Using Performance Models to Understand Scalable Krylov Solver Performance at Scale for Structured Grid Problems

Paul R. Eller
University of Illinois at Urbana-Champaign
Urbana, IL
eller3@illinois.edu

Torsten Hoefler
ETH Zurich
Zurich, Switzerland
htor@inf.ethz.ch

William Gropp
University of Illinois at Urbana-Champaign
Urbana, IL
wgropp@illinois.edu

ABSTRACT

Krylov solvers are key kernels in many large-scale science and engineering applications for solving sparse linear systems. Applications running at scale can experience significant slowdown due to factors such as network congestion, off-node congestion, network distance, and performance variation across processes. Performance models can help us better understand factors limiting performance, however simple models fail to capture slowdowns often occurring at scale and performance variation across multiple runs of the same code. This work develops performance models that capture behavior found at scale and uses these models to guide optimizations for Krylov solvers and related kernels using both blocking and non-blocking communication for structured grid problems at scale. We use detailed performance analysis with network performance counters to show how network behavior relates to observed performance and guide the development of performance models that capture the runtime impact of network congestion, network distance, communication and computation overlap, and process mappings. These models guide us to optimize kernels using MPI protocol changes, node-aware communication, and topology-aware communication. The resulting tools and analysis provide us with a better understanding of how to improve performance at scale that can benefit a wider range of applications.

CCS CONCEPTS

• **Theory of computation → Parallel computing models**; • **Computing methodologies → Massively parallel algorithms**.

KEYWORDS

Krylov Solvers, Performance Modeling, Performance Analysis

ACM Reference Format:
Paul R. Eller, Torsten Hoefler, and William Gropp. 2019. Using Performance Models to Understand Scalable Krylov Solver Performance at Scale for Structured Grid Problems. In *2019 International Conference on Supercomputing (ICS '19), June 26–28, 2019, Phoenix, AZ, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3330345.3330358

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
ICS '19, June 26–28, 2019, Phoenix, AZ, USA
© 2019 Association for Computing Machinery.
ACM ISBN 978-1-4503-6079-1/19/06…$15.00
https://doi.org/10.1145/3330345.3330358

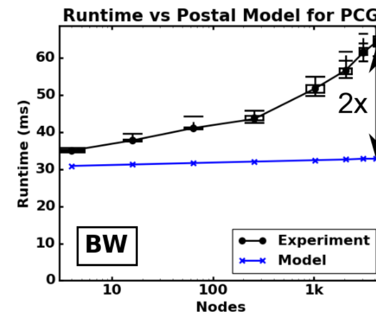

Figure 1: Runtimes vs. postal model expectations for PCG on Blue Waters. Median runtimes with box plots are shown for solving a 27-point Poisson matrix with 8k rows per core.

1 INTRODUCTION

Krylov solvers are key kernels for solving linear systems in many large-scale applications, however these algorithms often experience decreased performance at scale. Large-scale systems have many factors that can increase the cost to send a message through the network or result in performance variation across cores. Furthermore multiple runs of the same code can produce significantly different runtimes due to factors such as the state of the network at runtime and the node allocation a job is given.

However due to the complexity of modern supercomputers it is not always clear which factors heavily impact performance and which are minimized. Some kernels use optimizations such as overlapping communication and computation to seek improved performance, however these optimizations do not always significantly improve runtimes. If we have a better understanding of performance barriers at scale we can optimize kernels to minimize their impact and provide guidance to hardware and software developers for creating systems with improved performance at scale.

Performance models can help us understand key factors affecting performance at scale by comparing the expected performance with different assumptions to observed performance. However simple performance models often fail to capture decreased performance at scale. Furthermore many performance models produce a single predicted runtime for a given algorithm when in practice multiple runs of the same code will produce multiple different runtimes.

Figure 1 demonstrates the decreased performance we see in practice when scaling preconditioned conjugate gradient (PCG) solvers and the large differences between model expectations and observed performance. PCG experiences reduced performance as the node count increases with significant slowdowns at higher node counts however the postal model predicts consistently good performance. This suggests we need to develop more accurate

performance models that account for slowdowns observed at scale and can guide us to better optimize our algorithms.

Krylov solvers such as PCG are used to solve linear systems for a wide range of applications including computational fluid dynamics, wind energy, and particle physics. Quantum chromodynamics (QCD) codes often spend a significant portion of the runtime solving linear systems using Krylov solvers that require halo exchanges and require large supercomputers to run accurate simulations [6, 13]. These factors make QCD a good target application for scalable Krylov solvers and allow us to focus our study on structured grid problems to help prepare for future QCD application studies.

The remainder of this work is structured as follows. Section 2 provides relevant background on Krylov solvers, performance modeling, and HPC systems. Section 3 presents the blocking and non-blocking allreduce, halo exchange, and PCG kernels used in this study. Section 4 presents the test setup and initial performance results with network performance counters. Section 5 presents our performance modeling approach with penalty terms and compares model expectations with observed results. Section 6 presents optimizations guided by our models and improved results for optimized kernels. Section 7 presents key conclusions from this work.

2 BACKGROUND AND RELATED WORK

2.1 Krylov Solvers

Krylov solvers such as PCG [29] are used to iteratively solve linear systems for many scientific applications. More scalable variations have been developed including non-blocking pipelined [15, 19, 22, 25], communication-avoiding [35], and enlarged Krylov subspace methods [24, 43]. We focus on non-blocking pipelined Krylov solvers that rearrange Krylov solvers to decrease the number of allreduces and/or overlap communication and computation using non-blocking allreduces. Similar methods have been developed for GMRES [21], BiCG-Stab [14], and other Krylov solvers.

2.2 Performance Modeling

The postal model [30] uses a fixed cost latency term and per byte cost bandwidth term to model communication. However the simplicity of this model can result in inaccurate performance predictions especially at scale. Previous studies added penalty terms to the postal model for network distance, network congestion, and off-node bandwidth to produce more accurate models [10, 20, 27].

The LogP model and its expanded variations [1, 17, 34, 40] attempt to model parallel communication using more detailed parameters. These models were developed to capture the more complex behavior observed on HPC systems such as network pipelining and overlap, however accurately measuring parameters for this model can be difficult due to the complexity of modern HPC systems.

2.3 Network Performance

A number of studies showed network congestion can significantly reduce performance at scale [7, 8, 23]. Other studies have shown significant network performance variation exists on large scale systems [9, 12, 28] due to network congestion, other jobs, varying message distances, and varying message link types. A few studies [31, 36] discuss best practices for producing meaningful experimental results on large-scale systems.

Many studies looked at the effectiveness of overlapping communication and computation. Significant improvements for non-blocking algorithms over blocking variations have been shown [32, 33, 41]. Other studies analyzed the ability of modern interconnects to overlap communication and computation, showing effective overlap for small messages using an eager protocol [47]. A few studies used overlap to speed up applications [37, 42, 46].

2.4 HPC Systems

To better analyze performance at scale we need to understand the network design and the communication costs on systems we use. Blue Waters uses a Cray Gemini network [38] with a 24^3 3-d torus topology, two compute nodes per network node, and static routing. Messages can take up to 36 hops to cross the network while static routing can cause increased congestion in parts of the network.

Piz Daint uses a Cray Aries [2] network with a dragonfly topology, four compute nodes per network node, and dynamic routing. The network can send messages directly between any two nodes in at most 4 hops, but may use up to 10 hops to avoid congestion.

MPI on both systems provides different message passing protocols depending on message size. The short protocol eager sends very small messages, the eager protocol eager sends messages using extra copies to and from buffers, and the rendezvous protocol sends large messages using synchronized data transfers.

We access hardware performance counters using PAPI [48] to better understand network performance. Studies have analyzed Cray Gemini [45] and Aries [11] hardware performance counters to provide guidance on effectively using counters to understand and improve performance. Cray Gemini and Aries systems provide access to optimized collectives using the DMAPP library including optimized 8 and 16 byte allreduces.

3 KEY KERNELS

This study primarily focuses on a collection of PCG solvers including PCG [29], single allreduce PCG (SAPCG) [18], non-blocking PCG (NBPCG) [25], pipelined PCG (PIPECG) [22], and 2-iteration pipelined PCG (PIPE2CG) [19]. PCG solvers rely on allreduces for dot products and halo exchanges for matrix-vector multiplies and some preconditioners as the primary communication routines. Therefore we look at standalone kernels for these routines to better understand their performance in addition to full PCG solvers.

The NBPCG, PIPECG, and PIPE2CG solvers rely on non-blocking allreduces to overlap communication and computation, guiding us to experiment with blocking and non-blocking allreduce and halo exchange kernels. Busy waits are used to mimic computation while allowing us to easily control their runtime. One busy wait is used to mimic computation that can be overlapped with communication while a second mimics the computation that often follows communication routines within Krylov solvers.

3.1 PCG and Related Kernels

We use blocking and non-blocking allreduce and halo exchange kernels to help us better understand the effectiveness of overlapping communication and computation before moving to more complex Krylov solvers. The halo exchange kernels send messages to neighbors in each dimension to account for sending face elements that

Blocking Allreduce

```
MPI_Allreduce();
Busywait(delay);
Busywait(delay);
```

Non-blocking Allreduce

```
MPI_Iallreduce();
Busywait(delay);
MPI_Wait();
Busywait(delay);
```

Blocking Halo Exchange

```
Halo_Send();
Halo_Receive();
MPI_Waitall();
Busywait(delay);
Busywait(delay);
```

Non-blocking Halo Exchange

```
Halo_Send();
Halo_Receive();
Busywait(delay);
MPI_Waitall();
Busywait(delay);
```

Halo Send

```
for i=1 to ndims
    MPI_Isend(left);
    MPI_Isend(right);
```

Halo Receive

```
for i=1 to ndims
    MPI_Irecv(left);
    MPI_Irecv(right);
```

require larger messages. For simplicity we skip the smaller messages needed in practice to send corner and edge elements.

This work studies the standard PCG solver and four more scalable variations [19]. PCG solves linear systems using matrix-vector multiplies, preconditioner applications, allreduces, and vector operations. Each PCG iteration calls a single matrix-vector multiply and preconditioner application, although scalable variations require additional calls in the initialization step to create a pipeline. We use a matrix-vector multiply that performs a halo exchange overlapping communication with the diagonal block computation and a block-Jacobi incomplete Cholesky preconditioner.

PCG uses two blocking allreduces per iteration while the more scalable variations decrease the number of allreduces per iteration and/or replace blocking allreduces with non-blocking allreduces. The non-blocking allreduces overlap one or more matrix-vector multiply and/or preconditioner calls. We experiment with a 27-point Poisson problem commonly used in benchmarks. We implement these solvers in PETSc [3, 4] using modified versions of provided PCG methods and create a PIPE2CG method. Matrices are stored using the MPIAIJ compressed sparse row format and we use provided matrix-vector multiply and preconditioner routines.

4 INITIAL PERFORMANCE RESULTS

First we run experiments with each kernel to more clearly understand their performance using runtime measurements and network performance counters.

We run experiments on Blue Waters using 16 cores per node (1 process per bulldozer core) and on Piz Daint using 36 cores per multi-core node. We set the max short and eager protocol sizes at 1k and 8k bytes on both systems. We run each algorithm 20 iterations for 50 tests using a $1ms$ busy wait to clear the network prior to each test. Allreduce and halo exchange kernels use busy waits of $1ms$ and $100\mu s$ to mimic computation. Runtimes are measured for each iteration and test and network performance counters are measured for each test. We compute the statistics needed for box plots to show the range of measured values.

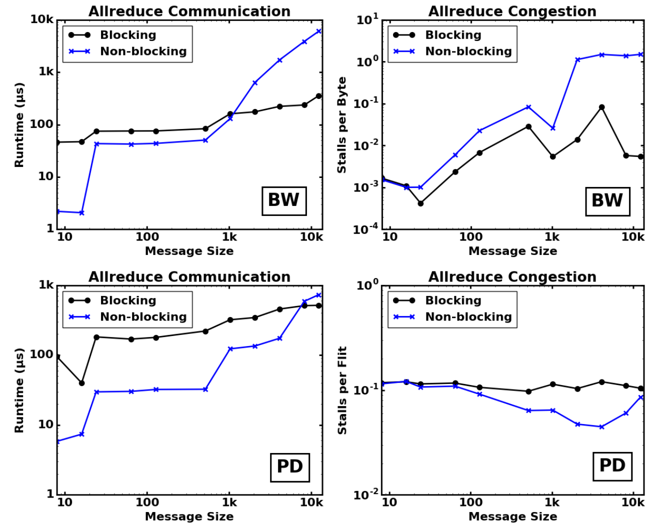

Figure 2: Blocking vs. non-blocking allreduce communication median iteration runtimes and test congestion with $1ms$ busy waits on 1k nodes (16k cores) on Blue Waters (top) and 512 nodes (18k cores) on Piz Daint (bottom).

We are primarily interested in the network performance counters for congestion and bandwidth due to their frequent correlation with runtime and ability to show the effectiveness of optimizations. Network congestion is computed as the ratio of stalls to traffic (measured in phits on Blue Waters and flits on Piz Daint) passing through each network node. Phits on Blue Waters are 3 bytes while flits on Piz Daint vary in size. Higher network congestion increases the time for a message to travel through the network.

We compute network tile counters for each link type. Cray Gemini systems differentiate between X-, Y-, Z-, and host dimension links while Cray Aries systems differentiate between black, green, blue, and host dimension links. The host links connect compute nodes to network nodes. The X-, Y-, and Z-links on Blue Waters differ in the number and type of connections in each dimension. Piz Daint has electrical groups containing 6 sets of 16 node chassis. Black links provide all-to-all connections within each chassis while green links provide 16 sets of all-to-all connections between the 6 chassis. Blue links provide all-to-all links between electrical groups.

Obtaining effective non-blocking communication requires using a progress thread or explicitly giving an MPI thread control to make progress. We split each overlapped computation into chunks and add calls to MPI_Test() to allow MPI to make progress on non-blocking communication. This approach outperformed progress threads for previous solver tests on Blue Waters.

4.1 Allreduce

Figure 2 shows significantly improved performance on both systems for the non-blocking allreduce at smaller message sizes but decreased performance at larger message sizes especially when using the rendezvous protocol.

On Blue Waters we see significantly decreased performance coinciding with increased congestion around 1 stall per byte starting at 2k bytes. This suggests Blue Waters may use a non-blocking allreduce routine with an increased number of messages. We see

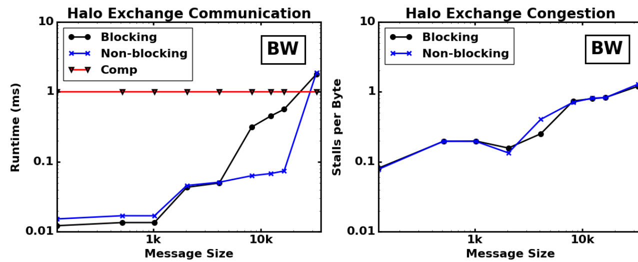

Figure 3: 3-d halo exchange communication median iteration runtimes and test congestion with $1ms$ busy waits on 1k nodes on Blue Waters.

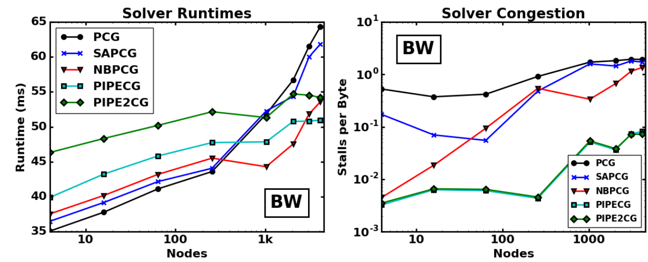

Figure 4: PCG solver median test runtimes and congestion for 27-point Poisson with 8k rows per core on Blue Waters.

similar congestion for all link types with the exception of Y-links. Most large jobs are assigned blocks of nodes that are narrow in the Y-dimensions, resulting in less Y-link traffic.

On Piz Daint we see a larger gap between blocking and non-blocking allreduce communication likely due to dynamic message routing producing increased background congestion throughout the network. This allows the less synchronized non-blocking allreduce to spread out messages more, decrease congestion, and hide the impact of congestion to improve performance. Blue, green, and black links experience reduced congestion, while host links experienced less congestion for very small messages and increased congestion for larger messages that slightly increased overall congestion.

These experiments suggest overlapping communication and computation can be effective for non-blocking allreduces provided the message size is not too large, eager message protocols are used, and the overlap period is large enough.

4.2 Halo Exchange

Figure 3 shows non-blocking halo exchange algorithms effectively overlap communication and computation for larger message sizes where blocking algorithms experience decreased performance on Blue Waters. The decreased performance coincides with increased network congestion, suggesting network congestion limits halo exchange performance. Once communication takes longer than overlapped computation we see a decrease in the non-blocking kernel runtime due to only hiding a small amount of communication. Network congestion is similar for both kernels and most network dimensions experience similar congestion.

Similar tests on Piz Daint show the same performance trends. However Piz Daint has slightly faster communication and less congestion resulting in more effective overlap for a larger range of message sizes and overlap computation periods. Host-links experience the worst congestion, however overall congestion most closely mirrors black-links. Black and green links encounter the most network traffic while host links encounter the least.

These experiments suggest congestion limits performance for halo exchanges, however we can hide the impact of congestion by overlapping communication and computation.

4.3 Preconditioned Conjugate Gradient Solvers

Figure 4 shows blocking PCG solvers experience decreased performance at higher node counts while non-blocking solvers maintain more consistent performance on Blue Waters. Blocking solvers experience more congestion than non-blocking solvers, with the highest congestion coinciding with large performance decreases. Non-blocking solvers reduce congestion by about two orders of magnitude. NBPCG transitions from low to high congestion in part due to the preconditioner failing to fully overlap a non-blocking allreduce. The highest bandwidth usage approaches the max link bandwidth and coincides with the highest congestion for blocking solvers, suggesting once the network is saturated solvers experience high congestion and decreased performance. Most network dimensions experience similar congestion and bandwidth.

Piz Daint shows similar behavior, however tests with fewer rows per core result in steeper performance decreases for blocking methods and slight decreases for less pipelined non-blocking methods. PIPECG and PIPE2CG experience significantly less congestion than other solvers that contributes to their improved performance. The highest bandwidth usage coincides with decreased performance at higher node counts. Host-links experienced the most congestion, while other link types experienced similar congestion. Similar to the halo exchange, black and green links have the most network traffic and host links the least.

These experiments suggest PCG solvers experience slowdown due to congestion, however non-blocking variations can reduce congestion to produce faster, more consistent performance at scale.

5 PERFORMANCE MODELS

Performance modeling is used both for predictive models that provide us with a performance expectation for an algorithm or for analytic models that show where runtime is spent and which factors limit performance. We primarily focus on analytic modeling to more clearly understand which factors limit solver performance at scale and how we can further optimize these methods to reduce the impact of performance barriers. This modeling approach should be effective on a wider range of systems since most face similar performance barriers, but system specific changes may be required.

5.1 Postal Model

The postal model [5, 30] uses a fixed cost latency term and a per byte cost bandwidth term to model communication. We model the time to send a message as $T = \alpha + \beta \cdot n$, where α is latency, β is the inverse of the asymptotic bandwidth, and n is the number of bytes.

This provides a simple approach to modeling parallel communication and a good starting point for developing accurate performance models at scale. However this model does not account for performance barriers such as network congestion and varying hop counts, suggesting penalty terms are needed to improve model accuracy.

To accurately model a given run on an HPC system we need to use data from that run as input to our performance models. Each

Postal Parameters for Blue Waters

Protocol	Off α	Off β	On α	On β
Short	$2.2467\mu s$	$0.3451ns$	$1.1284\mu s$	$0.8154ns$
Eager	$7.1826\mu s$	$0.5092ns$	$2.3950\mu s$	$0.0489ns$
Rdvz	$8.9720\mu s$	$0.1498ns$	$2.1162\mu s$	$0.0954ns$

Postal Parameters for Piz Daint

Protocol	Off α	Off β	On α	On β
Short	$1.7416\mu s$	$0.0041ns$	$0.51055\mu s$	$0.22926ns$
Eager	$4.3080\mu s$	$0.2224ns$	$0.97865\mu s$	$0.03879ns$
Rdvz	$5.9433\mu s$	$0.1083ns$	$0.85828\mu s$	$0.05603ns$

Table 1: Postal model parameters for latency and bandwidth terms for off- and on-node communication.

Hop Costs on Blue Waters

Dim	Host-hop	1-hop	2-hops	3-hops	Hop Lat.
X	$2.146\mu s$	$2.273\mu s$	$2.400\mu s$	$2.525\mu s$	$0.126\mu s$
Y	$2.231\mu s$	$2.309\mu s$	$2.495\mu s$	$2.544\mu s$	$0.104\mu s$
Z	$2.230\mu s$	$2.309\mu s$	$2.422\mu s$	$2.514\mu s$	$0.094\mu s$

Hop Costs on Piz Daint

Type	Host-hop	S. Hop	D. Hop	S. Hop Lat.	D. Hop Lat.
Blue	$2.864\mu s$	$5.607\mu s$	$5.991\mu s$	$2.754\mu s$	$3.113\mu s$
Black	$2.864\mu s$	$2.998\mu s$	$3.219\mu s$	$0.146\mu s$	$0.341\mu s$
Green	$2.864\mu s$	$3.348\mu s$	$3.639\mu s$	$0.496\mu s$	$0.761\mu s$

Table 2: Ping times for the host-hop up to three hops and the average hop latency for each network dimension on Blue Waters and ping times and hop latency for each network dimension using static and dynamic routing on Piz Daint.

run we save data about the node allocation, how the algorithm maps to the node allocation, and the network state during the run. This data allows us to better model the number of messages sent by each algorithm, where each message travels through the network, and the cost to travel through the network. This approach allows different sets of input parameters to produce different runtimes for the same algorithm on a given system.

5.2 Model Parameters

We determine postal model parameters using a ping-pong test with improvements to ensure accurate timings. We run a ping-pong prior to the first test to warm up the system. At the beginning of each iteration we use a barrier to synchronize the processes and delay all processes by $1ms$ to clear the network from previous tests.

We run each test 20 times for 20 iterations and compute statistics for ping-pong times. We compute separate postal model parameters for the short, eager, and rendezvous MPI protocols for both on- and off-node communication. We use linear least squares to fit a linear function to measured ping times to determine α and β and show the results in Table 1.

Table 2 shows the ping and latency costs for messages sent through each network dimension on Blue Waters to better understand the impact of message distance on performance. These values are computed by running 1-byte ping-pong tests with nodes separated by the host-, 1-, 2-, and 3-hops, and then dividing by 2. Taking the average of the average hop latencies for all three dimensions gives an estimated per hop latency of $108ns$, which is close to

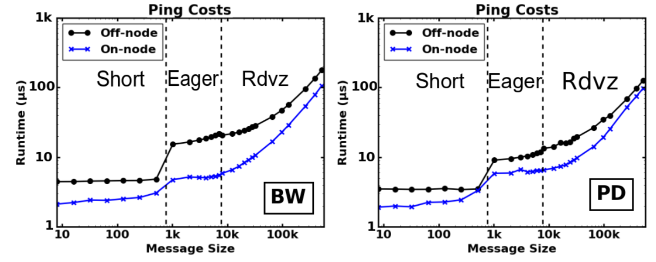

Figure 5: Ping costs for off- and on-node communication.

the white paper hop latency of $105ns$. These results suggest using host-dimension costs to determine model parameters and adding penalties to account for message distance.

Similar tests are run on Piz Daint, however we look at the cost to travel 1-hop across black and green links and 1 or more hops across blue links using both static and dynamic routing. The blue links connecting electrical groups are the most expensive in part due to the longer route they generally need to take and the more limited number of blue links connecting nodes. Static routing results in reduced latency compared to dynamic routing due to messages traveling through fewer nodes, however they may face more congestion. The observed per hop latency times are much larger than the white paper hop latency of $100ns$, suggesting network congestion due to other jobs and dynamic routing significantly increases latency. In practice we use dynamic routing to help reduce congestion, however we need to be aware of the increased latency costs and performance variation.

Figure 5 clearly shows the thresholds between MPI protocols and cheaper costs for on-node messages on both systems. The postal model produces a virtually identical match between ping runtimes and the corresponding postal model expectations. Network performance counters show some congestion on both systems at times during these tests which may produce some runtime variation.

5.3 Algorithms

Next we present postal models for allreduce, halo exchange, and PCG solvers. For these models we compute the cost of sending messages on-node (T_{onmsg}) and off-node (T_{offmsg}) using the postal model with penalty terms.

Allreduce Postal Model

$$
\begin{aligned}
n_{onmsgs} &= \log_2(n_{cores}); \qquad n_{offmsgs} = \log_2(n_{nodes}) \\
T_{comm} &= 2 \cdot n_{onmsgs} \cdot T_{onmsg} + 2 \cdot n_{offmsgs} \cdot T_{offmsg} \\
T_{comp} &= n \cdot T_{flop} \cdot (n_{onmsgs} + n_{offmsgs}) \\
T_{allreduce} &= T_{comm} + T_{comp}
\end{aligned}
$$

The allreduce postal model assumes a reduction is performed on each node and then a single process per node performs the reduction in the network. We assume recursive doubling is used with $\log_2(n_{cores})$ rounds of communication on each node and $\log_2(n_{nodes})$ rounds of communication between nodes. We assume each round of communication involves a message exchange between two processes but these messages are not overlapped.

We add the cost of computing the local reduction using the individual flop cost by computing the inverse of the average peak performance per core. Blue Waters nodes have 16 cores and a peak performance of 313.6 Gflops [44] resulting in an average flop cost of

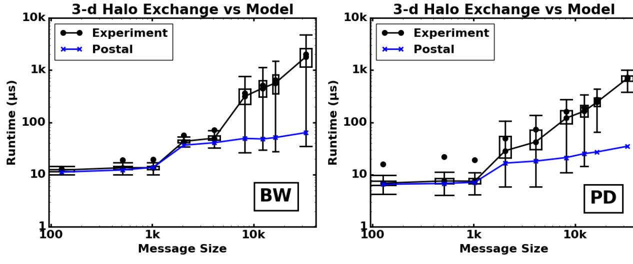

Figure 6: 3-d halo exchange communication vs. postal model on 1k nodes on Blue Waters with $1ms$ busy waits (left) and 512 nodes on Piz Daint with $100\mu s$ busy waits (right).

$0.0510ns$ while Piz Daint multi-core nodes have 36 cores and a peak performance of 1.210 Tflops [16] resulting in an average flop cost of $0.0297ns$. We explored other allreduce postal models however they were less accurate and had less reasonable assumptions.

Halo Exchange Postal Model

$$n_{\text{onmsgs}}, n_{\text{offmsgs}} = \text{load_saved_run_data}()$$
$$T_{\text{halo}} = T_{\text{onmsg}} \cdot n_{\text{onmsgs}} + T_{\text{offmsg}} \cdot n_{\text{offmsgs}}$$

The halo exchange postal model reads network data from saved runs to determine the average and max number of on- and off-node messages. We multiply the number of on- and off-node messages by the message costs and sum these values.

PCG Postal Model

$$T_{\text{comp}} = \text{load_saved_run_data}()$$
$$T_{\text{allr}} = \text{allr_model}() \qquad T_{\text{halo}} = \text{pcg_halo_model}()$$
$$T_{\text{init}} = T_{\text{comp}} + T_{\text{allr}} \qquad T_{\text{iter}} = T_{\text{comp}} + 2 \cdot T_{\text{allr}} + T_{\text{halo}}$$
$$T_{\text{pcg}} = T_{\text{init}} + n_{\text{iters}} \cdot T_{\text{iter}}$$

The PCG postal model uses a more detailed halo exchange to model solving a 3-d 27-point Poisson problem. The model reads data from saved runs for the local grid shape and on-node process grid. These values are used to estimate the number of on- and off-node messages and their size. The number of on- and off-node messages varies, so we estimate message counts for the average process and the slowest process on each node. We read the computation times for the vector operations, preconditioner application, and matrix-vector multiply to allow us to focus on accurately modeling communication. We model a full 3-d halo exchange including face, edge, and corner messages. This results in 6 face, 12 edge, and 8 corner messages for the 27-point stencil. Similar approaches are used for the SAPCG, NBPCG, PIPECG, and PIPE2CG models.

5.4 Initial Model Results

We compare postal model expectations to allreduce, halo exchange, and PCG solver tests. The postal model underpredicts runtimes for all tested kernels with the 3-d halo exchange (Figure 6) and PCG solver models having the largest differences. Therefore we need to add penalty terms to account for these differences.

5.5 Max-rate Penalty

Next we explore postal models using the max-rate penalty [27] to limit bandwidth when using multiple cores per node. In practice the node bandwidth is less than the single process bandwidth times

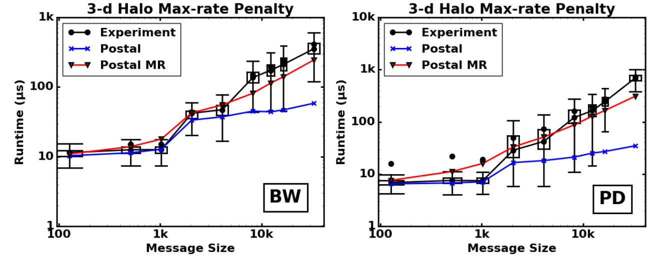

Figure 7: Postal model with max-rate penalty for 3-d halo exchange on 64 nodes on Blue Waters (left) and on 512 nodes on Piz Daint (right) with $1ms$ and $100\mu s$ busy waits.

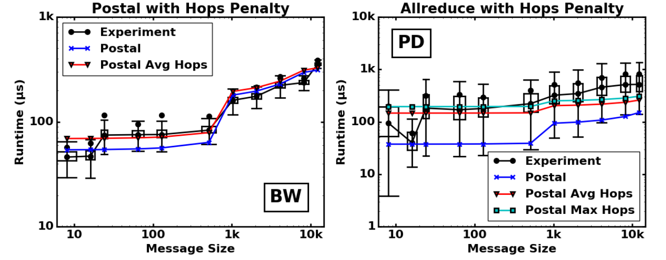

Figure 8: Postal model with hops penalty for blocking allreduce on 1k nodes on Blue Waters (left) and 512 nodes on Piz Daint (right) with $1ms$ busy waits.

the number of cores, allowing this model to produce more accurate predictions when using multiple cores per node.

$$T = \alpha + n_{cores} \cdot n/min(\beta_{node}, n_{cores} \cdot \beta_{core})$$

We run ping-pong tests with 1 to 16 pairs of processes on Blue Waters and 1 to 36 pairs of processes on Piz Daint on two nodes separated by the host-dimension to determine the max off-node bandwidth. We use a ping-pong test and divide the amount of data sent across the network by the difference between the final end time and the initial start time across all cores.

The ping-pong bandwidth shows a clear decrease in performance for larger message sizes and pairs of processes. The max measured node bandwidth used in our models for Blue Waters is 9.03 GB/s and for Piz Daint is 11.77 GB/s. Using the postal model with max-rate penalty matches observed ping-pong runtimes better than the postal model when using multiple cores.

Figure 7 shows improved accuracy for halo exchanges on both systems. On Blue Waters this penalty is especially accurate at lower node counts, however at higher node counts and message sizes there is still a large gap between the model and observed results. On Piz Daint we see a smaller gap between the model and observed results likely due to the increased number of cores per node having a larger performance impact.

5.6 Hops Penalty

The postal model predicted faster allreduce runtimes than achieved in practice for smaller message sizes in particular, but did not account for the increased number of hops a message may travel through. This suggests a hops penalty may account for these differences especially for kernels with multiple rounds of communication. We compute the hops penalty by multiplying the hop latency by the number of hops and adding this to the postal model for the first message in a series of sends.

$$T_{hops} = T_{hop_lat} \cdot n_{hops}$$

We use the $108ns$ computed hop latency on Blue Waters and we can accurately estimate the number of hops due to using static routing. We estimate best, average, and max cases. The best case distance assumes that $\log_2(n_{x/y/z\text{-nodes}})$ steps are used to perform an allreduce in the X-, Y-, Z-, and host network dimensions. We read the node allocation size from saved runs for dimension sizes.

However in practice the allreduce likely does not have an ideal mapping to the allocated node topology or take advantage of the network topology, resulting in longer message distances. We estimate the number of hops by using the node allocation size to determine the max distance a message could travel through the allocation. We assume each message travels on average 1/4 and 1/2 of the max network distance for the average and max cases.

On Piz Daint we use the dynamic hop latencies for each link type and weigh them based on the estimated number of hops along each link type. We estimate the number of hops as the min, average, and max number of hops to reach a node in the same chassis, same electrical group, and different electrical group. We assume nodes are evenly spread through the allocation and allreduces are performed along each link type. The node allocation is read from saved runs.

Figure 8 shows the hops penalty accounts for the difference between the postal model and observed results for smaller message sizes on Blue Waters. On Piz Daint the hops penalty results in a significantly more accurate model for all message sizes due to the increased hop latency costs, likely due to increased background congestion in the network. However for larger message sizes the model underestimates observed performance, suggesting adding additional terms could be helpful.

These results suggest minimizing message distance to improve performance especially for routines with many communication rounds and for smaller messages with a lower bandwidth cost. Systems with more interference from other jobs can experience slowdowns due to increased hop latency costs. The hops penalty did not significantly impact halo exchange or solver models.

5.7 Congestion Penalty

Next we need to account for increased runtimes found on larger node counts and message sizes for halo exchange and PCG routines. The network performance counters suggest a congestion penalty may help account for this reduced performance.

$$T_{congest} = (n_{stalls}/n_{bytes}) \cdot n \cdot n_{cores} \cdot T_{stall}$$

The congestion penalty is computed based on the description in the Gemini and Aries network white papers [2, 38]. On Blue Waters the Gemini NIC operates at 650 MHz and transfers 64 bytes of data in each direction every 5 cycles, resulting in T_{stall} = 1/650 MHz · 5 / 64 = 0.120ns. On Piz Daint the Aries NIC operates at 800 MHz but otherwise performs the same, resulting in T_{stall} = 0.0976ns. The model reads the stall rate from saved runs.

The Piz Daint congestion counters measure the number of stalls per flit, however the number of bytes per flit varies, requiring us to estimate the number of bytes per flit [39]. Network requests use 3, 5, and 14 flits to transfer 0, 1 to 8, and 64 payload bytes plus overhead. Network responses use 1, 3, and 12 flits to transfer 0, 4/8, and 64 payload bytes plus overhead. Therefore we estimate there are 0 to 5.33 bytes of data transferred for each flit. Experiments using 1, 2, and 4 bytes per flit produced fairly similar results, with a larger number of bytes per flit predicting slightly faster runtimes. We use 1 byte per flit since it generally produced more accurate models.

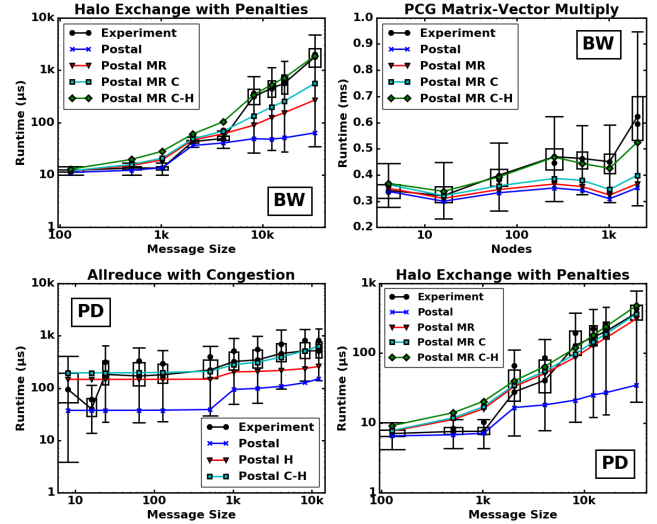

Figure 9: Postal models with max-rate (MR), congestion (C), and hops (H) penalties for halo exchange on 1k nodes and matrix-vector multiply with 4k rows per core for Blue Waters (top) and for allreduce and 3-d halo exchange on 512 nodes on Piz Daint (bottom).

$$T_{congest_hops} = T_{congest} \cdot n_{hops}$$

We combine the hop and congestion penalties by multiplying the congestion penalty by the number of hops. This provides a more accurate model since there is congestion throughout the network that can stall a message on each network tile.

Figure 9 shows adding congestion and hops penalties to the max-rate penalty produces more accurate models that account for the decreased performance found for larger node counts and messages. On Blue Waters all three penalties are needed to account for differences between observed and expected runtimes.

On Piz Daint the congestion and hops penalties are necessary to produce the most accurate models, however their impact is less significant than on Blue Waters. Adding the congestion penalty to the allreduce model in addition to the hops penalty produces a more accurate model likely due to the increased background congestion from dynamic routing. Adding congestion and hops penalties to the halo exchange produces a more accurate model for larger messages, although the max-rate term accounts for most of the runtime. The max-rate term likely has a larger impact than the congestion and hops terms on Piz Daint due to the larger number of cores per node increasing the max-rate penalty, dynamic routing decreasing network congestion, and the shorter max distance between any two nodes in the network limiting the hop count.

These results suggest congestion and hops penalties must be used together to accurately model the impact of network congestion. Systems that require messages to travel through more hops and do not take steps to limit congestion may experience significant slowdowns, while systems with dynamic routing are likely to experience more background congestion that even impacts kernels with lower communication costs.

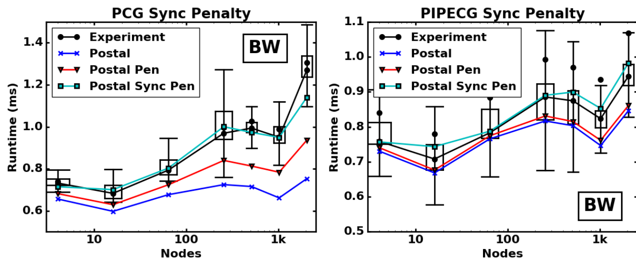

Figure 10: PCG and PIPECG vs. postal model, postal model with penalties, and postal with penalties and synchronization penalty for 4k rows per core on Blue Waters.

While congestion penalties produce more accurate models, they tend to overestimate 3-d halo exchange runtimes for smaller messages using short and eager protocols. These protocols may hide some communication costs by pipelining multiple messages, suggesting we need to model this overlap for blocking kernels.

5.8 Synchronization Penalty

While the matrix-vector multiply model within PCG solvers is reasonably accurate, allreduces within PCG solvers are significantly slower than standalone allreduces. The PCG solvers often have significant performance variation across processes even when there are an equal number of grid cells on every process, suggesting using a synchronization penalty to account for this slowdown.

Therefore we add a synchronization penalty to account for the time the average process must wait once reaching the allreduce until the last process reaches the allreduce and the collective can be completed. We estimate this by computing the difference between the median and max computation runtimes since the last synchronization point and adding this to each allreduce.

We use computation times from saved runs to estimate this penalty. The median runtime is computed from the runtimes for all iterations across all cores while the max runtime is computed from the slowest runtime across all cores each iteration. We take the median of each set of timings. Non-blocking solvers are less synchronized, so we use the first quartile instead of the median.

Figure 10 shows the synchronization penalty produces more reasonable models for both blocking and non-blocking PCG solvers on Blue Waters. The allreduce models within PCG are significantly more accurate and generally produce reasonably close matches to observed performance. Piz Daint produced similar results.

These results suggest performance variation across cores can greatly reduce PCG solver performance. Despite each process having the same number of grid cells, the time spent in computation can regularly vary by over 10% and have outliers that are an order of magnitude slower. This suggests significant performance variation may still exist on large-scale systems that can impact the performance of algorithms at synchronization points.

5.9 Communication and Computation Overlap

The postal model does not account for overlap, however we use two approaches to model overlap. For non-blocking methods we overlap the communication penalty term cost with computation occurring between the communication routine start and the wait call to ensure communication finished. For blocking methods using short or eager protocols we model pipelined communication by overlapping communication penalty terms for all messages with the postal model costs for all messages after the first message.

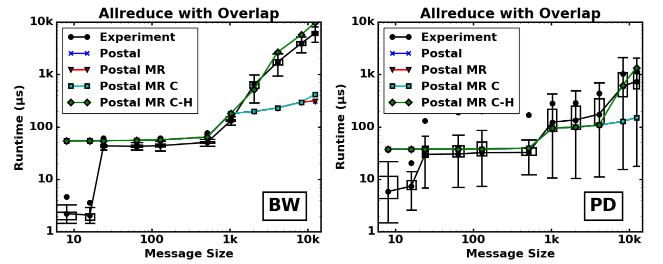

Figure 11: Blocking and non-blocking allreduces vs postal models with $1ms$ overlap on 1k nodes on Blue Waters (left) and 512 nodes on Piz Daint (right).

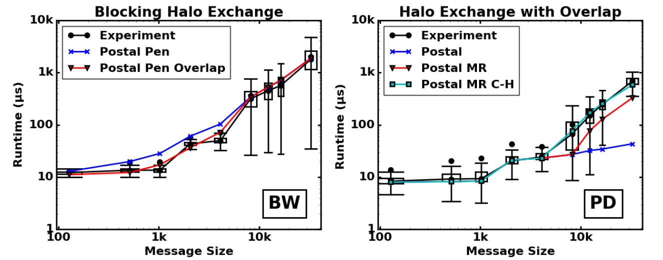

Figure 12: Postal models vs. blocking halo exchanges on 1k nodes on Blue Waters with $1ms$ busy waits (left) and non-blocking halo exchanges on 512 nodes on Piz Daint with $100\mu s$ busy waits (right).

Figure 11 shows the postal model with penalty terms and overlap effectively models a non-blocking allreduce on both systems, including decreased performance for larger messages. Models without all penalty terms incorrectly predict communication will be effectively overlapped.

Figure 12 shows modeling overlap due to pipelining messages in blocking halo exchanges produces more accurate models for smaller messages using short and eager protocols. Modeling overlap for non-blocking halo exchanges accurately predicts effective overlap for smaller messages and accounts for decreased performance for larger messages when overlap is less effective.

These results suggest modeling communication and computation overlap is necessary to produce accurate models both for routines that explicitly overlap communication and computation as well as blocking routines that send a series of messages.

6 IMPROVED RESULTS

Guided by our performance models, we explore optimizations with the potential to reduce the impact of performance barriers to improve performance at scale. Our results demonstrate the optimization effectiveness for scalable Krylov solvers and related kernels and show our models can account for optimization improvements.

6.1 MPI Protocol Changes

First we look at an optimization designed to increase communication and computation overlap to improve performance. Our performance models show accounting for communication and computation overlap for both blocking and non-blocking halo exchanges is

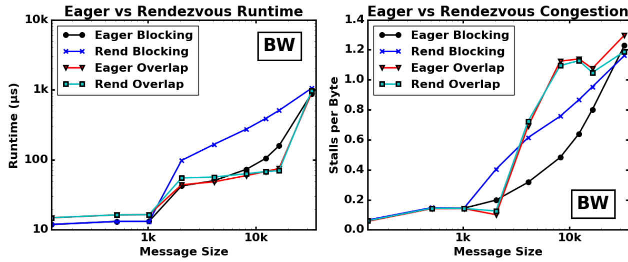

Figure 13: 3-d halo exchange with $1ms$ busy waits comparing eager and rendezvous protocols for 1k or larger messages for communication and congestion on 1k nodes of Blue Waters.

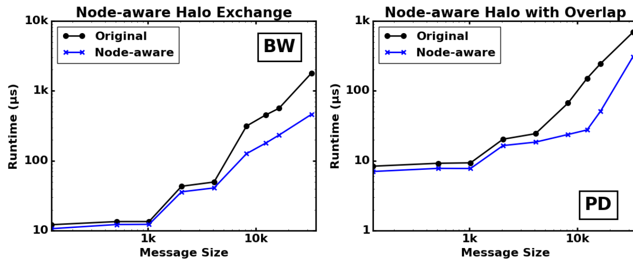

Figure 14: Node-aware 3-d halo exchange communication runtimes for the blocking kernel with $1ms$ busy waits on 1k nodes on Blue Waters and non-blocking kernel with $100\mu s$ busy waits on 512 nodes on Piz Daint.

necessary to produce accurate models. This overlap term produces more accurate models for the short and eager protocols, suggesting using the eager protocol for larger message sizes will allow more effective overlap due to avoiding synchronizing the sending and receiving processes.

Figure 13 shows improved performance for blocking tests for the eager protocol over the rendezvous protocol likely due to pipelining messages. The non-blocking tests show limited improvement due to already overlapping communication and computation. All kernels have similar performance at the largest message size once computation cannot fully overlap communication. Piz Daint shows similar trends for blocking methods, but has slightly reduced performance for non-blocking methods using the eager protocol for larger messages.

6.2 Node-aware Experiments

Next we look at an optimization designed to minimize off-node messages and reduce network congestion. Accounting for the difference between off-node and on-node message costs is critical to producing accurate models. On-node messages have cheaper postal model parameters than off-node messages and do not require penalty terms to produce accurate models. Furthermore decreasing off-node messages reduces network traffic and should reduce the congestion penalty for the remaining off-node messages. This suggests using node-aware communication to minimize the number of off-node messages will improve performance.

The halo exchange and solver kernels use MPI_Cart_create to produce the communication pattern. Ideally the on-node process grid should minimize off-node communication, however in practice this routine rarely chooses the most efficient on-node process grids.

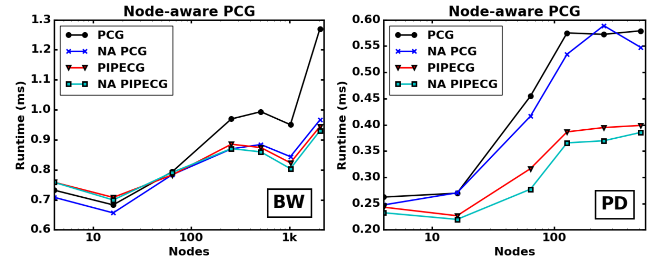

Figure 15: Original vs. node-aware PCG and PIPECG solver runtimes with 4k rows per core on Blue Waters (left) and 2k rows per core on Piz Daint (right).

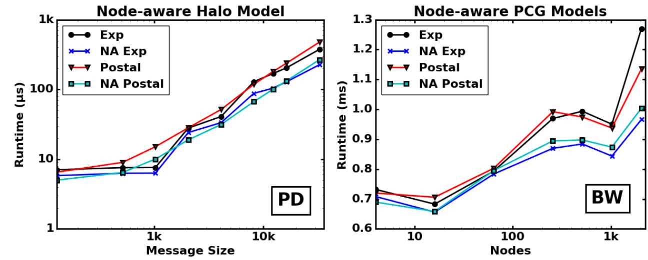

Figure 16: Postal models vs. original and node-aware 3-d halo exchanges with $1ms$ busy waits on 512 nodes on Piz Daint (left) and PCG with 4k rows per core on 1k nodes on Blue Waters (right).

We use a node-aware Cartesian communicator [26] to produce node-aware halo exchange and solver communication patterns. Figure 14 shows improved performance for blocking and non-blocking 3-d halo exchanges especially for larger message sizes. Non-blocking node-aware halo exchanges allow effective overlap to occur for larger message sizes. We observed similar performance trends on both systems.

On Blue Waters node-aware blocking kernels produced less congestion while non-blocking kernels produced less congestion for smaller messages. Reduced congestion and bandwidth was observed in all network dimensions in most cases. On Piz Daint both kernels produced less congestion for larger message sizes and used less bandwidth for all message sizes. Reduced congestion and bandwidth was observed along host, black, and green network links.

Figure 15 shows node-aware solvers produce speedups over the original solvers. Node-aware PCG solvers have runtimes similar to PIPECG solvers on Blue Waters while on Piz Daint node-aware solvers have small but consistent improvements. Node-aware solvers have slightly reduced congestion and bandwidth on Blue Waters while on Piz Daint node-aware PCG has congestion similar to both PIPECG solvers and reduced bandwidth. However host-link congestion is significantly lower for both PIPECG solvers and node-aware communication further decreases it.

Figure 16 shows the postal model with penalty terms can capture the differences between the original and node-aware kernels for halo exchanges and PCG solvers. While the original and node-aware kernels differ in how they assign processes to cores, they both execute the same code. Differentiating between the number of on- and off-node messages in the models and reading network information for each run allows us to produce accurate models for this optimization.

These results show node-aware routines can improve halo exchange and PCG solver performance by reducing off-node messages, which often results in reduced congestion and bandwidth. Our performance modeling approach allows us to accurately model node-aware kernels using the postal model with penalty terms.

On systems with dynamic routing there is potential for the system to route extra outside traffic through nodes using less communication. While this limits the potential benefit for optimizing communication, the ability to reduce bandwidth provides the potential to reduce overall system traffic. This suggests system administrators may want to ensure their software provides access to routines optimizing communication such as node-aware communicators.

6.3 Topology-aware Experiments

Lastly we look at an optimization designed to minimize message distance. The hop and congestion penalties account for significant performance decreases especially when these penalties are combined. Decreasing the number of hops messages travel will decrease both the hop latency cost and the multiplier for the congestion penalty. Decreasing messages distance also reduces network traffic and provides potential to further reduce the congestion penalty. This suggests using topology-aware communication to decrease message distance will improve performance.

We can map 2-d and 3-d halo exchanges to a 3-d torus topology so every node only communicates with neighboring nodes. This requires messages to pass through at most two network tiles, reducing message distance. This routine further minimizes message distance by using node-aware techniques and maximizing the number of host-dimension messages traveling across a single network tile. Unfortunately this requires using a cube of nodes, which is difficult to obtain on Blue Waters and not supported by some systems.

The topology-aware 3-d halo exchange reads network information to determine the position of each process within the allocation. It creates a grid matching the node allocation size and multiplies the grid size by the on-node grid size so that the largest dimension of the on-node grid matches the smallest dimension of the overall grid. The second smallest dimension of the grid is multiplied by the host-dimension size.

While node-aware communication is often effective, we have observed cases where network topology mappings have a larger impact than on-node mappings, allowing the original kernel to outperform the node-aware kernel. This suggests topology-aware communication is necessary to ensure the best performance.

Figure 17 shows improved performance for the topology-aware over the node-aware halo exchange. The topology-aware halo exchange has congestion similar to the node-aware algorithm but less than the original algorithm. Compared to the original algorithm the topology-aware algorithm decreases bandwidth by 66%-74% while the node-aware algorithm decreases bandwidth by 43%-56%. Both obtain the largest reductions for smaller messages.

The 6^3 node grid is the largest cubic grid we obtained on Blue Waters and Piz Daint did not allow requesting a specific node shape. Therefore we use our models to predict halo exchange performance on 4k nodes on Blue Waters and Piz Daint. The model predicts significantly improved performance for larger message sizes on Blue Waters and predicts small improvements for all message sizes on Piz Daint. These results suggest this optimization could be particularly valuable on systems where messages may have to travel a long distance, but should further improve performance even on systems with shorter max distances across the network.

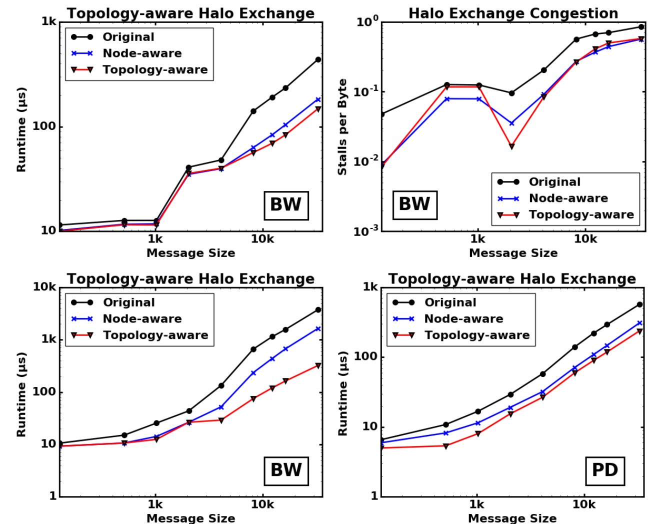

Figure 17: Topology-aware 3-d halo exchange results on 6^3x2 grid for runtime and congestion and predicted 3-d halo exchange results on 4k nodes for Blue Waters and Piz Daint.

7 CONCLUSIONS

This study analyzes the performance of scalable PCG solvers and related kernels for structured grid problems using detailed performance analysis and performance modeling to better understand the issues affecting performance at scale and guide the development of performance optimizations. Detailed performance analysis using runtimes and network performance counters for both blocking and non-blocking communication kernels provides greater insight into factors limiting performance at scale.

These results guide the development of performance models with penalty terms that account for decreased performance at scale and variation across multiple runs of the same code. These models guide us to use optimizations including node-aware and topology-aware communication to improve performance at scale. Experiments on Blue Waters and Piz Daint demonstrate the effectiveness of this performance modeling approach despite the significant differences between the networks as well as provide deeper insight into the issues faced by each network. This modeling approach based on the postal model demonstrated more complex models with more fine-grained penalty terms were not necessary to accurately model these kernels at scale and guide optimizations.

ACKNOWLEDGMENTS

This work was supported in part by the Office of Advanced Scientific Computing Research, Office of Science, U.S. Department of Energy award DE-FG02-13ER26138/DE-SC0010049. This research is part of the Blue Waters sustained-petascale computing project, which is supported by the National Science Foundation (award number OCI 07-25070) and the state of Illinois. We thank the Swiss National Supercomputing Center (CSCS) for providing access to Piz Daint.

REFERENCES

[1] Albert Alexandrov, Mihai F. Ionescu, Klaus E. Schauser, and Chris Scheiman. 1995. 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 (SPAA '95)*. ACM, New York, NY, USA, 95–105. https://doi.org/10.1145/215399.215427

[2] B. Alverson, E. Froese, L. Kaplan, and D. Roweth. 2012. Cray XC Series Network. https://www.cray.com/sites/default/files/resources/CrayXCNetwork.pdf.

[3] S. Balay, S. Abhyankar, M.F. Adams, J. Brown, P. Brune, K. Buschelman, L. Dalcin, V. Eijkhout, W.D. Gropp, D. Kaushik, M.G. Knepley, L.C. McInnes, K. Rupp, B.F. Smith, S. Zampini, and H. Zhang. 2015. *PETSc Users Manual*. Technical Report ANL-95/11 - Revision 3.6. Argonne National Laboratory. http://www.mcs.anl.gov/petsc

[4] Satish Balay, William D. Gropp, Lois Curfman McInnes, and Barry F. Smith. 1997. Efficient Management of Parallelism in Object Oriented Numerical Software Libraries. In *Modern Software Tools in Scientific Computing*, E. Arge, A. M. Bruaset, and H. P. Langtangen (Eds.). Birkhäuser Press, 163–202.

[5] A. Bar-Noy and S. Kipnis. 1994. Designing broadcasting algorithms in the postal model for message-passing systems. *Mathematical systems theory* 27, 5 (01 Sep 1994), 431–452. https://doi.org/10.1007/BF01184933

[6] Evan Berkowitz, M. A. Clark, Arjun Gambhir, Ken McElvain, Amy Nicholson, Enrico Rinaldi, Pavlos Vranas, André Walker-Loud, Chia Cheng Chang, Bálint Joó, Thorsten Kurth, and Kostas Orginos. 2018. Simulating the Weak Death of the Neutron in a Femtoscale Universe with Near-exascale Computing. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis (SC '18)*. IEEE Press, Piscataway, NJ, USA, Article 55, 9 pages. http://dl.acm.org/citation.cfm?id=3291656.3291730

[7] Abhinav Bhatele, Nikhil Jain, Yarden Livnat, Valerio Pascucci, and Peer Timo Bremer. 2016. *Analyzing Network Health and Congestion in Dragonfly-Based Supercomputers*. Institute of Electrical and Electronics Engineers Inc., United States, 93–102. https://doi.org/10.1109/IPDPS.2016.123

[8] A. Bhatele and V. Laxmikant. 2009. An evaluative study on the effect of contention on message latencies in large supercomputers. In *2009 IEEE International Symposium on Parallel Distributed Processing*. 1–8. https://doi.org/10.1109/IPDPS.2009.5161094

[9] A. Bhatele, K. Mohror, S. H. Langer, and K. E. Isaacs. 2013. There goes the neighborhood: Performance degradation due to nearby jobs. In *2013 SC - International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*. 1–12. https://doi.org/10.1145/2503210.2503247

[10] Amanda Bienz, William D. Gropp, and Luke N. Olson. 2018. Improving Performance Models for Irregular Point-to-Point Communication. In *Proceedings of the 25th European MPI Users' Group Meeting (EuroMPI'18)*. ACM, New York, NY, USA, Article 7, 8 pages. https://doi.org/10.1145/3236367.3236368

[11] James M. Brandt, Edwin Froese, Ann C. Gentile, Larry Kaplan, Benjamin A. Allan, and Edward J Walsh. 2016. Network Performance Counter Monitoring and Analysis on the Cray XC Platform. *Cray User's Group Meeting* (May 2016).

[12] Sudheer Chunduri, Kevin Harms, Scott Parker, Vitali Morozov, Samuel Oshin, Naveen Cherukuri, and Kalyan Kumaran. 2017. Run-to-run Variability on Xeon Phi Based Cray XC Systems. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC '17)*. ACM, New York, NY, USA, Article 52, 13 pages. https://doi.org/10.1145/3126908.3126926

[13] M. A. Clark, Bálint Joó, Alexei Strelchenko, Michael Cheng, Arjun Gambhir, and Richard. C. Brower. 2016. Accelerating Lattice QCD Multigrid on GPUs Using Fine-grained Parallelization. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC '16)*. IEEE Press, Piscataway, NJ, USA, Article 68, 12 pages. http://dl.acm.org/citation.cfm?id=3014904.3014995

[14] S. Cools and W. Vanroose. 2017. The Communication-Hiding Pipelined BiCGstab Method for the Parallel Solution of Large Unsymmetric Linear Systems. *Parallel Comput.* 65 (2017), 1 – 20. https://doi.org/10.1016/j.parco.2017.04.005

[15] Jeffrey Cornelis, Siegfried Cools, and Wim Vanroose. 2018. The Communication-Hiding Conjugate Gradient Method with Deep Pipelines. *CoRR* abs/1801.04728 (2018). arXiv:1801.04728 http://arxiv.org/abs/1801.04728

[16] CSCS. 2019. Piz Daint | CSCS. https://www.cscs.ch/computers/piz-daint. https://www.cscs.ch/computers/piz-daint

[17] David Culler, Richard Karp, David Patterson, Abhijit Sahay, Klaus Erik Schauser, Eunice Santos, Ramesh Subramonian, and Thorsten von Eicken. 1993. LogP: Towards a Realistic Model of Parallel Computation. In *Proceedings of the Fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPOPP '93)*. ACM, New York, NY, USA, 1–12. https://doi.org/10.1145/155332.155333

[18] Eduardo D'Azevedo, Victor Eijkhout, and Charles Romine. 1993. *LAPACK Working Note 56: Reducing Communication Costs in the Conjugate Gradient Algorithm on Distributed Memory Multiprocessors*. Technical Report. University of Tennessee, Knoxville, TN, USA.

[19] Paul R. Eller and William Gropp. 2016. Scalable Non-blocking Preconditioned Conjugate Gradient Methods. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC '16)*. IEEE Press, Piscataway, NJ, USA, Article 18, 12 pages. http://dl.acm.org/citation.cfm?id=3014904.3014928

[20] Hormozd Gahvari, Allison H. Baker, Martin Schulz, Ulrike Meier Yang, Kirk E. Jordan, and William Gropp. 2011. Modeling the Performance of an Algebraic Multigrid Cycle on HPC Platforms. In *Proceedings of the International Conference on Supercomputing (ICS '11)*. ACM, New York, NY, USA, 172–181. https://doi.org/10.1145/1995896.1995924

[21] Pieter Ghysels, Thomas Ashby, Karl Meerbergen, and W Vanroose. 2013. Hiding Global Communication Latency in the GMRES Algorithm on Massively Parallel Machines. *SIAM Journal on Scientific Computing* 35, 1 (01 2013), C48–C71.

[22] P. Ghysels and W. Vanroose. 2014. Hiding Global Synchronization Latency in the Preconditioned Conjugate Gradient Algorithm. *Parallel Comput.* 40, 7 (July 2014), 224–238.

[23] Ryan E. Grant, Kevin T. Pedretti, and Ann Gentile. 2015. Overtime: A Tool for Analyzing Performance Variation Due to Network Interference. In *Proceedings of the 3rd Workshop on Exascale MPI (ExaMPI '15)*. ACM, New York, NY, USA, Article 4, 10 pages. https://doi.org/10.1145/2831129.2831133

[24] Laura Grigori, Sophie Moufawad, and Frédéric Nataf. 2014. *Enlarged Krylov Subspace Conjugate Gradient Methods for Reducing Communication*. Research Report RR-8597. INRIA. https://hal.inria.fr/hal-01065985

[25] W. Gropp. 2010. Update on Libraries for Blue Waters. http://jointlab.ncsa.illinois.edu/events/workshop3/pdf/presentations/Gropp-Update-on-Libraries.pdf. http://jointlab.ncsa.illinois.edu/events/workshop3/pdf/presentations/Gropp-Update-on-Libraries.pdf

[26] W. Gropp. 2018. Using Node Information to Implement MPI Cartesian Topologies. In *Proceedings of EuroMPI 18 (EuroMPI18)*.

[27] William Gropp, Luke N. Olson, and Philipp Samfass. 2016. 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 (EuroMPI 2016)*. ACM, New York, NY, USA, 41–50. https://doi.org/10.1145/2966884.2966919

[28] T. Groves, Y. Gu, and N. J. Wright. 2017. Understanding Performance Variability on the Aries Dragonfly Network. In *2017 IEEE International Conference on Cluster Computing (CLUSTER)*. 809–813. https://doi.org/10.1109/CLUSTER.2017.76

[29] M.R. Hestenes and E. Stiefel. 1952. Methods of Conjugate Gradients for Solving Linear Systems. *J. Res. Nat. Bur. Standards* 49, 6 (1952), 409–436.

[30] R. W. Hockney and C. R. Jesshope. 1981. *Parallel Computers: Architecture, Programming and Algorithms*. Institute of Physics Publishing.

[31] Torsten Hoefler and Roberto Belli. 2015. Scientific Benchmarking of Parallel Computing Systems: Twelve Ways to Tell the Masses when Reporting Performance Results. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC '15)*. ACM, New York, NY, USA, Article 73, 12 pages. https://doi.org/10.1145/2807591.2807644

[32] Torsten Hoefler, Peter Gottschling, Andrew Lumsdaine, and Wolfgang Rehm. 2007. Optimizing a conjugate gradient solver with non-blocking collective operations. *Parallel Comput.* 33, 9 (2007), 624 – 633. https://doi.org/10.1016/j.parco.2007.06.006 Selected Papers from EuroPVM/MPI 2006.

[33] T. Hoefler, A. Lumsdaine, and W. Rehm. 2007. Implementation and Performance Analysis of Non-Blocking Collective Operations for MPI. In *Proceedings of the 2007 International Conference on High Performance Computing, Networking, Storage and Analysis, SC07*. IEEE Computer Society/ACM.

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

[35] Mark Frederick Hoemmen. 2010. *Communication-avoiding Krylov subspace methods*. Ph.D. Dissertation. EECS Department, University of California, Berkeley. http://www.eecs.berkeley.edu/Pubs/TechRpts/2010/EECS-2010-37.html

[36] Sascha Hunold, Alexandra Carpen-Amarie, and Jesper Larsson Träff. 2014. Reproducible MPI Micro-Benchmarking Isn'T As Easy As You Think. In *Proceedings of the 21st European MPI Users' Group Meeting (EuroMPI/ASIA '14)*. ACM, New York, NY, USA, Article 69, 8 pages. https://doi.org/10.1145/2642769.2642785

[37] Yasuhiro Idomura, Motoki Nakata, Susumu Yamada, Masahiko Machida, Toshiyuki Imamura, Tomohiko Watanabe, Masanori Nunami, Hikaru Inoue, Shigenobu Tsutsumi, Ikuo Miyoshi, and Naoyuki Shida. 2014. Communication-overlap techniques for improved strong scaling of gyrokinetic Eulerian code beyond 100k cores on the K-computer. *The International Journal of High Performance Computing Applications* 28, 1 (2014), 73–86. https://doi.org/10.1177/1094342013490973 arXiv:https://doi.org/10.1177/1094342013490973

[38] Cray Inc. 2010. The Gemini Network. https://wiki.alcf.anl.gov/parts/images/2/2c/Gemini-whitepaper.pdf.

[39] Cray Inc. 2015. Cray XC Series Network. Cray Doc S-0045-20.

[40] Fumihiko Ino, Noriyuki Fujimoto, and Kenichi Hagihara. 2001. LogGPS: A Parallel Computational Model for Synchronization Analysis. In *Proceedings of the Eighth ACM SIGPLAN Symposium on Principles and Practices of Parallel Programming (PPoPP '01)*. ACM, New York, NY, USA, 133–142. https://doi.org/10.1145/379539.379592

[41] K. Kandalla, U. Yang, J. Keasler, T. Kolev, A. Moody, H. Subramoni, K. Tomko, J. Vienne, B. R. de Supinski, and D. K. Panda. 2012. Designing Non-blocking Allreduce with Collective Offload on InfiniBand Clusters: A Case Study with Conjugate Gradient Solvers. In *2012 IEEE 26th International Parallel and Distributed Processing Symposium*. 1156–1167. https://doi.org/10.1109/IPDPS.2012.106

[42] S. Maeyama, Y. Idomura, M. Nakata, T. Watanabe, N. Nunami, and A. Ishizawa. 2013. Computation-communication overlap techniques for parallel spectral calculations in gyrokinetic Vlasov simulations. *Plasma and Fusion Research* 46, 10 (2013).

[43] Sophie Moufawad. 2014. *Enlarged Krylov Subspace Methods and Preconditioners for Avoiding Communication*. Ph.D. Dissertation. Universite Pierre et Marie Curie. www.theses.fr/2014PA066438.pdf

[44] NCSA. 2019. Blue Waters Sustained Petascale Computing. https://bluewaters.ncsa.illinois.edu. https://bluewaters.ncsa.illinois.edu

[45] Kevin Pedretti, Courtenay Vaughan, Richard Barrett, Karen Devine, and K. Scott Hemmert. 2013. Using the Cray Gemini Performance Counters.

[46] Sreeram Potluri, Ping Lai, Karen Tomko, Sayantan Sur, Yifeng Cui, Mahidhar Tatineni, Karl W. Schulz, William L. Barth, Amitava Majumdar, and Dhabhaleswar K. Panda. 2010. Quantifying Performance Benefits of Overlap Using MPI-2 in a Seismic Modeling Application. In *Proceedings of the 24th ACM International Conference on Supercomputing (ICS '10)*. ACM, New York, NY, USA, 17–25. https://doi.org/10.1145/1810085.1810092

[47] M. J. Rashti and A. Afsahi. 2007. Assessing the Ability of Computation/Communication Overlap and Communication Progress in Modern Interconnects. In *15th Annual IEEE Symposium on High-Performance Interconnects (HOTI 2007)*. 117–124. https://doi.org/10.1109/HOTI.2007.12

[48] Dan Terpstra, Heike Jagode, Haihang You, and Jack Dongarra. 2010. Collecting Performance Data with PAPI-C. In *Tools for High Performance Computing 2009*, Matthias S. Müller, Michael M. Resch, Alexander Schulz, and Wolfgang E. Nagel (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 157–173.