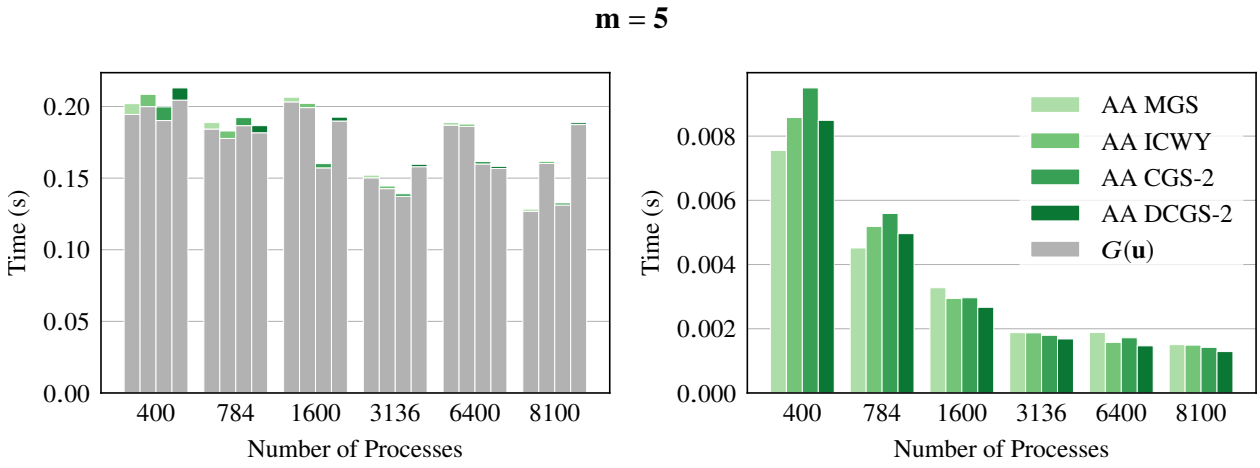


Figure 6.5: Time to convergence for **Heat 2D + Nonlinear Term 1** (11 iterations for all cases), using FP + AA with $m = 5$, including the function evaluation, $G(\mathbf{u})$, (left) and only time in AA(right).

begin around 1600 processes.

Nonlinear Term 2 As a second example, consider

$$c(u) = 100 \cdot (u - u^2). \tag{6.15}$$

With $m = 10$, Anderson acceleration requires more time to converge than for the $c(u)$ in Section 6.4.3.1, better highlighting potential performance benefits of the low synchronization orthogonalization algorithms.

Figure 6.6 displays the overall timing results, with the number of iterations required to converge listed on top of each bar. In most cases, the number of iterations is between 27 and 42, depending on the orthogonalization method and processor count. The three exceptions are for Anderson acceleration with DCGS-2, which requires more iterations to converge, particularly for 1600–6400 processes. This is expected since our `QRAdd_DCGS2` cannot lag the normalization and exhibits the same instability as Hernandez's original algorithm.

In most cases, Anderson acceleration with DCGS-2 results in degraded performance in comparison to the other variants due to the high number of iterations required to converge. Anderson acceleration with CGS-2 performs best, starting at 1600 processes and continues to be the fastest through 8100 processes. Although `QRAdd_ICWY` only requires two global synchronizations per iteration, there is an additional synchronization in `QRDelete` to update the matrix T, resulting in an implementation that is more comparable to Anderson acceleration with CGS-2 at higher process counts.

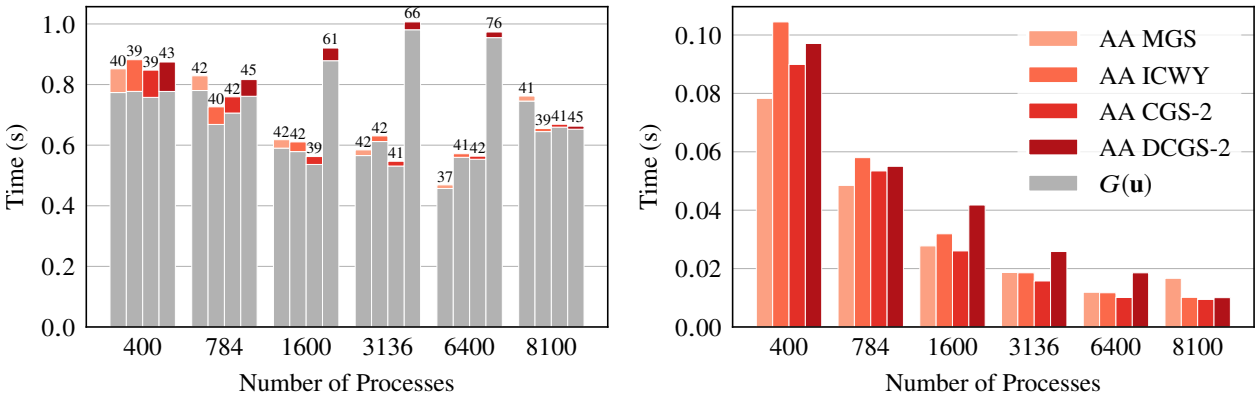


Figure 6.6: Time to convergence for **Heat 2D + Nonlinear Term 2**, using FP + AA with $m = 10$, including the function evaluation, $G(\mathbf{u})$, and the number of iterations to convergence (left) and only the time spent in AA (right).

6.4.3.2 2D Bratu Problem

We next consider an example from [95] (and originating from [124]). The Bratu problem is a nonlinear PDE boundary value problem defined as:

$$u_{xx} + u_{yy} + \lambda e^u = 0 \quad \text{in} \quad \mathcal{D} = [0,1] \times [0,1] \tag{6.16}$$

$$u = 0 \quad \text{on} \quad \partial\mathcal{D}. \tag{6.17}$$

We again solve the problem with centered differencing, resulting in a system of the form

$$A\mathbf{u} + \lambda e^{\mathbf{u}} = 0. \tag{6.18}$$

The associated FP function is then

$$G(\mathbf{u}) = A^{-1}\left(-\lambda e^{\mathbf{u}}\right) = \mathbf{u}. \tag{6.19}$$

We use a uniform grid of 1024×1024 points, which results in a system of equations of size $1\,048\,576 \times 1\,048\,576$. We select $\lambda = 6.7$ for testing, as it is close to the theoretical critical point, as discussed in [125], and is a difficult problem to solve. The SUNDIALS PCG solver is applied to perform the linear system solve with the hypre PFMG preconditioner performing two relaxation sweeps. Both the nonlinear and linear solver employed a tolerance of 10^{-10}. A zero vector is set as the starting guess.

For this example problem, $m = 30$, and the number of iterations required to converge for all variants of Anderson acceleration is less than 30; hence the timing results reflect the performance benefits of the `QRAdd` subroutines for the startup iterations. The strong-scaling timing and convergence results are presented in Figure 6.7. For this test case, we observe improvements with

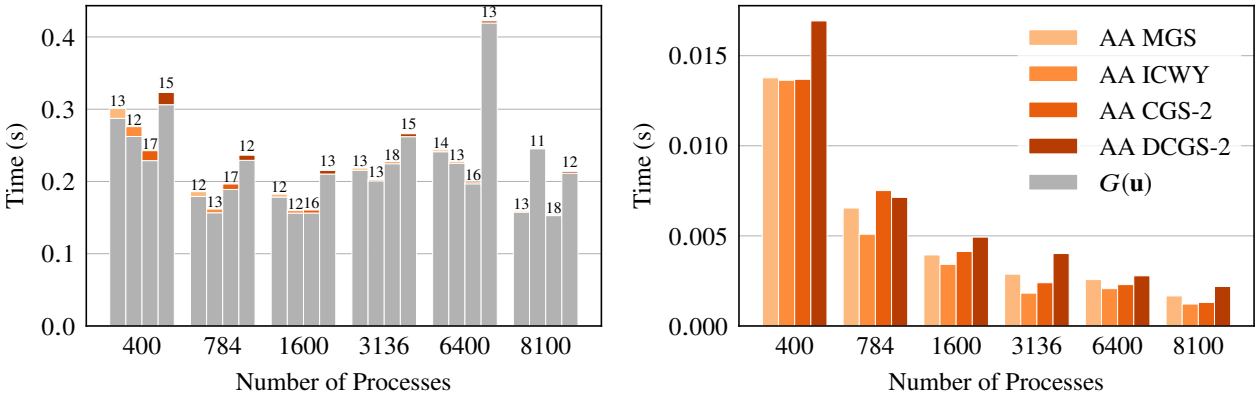


Figure 6.7: Time to convergence for **2D Bratu**, using FP + AA with $m = 30$, including the function evaluation, $G(\mathbf{u})$, and the number of iterations to convergence (left) and only the time spent in AA (right).

ICWY from the very beginning with 400 processes (although only slightly in this case). Anderson acceleration with ICWY continues to outperform up to 8100 processes. This is consistent with the results in Section 6.4.2.1 in which `QRAdd_ICWY` performed best for $m = 20$ as it only requires two dot products per iteration, one fewer than `QRAdd_CGS2` and has approximately the same amount of computation as `QRAdd_DCGS2`.

6.4.3.3 Expectation-Maximization Algorithm for Mixture Densities

For this example, we consider a variation of the expectation-maximization test problem presented in [95]. Consider a mixture density of three univariate normal densities with a mixture density given by $p(x) = \sum_{i=1}^{3} \alpha_i p_i(x|\mu_i, \sigma_i)$, with

$$p_i(x|\mu_i, \sigma_i) = \frac{1}{\sqrt{2\pi}\sigma_i} e^{-(x-\mu_i)^2/(2\sigma_i^2)}, \qquad 1 \leq i \leq 3 \tag{6.20}$$

Mixture proportions $\{\alpha_i\}_{i=1}^{3}$ are non-negative and sum to one. The mixture proportions and variances are assumed to be known and the means $\{\mu_i\}_{i=1}^{3}$ are estimated from a set of unlabeled samples $\{x_k\}_{k=1}^{N}$, or samples of unknown origin. Determining the unknown mean distribution parameters is given by the FP function

$$G(\mu_i) = \mu_i = \frac{\sum_{k=1}^{N} x_k \frac{\alpha_i p_i(x_k|\mu_i,\sigma_i)}{p(x_k)}}{\sum_{k=1}^{N} \frac{\alpha_i p_i(x_k|\mu_i,\sigma_i)}{p(x_k)}}, \qquad 1 \leq i \leq 3 \tag{6.21}$$

with current mean estimations $\{\mu_i\}_{i=1}^{3}$ being applied alongside the known mixture proportions and variances to determine the subsequent estimations until convergence. We keep the same mixture proportions and variances as the original test case, $(\alpha_1, \alpha_2, \alpha_3) = (0.3, 0.3, 0.4)$ and

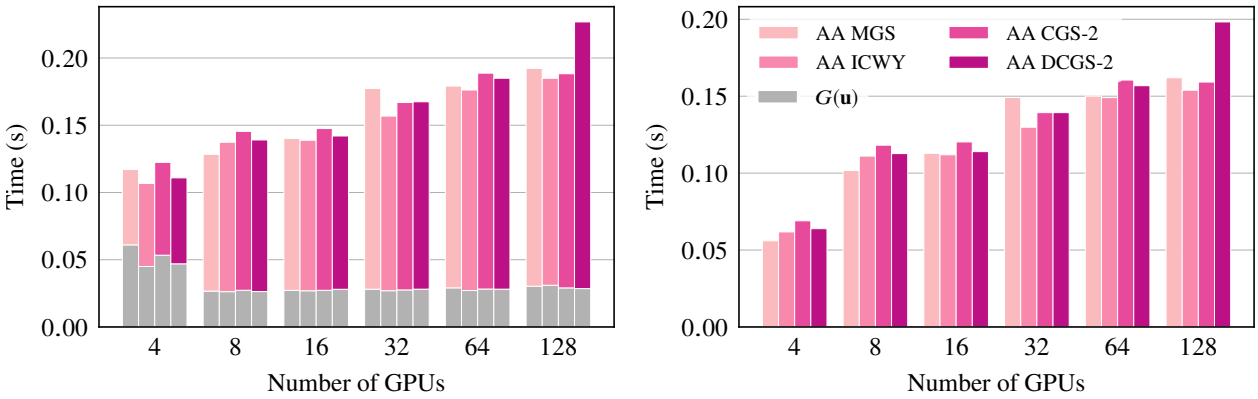


Figure 6.8: Time to convergence for **Expectation-Maximization** (21 iterations for all cases), using FP + AA with $m = 3$ and a local vector size of 1 500 000 for each GPU, including the function evaluation, $G(\mathbf{u})$, (left) and only the time spent in AA (right).

$(\sigma_1, \sigma_2, \sigma_3) = (1, 1, 1)$. We generated 100 000 samples for the mean distribution set $(\mu_1 = 0, \mu_2 = 0.5, \mu_3 = 1.0)$, corresponding to a poorly separated mixture, and used the same Anderson acceleration parameter of $m = 3$ as Walker and Ni [95].

For our test case, we estimate a single set of mean distribution parameters redundantly for every entry in a global vector, where the vector takes the form

$$\mathbf{u} = \left[\{\mu_i\}_{i=1}^3 \quad \cdots \quad \{\mu_i\}_{i=1}^3 \right]. \tag{6.22}$$

We do this to simulate a function that requires no communication other than that imposed by Anderson acceleration. The resulting FP function to be solved is then given by

$$G(\mathbf{u}) = \left[\{G(\mu_i)\}_{i=1}^3 \quad \cdots \quad \{G(\mu_i)\}_{i=1}^3 \right]. \tag{6.23}$$

Because m is small for this test, we expect to see only modest improvements in performance of the low synchronization routines over MGS. For $m = 3$, ICWY and DCGS-2 reduce the number of synchronizations per iteration by one over MGS, with ICWY gaining an additional synchronization after the space is filled.

The test is performed as a weak-scaling study with each GPU operating on a local vector size of 1 500 000 values. Tests were run with a tolerance of 10^{-8}, and each Anderson acceleration version requires 21 iterations to converge, independent of the orthogonalization subroutine used. The results of these tests are presented in Figure 6.8 and are consistent with the results of the weak-scaling study since no communication is required for $G(\mathbf{u})$. The function evaluation performance remains the same, independent of GPU count or Anderson acceleration variant (differing only slightly for runs on a single node with 4 GPUs). In addition, the time spent in $G(\mathbf{u})$ is lower for this example since $G(\mathbf{u})$ performance depends on the number of samples in the mixture, while the performance

of Anderson acceleration scales with the number of values in the global vector.

Results are consistent with observations in the GPU weak-scaling study in Section 6.4.2 where all of the `QRAdd` subroutines performed similar for $m = 5$. Additionally, because m is small, we do not incur a large overhead for data movement and multiple kernel launches in comparison to those observed with larger values of m, as presented in Section 6.4.2.2. Because the majority of the computation has moved to the GPU, we observe the expected cross-over point for ICWY with its performance being slightly faster than that of MGS for GPU counts of 16 and larger. There are no consistent improvements observed for CGS-2 or DCGS-2.

6.5 Performance Considerations for Alternating Anderson Acceleration and Composite Anderson Acceleration

As shown in Section 6.4.2, low synchronization orthogonalization routines drive down the performance cost of Anderson acceleration. However, as noted in Section 6.4.3, there are often cases where the function evaluation dominates performance costs. In this section, we explore the use of recently developed Anderson acceleration variants, alternating Anderson acceleration (Algorithm 6.5) and composite Anderson acceleration (Algorithm 6.6) to balance the cost of orthogonalization in Anderson acceleration with costly function evaluations.

First, we provide convergence comparisons for each of the newly developed Anderson acceleration variants for each of the test problems presented in Section 6.4.3, then we provide a performance study on the impact of low synchronization orthogonalization routines in composite Anderson acceleration in Section 6.5.2.

6.5.1 Convergence of Anderson Acceleration Variants

This section includes convergence results for the newest Anderson acceleration variants for each of the test problems presented in Section 6.4.3. Namely, we show the convergence history for each problem using standard Anderson acceleration, Alternating Anderson acceleration, and various Composite Anderson acceleration configurations. Accompanying each of the convergence plots is a table depicting the total number of function evaluations performed to attain convergence, as the convergence plots alone do not present the total number of function evaluations for Composite Anderson acceleration.

All convergence results were collected from parallel runs of the test cases, distributed across two nodes of the Lassen supercomputer.

6.5.1.1 Anisotropic 2D Heat Equation + Nonlinear Term

Nonlinear Term 1 Convergence results for the Heat 2D + Nonlinear Term 1 test case outlined in Section 6.4.3.1 are given in Figure 6.9, with number of iterations to convergence beside function

evaluations in Table 6.1. The values beside each solver in the legend ((m, m_N)) corresponds to the subspace sizes for outer and inner iterations (in the case of C-AA) for the various Anderson acceleration variants. While C-AA shows accelerated convergence, it comes at the cost of a few additional function evaluations, as seen in Table 6.1.

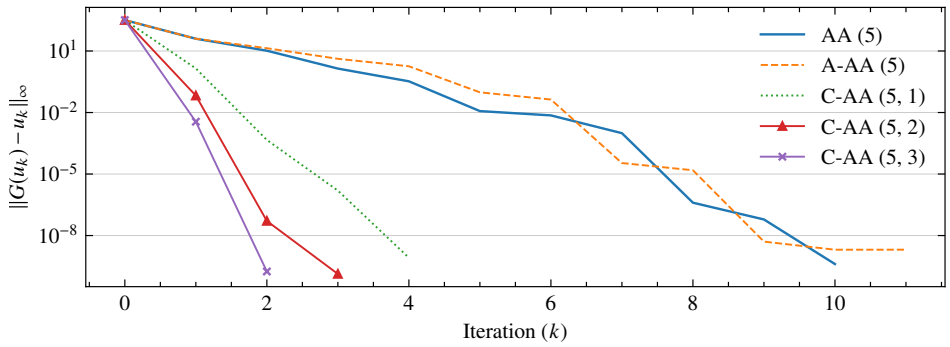


Figure 6.9: Convergence for **Heat 2D + Nonlinear Term 1**, using various AA variants with $m = 5$ and $m_N \in \{1, 2, 3\}$ for C-AA.

Test	Iterations	Function Evaluations
AA (5)	11	11
Alt AA (5)	6	12
Comp AA (5, 1)	5	15
Comp AA (5, 2)	4	16
Comp AA (5, 3)	3	15

Table 6.1: Iterations to convergence and FP function evaluations for **2D Heat + Nonlinear Term 1** test cases.

Nonlinear Term 2 Figure 6.10 shows the convergence for various Anderson acceleration configurations for the Heat 2D + Nonlinear Term 2 test case presented in Section 6.4.3.1. Both C-AA (10, 3) and C-AA (5, 3) exhibit the best convergence, but they also require extremely high numbers of function evaluations. Running with standalone Anderson acceleration results in the fewest overall function evaluations at 44, as seen in Table 6.2.

6.5.1.2 2D Bratu Problem

Figure 6.7 shows the convergence results for the 2D Bratu test case outlined in Section 6.4.3.2. Increasing the inner iterations count of C-AA drives down the required iterations for convergence, while alternating iterations of fixed point and Anderson acceleration sees convergence deterioration (also seen in Figure 6.10). Once again, the number of function evaluations required to converge for the newer Anderson acceleration variants is higher than standalone Anderson acceleration (see Table 6.3) though by a much smaller degree than in the Heat 2D + Nonlinear Term 2 case.

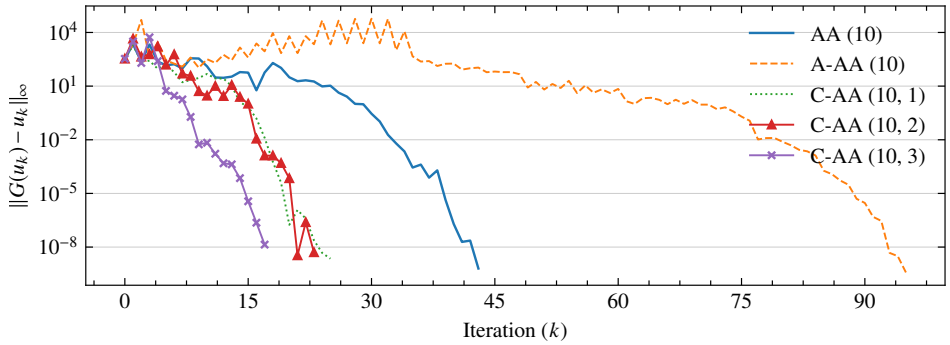


Figure 6.10: Convergence for **Heat 2D + Nonlinear Term 2**, using various AA variants with $m = 5$ and $m_N \in \{1, 2, 3\}$ for C-AA.

Test	Iterations	Function Evaluations
AA (10)	44	44
Alt AA (10)	96	96
Comp AA (10, 1)	26	78
Comp AA (10, 2)	24	96
Comp AA (10, 3)	18	90

Table 6.2: Iterations to convergence and FP function evaluations for **2D Heat + Nonlinear Term 2** test cases.

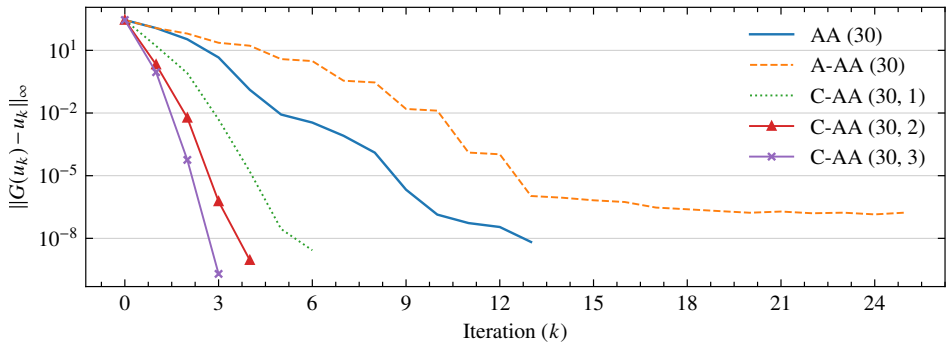


Figure 6.11: Convergence for **2D Bratu**, using various AA variants with $m = 30$ and $m_N \in \{1, 2, 3\}$ for C-AA.

6.5.1.3 Expectation-Maximization Algorithm for Mixture Densities

Convergence results for the Expectation-Maximization test case detailed in Section 6.4.3.3 are presented in Figure 6.8 for multiple Anderson acceleration configurations. Alternating Anderson acceleration converges more slowly than any other configuration. The composite Anderson acceleration configurations with $m_N = 2$ or $m_N = 3$ converged most quickly, however this came at the cost of many more function evaluations. Table 6.4 shows the number of iterations compared against the number of function evaluations performed for each Anderson acceleration variant, and

Test	Iterations	Function Evaluations
AA (30)	14	14
Alt AA (30)	26	26
Comp AA (30, 1)	7	21
Comp AA (30, 2)	5	20
Comp AA (30, 3)	4	20

Table 6.3: Iterations to convergence and FP function evaluations for **2D Bratu** test cases.

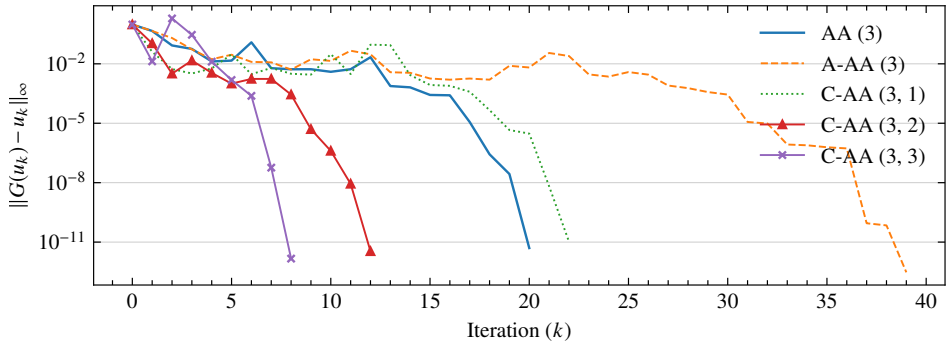


Figure 6.12: Convergence for **Expectation-Maximization**, using various AA variants with $m = 3$ and $m_N \in \{1, 2, 3\}$ for C-AA.

Test	Iterations	Function Evaluations
AA (3)	21	21
Alt AA (3)	40	40
Comp AA (3, 1)	23	69
Comp AA (3, 2)	13	52
Comp AA (3, 3)	9	45

Table 6.4: Iterations to convergence and FP function evaluations for **Expectation-Maximization** test cases.

this example exhibits similar increases in the number of function evaluations required to converge for composite Anderson acceleration as seen for the other example problems Tables 6.1 to 6.3.

6.5.2 Performance of ICWY-MGS in Composite Anderson Acceleration

In Sections 6.4.2 and 6.4.3, ICWY-MGS demonstrated the best performance for large values of m for the CPU-based experiments and all values of m in the GPU-based experiments. Additionally, MGS showed the best performance for small values of m in the CPU-based studies. While Composite Anderson acceleration did not reduce the number of fixed point function evaluations compared to Anderson acceleration for the presented test cases in Section 6.5.1, composite Anderson acceleration presents an opportunity in achieving optimal parallel performance of the

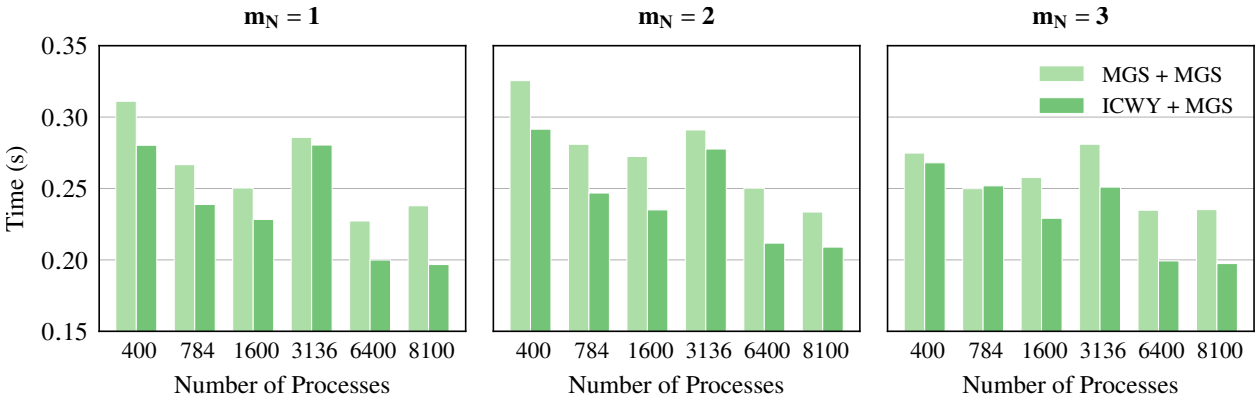


Figure 6.13: Total time to convergence for **Heat 2D + Nonlinear Term 1** using C-AA with $m = 5$ and varying m_N, including the function evaluation, $G(\mathbf{u})$.

method by utilizing different orthogonalization strategies for the inner and outer iterations.

In this section, we perform the same strong-scaling and weak-scaling performance tests done in Section 6.4.3, but instead of comparing the time to convergence of Anderson acceleration with various orthogonalization techniques, we look at the performance of Composite Anderson acceleration using combinations of MGS and ICWY-MGS for the inner and outer iterations. We use the same testing setup as in Section 6.4.2; namely, all experiments are performed on the LLNL Lassen supercomputer, and each run is executed 10 times. Average times are reported.

6.5.2.1 Anisotropic 2D Heat Equation + Nonlinear Term

For the Heat 2D + Nonlinear Term test cases, we consider the usage of MGS and ICWY-MGS for the outer iterations of Composite Anderson acceleration and only consider MGS for the inner iterations as m_N is kept small.

Nonlinear Term 1 Figure 6.13 presents the parallel performance for convergence of the Heat2D + Nonlinear Term 1 example when using composite Anderson acceleration with $m = 5$ and $m_N \in \{1, 2, 3\}$. Both MGS for the outer and inner iterations (MGS + MGS) is compared against the usage of ICWY-MGS for the outer iterations with MGS for the inner iterations (ICWY+MGS). There is a notable performance benefit to utilizing ICWY-MGS as the orthogonalization routine for the outer iterations compared to MGS. This is in stark contract to Figure 6.5, where the low synchronization orthogonaliztion routines barely showed a difference in the amount of time required to converge for this test case with standard Anderson acceleration.

Recall from Section 6.3.2 that an additional synchronization is gained when calling `QRDelete` with the ICWY-MGS orthogonalization method because of updating the projector correction matrix T. For convergence of the standard Anderson acceleration method, `QRDelete` was called four times, as the number of iterations to convergence was 11 with $m = 5$, resulting in two additional synchronizations. For composite Anderson acceleration, this was reduced to zero, as the

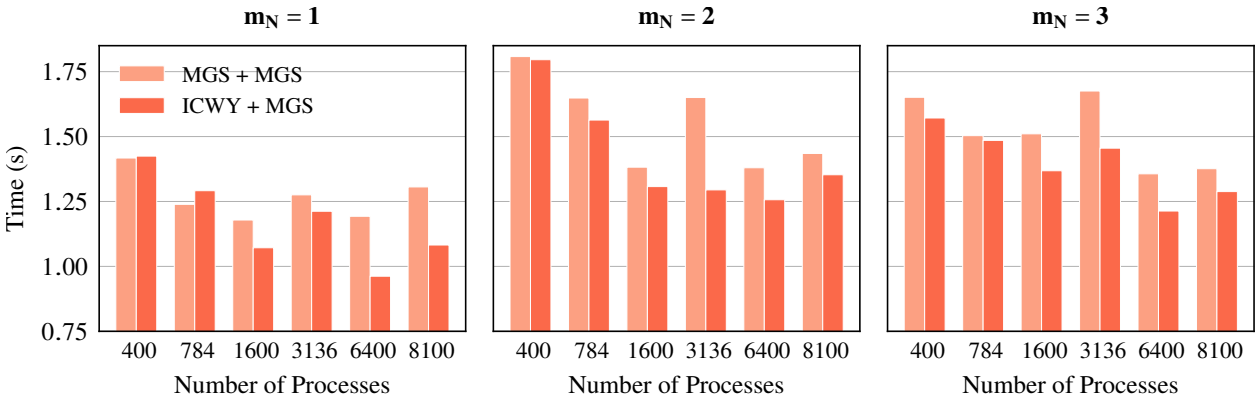


Figure 6.14: Total time to convergence for **Heat 2D + Nonlinear Term 2** using C-AA with $m = 10$ and varying m_N, including the function evaluation, $G(\mathbf{u})$.

number of outer iterations required to converge never required removing a vector from the outer Anderson acceleration subspace.

Nonlinear Term 2 Figure 6.14 presents the parallel performance for convergence of the Heat2D + Nonlinear Term 2 example when using composite Anderson acceleration with $m = 10$ and $m_N \in \{1, 2, 3\}$. Both MGS for the outer and inner iterations (MGS + MGS) is compared against the usage of ICWY-MGS for the outer iterations with MGS for the inner iterations (ICWY+MGS). While ICWY+MGS still demonstrates better performance than MGS+MGS, the performance beneifts are mostly noticeable at the larger scale runs (3136 – 8100 processes).

Unlike in the Heat 2D + Nonlinear Term 1 case, the number of `QRDelete` calls does not reduce to zero for composite Anderson acceleration. For standard Anderson acceleration, convergence required four additional synchronizations for `QRDelete` calls, but for composite Anderson acceleration, this reduces to two for C-AA (10, 1) and C-AA (10, 2) and one for C-AA (10, 3).

6.5.2.2 2D Bratu Problem

In Figure 6.15, we present the parallel performance for convergence of the 2D Bratu example problem when using composite Anderson acceleration with $m = 30$ and $m_N \in \{1, 2, 3\}$. Both MGS for the outer and inner iterations (MGS + MGS) is compared against the usage of ICWY-MGS for the outer iterations with MGS for the inner iterations (ICWY+MGS). In this case, `QRDelete` is never called, and the performance benefits of using ICWY-MGS for the outer iterations is very slight.

6.5.2.3 Expectation-Maximization Algorithm for Mixture Densities

Figure 6.16 presents the weak-scaling parallel performance for convergence of the Expectation-Maximization example when using composite Anderson acceleration with $m = 3$ and

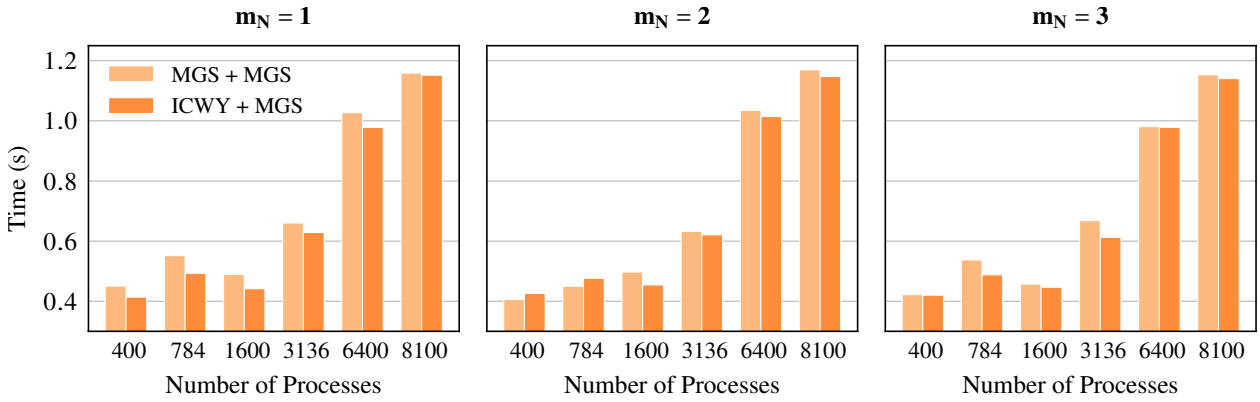


Figure 6.15: Total time to convergence for **2D Bratu**, using C-AA with $m = 30$ and varying m_N, including the function evaluation, $G(\mathbf{u})$

$m_N \in \{1, 2, 3\}$. Seeing as ICWY-MGS demonstrated the best performance in the GPU-based performance studies, we test both MGS and ICWY-MGS for both the outer and inner iterations. While performance improvements are slight, ICWY+ICWY is the most performant in almost all of the cases, further demonstrating the performance benefit of fewer CUDA kernel launches, as discussed in Section 6.4.2.

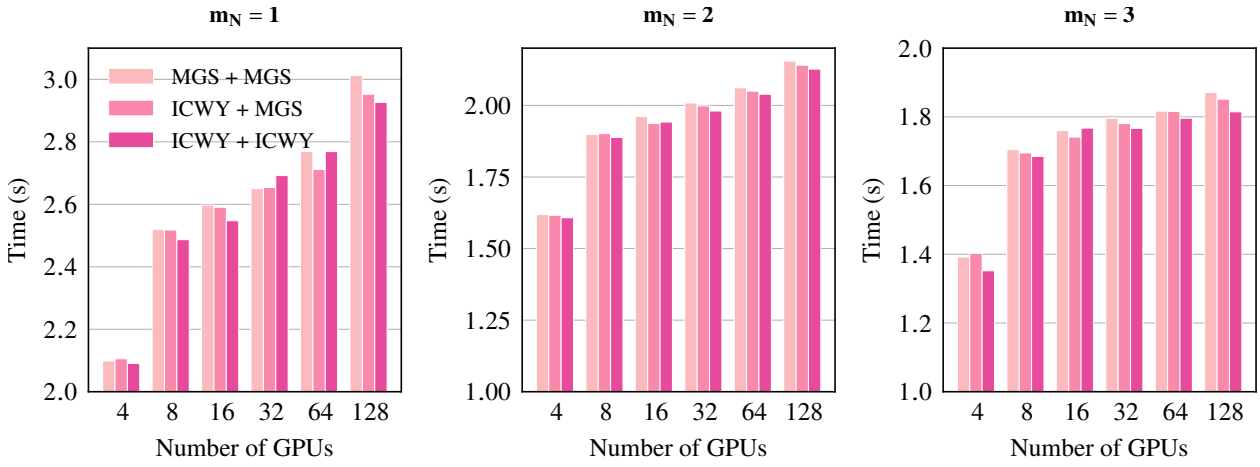


Figure 6.16: Total time to convergence for **Expectation-Maximization**, using C-AA with $m = 3$ and varying m_N with a local vector size of 1 500 000 for each GPU, including the function evaluation, $G(\mathbf{u})$.

Additionally, the number of `QRDelete` calls has been reduced over that of standard Anderson acceleration. For standard Anderson acceleration, there were six additional synchronizations introduced, while for C-AA (3, 2) and C-AA (3, 1) this is reduced to four and two, respectively. Overall, ICWY-MGS demonstrates the best performance for GPU-based Anderson acceleration within SUNDIALS.

6.5.3 Discussion

Achieving optimal parallel performance of Anderson acceleration is a complicated task. As is demonstrated in Section 6.4.3, the number of fixed point function evaluations is a large factor in overall performance, hence it is imperative to balance these costly function evaluations with optimal convergence properties, as is shown in Section 6.5.1. Unfortunately, the Anderson acceleration variant that results in the most optimal convergence properties is not necessarily the most performant. However, while alternating Anderson acceleration and composite Anderson acceleration do not result in a reduction in the number of function evaluations required for convergence for our example problems, this will not always be the case. Importantly, low synchronization orthogonalization routines demonstrate even greater performance benefits when utilized in composite Anderson acceleration.

Low synchronization orthogonalization routines are more performant than MGS for large m and composite Anderson acceleration presents an opportunity to pair them with MGS if either m or m_N is large. Additionally, composite Anderson acceleration can potentially result in fewer calls to `QRDelete` depending on the values of m and m_N. `QRDelete` is never called for the inner iterations of composite Anderson acceleration, hence for a large m in the outer iterations, the reduction in `QRDelete` calls will potentially be higher. While we focused on the resulting synchronization reduction, eliminating `QRDelete` calls also reduces the flops required to converge. Overall, composite Anderson acceleration presents a unique opportunity to achieve optimal performance for Anderson acceleration at scale when paired with low synchronization orthogonalization routines.

6.6 Conclusions

Anderson Acceleration is an efficient method for accelerating the convergence of fixed point solvers, but faces performance challenges in parallel distributed computing environments mainly due to the number of global synchronizations per iteration which is dependent upon the size of the Anderson acceleration iteration space. In this chapter, we introduced low synchronization orthogonalization subroutines into Anderson acceleration which effectively reduce the number of global synchronizations to a constant number per iteration independent of the size of the Anderson acceleration iteration space.

We presented a performance study that demonstrated the improved strong-scalability of Anderson acceleration with these low synchronization `QRAdd` subroutines when performed in a CPU-only parallel environment, as well as demonstrated performance and implementation concerns for these subroutines when operating in a multi-GPU computing environment. Furthermore, our numerical results display a realistic picture of the expected performance of Anderson acceleration in practice that matches the predictions of our performance analysis in Section 6.4.2 and suggests the use of ICWY-MGS for large values of m when operating in a CPU-only parallel environment and as the default method for distributed GPU computing.

Additionally, we provided convergence comparisons and a discussion on performance considerations for recently developed Anderson acceleration variants, alternating Anderson acceleration and composite Anderson acceleration. We demonstrated the importance of ICWY-MGS in achieving optimal performance for composite Anderson acceleration at large scales. Overall, this chapter provides a comprehensive study of low synchronization orthogonalization routines within Anderson acceleration and their parallel performance benefits.

The software used to generate the results in this chapter is available in SUNDIALS v6.0.0.

Chapter 7: Conclusions

7.1 Summary

This dissertation includes a number of strategies for reducing the cost of communication in parallel iterative solvers at scale. Specifically, performance bottlenecks attributed to irregular point-to-point communication and global reductions are addressed through the usage of message passage restructuring and algorithm redesign strategies. The message passage restructuring schemes presented within the dissertation are based on node-aware communication strategies for inter-node communication which exchange costly data flow paths for cheaper paths based on node topology. In Chapter 3, a number of contributions based on node-aware communication strategies are made in the development of a novel communication strategy, including the following:

- Development of a novel node-aware communication strategy for inter-node communication, Split, that takes into consideration the total communicated data volume between nodes.
- Development of performance models suggesting the usage of staged-through-host Split communication for inter-GPU communication on emerging heterogeneous arcitectures.
- Introduction of node-aware communication strategies to inter-node communication on heterogeneous architectures.

Chapter 4 addresses the communication bottlenecks of the enlarged conjugate gradient (ECG) method associated with irregular point-to-point communication and utilizes the novel node-aware communication strategy Split to effectively remove this bottleneck in the per iteration performance costs. Contributions of this chapter include:

- Implementation of a communication efficient enlarged Krylov method based on ECG.
- Performance study and analysis of the effects of block vectors on the balance of collective communication, point-to-point communication, and computation within ECG, noting the SpMBV as the main bottleneck.
- Introduction of Split communication into the SpMBV kernel, demonstrating 60x speedup for the kernel on two different distributed systems, effectively reducing the SpMBV kernel as the main bottleneck of the algorithm at large scales.

In Chapter 5, a novel communication scheme based on the Split communication strategy is designed to address the irregular point-to-point communication of the unstructured-mesh boundary exchanges within *MIRGE-Com*, the simulation framework for the Center for Exascale-enabled Scramjet Design. Contributions made within this chapter include:

- Performance model-based analysis of the expected communication for large-scale simulation runs of the *MIRGE-Com* framework which analyzed the expected performance of

communication for the various flux quantities based on the varying message sizes and data volume for each.

- Introduction of an efficient communication strategy for boundary exchanges in *MIRGE-Com* based on Split node-aware communication that utilizes all available CPU cores instead of a single host rank per GPU to perform communication, effectively collapsing the cost of communication.

Finally, Chapter 6 employs algorithm redesign strategies to reduce communication bottlenecks in Anderson accelerated fixed point solvers associated with the high synchronization costs of modified Gram Schmidt (MGS) orthogonalization. Low synchronization orthogonalization routines are introduced into the Anderson acceleration solver within the SUNDIALs library and extensive performance testing is performed, highlighting the importance of algorithmic redesign in reducing communication costs of large-scale codes. The contributions made within this chapter include the following:

- Introduction of low synchronization orthogonalization routines into Anderson acceleration, effectively reducing the number of global synchronizations to a constant per iteration independent of the size of the Anderson acceleration iteration space.

- Performance study of the low synchronization orthogonalization routines that demonstrated the improved strong-scalability of Anderson acceleration with low synchronization orthogonalization routines for CPU-based distributed computing and improved performance for multi-GPU distributed computing.

- Numerical experiments displaying the performance of Anderson acceleration with low synchronization orthogonalization routines in application settings, demonstrating ICWY-MGS as the most performant orthogonalization strategy for both CPU-based and GPU-based settings, and highlighting the cost of function evaluations in determining the overall performance of Anderson acceleration at large scales.

- Performance of a convergence analysis of recent Anderson acceleration variants, alternating Anderson acceleration and composite Anderson acceleration, highlighting the importance of the performed number of function evaluations in evaluating which Anderson acceleration scheme to choose.

- Introduction of ICWY-MGS into composite Anderson acceleration based on the results of the Anderson acceleration performance study, demonstrating the importance of ICWY-MGS in achieving optimal performance for composite Anderson acceleration at large scales.

In summary, this dissertation contributes multiple strategies for reducing communication in parallel iterative solvers. Message passage restructuring strategies reduce the cost of communication in various settings including sparse matrix operations and non-data redundant communications

such as large-scale unstructured-mesh boundary exchanges. These strategies exhibit performance improvements for both inter-CPU and inter-GPU communication on heterogeneous architectures. Additionally, algorithmic redesign strategies for communication reduction are highlighted within the inclusion of low synchronization orthogonalization routines in Anderson accelerated fixed point solvers. Performance of the low synchronization orthogonalization routines within Anderson acceleration demonstrates the importance of algorithm redesign in extending strong-scalability of iterative solvers and highlights the necessity of redesigning algorithms to achieve optimal performance on emerging supercomputer architectures.

7.2 Future Directions

As demonstrated throughout this dissertation, designing communication efficient iterative solvers requires the coupling of efficient parallel implementations and architecture-motivated algorithm design. While the message passage restructuring work performed within this dissertation focused on irregular point-to-point communication, this work is easily extended to collective communications, as seen in [126, 127]. Moving forward, it is straightforward to design fully node-aware iterative solvers that utilize node-aware communication strategies alongside redesigned algorithms, such as low synchronization orthogonalization routines, effectively reducing the cost of communication to a minimum.

Additionally, the performance models developed within the dissertation demonstrated effectiveness at determining the optimal communication strategy for a given problem when the parallel partitioning is known. While the sparse-matrix operations and unstructured mesh boundary exchanges tested required a single setup phase that allowed for tuning between the communication strategies, there are scenarios when this is not the case. For example, adapative time-stepping schemes often rely on repartitioning at each time step [128, 129, 130]. In this scenario, tuning between multiple communication strategies could create an overhead for each individual time step, but choosing a communication strategy based on the performance model could result in optimal performance without the tuning overhead.

Finally, Anderson acceleration could potentially be applied to nonlinear multigrid methods to reduce communication through iteration reduction. There are multiple ways to apply multigrid techniques to solve systems of nonlinear equations, $A(\mathbf{u}) = \mathbf{f}$, where $\mathbf{u}, \mathbf{f} \in \mathbb{R}^n$. However, the majority of these methods, termed linearization-multigrid methods, do not apply multigrid techniques to the nonlinearity directly [131]. The *full approximation scheme* (FAS) applies multigrid techniques directly to the nonlinearity and solves a coarse-grid problem for the full approximation as opposed to the error, like multigrid methods for solving systems of linear equations [132]. Extensive work has been done on the comparison of convergence and computational cost of linearization-multigrid methods to FAS, consistently painting FAS as the less performant scheme [133, 134, 135, 136, 137]. However, the fact that the exact solution of the fine-grid problem is a fixed point of the FAS iteration ([138]) often goes unused in most studies.

Applying Anderson acceleration to FAS could potentially accelerate convergence of the method, hence driving down communication costs via iteration reduction.

Appendix A: RAPtor Framework

RAPtor [7] is an open-source library providing a general, high performance parallel algebraic multigrid solver. Algebraic multigrid (AMG) is often used as a preconditioner for Krylov methods, hence my contributions to the library include the implementation of Krylov methods, the split node-aware communication strategy, and block-vector operations. The library is written using *C++* and MPI.

All methods within the library are written utilizing core sparse-matrix kernels. Sparse matrices and vectors are partitioned row-wise across processes, with each process' portion of a global matrix split into two local matrices, `on_process` and `off_process`, represented by solid and empty blocks in Figure A.1, respectively.

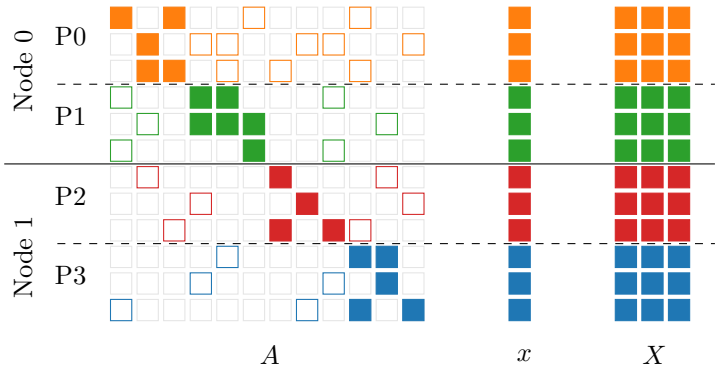


Figure A.1: Row-wise partitioning of matrices and vectors in RAPtor. Matrices are partitioned further into on-process and off-process blocks representing which values of the matrix correspond to pieces of the global vector housed either on-process (■) or off-process (□).

The `on_process` matrix corresponds to locally held values of global vectors and other matrices that are also stored on-rank. The `off_process` matrix corresponds to off-rank values of global vectors and other matrices, i.e. portions of the matrix that correspond to operations which would require MPI communication. Both of these matrices are stored in the same storage format as the global matrix, i.e. for a global CSR matrix, the on-rank `on_process` and `off_process` matrices are stored in CSR format. Block vectors are also partitioned row-wise across processes, but the local vectors are stored column-wise.

Below is a list of the available Krylov methods within RAPtor for solving a system of equations $Ax = b$, and their required inputs. Each of the non-preconditioned methods requires input of a parallel matrix (`ParCSRMatrix`) and two parallel vectors. The vectors `x` and `b` are the initial guess and solution, respectively. Preconditioned variants require the additional input of a RAPtor parallel AMG hierarchy, denoted `ParMultilevel`.

[7] https://github.com/raptor-library/raptor

For the enlarged conjugate gradient methods, the splitting of the vector `x` into a block vector is available through the following functions:

```
x.split_contig(ParVector& X, int t, int first_local).
```

```
x.split(ParVector& X, int t),
```

Here, `X` is the resulting block vector and `t` is the number of columns in the resulting block vector. Figure A.2 depicts the resulting block vector partitioning for each of the given functions when `t` is 3.For `split_contig`, the `first_local` parameter is the index of the column in which the process should store the contiguous chunk of the vector. Additionally, in the case where the number of processes is equal to `t`, `split_contig` is the default method used.

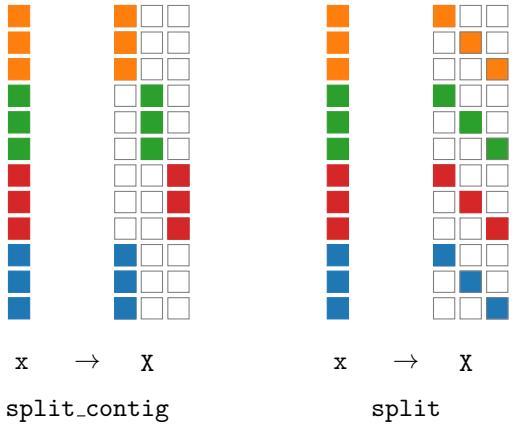


Figure A.2: Block vector splitting in RAPtor.

Conjugate Gradient The conjugate gradient method (with AMG preconditioning in `PCG`) for solving symmetric positive definite systems of equations is accessible through the following function calls.

```
CG(ParCSRMatrix* A, ParVector& x, ParVector& b, ...)
```

```
PCG(ParCSRMatrix* A, ParMultilevel* ml, ParVector& x, ParVector& b, ...)
```

Short Recurrence Enlarged Conjugate Gradient The short recurrence enlarged conjugate gradient method was first introduced in [66]. Similar to the the ECG method discussed in Chapter 4, this method splits the initial vector `x` into a block vector of `t` vectors. However, it retains the block vector of two prior iterations to A-orthonormalize against at each iteration, hence the storage costs and computational costs are higher than that of ECG.

```
SRECG(ParCSRMatrix* A, ParVector& x, ParVector& b, int t, ...)
```

Enlarged Conjugate Gradient Two implementations of ECG exist within RAPtor. `EKCG` contains implementation of the method directly following its implementation in [64]. `EKCG_MinComm` contains implementation of the communication efficient variant discussed in Section 4.3.1 and for which results are shown throughout Chapter 4.

```
EKCG(ParCSRMatrix* A, ParVector& x, ParVector& b, int t, ...)
```

```
EKCG_MinComm(ParCSRMatrix* A, ParVector& x, ParVector& b, int t, ...)
```

Biconjugate Gradient Stabilized For non-symmetric systems of equations, a BiCGStab implementation is provided. Similar to CG, preconditioning is available through AMG and the function call for `Pre_BiCGStab` is given below.

```
BiCGStab(ParCSRMatrix* A, ParVector& x, ParVector& b, ...)
```

```
Pre_BiCGStab(ParCSRMatrix* A, ParVector& x, ParVector& b, ParMultilevel* ml, ...)
```

References

[1] M. Mohiyuddin, M. Hoemmen, J. Demmel, and K. Yelick, "Minimizing communication in sparse matrix solvers," in *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, ser. SC '09. New York, NY, USA: Association for Computing Machinery, 2009. [Online]. Available: https://doi.org/10.1145/1654059.1654096

[2] J. Demmel, L. Grigori, M. Hoemmen, and J. Langou, "Communication-optimal parallel and sequential qr and lu factorizations," *SIAM Journal on Scientific Computing*, vol. 34, no. 1, pp. A206–A239, 2012.

[3] E. Carson, N. Knight, and J. Demmel, "Avoiding communication in nonsymmetric lanczos-based krylov subspace methods," *SIAM J. Sci. Comput.*, vol. 35, no. 5, pp. S42–S61, Jan. 2013. [Online]. Available: https://doi.org/10.1137/120881191

[4] H. M., "Communication-avoiding krylov subspace methods," Ph.D. dissertation, University of California, Berkeley, 2010.

[5] P. Koanantakool, A. Azad, A. Buluç, D. Morozov, S.-Y. Oh, L. Oliker, and K. Yelick, "Communication-avoiding parallel sparse-dense matrix-matrix multiplication," in *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2016, pp. 842–853.

[6] S. Cools and W. Vanroose, "The communication-hiding pipelined BiCGstab method for the parallel solution of large unsymmetric linear systems," *Parallel Comput.*, vol. 65, pp. 1–20, July 2017. [Online]. Available: https://doi.org/10.1016/j.parco.2017.04.005

[7] A. Bienz, W. D. Gropp, and L. N. Olson, "Reducing communication in algebraic multigrid with multi-step node aware communication," *The International Journal of High Performance Computing Applications*, vol. 34, no. 5, pp. 547–561, June 2020. [Online]. Available: https://doi.org/10.1177/1094342020925535

[8] K. Akbudak and C. Aykanat, "Simultaneous input and output matrix partitioning for outer-product–parallel sparse matrix-matrix multiplication," *SIAM Journal on Scientific Computing*, vol. 36, no. 5, pp. C568–C590, 2014.

[9] G. Ballard, A. Druinsky, N. Knight, and O. Schwartz, "Hypergraph partitioning for sparse matrix-matrix multiplication," *ACM Transactions on Parallel Computing (TOPC)*, vol. 3, no. 3, pp. 1–34, 2016.

[10] E. Solomonik and J. Demmel, "Communication-optimal parallel 2.5 d matrix multiplication and lu factorization algorithms," in *European Conference on Parallel Processing*. Springer, 2011, pp. 90–109.

[11] A. Azad, G. Ballard, A. Buluc, J. Demmel, L. Grigori, O. Schwartz, S. Toledo, and S. Williams, "Exploiting multiple levels of parallelism in sparse matrix-matrix multiplication," *SIAM Journal on Scientific Computing*, vol. 38, no. 6, pp. C624–C651, 2016.

[12] A. Lazzaro, J. VandeVondele, J. Hutter, and O. Schütt, "Increasing the efficiency of sparse matrix-matrix multiplication with a 2.5 d algorithm and one-sided mpi," in *Proceedings of the Platform for Advanced Scientific Computing Conference*, 2017, pp. 1–9.

[13] G. Ballard, A. Buluc, J. Demmel, L. Grigori, B. Lipshitz, O. Schwartz, and S. Toledo, "Communication optimal parallel multiplication of sparse random matrices," in *Proceedings of the Twenty-Fifth Annual ACM Symposium on Parallelism in Algorithms and Architectures*, ser. SPAA '13. New York, NY, USA: Association for Computing Machinery, 2013. [Online]. Available: https://doi.org/10.1145/2486159.2486196 p. 222–231.

[14] A. Chronopoulos and C. Gear, "s-step iterative methods for symmetric linear systems," *J. Comput. Appl. Math.*, vol. 25, no. 2, pp. 153–168, Feb. 1989. [Online]. Available: https://doi.org/10.1016/0377-0427(89)90045-9

[15] S. M. Moufawad, "s-step enlarged krylov subspace conjugate gradient methods," *SIAM J. Sci. Comput.*, vol. 42, no. 1, pp. A187–A219, Jan. 2020. [Online]. Available: https://doi.org/10.1137/18m1182528

[16] K. Świrydowicz, J. Langou, S. Ananthan, U. Yang, and S. Thomas, "Low synchronization Gram–Schmidt and generalized minimal residual algorithms," *Numerical Linear Algebra with Applications*, vol. 28, no. 2, p. e2343, 2021.

[17] D. Bielich, J. Langou, S. Thomas, K. Świrydowicz, I. Yamazaki, and E. Boman, "Low-synch Gram-Schmidt with delayed reorthogonalization for Krylov solvers," *Parallel Computing*, 2021.

[18] I. Yamazaki, S. Thomas, M. Hoemmen, E. G. Boman, K. Świrydowicz, and J. J. Elliott, "Low-synchronization orthogonalization schemes for s-step and pipelined Krylov solvers in trilinos," in *Proceedings of the 2020 SIAM Conference on Parallel Processing for Scientific Computing*. SIAM, 2020, pp. 118–128.

[19] S. Thomas, E. Carson, M. Rozložník, A. Carr, and K. Świrydowicz, "Iterated gauss-seidel gmres," 2022. [Online]. Available: https://arxiv.org/abs/2205.07805

[20] W. A. Hanson, "The CORAL supercomputer systems," *IBM Journal of Research and Development*, vol. 64, no. 3/4, pp. 1:1–1:10, 2020.

[21] S. S. Vazhkudai, B. R. de Supinski, A. S. Bland, A. Geist, J. Sexton, J. Kahle, C. J. Zimmer, S. Atchley, S. Oral, D. E. Maxwell, V. G. V. Larrea, A. Bertsch, R. Goldstone, W. Joubert, C. Chambreau, D. Appelhans, R. Blackmore, B. Casses, G. Chochia, G. Davison, M. A. Ezell, T. Gooding, E. Gonsiorowski, L. Grinberg, B. Hanson, B. Hartner, I. Karlin, M. L. Leininger, D. Leverman, C. Marroquin, A. Moody, M. Ohmacht, R. Pankajakshan, F. Pizzano, J. H. Rogers, B. Rosenburg, D. Schmidt, M. Shankar, F. Wang, P. Watson, B. Walkup, L. D. Weems, and J. Yin, "The design, deployment, and evaluation of the CORAL pre-exascale systems," in *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*, 2018, pp. 661–672.

[22] O. R. N. Laboratories, "Frontier Exascale Supercomputer." [Online]. Available: https://www.olcf.ornl.gov/frontier/

[23] S. Potluri, K. Hamidouche, A. Venkatesh, D. Bureddy, and D. K. Panda, "Efficient Inter-node MPI Communication Using GPUDirect RDMA for InfiniBand Clusters with NVIDIA GPUs," in *2013 42nd International Conference on Parallel Processing*, 2013, pp. 80–89.

[24] W. Gropp, L. N. Olson, and P. Samfass, "Modeling MPI Communication Performance on SMP Nodes: Is It Time to Retire the Ping Pong Test," in *Proceedings of the 23rd European MPI Users' Group Meeting*, ser. EuroMPI 2016. New York, NY, USA: ACM, 2016. [Online]. Available: http://doi.acm.org/10.1145/2966884.2966919 pp. 41–50.

[25] D. Culler, R. Karp, D. Patterson, A. Sahay, K. E. Schauser, E. Santos, R. Subramonian, and T. von Eicken, "Logp: Towards a realistic model of parallel computation," *SIGPLAN Not.*, vol. 28, no. 7, p. 1–12, July 1993. [Online]. Available: https://doi.org/10.1145/173284.155333

[26] A. Alexandrov, M. F. Ionescu, K. E. Schauser, and C. Scheiman, "LogGP: Incorporating long messages into the LogP model for parallel computation," *Journal of Parallel and Distributed Computing*, vol. 44, no. 1, pp. 71 – 79, 1997. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0743731597913460

[27] A. Bienz, W. D. Gropp, and L. N. Olson, "Improving Performance Models for Irregular Point-to-Point Communication," in *Proceedings of the 25th European MPI Users' Group Meeting, Barcelona, Spain, September 23-26, 2018*, 2018. [Online]. Available: https://doi.org/10.1145/3236367.3236368 pp. 7:1–7:8.

[28] A. Bienz, L. N. Olson, W. D. Gropp, and S. Lockhart, "Modeling data movement performance on heterogeneous architectures," in *2021 IEEE High Performance Extreme Computing Conference (HPEC)*, 2021, pp. 1–7.

[29] U. V. Çatalyürek, C. Aykanat, and B. Uçar, "On two-dimensional sparse matrix partitioning: Models, methods, and a recipe," *SIAM Journal on Scientific Computing*, vol. 32, no. 2, pp. 656–683, 2010. [Online]. Available: https://doi.org/10.1137/080737770

[30] B. Hendrickson and T. G. Kolda, "Graph partitioning models for parallel computing," *Parallel Computing*, vol. 26, no. 12, pp. 1519 – 1534, 200. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S016781910000048X

[31] B. Vastenhouw and R. H. Bisseling, "A two-dimensional data distribution method for parallel sparse matrix-vector multiplication," *SIAM Review*, vol. 47, no. 1, pp. 67–95, 2005. [Online]. Available: https://doi.org/10.1137/S0036144502409019

[32] U. Catalyurek and C. Aykanat, "Hypergraph-partitioning-based decomposition for parallel sparse-matrix vector multiplication," *IEEE Transactions on Parallel and Distributed Systems*, vol. 10, no. 7, pp. 673–693, 1999.

[33] T. Agarwal, A. Sharma, A. Laxmikant, and L. V. Kalé, "Topology-aware task mapping for reducing communication contention on large parallel machines," in *Proceedings 20th IEEE International Parallel & Distributed Processing Symposium.* IEEE, 2006, pp. 10–pp.

[34] T. Malik, V. Rychkov, and A. Lastovetsky, "Network-aware optimization of communications for parallel matrix multiplication on hierarchical hpc platforms," *Concurrency and Computation: Practice and Experience*, vol. 28, no. 3, pp. 802–821, 2016.

[35] J. L. Träff, "Implementing the mpi process topology mechanism," in *Proceedings of the 2002 ACM/IEEE Conference on Supercomputing*, ser. SC '02. Washington, DC, USA: IEEE Computer Society Press, 2002, p. 1–14.

[36] A. H. Baker, M. Schulz, and U. M. Yang, "On the performance of an algebraic multigrid solver on multicore clusters," in *International Conference on High Performance Computing for Computational Science*. Springer, 2010, pp. 102–115.

[37] B. A. Page and P. M. Kogge, "Scalability of hybrid sparse matrix dense vector (spmv) multiplication," in *2018 International Conference on High Performance Computing & Simulation (HPCS)*. IEEE, 2018, pp. 406–414.

[38] H. M. Aktulga, C. Yang, E. G. Ng, P. Maris, and J. P. Vary, "Improving the scalability of a symmetric iterative eigensolver for multi-core platforms," *Concurrency and Computation: Practice and Experience*, vol. 26, no. 16, pp. 2631–2651, 2014.

[39] A. Bienz, W. Gropp, and L. Olson, "Node aware sparse matrix–vector multiplication," *Journal of Parallel and Distributed Computing*, vol. 130, pp. 166–178, 08 2019.

[40] M. Hidayetoğlu, T. Bicer, S. G. de Gonzalo, B. Ren, V. De Andrade, D. Gursoy, R. Kettimuthu, I. T. Foster, and W.-m. W. Hwu, "Petascale XCT: 3D Image Reconstruction with Hierarchical Communications on Multi-GPU Nodes," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '20. IEEE Press, 2020.

[41] M. T. Heath, *Scientific Computing*. Society for Industrial and Applied Mathematics, 2018. [Online]. Available: https://epubs.siam.org/doi/abs/10.1137/1.9781611975581

[42] C. Bischof and C. Van Loan, "The wy representation for products of householder matrices," *SIAM Journal on Scientific and Statistical Computing*, vol. 8, no. 1, pp. s2–s13, 1987. [Online]. Available: https://doi.org/10.1137/0908009

[43] R. Schreiber and C. Van Loan, "A storage-efficient wy representation for products of householder transformations," *SIAM Journal on Scientific and Statistical Computing*, vol. 10, no. 1, pp. 53–57, 1989. [Online]. Available: https://doi.org/10.1137/0910005

[44] T. Joffrain, T. M. Low, E. S. Quintana-Ortí, R. v. d. Geijn, and F. G. V. Zee, "Accumulating householder transformations, revisited," *ACM Trans. Math. Softw.*, vol. 32, no. 2, p. 169–179, jun 2006. [Online]. Available: https://doi.org/10.1145/1141885.1141886

[45] C. Puglisi, "Modification of the householder method based on the compact wy representation," *SIAM Journal on Scientific and Statistical Computing*, vol. 13, no. 3, pp. 723–726, 1992. [Online]. Available: https://doi.org/10.1137/0913042

[46] J. Demmel, L. Grigori, M. Hoemmen, and J. Langou, "Communication-optimal parallel and sequential qr and lu factorizations," *SIAM Journal on Scientific Computing*, vol. 34, no. 1, pp. A206–A239, 2012. [Online]. Available: https://doi.org/10.1137/080731992

[47] L. Giraud, J. Langou, and M. Rozloznik, "The loss of orthogonality in the Gram-Schmidt orthogonalization process," *Computers & Mathematics with Applications*, vol. 50, no. 7, pp. 1069–1075, 2005, numerical Methods and Computational Mechanics.

[48] L. Giraud, J. Langou, M. Rozložník, and J. van den Eshof, "Rounding error analysis of the classical Gram-Schmidt orthogonalization process," *Numer. Math.*, vol. 101, no. 1, p. 87–100, July 2005.

[49] Å. Björck, "Solving linear least squares problems by Gram-Schmidt orthogonalization," *BIT Numerical Mathematics*, vol. 7, no. 1, pp. 1–21, 1967.

[50] V. Frayssé, L. Giraud, and H. Kharraz-Aroussi, "On the influence of the orthogonalization scheme on the parallel performance of GMRES," in *European Conference on Parallel Processing*. Springer, 1998, pp. 751–762.

[51] A. Ruhe, "Numerical aspects of Gram-Schmidt orthogonalization of vectors," *Linear algebra and its applications*, vol. 52, pp. 591–601, 1983.

[52] B. N. Parlett, *The symmetric eigenvalue problem*. SIAM, 1998.

[53] N. N. Abdelmalek, "Round off error analysis for Gram-Schmidt method and solution of linear least squares problems," *BIT Numerical Mathematics*, vol. 11, no. 4, pp. 345–367, 1971.

[54] J. W. Daniel, W. B. Gragg, L. Kaufman, and G. W. Stewart, "Reorthogonalization and stable algorithms for updating the Gram-Schmidt QR factorization," *Mathematics of Computation*, vol. 30, no. 136, pp. 772–795, 1976.

[55] W. Hoffmann, "Iterative algorithms for Gram-Schmidt orthogonalization," *Computing*, vol. 41, no. 4, pp. 335–348, 1989.

[56] V. Hernández, J. E. Román, and A. Tomás, "A parallel variant of the Gram-Schmidt process with reorthogonalization," in *PARCO*, 2005, pp. 221–228.

[57] S. Lockhart, A. Bienz, W. Gropp, and L. Olson, "Characterizing the performance of node-aware strategies for irregular point-to-point communication on heterogeneous architectures," *Parallel Computing*, Apr 2023, accepted. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0167819123000273

[58] S. Lockhart, A. Bienz, W. Gropp, and L. Olson, "Performance analysis and optimal node-aware communication for enlarged conjugate gradient methods," *ACM Trans. Parallel Comput.*, vol. 10, no. 1, Mar 2023. [Online]. Available: https://doi.org/10.1145/3580003

[59] B. Bode, M. Butler, T. Dunning, T. Hoefler, W. Kramer, W. Gropp, and W.-m. Hwu, "The Blue Waters super-system for super-science," in *Contemporary High Performance Computing*, ser. Chapman & Hall/CRC Computational Science. Chapman and Hall/CRC, April 2013, pp. 339–366. [Online]. Available: https://www.taylorfrancis.com/books/e/9781466568358

[60] W. Kramer, M. Butler, G. Bauer, K. Chadalavada, and C. Mendes, "Blue Waters Parallel I/O Storage Sub-system," in *High Performance Parallel I/O*, Prabhat and Q. Koziol, Eds. CRC Publications, Taylor and Francis Group, 2015, pp. 17–32.

[61] C. W. Smith, M. Rasquin, D. Ibanez, K. E. Jansen, and M. S. Shephard, "Improving Unstructured Mesh Partitions for Multiple Criteria Using Mesh Adjacencies," *SIAM Journal on Scientific Computing*, vol. 40, no. 1, pp. C47–C75, 2018.

[62] A. Bienz, W. D. Gropp, and L. N. Olson, "Improving performance models for irregular point-to-point communication," *CoRR*, vol. abs/1806.02030, 2018. [Online]. Available: http://arxiv.org/abs/1806.02030

[63] T. A. Davis and Y. Hu, "The university of florida sparse matrix collection," *ACM Trans. Math. Softw.*, vol. 38, no. 1, Dec. 2011. [Online]. Available: https://doi.org/10.1145/2049662.2049663

[64] L. Grigori and O. Tissot, "Scalable linear solvers based on enlarged krylov subspaces with dynamic reduction of search directions," *SIAM J. Sci. Comput.*, vol. 41, no. 5, pp. C522–C547, Jan. 2019. [Online]. Available: https://doi.org/10.1137/18m1196285

[65] D. P. O'Leary, "The block conjugate gradient algorithm and related methods," *Linear Algebra Appl.*, vol. 29, pp. 293–322, Feb. 1980. [Online]. Available: https://doi.org/10.1016/0024-3795(80)90247-5

[66] L. Grigori, S. Moufawad, and F. Nataf, "Enlarged krylov subspace conjugate gradient methods for reducing communication," *SIAM J. Matrix Anal. & Appl.*, vol. 37, no. 2, pp. 744–773, Jan. 2016. [Online]. Available: https://doi.org/10.1137/140989492

[67] P. Ghysels and W. Vanroose, "Hiding global synchronization latency in the preconditioned conjugate gradient algorithm," *Parallel Comput.*, vol. 40, no. 7, pp. 224–238, July 2014. [Online]. Available: https://doi.org/10.1016/j.parco.2013.06.001

[68] L. C. McInnes, B. Smith, H. Zhang, and R. T. Mills, "Hierarchical krylov and nested krylov methods for extreme-scale computing," *Parallel Comput.*, vol. 40, no. 1, pp. 17–31, Jan. 2014. [Online]. Available: https://doi.org/10.1016/j.parco.2013.10.001

[69] R. Anderson, J. Andrej, A. Barker, J. Bramwell, J.-S. Camier, J. Cerveny, V. Dobrev, Y. Dudouit, A. Fisher, T. Kolev, W. Pazner, M. Stowell, V. Tomov, I. Akkerman, J. Dahm, D. Medina, and S. Zampini, "MFEM: A modular finite element methods library," *Computers & Mathematics with Applications*, 2020. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0898122120302583

[70] A. Bienz and L. N. Olson, "RAPtor: parallel algebraic multigrid v0.1," https://github.com/raptor-library/raptor, 2017, release 0.1.

[71] S. Balay, S. Abhyankar, M. Adams, J. Brown, P. Brune, K. Buschelman, L. Dalcin, A. Dener, V. Eijkhout, W. Gropp et al., "Petsc users manual," 2019.

[72] S. Williams, A. Waterman, and D. Patterson, "Roofline: an insightful visual performance model for multicore architectures," *Communications of the ACM*, vol. 52, no. 4, pp. 65–76, 2009.

[73] O. Selvitopi, B. Brock, I. Nisa, A. Tripathy, K. Yelick, and A. Buluç, "Distributed-memory parallel algorithms for sparse times tall-skinny-dense matrix multiplication," in *Proceedings of the ACM International Conference on Supercomputing*, ser. ICS '21. New York, NY, USA: Association for Computing Machinery, 2021. [Online]. Available: https://doi.org/10.1145/3447818.3461472 p. 431–442.

[74] F. Ladeinde, "Advanced computational-fluid-dynamics techniques for scramjet combustion simulation," *Aiaa Journal*, vol. 48, no. 3, pp. 513–514, 2010.

[75] F. Ladeinde, "A critical review of scramjet combustion simulation," in *47th AIAA Aerospace Sciences Meeting Including the New Horizons Forum and Aerospace Exposition*, 2009, p. 127.

[76] R. Pecnik, V. E. Terrapon, F. Ham, G. Iaccarino, and H. Pitsch, "Reynolds-averaged navier-stokes simulations of the hyshot ii scramjet," *AIAA journal*, vol. 50, no. 8, pp. 1717–1732, 2012.

[77] W. D. Gropp, D. K. Kaushik, D. E. Keyes, and B. F. Smith, "High-performance parallel implicit cfd," *Parallel Computing*, vol. 27, no. 4, pp. 337–362, 2001.

[78] Z. Shang, "Impact of mesh partitioning methods in cfd for large scale parallel computing," *Computers & Fluids*, vol. 103, pp. 1–5, 2014.

[79] S. Posey, "Considerations for gpu acceleration of parallel cfd," *Procedia Engineering*, vol. 61, pp. 388–391, 2013.

[80] X. Álvarez-Farré, A. Gorobets, and F. X. Trias, "A hierarchical parallel implementation for heterogeneous computing. application to algebra-based cfd simulations on hybrid supercomputers," *Computers & Fluids*, vol. 214, p. 104768, 2021.

[81] F. Bassi and S. Rebay, "A high-order accurate discontinuous finite element method for the numerical solution of the compressible navier–stokes equations," *Journal of computational physics*, vol. 131, no. 2, pp. 267–279, 1997.

[82] Y. Lv and M. Ihme, "Discontinuous galerkin method for multicomponent chemically reacting flows and combustion," *Journal of Computational Physics*, vol. 270, pp. 105–137, 2014.

[83] A. W. Cook, "Enthalpy diffusion in multicomponent flows," *Physics of Fluids*, vol. 21, no. 5, p. 055109, 2009.

[84] G. Karypis and V. Kumar, "Metis: A software package for partitioning unstructured graphs, partitioning meshes, and computing fill-reducing orderings of sparse matrices," 1997.

[85] A. Klöckner, "Loo.py: transformation-based code generation for GPUs and CPUs," in *Proceedings of ARRAY '14: ACM SIGPLAN Workshop on Libraries, Languages, and Compilers for Array Programming*. Edinburgh, Scotland.: Association for Computing Machinery, 2014.

[86] K. Kulkarni, "Transforming (not just) DG-FEM Array Expressions (not just) on GPUs," 2022, SIAM Conference on Parallel Processing for Scientific Computing. [Online]. Available: https://www.siam.org/Portals/0/Conferences/PP22/PP22_ABSTRACTS.pdf

[87] K. Kulkarni, "Transforming (not just) DG-FEM Array Expressions (not just) on GPUs," 2023, SIAM Conference on Computational Science and Engineering. [Online]. Available: https://meetings.siam.org/sess/dsp_talk.cfm?p=123935

[88] A. Klöckner, N. Pinto, Y. Lee, B. Catanzaro, P. Ivanov, and A. Fasih, "Pycuda and pyopencl: A scripting-based approach to gpu run-time code generation," *Parallel Computing*, vol. 38, no. 3, pp. 157–174, 2012.

[89] P. Jääskeläinen, C. S. de La Lama, E. Schnetter, K. Raiskila, J. Takala, and H. Berg, "pocl: A performance-portable opencl implementation," *International Journal of Parallel Programming*, vol. 43, pp. 752–785, 2015.

[90] K. V. Manian, C.-H. Chu, A. A. Awan, K. S. Khorassani, H. Subramoni, and D. Panda, "Omb-um: Design, implementation, and evaluation of cuda unified memory aware mpi benchmarks," in *2019 IEEE/ACM Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS)*. IEEE, 2019, pp. 82–92.

[91] N. Hanford, R. Pankajakshan, E. A. León, and I. Karlin, "Challenges of gpu-aware communication in mpi," in *2020 Workshop on Exascale MPI (ExaMPI)*, 2020, pp. 1–10.

[92] K. Shafie Khorassani, J. Hashmi, C.-H. Chu, C.-C. Chen, H. Subramoni, and D. K. Panda, "Designing a rocm-aware mpi library for amd gpus: Early experiences," in *High Performance Computing*, B. L. Chamberlain, A.-L. Varbanescu, H. Ltaief, and P. Luszczek, Eds. Springer International Publishing, 2021, vol. 12728, pp. 118–136.

[93] S. Lockhart, D. J. Gardner, C. S. Woodward, S. Thomas, and L. N. Olson, "Performance of low synchronization orthogonalization methods in anderson accelerated fixed point solvers," in *Proceedings of the 2022 SIAM Conference on Parallel Processing for Scientific Computing*. SIAM, 2022, pp. 49–59.

[94] D. G. Anderson, "Iterative procedures for nonlinear integral equations," *JACM*, vol. 12, no. 4, pp. 547–560, 1965.

[95] H. F. Walker and P. Ni, "Anderson acceleration for fixed-point iterations," *SIAM Journal on Numerical Analysis*, vol. 49, no. 4, pp. 1715–1735, 2011.

[96] C. T. Kelley, "Numerical methods for nonlinear equations," *Acta Numerica*, vol. 27, pp. 207–287, 2018.

[97] H. Fang and Y. Saad, "Two classes of multisecant methods for nonlinear acceleration," *Numer. Linear Algebra Appl.*, vol. 16, pp. 197–221, 2009.

[98] Y. Saad and M. H. Schultz, "GMRES: A generalized minimal residual algorithm for solving nonsymmetric linear systems," *SIAM Journal on scientific and statistical computing*, vol. 7, no. 3, pp. 856–869, 1986.

[99] A. C. Hindmarsh, R. Serban, C. J. Balos, D. J. Gardner, D. R. Reynolds, and C. S. Woodward, "User documentation for KINSOL v5. 7.0 (SUNDIALS v5. 7.0)," 2021. [Online]. Available: https://computing.llnl.gov/projects/sundials/kinsol

[100] J. Loffeld and C. S. Woodward, "Considerations on the implementation and use of anderson acceleration on distributed memory and GPU-based parallel computers," in *Advances in the Mathematical Sciences*. Springer, 2016, pp. 417–436.

[101] A. Toth and C. T. Kelley, "Convergence analysis for anderson acceleration," *SIAM Journal on Numerical Analysis*, vol. 53, no. 2, pp. 805–819, 2015.

[102] C. Evans, S. Pollock, L. G. Rebholz, and M. Xiao, "A proof that anderson acceleration improves the convergence rate in linearly converging fixed-point methods (but not in those converging quadratically)," *SIAM Journal on Numerical Analysis*, vol. 58, no. 1, pp. 788–810, 2020.

[103] S. Pollock, L. G. Rebholz, and M. Xiao, "Anderson-accelerated convergence of picard iterations for incompressible navier–stokes equations," *SIAM Journal on Numerical Analysis*, vol. 57, no. 2, pp. 615–637, 2019.

[104] H. D. Sterck and Y. He, "On the asymptotic linear convergence speed of anderson acceleration, nesterov acceleration, and nonlinear gmres," *SIAM Journal on Scientific Computing*, vol. 43, no. 5, pp. S21–S46, 2021.

[105] D. Wang, Y. He, and H. De Sterck, "On the asymptotic linear convergence speed of anderson acceleration applied to admm," *Journal of Scientific Computing*, vol. 88, no. 2, pp. 1–35, 2021.

[106] W. Bian, X. Chen, and C. Kelley, "Anderson acceleration for a class of nonsmooth fixed-point problems," *SIAM Journal on Scientific Computing*, vol. 43, no. 5, pp. S1–S20, 2021.

[107] H. F. Walker, "Anderson acceleration: Algorithms and implementations," *WPI Math. Sciences Dept. Report MS-6-15-50*, 2011.

[108] N. N. Carlson and K. Miller, "Design and application of a gradient-weighted moving finite element code i: in one dimension," *SIAM Journal on Scientific Computing*, vol. 19, no. 3, pp. 728–765, 1998.

[109] K. Miller, "Nonlinear krylov and moving nodes in the method of lines," *Journal of computational and applied mathematics*, vol. 183, no. 2, pp. 275–287, 2005.

[110] C. W. Oosterlee and T. Washio, "Krylov subspace acceleration of nonlinear multigrid with application to recirculating flows," *SIAM Journal on Scientific Computing*, vol. 21, no. 5, pp. 1670–1690, 2000.

[111] T. Washio and C. W. Oosterlee, "Krylov subspace acceleration for nonlinear multigrid schemes," *Electron. Trans. Numer. Anal*, vol. 6, no. 271-290, p. 3, 1997.

[112] L. Lin and C. Yang, "Elliptic preconditioner for accelerating the self-consistent field iteration in kohn–sham density functional theory," *SIAM Journal on Scientific Computing*, vol. 35, no. 5, pp. S277–S298, 2013.

[113] P. Pulay, "Convergence acceleration of iterative sequences. the case of scf iteration," *Chemical Physics Letters*, vol. 73, no. 2, pp. 393–398, 1980.

[114] P. Pulay, "Improved scf convergence acceleration," *Journal of Computational Chemistry*, vol. 3, no. 4, pp. 556–560, 1982.

[115] R. Glowinski, H. B. Keller, and L. Reinhart, "Continuation-conjugate gradient methods for the least squares solution of nonlinear boundary value problems," *SIAM journal on scientific and statistical computing*, vol. 6, no. 4, pp. 793–832, 1985.

[116] D. G. Anderson, "Comments on "anderson acceleration, mixing and extrapolation"," *Numerical Algorithms*, vol. 80, no. 1, pp. 135–234, 2019.

[117] K. Chen and C. Vuik, "Non-stationary anderson acceleration with optimized damping," *arXiv preprint arXiv:2202.05295*, 2022.

[118] S. Pollock and L. G. Rebholz, "Anderson acceleration for contractive and noncontractive operators," *IMA Journal of Numerical Analysis*, vol. 41, no. 4, pp. 2841–2872, 2021.

[119] P. P. Pratapa, P. Suryanarayana, and J. E. Pask, "Anderson acceleration of the jacobi iterative method: An efficient alternative to krylov methods for large, sparse linear systems," *Journal of Computational Physics*, vol. 306, pp. 43–54, 2016.

[120] P. Suryanarayana, P. P. Pratapa, and J. E. Pask, "Alternating anderson–richardson method: An efficient alternative to preconditioned krylov methods for large, sparse linear systems," *Computer Physics Communications*, vol. 234, pp. 278–285, 2019.

[121] K. Chen and C. Vuik, "Composite anderson acceleration method with two window sizes and optimized damping," *International Journal for Numerical Methods in Engineering*, 2022.

[122] A. C. Hindmarsh, P. N. Brown, K. E. Grant, S. L. Lee, R. Serban, D. E. Shumaker, and C. S. Woodward, "SUNDIALS: Suite of nonlinear and differential/algebraic equation solvers," *ACM Trans. Math. Softw.*, vol. 31, no. 3, p. 363–396, Sep. 2005.

[123] C. J. Balos, D. J. Gardner, C. S. Woodward, and D. R. Reynolds, "Enabling GPU accelerated computing in the sundials time integration library," *Parallel Computing*, p. 102836, 2021.

[124] D. A. Frank-Kamenetskii, *Diffusion and Heat Exchange in Chemical Kinetics*. Princeton University Press, 2016.

[125] A. Mohsen, "A simple solution of the Bratu problem," *Computers & Mathematics with Applications*, vol. 67, no. 1, pp. 26–33, 2014.

[126] A. Bienz, L. Olson, and W. Gropp, "Node-Aware Improvements to Allreduce," in *Proceedings of ExaMPI 2019*, ser. Proceedings of ExaMPI 2019: Workshop on Exascale MPI - Held in conjunction with SC 2019: The International Conference for High Performance Computing, Networking, Storage and Analysis. United States: Institute of Electrical and Electronics Engineers Inc., nov 2019, 2019 IEEE/ACM Workshop on Exascale MPI, ExaMPI 2019 ; Conference date: 17-11-2019. pp. 19–28.

[127] A. Bienz, S. Gautam, and A. Kharel, "A locality-aware bruck allgather," in *Proceedings of the 29th European MPI Users' Group Meeting*, ser. EuroMPI/USA'22. New York, NY, USA: Association for Computing Machinery, 2022. [Online]. Available: https://doi.org/10.1145/3555819.3555825 p. 18–26.

[128] B. C. Vermeire and S. Nadarajah, "Adaptive imex time-stepping for the correction procedure via reconstruction scheme," in *21st AIAA Computational Fluid Dynamics Conference*, 2013, p. 2687.

[129] T. Coupez and E. Hachem, "Solution of high-reynolds incompressible flow with stabilized finite element and adaptive anisotropic meshing," *Computer methods in applied mechanics and engineering*, vol. 267, pp. 65–85, 2013.

[130] L. Wang, M. K. Gobbert, and M. Yu, "A dynamically load-balanced parallel p-adaptive implicit high-order flux reconstruction method for under-resolved turbulence simulation," *Journal of Computational Physics*, vol. 417, p. 109581, 2020.

[131] V. Henson, "Multigrid methods nonlinear problems: an overview," in *Proceedings of SPIE*, vol. 5016, 2003, pp. 36–48.

[132] A. Brandt, "Multi-level adaptive solutions to boundary-value problems," *Mathematics of computation*, vol. 31, no. 138, pp. 333–390, 1977.

[133] K. J. Brabazon, M. E. Hubbard, and P. K. Jimack, "Nonlinear multigrid methods for second order differential operators with nonlinear diffusion coefficient," *Computers & Mathematics with Applications*, vol. 68, no. 12, pp. 1619–1634, 2014.

[134] W. Hackbusch, "Comparison of different multi-grid variants for nonlinear equations," *ZAMM-Journal of Applied Mathematics and Mechanics/Zeitschrift für Angewandte Mathematik und Mechanik*, vol. 72, no. 2, pp. 148–151, 1992.

[135] D. J. Mavriplis, "Multigrid approaches to non-linear diffusion problems on unstructured meshes," *Numerical linear algebra with applications*, vol. 8, no. 8, pp. 499–512, 2001.

[136] D. J. Mavriplis, "An assessment of linear versus nonlinear multigrid methods for unstructured mesh solvers," *Journal of Computational Physics*, vol. 175, no. 1, pp. 302–325, 2002.

[137] L. Stals, "Comparison of non-linear solvers for the solution of radiation transport equations," *Electronic Transactions on Numerical Analysis*, vol. 15, pp. 78–93, 2003.

[138] W. L. Briggs, V. E. Henson, and S. F. McCormick, *A multigrid tutorial*. SIAM, 2000.