End-to-end Performance Modeling of Distributed GPU Applications

Jaemin Choi
Department of Computer Science
University of Illinois at Urbana-Champaign
Urbana, Illinois
jchoi157@illinois.edu

David F. Richards
Center for Applied Scientific Computing
Lawrence Livermore National Laboratory
Livermore, California
richards12@llnl.gov

Laxmikant V. Kale
Department of Computer Science
University of Illinois at Urbana-Champaign
Urbana, Illinois
kale@illinois.edu

Abhinav Bhatele
Department of Computer Science
University of Maryland
College Park, Maryland
bhatele@cs.umd.edu

ABSTRACT

With the growing number of GPU-based supercomputing platforms and GPU-enabled applications, the ability to accurately model the performance of such applications is becoming increasingly important. Most current performance models for GPU-enabled applications are limited to single node performance. In this work, we propose a methodology for end-to-end performance modeling of distributed GPU applications. Our work strives to create performance models that are both accurate and easily applicable to any distributed GPU application. We combine trace-driven simulation of MPI communication using the TraceR-CODES framework with a profiling-based roofline model for GPU kernels. We make substantial modifications to these models to capture the complex effects of both on-node and off-node networks in today's multi-GPU supercomputers. We validate our model against empirical data from GPU platforms and also vary tunable parameters of our model to observe how they might affect application performance.

CCS CONCEPTS

• **General and reference** → **Performance**; • **Networks** → **Network performance evaluation**.

KEYWORDS

Performance modeling, communication, GPU computing, trace-driven simulation

ACM Reference Format:
Jaemin Choi, David F. Richards, Laxmikant V. Kale, and Abhinav Bhatele. 2020. End-to-end Performance Modeling of Distributed GPU Applications. In *2020 International Conference on Supercomputing (ICS '20), June 29–July 2, 2020, Barcelona, Spain.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3392717.3392737

ICS '20, June 29–July 2, 2020, Barcelona, Spain

ACM ISBN 978-1-4503-7983-0/20/06…$15.00
https://doi.org/10.1145/3392717.3392737

1 INTRODUCTION

Achieving high performance is critical to make the most efficient use of supercomputing platforms and deliver scientific simulation output faster. However, parallel performance is a complex function of many related factors including the parallel algorithm, its implementation, system software, and on-node and off-node network hardware. Modeling the performance of high performance computing (HPC) applications on current and future platforms can help application developers and end users identify opportunities for performance improvement and make decisions to prepare for upcoming HPC systems.

An increasing number of HPC platforms are augmenting multi-core nodes with at least one GPU accelerator. As of November 2019, six out of the ten fastest machines on the TOP500 list [26] are equipped with GPUs. As we move toward these distributed multiple-GPUs-on-a-node environments, the overall performance of an application becomes dependent not only on the CPU and GPU behavior but also on data movement between the host and device memory, and across nodes on the network. There exist several performance models for applications running on GPUs of a single node or on CPU-based clusters. However, there is a lack of performance models for distributed GPU applications running on GPU-based clusters.

In this work, we propose a profiling-based approach to model the end-to-end performance of a distributed GPU application that is both accurate and applicable to a wide array of applications. We extract MPI communication traces and GPU kernel properties of the parallel application by profiling it on a source system, and then use these to predict performance on other systems (with different network and GPU hardware configurations). As a profiling-based approach, minimal preparation and analysis of the application code is required.

We combine trace-driven simulation of MPI communication using the TraceR-CODES framework [1, 24] with profiling-based performance prediction of GPU kernels via the quantitative roofline model [19]. We implement two new models in the TraceR-CODES framework to more accurately reflect the current multi-GPU architectures, first of which is the max-rate model [14] that incorporates the effects of concurrently communicating processes on the same

node. The second, the K-model, is a model that we develop from the max-rate model to more precisely model application-dependent behavior. In addition to these models, we enable detailed modeling of intra-node communication by distinguishing intra-socket, inter-socket and inter-node communication. These improvements allow us to accurately model the complex effects of both on-node and off-node networks prevalent in the latest supercomputers.

We test and validate our approach in the context of proxy applications on GPU-based supercomputers. We also use our validated model to predict the performance of distributed GPU codes in these what-if scenarios: varying number of GPUs per node (with the same total number of GPUs) and changing GPU hardware performance metrics such as flop/s and memory bandwidth. This demonstrates the ability of our model to estimate application performance on various platforms, including future and even hypothetical systems.

It is worth noting that this work focuses on modeling *weak scaling* performance; strong scaling changes the amount of work per process, necessitating a more complex modeling of GPU kernels, data transfers between the host and GPU memory, as well as the CPU behavior.

The primary contributions of this work are summarized below:

- a general methodology to create end-to-end performance models for distributed GPU applications and its validation.
- integration of on-node topology and node-aware analytical models into simulation-based MPI communication modeling.
- development of a new K-model that improves upon the max-rate model with simple application-dependent parameters.
- prediction of application performance in the face of varying platform and hardware parameters.

2 BACKGROUND AND RELATED WORK

Below, we present previous approaches and related work on modeling communication on the network and computation on GPUs.

2.1 Analytical Models for Communication

Analytical models are widely used to model communication performance due to their simplicity and relatively high accuracy. The postal model [4] is one example that defines communication cost simply as the sum of the start-up time α and the per-byte cost β multiplied by the message size n:

$$T = \alpha + \beta \cdot n \tag{1}$$

Separate sets of parameters may be used for short, eager and rendezvous protocols. The parameters are obtained by fitting a (piecewise) linear function to the results of a ping-pong benchmark. Later models attempt to incorporate more details associated with the communication cost: LogP [8] differentiates network latency L and software overhead o in the start-up cost α, and LogGP [2] extends LogP by adding an additional parameter G for large messages. Others also seek to capture network hop counts and contention, adding complexity to the model.

What these models fail to capture, however, is the effect of multiple processes within a node utilizing the network. The max-rate model [14] accounts for the limits of bandwidth coming in and out of an SMP node by considering the impact of multiple processes using the network hardware simultaneously. For example, in an environment with 16 cores per node, up to 16 process pairs (a total of 32 processes mapped to two nodes) could exchange messages simultaneously with their peers placed on the other node.

In this paper, we use a simplified version of the extended max-rate model, which is defined as

$$T = \alpha + \frac{k \cdot n}{R_{C_b} + (k-1) \cdot R_{C_i}} \tag{2}$$

where k is the number of processes per node utilizing the network, n is the message size, R_{C_b} is the base bandwidth sustained by a single process, and R_{C_i} is the bandwidth attainable by additional processes. This version of the model recognizes the fact that R_{C_b} is often greater than R_{C_i}. We simplify $min(R_N, R_{C_b} + (k-1) \cdot R_{C_i})$ in the original paper (R_N is the injection bandwidth) to $R_{C_b} + (k-1) \cdot R_{C_i}$, as we observe that in practice this term is always smaller than and eventually converges to R_N. The parameters of the model (α, R_{C_b}, R_{C_i}) can be empirically determined by fitting multi-process ping-pong benchmark results using a non-linear least squares method[1]. The negative parameters that can be observed with the short message protocol are treated as an artifact, where the model is further reduced to the following two-parameter model:

$$T = \alpha + k \cdot n \cdot \beta \tag{3}$$

Therefore, we use the two-parameter (α, β) model for the short message protocol, and the three-parameter (α, R_{C_b}, R_{C_i}) model for eager and rendezvous protocols.

The max-rate model ensures that we capture performance deviations caused by multiple processes within a node engaging in communication. This is not only important for traditional SMP nodes, but also for multi-GPU environments where one or more processes are mapped to each GPU and distributed-memory communication is performed through MPI.

2.2 Trace-based Simulation for Modeling Communication

Discrete-event simulation can be used to model network activity by explicitly modeling each communication event in an application. In the case of MPI applications, messages passing through the interconnect are modeled as events. These events are typically captured from an execution trace of the application being modeled. The TraceR-CODES framework [1, 24] is one example of a trace-driven simulator. It performs simulations at the packet level and has models for many popular HPC interconnect topologies (e.g. fat-tree, dragonfly, torus).

TraceR-CODES also supports the use of analytical models such as the postal model and LogGP, instead of a detailed network model. In this case, the chosen analytical model is used to calculate how long a message takes to traverse the network, instead of simulating how it travels through switches in the actual network topology. We adopt this method and implement support for the max-rate model and K-model to more accurately model communication performance on multi-GPU systems.

[1] We use a Python library function, `curve_fit` from `scipy.optimize`.

2.3 Modeling GPU Computation

Due to the increasing popularity of GPUs in parallel computation, there has been extensive research on modeling the performance of GPU kernels. Many analytical approaches [3, 17, 20] have been proposed but most require manual analysis based on a deep understanding of the kernel code. On the other hand, empirical approaches such as the quantitative roofline model [19], which is the model employed in this paper, can be applied to a wide range of kernels without a priori knowledge. It is a profiling-based approach that gathers performance metrics at runtime to create a performance model. The target kernel is executed once on any available GPU to obtain its key performance metrics, which is used to predict performance on other GPUs whose hardware properties have been determined via a simple micro-benchmark. Our modifications to the model and its usage are described in detail in Section 3.3.

2.4 Other Related Work

There has been considerable work on modeling the performance of CPU-based MPI applications, either with analytical models [5, 10, 12, 18], simulation [1, 16, 24, 28], or machine learning based methods [21]. Performance models for hybrid MPI-OpenMP applications have been studied in [13, 27]. However, there has been little work in modeling the performance of MPI applications that utilize GPUs [11]. Our work intends to fill this gap and provide a performance modeling methodology for such applications.

3 END-TO-END PERFORMANCE MODEL

We now describe our methodology to developing an end-to-end performance model for distributed GPU applications. We first introduce the K-model, which improves upon the max-rate model. These analytical models are implemented in our node-aware communication modeling framework, NACoM, which builds on TraceR-CODES to predict MPI communication performance. We then discuss the quantitative roofline model that estimates the performance of application kernels on different GPU hardware, and finally describe the method of predicting end-to-end performance by combining the individual modeling components.

3.1 The K-model

We introduced the max-rate model in Section 2.1. While this model improves upon the postal model when multiple processes are simultaneously utilizing network resources, we observed that it tends to overestimate communication time as all messages are assumed to be inter-node.

For example, in a two-dimensional (2D) halo exchange (Listing 1), each process communicates with its four neighbors at each iteration. The max-rate model assumes all the halo exchanges are inter-node; however, a significant number of them are in fact intra-node. This ratio depends on how the physical domain is mapped to MPI processes (which depends on the application) as well as how these processes are assigned to the hardware (nodes and sockets).

Let us consider an example where this code is executed on eight nodes of Summit, a GPU-based supercomputer at Oak Ridge National Laboratory, which has six GPUs per node. With one process per GPU, there will be a total of 48 processes decomposed into a 6 × 8 grid, as in Figure 1. Ranks 0-5 in the first column are mapped to the first node, ranks 6-11 in the second column to the second node, and so on. With this mapping, a process not on the boundary sends only two out of four messages to other nodes whereas the other two messages are sent to processes on the same node. The max-rate model, assuming all messages are sent inter-node, will over-penalize the traversal time of each message.

```
for (int i = 0; i < num_iters; i++) {
  // Each process has 4 neighbors
  for (int j = 0; j < 4; j++) MPI_Irecv(...);
  for (int j = 0; j < 4; j++) MPI_Isend(...);
  MPI_Waitall(...);
}
```

Listing 1: Pseudocode for 2D halo exchange

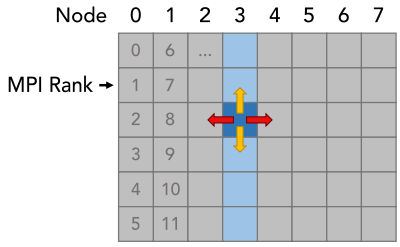


Figure 1: MPI process grid for 2D halo exchange between 48 processes on Summit. Six processes in each column are on the same physical node. The highlighted process (dark blue) sends two intra-node messages (up and down) and two inter-node messages (left and right).

To address this issue, we propose an improvement to the max-rate model, called the K-model. Let us denote the original value of k in the max-rate model (Equation 3) as k', the number of processes engaged in communication on each node. In the K-model, we redefine,

$$k = \frac{K_{inter}}{K_{total}} \cdot k' \tag{4}$$

where K_{inter} is the maximum number of inter-node messages sent by any node (sum over all processes on a node) in the program, and K_{total} is the maximum number of total messages (counting both intra-node and inter-node messages) sent by any node in the program. We take the maximum across all participating nodes because the node with the largest number of messages determines the duration of the communication phase. K_{inter}/K_{total} denotes the fraction of messages originating from a node that are sent over the network. We multiply this fraction with k' to calculate a new k for our model. Substituting k in Equation 3, we get

$$T = \alpha + \frac{K_{inter}}{K_{total}} \cdot k' \cdot n \cdot \beta \tag{5}$$

K_{inter} can either be calculated analytically or measured empirically by inserting a small piece of code in the application where (1) each process counts the total of number of messages that it sends to remote nodes, (2) a summation of the count is performed among processes on the same node, and (3) the maximum value of this per-node summation is computed across all nodes to compute the

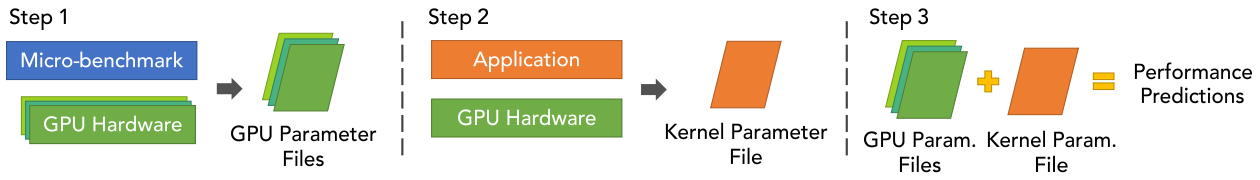


Figure 2: Using the quantitative roofline model (QRoof) in a three-step process to generate performance predictions of application kernels on various GPU hardware.

final value. It should be noted that the K-model adds application-dependent parameters (K_{inter} and K_{total}) to the max-rate model.

In the halo exchange example in Figure 1 with 48 processes, $k' = 6$, $K_{inter} = 2 \times 6 = 12$ and $K_{total} = 4 \times 6 = 24$, so the new k is computed as $(12/24) \times 6 = 3$. This assumes periodic boundary conditions. The K-model essentially scales down k to account for the fact that not all messages are sent across nodes within a communication phase. The validation of this model is performed in Section 5.

3.2 Node-aware Communication Model

Many HPC clusters have multi-socket nodes with each socket having a multi-core processor and optionally, accelerators. For example, a Summit node has two sockets with one processor and three GPUs on each socket for a total of six GPUs. What is often overlooked is that there can be a significant performance differential between intra-socket and inter-socket communication, in addition to inter-node communication. On multi-GPU systems, it is common practice to map one MPI process per GPU, and a process may perform three types of communication: intra-socket, inter-socket, and/or inter-node. All should be accounted for in a performance model for it to be accurate. Furthermore, multiple processes on the same node simultaneously utilizing the network may greatly affect communication performance, creating the need for models such as the max-rate model and K-model.

Our node-aware communication modeling (NACoM) framework implements node-awareness in TraceR-CODES, through modifications to the MPI-replay module and model-net layer as shown in Figure 3. The user can now provide intra-node topology information, including the number of MPI processes per socket, and separate model parameters for intra-socket, inter-socket and inter-node communication. The model-net layer calculates the traversal time of a message using this information. The MPI messaging events are processed in parallel by the underlying parallel discrete event simulation (PDES) framework, ROSS [6].

We use DUMPI [9] parallel execution traces as input in NACoM to build a performance model. These traces contain all the MPI routines called by the program in consideration (with dependencies), and are used by the MPI-replay module to replay the communication primitives. Message traversals are simulated in the model-net layer with user-provided network configuration and model parameters. In this work, the postal, max-rate, and K-models are used as the underlying analytical models to compute message traversal times during the simulation.

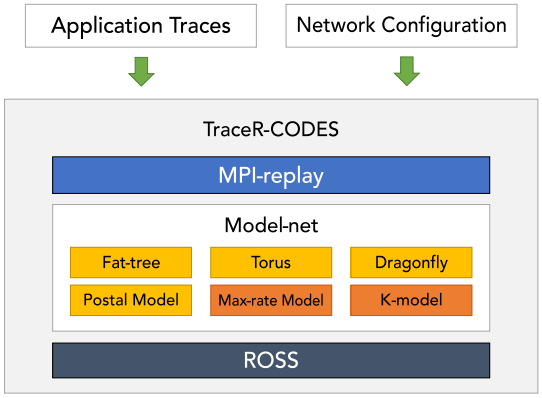


Figure 3: The NACoM framework implements node-awareness in TraceR-CODES by modifying the MPI-replay module and model-net layer.

At the end of the simulation, the framework outputs the predicted communication times in detail, such as the time spent in sending/receiving messages and waiting for requests to complete. This information is used to predict the communication performance of the application, and the same process can be repeated with different model parameters to generate predictions for other platforms.

3.3 Quantitative Roofline Model for GPUs

To model the performance of GPU kernels in the application under consideration, we adopt the quantitative roofline model (hereafter abbreviated as QRoof) briefly described in Section 2.3. We choose this model over others because (1) GPU kernels can be profiled in the same run as MPI trace generation, and (2) any number of kernels can be easily modeled without code inspection. As outlined in Figure 2, this is a three-step process:

(1) Generate GPU parameter files for different types of GPUs by executing a micro-benchmark. Each file contains performance metrics such as single-precision and double-precision flop/s, DRAM access bandwidth, and number of shared memory operations per second.
(2) Generate a kernel parameter file with a profiling run of the application. NVIDIA's nvprof [25] tool is invoked internally to obtain performance metrics such as number of compute operations, size of used DRAM and L2 cache, mix ratio of compute and other operations for the GPU kernels.

Table 1: Architectural details of the evaluation platforms

Platform	No. of nodes	CPU	NVIDIA GPU	GPUs/node	GPUs/socket	Network	MPI
Summit	4,608	IBM Power9	Tesla V100	6	3	Mellanox EDR	IBM Spectrum MPI
Lassen	792	IBM Power9	Tesla V100	4	2	Mellanox EDR tapered	IBM Spectrum MPI
Bridges	32	Xeon-E5 2683 v4	Tesla P100	2	1	Intel Omni-Path	Intel MPI

(3) Combine the kernel parameter file with a GPU parameter file to perform analysis with the quantitative roofline model and obtain performance predictions of the application kernels on that GPU.

If the user wants to predict performance on future GPU hardware or GPUs that they do not have access to, the parameter file can be manually specified with properties such as maximum attainable flop/s and memory bandwidth from the hardware specifications. More details regarding the roofline analysis and validation of kernels on various GPUs can be found in [19].

We have made a number of modifications to the original QRoof implementation to minimize profiling time. The number of profiling runs are now reduced to two: (1) a minimal-overhead profiling step to determine kernel execution times and (2) a metric profiling step to obtain performance characteristics of each kernel. The first step is required for selective profiling, where only the kernels with the most impact on overall performance are chosen to proceed to the metric profiling step. This is done to minimize the number of kernels whose metrics are profiled. We start with the kernel with the longest execution time and add more kernels until the sum of the kernel execution times reaches some threshold, which is currently set to 99% of the total time. We also exclude the CUDA API calls from the profiling set, and restrict the profiling scope to the critical sections of the application. These modifications significantly reduce profiling time in all proxy applications, e.g. from more than 2 hours to 30 minutes in the case of MiniFE.

3.4 Building the End-to-end Model

The individual models discussed above are brought together to create a performance model that accurately predicts the performance of a distributed GPU application. Figure 4 summarizes this method.

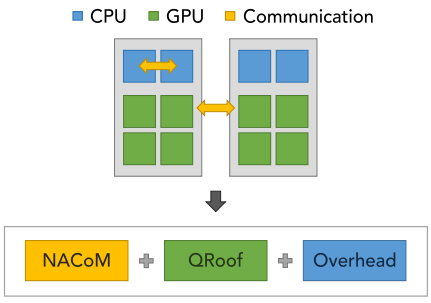


Figure 4: Modeling the end-to-end performance of a distributed GPU application

To estimate the end-to-end runtime, predictions for communication times from NACoM and computation times from QRoof are added together with the overhead, which is measured from the trace generation run at the smallest scale. The measured overhead includes the time for CPU-GPU data transfers and CPU utilization for computation and data structure management. For weak scaling, the modeled performance of GPU kernels and measured overhead can be assumed constant, whereas communication performance is estimated from the simulation results.

To model the same application on a different HPC platform, the platform-dependent parameters in NACoM and QRoof i.e. network and GPU hardware parameters should be obtained with the procedures described in Sections 3.2 and 3.3. The platform-dependent parameters can be reused to create performance models for other applications on the same platform. However, MPI trace files and GPU kernel parameters need to be obtained separately for each new application that needs to be modeled.

4 EXPERIMENTAL SETUP

Next, we describe the set of platforms and proxy applications used to demonstrate and validate our methodology of creating performance models. All applications utilize GPUs to accelerate computation and MPI for distributed memory communication.

4.1 Platforms Used for Validation

We validate the generality and accuracy of our approach by modeling proxy applications on three different supercomputing platforms: Summit at Oak Ridge National Laboratory, Lassen at Lawrence Livermore National Laboratory, and Bridges at Pittsburgh Supercomputing Center. Architectural details of these machines are presented in Table 1. Note that Bridges has an allocation limit of four nodes (eight GPUs) per job, and Lassen has a limit of 256 nodes (1,024 GPUs) in the regular batch queue. All platforms have many more CPU cores than the number of GPUs, and each MPI process mapped to a GPU is executed on a CPU core on the same socket.

The postal model and max-rate model parameters are obtained on all machines via a multi-process ping-pong benchmark, which are provided in Tables 2, 3, and 4. We did not obtain parameters for the intra-socket case on Bridges because only one MPI process is mapped on each socket. Similarly, since only one pair of processes can be mapped to a single socket on Lassen and single node on Bridges, the max-rate model is not applicable for intra-socket on Lassen and inter-socket on Bridges. We use the two-parameter max-rate model (Equation 3) for short messages, and the three-parameter max-rate model (Equation 2) for eager and rendezvous messages.

Table 2: Postal model parameters (α, β)

Mode	Protocol	Summit		Lassen		Bridges	
		α	β	α	β	α	β
Intra-socket	Short	$4.79E{-7}$	$2.99E{-10}$	$3.61E{-7}$	$1.16E{-10}$	N/A (1 process/socket)	
	Eager	$5.96E{-7}$	$1.12E{-10}$	$5.21E{-7}$	$6.44E{-11}$		
	Rendezvous	$2.18E{-6}$	$5.37E{-11}$	$2.92E{-6}$	$3.49E{-11}$		
Inter-socket	Short	$8.52E{-7}$	$3.33E{-10}$	$8.24E{-7}$	$1.05E{-9}$	$6.02E{-7}$	$1.46E{-9}$
	Eager	$1.03E{-6}$	$2.27E{-10}$	$1.11E{-6}$	$2.16E{-10}$	$7.68E{-7}$	$2.05E{-10}$
	Rendezvous	$4.60E{-6}$	$1.18E{-10}$	$2.54E{-6}$	$1.24E{-10}$	$8.76E{-7}$	$1.81E{-10}$
Inter-node	Short	$1.24E{-6}$	$1.01E{-9}$	$1.21E{-6}$	$1.19E{-9}$	$1.36E{-6}$	$3.55E{-9}$
	Eager	$2.86E{-6}$	$1.55E{-10}$	$2.86E{-6}$	$1.52E{-10}$	$2.98E{-6}$	$4.20E{-10}$
	Rendezvous	$7.59E{-6}$	$8.70E{-11}$	$8.81E{-6}$	$7.18E{-11}$	$2.94E{-5}$	$1.59E{-10}$

Table 3: Max-rate model parameters for the MPI short protocol (α, β)

Mode	Summit		Lassen		Bridges	
	α	β	α	β	α	β
Intra-socket	$6.29E{-7}$	$6.21E{-10}$	N/A (Apply postal model)		N/A (1 process/socket)	
Inter-socket	$1.02E{-6}$	$1.45E{-9}$	$1.08E{-6}$	$1.45E{-9}$	N/A (Apply postal model)	
Inter-node	$1.51E{-6}$	$6.32E{-10}$	$1.55E{-6}$	$8.84E{-10}$	$1.39E{-6}$	$5.51E{-9}$

Table 4: Max-rate model parameters for the MPI eager and rendezvous protocols (α, R_{C_b}, R_{C_i})

Mode	Protocol	Summit			Lassen			Bridges		
		α	R_{C_b}	R_{C_i}	α	R_{C_b}	R_{C_i}	α	R_{C_b}	R_{C_i}
Intra-socket	Eager	$7.65E{-7}$	$9.07E9$	$4.32E9$	N/A (Apply postal model)			N/A (1 process/socket)		
	Rendezvous	$3.59E{-6}$	$1.80E10$	$1.53E10$						
Inter-socket	Eager	$1.33E{-6}$	$5.29E9$	$2.69E9$	$1.53E{-6}$	$4.70E9$	$3.84E9$	N/A (Apply postal model)		
	Rendezvous	$4.04E{-6}$	$8.28E9$	$7.08E9$	$3.41E{-6}$	$8.41E9$	$7.09E9$			
Inter-node	Eager	$2.39E{-6}$	$6.68E9$	$1.27E9$	$2.36E{-6}$	$4.92E9$	$2.31E9$	$2.16E{-6}$	$2.20E9$	$1.35E9$
	Rendezvous	$9.33E{-6}$	$1.23E10$	$2.58E7$	$1.06E{-5}$	$1.79E10$	$2.22E9$	$3.26E{-5}$	$6.35E9$	$2.08E8$

4.2 Proxy Applications

Jacobi2D: Jacobi2D is a simple proxy application that implements a 2D Jacobi iteration using GPUs, and exchanges halo regions between neighboring processes. Each MPI process is responsible for a $16{,}384 \times 16{,}384$ block of cells stored on the GPU, with host-side send and receive buffers to temporarily store halo data. An end-to-end run performs 100 iterations, where each iteration consists of the following phases:

(1) Packing halo data (`PACK`): Halo data in non-contiguous memory regions are moved to temporary contiguous device buffers to prepare for data transfer to the host.
(2) Device-to-host transfer of halo regions.
(3) Halo communication between neighboring processes, implemented with non-blocking MPI as in Listing 1.
(4) Host-to-device transfer of received halo data.
(5) Unpacking halo data (`UNPACK`): Halo data is moved from temporary device buffers to the original blocks.
(6) Stencil computation (`STENCIL`, Equation 6).
(7) A sum reduction of all cells in the block (`REDUCE`).
(8) Host-to-device transfer of sum.
(9) Allreduce to compute the global sum.

The following stencil computation is performed on the GPU:

$$A_{i,j} = \frac{1}{2} \times \left(A_{i,j} + \frac{A_{i-1,j} + A_{i+1,j} + A_{i,j-1} + A_{i,j+1}}{4} \right) \quad (6)$$

where $A_{i,j}$ is the cell located at position (i, j) of the grid.

The phases of interest are the halo communication (phase 3) and execution of GPU kernels. We refer to the GPU kernels used in phases 1, 5, 6, and 7 as `PACK`, `UNPACK`, `STENCIL`, and `REDUCE`, respectively. Because we only model weak scaling, CPU-GPU data transfers in phases 2, 4 and 8 are regarded as overhead. Phase 9 is also not included in the model as it involves only one double value and has a negligible effect on iteration time.

We gathered MPI traces for Jacobi2D by weak scaling it up to 1,536 processes (256 nodes) on Summit. The GPU kernels were profiled and overhead times were measured in the two-process execution while collecting MPI traces. We also gathered execution

Table 5: K-model parameters

	Summit ($k' = 6$)			**Lassen** ($k' = 4$)			**Bridges** ($k' = 2$)		
	K_{inter}	K_{total}	k	K_{inter}	K_{total}	k	K_{inter}	K_{total}	k
Jacobi2D	14	24	3.50	10	16	2.50	4	8	1.00
MiniFE	135	156	5.19	92	104	3.54	50	52	1.92
MiniMD	28	36	4.67	18	24	3.00	10	12	1.67

times on Lassen and Bridges for validation, up to 768 processes (192 nodes) and six processes (three nodes), respectively.

MiniFE: MiniFE [22] is a proxy application for unstructured implicit finite-element codes included in the Mantevo suite [15]. The three-dimensional (3D) global grid is partitioned into boxes using a recursive bisection method, each of which is mapped to an MPI process for parallel execution. The core conjugate gradient (CG) solve loop performs the following sequence of operations: `dot` → `waxpby` → `matvec` → `dot` → `waxpby` → `waxpby`. This sequence is repeated for 200 iterations.

The `dot` operation consists of the following phases:

(1) Vector dot product kernel (`DOT`)
(2) Reduction kernel
(3) Device-to-host data transfer
(4) Allreduce of one double value

The reduction kernel is not included in the 99% threshold and only the `DOT` kernel (phase 1) is modeled using QRoof. The rest are considered overhead. The `waxpby` operation consists of a single GPU kernel invocation (`WAXPBY`) that computes $w = ax + by$, which is modeled using QRoof. The `matvec` operation consists of the following phases:

(1) Packing kernel
(2) Device-to-host data transfer
(3) Halo communication
(4) Matrix-vector multiplication kernel (`MATVEC`)

Halo communication (phase 3) is modeled using NACoM and `MATVEC` (phase 4) using QRoof, and the rest are considered as overhead.

For weak scaling, each MPI process is configured to store a 200 × 200 × 200 box of double values and exchanges halo data with neighbors in all three dimensions, with a maximum of 26 neighbors. The size of the halo data exchanged can vary greatly depending on the neighbor's relative position: only one double is sent to a neighbor touching at a corner, whereas 40,000 doubles are sent to one sharing a face.

We generated MPI traces for MiniFE by executing it on up to 256 processes (64 nodes) on Lassen. Similarly to Jacobi2D, GPU kernel parameters and overheads were obtained from the two-process execution. We gathered execution times on Summit and Bridges also for validation, up to 256 processes (43 nodes) and eight processes (four nodes), respectively.

MiniMD: MiniMD [23] is a proxy application that performs molecular dynamics simulations of a Lennard-Jones or EAM system, and is also part of the Mantevo suite. Kokkos [7] is used as the performance portability layer, which translates regular C++ functors and lambda functions to kernels that can be executed on the GPU. The 3D domain is spatially decomposed into boxes, each of which is mapped to an MPI process. Each iteration of the core loop in MiniMD performs the following:

(1) Initial integration (`INIT`)
(2) Exchange of atom information (when re-neighboring is off)
  (a) Packing kernel (`PACK`)
  (b) Device-to-host data transfer
  (c) Halo communication
  (d) Host-to-device data transfer
  (e) Unpacking kernel (`UNPACK`)
(3) Lennard-Jones force calculation (`FORCE`)
(4) Reverse communication (same sub-phases as phase 2)
(5) Final integration (`FIN`)

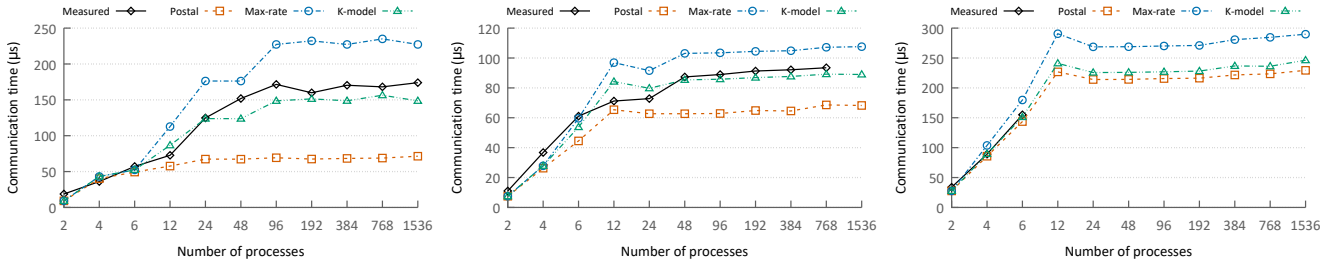
Phases 1 and 5 each consists of a single kernel invocation (`INIT` and `FIN`, respectively), whereas phases 2 and 4 contain non-blocking MPI communication with at most six neighboring processes. These communication calls are surrounded by packing and unpacking kernels (`PACK`/`UNPACK`). To separate the host-device data transfers (which are not modeled and regarded as overhead) and network communication time, we modify the code to not use CUDA-aware MPI. Phase 3 executes the core force calculation kernel, FORCE. For the purposes of this work, we use a Lennard-Jones system defined in the `in.lj.miniMD` file and simulate it for 100 iterations without re-neighboring.

Each MPI process is responsible for a subdomain of 64 × 64 × 64 that contains one million atoms. We weak scaled MiniMD up to 1,024 processes (256 nodes) on Lassen to gather MPI traces and GPU kernel parameters. On Summit and Bridges, we gathered execution times by scaling it up to 1,024 processes (171 nodes) and eight processes (four nodes), respectively.

Finally, we calculate the values of k for the K-model empirically for each application as described in Section 3.1. These values are calculated for each machine separately because the different number of GPUs per node changes the relative number of intra-node and inter-node messages. The K-model parameters obtained for each proxy application are provided in Table 5.

5 MODEL VALIDATION

We now validate our performance model using the various proxy applications and computing platforms. For each application, MPI communication time, GPU kernel time, and end-to-end runtime is measured on all processes. The process with the longest end-to-end runtime is chosen as the representative (since the end-to-end runtime depends on the slowest process), and its time measurements are recorded. These measurements are obtained for three separate



(a) Summit (b) Lassen (c) Bridges

Figure 5: Predictions of the MPI communication time in Jacobi2D using different models.

executions, which are then averaged and compared to the prediction of the performance model to validate its accuracy.

5.1 Jacobi2D

MPI Communication: Using the MPI traces obtained from Summit, we predict the communication performance of Jacobi2D with NACoM and compare the results with the measured baseline performance, as shown in Figure 5. Only up to 256 nodes (1,024 processes) on Lassen and four nodes (eight processes) on Bridges may be allocated, thus jobs requiring larger allocations are only simulated with NACoM.

We first experimented with the postal and max-rate models on Jacobi2D but found neither to come close to the measured times, which motivated us to develop the K-model. The postal model grossly under-predicts the communication time, as it does not take into account the effect of multiple processes simultaneously utilizing network resources. The max-rate model, while addressing this issue, assumes that all communication is inter-node, resulting in overestimated times. By factoring in intra-node communication, the K-model is able to generate more accurate predictions especially on machines with a relatively large number of communicating processes per node, including multi-GPU machines such as Summit.

On 1,536 processes running on 256 nodes of Summit, the postal and max-rate models yield prediction errors of 59% and 31%, respectively, while the K-model results in the lowest error of 15%. The error rates are lower on Lassen for 768 processes on 192 nodes, with 27% and 15% for the postal and max-rate models and 5% for the K-model. This is because with a smaller number of processes per node (four on Lassen versus six on Summit), the prediction accuracy is not as affected by the negligence of the inter-node bandwidth limit (postal model) or the assumption that all messages are inter-node (max-rate model). On Bridges with six processes, with an even smaller number of processes per node of two, the error rates are 7% and 16% for the postal and max-rate models, and 2% for the K-model.

GPU Kernels: We obtain predictions for the GPU kernels of Jacobi2D on the two different GPUs (Tesla V100 on Summit and Lassen, and Tesla P100 on Bridges) using QRoof and compare them with the measured times, as in Figure 6. As Tesla V100 is a newer hardware, we see better performance for all kernels. The predictions are relatively accurate except for PACK and UNPACK on Tesla V100,

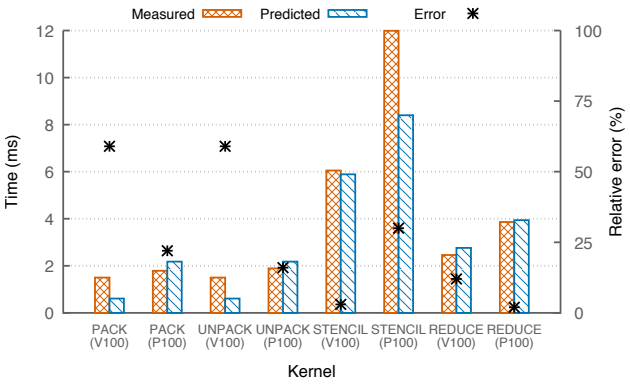
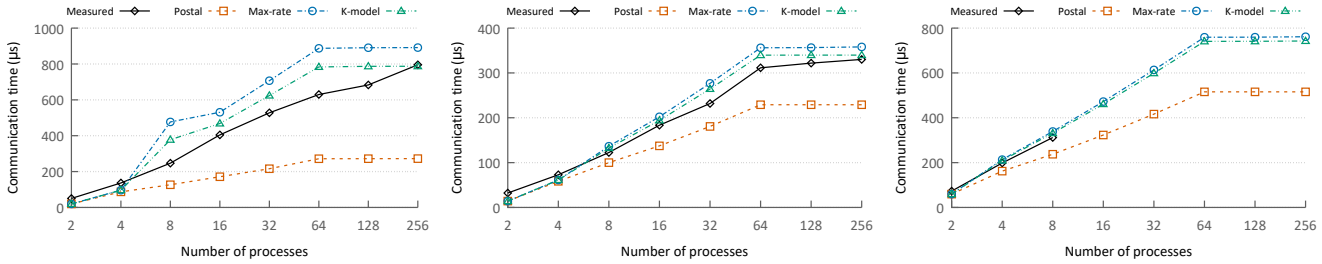


Figure 6: Predictions of the GPU kernel time in Jacobi2D.

with a prediction error of 59%. We believe PACK and UNPACK times were significantly underestimated because the kernels utilize only a small percentage of threads to move the halo data and the memory accesses are strided. Although the source GPU (where kernel parameters were obtained) and the target GPU (where performance is predicted) are the same Tesla V100, predictions are not guaranteed to be accurate since only a small number of performance metrics are used in the QRoof model.

Per Iteration Time: The end-to-end runtime of Jacobi2D is calculated as the product of the number of iterations and time per iteration. We predict the iteration time as the sum of MPI communication time, GPU kernel time, and overhead: the output of the K-model is used for MPI communication time, and QRoof predictions for GPU kernel time. The overhead is measured to be 200 μs from the two-process run on Summit (which also generates MPI traces and GPU kernel parameter files). The same overhead is used in predictions for all scales of execution on all platforms, as we expect it to stay constant with scaling and not change drastically on different platforms. Figure 8 depicts the measured and predicted iteration times for Jacobi2D on the three systems. Due to the node count limit per job on Lassen and Bridges, we only have predictions for configurations with 1,536 processes on 384 nodes of Lassen and 12 or more processes on six or more nodes of Bridges. Because the



(a) Summit (b) Lassen (c) Bridges

Figure 7: Predictions of the MPI communication time in MiniFE using different models.

GPU kernel time is much larger in scale compared to the communication time and overhead, it largely determines the prediction accuracy for iteration time.

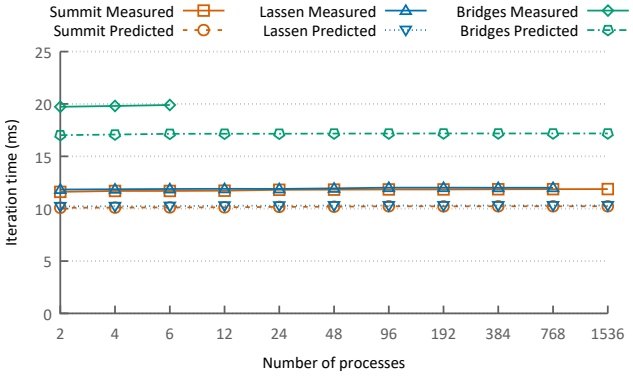


Figure 8: Predictions of the per iteration time in Jacobi2D on different platforms.

5.2 MiniFE

MPI Communication: MPI traces obtained from Lassen are used to simulate the communication time of MiniFE with the three models in NACoM. These predictions are validated against the measured communication time, as shown in Figure 7. The same allocation limits on Lassen and Bridges are applied. We observe that the K-model yields the most accurate predictions on all platforms.

Because of the three-dimensional simulation space in MiniFE, there is a high probability that some neighboring processes in the MPI Cartesian grid are placed on distant nodes at larger node counts. This increases the effect of network congestion and number of hops on communication performance (which is not modeled), causing the actual communication time to increase beyond 64 processes in contrast to the model predictions.

The K-model estimates are more similar to the max-rate model in MiniFE than Jacobi2D as the k values are closer to the original values, as described in Table 5. This demonstrates that the level of discrepancy between the K-model and the max-rate model depends on the application.

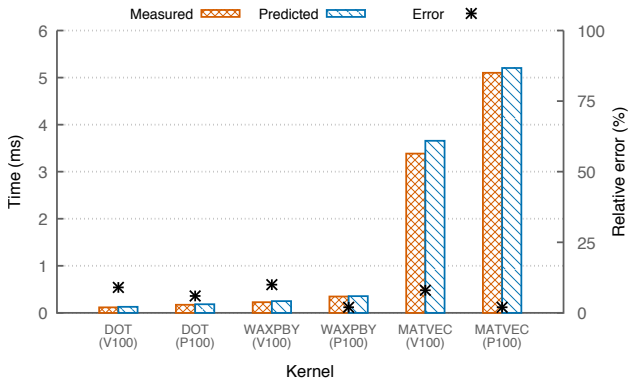


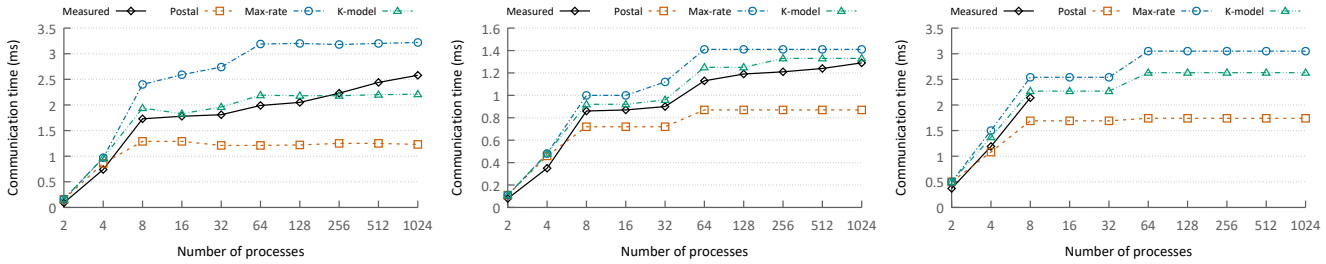
Figure 9: Predictions of the GPU kernel time in MiniFE.

GPU Kernels: As shown in Figure 9, predictions for the kernels in MiniFE are highly accurate for both devices as they are computationally intensive and fit well into the roofline model. The prediction errors range from 2-10%.

Per Iteration Time: As the CG solve step of MiniFE is iterative, its execution time can be estimated as the product of the number of iterations and time per iteration. Iteration time is predicted as the sum of MPI communication time, GPU kernel time and overhead. The overhead is measured to be 165 μs from the two-process profiling run on Lassen, which is used for MiniFE predictions on all process counts and platforms. The measured and predicted iteration times are compared in Figure 11. Due to the node count limit, configurations with more than four nodes (eight processes) on Bridges are only predicted through our performance model. The GPU kernel times constitute most of the iteration time, and as their accuracy is high on both GPU hardware, the resulting predictions for iteration time yield low prediction errors: 6% and 4% at 256 processes on 43 nodes of Summit and 64 nodes of Lassen, respectively, and 0.5% at eight processes on four nodes of Bridges.

5.3 MiniMD

MPI Communication: MiniMD performs two phases of communication in each iteration, once in step 2 and another in step 4. Using the MPI traces obtained on Lassen, the aggregated communication



(a) Summit (b) Lassen (c) Bridges

Figure 10: Predictions of the MPI communication time in MiniMD using different models.

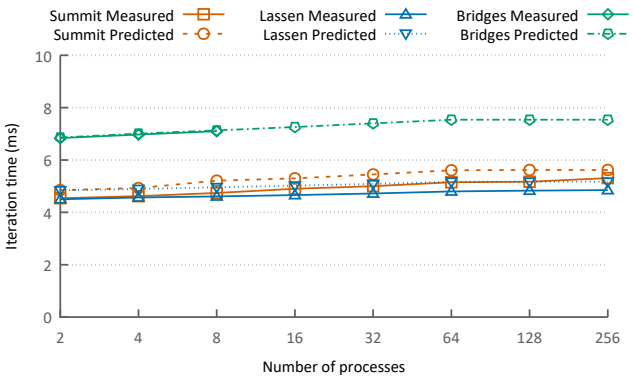


Figure 11: Predictions of the per iteration time in MiniFE on different platforms.

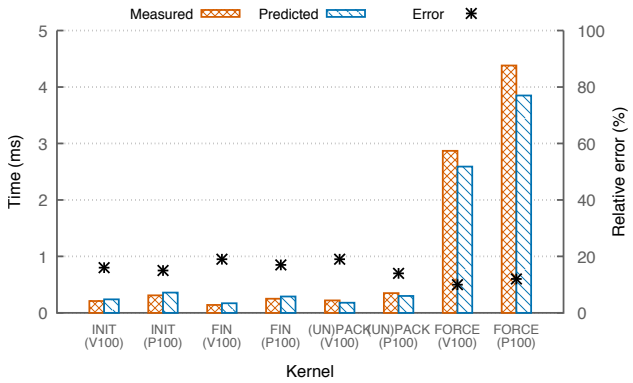


Figure 12: Predictions of the GPU kernel time in MiniMD.

time per iteration is predicted with the three models in NACoM and compared with the measured times on all platforms. The results are shown in Figure 10. Again, communication time is only simulated for configurations with more than four nodes (eight processes) on Bridges. Other factors such as network congestion and hop count negatively affect performance at larger node counts, but with less impact compared to MiniFE due to the smaller communication volume (maximum of six versus 26 neighbors).

GPU Kernels: The integration kernels (`INIT`, `FIN`), pack/unpack kernels (`PACK`, `UNPACK`), and force calculation kernel (`FORCE`) are included in the 99% execution time threshold and modeled through QRoof. The relative errors for the performance predictions range between 10% and 19%, as shown in Figure 12. In favor of space, we combine `PACK` and `UNPACK` as one since the measured and predicted times are very similar.

Per Iteration Time: The end-to-end runtime of MiniMD can be computed as the product of the number of iterations and iteration time. The measured overhead is 148 μs with two processes on Lassen, which is used to predict the iteration time along with the K-model predictions for MPI communication and QRoof predictions for GPU kernels, as shown in Figure 13. Executions with more than four nodes on Bridges are only predicted through the performance model. Since communication is a more significant factor of iteration time in MiniMD compared to the other proxy applications, the usage of the K-model over the postal and max-rate models significantly improves prediction accuracy of iteration time. The prediction errors are 15% and 11% with 1,024 processes on 171 nodes of Summit and on 256 nodes of Lassen, respectively, and 9% with eight processes on four nodes of Bridges. Note that using a simulation-based approach allows us to quickly predict performance at larger scales than limited by the physical machine, as illustrated by the predictions for Bridges.

6 PREDICTIONS USING THE MODEL

We take a step further with our validated model and study some what-if scenarios by tuning the parameters in our performance model. Using MiniFE as the target application with MPI traces at 256 processes and kernel parameters obtained on Lassen, we observe how changes in the model parameters affect its performance. Note that the same process can be applied to any distributed GPU application with MPI traces and kernel parameters. Some of the tunable parameters in our performance model are:

- number of GPUs per node
- GPU flop/s and memory bandwidth
- network latency and bandwidth (intra- and inter-node)

We tune the first two parameters and observe how they affect performance, as the results from tuning the third are self-evident (decrease/increase in communication time).

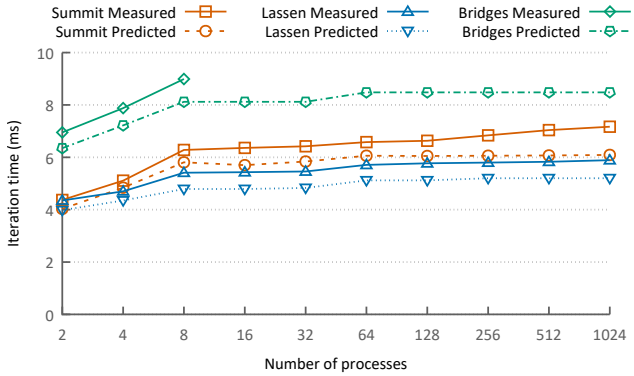


Figure 13: Predictions of the per iteration time in MiniMD on different platforms.

6.1 Number of GPUs per Node

We modify an architectural parameter, number of GPUs per node, and observe how our model predicts application performance. The total number of GPUs in the system is kept the same. The number of GPUs per node is increased from one to 16, with the total number of nodes decreasing from 256 to 16. This mimics compute nodes becoming "fatter", with a decreasing number of nodes in the system. GPUs on a node are evenly divided into two sockets, and the max-rate model with parameters from Lassen is used as the underlying communication model. This allows us to simply set k equal to the number of MPI processes (GPUs) per node.

Figure 14 depicts the prediction results. The postal model predicts that communication times will decrease as each node contains more GPUs (and processes), as more intra-node communication occurs in place of inter-node communication. However, the max-rate model predictions suggest that despite the increase in intra-node communication, communication performance degrades due to multiple processes simultaneously utilizing the network. Iteration time is not significantly affected by communication, as GPU kernels account for the majority of the execution time.

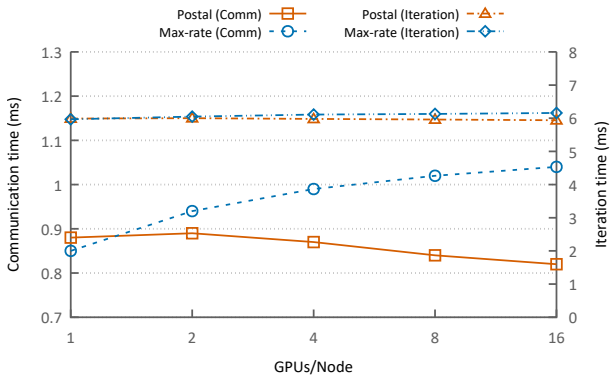


Figure 14: Predicted communication and per iteration time of MiniFE with varying number of GPUs per node.

6.2 GPU Flop/s and Memory Bandwidth

We modify the GPU parameter file for Tesla V100 to study which parameters affect kernel performance and how it affects the end-to-end runtime. The resulting hypothetical devices are named A, B, C, D: A has twice the flop/s of Tesla V100, B has half the flop/s, C has double the memory bandwidth, and D has half the bandwidth. As can be seen from Figure 15, the kernel times are only affected by the change in memory bandwidth as QRoof classifies all kernels as memory-bound. The change in MATVEC time has the most impact on iteration time due to its relative scale. This demonstrates the ability of the performance model to predict application performance in the face of changes in GPU hardware.

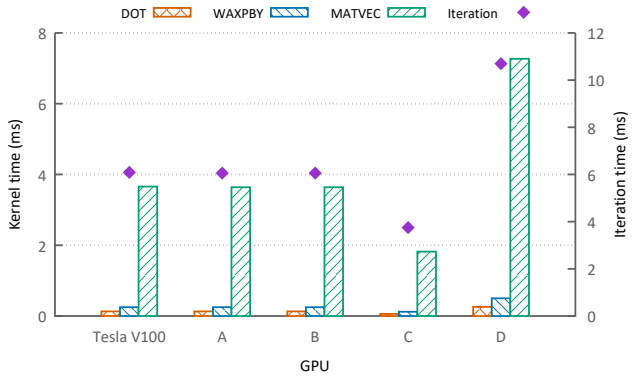


Figure 15: Predicted kernel and per iteration time of MiniFE with different GPU parameters.

7 CONCLUSION

We have developed a general methodology to create end-to-end performance models for distributed GPU applications that combines modeling of MPI communication and GPU computation. To model communication without a priori knowledge of the communication pattern, we adopt a simulation-based approach using the TraceR-CODES framework. We improve upon the max-rate model with simple application-dependent parameters and introduce the K-model, which is used as the underlying analytical method to compute message traversal times during simulation. The GPU quantitative roofline model is used to generate predictions for different GPU hardware, with improvements to reduce profiling time and thus enhance its usability. Weak scaling behavior of an application can then be predicted by combining the outputs of the communication and GPU performance models with the measured overhead.

Our approach is validated with a set of proxy applications on several multi-GPU platforms with varying network capabilities and GPU hardware. The created performance model allows quick and accurate predictions of application performance, and we demonstrate this by tuning different model parameters and comparing the results. There are applications with irregular workloads and/or computation-communication overlap that are not covered by our work. The challenge in predicting performance for such applications lies in determining the communication pattern and amount of overlap, and is left as future work.

In the future, we plan to integrate models for CPU-GPU data transfers, GPU kernels with varying input data sizes, and computation on CPU cores. This would allow us to model strong scaling performance in addition to weak scaling and improve the versatility of our approach. We also plan to support recent technologies for data movement such as CUDA-aware MPI and NVLink. Emulation-based extrapolation techniques will be also explored to obviate the need to obtain MPI traces at all scales of execution.

ACKNOWLEDGMENTS

This work was performed under the auspices of the U.S. Department of Energy (DOE) by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344 (LLNL-CONF-809401). This research was supported by the Exascale Computing Project (17-SC-20-SC), a collaborative effort of the U.S. DOE Office of Science and the National Nuclear Security Administration. This work was supported by funding provided by the University of Maryland College Park Foundation.

This research used resources of the Oak Ridge Leadership Computing Facility at the Oak Ridge National Laboratory, which is supported by the Office of Science of the U.S. DOE under Contract No. DE-AC05-00OR22725. This work used the Extreme Science and Engineering Discovery Environment (XSEDE), which is supported by National Science Foundation grant number ACI-1548562. Specifically, it used the Bridges system, which is supported by NSF award number ACI-1445606, at the Pittsburgh Supercomputing Center.

REFERENCES

[1] Bilge Acun, Nikhil Jain, Abhinav Bhatele, Misbah Mubarak, Christopher D. Carothers, and Laxmikant V. Kale. 2015. Preliminary Evaluation of a Parallel Trace Replay Tool for HPC Network Simulations. In *Euro-Par 2015: Parallel Processing Workshops*. Springer International Publishing, Cham, 417–429.

[2] Albert Alexandrov, Mihai F. Ionescu, Klaus E. Schauser, and Chris Scheiman. 1997. LogGP: Incorporating Long Messages into the LogP Model for Parallel Computation. *J. Parallel and Distrib. Comput.* 44, 1 (1997), 71 – 79. https://doi.org/10.1006/jpdc.1997.1346

[3] Sara S. Baghsorkhi, Matthieu Delahaye, Sanjay J. Patel, William D. Gropp, and Wen-mei W. Hwu. 2010. An Adaptive Performance Modeling Tool for GPU Architectures. In *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (Bangalore, India) *(PPoPP '10)*. ACM, New York, NY, USA, 105–114. https://doi.org/10.1145/1693453.1693470

[4] Amotz Bar-Noy and Shlomo Kipnis. 1992. Designing Broadcasting Algorithms in the Postal Model for Message-passing Systems. In *Proceedings of the Fourth Annual ACM Symposium on Parallel Algorithms and Architectures* (San Diego, California, USA) *(SPAA '92)*. ACM, New York, NY, USA, 13–22. https://doi.org/10.1145/140901.140903

[5] Abhinav Bhatele, Pritish Jetley, Hormozd Gahvari, Lukasz Wesolowski, William D. Gropp, and Laxmikant Kale. 2011. Architectural Constraints to Attain 1 Exaflop/s for Three Scientific Application Classes. In *Proceedings of the 2011 IEEE International Parallel & Distributed Processing Symposium (IPDPS '11)*. IEEE Computer Society, Washington, DC, USA, 80–91. https://doi.org/10.1109/IPDPS.2011.18

[6] C. D. Carothers, D. Bauer, and S. Pearce. 2000. ROSS: a high-performance, low memory, modular time warp system. In *Proceedings Fourteenth Workshop on Parallel and Distributed Simulation*. 53–60. https://doi.org/10.1109/PADS.2000.847144

[7] H. Carter Edwards, Christian R. Trott, and Daniel Sunderland. 2014. Kokkos. *J. Parallel Distrib. Comput.* 74, 12 (Dec. 2014), 3202–3216. https://doi.org/10.1016/j.jpdc.2014.07.003

[8] 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* (San Diego, California, USA) *(PPOPP '93)*. ACM, New York, NY, USA, 1–12. https://doi.org/10.1145/155332.155333

[9] DUMPI 2020. *SST DUMPI Trace Library*. Retrieved April 28, 2020 from https://github.com/sstsimulator/sst-dumpi

[10] 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 *Proceedings of the ACM International Conference on Supercomputing* (Phoenix, Arizona) *(ICS '19)*. ACM, New York, NY, USA, 138–149. https://doi.org/10.1145/3330345.3330358

[11] Christian Feichtinger, Johannes Habich, Harald Köstler, Ulrich Rüde, and Takayuki Aoki. 2015. Performance modeling and analysis of heterogeneous lattice Boltzmann simulations on CPU–GPU clusters. *Parallel Comput.* 46 (2015), 1 – 13. https://doi.org/10.1016/j.parco.2014.12.003

[12] H. Gahvari and W. Gropp. 2010. An introductory exascale feasibility study for FFTs and multigrid. In *2010 IEEE International Symposium on Parallel Distributed Processing (IPDPS)*. 1–9. https://doi.org/10.1109/IPDPS.2010.5470417

[13] H. Gahvari, W. Gropp, K. E. Jordan, M. Schulz, and U. M. Yang. 2012. Modeling the Performance of an Algebraic Multigrid Cycle Using Hybrid MPI/OpenMP. In *2012 41st International Conference on Parallel Processing*. 128–137. https://doi.org/10.1109/ICPP.2012.41

[14] 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* (Edinburgh, United Kingdom) *(EuroMPI 2016)*. ACM, New York, NY, USA, 41–50. https://doi.org/10.1145/2966884.2966919

[15] Michael A Heroux, Douglas W Doerfler, Paul S Crozier, James M Willenbring, H Carter Edwards, Alan Williams, Mahesh Rajan, Eric R Keiter, Heidi K Thornquist, and Robert W Numrich. 2009. *Improving Performance via Mini-applications*. Technical Report SAND2009-5574. Sandia National Laboratories.

[16] Torsten Hoefler, Timo Schneider, and Andrew Lumsdaine. 2010. LogGOPSim: Simulating Large-scale Applications in the LogGOPS Model. In *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing* (Chicago, Illinois) *(HPDC '10)*. ACM, New York, NY, USA, 597–604. https://doi.org/10.1145/1851476.1851564

[17] Sunpyo Hong and Hyesoon Kim. 2009. An Analytical Model for a GPU Architecture with Memory-level and Thread-level Parallelism Awareness. In *Proceedings of the 36th Annual International Symposium on Computer Architecture* (Austin, TX, USA) *(ISCA '09)*. ACM, New York, NY, USA, 152–163. https://doi.org/10.1145/1555754.1555775

[18] Darren J. Kerbyson and Philip W. Jones. 2005. A Performance Model of the Parallel Ocean Program. *Int. J. High Perform. Comput. Appl.* 19, 3 (Aug. 2005), 261–276. https://doi.org/10.1177/1094342005056114

[19] Elias Konstantinidis and Yiannis Cotronis. 2017. A quantitative roofline model for GPU kernel performance estimation using micro-benchmarks and hardware metric profiling. *J. Parallel and Distrib. Comput.* 107 (2017), 37 – 56. https://doi.org/10.1016/j.jpdc.2017.04.002

[20] K. Kothapalli, R. Mukherjee, M. S. Rehman, S. Patidar, P. J. Narayanan, and K. Srinathan. 2009. A performance prediction model for the CUDA GPGPU platform. In *2009 International Conference on High Performance Computing (HiPC)*. 463–472. https://doi.org/10.1109/HIPC.2009.5433179

[21] P. Malakar, P. Balaprakash, V. Vishwanath, V. Morozov, and K. Kumaran. 2018. Benchmarking Machine Learning Methods for Performance Modeling of Scientific Applications. In *2018 IEEE/ACM Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS)*. 33–44. https://doi.org/10.1109/PMBS.2018.8641686

[22] MiniFE 2020. *Mantevo/MiniFE*. Retrieved April 28, 2020 from https://github.com/Mantevo/miniFE

[23] MiniMD 2020. *Mantevo/MiniMD*. Retrieved April 28, 2020 from https://github.com/Mantevo/miniMD

[24] M. Mubarak, C. D. Carothers, R. B. Ross, and P. Carns. 2017. Enabling Parallel Simulation of Large-Scale HPC Network Systems. *IEEE Transactions on Parallel and Distributed Systems* 28, 1 (Jan 2017), 87–100. https://doi.org/10.1109/TPDS.2016.2543725

[25] nvprof 2019. *Profiler :: CUDA Toolkit Documentation*. Retrieved April 28, 2020 from https://docs.nvidia.com/cuda/profiler-users-guide/index.html

[26] Top500 2019. *November 2019 | TOP500 Supercomputer Sites*. Retrieved April 28, 2020 from https://www.top500.org/lists/2019/11/

[27] Xingfu Wu and Valerie Taylor. 2013. Performance Modeling of Hybrid MPI/OpenMP Scientific Applications on Large-scale Multicore Supercomputers. *J. Comput. Syst. Sci.* 79, 8 (Dec. 2013), 1256–1268. https://doi.org/10.1016/j.jcss.2013.02.005

[28] G. Zheng, Gunavardhan Kakulapati, and L. V. Kale. 2004. BigSim: a parallel simulator for performance prediction of extremely large parallel machines. In *18th International Parallel and Distributed Processing Symposium, 2004. Proceedings.* 78–. https://doi.org/10.1109/IPDPS.2004.1303013