REDUCING COMMUNICATION IN SPARSE SOLVERS

BY

AMANDA BIENZ

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 at Urbana-Champaign, 2018

Urbana, Illinois

Doctoral Committee:

Professor Luke N. Olson, Chair and Director of Research
Professor William D. Gropp
Assistant Professor Edgar Solomonik
Dr. Laura Grigori, INRIA

ABSTRACT

Sparse matrix operations dominate the cost of many scientific applications. In parallel, the performance and scalability of these operations is limited by irregular point-to-point communication. Multiple methods are investigated throughout this dissertation for reducing the cost associated with communication throughout sparse matrix operations. Algorithmic changes reduce communication requirements, but also affect accuracy of the operation, leading to reduced convergence of scientific codes. We investigate a method of systematically removing relatively small non-zeros throughout an algebraic multigrid hierarchy, yielding significant reductions to the cost of sparse matrix-vector multiplication that outweigh affects of reduced accuracy of the multiplication. Therefore, the reduction in per-iteration communication costs outweigh the cost of extra solver iterations. As a result, sparsification yields improvement of both the performance and scalability of algebraic multigrid.

Alterations to the parallel implementation of MPI communication also yield reduced costs with no effect on accuracy. We investigate methods of agglomerating messages on-node before injecting into the network, reducing the amount of costly inter-node communication. This node-aware communication yields improvements to both performance and scalability of matrix operations, particularly in strong scaling studies. Furthermore, we show an improvement in the cost of algebraic multigrid as a result of reduced communication costs in sparse matrix operations.

Finally, performance models can be used to analyze the costs of matrix operations, indicating the source of dominant communication costs, such as initializing messages or transporting bytes of data. We investigate methods of improving traditional performance models of irregular point-to-point communication through the addition of node-awareness, queue search costs, and network contention penalties.

To my family and friends

ACKNOWLEDGMENTS

This dissertation would not have been possible without many people. Most importantly, I would like to thank my advisor, Luke Olson, for all of his help with helping formulate clear research ideas, greatly improving the quality of publications, and teaching how to make easily readable graphs. I also would like to thank Bill Gropp for his helpful suggestions throughout my research projects. Furthermore, I owe a huge thanks to my committee members, Laura Grigori and Edgar Solomonik.

Several other teachers and professors have had a large impact on my education. I would like to thank my high school computer teacher, Kathy Hoeper, for providing a great introduction to programming. Furthermore, the Computer Science professors at Elon University provided an excellent foundation of programming knowledge. I would specifically like to thank Joel Hollingsworth for introducing me to research with REU programs, as well as for creating a thoroughly enjoyable introduction to parallel programming.

I am incredibly thankful for the support of my parents, Mark and Jean Bienz. From all the way back in elementary school, when they put the idea of becoming a computer scientist in my head, they have been incredibly supportive of my education. They provided me with positive computer experiences long before formal education, and my some of my oldest memories are hanging out with my mom for hours as she 'crashed' our slowed-down computer before reinstalling Windows 95 via dozens of floppy discs.

Finally, I would like to thank all of my friends and family who made these past six years of research and writing so enjoyable. Thanks to all of the students of the Scientific Computing group, for making the office an incredibly fun place to work. A huge thanks to my sister, Lindsay, for showing me exciting parts of the world before conferences, and to my brother, Brian, for adding entertainment to everyday of my life. I would like to thank Jordan Essman, for providing my office with entertainment from a few hundred miles away with ideas like Trash Tuesday. And last but certainly not least, I owe a great deal of gratitude to Carl Pearson, for being incredibly supportive and helping me get past the stress of writing.

The research throughout this dissertation was supported by the National Science Foundation Graduate Research Fellowship Program, under Grant Number DGE-1144245.

TABLE OF CONTENTS

CHAPTER 1 INTRODUCTION 1
1.1 Communication in Scientific Codes 1
1.2 State of the Field 3
1.3 Overview of Contributions 4

CHAPTER 2 BACKGROUND 6
2.1 Sparse Matrices and Operations 6
2.2 Algebraic Multigrid 7
2.3 Performance Modeling 9

CHAPTER 3 MODELING POINT-TO-POINT COMMUNICATION 12
3.1 Introduction 12
3.2 Background 13
3.3 Node-Aware Modeling 14
3.4 Additional Penalties 15
3.5 Applications 21
3.6 Conclusion 23

CHAPTER 4 SPARSIFICATION 25
4.1 Introduction 25
4.2 Method of non-Galerkin coarse grids 29
4.3 Sparse and Hybrid Galerkin approaches 33
4.4 Parallel Performance 37
4.5 Parallel results for Sparse and Hybrid Galerkin 38
4.6 Adaptive Solve Phase 49
4.7 Conclusion 50

CHAPTER 5 NODE-AWARE MESSAGE AGGLOMERATION 53
5.1 Introduction 53
5.2 Background 55
5.3 Communication Models 59
5.4 Node Aware Parallel SpMV 62
5.5 Results 72
5.6 Conclusion and Future Work 76

CHAPTER 6 EXTENSIONS TO SPGEMM 79
6.1 Introduction 79
6.2 Background 81
6.3 Results 86
6.4 Conclusion 89

CHAPTER 7 NODE-AWARE AMG 90
7.1 Introduction 90
7.2 Background 91
7.3 Node-Aware Communication 103
7.4 Results 104
7.5 Conclusions 110

CHAPTER 8 CONCLUSIONS 112
8.1 Summary 112
8.2 Concluding remarks and future directions 113

APPENDIX A RAPTOR 115

REFERENCES 118

CHAPTER 1: INTRODUCTION

Parallel computers are continuously advancing, yielding increased processing power with each new generation. Emerging architectures are comprised of increasingly large networks of symmetric multiprocessing (SMP) nodes, each consisting of several processors that share main memory. This increased processor count yields potential to solve increasingly large and difficult scientific applications. As the accuracy of linear systems arising from discretized partial differential equations (PDEs) is correlated to system dimension, the additional compute units of state-of-the-art computers can be used to solve PDEs to increased accuracy. While computational work can be partitioned across a large number of processes, reducing the per-core cost and yielding reduced time to solution, many computational kernels are unable to take full advantage of parallel computing advances due to limited parallel scalability.

1.1 COMMUNICATION IN SCIENTIFIC CODES

Scientific simulations, for example based on solving discretized partial differential equations (PDEs), often rely on solving a sparse linear system

$$Ax = b, \tag{1.1}$$

where A is a sparse $n \times n$ matrix, and x and b are n-dimensional vectors.

There are several approaches to solving sparse linear systems, from sparse direct methods to sparse iterative solvers. Sparse direct methods consist of factorizing the matrix A into multiple triangular matrices followed by triangular solves. Common factorization methods include Cholesky, QR, and Gaussian elimination methods. Furthermore, multi-frontal methods yield a factorization through a series of small dense factorizations. Alternatively, iterative methods consist of forming an initial guess to the solution x, and iteratively reducing error until convergence. There are many classes of iterative methods. Basic methods, such as Jacobi, Gauss-Seidel, and successive over-relaxation (SOR), reduce error by averaging solutions of neighboring points. Krylov subspace methods, such as conjugate gradient (CG) and generalized minimized residual method (GMRES), project the system onto a Krylov subspace, which is spanned by the vectors $A^k r$, for some k, where $r \leftarrow b - A \cdot x$. Furthermore, domain decomposition methods split the problem into smaller problems on subdomains, and multilevel methods such as geometric (MG) and algebraic (AMG) multigrid construct a global problem of lower dimension to successively reduce error. Finally, incomplete LU (ILU) is often used as a preconditioner, approximating Gaussian elimination by dropping a

subset of non-diagonal entries.

Sparse direct and iterative methods lack scalability due to inherently sequential portions of algorithms as well as large inter-process communication requirements. Direct method factorization such as Gaussian elimination consists of factorizing one row and using this factorization in all subsequent rows. Furthermore, direct methods require communication of the updated matrix during each step, as subsequent matrix entries depend on each update. Factorization yields fill-in, adding non-zeros to positions of the matrix that are initially zero, increasing both local computation and communication at further steps as well as the triangular solves. Finally, the triangular solves require communication of the vector at each step.

Sparse iterative methods require several types of inter-process communication. The Jacobi iteration method, for example, is inherently parallel, requiring only vector communication each iteration. However, other basic methods such as Gauss-Seidel and SOR yield inherently sequential portions similar to sparse direct methods, yielding large per-iteration communication requirements. Krylov subspace methods require sparse vector communication each iteration to compute the residual vector. Furthermore, orthogonalization requires an inner product, typically implemented as a collective `MPI_Allreduce` operation. Multilevel methods require a combination of sparse vector and sparse matrix communication, much of which is performed on matrices and vectors that are smaller in dimension than the original system.

The sparse matrix and vector communication required throughout both sparse direct and iterative methods consists of communicating a portion of the matrix or vector between sets of processes. For one-dimensional matrix partitions, this communication depends on the sparsity pattern of the matrix as well as the partition across processes. Therefore, communication throughout sparse solvers partitioned in only one-dimension is irregular, with communication-load imbalance between processes as well as irregular combinations of sending and receiving processes. This irregularity does not exist in all algorithm variants, particularly those which use two- or three-dimensional matrix partitions, but irregular point-to-point communication minimizes communication when matrices are sufficiently sparse.

Figure 1.1 shows the communication pattern between sets of processes for both a regular `MPI_Allreduce` operation as well as irregular communication in a sparse matrix operation when using a standard one-dimensional decomposition. This dissertation focuses on this irregular communication that is found throughout the sparse direct and iterative methods with one-dimensional partitions.

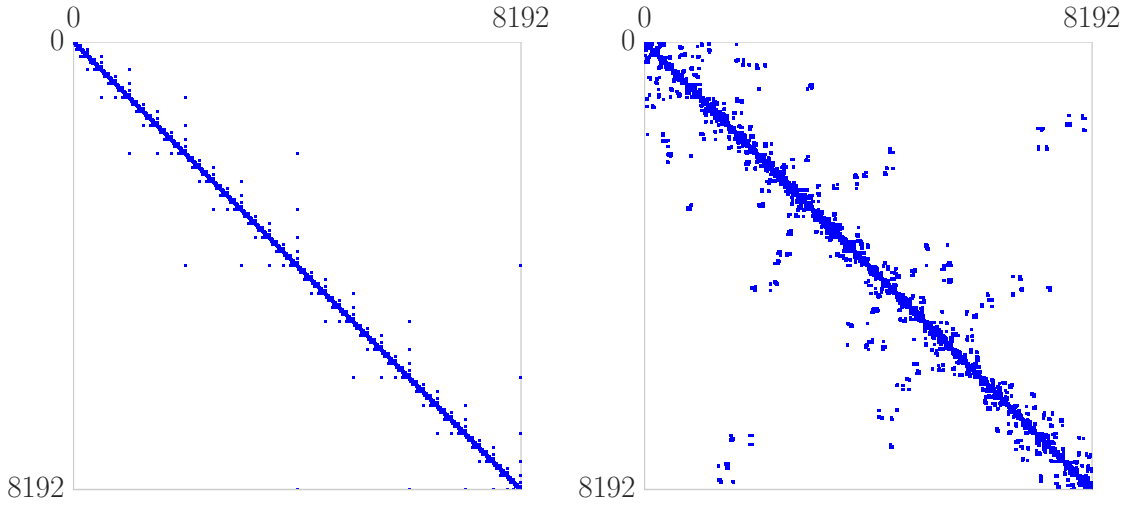


Figure 1.1: The inter-process communication pattern required to perform an MPI all-reduce collective operation (left) and irregular point-to-point communication inside a sparse matrix operation (right) for a coarse matrix in an AMG grad-div hierarchy on 8192 processes. Each non-zero A_{ij} represents that a message is sent from process i to rank j.

1.2 STATE OF THE FIELD

Sparse iterative methods, such as Krylov subspace methods, are comprised of three main costs:

1. local computation;
2. point-to-point communication in sparse matrix operations; and
3. MPI all-reduce collective communication.

Both point-to-point and collective communication yield reduced scalability of Krylov methods. Many methods exist to reduce the synchronization bottleneck that results from collective communication, including minimizing the number of Krylov iterations, preconditioning each iteration of the solver or increasing the amount of work per iteration. Enlarged Krylov methods reduce communication by increasing the number of vectors added to the Krylov subspace during each iteration, yielding increased work per-iteration over fewer iterations [1]. Communication-avoiding Krylov methods split the iterations into an outer loop and inner loop, performing several computation steps on the inner loop before communicating, resulting in reduced communication at the cost of convergence [2, 3]. Furthermore, pipelined Krylov methods hide global synchronization steps, yielding increased scalability [4].

Irregular communication in general sparse matrix operations dominates the cost of many sparse direct and iterative methods. This communication consists of sending either matrix or

vector entries that are associated with non-zero columns of the sparse matrix, requiring the irregular and unbalanced communication displayed in Figure 1.1. Graph partitioning and matrix reordering minimizes the number of edges between processes and reduces point-to-point communication in sparse matrix operations [5, 6]. Furthermore, data layout influences communication requirements: one-dimensional matrix partitions are often less scalable than two- and three-dimensional partitions, but [7, 8]. The sparse matrix storage format affects communication, as block formats reduce data requirements associated with sparse matrix communication [9]. Matrix operations performed redundantly on multiple processes avoid inter-process communication requirements. Ultimately, the design of new parallel algorithms leads to reduced communication; relying on algorithms developed for serial architectures offers limited opportunities [10].

1.3 OVERVIEW OF CONTRIBUTIONS

Point-to-point communication remains a costly component of many parallel codes, specifically with sparse matrix operations. This dissertation investigates methods for reducing the cost associated with irregular point-to-point communication in sparse matrix operations, improving the scalability of scientific simulations. The contributions of this thesis include the following:

Performance models for irregular point-to-point communication Traditional performance models are extended to include penalties that arise during irregular sparse matrix operations, such as queue search costs and network contention. These performance models are described in Chapter 3.

Removing non-zero entries, adding sparsity into the matrix: Sparsification methods are used throughout AMG to remove small non-zero entries that do not fit a predetermined sparsity pattern. These methods can greatly reduce communication requirements associated with each SpMV, with little impact on convergence. This is detailed in Chapter 4.

Node-aware communication in sparse matrix operations: Communication is agglomerated on each node before executing inter-node communication. The data is then redistributed among processes of the receiving node. Node-aware communication, described in Chapter 5, is analyzed throughout the following operations

Sparse matrix-vector multiplication (SpMV): Vector values are communicated using three-step node-aware communication before a local SpMV is performed on

local and received values. The node-aware SpMV is described in Chapter 5.

Sparse matrix-matrix multiplication (SpGEMM): Node-aware communication is developed for the sparse matrix communication in SpGEMMs in Chapter 6. Each sparse matrix is aggregated on node before being communicated through the network.

Algebraic multigrid setup and solve phases: Node-aware communication is applied throughout both the setup and solve phases of algebraic multigrid in Chapter 7. Both the classical and smoothed aggregation setup methods are analyzed.

CHAPTER 2: BACKGROUND

2.1 SPARSE MATRICES AND OPERATIONS

In parallel, a sparse $n \times n$ linear matrix A is partitioned row-wise across processes, such that each process holds a contiguous portion of the rows of the matrix, and corresponding vector values, as displayed in Figure 2.1. Note, one-dimensional partitions of non-contiguous rows are possible, as a matrix permutation can transform the rows into contiguous blocks. In

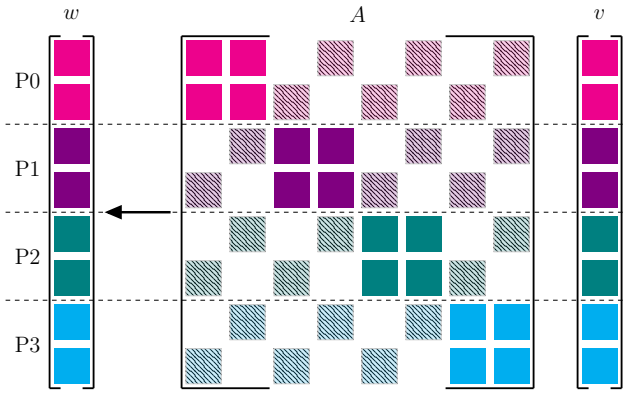


Figure 2.1: A matrix partitioned across four processes, where in this example each process stores two rows of the matrix, and the equivalent rows of each vector. The on-process block of each matrix partition is represented by solid squares ■, while the off-process block is represented by patterned entries ▧.

addition, the rows of A on a single process into two groups: an on-process block, containing the columns of the matrix that correspond to vector values stored locally, and an off-process block, containing matrix non-zeros that are associated with vector values that are stored on non-local processes. Therefore, non-zeros in the off-process block of the matrix require communication of vector values during each sparse matrix operation.

Sparse matrix operations lack parallel scalability due to large costs associated with communication, specifically in the strong scaling limit of a few rows per process. Increasing the number of processes that a matrix is distributed across increases the number of columns in the off-process blocks, yielding a growth in communication.

Figure 2.2 shows the percentage of time spent communicating during a SpMV operation for two large matrices from the SuiteSparse matrix collection [11] at scales varying from 50 000 to 500 000 non-zeros per process. The results show that the communication time dominates the computation as the number of processes is increased, thus decreasing the scalability.

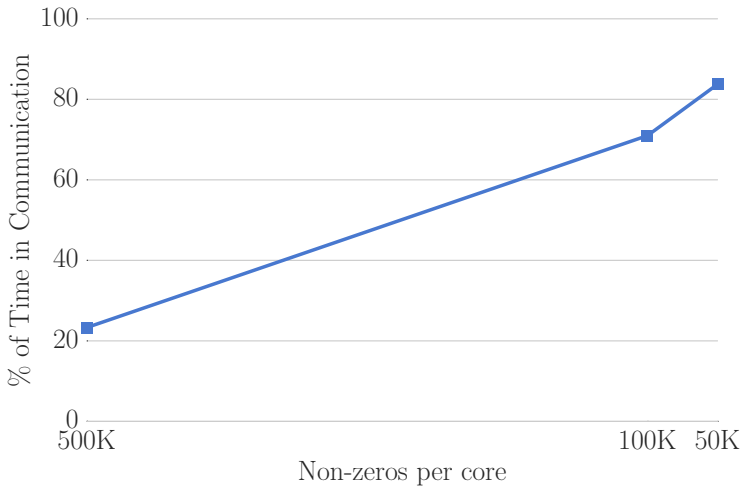


Figure 2.2: Percentage of total SpMV time spent during communication for matrix `nlpkkt240` with 760,648,352 non-zeros

2.2 ALGEBRAIC MULTIGRID

In this section we detail the AMG setup and solve phases. We let the *fine-grid* operator A be denoted with a subscript as A_0.

Algorithm 2.1 describes the setup phase and begins with `strength`, which identifies the strongly connected edges[1] in the graph of A_ℓ at level ℓ to construct a strength-of-connection matrix S_ℓ. From this, P_ℓ is constructed in `interpolation` to transfer vectors from level $\ell+1$ to level ℓ, with the goal of accurately interpolating (smooth) error not sufficiently reduced by relaxation. For classical AMG, `interpolation` first forms a disjoint splitting of the level ℓ index set $\{0,\ldots,n-1\} = C \cup F$, where C is the set of indices on the coarse level and where F is the set of indices for variables that reside only on the fine level. The size of the coarse grid is then given by $n_{\ell+1} = |C|$, and an interpolation operator, $P_\ell : \mathbb{R}^{n_{\ell+1}} \rightarrow \mathbb{R}^{n_\ell}$, is constructed using S_ℓ and A_ℓ to compute sparse interpolation formulas that are accurate for algebraically smooth functions. Finally, the coarse-grid operator is created through a Galerkin triple-matrix product, $A_{\ell+1} = P_\ell^T A_\ell P_\ell$. In a two-level setting, this ensures the desirable property that the coarse-grid correction process $I - P_\ell A_{\ell+1}^{-1} P_\ell^T A_\ell$ is an A_ℓ-orthogonal projection that ensures optimal coarse-grid correction of the error in the range of interpolation (in the A_ℓ norm). When an approximation to $A_{\ell+1}$ is introduced, this projection property is lost, leading to a possible deterioration in convergence. This is further discussed in Chapter 4.

[1] A degree-of-freedom i is strongly connected to j if algebraically smooth error (error not effectively reduced by relaxation) varies slowly between i and j. Strength information critically informs AMG coarsening [12] and interpolation [13]. While a variety of strength measures abound [14], the standard strength measure is sufficient for the problems tested.

Algorithm 2.1: `amg_setup`

Input: A_0: fine-grid operator
`max_size`: threshold for max size of coarsest problem
`nongalerkin`: (optional) non-Galerkin method
$\gamma_1, \gamma_2, \ldots$ (optional) drop tolerances for each level

Output: $A_1, \ldots, A_L$,
$P_0, \ldots, P_{L-1}$

while `size`$(A_\ell) >$ `max_size`
- $S_\ell =$ `strength`(A_ℓ) {Strength-of-connection of edges}
- $P_\ell =$ `interpolation`(A_ℓ, S_ℓ) {Construct interpolation and injection}
- $A_{\ell+1} = P_\ell^T A_\ell P_\ell$ {Galerkin product}
- **if** `nongalerkin` {(optional) described in Section 4.2}
  - $\hat{A}_{\ell+1} =$ `sparsify`$(A_{\ell+1}, A_\ell, P_\ell, S_\ell, \gamma_\ell)$ {Remove nonzeros in $A_{\ell+1}$}
  - $A_{\ell+1} = \hat{A}_{\ell+1}$

The density of each coarse-grid operator $A_{\ell+1}$ depends on that of the interpolation operator P_ℓ. Even interpolation operators with modest numbers of nonzeros typically lead to increasingly dense coarse-grid operators [15, 16]. Algorithm 2.1 addresses this with the optional step `sparsify`, which triggers the sparsification steps developed in Chapter 4. The approach [15] also fits within this framework, which is detailed in Section 4.2.

The solve phase of AMG, described in Algorithm 2.2 as a V-cycle, iteratively improves an initial guess x_0 through the use of the residual equation $A_0 e_0 = r_0$, where e_0 and r_0 are the fine-grid error and residual, respectively. High energy error in the approximate solution is reduced through relaxation in `relax` — e.g. Jacobi or Gauss-Seidel. The remaining error is reduced through coarse-grid correction: a combination of restricting the residual equation to a coarser level, followed by interpolating and correcting with the resulting approximate error. The coarsest-grid equation is computed with `solve`, using a direct solution method.

The dominant computational kernel in Algorithm 2.2 is the sparse matrix-vector (SpMV) product, found in `relax` and interpolation/restriction. Typically relaxation dominates since A_ℓ is larger and denser than P_ℓ. Thus, the performance on level ℓ of the solve phase depends strongly on the performance of a single SpMV with A_ℓ.

For a SpMV, communication is required for all off-process elements in the vector that correspond to matrix nonzeros Therefore, the density of a matrix contributes to the cost of communication complexity in the SpMV operation. This implies that the less sparse AMG

Algorithm 2.2: `amg_solve`

Input: x_0: fine-level initial guess
b_0: fine-level right-hand side
$A_0, \ldots, A_L$
$P_0, \ldots, P_{L-1}$
Output: x_0, fine-level approximation

for $\ell = 0, \ldots, L-1$ **do**
  $\texttt{relax}(A_\ell, x_\ell, b_\ell, \nu_1)$ {Pre-smooth ν_1 times}
  $b_{\ell+1} = P_\ell^T(b_\ell - A_\ell x_\ell)$ {Restrict residual}
$x_L = \texttt{solve}(A_L, b_L)$ {Coarsest-level direct solve}
for $\ell = L-1, \ldots, 0$ **do**
  $x_\ell = x_\ell + P_\ell x_{\ell+1}$ {interpolate and correct}
  $\texttt{relax}(A_\ell, x_\ell, b_\ell, \nu_2)$ {Post-smooth ν_2 times}

coarse levels yield large communication costs and, often, an inefficient solve phase [15, 16].

2.3 PERFORMANCE MODELING

Parallel operations, such as those involving sparse matrices, often require MPI point-to-point communication. This category of communication consists of sending a single message between a set of processes. In a typical implementation, the pairs of communicating processes, along with the size of associated messages, vary. In addition, point-to-point communication procedures vary with MPI implementation. Each message consists of both an envelope and data, where the envelope contains a message description including the tag, MPI communicator, message length, and process of origin. There are a variety of methods for sending data, such as sending the data immediately or waiting for the receive process to allocate buffer space. In the implementations investigated here, a message is communicated via a specific protocol of short, eager, or rendezvous, based on message size. The short protocol consists of sending very small messages as part of the envelope directly between processes. Messages that are too large to fit in the envelope, but remain relatively small, are communicated with eager protocol. This protocol assumes buffer space is available, and immediately communicates the data to the receiving process. Lastly, sufficiently large messages are communicated with rendezvous protocol, during which the envelope is communicated first, and the remainder of the data is only communicated after the receiving process has allocated buffer space.

The cost associated with each message is dependent on the time required to initialize

communication as well as the per-byte transport cost. Therefore, the short protocol is significantly less costly than the others as only a single envelope is communicated, yielding minimal costs associated with both latency and bandwidth. Messages communicated with eager protocol have low latency costs as the messages are sent directly between processes. However, the transport cost correlates with message size; messages can require significant amounts of buffering depending on size. Finally, rendezvous messages yield low per-byte transport costs but increased latency requirements associated with initial envelope communication and synchronization.

The traditional postal model estimates the cost of communicating a message as the sum of the message startup cost and the per-byte transport, with separate parameters for each message protocol. That is,

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

where α is the latency, $\frac{1}{\beta}$ is the rate at which a byte of data is transported, and s is the number of bytes to transport. As the associated costs vary with message protocol, separate values for α and β are used when communicating with short, eager, and rendezvous protocols.

The max-rate model [17] improves upon the postal model by defining the cost of communication as dependent on not only the latency and inter-process bandwidth costs, but also on the maximum bandwidth by which a node can inject data into the network. Therefore, the max-rate model accounts for the fact that injection bandwidth becomes a bottleneck when communicating from four or more processes per node, as is typical with state-of-the-art parallel computers [17].

The max-rate model is defined as

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

where ppn is the number of actively communicating processes per node, R_b is the rate at which data can be sent between two processes, or the inverse of β, and R_N is the maximum rate at which a node can inject data into the network. Therefore, when the value of $\texttt{ppn} \cdot R_b$ is less than injection bandwidth, this model reduces to the postal model. However, with a sufficiently large number of active processes per node, the per-byte transport rate is measured as injection bandwidth. Figure 2.3 displays the cost of communicating a single message of various size between pairs of processes on the same socket, on different sockets of a single node, and on different nodes. The intra-socket and inter-socket costs are measured with the traditional postal model, while the off-node communication is modeled with the max-rate model.

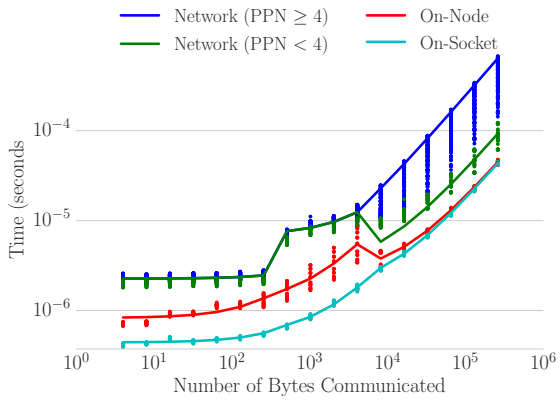


Figure 2.3: The max-rate model versus measured times for on-socket, on-node but off-socket, and off-node communication.

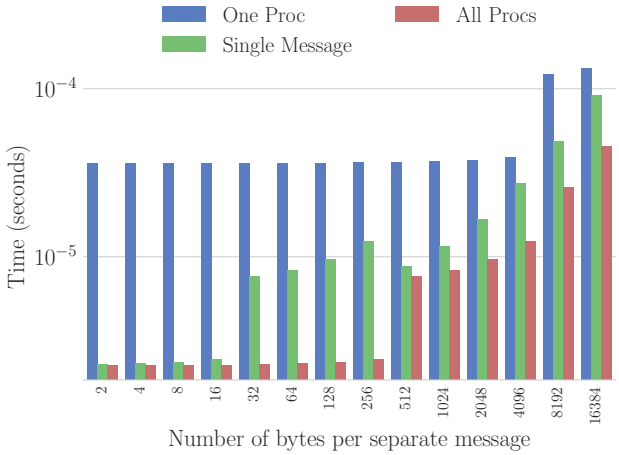


Figure 2.4: The cost of communicating between data of size $16s$ between two nodes, n and m, with the various values for s on the x-axis of the plot. All times are calculated with the max-rate model.

The cost of communicating data of between two nodes, varies with implementation, as shown in Figure 2.4. For example, gathering all data on one process of node n before communicating as individual messages of size s to each of 16 processes on node m is significantly more expensive than sending as a single message of size $16s$ to one of the processes on node m. However, this cost is further reduced by communicating one message of size s from each of the 16 processes on node n to a single process on node m. Therefore, the cost of inter-node communication is minimized by limiting both the number and size of messages communicated from any process.

CHAPTER 3: MODELING POINT-TO-POINT COMMUNICATION

Portions of this chapter appear in the paper "Improving Performance Models for Irregular Point-to-Point Communication", in submission to EuroMPI 2018 [18].

3.1 INTRODUCTION

Hardware advances in parallel computers continue to enable solving increasingly large problems. However, applications are often unable to take full advantage of state-of-the-art architectures due to scaling limitations, which result from inter-process communication costs. The cost associated with communication depends on a large number of factors, and varies across parallel systems, specific partitions, and application scale. Therefore, performance models are used to analyze the sources of communication costs among different architectures and network partitions. Accurate performance models indicate whether the cost is due mainly to the number of messages communicated, number of bytes transported, distance of transported bytes, etc.

Traditional models estimate point-to-point communication as a combination message latency and the cost of transporting bytes. Irregular operations, such as sparse matrix methods, incur costs that are not captured by traditional models. Figure 3.1 displays the measured and modeled communication costs acquired when performing a sparse matrix-matrix (SpGEMM) multiply on the levels of an algebraic multigrid (AMG) hierarchy for an unstructured linear elasticity matrix. These timings, as well as the associated model parameters,

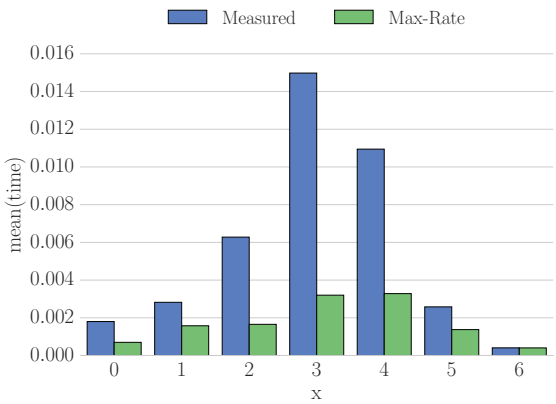


Figure 3.1: Measured and modeled communication costs associated with a SpGEMM on each level of a linear elasticity AMG hierarchy, on 8 192 processes of Blue Waters supercomputer.

are for 8 192 processes of Blue Waters supercomputer [1] [19]. The traditional postal model results in nearly identical timings to more robust models, such as the max-rate model [17], which takes into account the limitations of multiple communicating processes per node. For this problem both models capture only a fraction of the measured time.

This chapter extends traditional performance models to accurately model the irregular point-to-point communication that occurs in commonly used operations. This chapter presents three novel contributions, including an improvement to the max-rate model [17] measurements:

1. node-aware model parameters;
2. an extension to include quadratic queue search times in communication; and
3. an additional parameter estimating network contention.

The remainder of this chapter is outlined as follows. Section 3.2 describes parallel point-to-point communication as well as corresponding traditional performance models. Node-awareness is added to traditional models in Section 3.3. Section 3.4 describes a high-volume ping-pong algorithm along with additional acquired penalties, with a queue search parameter described in Section 3.4.1 and a network contention penalty in Section 3.4.2. The improved model parameters are applied to commonly used operations and compared against measured timings in Section 3.5. Finally, conclusions, limitations, and future directions are described in Section 3.6.

3.2 BACKGROUND

Traditional performance models, such as the postal and max-rate models accurately analyze the cost of a standard ping-pong test, in which two processes are sending messages to one another. However, they fail to account for a variety of penalties that occur during communication in typical operations on state-of-the-art supercomputers.

There are many alternatives to traditional models that account for many penalties that arise in standard supercomputers. The LogP model splits the α into latency, the cost required by the hardware, and overhead, the cost associated with the software [20, 21]. This addition of overhead allows this model to capture the cost of overlapping communication and computation. The LogGP model extends the LogP model to analyze the cost of long messages [22]. Network contention parameters are investigated with the LoPC and LoGPC

[1] `https://bluewaters.ncsa.illinois.edu/`

models [23, 24]. Accurate models exist for network contention in collective communication, such as the `MPI_Alltoall` operation [25]. Futhermore, learning algorithms yield accurate contention prediction [26]. Topology-awareness can improve models, as hop count, or the number of links traversed by a message, affects the cost of communication [27]. Computer simulations can accurately estimate the cost of communication, but at a significantly increased cost [28, 29, 30]. Network contention has been previously modeled for collective communication, specifically the `MPI_Alltoall` operation.

Throughout the remainder of this chapter, the max-rate model is used as a baseline. All ping-pong timings are collected through multiple runs of Baseenv [2], a topology-aware library useful for benchmarking performance. Each ping-pong test consists of four duplicate timings, with exception to the original max-rate tests, which test the various numbers of actively communicating processes-per-node one time each. Each Baseenv program is tested three different times.

The models throughout this chapter are tested with Blue Waters, a Cray XE/XK machine at the National Center for Supercomputing Applications (NCSA) at University of Illinois. Blue Waters contains a 3D torus Gemini interconnect, in which each Gemini consists of two nodes. The system contains 22 636 XE compute nodes, each comprised of two AMD 6276 Interlagos processors, as well as 4 228 XK compute nodes containing a single AMG processor along with an NVIDIA GK110 Kepler GPU. All tests in this chapter are performed on partitions of XE system nodes. The tests use a CrayMPI implementation that is similar to MPICH.

3.3 NODE-AWARE MODELING

The standard max-rate model improves upon the postal model by adding a parameter for injection bandwidth limits. Figure 3.2 displays the max-rate model versus measured times when communicating a single message of various size between pairs of processes. These associated models are computed with published Blue Waters parameters [17], and the measured times are acquired from sending messages between processes that lie on the same socket, different sockets of the same node, or on neighboring nodes of Blue Waters. While the addition of network injection limits yields large improvement over the standard postal model when communicating rendezvous messages from a large number of processes per node, the model overestimates for a large portion of these timings.

There is a large difference between intra-socket messages, intra-node messages that traverse

[2] `http://wgropp.cs.illinois.edu/projects/software/index.html`

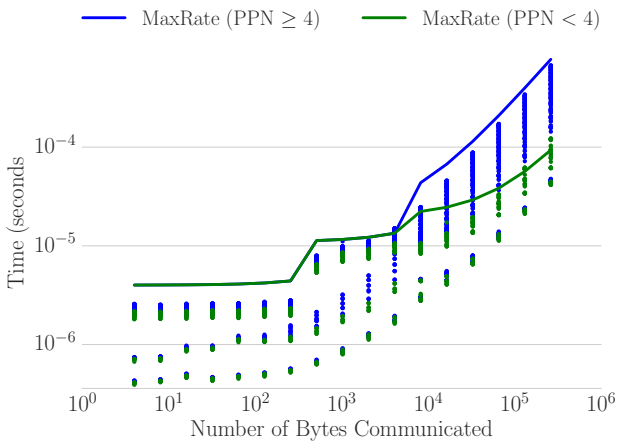


Figure 3.2: Ping-pong measured times (dots) versus max-rate model (lines) on Blue Waters using parameters from [17].

	intra-socket		intra-node		inter-node		
	α	R_b	α	R_b	α	R_b	R_N
short	4.4e-07	2.09	8.3e-07	4.8e08	2.3e-06	1.3e09	∞
eager	5.3e-07	3.09	1.2e-06	9.6e08	7.0e-06	7.5e08	∞
rend	1.7e-06	6.09	2.5e-06	6.2e09	3.0e-06	2.9e09	6.6e09

Table 3.1: Parameters for node-aware max-rate model on Blue Waters.

across sockets, and communication between two nodes. Therefore, different parameters should be used for each of these cases. Furthermore, intra-node messages are not injected into the network, and therefore the simple postal model is sufficient. Figure 2.3 displays the measured versus modeled times after splitting the model into on-socket, on-node but off-socket, and off-node messages for Blue Waters. The parameters corresponding this node-aware model are listed in Table 3.1.

3.4 ADDITIONAL PENALTIES

Realistic applications that involve point-to-point communication typically require communication of more than one message from any process. However, standard communication models assume the cost of sending a single message, and do not extend accurately to large message counts. A ping-pong test with large message counts, described in Algorithm 3.1, captures additional costs not observed in the postal or max-rate models.

Algorithm 3.1: `HighVolumePingPong`

Input: `rank`: MPI rank of current process
p: process with which to communicate
n: number of messages to communicate
s: size of each message (in bytes)
`send_tags`: list of n MPI send tags
`recv_tags`: list of n MPI receive tags

```
if rank < p
    for i < n do
        MPI_Isend(..., s, ..., p, send_tags_i, ...)
    MPI_Waitall(n, ...)

    for i < n do
        MPI_Irecv(..., s, ..., p, recv_tags_i, ...)
    MPI_Waitall(n, ...)
else
    for i < n do
        MPI_Irecv(..., s, ..., p, recv_tags_i, ...)
    MPI_Waitall(n, ...)

    for i < n do
        MPI_Isend(..., s, ..., p, send_tags_i, ...)
    MPI_Waitall(n, ...)
```

3.4.1 Queue Search

Point-to-point communication conceptually requires multiple queues to be formed and traversed, including a send queue comprised of sends that have been posted, a receive queue of similarly posted receives, and an unexpected message queue containing communicated messagesfor which no matching receive has been posted [31]. The function and availability of these queues, which are dependent on MPI implementation, greatly affect the performance of communicating a large number of messages.

The standard implementation of MPICH creates two separate receive queues, one for the posted messages and the other for unexpected messages [32]. When an envelope is received, the queue of posted messages is searched for a message in which all variables such as tag, datatype, communicator, and sending process match the envelope. If no associated message has been posted, the envelope and any corresponding data is added to the unexpected

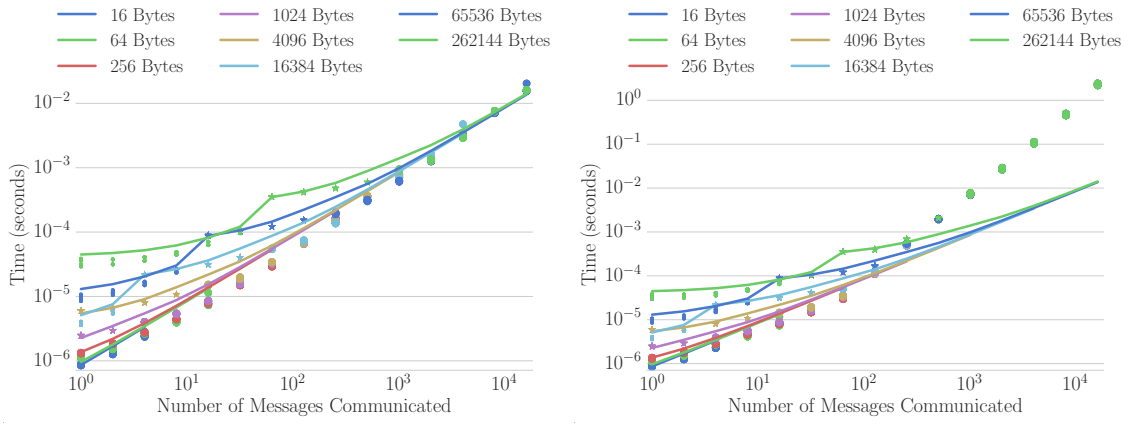


Figure 3.3: The measured versus modeled (max-rate) costs of sending a number of messages between two processes that lie on the same node of Blue Waters. On the left, all receives are posted in the same order that messages are received, resulting in no queue search cost. The right plot displays the cost when receives are posted in the opposite order from which they are received, resulted in a quadratic queue search cost that is not captured in the max-rate model.

message queue. Similarly, when a message is posted by the application, the unexpected message queue is traversed for any corresponding envelope. If no match is found, the message is added to the posted message queue.

The CrayMPI implementation requires a receive queue to be searched linearly, yielding an additional cost when communicating multiple messages. In the worst case, the messages are received in the order opposite of which they are posted, requiring a traversal of an entire queue for each receive. Therefore, the queue search is an $\mathcal{O}(n^2)$ operation. More specifically, the cost is $\frac{n\cdot(n+1)}{2}$ operations in the worst case, with an average case of $\frac{n\cdot(n+1)}{4}$ operations. Methods have been created to reduce this queue search cost, such as the use of multiple queues in combination with hash maps [33]. However, as a standard queue search is currently implemented in the version of MPI on Blue Waters, the large queue search cost is investigated.

Figure 3.3 displays both the measured and modeled costs for performing the `HighVolumePingPong` described in Algorithm 3.1 among all 16 processes local to a single node on Blue Waters. The number of messages communicated ranges from 1 to 10 000 with the total number of bytes injected into the network remaining constant. In the ideal scenario, the variables $\texttt{send\_tags}_i$ is equal to $\texttt{recv\_tags}_i$ for all $i < n$, resulting in messages being received in the same order as they are posted. Therefore, the first message in the searched queue yields a match, resulting in an $\mathcal{O}(n)$ queue search cost. As a result, the max-rate model accurately analyzes these measured times. In the worst-cast scenario, the variables $\texttt{send\_tags}_i$ is equal

to `recv_tags`$_{n-i-1}$ for all $i < n$, posting receives in the opposite order from which messages are received. Therefore, the entire queue is traversed for each receive, resulting in an $\mathcal{O}(n^2)$ queue search cost, yielding measured times that vary greatly from the model.

The large inaccuracies of traditional models for large message counts motivates adding an additional parameter for the time required to search the receive queues. This addition to the models is defined as

$$T_q = \gamma \cdot n^2 \text{ seconds}, \tag{3.1}$$

where γ is the cost of stepping through either the posted or unexpected message queue. This cost is independent of message sending protocol as well as relative locations of the send and receive processes. Therefore, there is a single parameter for all combinations of on-socket, on-node, off-node, and short, eager, and rendezvous. The cost of stepping searching the queue based on the worst case test, yields

$$\gamma = 8.4e-09 \text{ seconds per search operation}. \tag{3.2}$$

Figure 3.4 shows the measured versus modeled times for the `HighVolumePingPong` test in which the messages are received in the opposite order from which they are posted. This

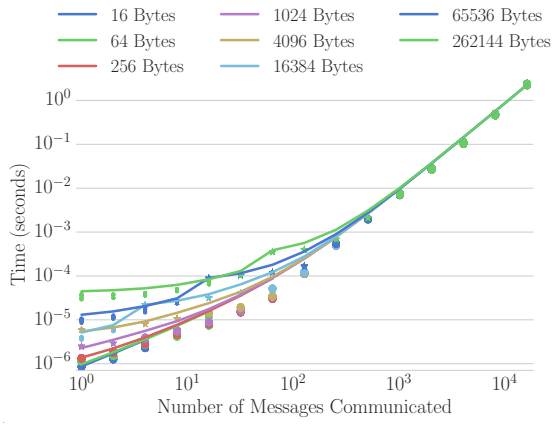


Figure 3.4: The measured versus modeled times for Blue Waters, where the model is a combination of the max-rate model and the contention, for `HighVolumePingPong` tests with a variety of message counts and sizes. The receives are posted in the opposite order of which messages are received.

figure adds the queue search parameter to the original max-rate model, yielding a more accurate analysis of the cost of large message counts.

3.4.2 Network Contention

When a large amount of data is communicated throughout the network, multiple messages are often required to traverse the same link, yielding contention within the network. Network contention can occur on a small partition of Blue Waters when a one-dimensional partition of the network is attained. Figure 3.5 shows a line of four Geminis, each containing two nodes, within a one-dimension network partition. Communicating messages from all 32 processes

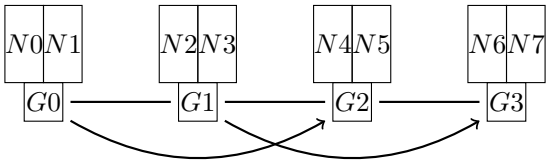


Figure 3.5: Four Gemini spanning a one-dimensional partition of the Blue Waters network. A `HighVolumePingPong` test between all processes on Geminis $G0$ and $G3$, and equivalent messages between $G1$ and $G3$ will result in a large amount of contention for the middle link in the partition.

on Gemini 0 to corresponding processes on Gemini 2, and similarly sending from Geminis 1 to 3, requires all data to traverse the network link between Geminis 1 and 2. Therefore, contention of this link occurs.

Figure 3.6 shows the measured and modeled costs of sending messages of various counts and sizes among all processes on the row of Geminis, where the model contains both the max-rate and queue search measures. The model underestimates the cost of communicating

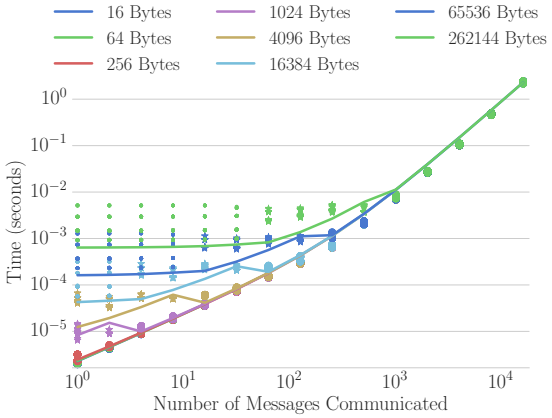


Figure 3.6: Measured versus modeled times for `HighVolumePingPong` communication among the sets of Blue Waters Geminis described in Figure 3.5. The modeled times, which are a combination of max-rate models and the queue search parameters, do not capture the additional costs associated with contention.

a large amount of data at smaller message counts, before queue search time dominates. The

additional measured cost can be modeled through an extra network contention parameter. This addition measure is defined as

$$T_c = \delta \cdot \ell \text{ seconds}, \tag{3.3}$$

where δ is the per-byte penalty acquired waiting for a network link and ℓ is the number of bytes to traverse each link. Network contention only occurs during inter-node communication. However, the cost of contention is constant regardless of the message sending protocol. The measure for all inter-node messages on Blue Waters is

$$\delta = 1.0e - 10 \text{ seconds per byte}. \tag{3.4}$$

The number of bytes to traverse any link, or ℓ, is dependent on the number of links each message traverses. Therefore, knowledge of the specific partition of the network is required to model the associated cost. This requirement is removed by assuming the nodes are connected through a perfect three-dimension cube portion of Blue Waters' three dimensional torus, as displayed in Figure 3.7. Therefore, ℓ is defined as the following

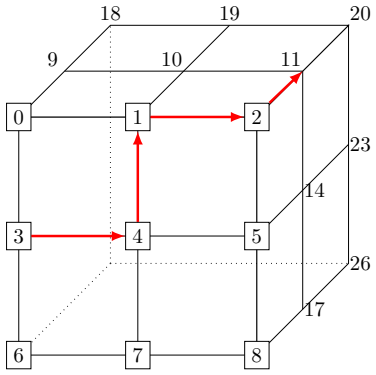


Figure 3.7: A perfect cube partition of Blue Waters Geminis is used to calculate the average number of hops traversed by each byte of data. In this example, a message from a process on Gemini 3 to Gemini 11 traverses 4 network links.

$$\ell = 2h^3 \cdot b \cdot \texttt{ppn} \text{ bytes}, \tag{3.5}$$

where h is the average number of hops, or network links, traversed by each byte of data, and b is the average number of bytes to be sent from any process. Therefore, as h^3 yields the number of Geminis within h hops of a given Gemini, this measure estimates network contention assuming all bytes that can traverse one single link do. Furthermore, $2b \cdot \texttt{ppn}$ calculates the average number of bytes communicated from each Gemini.

Figure 3.8 displays the measured and modeled costs of communicating a variety of message counts and sizes among four Geminis of Blue Waters, with a one-dimensional network contention. This figure includes the max-rate, queue search, and network contention models,

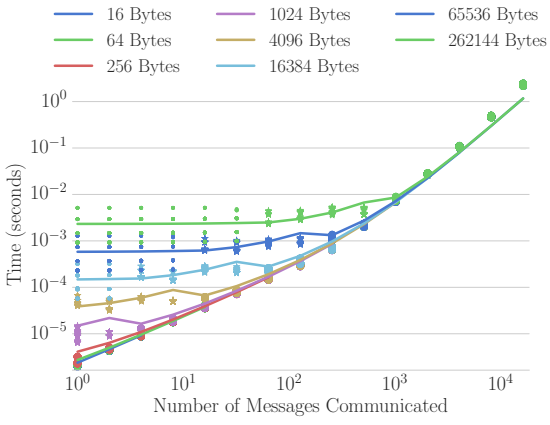


Figure 3.8: Modeled versus measured times for `HighVolumePingPong` communication about four Blue Waters Geminis spanning a one-dimensional partition of the networks. The processes on the Geminis and nodes communicate as described in Figure 3.5. The modeled times are a combination of max-rate models, queue search costs, and the contention parameter.

yielding improved accuracy in the model.

3.5 APPLICATIONS

Sparse matrix-vector (SpMV) and sparse matrix-matrix (SpGEMM) multiplication are commonly used in a variety of applications such as numerical methods and graph algorithms. When matrices are sufficiently sparse, point-to-point communication is used to send only necessary values to the processes that need them.

Algebraic multigrid (AMG) is a sparse linear solver comprised of matrix operations. An AMG hierarchy consists of successively coarser, but denser, matrices. Therefore, each level in the hierarchy decreases in dimension, but often increases in the number of non-zeros per row. The various levels require a variety of communication patterns, as the finer levels require communication of few large messages while coarse levels require communicating a larger number of small messages.

This section focuses on modeling the cost of matrix operations throughout an AMG hierarchy as communication costs vary drastically among the levels. The hierarchy is formed with classical AMG to solve a three-dimensional unstructured linear elasticity problems formed

with MFEM [3]. All tests are performed with RAPtor [34] on 512 nodes of Blue Waters. The original linear elasticity system consists of 840 000 unknowns and 65 million non-zeros.

Figure 3.9 displays the measured and modeled costs for performing a SpMV on each level of the AMG hierarchy. The modeled costs are partitioned into the max-rate model, queue

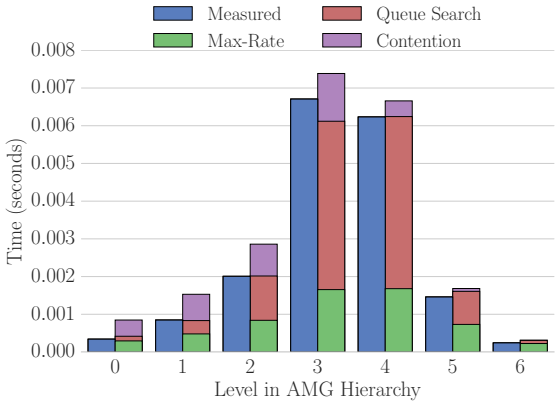


Figure 3.9: Measured versus modeled times for performing a **SpMV** on each level of linear elasticity AMG hierarchy.

search costs, and network contention penalties. The cost of each SpMV is accurately captured when all model parameters are included, with a large improvement over modeling with only the max-rate model. Furthermore, the model indicates that the majority of communication costs on coarse levels near the middle of the hierarchy are due mainly to queue search costs, motivating efforts for minimizing the number of messages received or posted at any time.

The measured and modeled costs for performing an SpGEMM on each level of the AMG hierarchy are shown in Figure 3.10. The models are again partitioned into max-rate, queue search, and network contention costs, displaying a large improvement in the model accuracy from the combination of queue search and contention parameters. These models show that more significant costs of the SpGEMMs are from network contention, and while queue search times could be reduced, larger savings would be acquired by reducing the number of bytes to traverse any link.

Combining the max-rate model with queue search and network contention parameters improves the accuracy of the model, but also over-predicts the timings. This over-prediction is a result of using an upper bound on the queue search parameter, corresponding to the cost of posting receives in the opposite order from which messages arrive. The upper bound assumes the $\frac{n\cdot(n+1)}{2}$ elements are searched, while only n elements are searched if the receives are posted in the correct order. The queue search cost was measured for each of these

[3] `http://mfem.org`

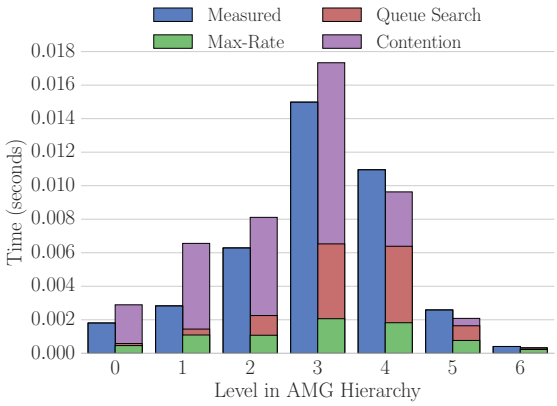


Figure 3.10: Measured versus modeled times for performing a **SpGEMM** on each level of linear elasticity AMG hierarchy.

applications by probing for the first available message and computing its position in the queue. The maximum cost on any process is consistently around $\frac{n^2}{3}$, which is still quadratic and close to the worst case. Furthermore, reversing the ordering of the receives results in a different process incurring the maximum queue search penalty, but the cost stays constant.

3.6 CONCLUSION

Common applications of point-to-point communication typically require a large number of messages to be communicated, with large variability in corresponding messages sizes. Furthermore, there are often combinations of intra-socket, intra-node, and inter-node messages, with the latter commonly traversing multiple links of the network. Therefore, the traditional postal and max-rate models can be improved by splitting standard parameters into on-socket, on-node, and off-node. Additional parameters for bounding the queue search cost and estimating network contention further improve the accuracy of these models. When all parameters are used, the models accurately capture the cost of irregular sparse matrix operations on Blue Waters.

The model parameters are all computed with ping-pong and `HighVolumePingPong` tests on few nodes, with the majority of tests requiring only a single node while network contention parameters are calculated on up to eight nodes. However, these parameters remain accurate when modeling sparse matrix operations on 512 nodes, indicating this model can be extended to large core counts with no additional work.

This model may need alterations to accurately capture the costs on different systems. For example, architectures with only a single socket per node, such as Blue Gene/Q machines, will only need to partition the max-rate model into on-node and off-node messages. Further-

more, MPI implementations with an optimized queue search will require alternative queue search penalties, while implementations with dynamic message routing will require a block of communicating nodes to capture network contention in the test in Figure 3.5.

Limitations for this model include using the upper bound for queue search time and assuming the process domain is mapped to a cube for network contention. The queue search time will overestimate the actual cost, while the accuracy of the network contention penalty can vary with actual partition acquired.

These models can be further extended to include topology-aware parameters, such as additional latency required for messages traversing a large number of links. Furthermore, the models motivate future directions for tested applications, such as methods for reducing queue search time in SpMVs and network contention in SpGEMMs.

CHAPTER 4: SPARSIFICATION

Portions of this chapter appear in the publication "Reducing Parallel Communication in Algebraic Multigrid through Sparsification" [35].

4.1 INTRODUCTION

Algebraic multigrid (AMG) [36, 37, 13] is an $\mathcal{O}(n)$ linear solver for standard discretizations of elliptic differential equations [38, 39, 40]. We consider AMG as a solver for the symmetric, positive definite matrix problem

$$Ax = b, \tag{4.1}$$

with $A \in \mathbb{R}^{n \times n}$ and $x, b \in \mathbb{R}^n$. AMG consists of two phases, a setup and a solve phase. The setup phase defines a sequence or *hierarchy* of L coarse-grid and interpolation operators, $A_1, \ldots, A_L$ and $P_0, \ldots, P_{L-1}$ respectively. The solve phase iteratively improves the solution through relaxation and coarse-grid correction.

The focus of this work is on the communication complexity of AMG in a distributed memory, parallel setting. To be clear, we refer to the *communication complexity* as the time cost of interprocessor communication, while referring to the *computational complexity* as the time cost of the floating point operations. The *complexity* or *total complexity* is then the cost of the algorithm, combining the communication and computational complexities.

There is a trade-off between per-iteration complexity and the resulting convergence — this is controlled by the setup phase in AMG. Indeed, more accurate interpolation leads to more entries in P_ℓ and this often improves the overall convergence. However, this also leads to more entries in $A_{\ell+1}$ through the Galerkin product, $A_{\ell+1} = P_\ell^T A P_\ell$, which is the most common way to form $A_{\ell+1}$ in AMG. As we will see, the number of entries in $A_{\ell+1}$ is a good proxy for the complexity of level $\ell + 1$. In contrast, sparser interpolation and fast coarsening reduce the complexity of a single iteration of an AMG cycle through fewer entries in $A_{\ell+1}$ and fewer overall AMG levels, but can lead to a deterioration in convergence [39, 40].

The sparse matrices, $A_1, \ldots, A_L$, in the multigrid hierarchy are, by design, smaller in dimension, yet are often more dense as the level increases. As an example of this, Table 4.1 shows the properties of a hierarchy for a 3D Poisson problem with a 27-point finite element stencil on a $100 \times 100 \times 100$ grid. Classical AMG (Ruge-Stüben) is used for this example[1]. As the problem size decreases on coarse levels, the average number of nonzero entries per

[1]See PyAMG [41] using `ruge_stuben_solver(poisson((100,100,100), type='FE'))` for more details. Similar results are seen using a 7-point finite difference discretization.

row increases. Here we denote by `nnz` the number of nonzero entries in the respective matrix. Figure 4.1 highlights this example, where we see that both the density and pattern of nonzeros increase on lower levels in the hierarchy.

level ℓ	matrix size n	nonzeros `nnz`	nonzeros per row $\texttt{nnz}/n$
0	1 000 000	26 463 592	26
1	124 984	3 645 644	29
2	23 042	1 466 006	64
3	2991	198 043	66
4	570	32 680	57
5	117	4705	40

Table 4.1: Matrix properties using classical AMG for a 3D Poisson problem.

Figure 4.1: Matrix sparsity pattern using classical AMG for three levels in the hierarchy: $\ell = 3$, 4, 5. The full matrix properties are given in Table 4.1.

In parallel, coarse levels that are more dense correlate with an increase in parallel communication costs [42]. Figure 4.2 shows this by plotting the time spent on each level in an AMG hierarchy during the solve phase. The time grows substantially on coarse levels, which is attributed to increased communication costs from a decrease in sparsity. This effect is common in AMG methods; Figure 4.2 shows two examples.

In this chapter we introduce a method for controlling the communication complexity in AMG. The method increases the sparsity of coarse-grid operators (A_ℓ, $\ell = 1, \ldots, L$) by eliminating entries in A_ℓ. This results in an improved balance between convergence and per-iteration complexity in comparison to the standard algorithm. In addition, we develop an

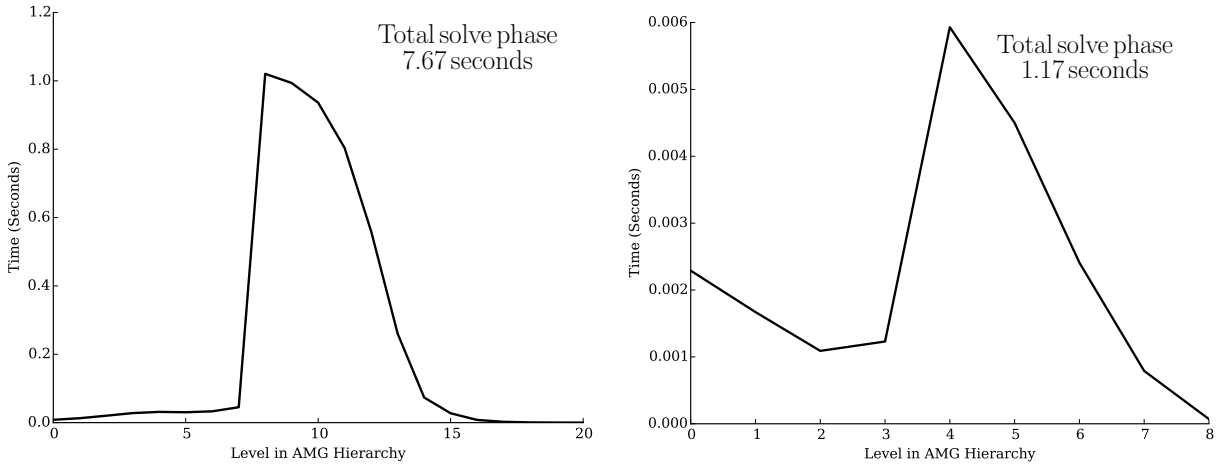


Figure 4.2: Left: Time spent on each level of the hierarchy during a single iteration of classical parallel AMG for a 3D Poisson problem with *hypre* using Falgout coarsening [43] and hybrid symmetric Gauss-Seidel relaxation. Right: Repeat experiment, but using aggressive HMIS coarsening. The total time is much lower; however, the qualitative feature of expensive coarse levels remains.

adaptive method which allows nonzero entries to be reintroduced into the AMG hierarchy, thus recovering convergence if entry elimination is too aggressive.

In the context of this work, we define *sparsity* and *density* in terms of the average number of nonzeros per row (or equivalently, the average degree of a node in the graph of the matrix). In particular, density of a matrix A_ℓ of size n_ℓ is defined to be $\texttt{nnz}(A_\ell)/n_\ell$. The performance of AMG is closely correlated with this metric, especially communication costs. In addition, note that if a matrix A_ℓ is "sparser" or "denser" under this definition, it is also the case under the more traditional density metric, $\texttt{nnz}(A_\ell)/n_\ell^2$. Another advantage is that this measure yields a meaningful comparison between matrices of different sizes. For example, a goal of our algorithm is to generate coarse matrices as close as possible in terms of sparsity structure to the fine grid matrix.

There are a number of existing approaches to reduce per-iteration communication complexity at the cost of convergence. Aggressive coarsening, such as HMIS [40] and PMIS [40], rapidly coarsens each level of the hierarchy, leading to a reduction in both the number and the density of coarse operators. While these coarsening strategies reduce the cost of each iteration or cycle in the AMG solve phase, they do so at the cost of accuracy, often resulting in reduced convergence. Likewise, interpolation schemes such as *distance-two interpolation* [44], improve the convergence for aggressive coarsening, but also result in an increase in complexity.

Figure 4.2 shows that the time per level during the solve phase is reduced in comparison

to standard coarsening, even though the same number of processes and problem size per core are used. The use of HMIS coarsening is the only difference in problem settings between these two runs. Regardless, while aggressive coarsening may reduce the total work required during an iteration of AMG, the problem of expensive coarse levels still persists.

Another strategy for reducing communication complexity in AMG consists of systematically improving sparsity in the interpolation operators [44]. Removing nonzeros from the interpolation operators reduces the complexity of the coarse-grid operators, however this process can also have an unpredictable impact on coarse-level performance if used too aggressively. Sparsity can alternatively be improved by considering a sparse approximation to the transformation that relates the fine-grid matrix A purely injected to the coarse grid with the less sparse but generally more desirable Galerkin coarse-grid matrix. This transformation along with the injected matrix A go on to form a coarse grid that is sparser than Galerkin but still yields good multigrid convergence for the several cases [45].

The typical approach to building coarse-grid operators, A_ℓ, is to form the Galerkin product with the interpolation operator: $A_{\ell+1} = P_\ell^T A_\ell P_\ell$. This ensures a *projection* in the coarse-grid correction process and a guarantee on the reduction in error in each iteration of the AMG solve phase (although the factor by which the error is reduced may be small for a poorly converging method). Yet it is the triple matrix product in the Galerkin construction that leads to a growth in the number of nonzeros in coarse-grid matrices. As such, there are several approaches to constructing coarse operators that do not use a Galerkin product and are termed *non-Galerkin* methods. These methods have been formed in a classical AMG setting [15] and also in a smoothed aggregation [46] context. In general, these methods selectively remove entries from coarse-grid operators, reducing the complexity of the multigrid cycle. Assuming the appropriate entries are removed from coarse-grid operators, the result is a reduction in complexity with little impact on convergence.

An alternative to limiting communication complexity is to directly determine the coarse-grid stencil, an approach used in geometric multigrid. For instance, simply rediscretizing the PDE on a coarse-level results in the same stencil pattern as for the original finest-grid operator, thus avoiding any increase in the number of nonzeros in coarse-grid matrices. More sophisticated approaches combine geometric and algebraic information and include BoxMG [47, 48] and PFMG [49], where a stencil-based coarse-grid operator is built. Additionally, collocation coarse grids (CCA) [50] have been used on coarse levels to effectively limit the number of nonzeros. Yet, all these methods rely on geometric properties of the problem being solved. One exception is the extension of collocation coarse grids to algebraic multigrid (ACCA) [51], which has shown similar performance to smoothed aggregation AMG.

Another approach is to consider sparsification of the graph of the sparse matrix. There

are several major approaches, grouped by how the difference between the original graph and the sparsified graph is measured. One major approach finds graphs where the weight of cuts in the original graph and the sparsified graph are close [52]. In another, called spectral sparsification [53, 5], edges of the graph are removed if the resulting graph Laplacian remains spectrally close to the original. Sparsification of graphs has been an active area of research recently [54], concentrating on developing *near* linear-time algorithms for the sparsification. While the immediate impact on the coarse-level matrices in a multigrid hierarchy has not been studied, this could point to an additional improvement to the methods presented in this work.

The approach developed in this work is to form a coarse-level operator that does not satisfy a Galerkin product by modifying *existing* hierarchies. The novel benefit of the proposed approach is that it is applicable to most AMG methods, requires no geometric information, and provides a mechanism for recovery if the dropping heuristic is chosen too aggressively (see Section 4.6). This chapter is outlined as follows. Section 4.2 describes the method introduced in [15]. Section 4.3 introduces two new methods for reducing the communication complexity of AMG: Sparse Galerkin and Hybrid Galerkin. Parallel performance models for these methods are described in Section 4.4, and the parallel results are displayed in Section 4.5. An adaptive method for controlling the trade-off between communication complexity and convergence is described in Section 4.6. Finally, Section 4.7 makes concluding remarks.

4.2 METHOD OF NON-GALERKIN COARSE GRIDS

In this section we introduce terminology related to [15], which we term the method of *non-Galerkin coarse grids* or non-Galerkin for the remainder of the chapter. In this case, coarse-level operators do not satisfy the Galerkin relationship where $A_{\ell+1} = P_\ell^T A_\ell P_\ell$ for each level ℓ. The coarse-grid operators are first formed through the Galerkin product, followed by a sparsification step that generates $\hat{A}_{\ell+1}$ — see the call to `sparsify` in Algorithm 2.1. As motivated in the previous section, fewer nonzeros in the coarse-grid operator reduce the communication requirements. The sparser matrix $\hat{A}_{\ell+1}$ replaces $A_{\ell+1}$ and is then used when forming the remaining levels of the hierarchy, creating a dependency between $\hat{A}_{\ell+1}$ and all successive levels as shown in Figures 4.5a and 4.5b. Thus, this approach does not preserve a coarse-grid correction corresponding to an A-orthogonal projection, as described in Section 2.2.

In the following we use $\texttt{edges}(A)$, for a sparse matrix A, to represent the set of edges in the graph of A. That is, $\texttt{edges}(A) = \{(i,j) \,\text{such that}\, A_{i,j} \neq 0\}$, where $A_{i,j} = (A)_{i,j}$ is the $(i,j)^{\text{th}}$ entry of A. In addition, we denote $\hat{P}_\ell$ as the *injection* interpolation operator that

Algorithm 4.1: `sparsify` from [15]

Input: A_c: coarse-grid operator
A: fine-grid operator
P: interpolation
$\hat{P}$: injection
S: classical strength matrix
γ: sparse dropping threshold parameter
Output: $\hat{A}_c$, a sparsified A_c

$\mathcal{M} = \texttt{edges}(\hat{P}^T A P + P^T A \hat{P})$ {Edges in the minimal sparsity pattern}
$\mathcal{N} = \emptyset$ {Edges to keep in A_c}
$\hat{A}_c = \mathbf{0}$ {Initialize sparsified A_c}

for $(A_c)_{i,j} \neq 0$ **do**
  if $(i,j) \in \mathcal{M}$ `or` $\left|(A_c)_{i,j}\right| \geq \gamma \max_{k \neq i} \left|(A_c)_{i,k}\right|$
    $\mathcal{N} \leftarrow \mathcal{N} \cup \{(i,j),(j,i)\}$ {Add strong edges or the required pattern}

for $(A_c)_{i,j} \neq 0$ **do**
  if $(i,j) \in \mathcal{N}$
    $(\hat{A}_c)_{i,j} = (A_c)_{i,j}$
  else
    $\mathcal{W} = \{k \,|\, S_{j,k} \neq 0, (i,k) \in \mathcal{N}\}$ {Find strong neighbors in the keep list}
    for $k \in \mathcal{W}$ **do**
      $\alpha = \frac{|S_{j,k}|}{\sum_{m \in \mathcal{W}} |S_{j,m}|}$ {Relative strength to k}
      $(\hat{A}_c)_{i,k} \leftarrow (\hat{A}_c)_{i,k} + \alpha (A_c)_{i,j}$
      $(\hat{A}_c)_{k,i} \leftarrow (\hat{A}_c)_{k,i} + \alpha (A_c)_{i,j}$
      $(\hat{A}_c)_{k,k} \leftarrow (\hat{A}_c)_{k,k} - \alpha (A_c)_{i,j}$

Algorithm 4.2: Diagonal Lumping – Alternative **for** loop (§ 4.3.1)

1 **for** $(A_c)_{i,j} \neq 0$ **do**
3 $\texttt{ismax} \leftarrow \left|(A_c)_{i,j}\right| = \max_{k \neq i} |(A_c)_{ik}|$ `and` $(i,k) \notin \mathcal{N} \, \forall \, k \neq i$ `and` $\sum_j A_{i,j} = 0$
4 **if** $(i,j) \in \mathcal{N}$ `or ismax` {Keep if entry is the single, maximum nonzero}
5 $(\hat{A}_c)_{i,j} = (A_c)_{i,j}$
  else {Otherwise add to the diagonal}
    $(\hat{A}_c)_{i,i} \leftarrow (\hat{A}_c)_{i,i} + (A_c)_{i,j}$

injects from level $\ell + 1$ to the C points on level ℓ so that $\hat{P}_\ell$ is defined as the identity over the coarse points, leaving $\hat{P}_\ell$ zero over the F-points.

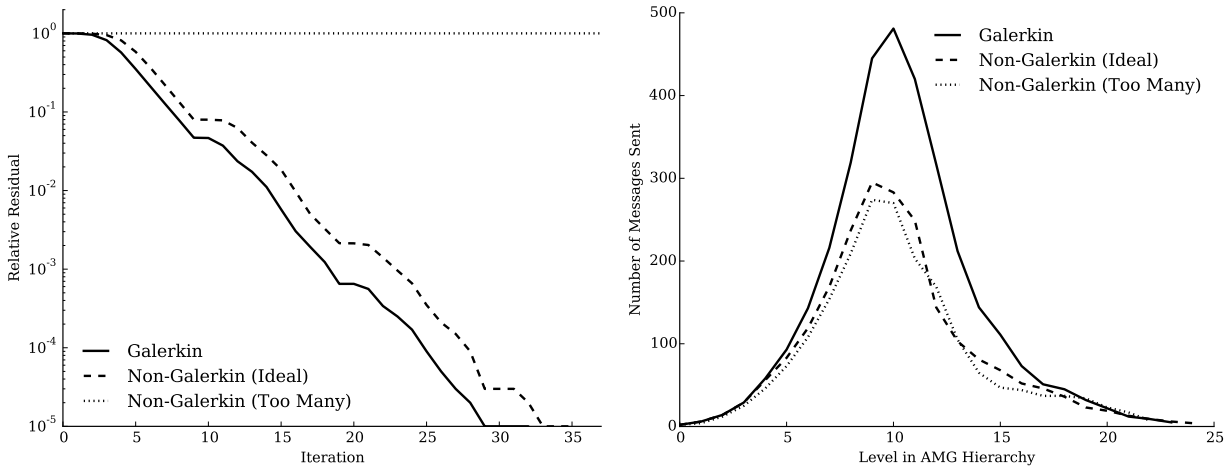
The `sparsify` method for reducing the nonzeros in a matrix is described in Algorithm 4.1, where the level subscripts are dropped for readability. The algorithm selectively removes small entries outside a minimal sparsity pattern[2] given by $\mathcal{M}_\ell$ where $\texttt{edges}(\mathcal{M}_\ell) = \texttt{edges}(\hat{P}_\ell^T A_\ell P_\ell + P_\ell^T A_\ell \hat{P}_\ell)$. For a given tolerance γ, any entry $A_{i,j}$ with $(i,j) \notin \mathcal{M}$ and $|A_{i,j}| < \gamma \max_{k \neq i} |A_{i,k}|$ is considered insignificant and is removed. When entry $A_{i,j}$ is removed, the value of $A_{i,j}$ is lumped to other entries that are strongly connected to $A_{i,j}$, and $A_{i,j}$ is set to zero. This reduces the per-iteration communication complexity and heuristically targets spectral equivalence between the sparsified operator and the Galerkin operator [15, 46].

There is a trade-off between the communication requirements and the convergence rate. Each entry in the matrix has a communication cost that is dependent on the number of network links that the corresponding message travels in addition to network contention. In addition, each entry in the matrix also influences convergence of AMG, with large entries generally having larger impact (although this is not uniformly the case). Any entry that has an associated communication cost outweighing the impact on convergence should be removed. However, while it is possible to predict this communication cost based on network topology and message size, the entry's contribution to convergence cannot be easily predetermined. When dropping via non-Galerkin coarse grids, if the chosen drop tolerance is too large, too many entries are removed and convergence deteriorates. Because the ideal drop tolerance is problem dependent and cannot be predetermined, it is likely that the chosen drop tolerance is suboptimal.

Figures 4.3a and 4.3b show the convergence and communication complexity, respectively, of various AMG hierarchies for solving a 3D Poisson problem with the method of non-Galerkin coarse grids. For both figures, the 27-point Laplacian was solved on 8192 processes with $10,000$ degrees-of-freedom per process. The original Galerkin hierarchy converges in the fewest number of iterations, but has the highest communication complexity. Non-Galerkin removes an ideal number of nonzeros from coarse-grid operators (labeled *ideal*) when no entries are removed from the first coarse level, and all successive levels have a drop tolerance of 1.0. In this case, the communication complexity of the solver is greatly reduced with little effect on convergence. However, if the first coarse level is also created with a drop tolerance of 1.0, essential entries are removed (labeled *too many*). While the complexity of

[2]The goal of the minimal sparsity pattern is to maintain, at the minimum, a stencil as wide for the coarse grid as exists for the fine grid. This is a critical heuristic for achieving spectral equivalence between the sparsified operator and the Galerkin operator. The current $\mathcal{M}$ achieves this in many cases. It is possible in some cases to reduce $\mathcal{M}$ further. See [15] for more details.

the hierarchy is further reduced, the method fails to converge.



(a) Relative residual per iteration

(b) Number of MPI sends per level

Figure 4.3: Sparsity is improved in the AMG hierarchy for the 27-Point Laplacian on 8192 processes with $10,000$ degrees-of-freedom per core. The time spent on middle levels of the AMG hierarchy is decreased (right) with little change to the residual after each iteration (left).

If a large drop tolerance is chosen in the non-Galerkin method, the effect on convergence can be determined after one or two iterations of the solve phase. At this point, if convergence is poor, eliminated entries can be re-introduced into the matrix. However, with this method, convergence improvements cannot be guaranteed. As shown in Algorithm 2.1, sparsifying on a level affects all coarser-grid operators. Hence, adding entries back into the original operator does not influence the impact of their removal on all coarser levels. Figure 4.4 shows how re-adding entries is ineffective by plotting the required communication costs verses the achieved convergence for both Galerkin and non-Galerkin AMG solve phases for the same 3D Poisson problem. The data set *Non-Galerkin (added back)* is generated by removing entries with a drop tolerance of 1.0 (everything outside of $\mathcal{M}$) on the first coarse-grid operator and 0.0 (retaining everything) on all successive levels. This results in a non-convergent method. We then add these removed entries back into the first coarse-grid operator, but this does not reintroduce the entries which were removed from coarser grid operators as a result of the non-Galerkin triple-matrix product $P_\ell^T \hat{A}_\ell P_\ell$. Figure 4.4 shows that this hierarchy requires little coarse-level communication after all entries have been reinstated to the first coarse-grid operator. However as the required entries are not added back into all coarser grid operators, the method still fails to converge.

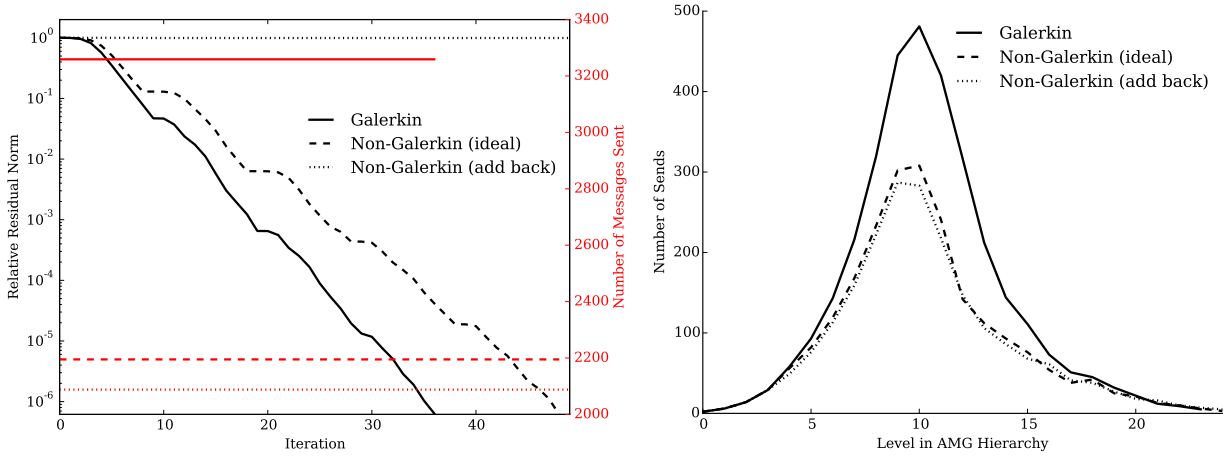


Figure 4.4: Convergence vs. communication of Galerkin and non-Galerkin hierarchies for the 27-point Poisson problem on 8192 processes, with 10,000 degrees-of-freedom per process. Relative residual per AMG iteration (black) vs the number of MPI sends per iteration (red) (left), and (maximum) number of sends per level in AMG hierarchy (right)

4.3 SPARSE AND HYBRID GALERKIN APPROACHES

In this section we present two methods as alternatives to the method of non-Galerkin coarse grids. The methods consist of forming the entire Galerkin hierarchy before sparsifying each operator, yielding a lossless approach for increasing sparsity in the AMG hierarchy. The first method, which is called the Sparse Galerkin method is described in Algorithm 4.3 (see Line 1). Sparse Galerkin creates the entire Galerkin hierarchy as usual. The hierarchy is then thinned as a post-processing step to remove relatively small entries outside of the minimal sparsity pattern $\mathcal{M} = \hat{P}^T AP + P^T A\hat{P}$ using `sparsify`.

The second method that we introduce is called Hybrid Galerkin since it combines elements of Galerkin and Sparse Galerkin to create the final hierarchy. The method is again lossless, and is outlined in Algorithm 4.3 (see Line 2). After the Galerkin hierarchy is formed, small entries outside are removed, this time using a modified, minimal sparsity pattern of $\mathcal{M} = \hat{P}^T \hat{A} P + P^T \hat{A} \hat{P}$.

The Sparse and Hybrid Galerkin methods retain the structure of the original Galerkin hierarchy. Consequently, these methods introduce error only into relaxation and residual calculations. The remaining components of each V-cycle in the solve phase (see `amg_solve`), such as restriction and interpolation are left unmodified. Therefore, the grid transfer operators do not depend on any sparsification, as shown in Figure 4.5. Here, we see that the Sparse Galerkin method does not use the modified (or sparsified) operators to create the next coarse-grid operator in the hierarchy. Conversely, Hybrid Galerkin uses the newly modified

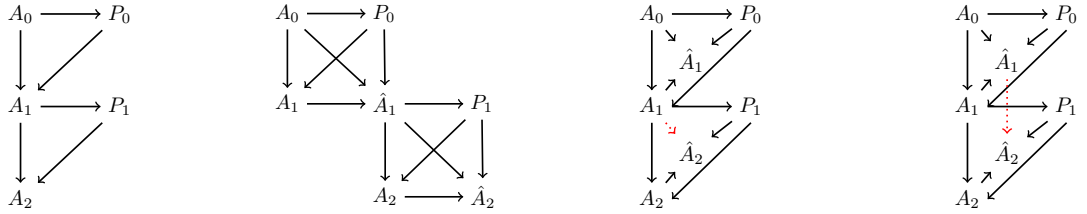
Algorithm 4.3: `sparse_hybrid_setup`

Input: A_0: fine-grid operator
`max_size`: threshold for max size of coarsest problem
$\gamma_1, \gamma_2, \ldots$ drop tolerances for each level
`sparse_galerkin`: Sparse Galerkin method
`hybrid_galerkin`: Hybrid Galerkin method

Output: $\hat{A}_1, \ldots, \hat{A}_L$

$A_1, \ldots, A_L, P_0, \ldots, P_{L-1} = \texttt{amg\_setup}(A_0, \texttt{max\_size}, \texttt{False})$
$\hat{A}_0 = A_0$
for $\ell \leftarrow 1$ **to** L **do**
  if `sparse_galerkin`
1 $\hat{A}_{\ell+1} = \texttt{sparsify}(A_{\ell+1}, A_\ell, P_\ell, S_\ell, \gamma_\ell)$ {Increase using the Sparse Method}
  else if `hybrid_galerkin`
2 $\hat{A}_{\ell+1} = \texttt{sparsify}(A_{\ell+1}, \hat{A}_\ell, P_\ell, S_\ell, \gamma_\ell)$ {Increase using the Hybrid Method}

operator to compute the sparsity pattern $\mathcal{M}$ for the next coarse-grid operator.



(a) Galerkin (b) Non-Galerkin (c) Sparse Galerkin (d) Hybrid Galerkin

Figure 4.5: Dependencies for forming each operator in the various AMG hierarchies. The difference between Sparse and Hybrid Galerkin dependencies is highlighted in red.

The new Sparse Galerkin and Hybrid Galerkin methods reduce the per-iteration cost in the AMG solve cycle as less communication is required by each sparse, coarse-grid operator. However, high-energy error may also be relaxed at a slower rate, yielding a reduction in the convergence factor. As a result, the solve phase is more efficient when the reduction in communication outweighs the change in convergence factor.

Similar to the method of non-Galerkin, it is difficult to predict the impact of removing entries from A_c in Algorithm 4.1 on the relaxation process. However, as the structure of the Galerkin hierarchy is retained, the convergence factor of the solve phase can be controlled on-the-fly. In our approach, differences between A_c and $\hat{A}_c$ are stored while forming the sparse approximations. Subsequently, if the convergence factor falls below a tolerance, entries can be reintroduced into the hierarchy, allowing improvement of the convergence factor up to

that of the original Galerkin hierarchy (see Section 4.6).

4.3.1 Diagonal Lumping

A significant amount of work is required in Algorithm 4.1 to improve the sparsity of each coarse operator. When forming non-Galerkin coarse grids, this additional setup cost is hidden by the reduced cost of the triple matrix product with the sparsified matrix $P^T \hat{A} P$. However, as the entire Galerkin hierarchy is initially formed as usual in our new methods (Algorithm 4.3) the additional work greatly reduces the scalability of the setup phase, as shown in Section 4.5.2. This significant cost suggests using an alternative method for sparsification of coarse-grid operators. When reducing the number of nonzeros from coarse-grid operators with Sparse Galerkin or with Hybrid Galerkin, the structure of the Galerkin hierarchy remains intact, allowing a more flexible treatment of increasing sparsity in the matrix. For instance, one option is to remove entries by lumping to the diagonal rather than strong neighbors, as described in Algorithm 4.1b. This variation of `sparsify` is beneficial for several reasons, including: a much cheaper setup phase when compared to Algorithm 4.1; potential to reduce the cost of the solve phase; reduced storage constraints for adaptive solve phases (see Algorithm 4.4); and retaining positive-definiteness of coarse operators.

Algorithm 4.1b replaces the **for** loop in Algorithm 4.1. For each nonzero entry in the matrix, the algorithm first checks if the entry is the maximum element in the row and if all other entries in the row are selected for removal (see Line 3). In this case, the nonzero entry is not removed if there is a zero row sum.

The method of diagonal lumping (Algorithm 4.1b) results in a cheaper setup phase than Algorithm 4.1. The original non-Galerkin `sparsify` requires each removed entry to be symmetrically lumped to significant neighbors. As a result, the process of calculating the associated strong connections requires a large amount of computation through a costly loop over neighbors of neighbors. Furthermore, to maintain symmetry, all matrix entries that are not stored locally must be updated, requiring a significant amount of interprocessor communication. Lumping these entries to the diagonal eliminates both the computational and communication complexities.

Eliminating the requirement of lumping to strong neighbors yields potential for removing a larger number of entries from the hierarchy, further reducing the communication costs of the solve phase. The original version of Algorithm 4.1 requires that an entry must have strong neighbors to be removed, as its value is lumped to these neighbors.

While relaxing the restrictions of the original non-Galerkin `sparsify` provides more opportunity to remove entries, the diagonal lumping also negatively influences convergence in

some cases. Indeed, the energy of the operator can be increased as a result of the diagonal lumping, leading to a decrease in (spectral) equivalence with the original operator. To mitigate this, if convergence suffers, entries can be easily reintroduced into the hierarchy, improving convergence during the solve phase.[3] As removed entries are only added to the diagonal, the storage of both the sparse matrix along with removed entries is minimal. In addition, these entries can be restored by inserting their values to the original positions, and subtracting these values from the associated diagonal entries as shown in Algorithm 4.4. The process of reintroducing these entries requires no interprocessor communication as well as a low amount of local computational work.

Diagonal lumping also preserves matrix properties such as symmetric positive-definiteness (SPD). As described in the following theorem, if the sparsity of a diagonally dominant SPD matrix is increased using diagonal lumping, the resulting matrix remains SPD. Consequently, Sparse and Hybrid Galerkin with diagonal lumping can be used in preconditioning many methods such as conjugate gradient. It is important to note, that while SPD matrices are an attractive property for AMG, AMG methods do not guarantee diagonal dominance of the coarse-grid operators. Yet, in many instances this property is preserved, for example for more standard elliptic operators.

Theorem 4.1. *Let A be SPD and diagonally dominant. If $\hat{A}$ is produced by Algorithm 4.1b, then it is symmetric positive semi-definite and diagonally dominant.*

Proof. Let A be SPD with diagonal dominance,

$$|A_{i,i}| \geq \sum_{k \neq i} |A_{i,k}|, \forall i. \tag{4.2}$$

Symmetry of $\hat{A}$ is guaranteed from the symmetry of both A and the $\mathcal{N}$ from Algorithm 4.1. For all off-diagonal entries $(i,j),(j,i) \in \mathcal{N}$,

$$\hat{A}_{i,j} = A_{i,j} = A_{j,i} = \hat{A}_{j,i}, \tag{4.3}$$

by Line 5 in Algorithm 4.1b and the symmetry of A.

The positive-definiteness is guaranteed by the diagonal dominance and a Gershgorin disc argument. The proof proceeds by starting with the matrix A and then considering the

[3]Another mitigating factor occurs when diagonal lumping is done after the construction of the hierarchy, when $\hat{A}_\ell$ will be used only for relaxation. At this point, $\hat{A}_\ell$ must only effectively damp high energy modes and leave low energy modes largely untouched. Since diagonal lumping of relatively small entries (as controlled by γ) largely impacts spectral equivalence for low energy modes, this mitigates any effect from reduced spectral equivalence.

change made to A by the elimination of each entry. Initially, all the Gershgorin discs of A are strictly on the right-side of the origin, thus implying that all eigenvalues are non-negative. Then, assume that we eliminate some arbitrary entry $A_{i,j}$, $(i,j) \in \mathcal{N}$. This results in row i being updated

$$A_{i,i} \leftarrow A_{i,i} + A_{i,j} \quad \text{and} \quad A_{i,j} \leftarrow 0 \tag{4.4}$$

If $A_{i,j} > 0$, then the center of the Gershgorin disc is shifted to the right, and the radius shrinks, thus keeping the disc to the right of the origin and preserving definiteness. If $A_{i,j} < 0$, then the center of the disc is shifted to the left by $|A_{i,j}|$, but the radius of the disc also shrinks by $|A_{i,j}|$. This also keeps the disc to the right of the origin and preserves semi-definiteness. Furthermore, since each disc is never shifted to the left half plane, diagonal dominance is also preserved. The proof then proceeds by considering all of the entries to be eliminated. □

Remark 4.1. *If any row of A is strictly diagonally dominant, as often happens with Dirichlet boundary conditions, then $\hat{A}$ will be positive definite. Essentially, Algorithm 4.1b never shifts a Gershgorin disc to the left, so $\hat{A}$ can have no 0 eigenvalue.*

4.4 PARALLEL PERFORMANCE

In this section we use a parallel performance model to illustrate the per-level costs associated with each of the six methods:

Galerkin - Classic coarsening in AMG, as outlined in Algorithm 2.1;

Non-Galerkin - The base algorithm presented in [15];

Sparse Galerkin - The new Algorithm 4.3 with `sparse_galerkin` and full lumping from Algorithm 4.1;

Sparse Galerkin (Diag) - The new Algorithm 4.3 with `sparse_galerkin` and diagonal lumping from Algorithm 4.1b;

Hybrid Galerkin - The new Algorithm 4.3 with `hybrid_galerkin` and full lumping from Algorithm 4.1; and

Hybrid Galerkin (Diag) - The new Algorithm 4.3 with `hybrid_galerkin` and diagonal lumping from Algorithm 4.1b.

The solve phase of AMG (see Algorithm 2.2) is largely comprised of sparse matrix-vector multiplication, thus we model each method by assessing the cost of performing a SpMV on each level of the hierarchy. We focus on the operators A_ℓ, as the work required for this matrix is more costly than the restriction and interpolation operations. Specifically, we employ an α–β model to capture the cost of the parallel SpMV based on the number of nonzeros in A. We denote p as the number of processors, α as the latency or startup cost of a message, and β as the reciprocal of the network bandwidth [16, 55]. In addition, $\texttt{nnz}_p$ represents the average number of nonzeros local to a process, while s_p and n_p are the maximum number of MPI sends and message size across all processors. Finally, we use c to represent the cost of a single floating-point operation. With this we model the total time as

$$T = 2\,c\,\texttt{nnz}_p + \max_p s_p(\alpha + \beta n_p). \tag{4.5}$$

For the model parameters above we use the Blue Waters supercomputer at the University of Illinois at Urbana-Champaign [56, 57]. The latency and bandwidth were measured through the HPCC benchmark [58], yielding $\alpha = 1.8 \times 10^{-6}$ and $\beta = 1.8 \times 10^{-9}$. Since the achieved floprate depends on matrix size, we determine the value of c by timing the local SpMV. Specifically, letting $\texttt{nnz}_{\text{local}}$ be the number of nonzeros local to the processor and T_{local} the time to perform the local portion of the SpMV, we compute $c = {}^{T_{\text{local}}}\!/_{2\texttt{nnz}_{\text{local}}}$ for each matrix in the hierarchy.

The minimal per-level cost associated with the non-Galerkin and Sparse/Hybrid Galerkin methods occurs when entries are removed with a drop tolerance of $\gamma = 1.0$. Using the model, (4.5), this is highlighted in Figure 4.6 for both the Laplace and rotated anisotropic diffusion problems (a full description of these problems is given in Section 4.5). We see that both non-Galerkin and Hybrid Galerkin have potential to minimize the per-level cost. However, when the per-level cost is minimized, the convergence of AMG often suffers. Therefore, less-aggressive drop tolerances such as $\gamma < 1.0$ may remove fewer entries, increasing the per-level cost, but due to better convergence will improve the overall cost of the solve phase. Indeed for the rotated anisotropic diffusion problem, this is the case, where we reduce γ at fine levels in the hierarchy in order to retain convergence (not shown for brevity).

4.5 PARALLEL RESULTS FOR SPARSE AND HYBRID GALERKIN

In this section we highlight the parallel performance of the Sparse and Hybrid Galerkin methods. We consider scaling tests on the familiar 3D Laplacian since this is a common multigrid problem used to establish a baseline. In order to test problems where AMG

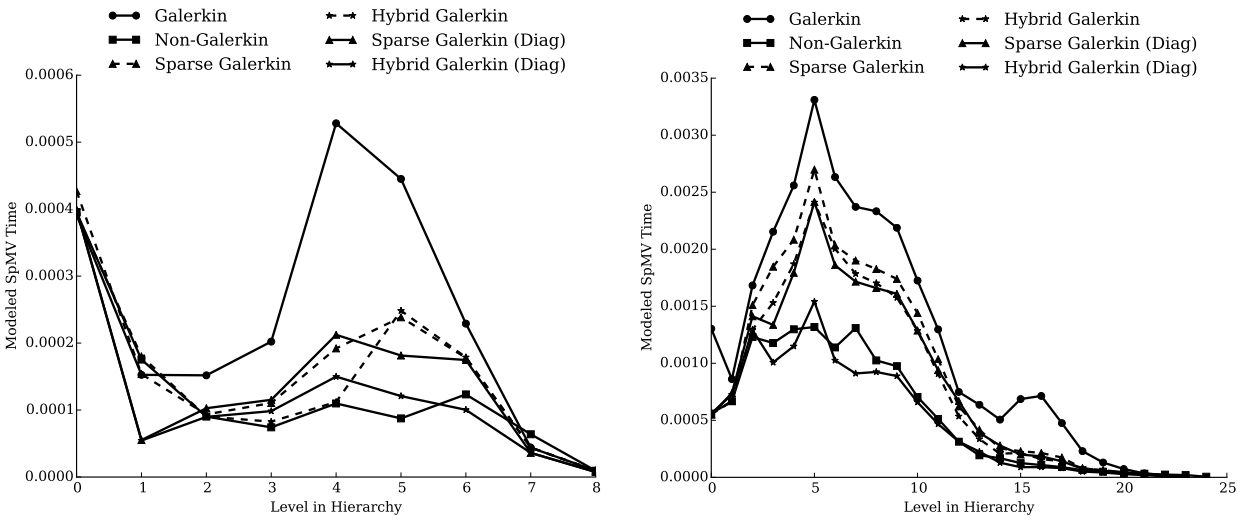


Figure 4.6: Modeled minimal cost of a single SpMV on each level of the AMG hierarchy for Laplace (left) and rotated anisotropic diffusion (right), for an aggressive drop tolerance of 1.0 on each level.

convergence is suboptimal, we consider a 2D rotated anisotropic diffusion problem. Finally, we test our methods on a suite of matrices from the Florida Sparse Matrix Collection. All computations were performed on the Blue Waters system at the University of Illinois at Urbana-Champaign [56]. Each method was implemented and solved with *hypre* [38, 59], using default parameters unless otherwise specified. In summary, we compare the solve and setup times for the six methods considered in Section 4.4 while preconditioning a Krylov method such as CG or GMRES in each test.

The drop tolerances for each method vary by level, using a combination of 0.0, 0.01, 0.1, and 1.0 across the coarse levels. Six combinations of these drop tolerances are tested for the various test cases, and the series yielding the minimum solve time for each is selected. *Note:* At $100,000$ cores, the best drop tolerances from the second largest run size are used due to large costs associated with running 6 drop tolerances at this core count. Details of the drop tolerances used in all the below tests are found in the Supplemental Materials due to length.[4]

Generally, the drop tolerances are aggressive for the simple isotropic 3D Laplacian example, where for instance in Figures 4.7 and 4.8, the Sparse Galerkin (Diag) method used the values of $[0.0, 0.1]$, i.e., no dropping on the first coarse level and then 0.1 on all subsequent levels. For the more complex examples such as rotated anisotropic diffusion, the

[4]In addition, the modifications to *hypre* `v2.11.0` are stored at `https://github.com/lukeolson/hypre/releases/tag/SISC-sparse-hybrid-galerkin` and `https://github.com/lukeolson/hypre/releases/tag/SISC-sparse-hybrid-galerkin-adaptive`.

drop tolerances are less aggressive and usually begin on coarser levels. For instance in Figures 4.7 and 4.8, the Sparse Galerkin (Diag) method used values of $[0.0, 0.0, 0.0, 0.1]$, i.e., no dropping occurs until the third coarse level, where 0.1 is used from that point onward on all coarser levels. Overall, the drop tolerance is similar to other multigrid parameters, such as the strength-of-connection drop tolerance. Some experimentation with a problem type is required, but thereafter a general, conservative parameter choice can be made for subsequent use.

We consider the diffusion problem

$$-\nabla \cdot K \nabla u = 0, \tag{4.6}$$

with two particular test cases for our simulations. Also, as problems with less structure result in increased density on coarse levels, we consider a subset from the Florida sparse matrix collection. The test problems are as follows:

Laplace - Here, we use $K = I$ on the unit cube with homogeneous Dirichlet boundary conditions. Q1 finite elements are used to discretize the problem using a uniform mesh, leading to a familiar 27-point stencil. The preconditioner formed for the 3D Laplacian uses aggressive coarsening (HMIS) and distance-two (extended classical modified) interpolation. The interpolation operators were formed with a maximum of five elements per row, and hybrid symmetric Gauss-Seidel was the relaxation method.

Rotated Anisotropic Diffusion - In this case, we consider a diffusion tensor with homogeneous Dirichlet boundary conditions of the form $K = Q^T D Q$, where Q is a rotation matrix and D is a diagonal scaling defined as

$$Q = \begin{pmatrix} \cos(\theta) & \sin(\theta) \\ -\sin(\theta) & \cos(\theta) \end{pmatrix} \qquad D = \begin{pmatrix} 1 & 0 \\ 0 & \epsilon \end{pmatrix}. \tag{4.7}$$

Q1 finite elements are used to discretize a uniform, square mesh. In the following tests we use $\theta = \frac{\pi}{8}$ and $\epsilon = 0.001$. In each case, the preconditioner uses Falgout coarsening [43], extended classical modified interpolation and hybrid symmetric Gauss-Seidel.

Florida Sparse Matrix Collection - We consider a subset of all real, symmetric, positive definite matrices from the Florida sparse matrix collection with size over 1,000,000 degrees-of-freedom. In addition we consider only the cases where GMRES preconditioned with Galerkin AMG converges in fewer than 100 iterations. Each problem uses HMIS coarsening and so-called *extended+i* interpolation if possible. In some cases, however,

Galerkin AMG does not converge with these options; in these cases Falgout coarsening and modified classical interpolation are used. Relaxation for all systems is hybrid symmetric Gauss-Seidel. *Note:* When necessary for convergence, some *hypre* parameters, such as the minimum coarse-grid size and strength tolerance, vary from the default.

The following results demonstrate that the diagonally lumped Sparse and Hybrid Galerkin methods are able to perform comparably to non-Galerkin. Non-Galerkin and Sparse/Hybrid Galerkin all significantly reduce the per-iteration cost by reducing communication on coarse levels. Since the method of non-Galerkin is multiplicative in construction, the setup times are often much lower in comparison to standard Galerkin. However, Sparse and Hybrid do not observe this benefit since they are constructed in a post-processing step. While the per-iteration work is decreased for all methods, the convergence suffers for the case of rotated anisotropic diffusion problems with non-Galerkin at large scales. However, Sparse and Hybrid Galerkin converge at rates similar to the original Galerkin hierarchy, yielding speedup in total solve times. Moreover, in a strong scaling study, we observe that Hybrid Galerkin is competitive, particularly at large core counts.

4.5.1 Improving sparsity in AMG Hierarchies

The significant number of nonzeros on coarse levels creates large, relatively dense matrices near the middle of the AMG hierarchy, yielding large communication costs for each SpMV performed on these levels. As the solve phase of AMG consists of many SpMVs on each level of the hierarchy, the time spent on coarse levels can increase dramatically. Sparse, Hybrid, and non-Galerkin can all reduce both the cost associated with communication as well as the time spent on each level during a solve phase.

Figure 4.7 shows the time spent on each level of the hierarchy during a single iteration of AMG, for both test cases with 10,000 degrees-of-freedom per core using 8192 cores. Both the method of non-Galerkin coarse grids, as well as the Sparse and Hybrid Galerkin methods, reduce the time required on levels near the middle of the hierarchy. Non-Galerkin has a larger impact on the time spent on middle levels of the hierarchy for the Laplace problem than Sparse and Hybrid Galerkin. However, for the anisotropic problem, diagonally-lumped Hybrid Galerkin reduces level-time similarly to non-Galerkin. This is due to a large reduction in the number of messages required in each SpMV as shown in Figure 4.8. The reduction in total size of all messages communicated is relatively small.

The increase in time spent on each level, as well as the associated communication costs of these levels, becomes more pronounced at higher processor counts in a strong scaling

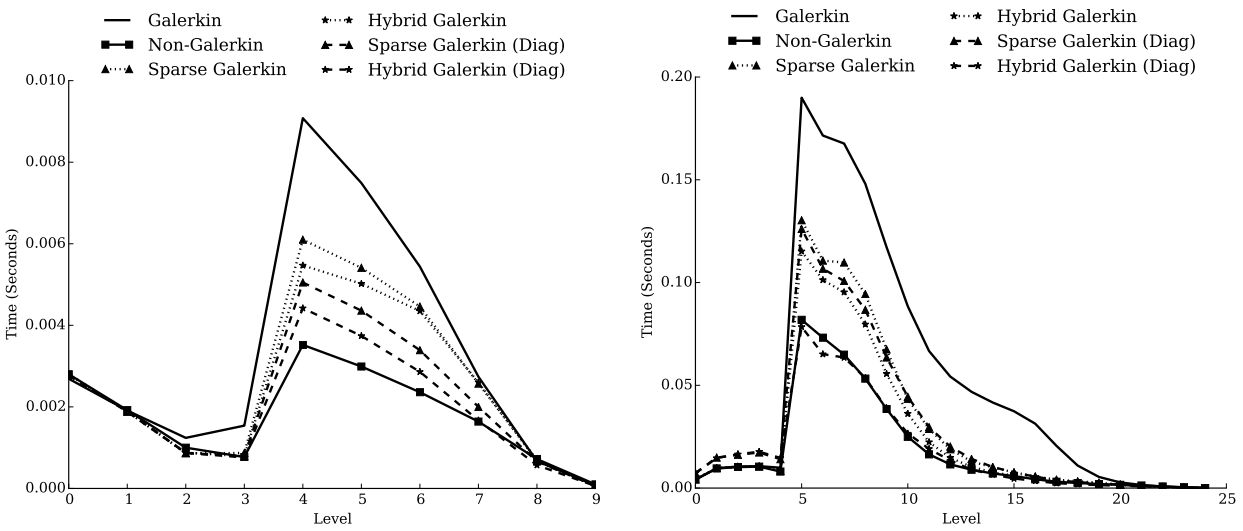


Figure 4.7: Time spent on each level of the AMG hierarchy during a single iteration of the solve phase for **Laplace (left)** and **Rotated Anisotropic Diffusion (right)**, each with 10,000 degrees-of-freedom per core.

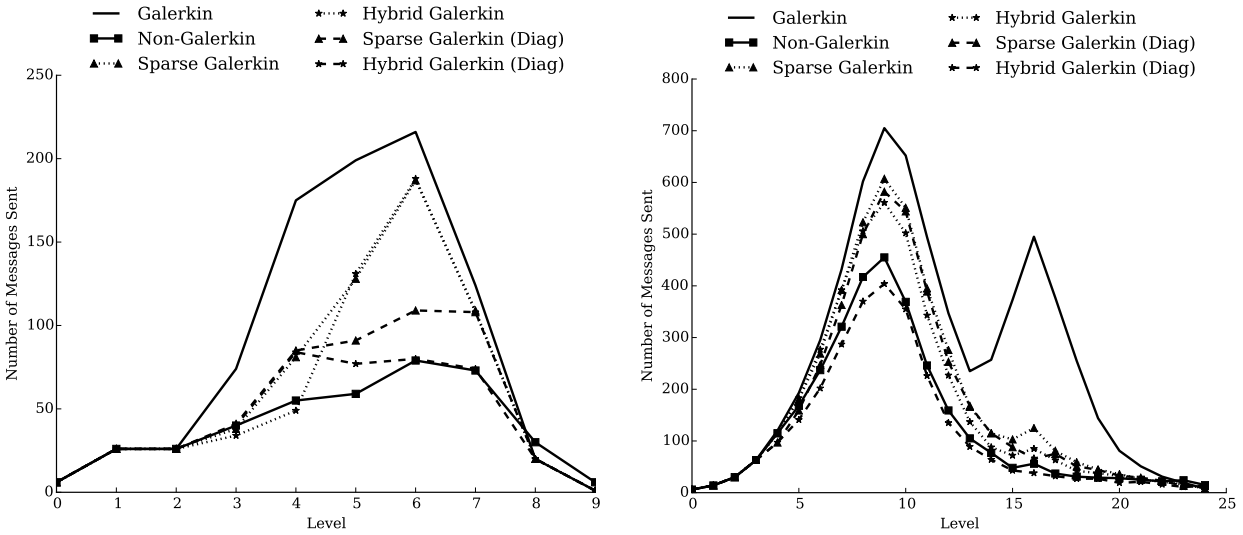


Figure 4.8: Number of sends required to perform a single SpMV on each level of the AMG hierarchy for: **Laplace (left)** and **Rotated Anisotropic Diffusion (right)**, each with 10,000 degrees-of-freedom per core.

study. Figure 4.9 illustrates this by plotting the per-level times required during a single iteration of AMG, as well as the number of messages communicated during a SpMV for the rotated anisotropic diffusion problem with 1,250 degrees-of-freedom per core using 8192 cores. Compared with the 10,000 degrees-of-freedom per core in Figures 4.7 and 4.8 there is a sharper increase in time required for levels near the middle of the hierarchy due to the

increasing dominance of communication complexity.

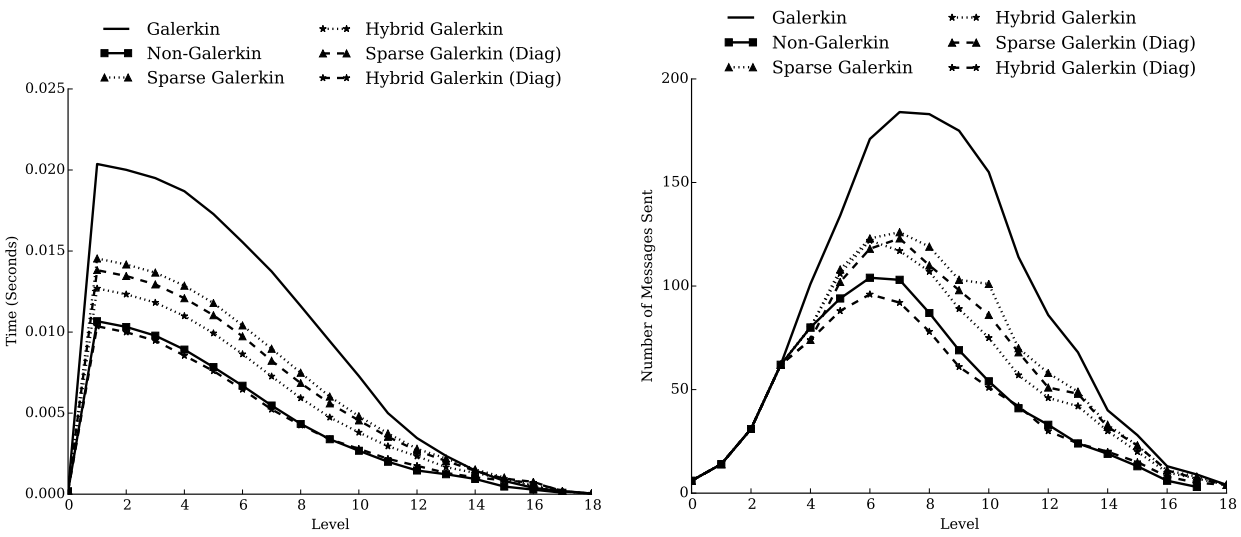


Figure 4.9: For each level of the AMG hierarchy, time per iteration of AMG (left) and number of messages sent during a single SpMV (right) for the **Rotated Anisotropic Diffusion** problem with 1,250 degrees-of-freedom per core.

4.5.2 Costs of Weakly Scaled Setup Phases

Each sparsification method can lead to reduced communication costs in the middle of the hierarchy. However, removing insignificant entries from coarse-grid operators requires additional work in the setup phase. In the non-Galerkin method, setup times are reduced since the increased sparsity is used directly in the triple-matrix product required to form each successive coarse-grid operator. However, for the new methods, Sparse and Hybrid Galerkin, the entire Galerkin hierarchy is first constructed so that the sparsify process on each level requires additional work. Figure 4.10 shows the times required to setup an AMG hierarchy for rotated anisotropic diffusion, with Laplace setup times scaling in a similar manner. While there is a slight increase in setup cost associated with the Sparse and Hybrid Galerkin hierarchies, this extra work is nominal. Therefore, while the majority of this additional work is removed when using diagonal lumping, the differences in work required in the setup phase between these two lumping strategies is insignificant for the problems being tested.

4.5.3 Weak Scaling of GMRES Preconditioned by AMG

In this section we investigate the *weak* scaling properties of the methods. Figure 4.11 shows both the average convergence factor and total time spent in the solve phase for a weak scaling study with rotated anisotropic diffusion problems at 10,000 degrees-of-freedom per core using GMRES preconditioned by AMG. GMRES is used over CG because Algorithm 4.1 guarantees symmetry but not positive-definiteness of the preconditioner. In many cases, positive-definiteness is preserved, but when using more aggressive drop tolerances, we have observed this property being lost. While the convergence of both diagonally-lumped Sparse and Hybrid Galerkin remain similar to that of Galerkin, the non-Galerkin method converges more slowly. Therefore, while non-Galerkin and diagonally-lumped Hybrid Galerkin yield similar communication requirements, GMRES preconditioned by Hybrid Galerkin performs significantly better as fewer iterations are required.

Remark 4.2. *With the chosen drop tolerances, non-Galerkin does not converge for this anisotropic problem at* $100,000$ *cores. In this case, nothing was dropped from the first three coarse levels of the hierarchy. On the fourth coarse level a drop tolerance of* 0.01 *was used, and the fifth was sparsified with a tolerance of* 0.1*. The remaining levels were sparsified with a drop tolerance of* 1.0*. This was determined to be the best tested drop tolerance sequence for smaller run sizes, and multiple drop tolerance sequences were not tuned at this large problem size due to the significant costs. However, a better drop tolerance could yield a convergent non-Galerkin method at this scale.*

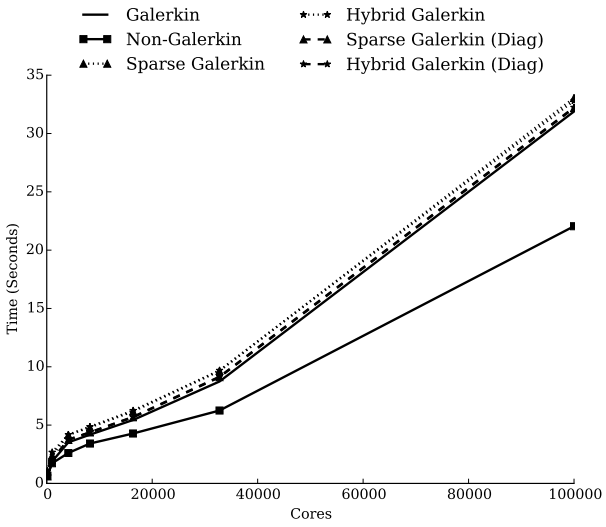


Figure 4.10: Time required to setup AMG hierarchy for **Rotated Anisotropic Diffusion** with 10,000 degrees-of-freedom per core.

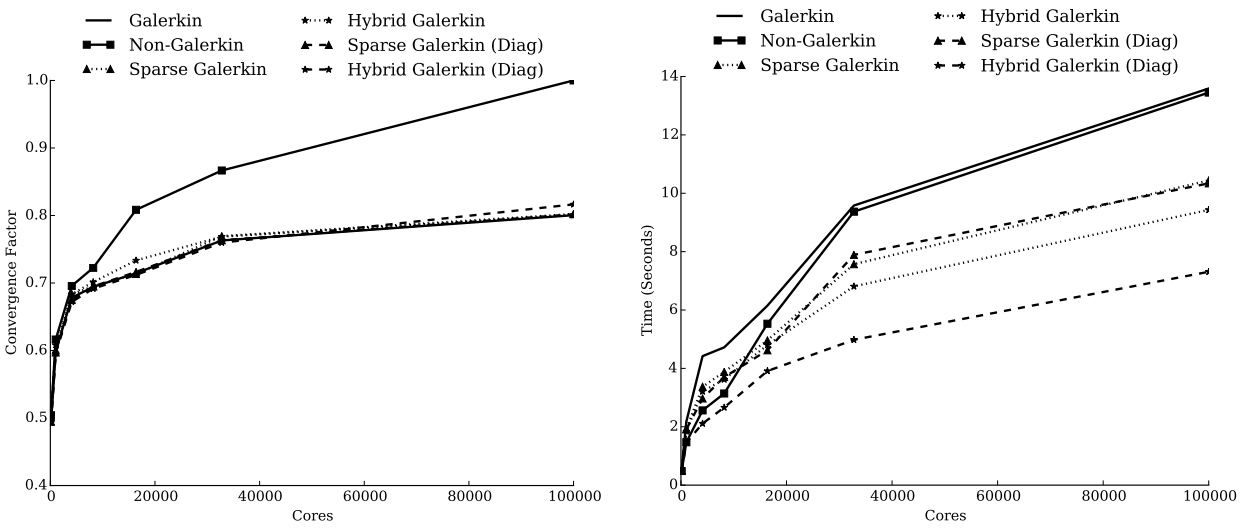


Figure 4.11: Convergence factors (left) and times (right) for weak scaling of **Rotated Anisotropic Diffusion** (10,000 degrees-of-freedom per core), solved by preconditioned GMRES. For large problem sizes, non-Galerkin AMG does not converge, and timings indicate when the maximum iteration count was reached.

The efficiency of weakly scaling to p processes is defined as $E_p = \frac{T_1}{T_p}$, where T_1 is the time required to solve the problem on a single process and T_p is the time to solve on p processes. The efficiency of solving weakly scaled rotated anisotropic diffusion problems with non-Galerkin, Sparse Galerkin, and Hybrid Galerkin, relative to the efficiency of Galerkin AMG, are shown in Figure 4.12. While both the original and diagonally-lumped Sparse and Hybrid Galerkin methods scale more efficiently than Galerkin, the poor convergence of non-Galerkin on large run sizes yields a reduction in relative efficiency.

4.5.4 Strong Scaling of GMRES Preconditioned by AMG

We next consider the rotated anisotropic diffusion system with approximately $10,240,000$ unknowns using cores ranging from 128 to $100,000$. Therefore, the simulation is reduced from $80,000$ degrees-of-freedom per core when run on 128 cores, to just over 100 degrees-of-freedom per core on $100,000$ cores. Computation dominates the total cost of solving a problem partitioned over relatively few processes, as each process has a large amount of local work. However, as the problem is distributed across an increasing number of processes, the local work decreases while communication requirements increase. Therefore, the time required to solve a problem is reduced with strong scaling, but only to the point where communication complexity begins to dominate. The efficiency of strongly scaling to p processes is defined as $E_p = \frac{T_1}{pT_p}$. Figure 4.13 shows the efficiency of solving a strongly scaled rotated anisotropic

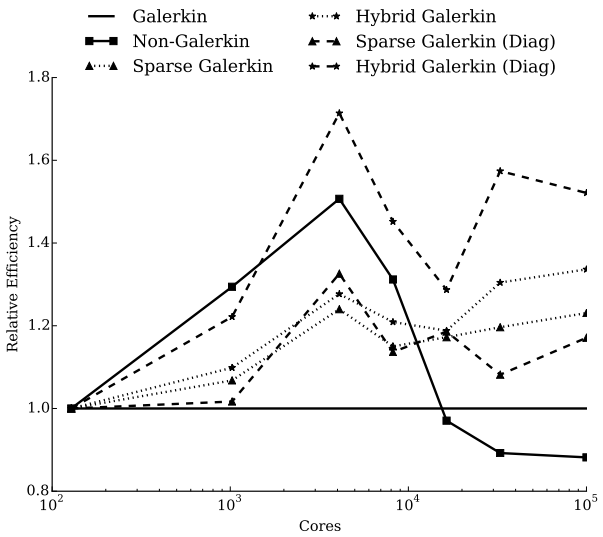


Figure 4.12: Efficiency of solving weakly scaled **Rotated Anisotropic Diffusion** at 10,000 degrees-of-freedom per core with various methods, **relative to that of the Galerkin hierarchy**.

diffusion problem with GMRES preconditioned by the various sparse methods. In each case we observe improvements over standard Galerkin.

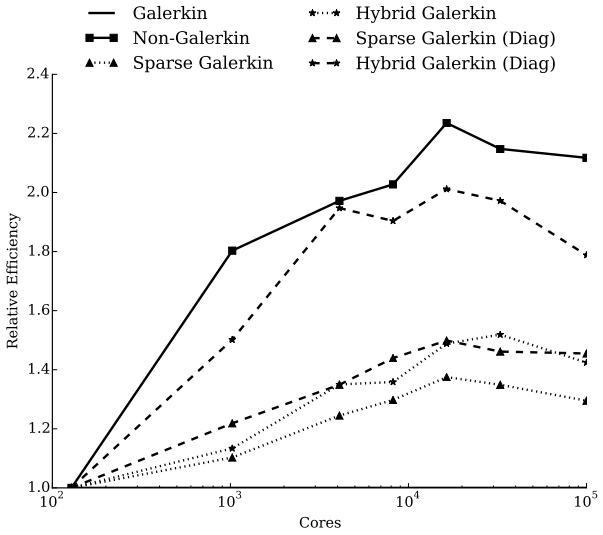


Figure 4.13: Efficiency of non-Galerkin and Sparse/Hybrid Galerkin methods in a strong scaling study, **relative to Galerkin AMG** for **Rotated Anisotropic Diffusion**.

A strong scaling study is also performed on the subset of matrices from the Florida sparse matrix collection. These problems were tested on 64, 128, 256, and 512 processes. Figure 4.14 shows the time required to perform a single V-cycle for each of the matrices in the subset,

relative to the time required by Galerkin AMG. All methods reduce the per-iteration times for each matrix in the subset. Furthermore, the total time required to solve each of these matrices is also reduced, as shown in Figure 4.15. While Sparse Galerkin provides some improvement, the Hybrid and non-Galerkin methods are comparable, particularly at high core counts.

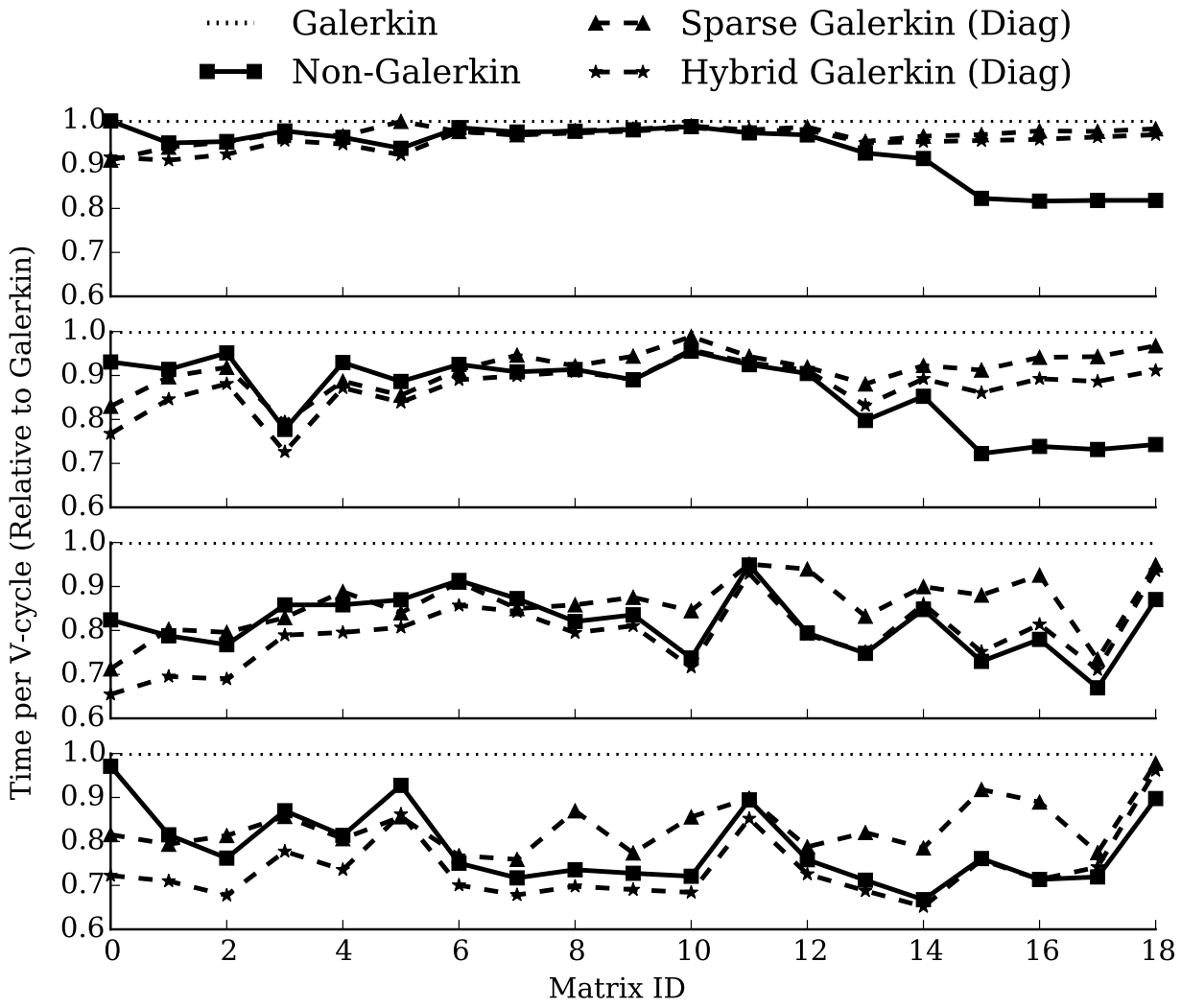


Figure 4.14: Time (relative to Galerkin) per iteration for each matrix in the Florida Sparse Matrix Collection, using $p = 64$, 128, 256, and 512.

4.5.5 Diagonal Lumping Alternative and PCG

Diagonal lumping retains positive definiteness of diagonally-dominant coarse-grid operators as described in Theorem 4.1. Therefore, as the preconditioned conjugate gradient (PCG)

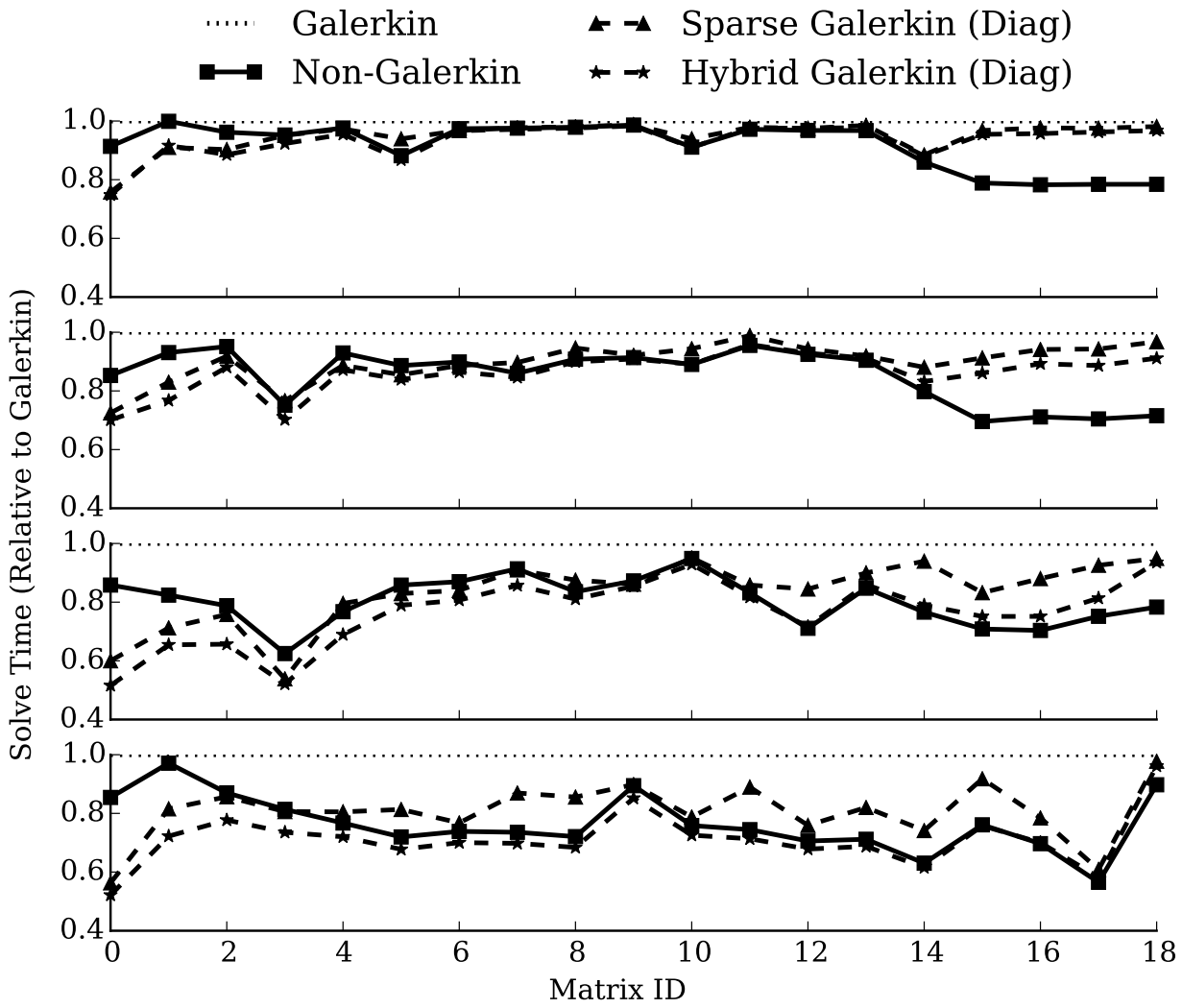


Figure 4.15: Time (relative to Galerkin) per AMG solve for each matrix in the Florida Sparse Matrix Collection, using $p = 64$, 128, 256, and 512.

method requires both the matrix and preconditioner to be symmetric and positive-definite, the Laplace and anisotropic diffusion problems are solved by conjugate gradient preconditioned by the diagonally-lumped Sparse and Hybrid Galerkin hierarchies. Figure 4.16 shows the solve phase times for solving the weakly scaled rotated anisotropic diffusion problem with PCG. As with GMRES, both the Sparse and Hybrid Galerkin preconditioners decrease the time required in the AMG solve phase during a weak scaling study.

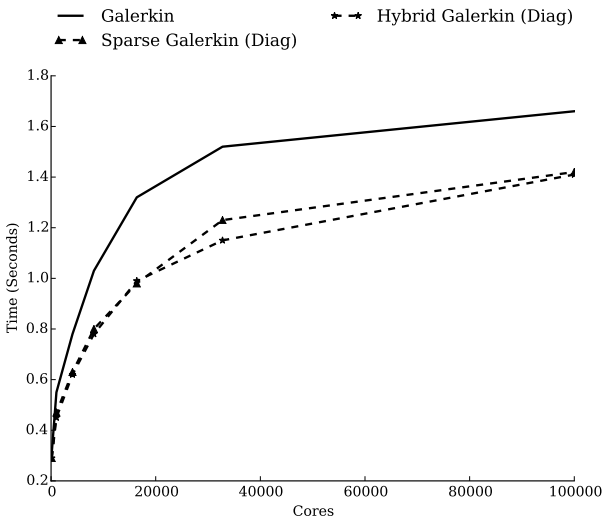


Figure 4.16: Weak scaling solve time for **Rotated Anisotropic Diffusion**, solved by PCG preconditioned by various AMG hierarchies.

4.6 ADAPTIVE SOLVE PHASE

The previous results describe the case where good drop tolerances were known a priori for `sparsify`. However, as the appropriate drop tolerance changes with problem type, problem size, and even level of the AMG hierarchy, a good drop tolerance is often not easily realized. When the drop tolerance is too small, few entries are removed from the hierarchy and the communication complexity remains the same. However, if the drop tolerance is too large, the solver is non-convergent, as described in Section 4.2.

In this section we consider an *adaptive* method that attempts to add entries back into the hierarchy as a deterioration in convergence is observed. This is detailed in Algorithm 4.4. The algorithm initializes a Sparse or Hybrid Galerkin hierarchy and proceeds by executing k iterations of a preconditioned Krylov method — e.g. PCG. If the convergence is below a tolerance, the coarse levels are traversed until a coarse grid operator is found on which entries were removed with a drop tolerance greater than 0.0. Entries are then added back to this coarse-grid operator, reducing the drop tolerance by a factor of 10. Any new drop tolerance below $\gamma_{\min} = 0.01$ is rounded down to 0.0. This continues until entries have been reintroduced into s coarse-grid operators. At this point, the Krylov method continues, using the most recent value for x unless the previous iterations diverged from the true solution.

This entire process is then repeated until convergence. The adaptive solve phase requires additional iterations over Galerkin AMG, as initial iterations of this method may not converge. However, the goal of this solver is to guarantee convergence similar to Galerkin AMG. Speed-up over Galerkin AMG is still dependent on choosing reasonable initial drop

tolerances.

If the Krylov method is not flexible, such as PCG or GMRES, then it must be restarted after the preconditioner has been edited. On the other hand, a flexible method, such as FGMRES, would not have to be restarted, but requires greater memory storage (and possibly also more computational work) than PCG. While the new algorithms are agnostic to the Krylov scheme, we use restarted PCG in order to directly compare schemes.

Example 4.1. *As an example, consider the case of a hierarchy with 6 levels using drop tolerances of* $[0, 0.01, 0.1, 1.0, 1.0, 1.0]$ *— i.e.,* $\hat{A}_1$ *retains all entries from* A_1*,* $\hat{A}_2$ *and* $\hat{A}_3$ *result from* `sparsify` *with* $\gamma = 0.01$ *and* $\gamma = 0.1$*, etc. Suppose that* `adaptive_solve` *with* $k = 3$ *and* $s = 2$ *results in 3 iterations of PCGand a large residual. The adaptive solve finds the first level containing a sparsified coarse grid matrix, namely* $\hat{A}_2$*. The drop tolerance on this level is changed from* 0.01 *to* 0.0*, and the original coarse matrix* A_2 *is sparsified with the new drop tolerance. Furthermore, since* $s = 2$ *the drop tolerance on level* 3 *is reduced from* 0.1 *to* 0.01*, and* A_3 *is also sparsified. PCG then restarts with the new hierarchy. If convergence continues to suffer after 3 iterations, the hierarchy is updated again, but since* $\hat{A}_2$ *has* $\gamma_2 = 0.0$*, entries are reintroduced into coarse matrices* $\hat{A}_3$ *and* $\hat{A}_4$ *instead.*

Using Algorithm 4.4, Figure 4.17 shows both the relative residual of the system after each iteration as well as the communication costs of PCG using three different AMG hierarchies: standard Galerkin, Sparse Galerkin with diagonal lumping and aggressive dropping, and Sparse Galerkin with diagonal lumping modified with adaptivity. For the adaptive case, we purposefully choose an overly aggressive initial drop tolerance so that entries can be added back multiple times and one coarse level at a time to show the effect on convergence and communication. Initially, when the drop tolerance is aggressive, the associated communication costs are low, but the resulting PCG iterations do not converge; this provides a baseline. As sparse entries are reintroduced into the hierarchy, convergence improves, while only slightly increasing the associated communication cost. When entries are reintroduced into the hierarchy, the preconditioner for PCG changes, and hence, the method must be restarted. After restarting the method, convergence improves.

4.7 CONCLUSION

We have introduced a lossless method to reduce the work required in parallel algebraic multigrid by removing weak or unimportant entries from coarse-grid operators after the multigrid hierarchy is formed. This alternative to the original method of non-Galerkin coarse grids is similarly capable of reducing the communication costs on coarse levels, yielding an

Algorithm 4.4: `adaptive_solve`

Input: A, b, x_0

$\hat{A}_1, \ldots, \hat{A}_L$	Sparse/Hybrid Galerkin coarse grid matrices
$A_1, \ldots, A_L$	original Galerkin coarse grid matrices
$P_0, \ldots, P_{L-1}$	
k	PCG iterations before convergence test
s	AMG levels per update
$\gamma_0, \ldots, \gamma_L$	sparsification drop tolerance used at each level
`tol`	convergence tolerance
`sparse_galerkin`	Sparse Galerkin method
`hybrid_galerkin`	Hybrid Galerkin method

Output: x

$x = x_0$
$r_0 = b - Ax_0$

while $\|r\|/\|r_0\| \leq$ `tol`
- $M =$ `preconditioner`$(\texttt{amg\_solve}, \hat{A}_1, \ldots, \hat{A}_L, P_0, \ldots, P_{L-1})$
- $x = \mathrm{PCG}(A, b, x, k, M)$ {Call k steps of preconditioned CG}
- $r = b - Ax$
- **if** $\frac{\|r\|}{\|r_0\|} \leq$ `tol`
  - `continue`
- **else**
  - **for** $\ell = 0, \ldots, L$ **do**
    - **if** $\gamma_\ell > 0$
      - $\ell_{\text{start}} \leftarrow \ell$ {Find finest level that uses dropping}
  - **for** $\ell = \ell_{start} \ldots \ell_{start} + s$ **do**
    - $\gamma_\ell = \begin{cases} \frac{\gamma_\ell}{10}, & \text{if } \frac{\gamma_\ell}{10} > \gamma_{\min} \\ 0, & \texttt{otherwise} \end{cases}$ {Determine new dropping parameter}
    - {$\gamma_{\min}$ is the $\min(\gamma_0 \ldots \gamma_L)$}
    - **if** `sparse_galerkin` {Re-add entries at the new dropping tolerance}
      - $\hat{A}_\ell =$`sparsify`$(A_\ell, A_{\ell-1}, P_{\ell-1}, S_{\ell-1}, \gamma_\ell)$
    - **else if** `hybrid_galerkin` {Re-add entries at the new dropping tolerance}
      - $\hat{A}_\ell =$`sparsify`$(A_\ell, \hat{A}_{\ell-1}, P_{\ell-1}, S_{\ell-1}, \gamma_\ell)$

overall reduction in solve times. Furthermore, this method retains the original Galerkin hierarchy, allowing many of the restrictions of non-Galerkin to be relaxed. As a result, removed entries are easily lumped directly to the diagonals, greatly reducing setup costs, while also reducing communication complexity during the solve phase. Furthermore, as entries are added to the diagonal, entries removed from the matrix are stored and adaptively

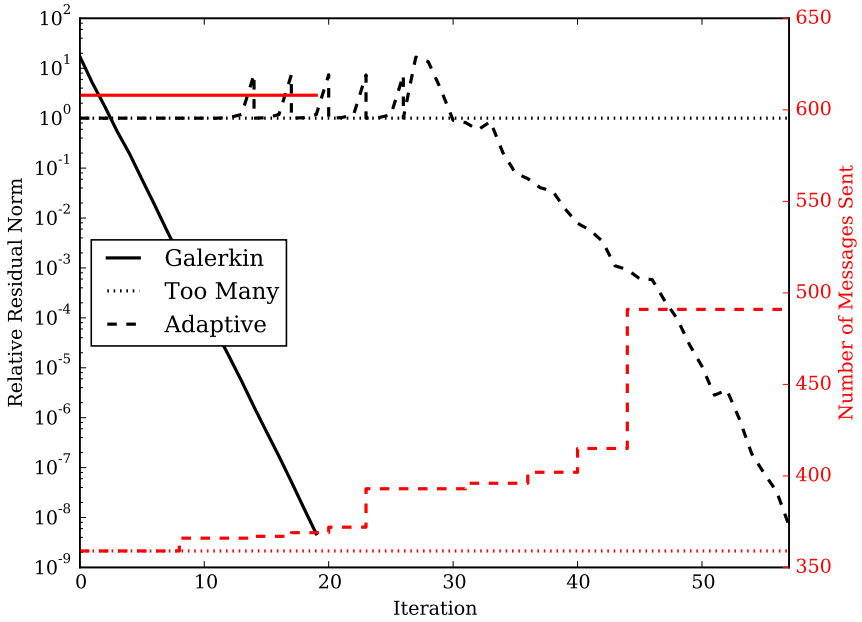


Figure 4.17: Relative residual (black) and number of MPI sends (red) per iteration when solving the **Laplace** problem with: (1) PCG using Galerkin AMG; (2) Hybrid Galerkin with aggressive dropping (labeled Too Many); (3) Hybrid Galerkin solved with Algorithm 4.4, using $k = 3$, $s = 1$, and $\gamma_0 \dots \gamma_L$ set to the same drop tolerances as the aggressive case.

reintroduced into the hierarchy if necessary for convergence. Hence, the trade-off between convergence and the communication costs is controlled at solve-time with little additional work.

CHAPTER 5: NODE-AWARE MESSAGE AGGLOMERATION

Portions of this chapter appear in the paper "Node Aware Sparse Matrix-Vector Multiplication", to appear in JPDC [60].

5.1 INTRODUCTION

Sparse matrix-vector multiplication (SpMV) is a widely used operation in many simulations and the is main kernel in iterative solvers. The focus of this section is on the parallel SpMV, namely

$$w \leftarrow A \cdot v \tag{5.1}$$

where A is a sparse $N \times N$ matrix and v is a dense N-dimensional vector. This operation, described in Section 2.1, lacks scalability due to large costs associated with communication.

Machine topology plays an important role in the cost of communication [61]. Multicore distributed systems present new challenges in communication as the bandwidth is limited while the number of cores participating in communication increases [62]. Injection limits and network contention are significant roadblocks in the SpMV operation [63], motivating the need for SpMV algorithms that take advantage of the machine topology. The focus of the approach developed in this section is to use the node-processor hierarchy to more efficiently map communication, leading to notable reductions in SpMV costs on modern HPC systems for a range of sparse matrix patterns. Throughout this section, the term *node aware* refers to knowledge of the mapping of processes to physical nodes, although other aspects of the topology — e.g. socket information — could be used in a similar fashion. The mapping of virtual ranks to physical processors can be easily determined on many super computers. The flag `MPICH_RANK_REORDER_METHOD` can be set to a predetermined ordering on Cray machines, while modern Blue Gene machines allow the user to specify the ordering among the coordinates A, B, C, D, E, and T through the variable `RUNJOB_MAPPING` or a runscript option of `--mapping`.

There are a number of existing approaches for reducing communication costs associated with sparse matrix-vector multiplication. Communication volume in particular is a limiting factor and the ordering and parallel partition of a matrix both influence the total volume of communicated data. In order to reduce this communication, graph partitioning techniques are used to identify more efficient layouts in the data [64, 65, 7, 8]. ParMETIS [66] and PT-Scotch [67], for example, provide parallel partitioning of matrices that often lead to improved system loads and more efficient sparse matrix operations. Communication volume

is accurately modeled through the use of a hypergraph [68]. As a result, hypergraph partitioning also leads to a reduction in parallel communication requirements, albeit at a larger one-time setup cost. Topology-aware task mapping is used to accurately map partitions to the allocated nodes of a supercomputer, reducing the overall cost associated with communication [69, 70, 71, 72, 73]. The approach introduced in this section complements these efforts by providing an additional level of optimization in handling communication.

Topology-aware methods and aggregation of data are commonly used to reduce communication costs, particularly in collective operations [74, 75, 76, 77]. Aggregation of data is used in point-to-point communication through Tram, a library for streamlining messages in which data is aggregated and communicated only through neighboring processors [78]. Furthermore, hybrid programming is often used on SMP nodes, with the combination of MPI and OpenMP yielding improvement over standard MPI approaches by reducing inter-node communication [79]. Furthermore, the task-based hybrid approach further reduces costs through an overlap of communication and local computation [80, 81, 82]. The method presented in this section aggregates messages at the node level and communicates all aggregated data at once, yielding little structural change from standard MPI communication while reducing overall cost.

The performance of matrix operations is also improved through the use of hybrid architectures and accelerators, such a graphics processing units (GPUs). The throughput of GPUs allows for improved performance when memory access patterns are optimized [83, 84, 85].

Many preconditioners for iterative methods, such as algebraic multigrid, are dominated in execution cost by SpMV operations and therefore lack scalability due to large communication costs. A variety of methods exist for altering the preconditioning algorithms to reduce the communication costs associated with each SpMV [86, 35, 87].

This section focuses on increasing the locality of communication during a SpMV to reduce the amount of communication injected into the network. Section 5.2 describes a reference algorithm for a parallel SpMV, which resembles the approach commonly used in practice. A performance model is also introduced in Section 5.3, which considers the cost of intra- and inter-node communication and the impact on performance. A new SpMV algorithm is presented in Section 5.4, which reduces the number and size of inter-node messages by increasing the significantly cheaper intra-node communication. The code and numerics are presented in Section 5.5 to verify the performance.

5.2 BACKGROUND

Modern supercomputers incorporate a large number of nodes through an interconnect to form a multi-dimensional grid or torus network. Standard compute nodes are comprised of one or more multicore processors that share a large memory bank. The algorithm developed in this section targets a general machine with this layout and the results are highlighted on Blue Waters, a Cray machine at the National Center for Supercomputing Applications. Blue Waters consists of 22 640 Cray XE nodes, each containing two AMD 6276 Interlagos processors for a total of 16 cores per node, and 4 228 Cray XK nodes consisting of a single AMD processor along with an NVIDIA Kepler GPU[1]. The nodes are connected through a three-dimensional torus Gemini interconnect, with each Gemini serving two nodes. The remainder of this section with focus on only the Cray XE nodes within Blue Waters.

Consider a system with n_p processes distributed across n_n nodes, resulting in `ppn` processes per node. Rank $r \in [0, n_p - 1]$ is described by the tuple (p, n) where $0 \leq p < \texttt{ppn}$ is the local process number of rank r on node n. Assuming SMP-style ordering, the first `ppn` ranks are mapped to the first node, the next `ppn` to the second node, and so on. Therefore, rank r is described by the tuple $\left(r \mod \texttt{ppn}, \lfloor \frac{r}{\texttt{ppn}} \rfloor\right)$. Thus, for the remainder of the section, the notation of rank r is interchangeable with (p, n).

Parallel matrices and vectors are distributed across all n_p ranks such that each process holds a portion of the linear system. Let $\mathcal{R}(r)$ be the rows of an $N \times N$ sparse linear system, $w \leftarrow A \cdot v$, stored on rank r. In the case of an even, contiguous partition where the k^{th} partition is placed on the k^{th} rank, $\mathcal{R}(r)$ is defined as

$$\mathcal{R}(r) = \left\{ \left\lfloor \frac{N}{n_p} \right\rfloor r, \ldots, \left\lfloor \frac{N}{n_p} \right\rfloor (r+1) - 1 \right\} \tag{5.2}$$

or equivalently as

$$\mathcal{R}((p,n)) = \left\{ \left\lfloor \frac{N}{n_p} \right\rfloor (p,n), \ldots, \left\lfloor \frac{N}{n_p} \right\rfloor ((p,n)+1) - 1 \right\}. \tag{5.3}$$

The rows of a matrix A are partitioned into on-process and off-process blocks, as described in Section 5.1. Accounting for parallel nodal awareness, the off-process block is further partitioned into on-node and off-node blocks, as described in Example 5.1.

Example 5.1. *Suppose the parallel system consists of six processes distributed across three nodes, as displayed in Figure 5.1. Let the linear system $w \leftarrow A \cdot v$ displayed in Figure 5.2 be partitioned across this processor layout with each process holding a single row of the matrix*

[1] `https://bluewaters.ncsa.illinois.edu/hardware-summary`

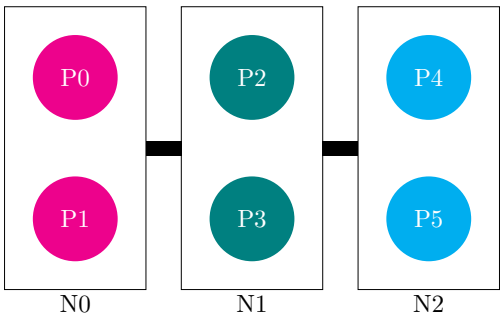


Figure 5.1: An example parallel system with six processes distributed across three nodes.

and associated row of the input vector. In this example, the diagonal entry falls into the

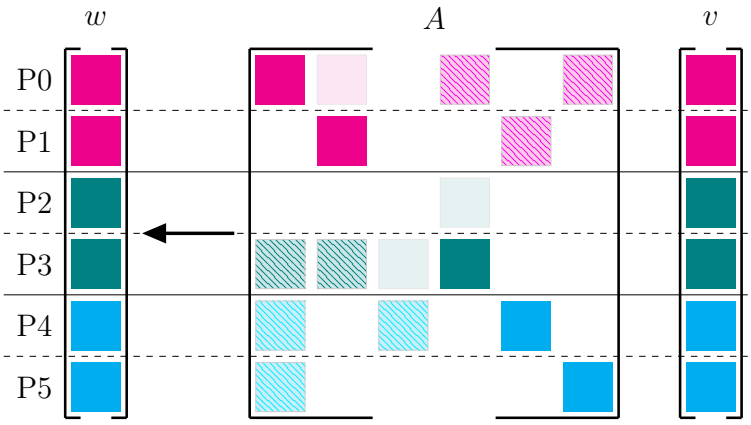


Figure 5.2: An example 6×6 sparse matrix for the parallel system in Figure 5.1. The solid shading denotes blocks that require only on-node communication, while the striped shading denotes blocks that require communication with distant nodes.

on-process block, as the corresponding vector value is stored locally. The off-process block, which requires communication, consists of all off-diagonal non-zeros as the associated vector values are stored on other processes.

For any process (p, n), the on-node columns of A correspond to vector values that are stored on some process (s, n), where $s \neq p$. Similarly, the off-node columns of A correspond to vector values stored on some process (q, m), where $m \neq n$. To make this clearer, we define the following

$$\texttt{on\_process}(A, (p, n)) = \{A_{ij} \neq 0 \,|\, i, j \in \mathcal{R}((p, n))\} \tag{5.4}$$

$$\texttt{off\_process}(A,(p,n)) = \{A_{ij} \neq 0 \,|\, i \in \mathcal{R}((p,n)), j \notin \mathcal{R}((p,n))\} \quad (5.5)$$

$$\texttt{on\_node}(A,(p,n)) = \{A_{ij} \neq 0 \,|\, \exists\, q \neq p \text{ with } i \in \mathcal{R}((p,n)),\ j \in \mathcal{R}((q,n))\} \quad (5.6)$$

and

$$\texttt{off\_node}(A,(p,n)) = \{A_{ij} \neq 0 \,|\, \exists\, q, m \neq n \text{ with } i \in \mathcal{R}((p,n)),\ j \in \mathcal{R}((q,m))\}\,. \quad (5.7)$$

5.2.1 Standard SpMV

For a sparse matrix-vector multiply, $w \leftarrow A \cdot v$, each process receives all values of v associated with the non-zero entries in the off-process block of A. For example, if rank r contains a non-zero entry of A, A_{ij}, at row i, column j, then rank s with row $j \in \mathcal{R}(s)$ sends the j^{th} vector value, v_j, to rank r. Typically, these communication requirements are determined as the sparse matrix is formed [88, 89, 90].

Example 5.2. *During a SpMV, data corresponding to off-process columns is gathered on process and communicated as a single message between processes, regardless of the node on which each process lies. Figure 5.3 displays the process of sending data from all processes on node n to a single process on node m. Multiple messages are communicated between the pair of nodes. Furthermore, Figure 5.4 exemplifies the process of communicating data from a single process on node n to all processes on node m. Not only does standard communication result in communication of multiple messages between the set of nodes, but duplicate data is also communicated.*

In the reference SpMV, for each rank r there is a list of processes to which data is sent, as well as the global vector indices to be sent to each. The function $\mathcal{P}(r)$ defines the list of processes to which a rank r sends. Specifically,

$$\mathcal{P}(r) = \{t \,|\, A_{ij} \neq 0 \text{ with } i \in \mathcal{R}(t),\ j \in \mathcal{R}(r),\ r \neq t\} \quad (5.8)$$

For each t in $\mathcal{P}(r)$, define the function $\mathcal{D}(r,t)$ to return the global vector indices that process

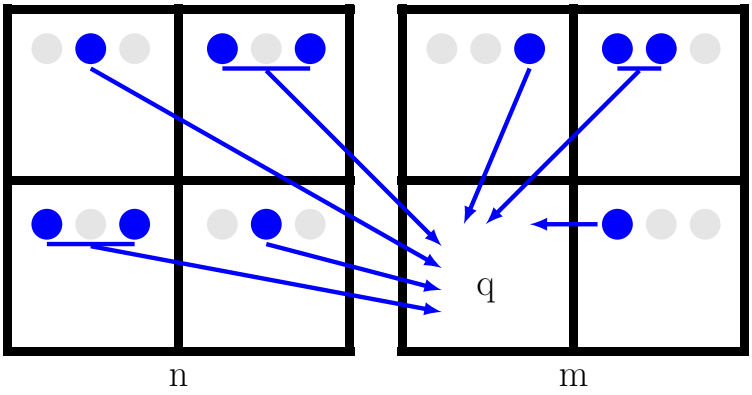


Figure 5.3: Standard communication of data from processes across node n and node m to a process q on node m results in multiple messages between sets of nodes.

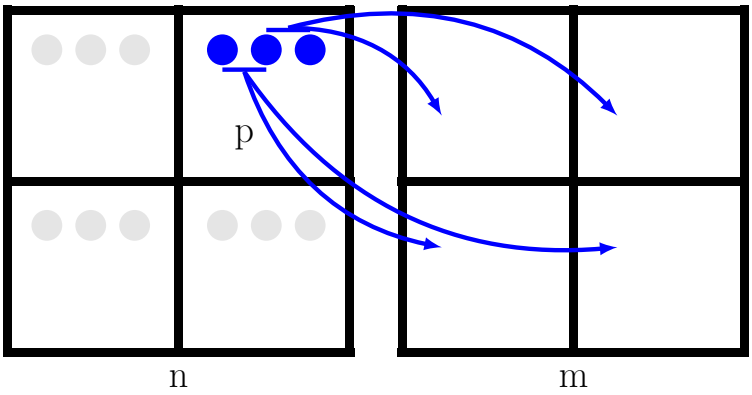


Figure 5.4: Standard communication from a process p on node n to all processes on node m results in multiple messages and duplication data communicated between the pair of nodes.

r sends to process t. This function is defined as follows.

$$\mathcal{D}(r,t) = \{i \,|\, A_{ij} \neq\ 0 \text{ with } i \in \mathcal{R}(t),\ j \in \mathcal{R}(r),\ r \neq t\} \tag{5.9}$$

Consider a standard SpMV for the linear system described in Example 5.1. Table 5.1 lists the processes to which each rank must send, while Table 5.2 displays the indices that each rank r sends to any rank t.

	r					
	0	1	2	3	4	5
$\mathcal{P}(r)$	$\{3,4,5\}$	$\{0,3\}$	$\{3,4\}$	$\{0,2\}$	$\{1\}$	$\{0\}$

Table 5.1: Communication pattern for rank r in Example 5.1, containing the values for $\mathcal{P}(r)$.

.

		r					
		0	1	2	3	4	5
t	0	{}	{1}	{}	{3}	{}	{5}
	1	{}	{}	{}	{}	{4}	{}
	2	{}	{}	{}	{3}	{}	{}
	3	{0}	{1}	{2}	{}	{}	{}
	4	{0}	{}	{2}	{}	{}	{}
	5	{0}	{}	{}	{}	{}	{}

Table 5.2: Each column r lists the indices of values sent to each process t in $\mathcal{P}(r)$, namely $\mathcal{D}(r,s)$.

.

With these definitions, the *standard* or reference SpMV is described in Algorithm 5.1. It is important to note that the parallel communication in Algorithm 5.1 is executed independent of any locality in the problem. That is, messages sent to another process may be both on-node or off-node depending on the process, however this is not considered in the algorithm.

5.3 COMMUNICATION MODELS

The performance of Algorithm 5.1 is sub-optimal since it does not take advantage of node locality in the communication. To see this, a communication performance model is developed in this section. One approach is that of the *max-rate model* [62], which describes the communication time as

$$T = \alpha + \frac{\texttt{ppn} \cdot s}{\min(R_{\text{N}}, \texttt{ppn}R_{\max}}, \tag{5.10}$$

Algorithm 5.1: `standard_spmv`

Input: r
$A|_{\mathcal{R}(r)}$
$v|_{\mathcal{R}(r)}$

Output: $w|_{\mathcal{R}(r)}$

$A_{\text{on\_process}} = \texttt{on\_process}(A|_{\mathcal{R}(r)})$
$A_{\text{off\_process}} = \texttt{off\_process}(A|_{\mathcal{R}(r)})$
for $t \in \mathcal{P}(r)$ **do**
  for $i \in \mathcal{D}(r,t)$ **do**
    $b_{\text{send}} \leftarrow v|_{\mathcal{R}(r)_i}$
  `MPI_Isend`$(b_{\text{send}}, \ldots, t, \ldots)$
$b_{\text{recv}} \leftarrow \emptyset$
for t s.t. $r \in \mathcal{P}(t)$ **do**
  `MPI_Irecv`$(b_{\text{recv}}, \ldots, t, \ldots)$
`local_spmv`$(A_{\text{on\_process}}, v|_{\mathcal{R}(r)})$
`MPI_Waitall`
`local_spmv`$(A_{\text{off\_process}}, b_{\text{recv}})$

where α is the *latency* or start-up cost of a message, which may include preparing a message for transport or determining the network route; s is the number of bytes to be communicated; `ppn` is again the number of communicating processes per node; $R_{\max}$ is the achievable message rate of each process or *bandwidth*; and R_{N} is the peak rate of the network interface controller (NIC). In the simplest case of $\texttt{ppn} = 1$, the familiar *postal model* suffices:

$$T = \alpha + \frac{s}{R_{\max}}. \tag{5.11}$$

MPI contains multiple message passing protocols, including short, eager, and rendezvous. Each message consists of an envelope, including information about the message such as message size and source information, as well as message data. Short messages contain very little data which is sent as part of the envelope. Eager and rendezvous messages, however, send the envelope followed by packets of data. Eager messages are sent under the assumption that the receiving process has buffer space available to store data that is communicated. Therefore, a message is sent without checking buffer space at the receiving process, limiting the associated latency. However, if a message is sufficiently large, rendezvous protocol must be used. This protocol requires the sending process to inform the receiving rank of the message so that buffer space is allocated. The message is sent only once the sending process is informed that this space is available. Therefore, there is a larger overhead with sending a

	α	$B_{\max}$	B_{N}
Short	$4.0 \cdot 10^{-6}$	$-1.8 \cdot 10^{7}$	∞
Eager	$1.1 \cdot 10^{-5}$	$6.2 \cdot 10^{7}$	∞
Rend	$2.0 \cdot 10^{-5}$	$6.1 \cdot 10^{8}$	$5.5 \cdot 10^{9}$

Table 5.3: Measurements for α, $R_{\min}$, and R_{N} for Blue Waters.

	α_ℓ	$B_{\max_\ell}$
Short	$1.3 \cdot 10^{-6}$	$4.2 \cdot 10^{8}$
Eager	$1.6 \cdot 10^{-6}$	$7.4 \cdot 10^{8}$
Rend	$4.2 \cdot 10^{-6}$	$3.1 \cdot 10^{9}$

Table 5.4: Measurements for intra-node variables, α_ℓ and $R_{\max_\ell}$.

message using rendezvous protocol. Table 5.3 displays the measurements for α, $R_{\max}$, and R_{N} for Blue Waters, as determined for the *max-rate* model.

The *max-rate* model can be improved by distinguishing between intra- and inter-node communication. If the sending and receiving processes lie on the same physical node, data is not injected into the network, yielding low start-up and byte transport costs. As intra-node messages are not injected into the network, communication local to a node can be modeled as

$$T_\ell = \alpha_\ell + \frac{s_\ell}{R_{\max_\ell}}, \tag{5.12}$$

where α_ℓ is the start-up cost for intra-node messages; s_ℓ is the number of bytes to be transported; and $R_{\max_\ell}$ is the achievable intra-node message rate.

Nodecomm[2], a topology-aware communication program, measures the time required to communicate on various levels of the parallel system, such as between two nodes of varying distances and between processes local to a node. Communication tests between processes local to one node were used to calculate the intra-node model parameters, as displayed in Table 5.4.

Furthermore, Figure 2.3 shows the time required to send a single message of varying sizes. The thin lines display Nodecomm measurements for time required to send a single message, as either inter- or intra-node communication. Furthermore, the thick lines represent the time required to send a message of each size, according to the *max-rate* model in (5.10) and intra-node model in (5.12). This figure displays a significant difference between the costs of intra- and inter-node communication.

[2]See `https://bitbucket.org/william_gropp/baseenv`

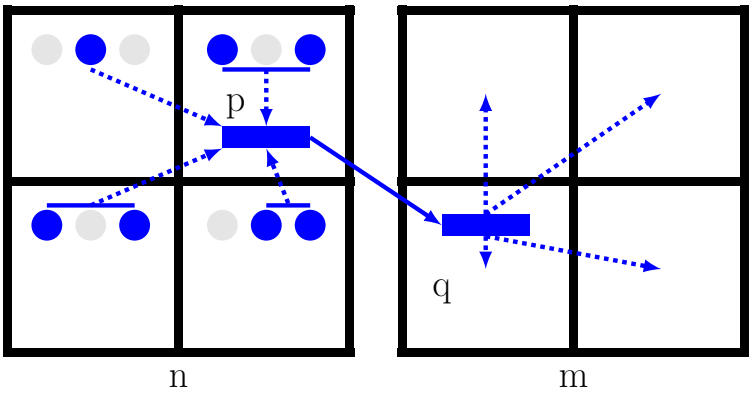


Figure 5.5: The various arrows exemplify the process of communicating data from each process on node n to processes on node m through a three-step algorithm. The bold circles on node n represent vector values communicated to node m.

5.4 NODE AWARE PARALLEL SPMV

To reduce communication costs, the algorithm proposed in this section decreases the number and size of messages being injected into the network by increasing the amount of intra-node communication, which is less-costly than inter-node communication. This trade-off is accomplished through a so-called *node aware parallel* SpMV (NAPSpMV), where values are gathered in processes local to each node before being sent across the network, followed by a distribution of processes on the receiving node. As a result, as the matrix is formed each process (p, n) determines the communicating processes during the various steps of a NAPSpMV, as well as the accompanying data. A high level overview of the process is described in Example 5.3. It is important to note that the communication for each NAPSpMV is load-balanced such that all processes local to node n send and receive both a similar number and size of messages through inter-node communication. Therefore, it is assumed that the nodes n and m in Example 5.3 are only a portion of the parallel system, and n is communicating with other nodes in a similar fashion. If the parallel system consists only of nodes n and m, each process on node n would send a portion of the data to node m.

Example 5.3. *During each NAPSpMV, off-node data is communicated through a three-step process, as displayed in Figure 5.5. This figure displays a portion of a parallel system consisting of 8 processes partitioned across two nodes, labeled n and m. The solid circles on node n represent vector values sent to node m. Therefore, each process on node n must send values to processes on node m. Instead of sending directly to destination processes, each process (s, n) sends to the process labeled (p, n), displayed by the dashed arrows on node n. Process (p, n) then sends all collected values through the network to process (q, m). Finally, process (q, m) distributes received values among the processes local to node m, displayed by*

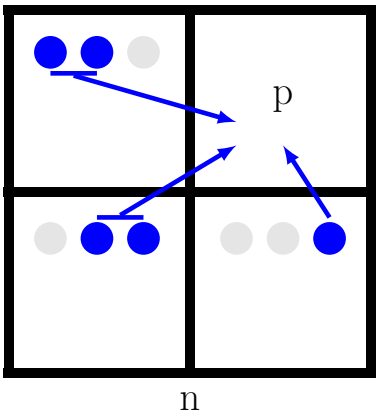


Figure 5.6: An example of how vector values corresponding with matrix entries in the on-node block are communicated. All values (p, n) must receive from other processes (q, n) are communicated directly as nothing is injected into the network.

the dashed arrows on node m.

On-node data is communicated directly between the process on which the vector values are stored and that which requires the data, as displayed in Figure 5.6. In this example, the solid circles represent vector values that are stored on each process (s, n) and needed by (p, n). This data is sent directly between the processes in a single step.

5.4.1 Inter-node communication setup

To eliminate the communication of duplicated messages, a list of communicating nodes is formed for each node n along with the accompanying data values. These lists are then distributed across all processes local to n by balancing the number of nodes and volume of data for communication. To facilitate this, the function $\mathcal{N}(n)$ defines the set of nodes to which the processes on node n must send,

$$\begin{aligned}\mathcal{N}(n) = \{m \,|\, &\exists\ p,\ q \text{ s.t. } A_{ij} \neq 0 \\ &\text{with } i \in \mathcal{R}((q,m)),\ j \in \mathcal{R}((p,n)),\ n \neq m\}\,.\end{aligned} \tag{5.13}$$

Table 5.5 contains $\mathcal{N}(n)$, the list of nodes to which each node n sends. The associated

	n		
	0	1	2
$\mathcal{N}(n)$	$\{1,2\}$	$\{0,2\}$	$\{0\}$

Table 5.5: Communication requirements for each node n in Example 5.1.

data values are defined for each node $m \in \mathcal{N}(n)$ with $\mathcal{E}(n, m)$, which returns the data indices

to be sent from node n to node m. That is,

$$\mathcal{E}(n,m) = \{i \,|\, \exists\ p,\ q \text{ s.t. } A_{ij} \neq\ 0 \text{ with } i \in \mathcal{R}((q,m))\ , \\ j \in \mathcal{R}((p,n)),\ n \neq\ m\}\,. \quad (5.14)$$

Extending Example 5.1, Table 5.6 displays the global vector indices, $\mathcal{E}(n,m)$, for each set of nodes n and m.

		n = 0	1	2
	0	{}	{3}	{4, 5}
m	1	{0, 1}	{}	{}
	2	{0}	{2}	{}

Table 5.6: In Example 5.1, each column n contains the values sent from n to m, as in $\mathcal{E}(n,m)$.

$\mathcal{T}((p,n))$ defines the nodes to which (p,n) must send, that is the nodes in $\mathcal{N}(n)$ that are distributed to process (p,n). Similarly, $\mathcal{U}((p,n))$ contains the nodes that send to (p,n). Specifically,

$$\mathcal{T}((p,n)) = \{m \in \mathcal{N}(n) \,|\, m \text{ maps to } (p,n)\}\,, \quad (5.15)$$

$$\mathcal{U}((p,n)) = \{m \,|\, n \in \mathcal{N}(m),\ n \text{ maps to } (p,n)\}\,. \quad (5.16)$$

This section considers a simple distribution in which the node $m \in \mathcal{N}(n)$ to which the most data $|\mathcal{D}(n,m)|$ is sent is mapped to process $(0,n)$, the node with the second most data is mapped to process $(1,n)$, and so on. The opposite ordering is used for $\mathcal{U}((p,n))$, mapping the node $n \in \mathcal{N}(m)$ with largest $|\mathcal{D}(m,n)|$ to process $(\texttt{ppn}-1,n)$, the second largest to process $(\texttt{ppn}-2,n)$, etc. If there are fewer nodes in $\mathcal{N}(n)$ than there are processes per node, a single node is mapped to multiple local processes so that all processes communicate. There are various other possible mapping strategies, such as mapping a node m to the process (p,n) storing the majority of the data in $\mathcal{D}(n,m)$. However, as this would only affect intra-node communication requirements, these mappings are not explored in this section.

The processor layout in Example 5.1 is displayed in Table 5.7, where the columns contain the send and receive nodes that are mapped to each process.

Finally, $\mathcal{G}((p,n))$ defines the set of all off-node processes to which process (p,n) sends data

	(p,n)					
	(0, 0)	(1, 0)	(0, 1)	(1, 1)	(0, 2)	(1, 2)
$\mathcal{T}((p,n))$	{1}	{2}	{0}	{2}	{0}	{}
$\mathcal{U}((p,n))$	{2}	{1}	{}	{0}	{1}	{0}

Table 5.7: Processor mappings for $\mathcal{N}(n)$, namely $\mathcal{T}((p,n))$ and $\mathcal{U}((p,n))$ for Example 5.1.

during the inter-node communication step of the NAPSpMV. Specifically,

$$\mathcal{G}((p,n)) = \{(q,m) \,|\, m \in \mathcal{T}((p,n)),\, n \in \mathcal{U}((q,m))\}\,. \tag{5.17}$$

Following Example 5.1, the columns of Table 5.8 list the indices of the values that each (p,n) sends, $\mathcal{G}((p,n))$. Finally, let $\mathcal{I}((p,n),(q,m))$ define the global data indices corresponding to

	(p,n)					
	(0, 0)	(1, 0)	(0, 1)	(1, 1)	(0, 2)	(1, 2)
$\mathcal{G}((p,n))$	$\{(1,1)\}$	$\{(1,2)\}$	$\{(1,0)\}$	$\{(0,2)\}$	$\{(0,0)\}$	{}

Table 5.8: Inter-node communication requirements of each process (p,n) for Example 5.1
.

the values sent from process (p,n) to (q,m):

$$\mathcal{I}((p,n),(q,m)) = \{\mathcal{E}(n,m) \,|\, m \in \mathcal{T}((p,n)),\, n \in \mathcal{U}((q,m)\} \tag{5.18}$$

The global vector indices to which each process (p,n) sends and receives for Example 5.1 are displayed in Table 5.9.

		(p,n)					
		(0, 0)	(1, 0)	(0, 1)	(1, 1)	(0, 2)	(1, 2)
(q,m)	(0, 0)	{}	{}	{}	{}	$\{4,5\}$	{}
	(1, 0)	{}	{}	{3}	{}	{}	{}
	(0, 1)	{}	{}	{}	{}	{}	{}
	(1, 1)	{0}	{}	{}	{}	{}	{}
	(0, 2)	{}	{}	{}	{2}	{}	{}
	(1, 2)	{}	$\{0,1\}$	{}	{}	{}	{}

Table 5.9: Inter-node communication requirements for each set of processes (p,n) and (q,m). Each column (p,n) contains the indices of values sent from (p,n) to (q,m).

5.4.2 Local Communication

The function $\mathcal{G}_{\text{send}}((p,n))$ for $p = 0, \ldots, \texttt{ppn} - 1$, describes evenly distributed inter-node communication requirements for all processes local to node n. However, many of the vector indices to be sent to off-node process $(q,m) \in \mathcal{D}((p,n),(q,m))$, are not stored on process (p,n). For instance, in Table 5.9, process $(0,1)$ sends global vector indices 0 and 1. However, only row 1 is stored on process $(0,1)$, requiring vector component 0 to be communicated before inter-node messages are sent.

Similarly, many of the indices that a process (q,m) receives from (p,n) are redistributed to various processes on node n. Table 5.9 requires process $(1,2)$ to receive vector data according to indices 0 and 1. Process $(0,2)$ uses both of these vector values, yielding a requirement for redistribution of data received from inter-node communication. Therefore, local communication requirements must be defined.

Each NAPSpMV consists of multiple steps of intra-node communication. Let a function $\mathcal{L}((p,n),\texttt{locality})$ define all processes, local to node n, to which process (p,n) sends messages, where `locality` is a tuple describing the locality of both the original location of the data as well as its final destination. The locality of each position is described as either `on_node`, meaning a process local to node n, or `off_node`, meaning a process local to node $m \neq n$.

There are three possible combinations for `locality`:

- the data is initialized `on_node` with a final destination `off_node`;
- the original data is `off_node` while the final destination is `on_node`; or
- both the original data and the final location are `on_node`.

These three types of intra-node communication are described in more detail in the remainder of Section 5.4.2.

For each process $(s,n) \in \mathcal{L}((p,n),\texttt{locality})$, $\mathcal{J}((p,n),(s,n),\texttt{locality})$ defines the global vector indices to be sent from process (p,n) to (s,n) through intra-node communication. This notation is used in following sections.

Local redistribution of initial data

During inter-node communication, process (p,n) sends all vector values corresponding to the global indices in $\mathcal{I}((p,n),(q,m))$ to each process $(q,m) \in \mathcal{G}((p,n))$. The indices in

$\mathcal{I}((p,n),(q,m))$ originate on node n, but not necessarily process (p,n). Therefore, the initial vector values must be redistributed among all processes local to node n.

Let $\mathcal{L}((p,n),(\texttt{on\_node},\texttt{off\_node}))$ represent all processes, local to node n, to which (p,n) sends initial vector values. This function is defined as

$$\mathcal{L}((p,n),(\texttt{on\_node},\texttt{off\_node})) = \{(s,n)\,|\,\exists\, j \in \mathcal{R}((p,n)),\ j \in \mathcal{I}((s,n),(q,m))\}\,. \quad (5.19)$$

The local processes to which each (p,n) sends initial data in Example 5.1 are displayed in Table 5.10.

	(p,n)					
	(0, 0)	(1, 0)	(0, 1)	(1, 1)	(0, 2)	(1, 2)
$\mathcal{L}$	$\{(1,0)\}$	$\{\}$	$\{(1,1)\}$	$\{(0,1)\}$	$\{\}$	$\{(0,2)\}$

Table 5.10: Initial intra-node communication requirements for each process (p,n) in Example 5.1. The row of the table describes $\mathcal{L}((p,n),(\texttt{on\_node},\texttt{off\_node}))$.

Furthermore, the data global vector indices sent from process (p,n) to each $(s,n) \in \mathcal{L}((p,n),(\texttt{on\_node},\texttt{off\_node}))$ are defined as

$$\mathcal{J}((p,n),(s,n),(\texttt{on\_node},\texttt{off\_node})) = \{i\,|\,i \in \mathcal{R}((p,n)), \forall\, i \in \mathcal{G}((s,n))\}\,. \quad (5.20)$$

The global vector indices that each (p,n) must send to other processes on node n in Example 5.1 are displayed in Figure 5.11.

		(p,n)					
		(0, 0)	(1, 0)	(0, 1)	(1, 1)	(0, 2)	(1, 2)
(q,n)	(0, 0)	{}	{}	—	—	—	—
	(1, 0)	{0}	{}	—	—	—	—
	(0, 1)	—	—	{}	{3}	—	—
	(1, 1)	—	—	{2}	{}	—	—
	(0, 2)	—	—	—	—	{}	{5}
	(1, 2)	—	—	—	—	{}	{}

Table 5.11: Global vector indices of initial data that is communicated between processes local to each node n in Example 5.1. Each column contains the indices of values sent from (p,n) to (q,n). Note: dashes (—) throughout the table represent processes on separate nodes, which do not communicate during intra-node communication.

Local redistribution of received off-node data

During inter-node communication, process (p, n) sends all data with final destination on node m to process $(q, m) \in \mathcal{G}((p, n))$. Process (q, m) then distributes these values across the processes local to node m. Let $\mathcal{L}((q, m), (\texttt{off\_node}, \texttt{on\_node}))$ define all processes local to node m to which process (q, m) sends vector values that have been received through inter-node communication. This function is defined as

$$\mathcal{L}((q, m), (\texttt{off\_node}, \texttt{on\_node})) = \{(s, m) \,|\, \exists\, A_{ij} \neq 0 \text{ with } i \in \mathcal{R}((s, m)), \; j \in \mathcal{I}((p, n), (q, m))\}\,. \quad (5.21)$$

This is highlighted, for Example 5.1, in Table 5.12. Furthermore, the data global vector

	(p,n)					
	(0, 0)	(1, 0)	(0, 1)	(1, 1)	(0, 2)	(1, 2)
$\mathcal{L}$	{}	$\{(0,0)\}$	{}	{}	{}	$\{(0,2)\}$

Table 5.12: Intra-node communication requirements containing processes to which each (p, n) sends received inter-node data, according to Example 5.1. The row of the table describes $\mathcal{L}((p, n), (\texttt{off\_node}, \texttt{on\_node}))$.

indices sent from process (q, m) to each $(s, m) \in \mathcal{L}((q, m), (\texttt{off\_node}, \texttt{on\_node}))$ are defined as

$$\mathcal{J}((q, m), (s, m), (\texttt{off\_node}, \texttt{on\_node})) = \{j \in \mathcal{I}((p, n), (q, m)) \,|\, A_{ij} \neq 0 \text{ with } i \in \mathcal{R}((s, m))\}\,. \quad (5.22)$$

The global vector indices that (p, n) sends to local process (q, n), received through inter-node communication in Example 5.1, are displayed in Table 5.13.

Fully Local Communication

A subset of the values needed by a process (p, n) are stored on local process (s, n). One advantage is that these values bypass the three-step communication, and are communicated

(q,n) \ (p,n)	(0, 0)	(1, 0)	(0, 1)	(1, 1)	(0, 2)	(1, 2)
(0, 0)	{}	{3}	—	—	—	—
(1, 0)	{}	{}	—	—	—	—
(0, 1)	—	—	{}	{}	—	—
(1, 1)	—	—	{}	{}	—	—
(0, 2)	—	—	—	—	{}	{1}
(1, 2)	—	—	—	—	{}	{}

Table 5.13: Global vector indices of received inter-node data that must be communicated between processes local to each node n in Example 5.1. Each column contains the indices of values sent from (p,n) to (q,n). Note: dashes (—) throughout the table represent processes on separate nodes, which cannot communicate during intra-node communication.

directly. Let $\mathcal{L}((p,n),(\texttt{on\_node},\texttt{on\_node}))$ define all processes local to node n to which (p,n) sends vector data. This function is defined as

$$\mathcal{L}((p,n),(\texttt{on\_node},\texttt{on\_node})) = \{(s,n)\,|\,\exists\, A_{ij} \neq 0 \text{ with } i \in \mathcal{R}((s,n)), j \in \mathcal{R}((p,n))\}\,. \quad (5.23)$$

The processes local to node n, to which (p,n) must send initial vector data in Example 5.1 are displayed in Table 5.14. Furthermore, the global vector indices sent from process (p,n)

$\mathcal{L}$ \ (p,n)	(0, 0)	(1, 0)	(0, 1)	(1, 1)	(0, 2)	(1, 2)
	{}	{(0, 0)}	{}	{(0, 1)}	{}	{}

Table 5.14: Intra-node communication requirements containing processes to which each process (p,n) must send vector data, according to Example 5.1. The row of the table describes $\mathcal{L}((p,n),(\texttt{on\_node},\texttt{on\_node}))$.

to each $(s,n) \in \mathcal{L}((p,n),(\texttt{on\_node},\texttt{on\_node}))$ is defined as follows.

$$\mathcal{J}((p,n),(s,n),(\texttt{on\_node},\texttt{on\_node})) = \{j\,|\,\exists\, A_{ij} \neq 0 \text{ with } i \in \mathcal{R}((s,n)), j \in \mathcal{R}((p,n))\}\,. \quad (5.24)$$

The global vector indices which (p,n) must send to each local process (s,n) in Example 5.1 are displayed in Table 5.15.

(s,n) \ (p,n)	(0, 0)	(1, 0)	(0, 1)	(1, 1)	(0, 2)	(1, 2)
(0, 0)	{}	{1}	—	—	—	—
(1, 0)	{}	{}	—	—	—	—
(0, 1)	—	—	{}	{3}	—	—
(1, 1)	—	—	{}	{}	—	—
(0, 2)	—	—	—	—	{}	{}
(1, 2)	—	—	—	—	{}	{}

Table 5.15: Global vector indices communicated between processes local to each node n in Example 5.1. Each column contains the indices of values sent from (p, n) to (q, n). Note: dashes (—) throughout the table represent processes on separate nodes, which cannot communicate during intra-node communication.

Algorithm 5.2: `local_comm`

Input: (p, n) : tuple describing local rank and node of process
$v|_{\mathcal{R}((p,n))}$: rows of input vector v local to process (p, n)
`locality`: locality of input and output data

Output: ℓ_{recv}: values that rank (p, n) receives from other processes

{Initialize sends}

for $(s, n) \in \mathcal{L}((p, n), \texttt{locality})$ **do**
  for $i \in \mathcal{J}((p, n), (s, n), \texttt{locality})$ **do**
    $\ell_{\text{send}} \leftarrow v|_{\mathcal{R}((p,n))_i}$
  `MPI_Isend`$(\ell_{\text{send}}, \ldots, (s, n), \ldots)$

{Initialize receives}

$\ell_{\text{recv}} \leftarrow \emptyset$
for (s, n) *s.t.* $(p, n) \in \mathcal{L}((s, n), \texttt{locality})$ **do**
  `MPI_Irecv`$(\ell_{\text{recv}}, \ldots, (s, n), \ldots)$

{Complete sends and receives}

`MPI_Waitall`

5.4.3 Alternative SpMV Algorithm

The method of communicating vector values to on-node processes is described in Algorithm 5.2. Using the definitions for the various steps of intra- and inter-node communication, the NAPSpMV is described in Algorithm 5.3, where `local_spmv`() refers to a row-wise,

non-distributed SpMV — e.g. with Intel's MKL library or with the Eigen Library. It is important to note that many slight variations to the algorithm are possible. The fully local communication has no dependencies, and may be performed anytime before calling `local_spmv`$(A_{\text{on\_node}}, b_{\ell\to\ell})$. Furthermore, the function `local_spmv`$(A_{\text{on\_process}}, v|\mathcal{R})$ has no communication requirements and, hence, can be performed at any point in the algorithm.

Algorithm 5.3: `NAPSpMV`

Input: (p, n): tuple describing local rank and node of process
$A|R$: rows of matrix A local to process (p, n)
$v|R$: rows of input vector v local to process (p, n)

Output: $w|\mathcal{R}$: rows of output vector $w \leftarrow Av$, local to process (p, n)

$A_{\text{on\_process}} = \texttt{on\_process}(A|\mathcal{R})$
$A_{\text{on\_node}} = \texttt{on\_node}(A|\mathcal{R})$
$A_{\text{off\_node}} = \texttt{off\_node}(A|\mathcal{R})$
$b_{\ell\to\ell} \leftarrow \texttt{local\_comm}((p, n), v|\mathcal{R}, (\texttt{on\_node} \to \texttt{on\_node}))$
$b_{\ell\to n\ell} \leftarrow \texttt{local\_comm}((p, n), v|\mathcal{R}, (\texttt{on\_node} \to \texttt{off\_node}))$

{Initialize sends}

for $(q, m) \in \mathcal{G}((p, n))$ **do**
  for $i \in \mathcal{I}((p, n), (q, m))$ **do**
    $g_{\text{send}} \leftarrow b^i_{\ell\to n\ell}$
  `MPI_Isend`$(g_{\text{send}}, \ldots, (q, m), \ldots)$

{Initialize receives}

$g_{\text{recv}} \leftarrow \emptyset$
for (q, m) *s.t.* $(p, n) \in \mathcal{G}((q, m))$ **do**
  `MPI_Irecv`$(g_{\text{recv}}, \ldots, (q, m), \ldots)$

{Serial SpMV for local values}

`local_spmv`$(A_{\text{on\_process}}, v|\mathcal{R})$

{Serial SpMv for on-node values}

`local_spmv`$(A_{\text{on\_node}}, b_{\ell\to\ell})$

{Complete sends and receives}

`MPI_Waitall`
$b_{n\ell\to\ell} \leftarrow \texttt{local\_comm}((p, n), v|\mathcal{R}, (\texttt{off\_node} \to \texttt{on\_node}))$

{Serial SpMV for off-node values}

`local_spmv`$(A_{\text{off\_node}}, b_{n\ell\to\ell})$

5.5 RESULTS

In this section, the parallel performance and scalability of the NAPSpMV in comparison to the standard SpMVis presented. The matrix-vector multiplication in an algebraic multigrid (AMG) hierarchy is tested for both a structured 2D rotated anisotropic and for unstructured linear elasticity on 32 768 processes in order to expose a variety of communication patterns. In addition, scaling tests are considered for random matrices with a constant number of non-zeros per row to investigate problems with no structure. Lastly, scaling tests on the largest 15 matrices from the SuiteSparse matrix collection are presented. All tests are performed on the Blue Waters parallel computer at University of Illinois at Urbana-Champaign.

AMG hierarchies consist of successively coarser, but denser levels. Therefore, while a standard SpMV performed on the original matrix often requires communication of a small number of large messages, coarse levels require a large number of small messages to be injected into the network. Figure 5.7 shows that both the number and size of inter-node messages required on each level of the linear elasticity hierarchy are reduced through use of the NAPSpMV. There is a large reduction in communication requirements for coarse levels of the hierarchy, which includes a high number of small messages. However, as the NAPSpMV requires redistribution of data among processes local to each node, the intra-node communication requirements increase greatly for the NAPSpMV, as shown in Figure 5.8.

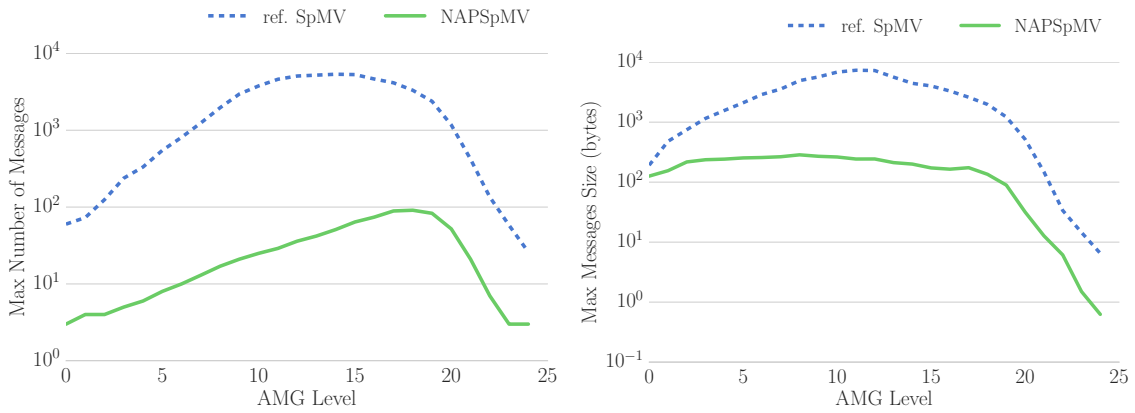


Figure 5.7: The maximum number (top) and size (bottom) of **inter-node** messages communicated by a single process during a standard SpMV and NAPSpMV on each level of the **linear elasticity AMG hierarchy**.

While there is an increase in intra-node communication requirements, the reduction in more expensive inter-node messages results in a significant reduction in total time for the NAPSpMV algorithm, particularly on coarser levels near the middle of each AMG hierarchy, as shown in Figure 5.9.

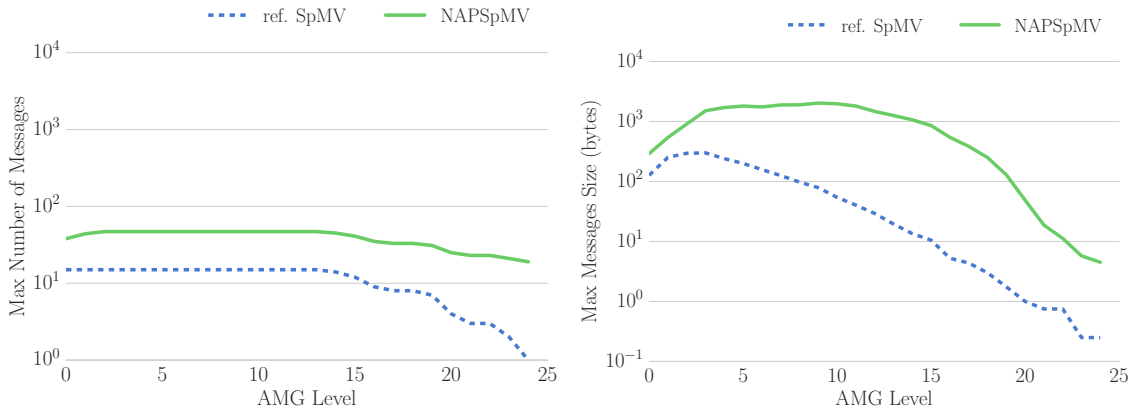


Figure 5.8: The maximum number (top) and size (bottom) of **intra-node** messages communicated by a single process during a standard SpMV and NAPSpMV on each level of the **linear elasticity AMG hierarchy**.

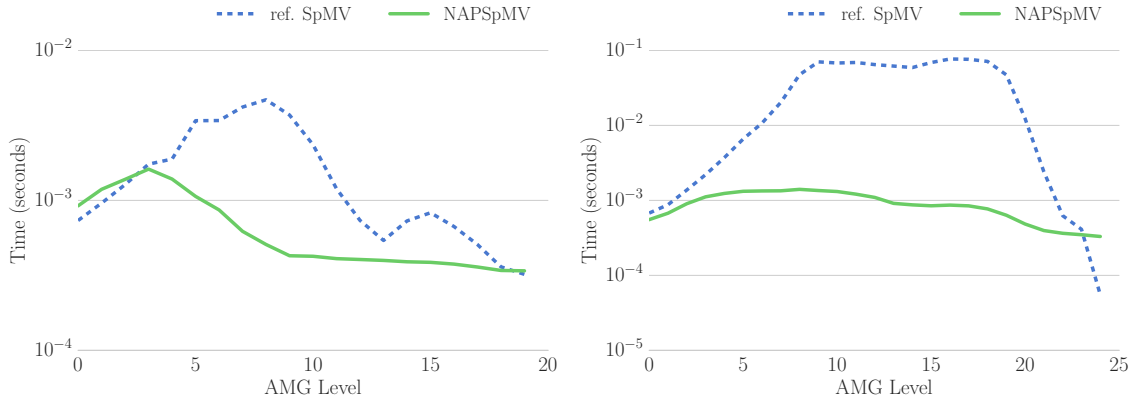


Figure 5.9: The time required to perform the various SpMVs on each level of the rotated anisotropic (left) and linear elasticity (right) AMG hierarchies.

Random matrices, formed with a constant number of non-zeros per row, lack structure that is found in many finite element discretizations. As these matrices are distributed across an increasingly large number of processes, non-zeros are more likely to be located in off-process blocks of the matrix. Therefore, both weak and strong scaling studies of random matrices yield increases in communication requirements with scale. The sparsity pattern of random matrices varies with random number generator seeds and are dependent on the number of non-zeros per row. Therefore, the standard SpMV and NAPSpMV were performed on five different random matrices for each tested density of 25, 50, and 100 non-zeros per row, as shown in Figure 5.10. The standard and NAPSpMV costs for all random matrices of equivalent density are comparable. Furthermore, there is little difference in costs between each density. Therefore, extended tests are performed on only a single random matrix with

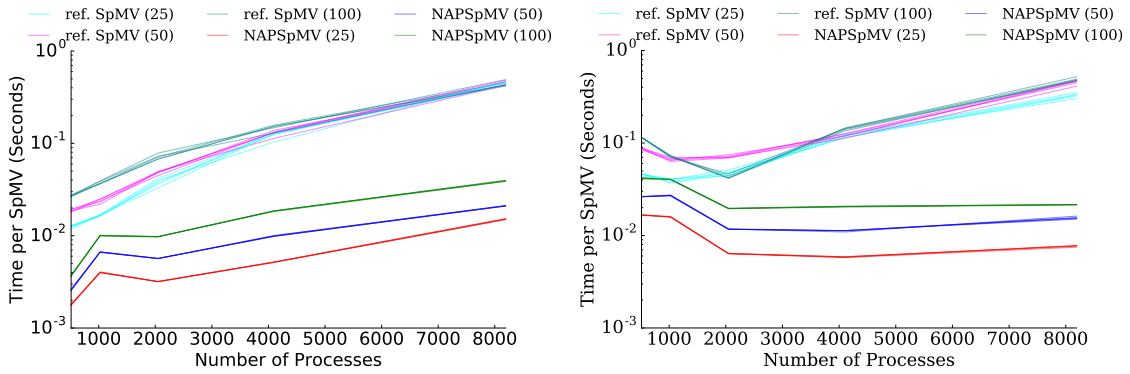


Figure 5.10: The time required to perform the various SpMVs on weakly (left) and strongly (right) scaled random matrices. Five different random matrices are tested for each density of 25, 50, and 100 non-zeros per row. The weak-scaling study tests matrices with 1 000 rows per process, while the strongly-scaled matrix contains 4 096 000 rows.

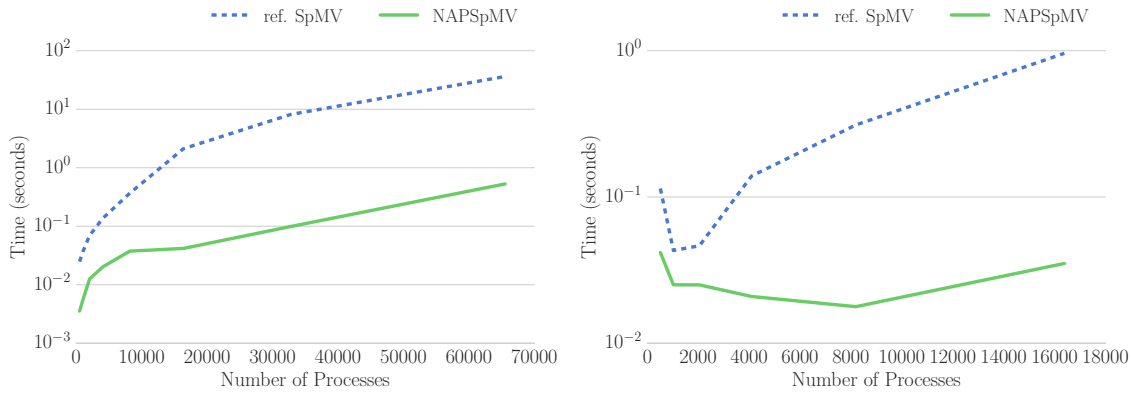


Figure 5.11: The time required to perform the various SpMVs on weakly (left) and strongly (right) scaled random matrices, each with 100 non-zeros per row. The weak-scaling study tests matrices with 1 000 rows per process, while the strongly-scaled matrix contains 4 096 000 rows.

100 non-zeros per row. Figure 5.11 displays the time required for a NAPSpMV in comparison to the standard SpMV in both weak and strong scaling studies. For these random matrices, the NAPSpMV exhibits improved performance over the reference implementation by up to two orders of magnitude and also improves scalability.

The time required to perform the various SpMVs on 13 of the 15 largest matrices from the SuiteSparse matrix collection are shown with strided and balanced partitions, in Figures 5.12 and 5.13 respectively. The remaining 2 large matrices were not included due to partitioning constraints. For the strided partitions with n_p processes, each row r is local to process $p = r \mod n_p$. As some matrices in this subset have nearly dense blocks of rows, allowing for improved load balancing over each process holding a contiguous block of rows.

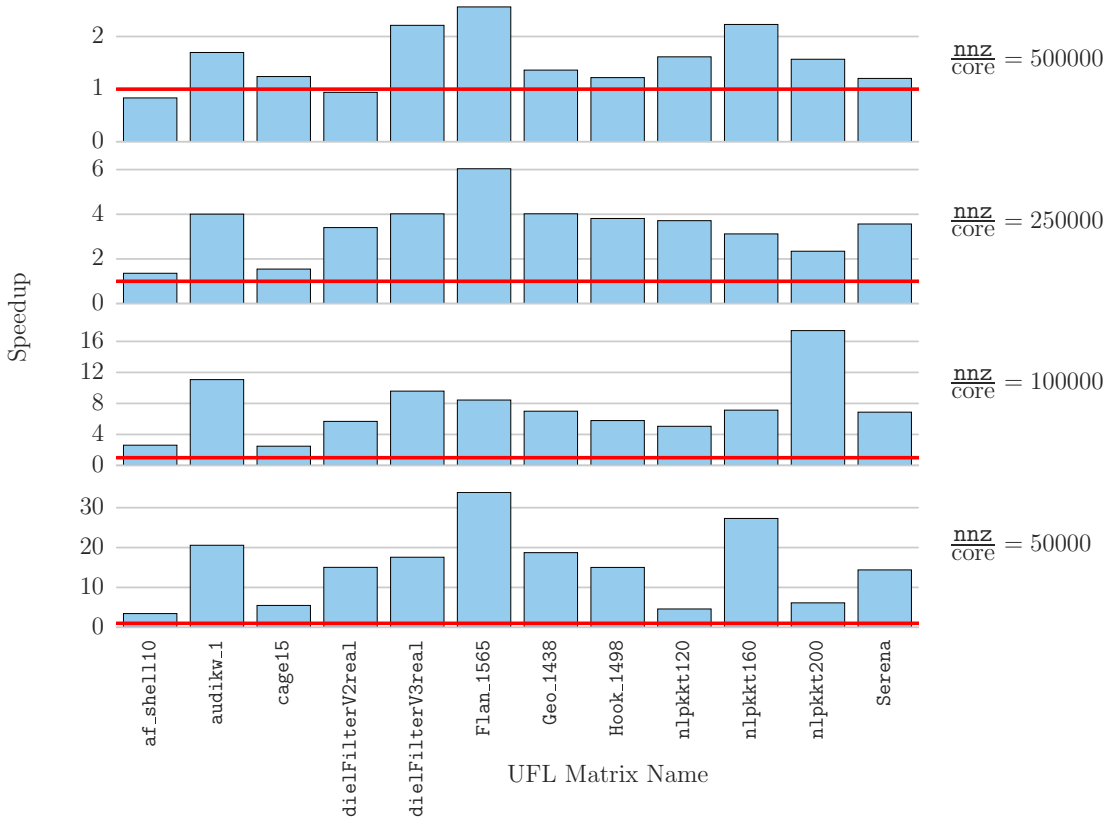


Figure 5.12: The speedup of NAPSpMVs over reference SpMVs on a subset of the largest real matrices from the SuiteSparse matrix collection at various scales, where $\frac{nnz}{\text{core}}$ is the average number of non-zeros per core, partitioned so that each row r is stored on process $p = r \bmod n_p$, where n_p is the number of processes.

The balanced partitions were formed with PT Scotch graph partitioning, using the strategy `SCOTCH_STRATBALANCE`.

The NAPSpMV improves upon many of the matrices with strided partitions, as communication patterns are far from optimal, while only minimally improving upon the graph partitioned matrices as expected. However, the cost of partitioning motivates the use of less optimal partitions when a smaller number of SpMVs are anticipated. Figure 5.14 shows the time required to perform various numbers of NAPSpMVs on both the strided and balanced partitions at the strongest scale tested, with 50 000 non-zeros per core.

In these tests, the balanced partitioned timings include the time required to graph partition and redistribute the matrix. The crossover point for the various SuiteSparse matrices, at which the graph partitioning becomes less costly than performing NAPSpMVs on strided partitions, occurs only after hundreds, or often thousands, of SpMVs have been performed.

Finally, the NAPSpMV yields improvement over other commonly used node-aware com-

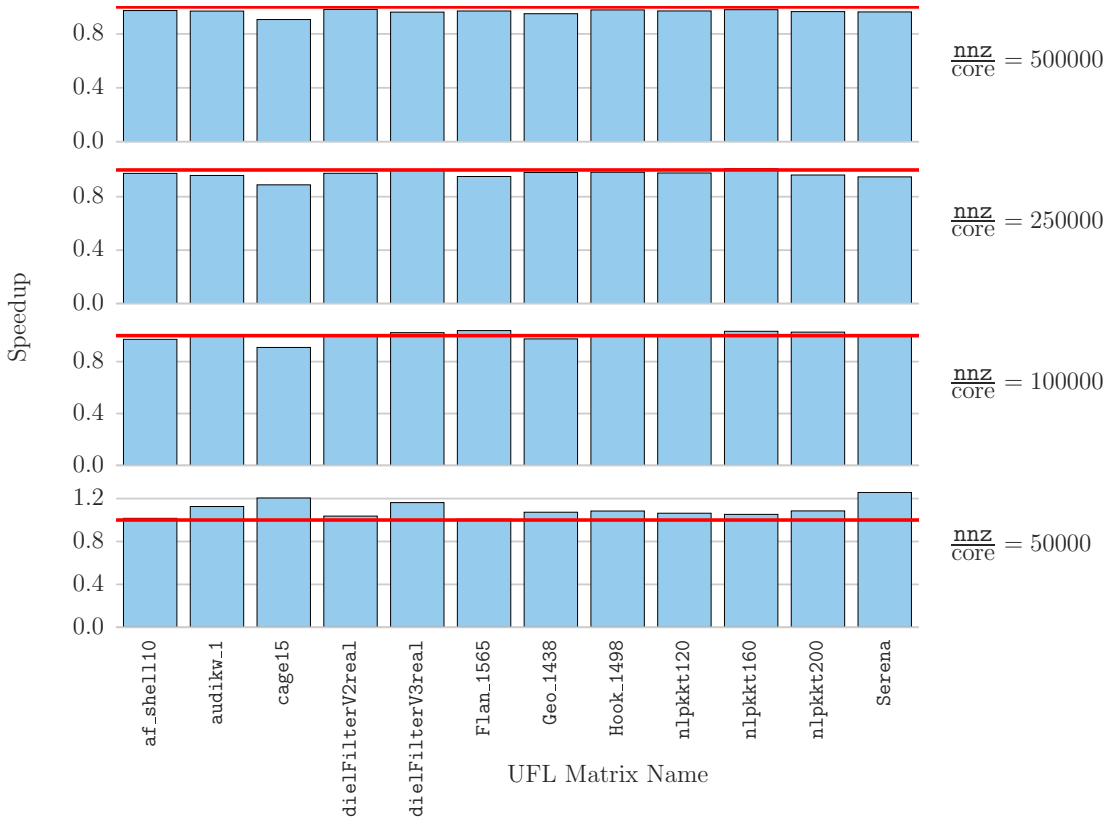


Figure 5.13: The speedup of NAPSpMVs over reference SpMVs on a subset of the largest real matrices from the SuiteSparse matrix collection at various scales, where $\frac{nnz}{\text{core}}$ is the average number of non-zeros per core, partitioned with PT Scotch.

munication, such as hybrid programming with MPI plus open multi-processing (OpenMP). Figure 5.15 shows the cost of a variety of sparse matrix-vector multiplies throughout an AMG hierarchy for linear elasticity. While hybrid programming improves over the standard SpMV on many levels of the hierarchy, the SpMV with node-aware communication yields less cost than the MPI and OpenMP combinations.

5.6 CONCLUSION AND FUTURE WORK

This section introduces a method to reduce communication that is injected into the network during a sparse matrix-vector multiply by reorganizing messages on each node. This results in a reduction of the inter-node communication, replaced by less-costly intra-node communication, which reduces both the number and size of messages that are injected into the network. The current implementation could be extended to take various levels of the hierarchy into account, such as splitting intra-node messages into on-socket and off-socket.

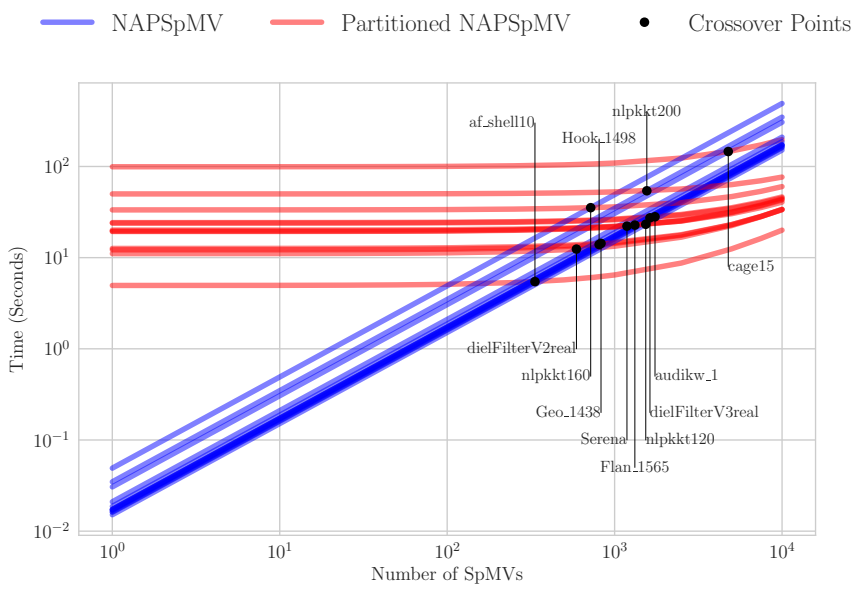


Figure 5.14: The time required to perform various numbers of NAPSpMVs on strided and balanced partitions of the largest real SuiteSparse matrices with 50 000 non-zeros per process. The time to perform a NAPSpMV on a balanced partition includes the setup cost of partitioning and redistributing the matrix. The crossover points represent the number of NAPSpMVs required before graph partitioning becomes less costly than performing NAPSpMVs on the strided partition.

Figure 2.3 shows that on-socket messages are significantly cheaper and could be targeted to further reduce communication costs.

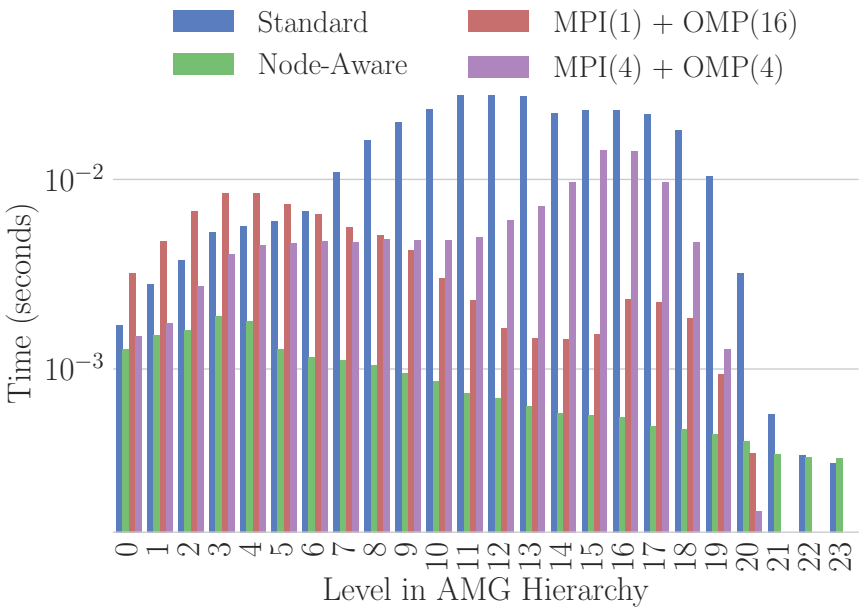


Figure 5.15: The cost of performing various SpMVs on each level of a linear elasticity hierarchy. Two hybrid programming combinations are tested, with MPI(1) + OMP(16) presenting results for one MPI process-per-node, each with 16 OpenMP threads, while the MPI(4) + OMP(4) results are for 4 MPI processes-per-node, each containing 4 threads.

CHAPTER 6: EXTENSIONS TO SPGEMM

6.1 INTRODUCTION

Sparse matrix operations are key components of numerical and graph algorithms. The sparse matrix-matrix multiply (SpGEMM) is fundamental to graph contraction [91], multiple vertex breadth first search [92], matching [93], and cycle detection [94]. Furthermore, the SpGEMM yields a dominant cost in Schur complement methods as well as the setup phase of algebraic multigrid (AMG).

The parallel SpGEMM is described as

$$C \leftarrow A \cdot B \tag{6.1}$$

where A and B are sparse input matrices, and C is the resulting sparse matrix. The matrices are partitioned across the processes, most commonly in a one-dimensional manner. A one-dimensional matrix partition consists of partitioning either the rows or columns, each displayed in Figure 6.1.

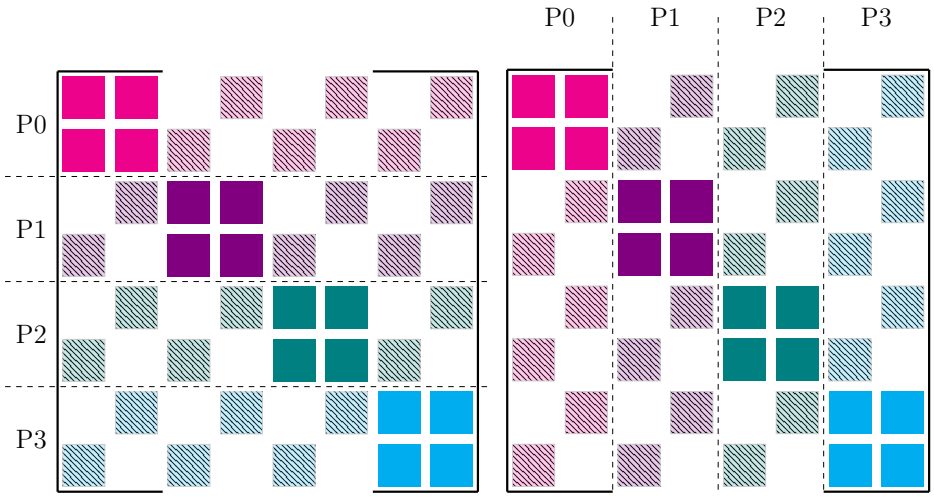


Figure 6.1: Sparse matrix distributed across four processes row-wise (left) and column-wise (right).

If A is partitioned row-wise, the rows local to a process are further partitioned into an on-process block, containing columns that correspond to local entries of B, and an off-process block, comprised of columns that are associated with entries of B that are stored on other processes. Equivalently, a column-wise partition of B is split into on-process rows corresponding to local entries of A and off-process rows associated with entries of A that are

stored on other processes.

Three possible parallel SpGEMM methods exist for one-dimensional partitions: row-wise, in which all matrices are partitioned by rows; column-wise, in which matrices are instead partition by column; and outer-product methods, consisting of a column-wise partition of A multiplied by a row-wise partition of B [95]. These approaches, which are described in detail in Section 6.2, consist of a combination of communicating matrix entries associated with non-zeros in off-process blocks and local computation. As a problem is strongly scaled, the number of rows local to each process decreases while the number of off-process columns increases, yielding decreases local computation requirements but increases in communication. Therefore, large communication requirements reduce scalability of one-dimensional SpGEMMs.

Alternatives to standard one-dimensional partitions yield increased scalability. Three-dimensional problems can be partitioned into either two or three-dimensional blocks [96]. The most common of these methods, sparse Cannon and sparse Summa, consist of partitioning into two-dimensional blocks, which are communicated to all necessary processes in a similar manner to dense multiplication [97]. Therefore, communication requires $\sqrt{p}$ messages of size $\frac{\texttt{nnz}}{p}$, yielding improved performance over sufficiently dense one-dimensional partitions. These improved communication requirements can be further reduced through 2.5D and 3D matrix partitions [98, 99]. Randomized two-dimensional partitions have also shown improvements through near optimal load balancing [100]. Furthermore, 2D and 3D matrix partitions in which only necessary matrix entries are communicated have been theoretically analyzed, indicating potential to further reduce communication costs and improve scalability [101]. Finally, the matrices can be restructured with respect to sparsity to reduce communication requirements in the congested clique model [102].

The performance and scalability of one-dimensional SpGEMMs can be improved through graph and hypergraph partitioning, in which inter-process communication edge cuts are minimized [5, 64, 65, 68]. Furthermore, the performance of each local SpGEMM can be improved through fast SpGEMM algorithms [103].

Large communication requirements, as seen in many sparse matrix-vector (SpMV) multiplies, can be reduced through the use of node-aware communication. This chapter extends the idea of the NAPSpMV to the SpGEMM. The remainder of the chapter is outlined as follows. Section 6.2 details the one-dimensional SpGEMM algorithms. Section 6.2.4 extends the idea of node-aware communication to the one-dimensional SpGEMM. Finally, code and numerics are presented in Section 6.3.

6.2 BACKGROUND

The parallel sparse matrix-matrix multiply $C \leftarrow A \cdot B$ consists of a combination of inter-process communication and local computation. The local computation consists of multiplying serial matrices. Therefore, let the function `on_process`(A) return the serial matrix corresponding to the on-process block of A while `off_process`(A) returns the off-process block. Furthermore, the function
`multiply`(`on_process`(A), `off_process`(B)) defines the serial SpGEMM of the on-process block of A times the off-process block of B, using the SMMP approach [104].

Inter-process communication requires communication of rows in the case of a row-wise matrix, or columns in the case of a column-wise matrix, corresponding to non-zero off-process entries. The three one-dimensional SpGEMM approaches are further described below.

6.2.1 Row-Wise Parallel SpGEMM

A row-wise SpGEMM consists of multiplying two matrices that are each partitioned by rows, as displayed in Figure 6.2. The function `communicate_rows(`B`)` communicates the

Figure 6.2: Row-wise partitions of all matrices in the SpGEMM $C \leftarrow A \cdot B$. The outlined columns of B must be communicated and multiplied by the outlined rows of A, forming the outlined rows of C.

rows of B corresponding to non-zero off-process columns of A. Specifically, rank r receives the rows of B, stored on other ranks, that correspond to non-zero columns in the off-process block of A on rank r. The local rows of A are multiplied by both the local rows of B as well as the rows of B received through inter-process communication. Algorithm 6.1 describes this row-wise SpGEMM in detail.

Algorithm 6.1: `row_wise_spgemm`

Input: A: row-wise parallel matrix, local to rank
B: row-wise parallel matrix, local to rank

Output: C: row-wise parallel solution matrix, local to rank

{Inter-process communication}

$R \leftarrow$ `communicate_rows(B)`

{Local matrix-matrix multiplication}

$C_{\text{on\_on}} \leftarrow$ `multiply(on_process(`A`), on_process(`B`))`
$C_{\text{on\_off}} \leftarrow$ `multiply(on_process(`A`), off_process(`B`))`
$C_{\text{off\_on}} \leftarrow$ `multiply(off_process(`A`), on_process(`R`))`
$C_{\text{off\_off}} \leftarrow$ `multiply(off_process(`A`), off_process(`R`))`

{Combine locally computed matrices}

`on_process(`C`)` $\leftarrow C_{\text{on\_on}} + C_{\text{on\_off}}$
`off_process(`C`)` $\leftarrow C_{\text{off\_on}} + C_{\text{off\_off}}$

6.2.2 Column-Wise Parallel SpGEMM

A column-wise SpGEMM is equivalent to the row-wise, but the matrices are partitioned by columns rather than rows, as displayed in Figure 6.3. Similar to `communicate_rows`, the

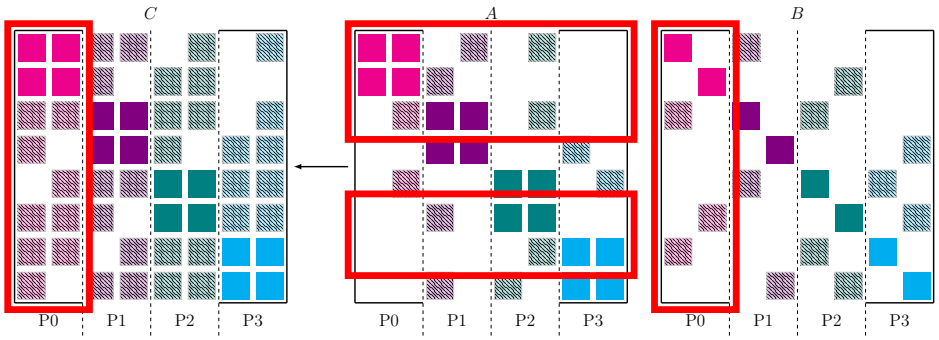


Figure 6.3: Column-wise partition of all matrices when computing the SpGEMM $C \leftarrow A \cdot B$. The outlined rows of A are communicated before being multiplied by the outlined columns of B, resulting in the outlined columns of C.

function `communicate_cols`(A) communicate the columns of A corresponding to non-zero off-process rows of B. Finally, both the local and received columns of A are multiplied by the local columns of B. The column-wise SpGEMM is described in detail in Algorithm 6.2.

Algorithm 6.2: `column_wise_spgemm`

Input: A: column-wise parallel matrix, local to rank
B: column-wise parallel matrix, local to rank

Output: C: column-wise parallel solution matrix, local to rank

{Inter-process communication}

$R \leftarrow$ `communicate_cols(A)`

{Local matrix-matrix multiplication}

$C_{\text{on\_on}} \leftarrow$ `multiply(on_process(`A`), on_process(`B`))`
$C_{\text{off\_on}} \leftarrow$ `multiply(off_process(`A`), on_process(`B`))`
$C_{\text{on\_off}} \leftarrow$ `multiply(on_process(`R`), off_process(`B`))`
$C_{\text{off\_off}} \leftarrow$ `multiply(off_process(`R`), off_process(`B`))`

{Combine locally computed matrices}

`on_process(`C`)` $\leftarrow C_{\text{on\_on}} + C_{\text{on\_off}}$
`off_process(`C`)` $\leftarrow C_{\text{off\_on}} + C_{\text{off\_off}}$

6.2.3 Outer-Product Parallel SpGEMM

The outer-product SpGEMM consists of a column-wise matrix A multiplied by a row-wise matrix B, as displayed in Figure 6.4. The local rows of A are multiplied by the

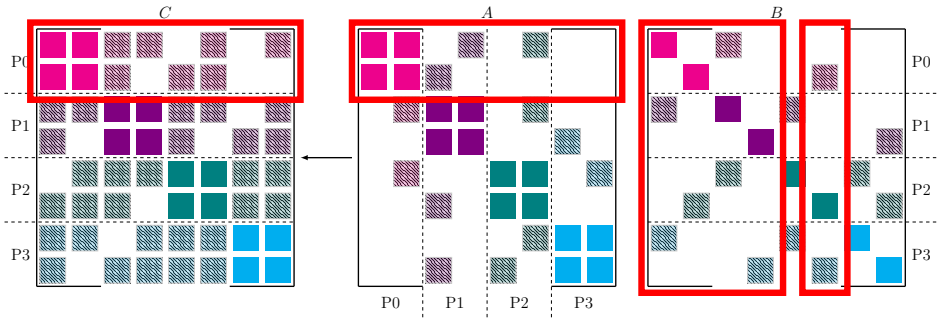


Figure 6.4: The SpGEMM $C \leftarrow A \cdot B$ is computed with a column-wise partition of A, and row-wise partitions of B and C. Partial products of C are computed with the local columns or rows of A and B, before being communicated and summed.

local columns of B, yielding partial results for the non-zeros of C. Assuming a row-wise partition of the resulting matrix C, the partial results of rows of C that are stored on distant processes must be communicated. Equivalently, in a column-wise partition of C, partial columns of C must similarly be communicated. Therefore, the function `communicate_T(C)` communicate the partial results of C to the appropriate processes, yielding the transpose of `communicate_rows`. Finally, all partial results corresponding to either local rows of a row-wise C, or local columns of a column-wise C, are combined. This algorithm is described

in detail in Algorithm 6.3.

Algorithm 6.3: `outer_product_spgemm`

Input: A: column-wise parallel matrix, local to rank
B: row-wise parallel matrix, local to rank

Output: C: parallel solution matrix, local to rank

{Local matrix-matrix multiplication}

$C_{\text{on\_on}} \leftarrow \texttt{multiply}(\texttt{on\_process}(A), \texttt{on\_process}(B))$
$C_{\text{on\_off}} \leftarrow \texttt{multiply}(\texttt{on\_process}(A), \texttt{off\_process}(B))$
$C_{\text{off\_on}} \leftarrow \texttt{multiply}(\texttt{off\_process}(A), \texttt{on\_process}(B))$
$C_{\text{off\_off}} \leftarrow \texttt{multiply}(\texttt{off\_process}(A), \texttt{off\_process}(B))$

{Inter-process communication}

$S \leftarrow C_{\text{on\_on}} + C_{\text{on\_off}}$
$R \leftarrow \texttt{communicate\_T(S)}$

{Combine locally computed matrices}

$\texttt{on\_process}(C) \leftarrow C_{\text{on\_on}} + \texttt{on\_process}(R)$
$\texttt{off\_process}(C) \leftarrow C_{\text{on\_off}} + \texttt{off\_process}(R)$

6.2.4 Node-Aware Communication in the SpGEMM

Communication of sparse matrices during an SpGEMM operation requires a point-to-point communication pattern similar to that of the SpMV communication described in Algorithm 5.1. In a row-wise SpGEMM $C \leftarrow A \cdot B$, the rows of B corresponding to off-process columns of A are communicated. Similarly, column-wise SpGEMM communication consists of sending columns of A corresponding to off-process rows in B. Finally, outer-product multiplication consists of sending all partial products of C to the appropriate process. In other words, values of C resulting from multiplication of an off-process row i of A are sent to the process holding the corresponding column i of A.

Standard communication of matrix rows or columns consists of communicating each row size, followed by the corresponding column indices and data values. This data is typically sent directly between processes as a single message. However, this results in duplicate messages being communicated between pairs of nodes, as described in Example 5.2.

One-dimensional partitions of the matrices in $C \leftarrow A \cdot B$ can be updated to include node-awareness. In a row-wise SpGEMM, the off-process columns of A can be further partitioned into on-node columns, corresponding to rows of B that are stored on processes located on the same node, and off-process columns associated with rows of B stored on other nodes. This

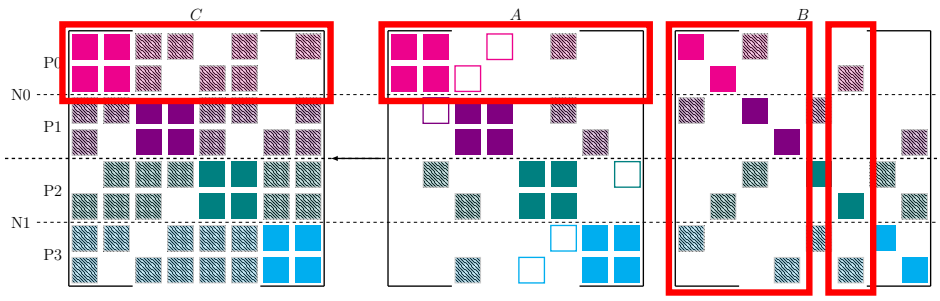


Figure 6.5: Row-wise node-aware SpGEMM. The off-process columns of A are split into on-node columns, containing the the outlined blocks, and off-node columns, consisting of the patterned blocks.

is exemplified in Figure 6.5. Similarly, the column-wise SpGEMM is updated so that the off-process rows of B are split into on-node and off-node rows, as shown in Figure 6.6. Finally,

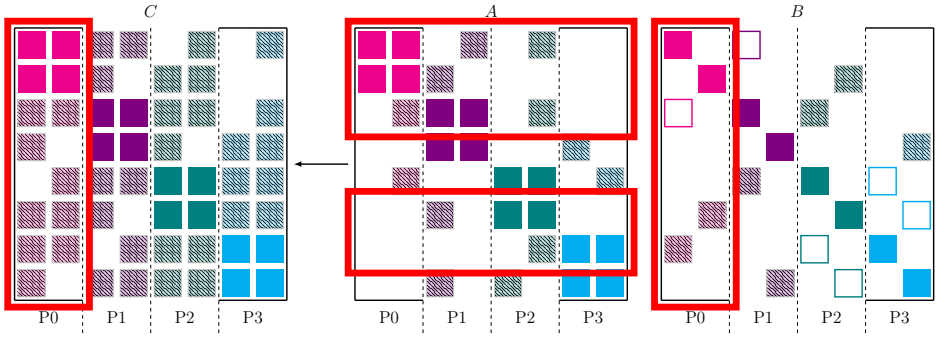


Figure 6.6: Column-wise node-aware SpGEMM. The off-process rows of B are split into on-node and off-node blocks.

the rows of C are updated in the outer-product SpGEMM so that off-process columns are split into on- and off-node, as displayed in Figure 6.7.

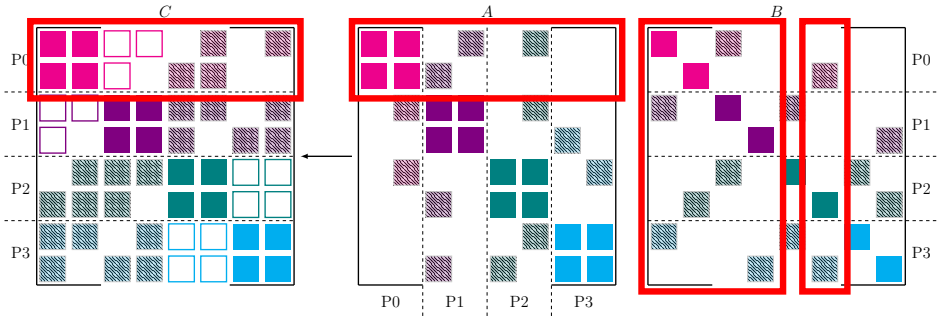


Figure 6.7: Outer-product node-aware SpGEMM. The off-process columns of C are partitioned into on- and off-node.

Node-aware communication can be extended to the SpGEMM by communicating rows or

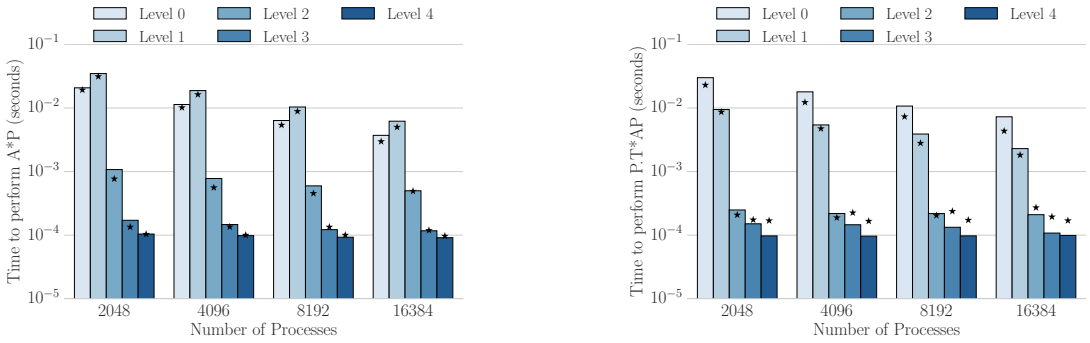
columns of the sparse matrix with the three-step algorithm described in Example 5.3. All data is first gathered on-node, before being injected into the network as a single message, and finally redistributed on the receiving node. As each node typically communicates with many other nodes, all processes remain active in inter-node communication. This process greatly reduces the number and size of inter-node communication at an increase of less-costly intra-node communication.

6.3 RESULTS

Node-aware SpGEMMs are tested for AMG hierarchies that arise from a range of problems. Two SpGEMMs are required in the formation of each coarse level of the AMG hierarchy. First $AP \leftarrow A \cdot P$ performs row-wise multiplication. This product is then used in the outer product SpGEMM $A_c \leftarrow P^T \cdot (AP)$ to form the coarse grid operator, as proven the most efficient method by Ballard et al. [105].

The SpGEMM operations were tested throughout a variety of hierarchies, created with RAPtor [34], for solving MFEM example matrices. The figures plot the node-aware SpGEMM timings with bars, and corresponding reference SpGEMM timings are shown as stars.

Figure 6.8 shows the effects of node-aware communication in the SpGEMMs of the Laplacian hierarchy, from MFEM example 1. Similarly, node-aware speedup in the hierarchy for

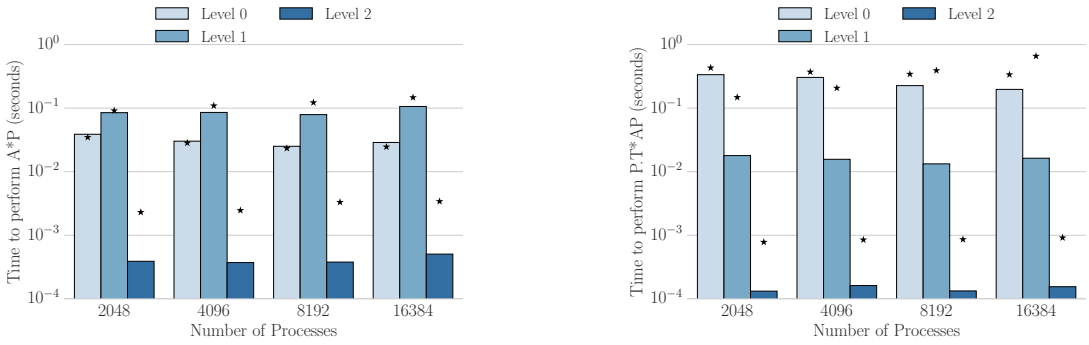


(a) $A \cdot P$ (b) $P^T \cdot (AP)$

Figure 6.8: MFEM Laplacian AMG hierarchy.

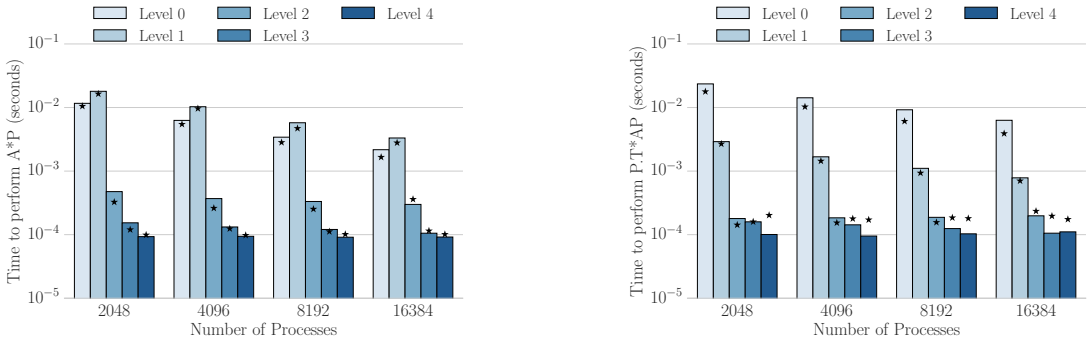
the linear elasticity system from MFEM example 2 is displayed in Figure 6.9. Speedups for the grad-div problem from example 4 are shown in Figure 6.10. Finally, Figures 6.11 and 6.12 show timings for discontinuous galerkin discretizations of diffusion and linear elasticity problems, respectively.

The node-aware SpGEMM was tested for a number of matrices from the Stanford large network dataset (SNAP) collection, including the Amazon product co-purchasing directed



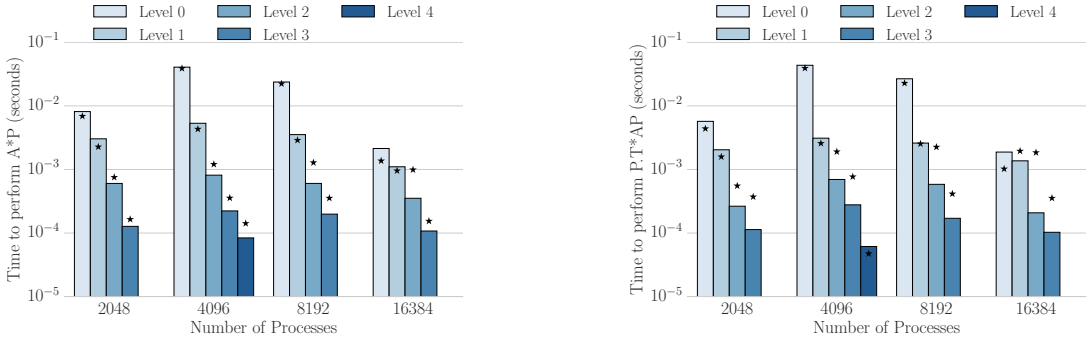
(a) $A \cdot P$ (b) $P^T \cdot (AP)$

Figure 6.9: Linear elasticity AMG hierarchy timings.



(a) $A \cdot P$ (b) $P^T \cdot (AP)$

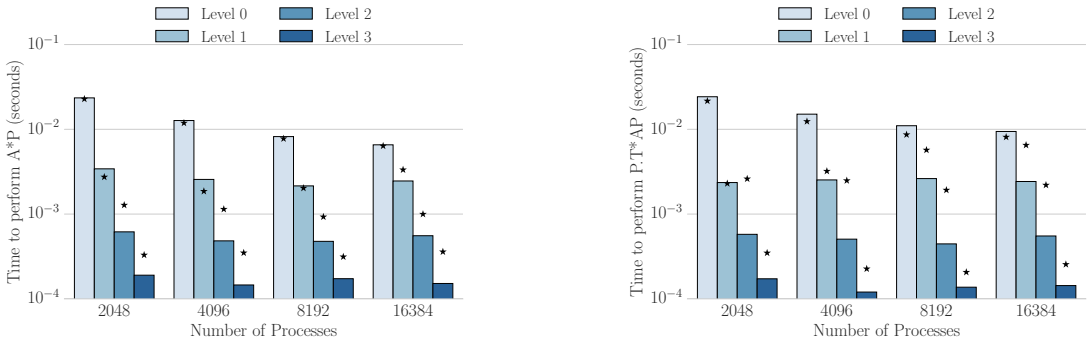
Figure 6.10: Timings for Grad-Div throughout AMG hierarchy.



(a) $A \cdot P$ (b) $P^T \cdot (AP)$

Figure 6.11: Timings for Discontinuous Galerkin Diffusion hierarchy.

networks, the road networks, the citation network among US patents, DBLP collaboration network, and Facebook and Twitter networks. The cost of row-wise multiplication $A^T \cdot A$ on this subset of matrices is presented in Figure 6.13. Furthermore, Figure 6.14 displays the performance of standard and node-aware communication throughout outer-product multipli-



(a) $A \cdot P$ (b) $P^T \cdot (AP)$

Figure 6.12: SpGEMM times throughout Discontinuous Galerkin Linear Elasticity AMG hierarchy.

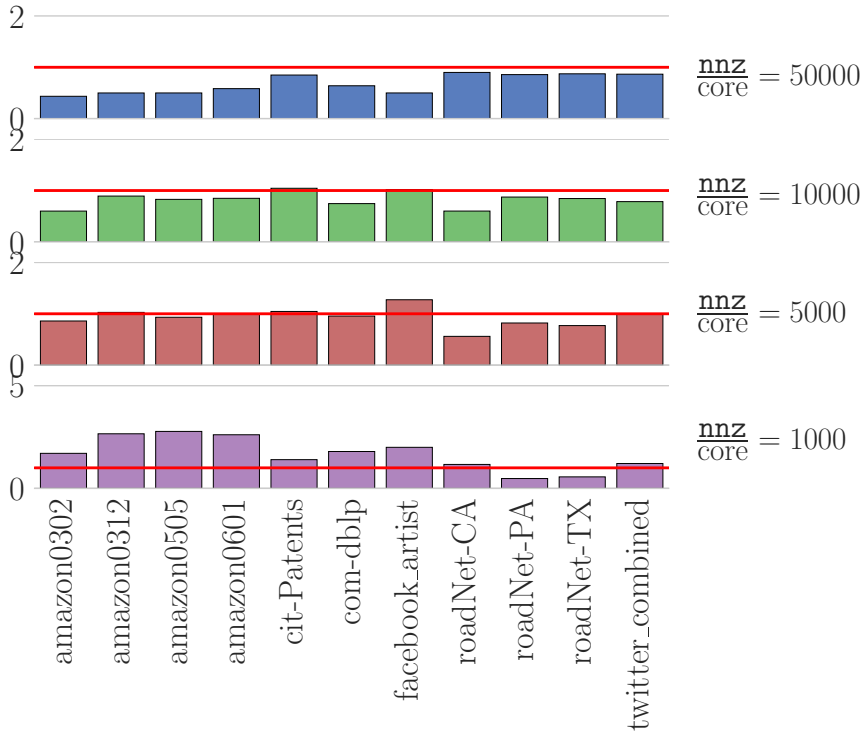


Figure 6.13: Row-wise multiplication of $A^T \cdot A$ for a subset of SNAP matrices.

cation of $A^T \cdot A$ for the same subset from the SNAP collection. Node-aware communication yields improvement in both operations for the majority of the subset, particularly when strongly scaled to 1 000 non-zeros per row.

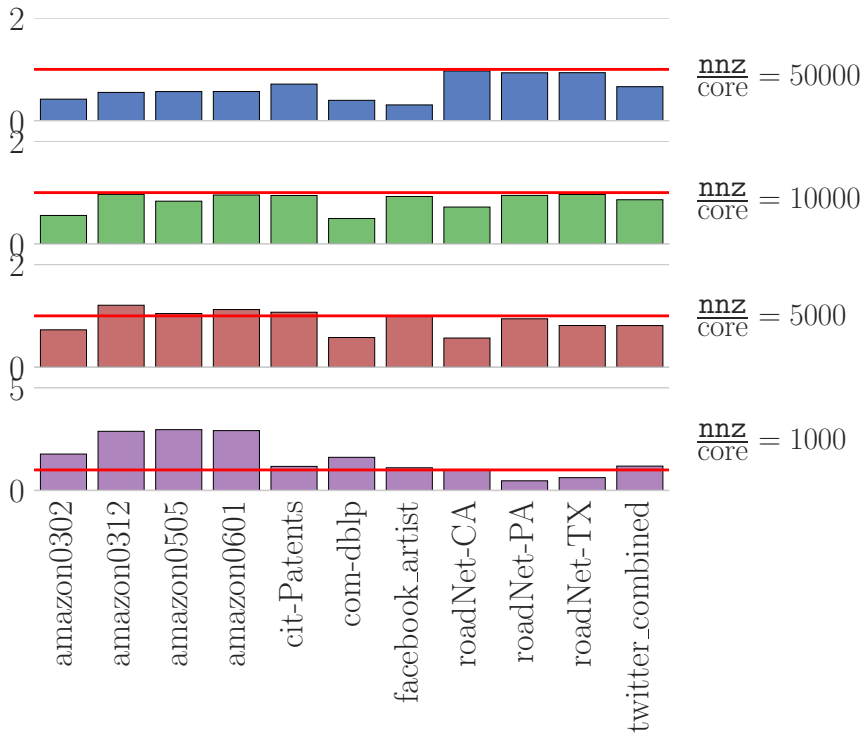


Figure 6.14: Outer-product SpGEMMs performing $A^T \cdot A$ on a subset of SNAP matrices.

6.4 CONCLUSION

Node-aware communication improves the performance and strong scalability of a variety of matrices, particularly those with large communication patterns such as the coarse levels of AMG hierarchies. Finer levels, as well as SNAP matrices with large amounts of sparsity, yield little improvement, and often acquire slowdown from node-aware communication. Therefore, future directions include creating performance models to determine when matrices have large enough communication requirements to warrant node-aware communication throughout SpGEMM operations.

CHAPTER 7: NODE-AWARE AMG

Portions of this chapter appear in the paper "Improving Strong Scalability of Parallel Algebraic Multigrid through Message Agglomeration", in submission to SISC [106].

7.1 INTRODUCTION

Standard parallel algebraic multigrid typically exhibits strong scaling to five or ten thousand degrees-of-freedom per core before communication costs outweigh local computation. Further extending the core-count yields an increase in total solve time due to dominant communication costs. Figure 7.1 shows the time required to solve a rotated anisotropic diffusion problem with 10-million rows at a variety of scales, partitioned into local computation and inter-process communication costs. As the processor count reaches the strong scaling limit, communication becomes increasingly dominant. The problem scales to 1 024 processes, after which communication costs outweigh any reductions in local computation.

Most methods for reducing communication costs in AMG focus on a redesign of the method or on the underlying sparse matrix operations. Aggressive coarsening, for example, reduces the dimensions of coarse levels at a faster rate, yielding reduced density and communication requirements [39, 40, 44]. Similarly, the smoothed aggregation solver allows large aggregates, coarsening a larger number of fine points into a single coarse point [107, 46]. Small nonzeros resulting from fill-in on coarse levels may be systematically removed, adding sparsity into coarse-grid operators [15, 46, 35]. Furthermore, matrix ordering and graph partitioning yield reduced communication costs throughout sparse matrix operations [64, 65, 7, 8, 68], and coarse level repartitioning has potential to reduce the cost of the solver. Likewise, coarse level redistribution and duplication of the coarsest level solves yield large reductions in communication time. The approach presented in this chapter augments these approaches, reducing off-node message counts and sizes through aggregation of data.

Topology-aware methods and message agglomeration are commonly used to reduce communication costs in MPI applications. Topology-aware task mapping minimizes message hop counts, reducing the cost associated with communication [108, 109]. Message agglomeration is commonly used to reduce the cost of communication, for example in MPI collectives [74, 75, 76, 77]. The Tram library [78] explores agglomeration of point-to-point messages, by streamlining messages between neighboring processes [78].

This chapter presents a method for reducing communication costs in both the setup and solve phases of parallel algebraic multigrid through agglomeration of messages among nodes.

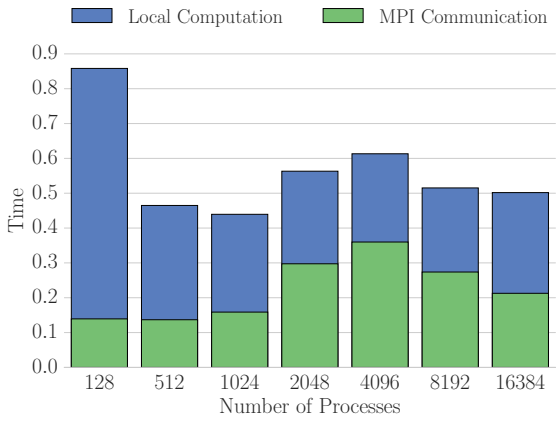


Figure 7.1: The total time required to solve an anisotropic diffusion problem with 10-million degrees-of-freedom.

Section 7.2 covers algebraic multigrid and common parallel implementations. Section 7.4 covers numerical experiments in support of the approach and Section 7.5 contains concluding remarks and future directions.

7.2 BACKGROUND

Common algorithms for constructing an algebraic multigrid hierarchy include the Ruge-Stüben solver, described in Section 7.2.1, and the smoothed aggregation solver, outlined in Section 7.2.2. Both solvers consist of a combination of local computation and point-to-point communication, namely communication of vectors and sparse matrices.

This *point-to-point* communication [110] dominates the total time (both setup and solve) in coarse levels of AMG, as show in Figure 7.2. The cost associated with communication increases on coarse levels.

The next two sections investigate point-to-point communication in both Ruge-Stüben and Smoothed Aggregation AMG throughout two example systems, the rotated anisotropic diffusion problem presented in Example 7.1 and the 27-point Laplacian in Example 7.2.

Example 7.1. *A two-dimensional rotated anisotropic diffusion problem $K = Q^T D Q$ is analyzed, where the rotation matrix Q and diagonal scaling D are defined as*

$$Q = \begin{pmatrix} \cos(\theta) & \sin(\theta) \\ -\sin(\theta) & \cos(\theta) \end{pmatrix} \qquad D = \begin{pmatrix} 1 & 0 \\ 0 & \epsilon \end{pmatrix}, \tag{7.1}$$

with $\theta = \frac{\pi}{4}$ and $\epsilon = 0.001$. The linear system has 10-million degrees-of-freedom, unless otherwise specified. The Ruge-Stüben hierarchy is created with Falgout coarsening and mod-

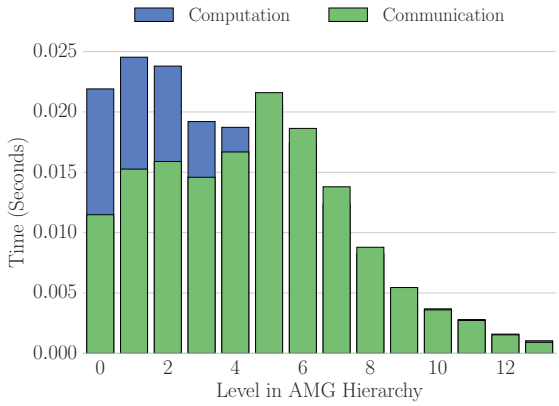


Figure 7.2: The total cost of AMG per level.

ified classical interpolation. A strength threshold of 0.25 is used for both Ruge-Stüben and smoothed aggregation hierarchies for this system.

Example 7.2. *A three-dimensional, structured* 27*-point Laplacian system,* $\triangle u = 0$*, with* 10*-million degrees-of-freedom is solved with both Ruge-Stüben and smoothed aggregation AMG. As fill-in greatly reduces performance of 3D systems, aggressive coarsening is used when creating the Ruge-Stüben hierarchy, specifically HMIS coarsening and extended+i interpolation.*

7.2.1 Classical Ruge-Stüben Setup

For an $n \times n$ matrix A, Algorithm 7.1 outlines a typical Ruge-Stüben algebraic multigrid setup routine.

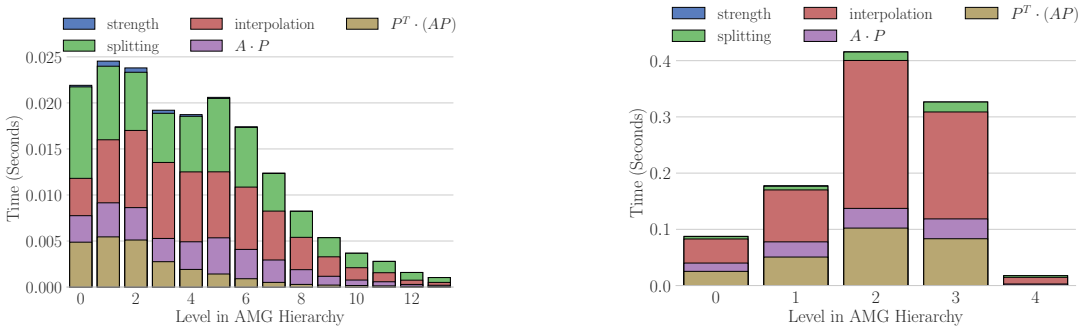
Algorithm 7.1: Classical Setup: `rs_setup`

Input: A // sparse system matrix
max_coarse // maximum size of coarse operators

Output: $A_0,\ A_1,\ \ldots,\ A_N$ // hierarchy of sparse operators
$P_0,\ P_1,\ \ldots,\ P_{N-1}$ // hierarchy of interpolation operators

1 $A_0 \leftarrow A$
2 $\ell \leftarrow 0$
3 **while** $|A_\ell| >$ max_coarse
4 $\quad S_\ell \leftarrow$ `strength`(A_ℓ) {strength-of-connection}
5 $\quad C_\ell, F_\ell \leftarrow$ `cf_splitting`(S_ℓ) {coarse-fine splitting}
6 $\quad P_\ell \leftarrow$ `interpolation`(C_ℓ, F_ℓ) {interpolation}
7 $\quad A_{\ell+1} \leftarrow P_\ell^T \cdot A_\ell \cdot P_\ell$ {coarse operator, Galerkin product}
8 $\quad \ell \leftarrow \ell + 1$

The remainder of this subsection further investigates the cost of each method used to create a Ruge-Stüben hierarchy for e on 8192 processes of Blue Waters. Classical parameters are analyzed for a two-dimensional rotated anisotropic diffusion system with 10-million degrees-of-freedom, while aggressive coarsening methods are profiled for a 27-point Laplacian system with 10-million rows. The cost of the various methods in these hierarchies, partitioned by level, is displayed in Figure 7.3.



(a) Anisotropic diffusion from Example 7.1. (b) 27-point Laplacian from Example 7.2.

Figure 7.3: Profile of Ruge-Stüben AMG setup costs from `rs_setup`.

Strength of Connection: `strength` The `strength` method identifies the strength-of-connection between nodes of A based on a tolerance θ. The classical `strength` method retains an entry $(i, j) \in A$ if the diagonal value A_{ii} is positive and

$$A_{ij} < \theta \cdot \min_{k \neq i}(A_{ik}) \tag{7.2}$$

or the diagonal A_{ii} is negative and

$$A_{ij} > \theta \cdot \max_{k \neq i}(A_{ik}). \tag{7.3}$$

In parallel, this operation is fully local, as entries in a row i of S depend only on other entries in row i. This cost, partitioned across levels, is displayed in Figure 7.4.

Coarse-Fine Splitting: `splitting` The CF `splitting` method partitions the index set based on the strength matrix, S_ℓ to yield to sets C and F with $\{0, \ldots, n-1\} = C \cup F$ and $C \cap F = \emptyset$.

There are a variety of methods in parallel [12] that consist of splittings on local partitions, yielding a large number of coarse points along processor boundaries along with partitioning

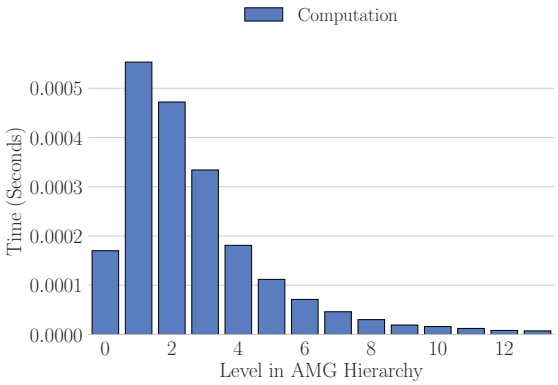


Figure 7.4: Cost of `strength` for Example 7.1.

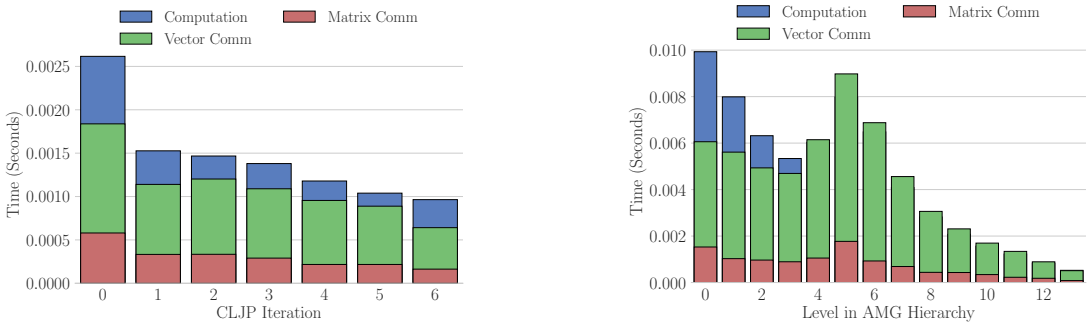
constraints limiting the minimum partition count to the number of processes. The Cleary-Luby-Jones-Plassmann (CLJP) algorithm for `splitting` is commonly used: Each initial weight w_i is the sum of the number of nodes strongly influenced by i, $\left|S_i^T\right|$ and a random value between zero and one. Therefore, each weight is unique allowing for many coarse points to be selected each iteration. After setting initial weights, CLJP iteratively selects coarse points through a combination of selecting independent sets and updating corresponding weights. An independent set D is selected to contain every node i such that the weight of i is larger than that of all neighbors. This is defined as

$$D = \left\{ i \; w_i > w_j \forall j \in S_i \bigcup S_i^T \right\}. \tag{7.4}$$

Neighboring weights are then updated such that the weight of all nodes j that strongly influence any new coarse point $i \in D$ are decreased. Furthermore, if two nodes j and k both depend on an $i \in D$, and j strongly influences k, then the weight w_j is decreased.

A large amount of vector communication is associated with CLJP. Column sizes are communicated to form initial weights, followed by communication of calculated weights. During each iteration, the maximum weight of each column is communicated before determining the independent set D. After forming D, updated coarse states must be communicated, and after weights are updated they too must be communicated. Each iteration also consists of sending a partial sparsity pattern of the matrix S, containing all coarse nodes influenced by each off-process column. Figure 7.5a displays the cost of each iteration of CLJP, partitioning each into both vector communication and matrix sparsity pattern communication. Furthermore, the total cost of CLJP on each level of the anisotropic hierarchy is displayed in Figure 7.5b.

Alternative approaches such as Falgout coarsening can yield improved convergence over



(a) Per-iteration cost of finest level.

(b) Per-level cost of full method.

Figure 7.5: Profile of CLJP `splitting` for Example 7.1

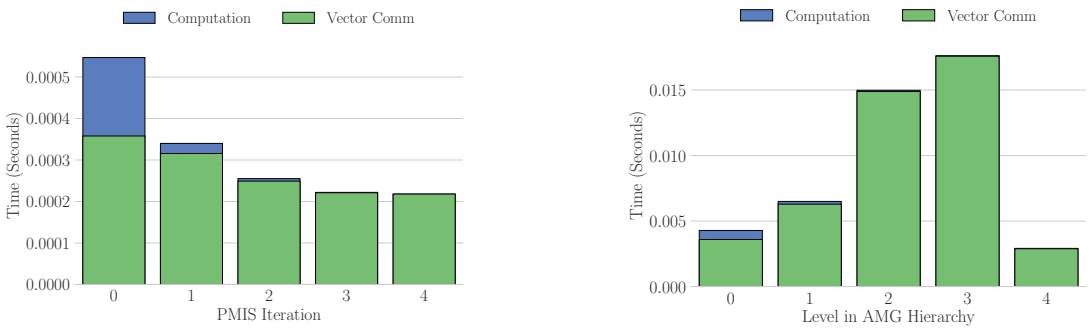
CLJP by using classic Ruge-Stüben coarsening on interior points followed by CLJP on boundary nodes. In addition, data structures may be optimized in order to reduce communication volume [111]

Three dimensional systems yield complex hierarchies when coarsened with standard approaches, such as CLJP or Falgout coarsening. Therefore, 3D systems are coarsened more aggressively with approaches such as Parallel Maximal Independent Set (PMIS). PMIS is similar to CLJP, with identical initial weights and independent set selection. However, restrictions are relaxed on fine points, allowing all nodes strongly influenced by any coarse point to be fine. Therefore, each iteration consists of selecting the independent set D of new coarse points, defined in 7.4. For each new coarse point $i \in D$, all nodes $k \in S_j^T$ are selected as fine points.

PMIS requires only of vector communication. The initial weights require previously described communication of column sizes and calculated weights. Each iteration requires only communication maximum column weights followed by updated states after selecting new coarse points and again after adding to the set F. Figure 7.6a displays the cost of each PMIS iteration in terms of both local computation and vector communication. Furthermore, the total cost of PMIS for the 27-point Laplacian, partitioned by level, is displayed in Figure 7.6b, and shows that vector communication dominates the cost.

Forming Interpolation: `interpolation` After a splitting is constructed, interpolation and restriction operators are formed to transfer data between fine and coarse levels. These operators inject each coarse point with a linear combination of neighboring fine points.

$$(P_\ell \cdot e)_i = \begin{cases} e_i & \text{If } i \in C \\ w_{ij} e_j & \text{If } i \in F \end{cases} \tag{7.5}$$



(a) Per-iteration cost of finest level.

(b) Full method cost per-level.

Figure 7.6: Profile of PMIS `splitting` for Example 7.2

The weights associated with neighboring fine points vary with method of interpolation. Direct interpolation yields simple weights and requires no communication in parallel, but often yields poor convergence. The focus here is on modified classical interpolation for standard coarse-fine splittings and extended classical interpolation in combination with aggressive coarsening as both methods yield convergent algorithms.

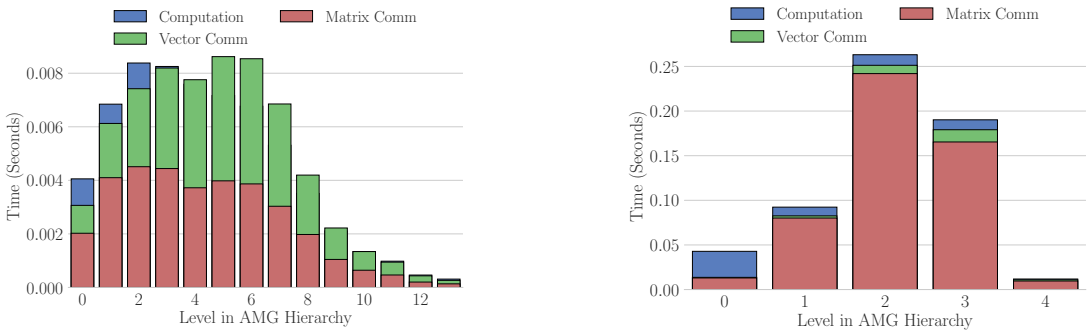
The modified classical interpolation weights are defined as

$$w_{ij} = -\frac{1}{a_{ii} + \sum_{k \in N_i^w \bigcup F_i^{s*}} a_{ik}} \left(a_{ij} + \sum_{k \in F_i^S \setminus F_i^{s*}} \frac{a_{ik} \cdot \overline{a}_{kj}}{\sum_{m \in C_i^s} \overline{a}_{km}} \right), \tag{7.6}$$

where N_i^w is the set of weakly connected neighbors of i, F_i^{s*} contains all strongly connected fine points that do not have a common neighbor with i, F_i^s are the remaining strongly connected fine points, and C_i^s are the strongly connected coarse points. Furthermore,

$$\overline{a}_{ij} = \begin{cases} 0 & sign(a_{ij}) = sign(a_{ii}) \\ a_{ij} & \text{otherwise.} \end{cases} \tag{7.7}$$

Parallel modified classical interpolation requires matrix communication of all coarse points in S as associated values in A, necessary to determine $\sum_{m \in C_i^s} \overline{a}_{km}$ for all $k \in F_i^s \setminus F_i^{s*}$. Communication is performed at the beginning of the method along with communication of initial states; all subsequent work is fully local. After the interpolation operator is created, the associated communication pattern is determined, requiring vector communication. This cost, as required for the anisotropic system, is displayed in Figure 7.7a, where point-to-point communication dominates, with nearly equal portions due to vector and matrix communication.



(a) Modified classical `interpolation` cost per level for Example 7.1.

(b) Extended `interpolation` for a PMIS splitting for Example 7.2.

Figure 7.7: Profile of `interpolation` methods.

Classical interpolation does not accurately interpolate the necessary distance-two neighbors for optimal convergence with aggressive coarsening. Therefore, extended interpolation is used in combination with PMIS and HMIS to more accurately project data to coarse levels. Extended interpolation weights are defined as

$$w_{ij} = -\frac{1}{\widetilde{a}_{ii}} \left(a_{ij} + \sum_{k \in F_i^s} \frac{a_{ik} \cdot \overline{a}_{kj}}{\sum_{\ell \in \hat{P}_i \bigcup \{i\}} \overline{a}_{k\ell}} \right), j \in \hat{C}_i, \tag{7.8}$$

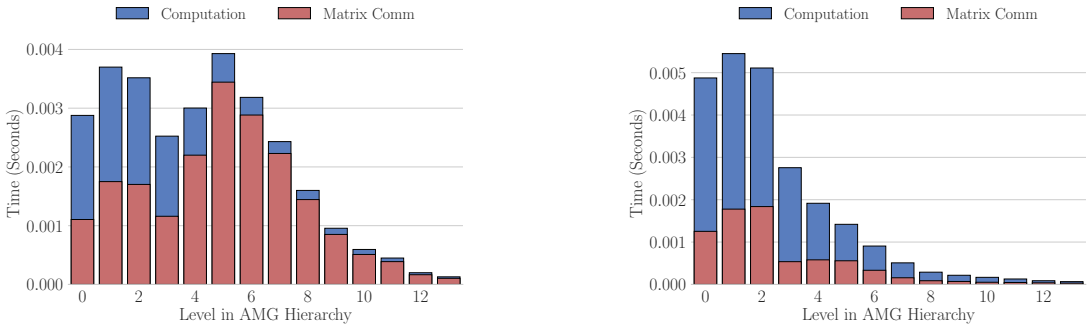
where $\hat{C}_i = C_i \bigcup C_j$ for all strong fine neighbors $j \in F_i^s$ and

$$\widetilde{a}_{ii} = a_{ii} + \sum_{n \in N_i^w} a_{in} + \sum_{k \in F_i^s} a_{ik} \frac{\overline{a}_{ki}}{\sum_{\ell \in \hat{C}_i \bigcup \{i\}} \overline{a}_{k\ell}}. \tag{7.9}$$

Parallel extended interpolation requires the matrix A to be communicated as a weight w_{ij} depends on strong neighbors similar to classical interpolation, and a subset of weak neighbors is also required to determine $\sum_{\ell \in \hat{C}_i \bigcup \{i\}} \overline{a}_{k\ell}$. The cost of extended interpolation for the 27-point Laplacian hierarchy is profiled in Figure 7.7b.

Galerkin Triple Matrix Product: $P^T \cdot A \cdot P$ The Galerkin triple matrix product is often implemented as two separate SpGEMMs, $AP \leftarrow A \cdot P$, a row-wise multiply, and $Ac \leftarrow P^T \cdot AP$, an outer-product SpGEMM. The former row-wise multiplication consists of communicating the off-process portion of P before performing matrix multiplication on local and received portions of A and P. The outer-product multiply consists of performing partial products of A_c with the local columns of P^T and rows of AP, before communicating the portions of A_c that are stored on other processes. Finally, local partial products of A_c

are combined with those calculated on distant processes. Therefore, each SpGEMM requires a single instance of matrix communication. The cost of each SpGEMM required during construction of the anisotropic diffusion hierarchy is displayed in Figure 7.8.



(a) Cost of $A \cdot P$.

(b) Cost of $P^T \cdot (AP)$.

Figure 7.8: Cost of $P^T \cdot A \cdot P$ per level for Example 7.1

7.2.2 Smoothed Aggregation Setup

Smoothed aggregation AMG (Algorithm 7.2) consists of grouping strongly connected nodes to identify a coarse grid. Transfer operators between levels are initially formed through the aggregation pattern and by injecting candidate vectors. Finally, the tentative interpolation operators are smoothed through Jacobi smoothing before creating the coarse grid operator through the previously described Galerkin triple matrix product.

The cost of each method required to create a smoothed aggregation hierarchy for the anisotropic diffusion system is displayed in Figure 7.9. The cost of the setup phase is parti-

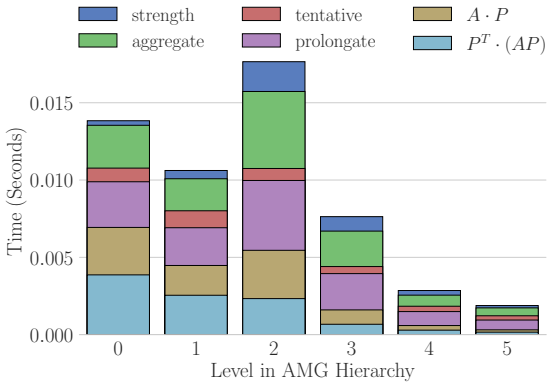


Figure 7.9: Smoothed Aggregation setup costs from `sa_setup` for Example 7.1.

Algorithm 7.2: Smoothed Aggregation Setup: `sa_setup`

Input: A // sparse system matrix
B // candidate vectors
max_coarse // maximum size of coarse operators

Output: $A_0, A_1, \ldots, A_N$ // hierarchy of sparse operators
$P_0, P_1, \ldots, P_{N-1}$ // hierarchy of interpolation operators

```
1  A_0 ← A
2  B_0 ← B
3  ℓ ← 0
4  while |A_ℓ| > max_coarse
5      S_ℓ ← strength(A_ℓ)                          // strength-of-connection
6      Agg_ℓ ← aggregate(S_ℓ)                       // form coarse aggregates
7      T_ℓ, B_{ℓ+1} ← tentative(Agg_ℓ, B_ℓ)         // tentative interpolation
8      P_ℓ ← prolongate(A_ℓ, T_ℓ)                   // smooth interpolation
9      A_{ℓ+1} ← P_ℓ^T · A_ℓ · P_ℓ                  // coarse operator, Galerkin product
10     ℓ ← ℓ + 1
```

tioned more evenly (in comparison to Ruge-Stüben) across the methods, with the majority of the cost occurring on coarse levels.

Symmetrized Strength of Connection: `strength` The `strength` method used in smoothed aggregation is similar to that of Section 7.2.1. However, `aggregation` requires an undirected graph as input thus necessitating a symmetric sparsity pattern. The smoothed aggregation `strength` method retains any entry $(i, j) \in A$ if any of the following are true.

- $A_{ij} < \theta \cdot \min_{k \neq i}(A_{ik})$ with $A_{ii} > 0$
- $A_{ij} < \theta \cdot \min_{k \neq j}(A_{jk})$ with $A_{jj} > 0$
- $A_{ij} > \theta \cdot \max_{k \neq i}(A_{ik})$ with $A_{ii} < 0$
- $A_{ij} > \theta \cdot \max_{k \neq j}(A_{jk})$ with $A_{jj} < 0$

This symmetrized `strength` method requires communication of the maximum off-diagonal for all rows with negative diagonal, along with the minimum off-diagonal for each row with positive diagonal. Furthermore, a vector is indicating whether each diagonal is positive or negative is also communicated. As a result, only vector communication is required, and a profile is given in Figure 7.10.

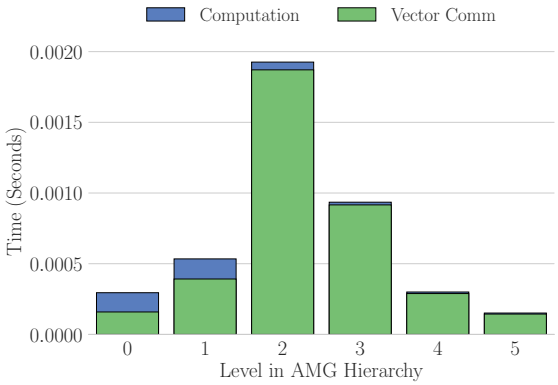
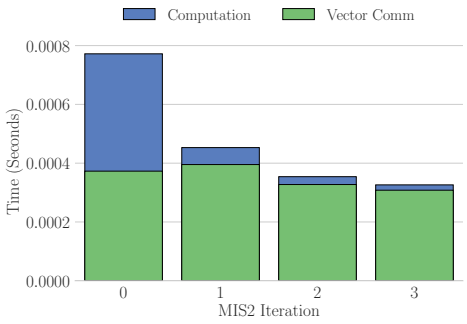


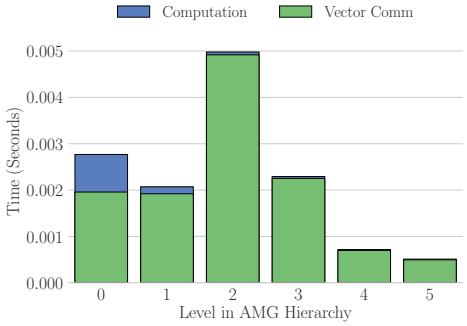
Figure 7.10: Symmetry `strength` per level for Example 7.1.

Aggregation: `aggregation` Aggregation clusters nodes in the sparse matrix graph of S_ℓ to form a coarse grid. Parallel aggregation in this chapter consists of forming a distance-two maximal independent set (MIS-2) before forming one aggregate from each selected node. This approach yields aggregates of similar quality to the standard sequential algorithms [87]. The aggregates are created by grouping unselected nodes with the strongly connected node in the MIS-2 set. If no node exists, the node is added to the aggregate of its strongest connection.

Aggregation consists only of vector communication. The MIS-2 algorithm iteratively selects nodes based on an initial random value, similar to PMIS. Each iteration consists of sending updated selections five times. The cost of each iteration of MIS-2, partitioned into local computation and inter-process communication, is displayed in Figure 7.11a. After a



(a) Per-iteration cost of MIS-2 for finest level.



(b) Full `aggregate` cost, per-level.

Figure 7.11: Cost of `aggregate` for Example 7.1

maximal independent set is selected, the remaining aggregation is mostly local computation. After the first pass, in which nodes are added to the aggregate of the neighboring node selected in MIS-2, updated aggregation information is communicated. After this communi-

cation, the second pass of aggregating with strongest neighbors is fully local. An example of the cost of aggregating is given in Figure 7.11b.

Tentative Interpolation: `tentative` Tentative interpolation, T, is formed by injecting candidate vectors in to the sparsity pattern of each aggregate. Therefore, the number of columns in the interpolation operator is equal to the product of the number of aggregates and the number of candidate vectors; a single candidate vector is assumed for this work, but the discussion extends to multiple candidate vectors as well.

Tentative interpolation requires only a single instance of vector communication, required to find the norm of non-zero entries in each column. The cost of forming a tentative interpolation for each level is displayed in Figure 7.12. After the tentative interpolation operator is created, a communication pattern must be determined for the matrix, resulting in vector communication. However, because the matrix is relatively sparse local computation dominates the cost.

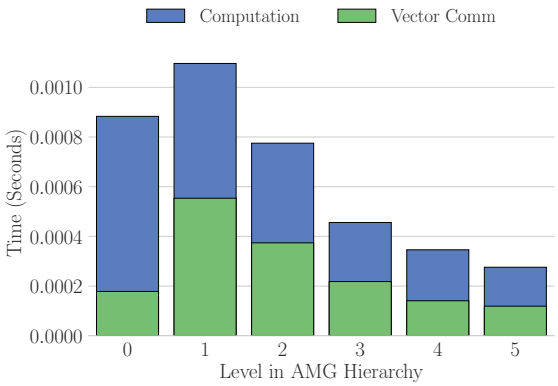


Figure 7.12: Cost of `tentative` for Example 7.1.

Improving Interpolation: `smooth` The accuracy of tentative interpolation is improved through iterations of Jacobi smoothing:

$$P \leftarrow P - \left(D^{-1} \cdot S \cdot P\right), \tag{7.10}$$

where D^{-1} is the inverse diagonal of S and where the initial value of P is equal to the tentative interpolation operator T. The formation of $DS \leftarrow D^{-1} \cdot S$ is fully local. However, the product $DS \cdot T$ is a standard row-wise SpGEMM, requiring communication of the matrix T. Figure 7.13 profiles the cost of a single sweep of Jacobi smoothing on each level, where point-to-point communication costs dominate.

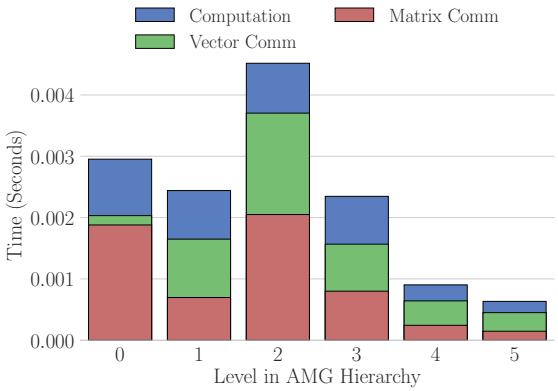


Figure 7.13: Cost of `prolongate` for Example 7.1.

7.2.3 AMG Solve

The AMG solve phase (Algorithm 7.3) is the same for each method and this work focuses on the standard v-cycle.

Algorithm 7.3: AMG Solve: `solve`

Input: A_ℓ // sparse system operator
P_ℓ // interpolation operator
x_ℓ // solution vector
b_ℓ // right-hand-side vector

Output: x_ℓ // updated solution vector

if $\ell = N$
  `solve` $A_\ell x_\ell = b_\ell$
else
  `relax` $A_\ell x_\ell = b_\ell$ {pre-relaxation}
  $r_\ell \leftarrow b_\ell - A_\ell \cdot x_\ell$ {calculate residual}
  $r_{\ell+1} \leftarrow P_\ell^T \cdot r_\ell$ {restrict residual}
  $e_{\ell+1} \leftarrow$ `solve`$(A_{\ell+1}, P_{\ell+1}, 0, r_{\ell+1})$ {coarse-grid solve}
  $e_\ell \leftarrow P_\ell \cdot e_{\ell+1}$ {interpolate error }
  $x_\ell \leftarrow x_\ell + e_\ell$ {update solution}
  `relax` $A_\ell x_\ell = b_\ell$ {post-relaxation}

The solve phase for a Ruge-Stüben hierarchy is profiled in Figure 7.14. The results show that relaxation and residual construction dominate the cost; this is consistent across methods.

Each step of the solve phase is represented as a sparse matrix-vector product. As a result, per level costs are dominated by (vector) communication, particularly at deeper levels in the hierarchy.

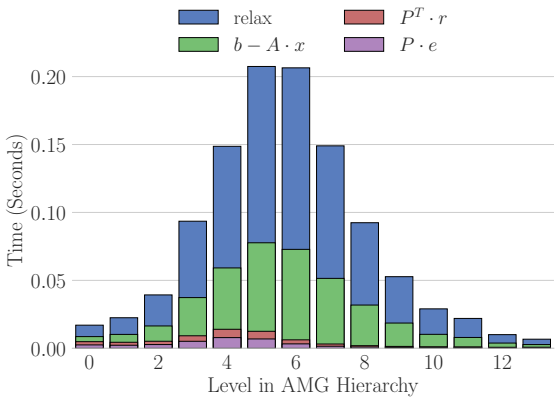


Figure 7.14: AMG V-cycle costs for Example 7.1.

7.3 NODE-AWARE COMMUNICATION

The cost associated with standard point-to-point communication throughout AMG can be reduced through the use of node-awareness, particularly when a large number of messages are communicated, as is the case on coarse levels of AMG. This concept is introduced in [60] for the SpMV and is extended here to all components of the AMG setup and solve phases. In particular, a new two-step communication process is introduced for the coarsest levels of the AMG hierarchy.

Standard communication requires sending data directly between processes, regardless of their locations within the parallel topology. For example, Figure 5.3 displays standard communication in which a number of processes on node n and m send data directly to a process q. Furthermore, Figure 5.4 shows the standard process of communicating data from some process p on node n to all processes on node m. In both cases, multiple messages are communicated between the two nodes. Furthermore, in the latter example, duplicate data is sent to multiple processes on node m, indicating both the number and size of messages communicated between nodes n and m is larger than ideal.

Typical three-step node-aware parallel (NAP) communication, described in Section 5.4, reduces the number and size of messages injected into the network while increasing the amount of less costly on-node communication. NAP communication gathers all data to be sent to node m on some process local to the node n on which it originates. This data is then sent as a single message through the network, before being distributed to the necessary processes on node m, as exemplified in Figure 5.5.

Alternatively, this chapter introduces a method to allow *all* processes to remain active in inter-node communication. This new type of node-aware communication, displayed in Figure 7.15 consists of gathering all data on process to be sent to a node m, and sending

this directly to the corresponding process. This is followed by redistribution of values on the

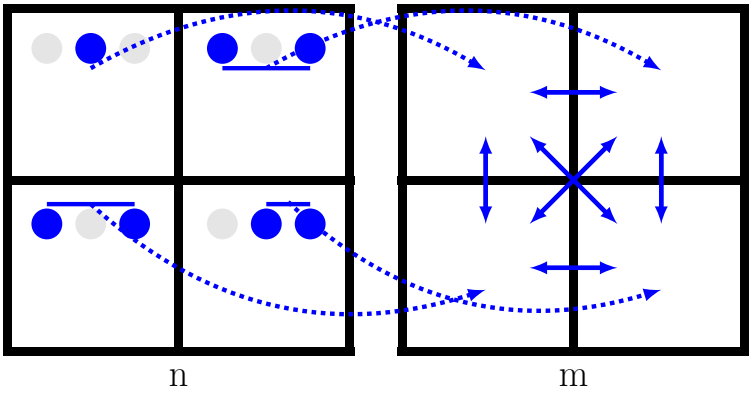


Figure 7.15: An alternative, two-step node-aware communication: (1) gather all data to be sent to node m on process and (2) send directly to the corresponding process on node m.

receiving node. This alternate node-aware method reduces the number and size of data by eliminating the duplication displayed in Figure 5.4, but the multiple messages communicated between nodes in Figure 5.3 remains. The two forms of node-aware communication, two-step and three-step, are used together to dramatically reduce communication in AMG.

7.4 RESULTS

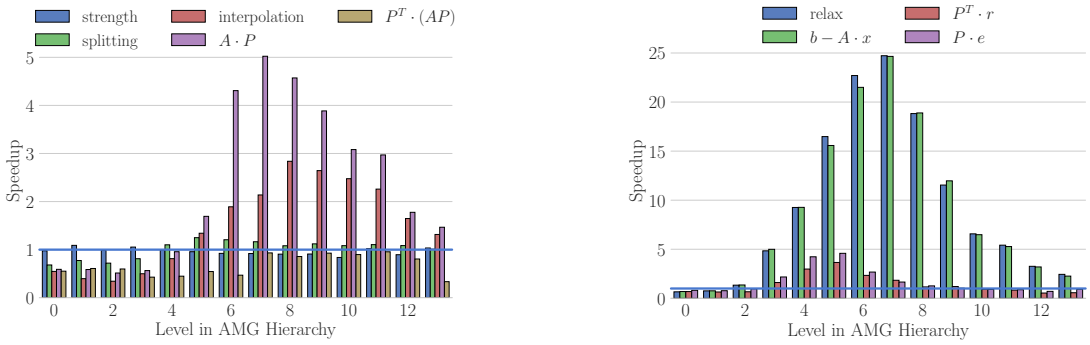
Node-aware communication can be used when communicating any vector or sparse matrix throughout AMG. Node-aware vector communication is implemented as previously described in Section 5, while matrices are communicated as outlined in Section 6. This section presents numerical results for using node-aware communication throughout algebraic multigrid. Both two and three-dimensional systems are analyzed, as two dimensional systems use classical Ruge-Stüben parameters such as CLJP and modified classical interpolation, while three-dimensional systems are improved with aggressive coarsening and extended interpolation. All tests are performed with RAPtor [34] on Blue Waters, a supercomputer at the National Center for Supercomputing Applications [56, 57].

An array of three-dimensional problems are analyzed with both Ruge-Stüben and smoothed aggregation AMG. For all 3D problems, HMIS coarsening is used in combination with extended+i interpolation for Ruge-Stüben hierarchies. These systems include a standard 27-point Laplacian, $\triangle u = 0$, for which a strength tolerance of 0.25 is used. Furthermore, an array of MFEM systems are analyzed, including the Laplace problem $-\triangle u = 1$ on the mobius strip mesh, the grad-div problem, $-\bigtriangledown(\alpha \bigtriangledown \cdot (F)) + \beta F = f$, on the star-surf mesh, and the discontinuous Galerkin discretization of the Laplace problem on the escher mesh.

These systems come from MFEM examples 1, 4, and 14, and use strength thresholds of 0.5, 0.0, and 0.25, respectively.

Node-aware communication can be used in all methods on each level of the AMG hierarchy, throughout both the setup and solve phases. However, speedup is only obtained when sufficient amounts of data are communicated. For instance, the communication pattern of a stenciled matrix is near optimal, as processes only talk with neighbors. The three-step nature of node-aware communication adds synchronization costs, and will likely slow down any near-ideal standard communication. However, communication patterns expand on coarse levels, allowing for reduction in cost through the use of node-aware communication.

Figure 7.16 shows the speedup acquired with node-aware communication in each methods throughout the Ruge-Stüben hierarchy for anisotropic diffusion. There is no change to the

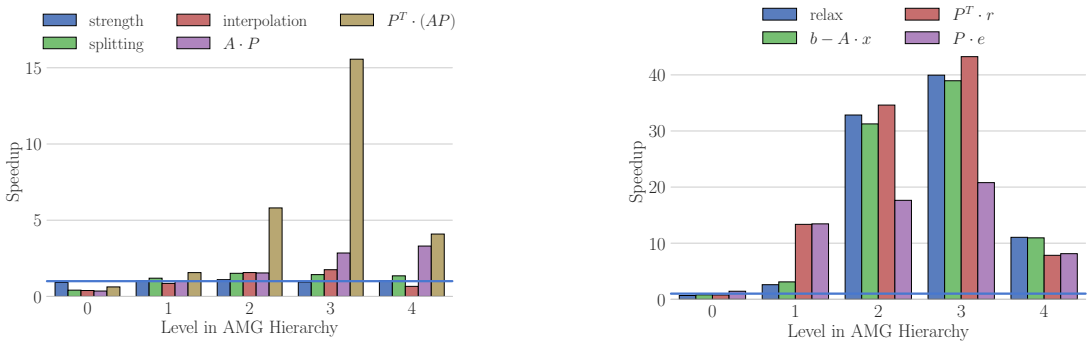


(a) `rs_setup` (b) `solve`

Figure 7.16: Speedup from node-aware communication throughout the Ruge-Stüben hierarchy for Example 7.1

cost of strength-of-connection, as this is a fully local method. However, all other methods yield slowdown on the finer levels, before achieving speedup on coarse levels, particularly near the middle of the hierarchy, where substantial costs occur. As finer levels yield little to no improvements, node-aware communication is only implemented from level 5 onward for the slowly coarsening classical AMG. Therefore, all further standard Ruge-Stüben tests use standard communication until level 5, after which node-aware communication occurs.

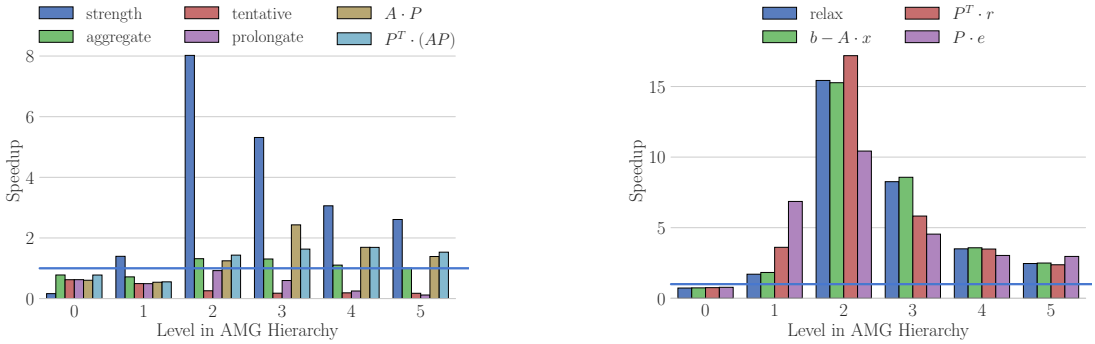
Similarly, speedups obtained through node-aware communication in each method of an aggressively coarsened Ruge-Stüben hierarchy for the 27-point Laplacian are displayed in Figure 7.17. As previously shown, node-aware communication yields little to no improvement over standard communication on fine levels. As aggressively coarsened hierarchies increase in density more rapidly than the standard approach, communication patterns expand at a quicker rate. Therefore, all remaining aggressive coarsening results communicate with node-awareness only from level 3 and coarser.



(a) `rs_setup` (b) `solve`

Figure 7.17: Speedup from node-aware communication throughout the aggressively coarsening Ruge-Stüben hierarchy for Example 7.2

Lastly, node-aware communication yields similar performance throughout smoothed aggregation hierarchies. Figure 7.18 shows speedups over standard communication for the methods of a smoothed aggregation hierarchy for anisotropic diffusion. Similar to aggres-



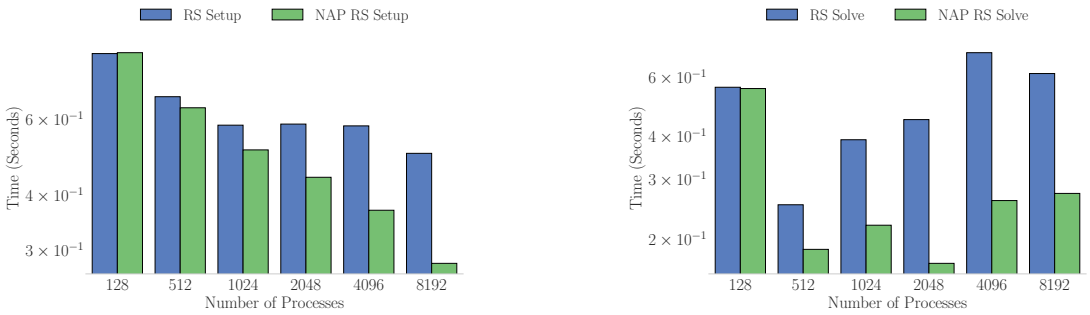
(a) `sa_setup` (b) `solve`

Figure 7.18: Speedup from node-aware communication throughout the smoothed aggregation hierarchy for Example 7.1

sively coarsened hierarchies, hierarchies coarsen at a rapid rate. All subsequent smoothed aggregation tests use node-aware communication starting on level 3.

Node-aware communication yields performance improvements, particularly near strong scaling limits, and improves scalability of both the setup and solve phases. Figure 7.19 shows the cost of setting up and solving a classical AMG hierarchy for the anisotropic problem.

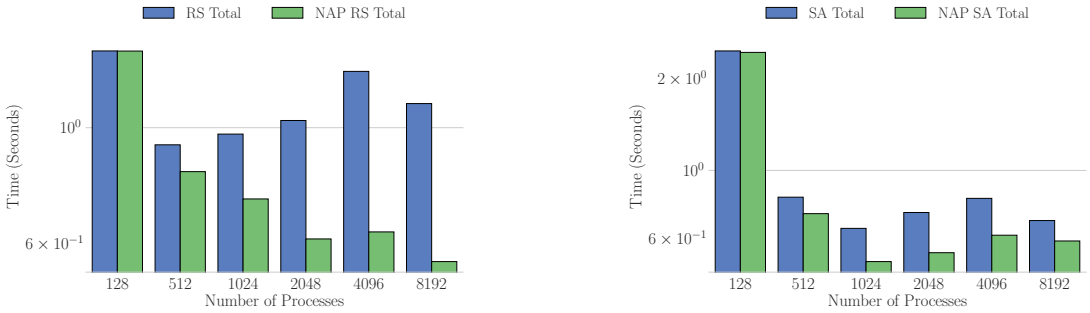
While the scalability of each phase is improved independently, the setup and solve phases are typically run on equivalent process counts. As the setup phase scales further than the solve, it is important to note that the solve phase yields little increase in cost after the strong scaling limit when node-aware communication is used. As a result, the performance



(a) `rs_setup` (b) `solve`

Figure 7.19: Time required to setup and solve the Ruge-Stüben hierarchy for Example 7.1

and scalability of the full AMG solver is improved. Figure 7.20 shows improvements to both the strongly scaled Ruge-Stüben and smoothed aggregation hierarchies for anisotropic diffusion.



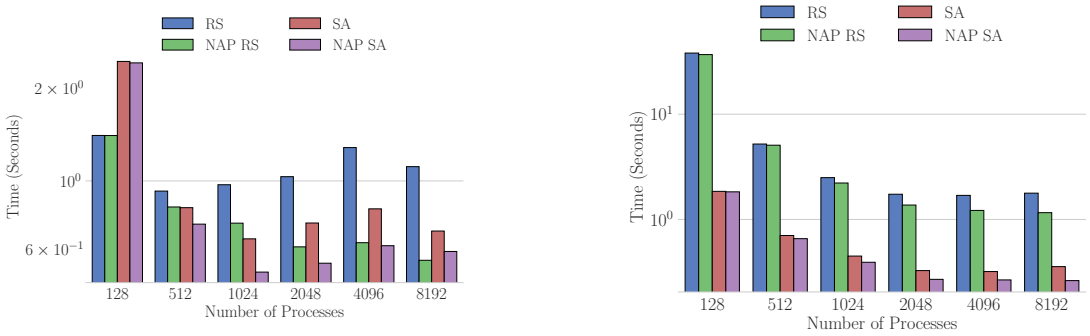
(a) Ruge-Stüben AMG (b) Smoothed aggregation solver

Figure 7.20: Time required to for full AMG solver for Example 7.1

Figure 7.21 displays a strong scaling comparison of the various solvers for both anisotropic diffusion and the 27-point Laplacian, showing improvements to both solvers with the use of node-aware communication. Furthermore, node-aware communication yields similar improvements to weak scalability. Figure 7.22 displays performance and scaling improvements for weakly scaled anisotropic diffusion.

Finally, three-dimensional systems created with MFEM were also analyzed with node-aware communication throughout both AMG solvers. Improvements to performance and scalability were seen in all cases. This is displayed in Figure 7.23.

Further improvements to node-aware AMG are possible, as many operations yield little to no speedup, particularly in the setup phase. While the majority of operations throughout the setup phase are dependent on the sparsity pattern of A, the transpose multiplication step $P^T \cdot AP$ is dependent on the sparsity pattern of P. Therefore, node-aware communication



(a) Example 7.1

(b) Example 7.2

Figure 7.21: A comparison of Ruge-Stüben (RS) and Smoothed Aggregation (SA) AMG, with and without node-aware communication, for Example problems.

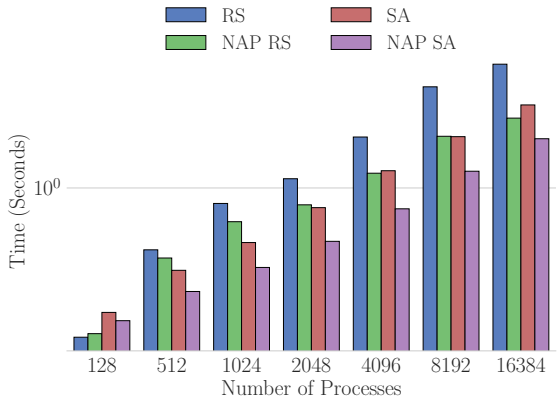
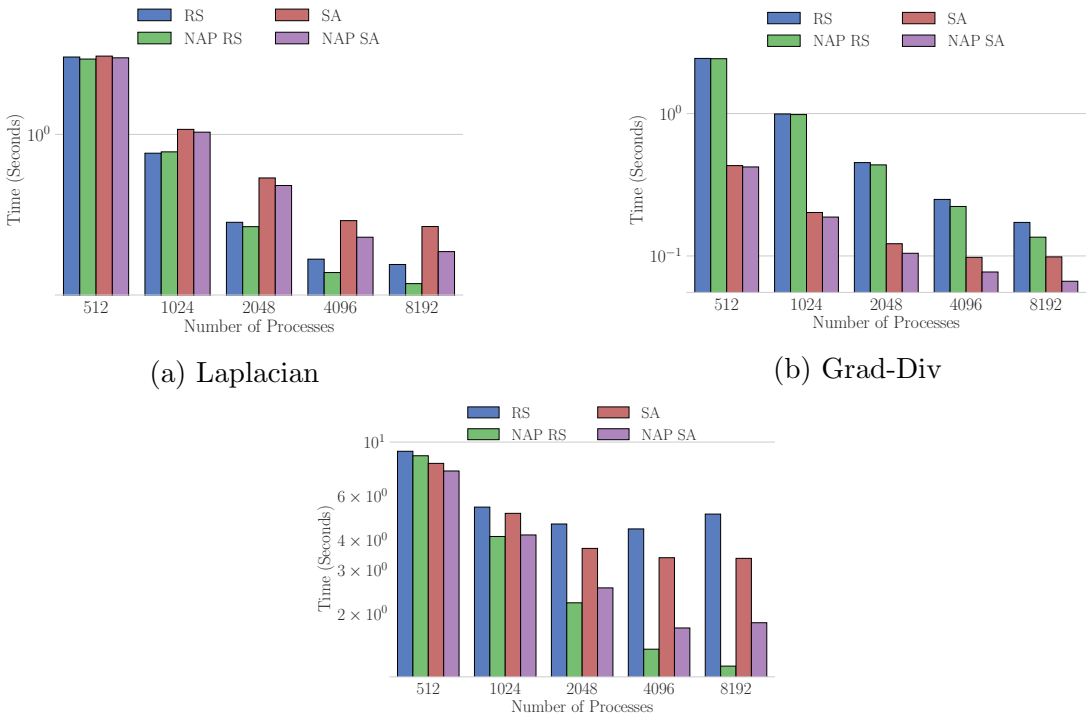


Figure 7.22: A comparison of Ruge-Stüben and Smoothed Aggregation AMG for a weak scaling study of the problem from Example 7.1, with 10 000 degrees-of-freedom per core.

will have larger improvements for transpose multiplication with denser P matrices. This motivates increasing the density of P to increase the accuracy of projecting data between methods, such as using multiple sweeps of Jacobi smoothing during the smoothed aggregation AMG setup. Figure 7.24 shows the speedups obtained with node-aware communication during $P^T \cdot AP$ on each level for both the standard hierarchy and when using multiple smoothing sweeps. There is significant speedup for the denser P resulting from the latter.

Furthermore, there is potential to improve operations on the finer levels of both the setup and solve phase through the use of the alternative node-aware communication described in Figure 7.15. Figure 7.25 displays the cost of matrix multiplication $A \cdot P$, $P^T \cdot AP$, and $A \cdot x$ with standard communication, typical node-aware communication, and the alternative two-step node-aware communication. Two-step node-aware communication yields less potential for slowdown on the fine levels, and improves over other methods for a subset of the levels.

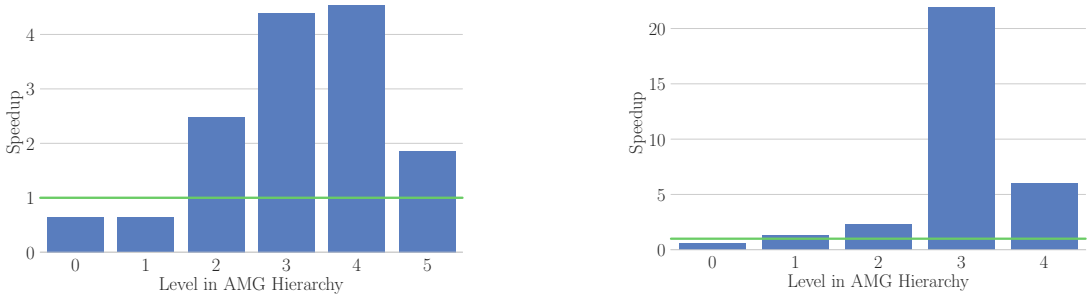


(a) Laplacian

(b) Grad-Div

(c) Discontinuous Galerkin

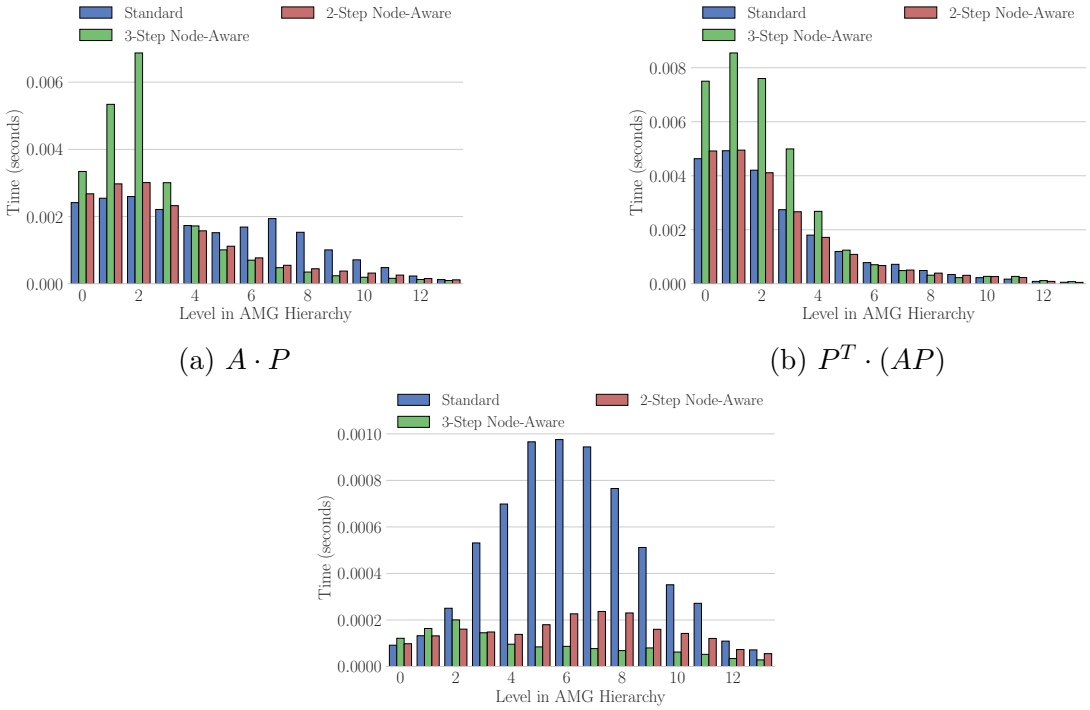
Figure 7.23: A comparison of Ruge-Stüben (RS) and Smoothed Aggregation (SA) AMG for MFEM systems.



(a) One Sweep

(b) Two Sweeps

Figure 7.24: Speedup in $P^T \cdot (AP)$ in `sa_setup` for Example 7.1 when using a single sweep of Jacobi prolongation, as typical, or increasing to two smoothing sweeps.



(a) $A \cdot P$

(b) $P^T \cdot (AP)$

(c) $A \cdot x$

Figure 7.25: Cost of standard, three-step node-aware, and alternative two-step node aware communication throughout operations from Example 7.1

7.5 CONCLUSIONS

This chapter has introduced a method of using node-awareness to reduce the amount of inter-node communication at a trade-off of less costly intra-node communication, and applied this communication method to the various parts of algebraic multigrid. Node-aware

communication yields improvements in the performance and scalability of both the setup and solve phases for both a model rotated anisotropic diffusion problem as well as all tested matrices from the SuiteSparse collection. Future work will be done to further improve the performance of node-aware communication inside the setup phase by more efficiently applying it to remaining methods such as CLJP.

CHAPTER 8: CONCLUSIONS

8.1 SUMMARY

In summary, this dissertation presents multiple methods for reducing costs associated with communication in sparse matrix operations, yielding performance and scalability improvements of linear solvers such as algebraic multigrid. Performance models for point-to-point communication are presented in Chapter 3. Contributions of this chapter include the following.

- Accuracy of the max-rate model is improved through the addition of node-awareness.
- An upper-bound quadratic queue search term is added to traditional models, improving the analysis of irregular point-to-point communication during which many messages are communicated.
- An approximate network contention penalty for communication of a large number of bytes between non-neighboring processes.

Algorithmic changes to algebraic multigrid are presented in Chapter 4. This chapter introduces a new method, similar to the method of non-Galerkin coarse grids, which adds sparsity into matrices to reduce communication requirements at the cost of convergence. Specifically, the contributions of this chapter are as follows.

- Relatively small non-zeros are systematically removed from coarse matrices after formation of the algebraic multigrid hierarchy.
- Removed non-zeros are lumped to the diagonal rather than strongly connected neighbors, allowing for removal of a larger number of non-zeros.
- The trade-off between per-iteration communication costs and convergence is controlled through re-introduction of previously removed entries through and adaptive solve phase.

Finally, adjustments to the parallel implementation of MPI communication are presented in Chapters 5, 6, and 7. Node-aware communication is introduced, through which standard messages are aggregated, to reduce the amount of costly inter-node communication. The main contributions of these chapters are described as follows.

- Agglomeration of messages on-node preceding inter-node communication yields an increase in intra-node communication while reducing both the number and size of more costly inter-node messages.
- Three-step node-aware communication yields large improvements to both performance and scalability of sparse matrix-vector multiplication.
- Sparse matrix-matrix multiplication performance is improved, particularly on matrices with relatively large communication requirements and near strong scaling limits.
- Three-step node-aware communication throughout sparse matrix operations yields large improvements to both the setup and solve phase of algebraic multigrid, extending strong scalability of many systems for both Ruge-Stüben and smoothed aggregation solvers.
- An alternative two-step node-aware communication indicates potential to improve performance of matrix communication on finer levels of the algebraic multigrid hierarchy.
- Node-aware communication yields larger speedups for increasingly dense matrices, indicating potential speedup with dense interpolation matrices. This motivates the use of increasingly accurate projection between levels.

An open-source parallel AMG codebase, RAPtor, is outlined in Appendix A. The main contributions of RAPtor are outlined as follows.

- A parallel algebraic multigrid codebase similar to PyAMG in style, which allows a user to easily switch out main methods based on user preference.
- The communication method can be switched throughout AMG, allowing for use of two and three-step node-aware communication in any method.
- Both Ruge-Stüben and smoothed aggregation solvers are implemented, using identical communication, matrix operations, and solve phases, and allowing for direct comparison of the solvers.

8.2 CONCLUDING REMARKS AND FUTURE DIRECTIONS

Alterations to both the algorithm and parallel implementation of sparse matrix operations are presented throughout this dissertation, with a focus on reducing costs associated with inter-process communication. Parallel communication costs are greatly reduced through the

removal of insignificant entries, and the trade-off between algebraic multigrid per-iteration cost and convergence rate is controlled through an adaptive solve phase. Node-aware communication improves performance and strong scalability, particularly when large amounts of inter-process communication is required. Finally, improved performance models for inter-process communication yield the potential to analyze the source of dominant costs to target for removal.

There are many future directions for communication reduction in sparse operations. Node-awareness can be added to sparsification methods, as non-zeros corresponding to inter-node communication yield larger cost than those associated with on-node values, motivating targeting larger non-zeros in the former position. Furthermore, improved performance models can be used for a more accurate prediction of sparsification tolerances.

Similarly, performance models yield potential to improve node-aware communication. Sparse matrix communication, and the setup phase of AMG, can be improved through the use of the two-step communication, particularly for sparser matrices with small communication requirements. Furthermore, node-aware communication motivates the use of increasingly accurate and dense transfer operators, as speedup over standard communication increases with density. Finally, this motivates the use of sparsification and node-aware communication in combination, as denser interpolation operators yield increasingly dense coarse-grid operators.

Finally, both sparsification and node-aware communication can be added to preconditioners other than algebraic multigrid. Sparsification can be applied to iterative preconditioners such as ILU in an effort to reduce per-iteration cost. Furthermore, node-aware communication has potential to greatly improve the cost of multi-vector communication, motivating implementation in enlarged krylov subspace methods and solution of systems with multiple right-hand-sides.

APPENDIX A: RAPTOR

RAPtor is an open-source parallel algebraic multigrid solver, written in $C++$ and MPI. The codebase was created in an effort to reduce communication costs throughout the algebraic multigrid method, allowing for the use of node-aware communication throughout the setup and solve phases. Secondly, the software implementation is similar to Pyamg [41] in style, providing a straightforward extension to parallel AMG.

There are a large number of parallel algebraic multigrid codebases in current production. Two commonly used solvers include Hypre [38, 59], a Ruge-Stüben solver created by Lawrence Livermore National Laboratory, and ML [112], a smoothed aggregation solver created by Sandia National Laboratory. Other distributed implementations of AMG include GAMG [113], SAMG [114], AGMG [115], and ViennaCL [116]. Similarly, there are multiple GPU implementations of AMG, including CUSP [117] and AMGX [118].

RAPtor contains a `ParMultilevel` object `ml`, which can be created as either a `ParRugeStubenSolver`, yielding a classical Ruge-Stüben hierarhcy as described in Figure A.1, or a `ParSmoothedAggregationSolver` described in Figure A.2, which creates a smoothed aggregation hierarchy.

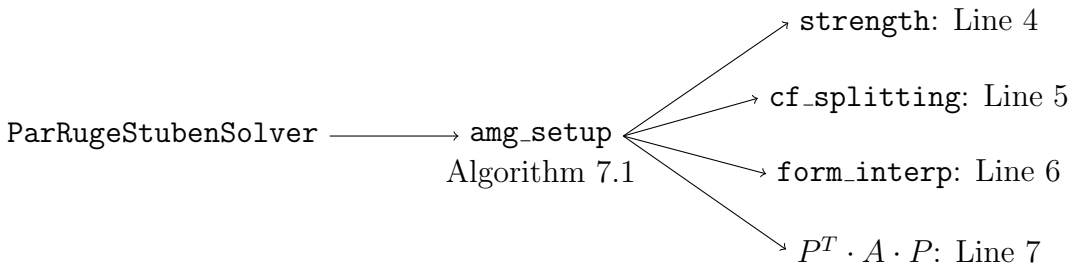


Figure A.1: Methods in `ParRugeStubenSolver`

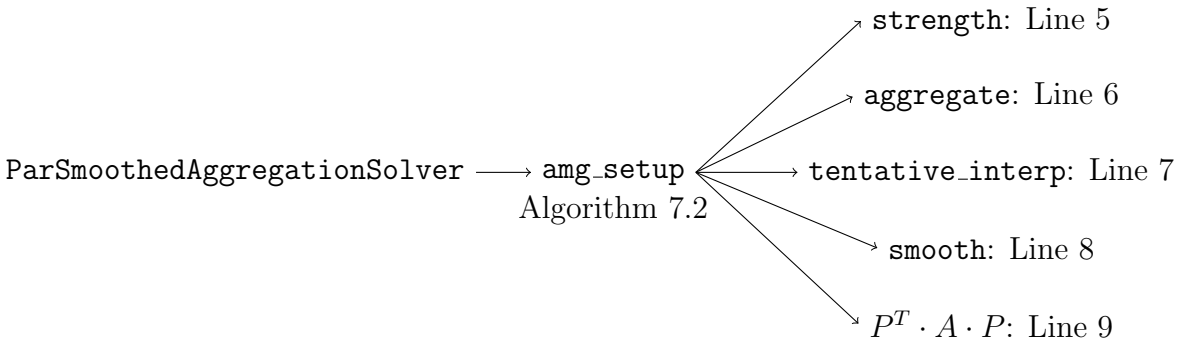


Figure A.2: Methods in `ParSmoothedAggregationSolver`

The ParMultilevel object has only a single solve phase, as outlined in Figure A.3. There-

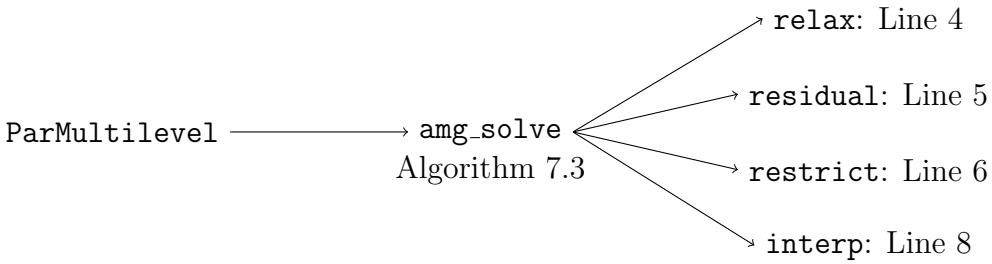
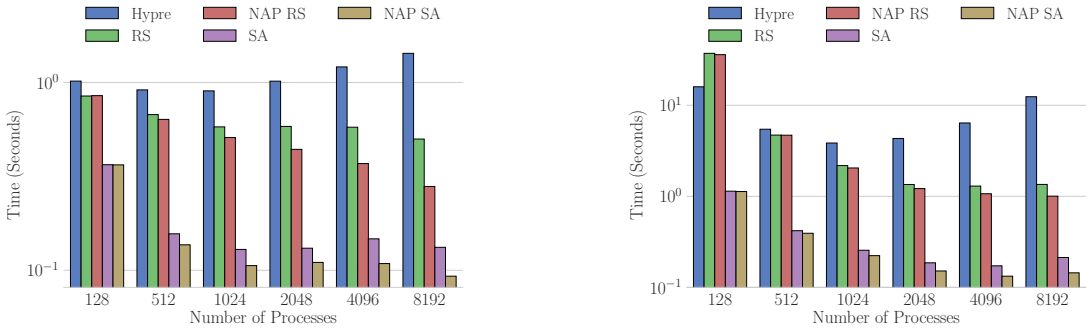


Figure A.3: Methods in `ParMultilevel`

fore, RAPtor allows for straightforward comparisons of Ruge-Stüben and smoothed aggregation AMG, in which communication, matrix operations, and full AMG solve phase are equivalent.

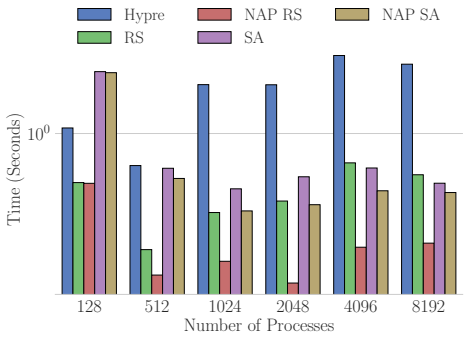
Figures A.4 and A.5 compare the setup and solve times, respectively, of Hypre to the various methods of RAPtor, including standard and node-aware Ruge-Stüben and smoothed aggregation hierarchies.



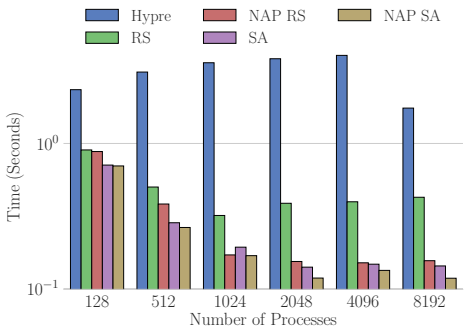
(a) Anisotropic diffusion from Example 7.1. (b) 27-point Laplacian from Example 7.2.

Figure A.4: Setup times for RAPtor, in comparison to Hypre, strongly scaled, 10 240 000 non-zeros.

Furthermore, the full AMG times for strong scaling studies of the various solvers in RAPtor, in comparsion to Hypre, are displayed in Figure A.6, while the equivalent costs of a weak scaling study are shown in Figure A.7.

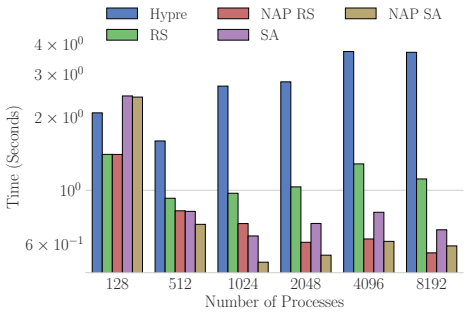


(a) Anisotropic diffusion from Example 7.1.

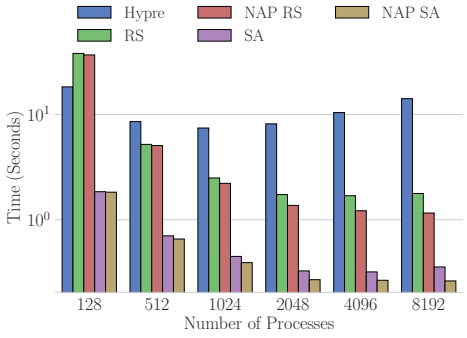


(b) 27-point Laplacian from Example 7.2.

Figure A.5: Solve times for RAPtor, in comparison to Hypre, strongly scaled, 10 240 000 non-zeros.



(a) Anisotropic diffusion from Example 7.1.



(b) 27-point Laplacian from Example 7.2.

Figure A.6: Full AMG ties for RAPtor, in comparison to Hypre, strongly scaled, 10 240 000 non-zeros.

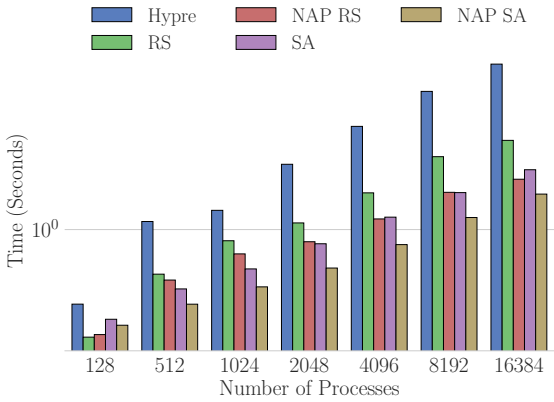


Figure A.7: Full AMG ties for RAPtor, in comparison to Hypre, weakly scaled, from Example 7.1 with 10 000 degrees-of-freedom per core.

REFERENCES

[1] L. Grigori, S. Moufawad, and F. Nataf, "Enlarged krylov subspace conjugate gradient methods for reducing communication," *SIAM Journal on Matrix Analysis and Applications*, vol. 37, no. 2, pp. 744–773, 2016. [Online]. Available: https://doi.org/10.1137/140989492

[2] J. Demmel, M. Hoemmen, M. Mohiyuddin, K. A. Yelick, Mark, Hoemmen, and . Mohiyuddin, "Avoiding communication in computing krylov subspaces," 2007.

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

[4] P. Ghysels and W. Vanroose, "Hiding global synchronization latency in the preconditioned conjugate gradient algorithm," *Parallel Computing*, vol. 40, no. 7, pp. 224 – 238, 2014, 7th Workshop on Parallel Matrix Algorithms and Applications. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0167819113000719

[5] D. A. Spielman and S.-H. Teng, "Nearly-linear time algorithms for graph partitioning, graph sparsification, and solving linear systems," in *Proceedings of the Thirty-sixth Annual ACM Symposium on Theory of Computing*, ser. STOC '04. New York, NY, USA: ACM, 2004. [Online]. Available: http://doi.acm.org/10.1145/1007352.1007372 pp. 81–90.

[6] A. A. Benczúr and D. R. Karger, "Approximating s-t minimum cuts in $\tilde{O}$(n2) time," in *Proceedings of the Twenty-eighth Annual ACM Symposium on Theory of Computing*, ser. STOC '96. New York, NY, USA: ACM, 1996. [Online]. Available: http://doi.acm.org/10.1145/237814.237827 pp. 47–55.

[7] 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

[8] Ümit 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

[9] V. Karakasis, G. Goumas, and N. Koziris, "Perfomance models for blocked sparse matrix-vector multiplication kernels," in *2009 International Conference on Parallel Processing*, Sept 2009, pp. 356–364.

[10] C. S. Murthy, M. K. N. Balasubramanya, and S. Aluru, *New Parallel Algorithms for Direct Solution of Linear Equations*, 1st ed. New York, NY, USA: John Wiley & Sons, Inc., 2000.

[11] T. A. Davis and Y. Hu, "The university of florida sparse matrix collection," *ACM Transactions on Mathematical Software*, vol. 32, pp. 1:1 – 1:25, 2011. [Online]. Available: http://www.cise.ufl.edu/research/sparse/matrices

[12] D. M. Alber and L. N. Olson, "Parallel coarse-grid selection," *Numerical Linear Algebra with Applications*, vol. 14, no. 8, pp. 611–643, 2007.

[13] J. W. Ruge and K. Stüben, "Algebraic multigrid (AMG)," in *Multigrid Methods*, ser. Frontiers Appl. Math., S. F. McCormick, Ed. Philadelphia: SIAM, 1987, pp. 73–130.

[14] L. N. Olson, J. Schroder, and R. S. Tuminaro, "A new perspective on strength measures in algebraic multigrid," *Numerical Linear Algebra with Applications*, vol. 17, no. 4, pp. 713–733, 2010.

[15] R. D. Falgout and J. B. Schroder, "Non-Galerkin coarse grids for algebraic multigrid," *SIAM Journal on Scientific Computing*, vol. 36, no. 3, pp. C309–C334, 2014. [Online]. Available: http://dx.doi.org/10.1137/130931539

[16] H. Gahvari, A. H. Baker, M. Schulz, U. M. Yang, K. E. Jordan, and W. Gropp, "Modeling the performance of an algebraic multigrid cycle on HPC platforms," in *Proceedings of the International Conference on Supercomputing*, ser. ICS '11. New York, NY, USA: ACM, 2011. [Online]. Available: http://doi.acm.org/10.1145/1995896.1995924 pp. 172–181.

[17] 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.

[18] A. Bienz, W. D. Gropp, and L. N. Olson, "Improving performance models for irregular point-to-point communication," in *EuroMPI 2018*, Barcelona, Spain, 2018.

[19] 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

[20] D. E. Culler, R. M. Karp, D. Patterson, A. Sahay, E. E. Santos, K. E. Schauser, R. Subramonian, and T. von Eicken, "Logp: A practical model of parallel computation," *Commun. ACM*, vol. 39, no. 11, pp. 78–85, Nov. 1996. [Online]. Available: http://doi.acm.org/10.1145/240455.240477

[21] P. B. Gibbons, "A more practical pram model," in *Proceedings of the First Annual ACM Symposium on Parallel Algorithms and Architectures*, ser. SPAA '89. New York, NY, USA: ACM, 1989. [Online]. Available: http://doi.acm.org/10.1145/72935.72953 pp. 158–168.

[22] A. Alexandrov, M. F. Ionescu, K. E. Schauser, and C. Scheiman, "Loggp: Incorporating long messages into the logp model – one step closer towards a realistic model for parallel computation," in *Proceedings of the Seventh Annual ACM Symposium on Parallel Algorithms and Architectures*, ser. SPAA '95. New York, NY, USA: ACM, 1995. [Online]. Available: http://doi.acm.org/10.1145/215399.215427 pp. 95–105.

[23] M. I. Frank, A. Agarwal, and M. K. Vernon, "LoPC: Modeling contention in parallel algorithms," in *Proceedings of the Sixth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPOPP '97. New York, NY, USA: ACM, 1997. [Online]. Available: http://doi.acm.org/10.1145/263764.263803 pp. 276–287.

[24] C. A. Moritz and M. I. Frank, "LoGPC: Modeling network contention in message-passing programs," in *Proceedings of the 1998 ACM SIGMETRICS Joint International Conference on Measurement and Modeling of Computer Systems*, ser. SIGMETRICS '98/PERFORMANCE '98. New York, NY, USA: ACM, 1998. [Online]. Available: http://doi.acm.org/10.1145/277851.277933 pp. 254–263.

[25] L. A. Steffenel, "Modeling network contention effects on all-to-all operations," in *2006 IEEE International Conference on Cluster Computing*. Barcelona, Spain: IEEE, Sept 2006, pp. 1–10.

[26] N. Jain, A. Bhatele, M. P. Robson, T. Gamblin, and L. V. Kale, "Predicting application performance using supervised learning on communication features," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, ser. SC '13. New York, NY, USA: ACM, 2013. [Online]. Available: http://doi.acm.org/10.1145/2503210.2503263 pp. 95:1–95:12.

[27] T. Agarwal, A. Sharma, and L. V. Kalé, "Topology-aware task mapping for reducing communication contention on large parallel machines," in *Proceedings of the 20th International Conference on Parallel and Distributed Processing*, ser. IPDPS'06. Washington, DC, USA: IEEE Computer Society, 2006. [Online]. Available: http://dl.acm.org/citation.cfm?id=1898953.1899075 pp. 145–145.

[28] N. Saboo, A. K. Singla, J. M. Unger, and L. V. Kale, "Emulating petaflops machines and blue gene," in *Proceedings 15th International Parallel and Distributed Processing Symposium. IPDPS 2001.* San Francisco, CA, USA: IEEE, April 2001, pp. 2084–2091.

[29] T. Schneider, T. Hoefler, and A. Lumsdaine, "ORCS: An Oblivious Routing Congestion Simulator," Indiana University, Tech. Rep. 675, Feb. 2009.

[30] T. Hoefler, T. Schneider, and A. Lumsdaine, "LogGOPSim - Simulating Large-Scale Applications in the LogGOPS Model," in *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*. Chicago, Illinois: ACM, Jun. 2010, pp. 597–604.

[31] J. Cownie and W. Gropp, "A standard interface for debugger access to message queue information in mpi," in *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, J. Dongarra, E. Luque, and T. Margalef, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 1999, pp. 51–58.

[32] G. Dózsa, S. Kumar, P. Balaji, D. Buntinas, D. Goodell, W. Gropp, J. Ratterman, and R. Thakur, "Enabling concurrent multithreaded mpi communication on multicore petascale systems," in *Proceedings of the 17th European MPI Users' Group Meeting Conference on Recent Advances in the Message Passing Interface*, ser. EuroMPI'10. Berlin, Heidelberg: Springer-Verlag, 2010. [Online]. Available: http://dl.acm.org/citation.cfm?id=1894122.1894125 pp. 11–20.

[33] M. Flajslik, J. Dinan, and K. D. Underwood, "Mitigating mpi message matching misery," in *High Performance Computing*, J. M. Kunkel, P. Balaji, and J. Dongarra, Eds. Cham: Springer International Publishing, 2016, pp. 281–299.

[34] A. Bienz and L. N. Olson, "RAPtor: parallel algebraic multigrid v0.1," 2017, release 0.1. [Online]. Available: https://github.com/lukeolson/raptor

[35] A. Bienz, R. D. Falgout, W. Gropp, L. N. Olson, and J. B. Schroder, "Reducing parallel communication in algebraic multigrid through sparsification," *SIAM Journal on Scientific Computing*, vol. 38, no. 5, pp. S332–S357, 2016. [Online]. Available: https://doi.org/10.1137/15M1026341

[36] S. F. McCormick and J. W. Ruge, "Multigrid methods for variational problems," *SIAM J. Numer. Anal.*, vol. 19, no. 5, pp. 924–929, 1982.

[37] A. Brandt, S. F. McCormick, and J. W. Ruge, "Algebraic multigrid (AMG) for sparse matrix equations," in *Sparsity and Its Applications*, D. J. Evans, Ed. Cambridge: Cambridge Univ. Press, 1984, pp. 257–284.

[38] "HYPRE: High performance preconditioners," http://www.llnl.gov/CASC/hypre/.

[39] U. M. Yang, "On long-range interpolation operators for aggressive coarsening," *Numerical Linear Algebra with Applications*, vol. 17, no. 2–3, pp. 453–472, 2010. [Online]. Available: http://dx.doi.org/10.1002/nla.689

[40] H. D. Sterck, U. M. Yang, and J. J. Heys, "Reducing complexity in parallel algebraic multigrid preconditioners," *SIAM J. Matrix Anal. Appl.*, vol. 27, no. 4, pp. 1019–1039, Dec. 2005. [Online]. Available: http://dx.doi.org/10.1137/040615729

[41] W. N. Bell, L. N. Olson, and J. B. Schroder, "PyAMG: Algebraic multigrid solvers in Python v3.0," 2015, release 3.0. [Online]. Available: http://www.pyamg.org

[42] A. H. Baker, M. Schulz, and U. M. Yang, "On the performance of an algebraic multigrid solver on multicore clusters," in *Proceedings of the 9th International Conference on High Performance Computing for Computational Science*, ser. VECPAR'10. Berlin, Heidelberg: Springer-Verlag, 2011. [Online]. Available: http://dl.acm.org/citation.cfm?id=1964238.1964252 pp. 102–115.

[43] E. Chow, R. D. Falgout, J. J. Hu, R. S. Tuminaro, and U. M. Yang, "A survey of parallelization techniques for multigrid solvers," in *Frontiers of Parallel Processing for Scientific Computing*. Philadelphia, PA, USA: Society for Industrial and Applied Mathematics, 2005.

[44] H. D. Sterck, R. D. Falgout, J. W. Nolting, and U. M. Yang, "Distance-two interpolation for parallel algebraic multigrid," *Numerical Linear Algebra with Applications*, vol. 15, no. 2-3, pp. 115–139, 2008. [Online]. Available: http://dx.doi.org/10.1002/nla.559

[45] M. Bolten, T. K. Huckle, and C. D. Kravvaritis, "Sparse matrix approximations for multigrid methods," *Linear Algebra and its Applications*, 2015, in press. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0024379515006710

[46] E. Treister and I. Yavneh, "Non-Galerkin multigrid based on sparsified smoothed aggregation," *SIAM Journal on Scientific Computing*, vol. 37, no. 1, pp. A30–A54, 2015. [Online]. Available: http://dx.doi.org/10.1137/140952570

[47] J. Dendy, "Black box multigrid," *Journal of Computational Physics*, vol. 48, no. 3, pp. 366–386, 1982.

[48] J. Dendy, "Black box multigrid for nonsymmetric problems," *Appl. Math. Comput.*, vol. 13, pp. 261–283, 1983.

[49] S. F. Ashby and R. D. Falgout, "A parallel multigrid preconditioned conjugate gradient algorithm for groundwater flow simulations," *Nuclear Science and Engineering*, vol. 124, no. 1, pp. 145–159, September 1996, uCRL-JC-122359.

[50] R. Wienands and I. Yavneh, "Collocation coarse approximation in multigrid," *SIAM Journal on Scientific Computing*, vol. 31, no. 5, pp. 3643–3660, 2009. [Online]. Available: http://dx.doi.org/10.1137/08074461X

[51] E. Treister, R. Zemach, and I. Yavneh, "Algebraic collocation coarse approximation (ACCA) multigrid," in *12th Copper Mountain Conference on Iterative Methods*, 2012.

[52] W. S. Fung, R. Hariharan, N. J. Harvey, and D. Panigrahi, "A general framework for graph sparsification," in *Proceedings of the Forty-third Annual ACM Symposium on Theory of Computing*, ser. STOC '11. New York, NY, USA: ACM, 2011. [Online]. Available: http://doi.acm.org/10.1145/1993636.1993647 pp. 71–80.

[53] D. A. Spielman and N. Srivastava, "Graph sparsification by effective resistances," *SIAM Journal on Computing*, vol. 40, no. 6, pp. 1913–1926, 2011. [Online]. Available: http://dx.doi.org/10.1137/080734029

[54] I. Koutis, A. Levin, and R. Peng, "Faster spectral sparsification and numerical algorithms for sdd matrices," *ACM Trans. Algorithms*, vol. 12, no. 2, pp. 17:1–17:16, Dec. 2015. [Online]. Available: http://doi.acm.org/10.1145/2743021

[55] H. Gahvari, M. Hoemmen, J. Demmel, and K. Yelick, "Benchmarking sparse matrix-vector multiply in five minutes," in *SPEC Benchmark Workshop 2007*, Austin, TX, January 2007.

[56] "Blue Waters," https://bluewaters.ncsa.illinois.edu/.

[57] B. Bode, M. Butler, T. Dunning, T. Hoefler, W. Kramer, W. Gropp, and W. Hwu, "The Blue Waters super-system for super-science," in *Contemporary High Performance Computing: From Petascale Toward Exascale*, 1st ed., ser. CRC Computational Science Series, J. S. Vetter, Ed. Boca Raton: Taylor and Francis, 2013, vol. 1, pp. 339–366. [Online]. Available: http://j.mp/RrBdPZ

[58] P. Luszczek, J. J. Dongarra, D. Koester, R. Rabenseifner, B. Lucas, J. Kepner, J. Mccalpin, D. Bailey, and D. Takahashi, "Introduction to the HPC challenge benchmark suite," Tech. Rep., 2005.

[59] V. E. Henson and U. M. Yang, "BoomerAMG: A parallel algebraic multigrid solver and preconditioner," *Appl. Numer. Math.*, vol. 41, no. 1, pp. 155–177, Apr. 2002. [Online]. Available: http://dx.doi.org/10.1016/S0168-9274(01)00115-5

[60] A. Bienz, W. D. Gropp, and L. N. Olson, "Node aware sparse matrix-vector multiplication," *(submitted)*, 2018. [Online]. Available: https://arxiv.org/abs/1612.08060

[61] S. Williams, L. Oliker, R. Vuduc, J. Shalf, K. Yelick, and J. Demmel, "Optimization of sparse matrix-vector multiplication on emerging multicore platforms," in *Proceedings of the 2007 ACM/IEEE Conference on Supercomputing*, ser. SC '07. New York, NY, USA: ACM, 2007. [Online]. Available: http://doi.acm.org/10.1145/1362622.1362674 pp. 38:1–38:12.

[62] 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.

[63] A. Bienz, W. D. Gropp, and L. Olson, "Topology-aware performance modeling of parallel spmvs," Paris, France, 2016, 17$^{\text{th}}$ SIAM Conference on Parallel Processing for Scientific Computing. [Online]. Available: http://meetings.siam.org/sess/dsp_talk.cfm?p=75934

[64] A. Pinar and C. Aykanat, "Fast optimal load balancing algorithms for 1d partitioning," *J. Parallel Distrib. Comput.*, vol. 64, no. 8, pp. 974–996, Aug. 2004. [Online]. Available: http://dx.doi.org/10.1016/j.jpdc.2004.05.003

[65] B. Hendrickson and T. G. Kolda, "Graph partitioning models for parallel computing," *Parallel Comput.*, vol. 26, no. 12, pp. 1519–1534, Nov 2000. [Online]. Available: http://dx.doi.org/10.1016/S0167-8191(00)00048-X

[66] G. Karypis and V. Kumar, "A parallel algorithm for multilevel graph partitioning and sparse matrix ordering," *J. Parallel Distrib. Comput.*, vol. 48, no. 1, pp. 71–95, 1998. [Online]. Available: http://dx.doi.org/10.1006/jpdc.1997.1403

[67] C. Chevalier and F. Pellegrini, "PT-Scotch: A tool for efficient parallel graph ordering," *Parallel Computing*, vol. 34, no. 68, pp. 318 – 331, 2008, parallel Matrix Algorithms and Applications. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0167819107001342

[68] U. V. Çatalyürek 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, Jul 1999.

[69] E. Jeannot, E. Meneses, G. Mercier, F. Tessier, and G. Zheng, "Communication and topology-aware load balancing in charm++ with treematch," in *2013 IEEE International Conference on Cluster Computing (CLUSTER)*, Sept 2013, pp. 1–8.

[70] T. Agarwal, A. Sharma, and L. V. Kalé, "Topology-aware task mapping for reducing communication contention on large parallel machines," in *Proceedings of the 20th International Conference on Parallel and Distributed Processing*, ser. IPDPS'06. Washington, DC, USA: IEEE Computer Society, 2006. [Online]. Available: http://dl.acm.org/citation.cfm?id=1898953.1899075 pp. 145–145.

[71] S. Sreepathi, E. D'Azevedo, B. Philip, and P. Worley, "Communication characterization and optimization of applications using topology-aware task mapping on large supercomputers," in *Proceedings of the 7th ACM/SPEC on International Conference on Performance Engineering*, ser. ICPE '16. New York, NY, USA: ACM, 2016. [Online]. Available: http://doi.acm.org/10.1145/2851553.2851575 pp. 225–236.

[72] T. Malik, V. Rychkov, and A. Lastovetsky, "Network-aware optimization of communications for parallel matrix multiplication on hierarchical hpc platforms," *Concurr. Comput. : Pract. Exper.*, vol. 28, no. 3, pp. 802–821, Mar. 2016. [Online]. Available: http://dx.doi.org/10.1002/cpe.3609

[73] J. L. Träff, "Implementing the mpi process topology mechanism," in *Proceedings of the 2002 ACM/IEEE Conference on Supercomputing*, ser. SC '02. Los Alamitos, CA, USA: IEEE Computer Society Press, 2002. [Online]. Available: http://dl.acm.org/citation.cfm?id=762761.762767 pp. 1–14.

[74] E. Solomonik, A. Bhatele, and J. Demmel, "Improving communication performance in dense linear algebra via topology aware collectives," in *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '11. New York, NY, USA: ACM, 2011. [Online]. Available: http://doi.acm.org/10.1145/2063384.2063487 pp. 77:1–77:11.

[75] T. Kielmann, R. F. H. Hofman, H. E. Bal, A. Plaat, and R. A. F. Bhoedjang, "Magpie: Mpi's collective communication operations for clustered wide area systems," *SIGPLAN Not.*, vol. 34, no. 8, pp. 131–140, May 1999. [Online]. Available: http://doi.acm.org/10.1145/329366.301116

[76] N. T. Karonis, B. R. de Supinski, I. Foster, W. Gropp, E. Lusk, and J. Bresnahan, "Exploiting hierarchy in parallel computer networks to optimize collective operation performance," in *Proceedings 14th International Parallel and Distributed Processing Symposium. IPDPS 2000*, 2000, pp. 377–384.

[77] P. Sack and W. Gropp, "Faster topology-aware collective algorithms through non-minimal communication," in *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '12. New York, NY, USA: ACM, 2012. [Online]. Available: http://doi.acm.org/10.1145/2145816.2145823 pp. 45–54.

[78] L. Wesolowski, R. Venkataraman, A. Gupta, J. S. Yeom, K. Bisset, Y. Sun, P. Jetley, T. R. Quinn, and L. V. Kale, "Tram: Optimizing fine-grained communication with topological routing and aggregation of messages," in *2014 43rd International Conference on Parallel Processing*, Sept 2014, pp. 211–220.

[79] E. Chow and D. Hysom, "Assessing performance of hybrid mpi/openmp programs on smp clusters," Tech. Rep., 2001.

[80] G. Schubert, G. Hager, H. Fehske, and G. Wellein, "Parallel sparse matrix-vector multiplication as a test case for hybrid mpi+openmp programming," *2011 IEEE International Symposium on Parallel and Distributed Processing Workshops and Phd Forum*, pp. 1751–1758, 2011.

[81] G. Wellein, G. Hager, A. Basermann, and H. Fehske, "Fast sparse matrix-vector multiplication for teraflop/s computers," in *High Performance Computing for Computational Science — VECPAR 2002*, J. M. L. M. Palma, A. A. Sousa, J. Dongarra, and V. Hernández, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, pp. 287–301.

[82] C. Mei, Y. Sun, G. Zheng, E. J. Bohm, L. V. Kale, J. C. Phillips, and C. Harrison, "Enabling and scaling biomolecular simulations of 100 million atoms on petascale machines with a multicore-optimized message-driven runtime," in *2011 International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, Nov 2011, pp. 1–11.

[83] N. Bell and M. Garland, "Efficient sparse matrix-vector multiplication on CUDA," NVIDIA Corporation, NVIDIA Technical Report NVR-2008-004, Dec. 2008.

[84] S. Dalton, L. Olson, and N. Bell, "Optimizing sparse matrix-matrix multiplication for the gpu," *ACM Trans. Math. Softw.*, vol. 41, no. 4, pp. 25:1–25:20, Oct. 2015. [Online]. Available: http://doi.acm.org/10.1145/2699470

[85] D. Guo, W. Gropp, and L. N. Olson, "A hybrid format for better performance of sparse matrix-vector multiplication on a gpu," *The International Journal of High Performance Computing Applications*, vol. 30, no. 1, pp. 103–120, 2016. [Online]. Available: https://doi.org/10.1177/1094342015593156

[86] H. D. Sterck, U. M. Yang, and J. J. Heys, "Reducing complexity in parallel algebraic multigrid preconditioners," *SIAM Journal on Matrix Analysis and Applications*, vol. 27, no. 4, pp. 1019–1039, 2006. [Online]. Available: https://doi.org/10.1137/040615729

[87] N. Bell, S. Dalton, and L. N. Olson, "Exposing fine-grained parallelism in algebraic multigrid methods," *SIAM Journal on Scientific Computing*, vol. 34, no. 4, pp. C123–C152, 2012. [Online]. Available: https://doi.org/10.1137/110838844

[88] R. H. Bisseling and W. Meesen, "Communication balancing in parallel sparse matrix-vector multiplication." *ETNA. Electronic Transactions on Numerical Analysis [electronic only]*, vol. 21, pp. 47–65, 2005. [Online]. Available: http://eudml.org/doc/128024

[89] G. Schubert, H. Fehske, G. Hager, and G. Wellein, "Hybrid-parallel sparse matrix-vector multiplication with explicit communication overlap on current multicore-based systems," *Parallel Processing Letters*, vol. 21, no. 03, pp. 339–358, 2011. [Online]. Available: http://www.worldscientific.com/doi/abs/10.1142/S0129626411000254

[90] R. Geus and S. Röllin, "Towards a fast parallel sparse symmetric matrix-vector multiplication," *Parallel Computing*, vol. 27, no. 7, pp. 883 – 896, 2001, linear systems and associated problems. [Online]. Available: //www.sciencedirect.com/science/article/pii/S0167819101000734

[91] J. R. Gilbert, S. Reinhardt, and V. B. Shah, "A unified framework for numerical and combinatorial computing," *Computing in Science and Engg.*, vol. 10, no. 2, pp. 20–25, Mar. 2008. [Online]. Available: http://dx.doi.org/10.1109/MCSE.2008.45

[92] A. Buluç and J. R. Gilbert, "The combinatorial blas: Design, implementation, and applications," *Int. J. High Perform. Comput. Appl.*, vol. 25, no. 4, pp. 496–509, Nov. 2011. [Online]. Available: http://dx.doi.org/10.1177/1094342011403516

[93] M. O. Rabin and V. V. Vazirani, "Maximum matchings in general graphs through randomization," *J. Algorithms*, vol. 10, no. 4, pp. 557–567, Dec. 1989. [Online]. Available: https://doi.org/10.1016/0196-6774(89)90005-9

[94] R. Yuster and U. Zwick, "Detecting short directed cycles using rectangular matrix multiplication and dynamic programming," in *Proceedings of the Fifteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, ser. SODA '04. Philadelphia, PA, USA: Society for Industrial and Applied Mathematics, 2004. [Online]. Available: http://dl.acm.org/citation.cfm?id=982792.982828 pp. 254–260.

[95] A. Bulu, J. Gilbert, and V. B. Shah, *13. Implementing Sparse Matrices for Graph Algorithms*, pp. 287–313. [Online]. Available: https://epubs.siam.org/doi/abs/10.1137/1.9780898719918.ch13

[96] G. Ballard, A. Druinsky, N. Knight, and O. Schwartz, "Hypergraph partitioning for sparse matrix-matrix multiplication," *ACM Trans. Parallel Comput.*, vol. 3, no. 3, pp. 18:1–18:34, Dec. 2016. [Online]. Available: http://doi.acm.org/10.1145/3015144

[97] A. Bulu and J. R. Gilbert, "Highly parallel sparse matrix-matrix multiplication," vol. abs/1006.2183, 01 2010.

[98] E. Solomonik and J. Demmel, "Communication-optimal parallel 2.5d matrix multiplication and lu factorization algorithms," in *Proceedings of the 17th International Conference on Parallel Processing - Volume Part II*, ser. Euro-Par'11. Berlin, Heidelberg: Springer-Verlag, 2011. [Online]. Available: http://dl.acm.org/citation.cfm?id=2033408.2033420 pp. 90–109.

[99] A. Azad, G. Ballard, A. Bulu, 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. [Online]. Available: https://doi.org/10.1137/15M104253X

[100] A. Lazzaro, J. VandeVondele, J. Hutter, and O. Schütt, "Increasing the efficiency of sparse matrix-matrix multiplication with a 2.5d algorithm and one-sided mpi," in *PASC*, 2017.

[101] 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: ACM, 2013. [Online]. Available: http://doi.acm.org/10.1145/2486159.2486196 pp. 222–231.

[102] K. Censor-Hillel, D. Leitersdorf, and E. Turner, "Sparse matrix multiplication in the congested clique model," 2018.

[103] F. G. Gustavson, "Two fast algorithms for sparse matrices: Multiplication and permuted transposition," *ACM Trans. Math. Softw.*, vol. 4, no. 3, pp. 250–269, Sep. 1978. [Online]. Available: http://doi.acm.org/10.1145/355791.355796

[104] R. E. Bank and C. C. Douglas, "Sparse matrix multiplication package (smmp)," *Advances in Computational Mathematics*, vol. 1, no. 1, pp. 127–137, 1993. [Online]. Available: http://dx.doi.org/10.1007/BF02070824

[105] G. Ballard, C. Siefert, and J. Hu, "Reducing communication costs for sparse matrix multiplication within algebraic multigrid," *SIAM Journal on Scientific Computing*, vol. 38, no. 3, pp. C203–C231, 2016. [Online]. Available: https://doi.org/10.1137/15M1028807

[106] A. Bienz and L. N. Olson, "Improving strong scalability of parallel algebraic multigrid through message agglomeration," *(To be submitted) SISC*, 2018.

[107] R. S. Tuminaro and C. Tong, "Parallel smoothed aggregation multigrid : Aggregation strategies on massively parallel machines," in *Supercomputing, ACM/IEEE 2000 Conference*, Nov 2000, pp. 5–5.

[108] A. Bhatele and L. V. Kale, "Application-specific topology-aware mapping for three dimensional topologies," in *2008 IEEE International Symposium on Parallel and Distributed Processing*, April 2008, pp. 1–8.

[109] A. Bhatele, T. Gamblin, S. H. Langer, P.-T. Bremer, E. W. Draeger, B. Hamann, K. E. Isaacs, A. G. Landge, J. A. Levine, V. Pascucci, M. Schulz, and C. H. Still, "Mapping applications with collectives over sub-communicators on torus networks," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, ser. SC '12. Los Alamitos, CA, USA: IEEE Computer Society Press, 2012. [Online]. Available: http://dl.acm.org/citation.cfm?id=2388996.2389128 pp. 97:1–97:11.

[110] W. Gropp, E. Lusk, N. Doss, and A. Skjellum, "A high-performance, portable implementation of the mpi message passing interface standard," *Parallel Computing*, vol. 22, no. 6, pp. 789 – 828, 1996. [Online]. Available: http://www.sciencedirect.com/science/article/pii/0167819196000245

[111] D. Alber and L. Olson, "Coarsening invariance and bucket-sorted independent sets for algebraic multigrid," *Electronic Transactions on Numerical Analysis*, vol. 37, pp. 367–385, 2010. [Online]. Available: http://etna.mcs.kent.edu/vol.37.2010/pp367-385.dir/pp367-385.html

[112] M. Gee, C. Siefert, J. Hu, R. Tuminaro, and M. Sala, "ML 5.0 smoothed aggregation user's guide," Sandia National Laboratories, Tech. Rep. SAND2006-2649, 2006.

[113] T. Behrens, "Openfoam's basic solvers for linear systems of equations," 01 2009.

[114] "SAMG: Efficiently solving large linear systems of equations," https://www.scai.fraunhofer.de/en/business-research-areas/fast-solvers/products/samg.html.

[115] Y. Notay, "An aggregation-based algebraic multigrid method." *ETNA. Electronic Transactions on Numerical Analysis [electronic only]*, vol. 37, pp. 123–146, 2010. [Online]. Available: http://eudml.org/doc/230391

[116] K. Rupp, P. Tillet, F. Rudolf, J. Weinbub, A. Morhammer, T. Grasser, A. Jngel, and S. Selberherr, "Viennacl—linear algebra library for multi- and many-core architectures," *SIAM Journal on Scientific Computing*, vol. 38, no. 5, pp. S412–S439, 2016. [Online]. Available: https://doi.org/10.1137/15M1026419

[117] S. Dalton, N. Bell, L. Olson, and M. Garland, "Cusp: Generic parallel algorithms for sparse matrix and graph computations," 2014, version 0.5.0. [Online]. Available: http://cusplibrary.github.io/

[118] M. Naumov, M. Arsaev, P. Castonguay, J. Cohen, J. Demouth, J. Eaton, S. Layton, N. Markovskiy, I. Reguly, N. Sakharnykh, V. Sellappan, and R. Strzodka, "Amgx: A library for gpu accelerated algebraic multigrid and preconditioned iterative methods," *SIAM Journal on Scientific Computing*, vol. 37, no. 5, pp. S602–S626, 2015. [Online]. Available: https://doi.org/10.1137/140980260