SAND2024-02557C

CMB: A Configurable Messaging Benchmark to Explore Fine-Grained Communication

W. Pepper Marts*†, Donald A. Kruse*, Matthew G. F. Dosanjh*,
Whit Schonbein*, Scott Levy*,
*Center for Computing Research
Sandia National Laboratories
Albuquerque, New Mexico, USA
{wmarts,dkruse,mdosanj,wwschon,sllevy}@sandia.gov
Patrick G. Bridges†
†Department of Computer Science
University of New Mexico
Albuquerque, New Mexico, USA
patrickb@unm.edu

***Abstract*—Modern communication APIs provide increased ability to specify when, where, and how to send data between processes. One recent innovation is fine-grained communication, where processes are able to send subsets of data as it is ready rather than waiting for the entirety of the data to be completed. Allowing data to be sent when it is ready increases opportunities for overlapping communication and computation. However, with multiple fine-grained, thread-safe interfaces, the task of optimizing an application's peer-to-peer fine-grained communication is complex. In this paper, we present the Configurable Messaging Benchmark (CMB), a tool for evaluating the application impact of fine-grained communication. Using the CMB we perform a case study to measure the impact of different fine-grained implementations on a variety of realistic application profiles. Initial results reveal a large optimization space ranging from potential speedups as high as 52.97% to slowdowns as high as 289.55% relative to bulk-synchronous MPI message passing.**

***Index Terms*—high-performance computing , computer networks , fine-grained communication , models , benchmarks**

I. Introduction

Scientific applications typically follow a Bulk Synchronous Parallel (BSP) model, where the application is split into distinct compute and communication phases, and all threads must synchronize between phases. *Fine-grained communication* aims to improve application performance by sending pieces of the data as those pieces are completed, rather than waiting for a distinct communication phase. This approach leverages network resources during the computation phase where a BSP application would otherwise leave those resources idle. Because of this potential, fine-grained communication is an active topic in the high-performance computing (HPC) research community [1]–[3].

The performance of fine-grained communication depends on several factors. BSP applications are mainly impacted by size of communicated data, communication interface (e.g., MPI message passing or RMA), number of peers, and system performance. Additional factors impact fine-grained communication, such as the size and number of pieces being sent, additional overheads of the communication interface (locking, matching, etc.), and the distribution of thread completions. The combination of all of these factors creates a daunting optimization space for applications. Choosing the right number and size of the pieces being sent and which communication interface to use is a complex problem that can vary between applications, systems, and software versions.

To address this challenge, we present the Configurable Messaging Benchmark (CMB), a tool to measure the application impact of different fine-grained communication strategies. Based on the MiniMod framework [4], the CMB allows the user to define an application profile (compute time, number of peers, total communicated volume, and thread arrival distribution) and a fine-grained communication approach (number of pieces to divide message buffers into and communication interface). The benchmark evaluates the impact of the communication approach on the application profile, providing valuable feedback regarding expected application behavior.

This paper makes the following contributions:

- The Configurable Messaging Benchmark (CMB): an open-source[1] application performance benchmark for evaluating fine-grained communication for different application profiles;
- Extensions to MiniMod enabling the exploration of different message aggregation techniques;
- A case study exploring the impact of fine-grained communication on 7- and 27-point stencils using 5 different application noise profiles (Laggard Thread, Normal, and 3 distributions from measured application behavior).

The remainder of this paper is structured as follows. We provide a background for our paper in Section II. We detail the design of the CMB in Section III. We describe how we conducted our experiments and present the results of our experiments in Section IV. We discuss the implications of the case study in Section V. Finally, we distinguish our work from existing related work in Section VI and present our conclusions in Section VII.

*Sandia National Laboratories is a multimission laboratory managed and operated by National Technology & Engineering Solutions of Sandia, LLC, a wholly owned subsidiary of Honeywell International Inc., for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA0003525.

[1]We are in the process of getting the CMB approved for open source release, and anticipate replacing this text with a link to the source code before publication.

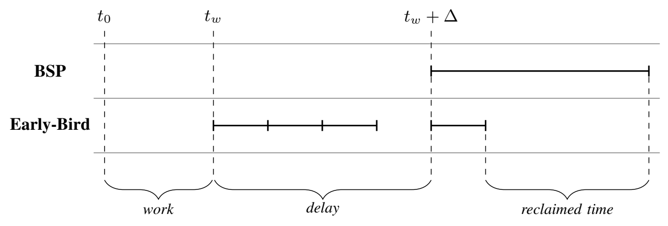


Fig. 1: Traditional BSP communication (top) and best-case early-bird communication (bottom) for a scenario where the buffer is divided into four parts. See main text for more information.

II. Background and Motivation

A. Bulk Synchronous and Fine-Grained Communication

In traditional Bulk Synchronous Parallel (BSP) applications, communication and computation are separated into distinct phases, where data is exchanged only after all current work is completed and before any new work begins. This includes situations where work is multi-threaded: Even if some threads complete their contributions earlier than others, the results are not communicated until all threads have completed. Unbalanced workloads or operating system noise can lead to lagging threads [1], [5], [6], delaying the transmission of data that is otherwise ready to send. This scenario is depicted in Figure 1 (top): Work begins at t_0 and ideally would complete at t_w. However, in this case, OS noise delays the completion of one thread, so bulk communication cannot begin until $t_w + \Delta$.

A *fine-grained communication* strategy attempts to leverage these opportunities by allowing each thread to initiate data transfer when their portion of work is completed, independently of others [1]–[3]. A potential benefit of fine-grained communication is *early-bird* transmission, shown in Figure 1 (bottom): While one thread has not yet completed its contribution to the buffer as of t_w, the other three have completed theirs, so the data transfer can begin earlier than in the BSP case. The figure shows a best-case scenario, where the delay is sufficiently large that the contributions from all but one of the threads can be moved before the last thread completes, maximizing the amount of reclaimed time. Even if an application already overlaps communication and computation by other means, fine-grained communication can still expose additional overlap.

B. Factors Affecting Fine-Grained Communication

The efficacy of fine-grained communication depends on both application behaviors as well as the underlying communication stack. These factors interact, making the task of optimizing fine-grained communication challenging. In this subsection, we survey some of these factors, emphasizing those that are exposed to middleware and scientific application developers.

1) Buffer Size: The communication overheads incurred by splitting a buffer into multiple messages are proportionally greater for small buffer sizes than for large buffers. This is due to overheads being roughly constant in regards to buffer size while the time spent transferring data scales linearly. We expect fine-grained communication to be less performant for small buffers. However, larger messages require more time to inject into the network, so best-case scenarios (cf. Figure 1 (bottom)) may require a longer delay to be realized.

2) Number of Peers: Splitting a buffer into multiple messages increases the number messages to be sent, and this number is further increased by the number of peers. For example, a 7-point halo exchange with buffers divided into four sub-messages yields 24 individual messages, while a 27-point exchange requires 104. This rapid escalation in the number of messages may interact with communication middleware, e.g., by exhausting bounce buffers, complicating message matching, or requiring many more rendezvous protocol transactions than if the buffers were not split.

3) Actor Completion Distributions: In Figure 1 (bottom) we illustrate the potential benefits of fine-grained communication by assuming every actor but one completes at time t_w, with one thread lagging behind until time $t_w + \Delta$. Of course, this *Lagging Thread* scenario is only one way actor completions might be distributed over time. For instance, while laggards do exist, threads sometimes complete in accordance with a normal distribution with a mean centered at t_w, or follow more complex distributions [7]. The distribution of actor completion times affects available opportunities for early-bird communication. Completions following a normal distribution with a small standard deviation, for example, have (on average) less time to exploit than those that involve a large standard deviation.

4) User Partitions and Transport Partitions: Complementary to the importance of buffer size is the number of partitions. Each additional partition adds send and receive overheads [8], which could swamp any potential benefits of fine-grained communication, especially for smaller buffer sizes. For the remainder of this paper we will follow the terminology used by Temuçin et al. in their work [9]. A *user partition* refers to the granularity of communication requested by a user application, i.e. the contribution of an individual thread to a communicated buffer. A *transport partition* refers to the granularity used to communicate over the network, i.e. the size of messages when using MPI two-sided message passing or the size of an individual put when using MPI one-sided RMA. These may be of different sizes; a transport partition is composed of one or more user partitions. A communication strategy that sends fewer user partitions per transport partition might be described as *more fine-grained* than one that sends more.

5) Communication interface: The number of communication middleware interfaces continues to grow; from MPI's point-to-point, non-blocking, persistent, RMA [10], and Partitioned Communication [1] to OpenSHMEM, PGAS, and others. Each of these can exhibit different performance behaviors with the biggest impact being how well an interface maps to the underlying RDMA semantics.

C. The Need For a Better Benchmark

As illustrated in Section II-B, extracting performance benefits from early-bird communication presents a difficult challenge because of the sheer number of relevant factors and the ways in which they interact. There is likely no 'one right answer', applicable to all applications across all systems, regarding whether to utilize fine-grained communication, and if so, how fine-grained the communication should be, which communication interface should be used, and so forth. In the next section, we introduce the CMB, a benchmark that allows users to investigate fine-grained communication performance while varying application and communication factors.

III. Design of the CMB

The Configurable Messaging Benchmark (CMB) is an application impact benchmark built using the MiniMod framework [4] that explores the performance of different methods of implementing fine-grained communication across different application profiles. In this section, we present the CMB and discuss its design philosophy, general architecture, extensions made to the MiniMod framework, and how it handles thread arrival times.

A. Design Principles

There are a few key design principles that are integral to the CMB. First, the benchmark needs to be able to assess the performance impact of fine-grained communication on applications. There are three types of sources of impact on the performance of the benchmark; system, application profile, and fine-grained implementation.

System comprises factors that are externally determined. For example CPU, network, and MPI distribution all fit in this category. While these are not explicitly referenced inside the CMB, these factors play a significant role in the performance of an application.

Application profiles are a set of independent variables that specify how the application the CMB is mimicking will behave. These variables include compute time, communication size, number of threads, thread completion distribution, and number of peers. Each combination of these parameters creates a profile of a different hypothetical application.

Fine-grained implementations are another set of independent variables that represent the decision-space within an application on how to implement fine-grained communication. These include the number of user partitions per transport partition in each direction in a halo exchange and the communication methodology (e.g., MPI message passing vs. MPI RMA). Each combination of these parameters represents a different way the middleware stack could accomplish the communication requirements described by the application.

It is important to distinguish between the characteristics that should be directly comparable and those that are not. Benchmark results across different application profiles are likely not directly comparable performance wise, in the same way the performance of different applications are not. These are meant to represent the requirements of different scientific computations. Conversely, results across the fine-grained implementations are directly comparable, as none of these factors represent the underlying needs of the scientific application but rather the implementation decisions being made as means to those ends.

The second design principle is to use a combination of estimated and data-driven application profiles. A direct example of this is thread arrival times. Previous work has relied on assumptions about how threads behave, e.g., assuming OS noise will delay a thread or assuming a normal distribution. By measuring the thread behaviors of applications and creating distributions based on empirical data, the CMB will be able to better evaluate real-world impact of fine-grained communication on application performance.

B. The CMB Architecture

Taking these principles as a basis, we designed and built the CMB to emulate application profiles (compute time, communication volume, number of peers, thread counts, and thread arrival distributions) and evaluate different approaches to fine-grained communication. The CMB was created to allow users to empirically explore the factors described in Section II-B. The CMB allows user to explore the impact of fine-grained communication not just on individual applications but across entire swaths of application classes.

The CMB draws upon concepts from two previous works. The first is the RMA-MT version of the Sandia Microbenchmark message rate benchmark [11], which emulates the network behavior of applications by using the number of peers to create a ring where each process communicates to a given number of peers. Other adaptations include clearing the cache between iterations and making communication calls from different threads. The second is the original partitioned communication paper [1] which introduced a bandwidth benchmark that emulated computation and system noise by using sleep to pause thread execution and extending the sleep time for an impacted thread to represent system noise and load imbalance. We leveraged these concepts in the CMB, which was designed to emulate application behavior in depth including these factors.

The CMB takes as inputs the number of peers that each process communicates with each iteration (stencil size), the amount of data sent to each peer (buffer size), the number of threads that contribute to each buffer (thread count), and a number of parameters to define the times at which those threads complete their contributions as described in III-D. To manage different fine-grained implementations we built the benchmark using the MiniMod framework [4], which allows for runtime changes of fine-grain communication and communication interface.

C. Extensions to MiniMod

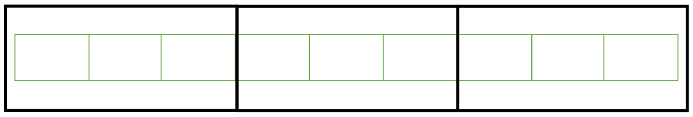


Fig. 2: Although an application may define user partitions of any size (demonstrated by the green boxes), the communication middleware may determine that it is instead optimal to aggregate user partitions into bins and send transport partitions of a coarser granularity (demonstrated by the black boxes).

Fine-grained communication operates at the level of individual threads: when a thread completes a user partition (contributed portion of a communicated buffer by single thread), that partition can be sent immediately (see Section II). However, this level of granularity may not be optimal. For instance, if we send each user partition individually a small buffer with many contributors will be sent as many even smaller messages, potentially incurring large overheads. In situations like this, it may be advantageous to allow communication middleware to make decisions regarding the size and number of messages to send over the network. By aggregating user partitions into a smaller number of larger transport partitions,

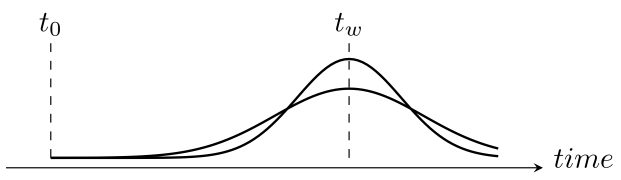


Fig. 3: Abstract illustration of the normal distribution completion pattern. Completion times are sampled from the normal distribution, where the mean is held fixed at t_w while standard deviation is varied.

it is possible to avoid overheads while still taking advantage of opportunities afforded by fine-grained communication.

To capture this functionality, we extended MiniMod by adding an *aggregation* module. The aggregation module allows the framework to aggregate a number of multiple contiguous user partitions into larger transport partitions. The module has some constraints. When using MPI message passing, the size of each message needs to be pre-determined (`MPI_Isend` must match a corresponding `MPI_Irecv`), and also needs to avoid adding overhead through an extra data copy. Without the use of an additional round trip message to prepare the receiving process, both processes must decide on a strategy in advance. Our implementation is a simple aggregation strategy shown in Figure 2: The send buffer is split into a number of contiguous *bins* and a bin is sent only after all of the user partitions within it have completed. While more advanced aggregation strategies may be required to resolve data dependencies in actual applications, this approach allows us to explore the performance impact of aggregation in the CMB.

D. Thread Arrival Times

The potential benefits of fine-grained communication are largely determined by the times at which user partitions are completed by threads. In order to allow the greatest control over these times, the CMB emulates computation by suspending the execution of threads for the duration of their emulated computation. Depending on runtime configuration, each thread independently determines their thread arrival time and then sleeps for that long. The CMB can emulate the thread arrival times of applications with an individual laggard thread (Figure 1), applications where threads complete according to a normal distribution (Figure 3), or by using kernel density estimation (KDE) can model arbitrary applications based on measurements of thread completion times obtained from real applications [7].

Kernel density estimation is a non-parametric method for approximating the probability density function of a distribution based on a collection of samples. Using this approximation we can generate new samples that will closely match the distribution of the input samples. This allows us to replicate the thread arrival distribution of a target application and study its amenability to fine-grained communication without needing to rewrite the entire application to use fine-grained communication. We generated thread arrival timings using kernel density estimates of the data collected by Marts et. al. [7] for three mini-applications: MiniFE [12], MiniMD [13], and MiniQMC [14], [15]. As shown in Figure 4, real-world thread timings can be more-or-less Gaussian, but also skewed or even multimodal.

E. CMB Execution

Every iteration, the CMB begins by generating for each thread an arrival time (time the thread completes its work in a parallel section) in nanoseconds as described in section III-D. For example, if the CMB was configured to use KDE for determining thread arrivals, it would generate a sample from that KDE for each of its threads at the beginning of every iteration. The CMB then begins the timed section, an OpenMP parallel region. Each thread in this region begins by sleeping for the amount amount of time determined earlier, and then calling into MiniMod to communicate that thread's user partitions. Minimod will then communicate these partitions as described in section III-C to a number of peers as described in section III-B. The timed section ends after all threads have joined and thus the communication has finished. Reported times are summation of the times measured in each iteration.

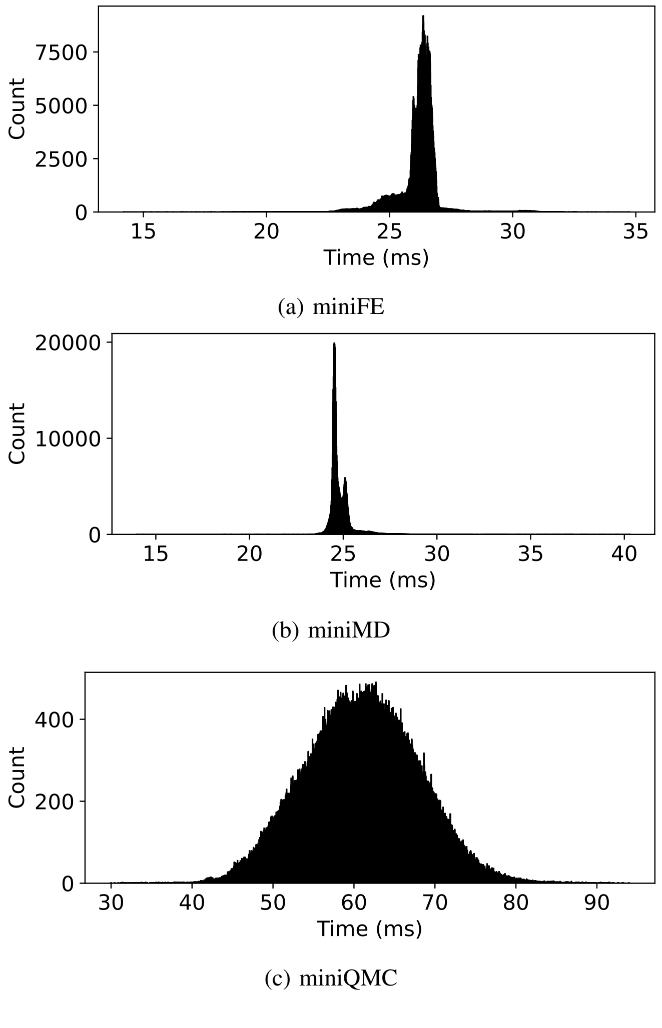


Fig. 4: Histograms of the three application thread arrival distributions used to generate our kernel density estimates. Each histogram has a bin width of $10\mu s$.

IV. Results

In this section we detail the results of our case study using the CMB to explore the impact of different fine-grained implementations across a variety of application profiles.

A. Setup

1) Hardware and Software Stack: Data was collected on two machines: Sandia National Laboratories' Manzano and Mutrino clusters. Manzano, a CTS-1 system, has two 2.9 GHz Intel Cascade Lake CPUs and 192 GB of RAM per node. The machine uses the RHEL7 operating system and runs on an Intel Omni-Path network. Data collected on this system used OpenMPI 4.1.3 and all executables were compiled with GCC version 10.2.1. Mutrino, a Cray XC40 system that has the same hardware configuration as NERSC's Cori [16] and LANL's Trinity [17] super computers, has two 2.3 GHz Intel Haswell CPUs and 128 GB of RAM per node. The machine uses the SLES11 operating system and runs on a Cray Aries Dragonfly network. Data collected on this system was run on the vendor-provided Cray software stack with Cray clang version 11.0.2 and Cray MPICH version 7.7.16.

2) Experiments: Data was generated via the use of the CMB, running across a variety of systems, application profiles, and fine-grained implementations as described in Section II. Runs on all systems used 32 nodes with one process per node. Each data point is the mean of five trials. Each trial is a run-time summing the time spent in its 200 application iterations. The cache is cleared before each iteration. Percent speedup is the percent difference in run-time between a fine-grained implementation and an equivalent run with MPI message passing and a BSP execution model. This is calculated as $((BSP - FINE)/(BSP)) * 100$.

Our fine-grained implementations vary in communication interface and granularity of communication. For communication interface we tested MPI RMA and MPI message passing. For communication granularity we tested BSP, sending user partitions individually, and simple aggregation. On Manzano we used 32 threads and transport partition counts between 2 and 16 by powers of two. On Mutrino we used 16 threads and transport partition counts between 2 and 8 by powers of two.

Our application profiles vary in buffer size, computation time, thread arrival distribution, and stencil size. We varied buffer size (the size of of the communicated buffer between each communicating process pair) from 256 bytes to 4 mebibytes by every other power of two. We kept computation time fixed at 2^{22} nanoseconds (approximately 67.1 ms) for application profiles that do not use a kernel density estimate. This is the time in nanoseconds that each non-laggard thread sleeps per iteration for our laggard thread results and the mean value for our normal distribution (See t_w in Fig. 1 and Fig. 3). We used the values 1%, 4%, and 10% as the noise parameter for our laggard thread distribution (the percentage by which the laggard thread is slower than the rest). These numbers were selected to match Grant et al.'s previous work [1]. For our normal results we used standard deviations of 200ns, 2000ns, and 20000ns. We used thread timings sampled from KDE's based on the timing data from MiniFE, MiniMD, and MiniQMC. We ran each configuration with both a seven point and 27 point stencil.

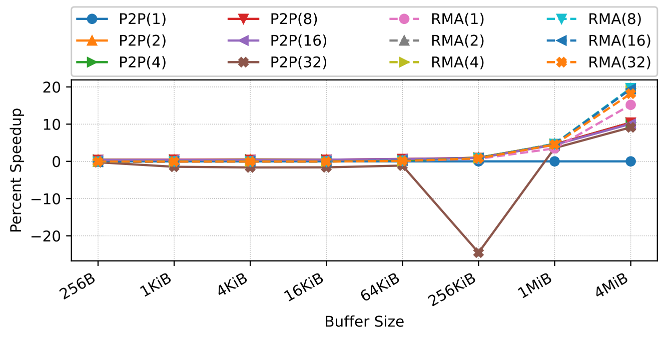
Due to the number of tunable parameters exposed by the CMB we limit our search space. This case study was chosen as a reasonable balance of breadth of exploration, sample resolution, and verisimilitude of application behavior. Not every data point will necessarily correspond to a reasonable application profile and fine-grained implementation combination. But by including such corner cases, the CMB reveals broader trends in application behavior.

B. Laggard Thread Arrival

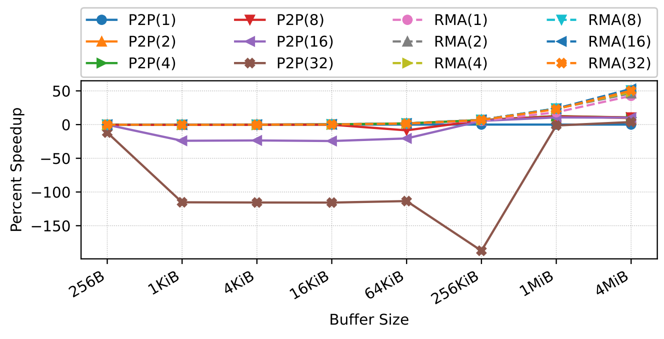
Fig. 5 presents results for application profiles with a laggard thread arrival distribution, i.e., when one thread is delayed for some percentage (1%, 4%, or 10%) of work time. On Mutrino, the CMB does not show performance benefits from fine-grained implementations when using MPI message passing as a communication interface. With a seven-point stencil, iteration times were within 0.310% of BSP MPI message passing for buffer sizes below 1MiB. For buffers 1MiB and larger they were slower by between 1.105% and 7.428%. The corresponding application profile with a 27-point stencil resulted in iteration times that were slower than BSP MPI message passing by between 0.018% and 1.606% for buffer sizes below 1MiB and between 1.227% and 16.579% slower for buffers 1MiB and larger. The only application profiles where itteration times were lower than BSP MPI message passing was for 16KiB buffers. We observed this difference for all laggard noise parameters. No difference exceeded 0.061% for a 7-point stencil or 0.074% for a 27-point stencil.

Using MPI RMA for the same application profile resulted in performance gains on Mutrino, but only for large buffer sizes. Application profiles with a 7-point stencil performed within 0.566% of BSP MPI message passing for buffer sizes below 256KiB. Application profiles with buffers of size 256KiB and larger saw performance benefits between 0.096% and 7.395%. Application profiles with a 27-point stencil performed within 1.817% of BSP MPI message passing for buffer sizes below 256KiB. Application profiles with buffers of size 256KiB and larger saw performance benefits between 1.465% and 8.955%.

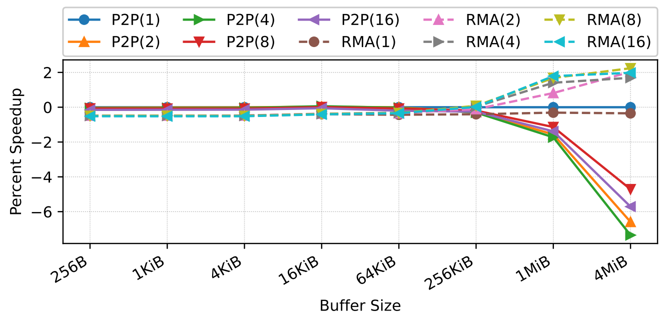
On Manzano the CMB shows consistent opportunity for benefit from fine-grained communication for application profiles with a laggard thread arrival distribution. When using fine-grained implementations with MPI message passing there was no combination of buffer size, stencil size, and thread delay where there was not performance improvement when using 2 transport partitions. For application profiles with a 7-point stencil iteration times were faster by 0.412% to 10.453% for transport partition counts between 2 and 16 with our maximal speedups seen between 2 and 8 transport partitions. For runs with 32 transport partitions and buffer sizes below 1MiB iteration times were slower by 0.245% to 24.195% and for buffers 1MiB and larger iterations were faster by 3.579% to 9.187%. Although the speedups are less consistent for a many-then-one laggard thread arrival distribution with a 27-point stencil, the application profiles with two transport partitions were able to see some benefit. There were two application profiles that were not able to achieve a speed up for any transport partition count, but both had relative slowdowns below 0.001%. Otherwise, iteration times for 27-point stencils were faster by 0.271% to 12.560% when using 2 transport partitions. The fastest iteration times were always observed when using two transport partitions, and the slowest iteration times when using 16 or 32 transport



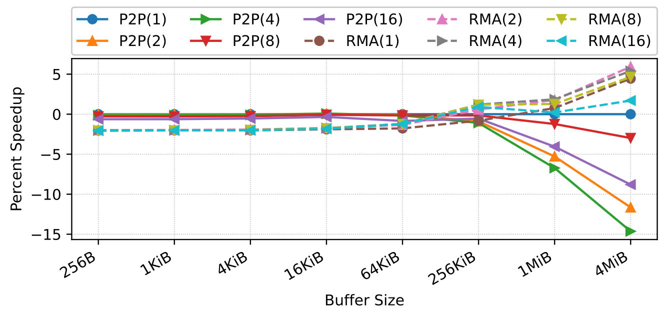
(a) Manzano: 7-Point Stencil



(b) Manzano: 27-Point Stencil



(c) Mutrino: 7-Point Stencil



(d) Mutrino: 27-Point Stencil

Fig. 5: Percent speedup for fine-grained communication relative to BSP MPI message passing for application profiles with laggard thread arrival distributions. Each line corresponds to a fine-grained implementation. "P2P" refers to two-sided message matching. "RMA" refers to one-sided MPI RMA. Numbers in parentheses are the number of transport partitions. Results shown use a 4% laggard delay parameter.

partitions. For buffer sizes below 1MiB there was a maximum slowdown of 8.887% to a maximum speedup of 6.674% using 4 or 8 transport partitions, and maximum slowdown of 187.155% to a maximum speedup of 5.484% for 16 or 32 transport partitions. For buffers 1MiB and larger there was a minimum speedup of 9.267% to a maximum speedup of 12.569% using 4 or 8 transport partitions, and a maximum slowdown of 19.315% to a maximum speedup of 10.901% for 16 or 32 transport partitions.

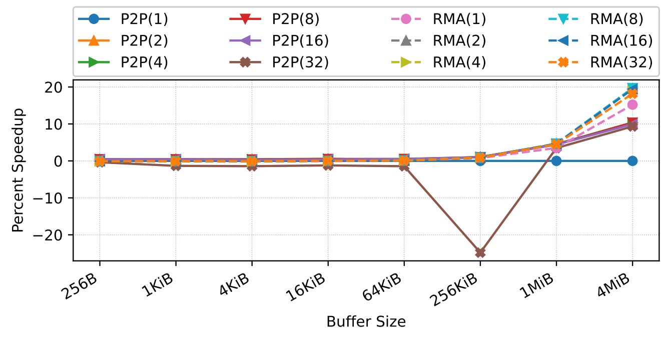
Using MPI RMA for application profiles with laggard threads arrivals on Manzano proved similar to Mutrino. The key difference was that Manzano saw much greater reduction in iteration times. Application profiles with 7-point stencil performed within 0.225% of BSP MPI message passing for buffer sizes below 256KiB. Application profiles with buffers of size 256KiB and larger saw performance benefits between 0.969% and 19.770%. Application profiles with a 27-point stencil performed within 0.3507% of BSP MPI message passing for buffer sizes below 64KiB. Application profiles with buffers of size 64KiB and larger saw performance benefits between 1.713% and 52.972%.

C. Normal Thread Arrival

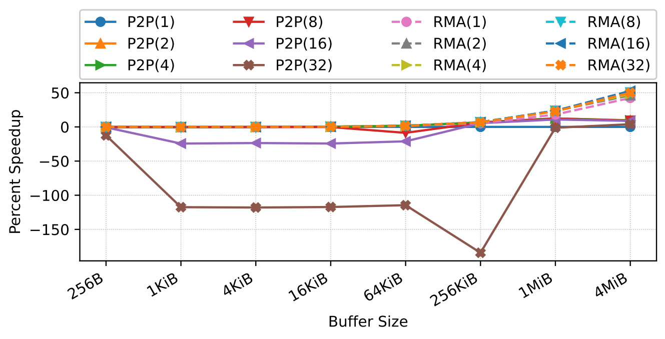
Fig. 6 presents case study results for application profiles with a normal thread arrival distribution. On Mutrino these application profiles saw lower relative iteration times than analogous application profiles with laggard thread arrivals for fine-grained implementations using MPI message passing. A seven point stencil resulted in iteration times that were slower than BSP MPI message passing by between 0.032% and 0.825% for buffer sizes below 1MiB and between 1.178% and 7.391% slower for buffers 1MiB and larger. Application profiles with a 27-point stencil resulted in iteration times that were slower than BSP MPI message passing by between 0.200% and 3.473% for buffer sizes below 1MiB and between 5.665% and 22.112% slower for buffers 1MiB and larger.

When using MPI RMA for the same application profiles we also saw no performance gain from fine-grained communication. The relative performance difference varied little across tested application profiles. Application profiles with a 7-point stencil saw slowdowns of between 0.416% and 1.843%. Application profiles with a 27-point stencil saw slowdowns of between 1.445% and 4.775% for buffers sizes 1MiB and below. Application profiles with buffers of size 4MiB varied between a slowdown of 3.618% and a speedup of 2.855%

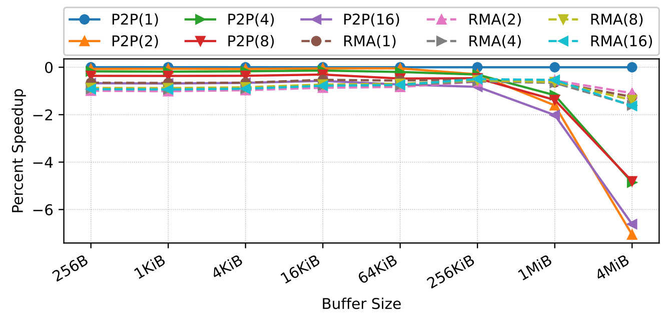
When using MPI message passing for normally-distributed thread arrival times on Manzano results were very similar to the laggard results. Iteration times for 7-point stencils were faster by 0.271% to 10.397% for transport partition counts between 2 and 16 with our maximal speedups seen between 2 and 8 transport partitions. For runs with 32 transport partitions and buffer sizes below 1MiB iteration times were slower by 0.367% to 25.085% and for buffers 1MiB and larger iterations were faster by 3.174% to 9.332%. There were three application profiles



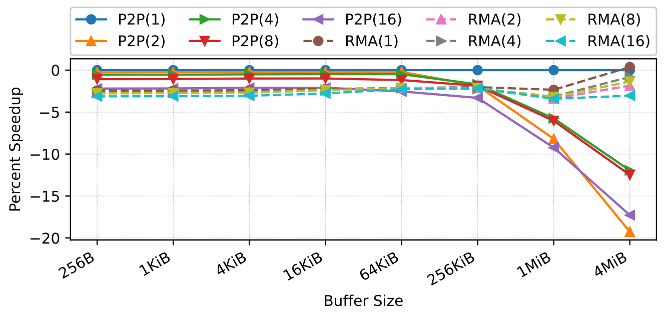
(a) Manzano: 7-Point Stencil



(b) Manzano: 27-Point Stencil



(c) Mutrino: 7-Point Stencil



(d) Mutrino: 27-Point Stencil

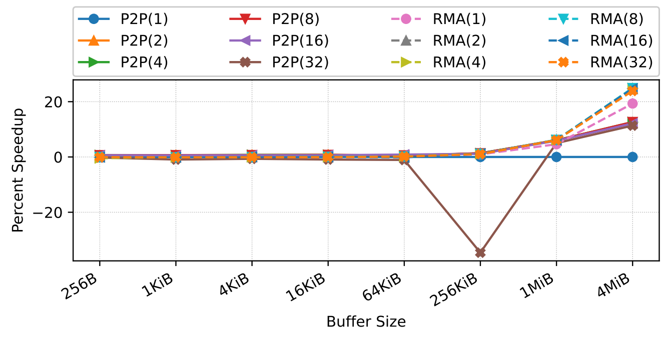
Fig. 6: Percent speedup for fine-grained communication relative to BSP MPI message passing for application profiles with normal thread arrival distributions. Each line corresponds to a fine-grained implementation. ”P2P” refers to two-sided message matching. ”RMA” refers to one-sided MPI RMA. Numbers in parentheses are the number of transport partitions. Results shown use a $2\mu s$ standard deviation.

with normally distributed thread arrivals and 27-point stencils that were not able to achieve a speed up for any transport partition count, but all three had relative slowdowns below 0.001%. Otherwise, iteration times for 27-point stencils were faster by 0.021% to 12.552% when using 2 transport partitions. For buffer sizes below 1MiB there was a maximum slowdown of 8.780% to a maximum speedup of 6.615% using 4 or 8 transport partitions, and maximum slowdown of 186.212% to a maximum speedup of 5.498% for 16 or 32 transport partitions. For buffers 1MiB and larger there was a minimum speedup of 9.226% to a maximum speedup of 12.556% using 4 or 8 transport partitions, and a maximum slowdown of 3.296% to a maximum speedup of 10.744% for 16 or 32 transport partitions.

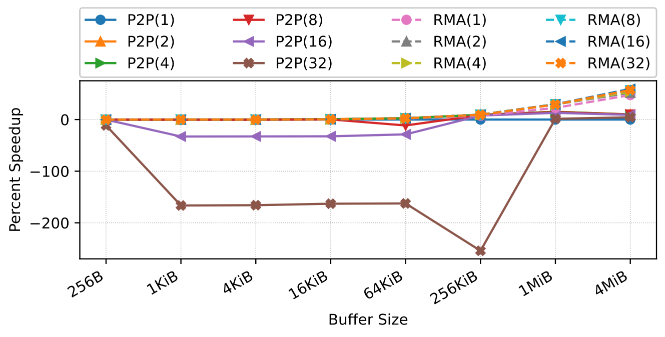
Using fine-grained implementations MPI RMA for application profiles with normally distributed thread arrival times on Manzano resulted in relative iteration time differences very similar to those seen with laggard on the same system. Application profiles with 7-point stencils using MPI RMA in their fine-grained implementations performed within 0.223% of BSP MPI message passing for buffer sizes below 256KiB. Application profiles with buffers of size 256KiB and larger saw performance benefits between 0.925% and 19.719%. Application profiles with a 27-point stencil performed within 0.33% of BSP MPI message passing for buffer sizes below 64KiB. Application profiles with buffers of size 64KiB and larger saw performance benefits between 1.730% and 52.760%.

D. Kernel Density Estimate

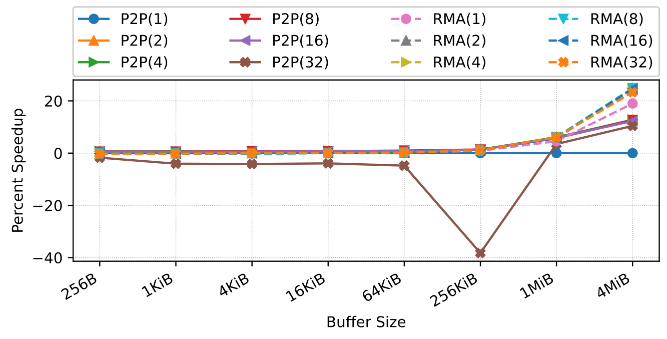
Fig. 7 presents case study results for application profiles with thread arrival distributions generated using kernel density estimates. These application profiles were all timed on Manzano, the same system where the thread timing data for our KDEs was collected. Like other application profiles tested on Manzano, the CMB shows consistent opportunity for benefit from fine-grained communication provided we do not split our buffers into too many messages. The CMB shows that when using MPI message passing with between 2 and 16 messages per buffer there was a relative performance difference of between 0.160% to 0.991% for application profiles with a 7-point stencil and buffer sizes of 64KiB and below and a speed up of between 1.008% to 12.794% for application profiles with a 7-point stencil and buffer sizes of 256KiB and above. When using 32 messages, the CMB showed that for the same 7-point stencil application profiles there was a relative iteration time difference of between -4.160% to 0.169% for application profiles with buffer sizes of 64KiB and below, a slowdown of between 27.370% to 38.225% for application profiles with a buffer size of 256KiB, and a speedup of between 3.478% to 11.347% for application profiles with buffer sizes of 1MiB and above. When using MPI RMA based fine-grained implementations for application profiles with a 7-point stencil



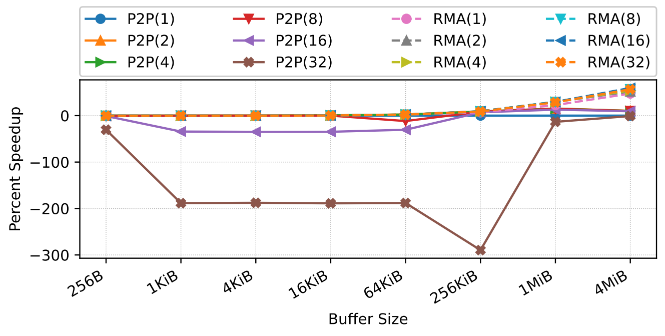
(a) miniFE: 7-Point Stencil



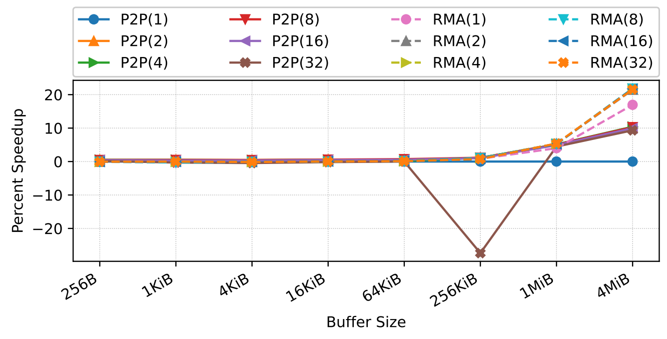
(b) miniFE: 27-Point Stencil



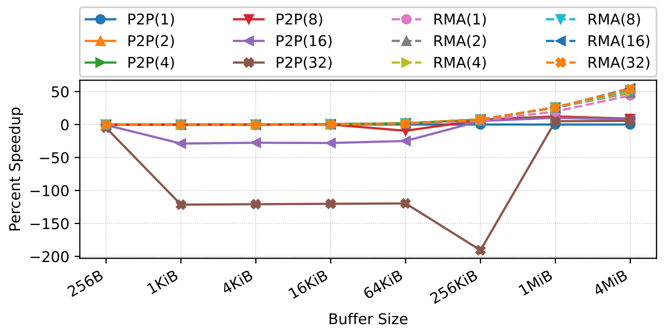
(c) miniMD: 7-Point Stencil



(d) miniMD: 27-Point Stencil



(e) miniQMC: 7-Point Stencil



(f) miniQMC: 27-Point Stencil

Fig. 7: Percent speedup for fine-grained communication relative to BSP MPI message passing for application profiles with thread arrival distributions generated with a KDE. Each line corresponds to a fine-grained implementation. ”P2P” refers to two-sided message matching. ”RMA” refers to one-sided MPI RMA. Numbers in parentheses are the number of transport partitions. Thread arrival times randomly generated from corresponding kernel density estimate

iteration times remained within -0.479% to 0.247% of BSP MPI message passing for application profiles with buffer sizes of 64KiB and below and a speed up of between 0.741% to 24.911% for application profiles with buffer sizes of 256KiB and above.

For application profiles with 27-point stencils the CMB shows similar trends to other 27-point application configurations on Manzano. When using MPI message passing with between 2 and 8 messages per buffer iteration times remained within -0.724% to 0.782% of BSP MPI message passing for application profiles with buffer sizes of 16KiB and below, within -11.683% to 2.695% of BSP MPI message passing for application profiles with a buffer size of 64KiB, and a speed up of between 6.897% to 15.281% for application profiles with buffer sizes of 256KiB and above. When using between 16 to 32 messages with the same application profiles there were slowdowns of between 0.129% to 289.552% for application profiles with buffer sizes of 256KiB and below and within -13.4047% to 12.987% of BSP MPI message passing for application profiles with a buffer size of 1MiB and above. When using MPI RMA for application profiles with a 27-point stencil iteration times remained within -0.479% to 0.471% of BSP MPI message passing for application profiles with buffer sizes of 64KiB and below and a speed up of between 1.604% to 59.509% for application profiles with buffer sizes of 256KiB and above.

V. Discussion

The results reported by the CMB and described above above suggest several broad conclusions and heuristics regarding the use of fine-grained communication:

1) Necessity of Empirical Analysis: Our evaluation demonstrates that the same application profile (combination of thread arrival distribution, communication stencil, and volume communicated between peer processes) using the same fine-grained implementation (combination of communication interface and number of transport partitions) can exhibit different behaviors depending on the system used. For example, some techniques that prove consistently beneficial on Manzano result in slowdowns on Mutrino. The factors that govern performance are complex and hard to disentangle when their impacts are so interdependent. Given this, a tool like the CMB can be invaluable in making design decisions.

2) Avoiding Pitfalls: Figures 5, 6, and 7 show the potential hazard of poor fine-grained implementations. Although benefits are possible, these results show that some configurations perform radically worse than bulk-synchronous two-sided message passing. Conveniently, these configurations are easy to avoid. We see that the for both systems the worst performing configurations are those with a large number of transport partitions coupled with two-sided MPI message passing. Although this configuration may start communication earlier, the message and matching overheads are considerable.

3) Safe Configurations: There are generally safe configurations when performing fine-grained communication. MPI two-sided message passing using a small number of messages between pairs (two or four transport partitions) do not exhibit the performance pitfalls of large message counts while still benefiting from the overlap of communication and computation. For Manzano it was consistently beneficial to aggregate and send two messages, with speedups as high as 12.79%. On Mutrino, the same configurations generally had slowdowns of less than 1%. This is especially true for application profiles that communicate buffers smaller that 1 MiB. Although Manzano has its largest speedups for application profiles that send these especially large buffers, Mutrino has its largest slowdowns. Existing modeling work predicts that our Mutrino results are atypical [18], [19], but it does limit our recommendation for applications sending buffers of this size.

4) One-Sided Configurations: Configurations that used MPI RMA as the communication interface proved to be very prudent. For buffers smaller than 1 MiB they did not see the same consistent benefits on Manzano that were observed using MPI two-sided message passing. Instead they generally performed within 1% of baseline with a worst case slowdown of less than 4.775%. What they lose in best case performance for small messages they make up for by avoiding the sensitivity to message aggregation and in their speed ups for large buffers. One-sided communication determines a consistent target buffer at window creation and performs explicit epoch synchronization, dramatically reducing the per-send overhead of communication. The biggest benefits to configurations with MPI RMA are found for large messages. When communicating buffers that are 1 MiB or larger, RMA dramatically outperforms MPI two-sided message passing regardless of transport partition count. These are remarkable speedups with reductions in iteration time as high as 52.972% compared to baseline. Although there are additional complexities and restraints to using MPI RMA, results derived from the CMB suggest it is a very promising interface for threaded communication.

VI. Related Work

Determining the right granularity for concurrent computations is a challenging task. Nonetheless it is required for future generation codes more than any previous generation due to the migration to accelerators and multi-threaded environments. Previous surveys have shown that [20] there is great desire to explore different concurrency methods but that most applications have not yet moved to new methods. Part of this reluctance is the difficult in determining how fine-grained a concurrency method must be and how to communicate and partition work. Using non-mainstream mechanisms is a significant amount of work, as evidenced by past efforts not aided by methods like MiniMod [21]. MiniMod first provided the ability to examine different communication middleware solutions [4], helping application code teams decide which middleware to use, but it did not resolve the problem of the granularity of concurrency question. This work provides the solution for studying the answer to this question for code teams.

Proxy applications have been well studied, built and used in the past. Major proxy application suites like Mantevo [22] have offered several versions of their proxy applications that allow comparisons between different middleware types and threading models. Other single proxy applications like LULESH [23] have many modified versions as well. Unfortunately, for each communication subsystem (e.g., MPI, OpenSHMEM, RDMA), granularity (single thread per process, multiple threads, message granularity, etc), and threading library a separate proxy app needs to be written.

Frameworks for performance portability between hardware architecture have been developed and are in use such as Kokkos [24] and RAJA [25]. These approaches solve some difficulties in having to recreate miniapps for new system architectures, but of course are not easy to compare portability layers to using underlying hardware specific approaches. In contrast, the MiniMod modular miniapp approach allows for a single code modification point that can be used to compare many different communication subsystems and parallelism granularity.

There have been many prior works evaluating communication middleware and comparing approaches. MPI has been extensively studied in traditional modes [26], [27] as well as one-sided [10], [11]. MPI multi-threading is currently a hot topic and has been extensively explored recently [28]–[33]. Many MPI comparisons are limited to studying MPI libraries themselves by necessity due to comparisons of new features or proposed features [1], [34]–[37]. However, such new approaches would be desirable to compare broadly across many different communication subsystems which this work enables and makes significantly less burdensome. It also solves questions relating to solutions that have variable granularity built into them, like MPI Partitioned communication where granularity of communications can be easily built into the overall communication profile [1], [38].

Previous works have focused on highly concurrent communication in MPI [39]–[41], but these works form the basis of the mechanisms that the CMB uses to assess the best performance mode for an application. Using prior works on highly concurrent MPI may be a method of determining if finer granularity is a good performance option for an application.

VII. Conclusion

In this paper we have presented motivation, design, and implementation of a Configurable Messaging Benchmark, the CMB. This benchmark allows for exploration of potential application performance impact of different fine-grained communication to a degree that has not been explored in prior work. To showcase this we presented a case study of the potential impact of different fine-grained implementations on different application profiles and two different HPC systems. The case study revealed the factors that effect the applicability of a particular fine-grained communication implementation vary depending on system. We provide high-level heuristic take-aways on when and how to best use partition communication. These results highlight the need for tools like the CMB to better understand fine-grained communication dynamics and guide the continued path of application and middleware codesign.

References

[1] R. E. Grant, M. G. Dosanjh, M. J. Levenhagen, R. Brightwell, and A. Skjellum, "Finepoints: Partitioned multithreaded MPI communication," in *International Conference on High Performance Computing*. Springer, 2019, pp. 330–350.

[2] Y. Hassan Temucin, R. E. Grant, and A. Afsahi, "Micro-benchmarking mpi partitioned point-to-point communication," in *Proceedings of the 51st International Conference on Parallel Processing*, ser. ICPP '22. New York, NY, USA: Association for Computing Machinery, 2023. [Online]. Available: https://doi.org/10.1145/3545008.3545088

[3] Message Passing Interface Forum, "MPI: A Message-Passing Interface Standard Version 4.0," 2021. [Online]. Available: https://www.mpi-forum.org/docs/

[4] W. P. Marts, M. G. Dosanjh, S. Levy, W. Schonbein, R. E. Grant, and P. G. Bridges, "Minimod: A modular miniapplication benchmarking framework for hpc," in *2021 IEEE International Conference on Cluster Computing (CLUSTER)*. IEEE, 2021, pp. 12–22.

[5] O. H. Mondragon, P. G. Bridges, S. Levy, K. B. Ferreira, and P. Widener, "Understanding performance interference in next-generation HPC systems," in *SC'16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 2016, pp. 384–395.

[6] K. B. Ferreira, P. Bridges, and R. Brightwell, "Characterizing application sensitivity to OS interference using kernel-level noise injection," in *High Performance Computing, Networking, Storage and Analysis, 2008. SC 2008. International Conference for*. IEEE, 2008, pp. 1–12.

[7] W. P. Marts, M. G. F. Dosanjh, W. Schonbein, S. Levy, and P. G. Bridges, "Measuring thread timing to assess the feasibility of early-bird message delivery," in *Proceedings of the 52nd International Conference on Parallel Processing Workshops*, ser. ICPP Workshops '23. New York, NY, USA: Association for Computing Machinery, 2023, p. 119–126. [Online]. Available: https://doi.org/10.1145/3605731.3605884

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

[9] Y. H. Temuçin, S. Levy, W. Schonbein, R. E. Grant, and A. Afsahi, "A dynamic network-native mpi partitioned aggregation over infiniband verbs," in *2023 IEEE International Conference on Cluster Computing (CLUSTER)*, 2023, pp. 259–270.

[10] N. Hjelm, M. G. F. Dosanjh, R. E. Grant, T. Groves, P. Bridges, and D. Arnold, "Improving MPI multi-threaded RMA communication performance," in *Proc. of the Int. Conf. on Parallel Processing*, 2018, pp. 1–10.

[11] M. G. F. Dosanjh, T. Groves, R. E. Grant, R. Brightwell, and P. G. Bridges, "RMA-MT: a benchmark suite for assessing MPI multi-threaded RMA performance," in *16th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*. IEEE, 2016, pp. 550–559.

[12] Mantevo, "Minife finite element mini-application," https://github.com/Mantevo/miniFE, accessed: 15-May-2023.

[13] ——, "Minimd molecular dynamics mini-application," https://github.com/Mantevo/miniMD, accessed: 12-May-2020.

[14] U. of Illinois/NCSA, "miniqmc - qmcpack miniapp," https://github.com/QMCPACK/miniqmc, accessed: 15-May-2023.

[15] A. Mathuriya, Y. Luo, A. Benali, L. Shulenburger, and J. Kim, "Optimization and parallelization of b-spline based orbital evaluations in qmc on multi/many-core shared memory processors," in *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 2017, pp. 213–223.

[16] National Energy Research Scientific Computing Center, "Cori," https://docs.nersc.gov/systems/cori/, retrieved 19 May 2022.

[17] Los Alamos National Laboratory, "Trinity," https://www.lanl.gov/projects/trinity/, retrieved 12 April 2019.

[18] W. Schonbein, S. Levy, M. G. F. Dosanjh, W. P. Marts, E. Reid, and R. E. Grant, "Modeling and benchmarking the potential benefit of early-bird transmission in fine-grained communication." New York, NY, USA: Association for Computing Machinery, 2023.

[19] T. Gillis, K. Raffenetti, H. Zhou, Y. Guo, and R. Thakur, "Quantifying the performance benefits of partitioned communication in mpi," in *Proceedings of the 52nd International Conference on Parallel Processing*, ser. ICPP '23. New York, NY, USA: Association for Computing Machinery, 2023, p. 285–294. [Online]. Available: https://doi.org/10.1145/3605573.3605599

[20] D. E. Bernholdt, S. Boehm, G. Bosilca, M. Venkata, R. E. Grant, T. Naughton, H. Pritchard, and G. Vallee, "A survey of MPI usage in the U.S. Exascale Computing Project," *Concurrency and Computation: Practice and Experience*, 2018, dOI: 10.1002/cpe.4851.

[21] P. Mendygral, N. Radcliffe, K. Kandalla, D. Porter, B. J. O'Neill, C. Nolting, P. Edmon, J. M. Donnert, and T. W. Jones, "WOMBAT: A scalable and high-performance astrophysical magnetohydrodynamics code," *The Astrophysical Journal Supplement Series*, vol. 228, no. 2, p. 23, 2017.

[22] M. A. Heroux, D. W. Doerfler, P. S. Crozier, J. M. Willenbring, H. C. Edwards, A. Williams, M. Rajan, E. R. Keiter, H. K. Thornquist, and R. W. Numrich, "Improving Performance via Mini-applications," Sandia National Laboratories, Tech. Rep. SAND2009-5574, 2009.

[23] I. Karlin, J. Keasler, and J. Neely, "Lulesh 2.0 updates and changes," Lawrence Livermore National Lab.(LLNL), Livermore, CA (United States), Tech. Rep., 2013.

[24] H. C. Edwards, C. R. Trott, and D. Sunderland, "Kokkos: Enabling manycore performance portability through polymorphic memory access patterns," *Journal of Parallel and Distributed Computing*, vol. 74, no. 12, pp. 3202–3216, 2014.

[25] R. D. Hornung and J. A. Keasler, "The RAJA portability layer: overview and status," Lawrence Livermore National Lab.(LLNL), Livermore, CA (United States), Tech. Rep., 2014.

[26] W. Gropp and E. Lusk, "Reproducible measurements of MPI performance characteristics," in *European Parallel Virtual Machine/Message Passing Interface Users' Group Meeting*. Springer, 1999, pp. 11–18.

[27] J. Liu, J. Wu, and D. K. Panda, "High performance RDMA-based MPI implementation over infiniband," *International Journal of Parallel Programming*, vol. 32, no. 3, pp. 167–198, 2004.

[28] M. Flajslik, J. Dinan, and K. D. Underwood, "Mitigating MPI message matching misery," in *International Conference on High Performance Computing*. Springer, 2016, pp. 281–299.

[29] K. B. Ferreira, S. Levy, K. Pedretti, and R. E. Grant, "Characterizing MPI matching via trace-based simulation," in *Proceedings of the 24th European MPI Users' Group Meeting*, ser. EuroMPI '17. New York, NY, USA: ACM, 2017, pp. 8:1–8:11.

[30] S. Levy, K. B. Ferreira, W. Schonbein, R. E. Grant, and M. G. Dosanjh, "Using simulation to examine the effect of MPI message matching costs on application performance," *Parallel Computing*, vol. 84, pp. 63–74, 2019.

[31] W. Schonbein, M. G. Dosanjh, R. E. Grant, and P. G. Bridges, "Measuring multithreaded message matching misery," in *European Conference on Parallel Processing*. Springer, 2018, pp. 480–491.

[32] S. M. Ghazimirsaeed, R. E. Grant, and A. Afsahi, "A dynamic, unified design for dedicated message matching engines for collective and point-to-point communications," *Parallel Computing*, vol. 89, p. 102547, 2019.

[33] W. P. Marts, M. G. Dosanjh, W. Schonbein, R. E. Grant, and P. G. Bridges, "MPI tag matching performance on ConnectX and ARM," in *Proceedings of the 26th European MPI Users' Group Meeting*, 2019, pp. 1–10.

[34] R. Grant, A. Skjellum, and P. V. Bangalore, "Lightweight threading with MPI using persistent communications semantics," Sandia National Laboratories (SNL-NM), Albuquerque, NM (United States), Tech. Rep., 2015.

[35] D. Holmes, K. Mohror, R. E. Grant, A. Skjellum, M. Schulz, W. Bland, and J. M. Squyres, "MPI sessions: Leveraging runtime infrastructure to increase scalability of applications at exascale," in *Proceedings of the 23rd European MPI Users' Group Meeting*, 2016, pp. 121–129.

[36] J. Dinan, R. E. Grant, P. Balaji, D. Goodell, D. Miller, M. Snir, and R. Thakur, "Enabling communication concurrency through flexible MPI endpoints," *The International Journal of High Performance Computing Applications*, vol. 28, no. 4, pp. 390–405, 2014.

[37] R. Zambre, A. Chandramowlishwaran, and P. Balaji, "How i learned to stop worrying about user-visible endpoints and love MPI," *arXiv preprint arXiv:2005.00263*, 2020.

[38] M. G. Dosanjh, A. Worley, D. Schafer, P. Soundararajan, S. Ghafoor, A. Skjellum, P. V. Bangalore, and R. E. Grant, "Implementation and evaluation of mpi 4.0 partitioned communication libraries," *Parallel Computing*, vol. 108, p. 102827, 2021.

[39] H. Kamal and A. Wagner, "FG-MPI: Fine-grain MPI for multicore and clusters," in *2010 IEEE International Symposium on Parallel & Distributed Processing, Workshops and Phd Forum (IPDPSW)*. IEEE, 2010, pp. 1–8.

[40] D. T. Stark, R. F. Barrett, R. E. Grant, S. L. Olivier, K. T. Pedretti, and C. T. Vaughan, "Early experiences co-scheduling work and communication tasks for hybrid MPI+X applications," in *Workshop on Exascale MPI*. IEEE Press, 2014, pp. 9–19.

[41] R. F. Barrett, D. T. Stark, C. T. Vaughan, R. E. Grant, S. L. Olivier, and K. T. Pedretti, "Toward an evolutionary task parallel integrated MPI+X programming model," in *6th Intl. Workshop on Programming Models and Applications for Multicores and Manycores*. ACM, 2015, pp. 30–39.