SAND2023-06315C

Measuring Thread Timing to Assess the Feasibility of Early-bird Message Delivery

W. Pepper Marts
wmarts@sandia.gov
Sandia National Laboratories
Albuquerque, New Mexico, USA

Matthew G. F. Dosanjh
mdosanj@sandia.gov
Sandia National Laboratories
Albuquerque, New Mexico, USA

Whit Schonbein
wwschon@sandia.gov
Sandia National Laboratories
Albuquerque, New Mexico, USA

Scott Levy
sllevy@sandia.gov
Sandia National Laboratories
Albuquerque, New Mexico, USA

Patrick G. Bridges
bridges@unm.edu
University of New Mexico
Albuquerque, New Mexico, USA

ABSTRACT

Early-bird communication is a communication/computation overlap technique that combines fine-grained communication with partitioned communication to improve application run-time. Communication is divided among the compute threads such that each individual thread can initiate transmission of its portion of the data as soon as it is complete rather than waiting for all of the threads. However, the benefit of early-bird communication depends on the completion timing of the individual threads.

In this paper, we measure and evaluate the potential overlap, the idle time each thread experiences between finishing their computation and the final thread finishing. These measurements help us understand whether a given application could benefit from early-bird communication. We present our technique for gathering this data and evaluate data collected from three proxy applications: MiniFE, MiniMD, and MiniQMC. To characterize the behavior of these workloads, we study the thread timings at both a macro level, i.e., across all threads across all runs of an application, and a micro level, i.e., within a single process of a single run. We observe that these applications exhibit significantly different behavior. While MiniFE and MiniQMC appear to be well-suited for early-bird communication because of their wider thread distribution and more frequent laggard threads, the behavior of MiniMD may limit its ability to leverage early-bird communication.

ACM Reference Format:
W. Pepper Marts, Matthew G. F. Dosanjh, Whit Schonbein, Scott Levy, and Patrick G. Bridges. 2023. Measuring Thread Timing to Assess the Feasibility of Early-bird Message Delivery. In *Proceedings of (PREPRINT).* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

PREPRINT, ,

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

To deal with the increased network demands of exascale supercomputers, many approaches have been proposed to support fine-grained communication in applications [5, 11]. Recent work explores the performance effects of utilizing different multithreaded communication models [15] as well as the distribution of times when threads complete their work [27]. Existing work has assumed that thread arrival times (the relative time at which threads rejoin at the end of a parallel compute section) follow a normal distribution, and that there regularly laggard threads (thread arrival times that are significantly later than the mean arrival time). However, previous work has not used empirical data to characterizing thread arrival distributions.

In this paper, we instrument and profile three proxy applications to capture the distribution of their thread arrival times. A statistical analysis of the resulting data creates a real world characterization of threaded HPC communication, and advances our understanding of how thread arrival times may change over the course of an application run. By providing a better understanding of these factors, the results reported in this paper provide an important empirical basis for the evaluation of different communication interfaces and multi-threaded communication models, such as partitioned communication.

The contributions of this paper are:

- A methodology for evaluating application thread behavior for multithreaded communication models;
- A study of three proxy applications to identify thread arrival distributions; and
- An analysis of the applicability of early-bird communication given the arrival distributions in three important HPC proxy applications.

The rest of this paper is structured as follows. Section 2 explains the background and the problem the paper addresses. Section 3 discusses the instrumentation and experimental set-up for this paper. Section 4 presents the results of our experiments. Section 5 discusses the implications of this work and presents our plan for future extensions. Section 6 contextualizes our work in the body of related work. Section 7 concludes our paper.

2 BACKGROUND

HPC applications traditionally adopt a bulk synchronous processing (BSP) model, where computation phases alternate with communication phases. In the BSP model, multithreading is often limited to the computation phase because communication between threads in different processes can incur significant overheads [28]. Regardless, application developers continue to express a desire to use multithreaded communication [1], and a variety of solutions to the problem have been explored, including optimizing message matching [7], employing software offloading [30], and using partitioned communication [11].

Partitioned communication is a strategy for dividing a communication buffer into smaller pieces such that communication can be independently initiated from different execution contexts (e.g., threads). Determining how to divide communication buffers and when to initiate communication is a complex, application-dependent question. For the purposes of the discussion in this paper, we use a simple model whereby each thread is assigned an equal, contiguous portion of the communication buffer and is responsible for initiating transmission of its portion of the data.

Currently, the most prominent approach for partitioned communication is described in the MPI 4.0 standard. However, the concept of partitioned communication is not limited to MPI. Therefore, the data and discussion in this paper are intended to apply to partitioned communication broadly rather than to MPI's definition of partitioned communication specifically.

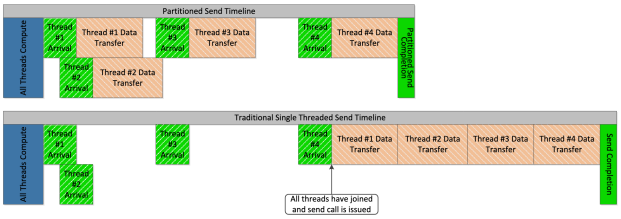


Figure 1: Early-bird model of communication.

One benefit of efficient multithreaded communication is that programmers can move communication calls to be near compute, increasing network utilization. This creates a form of communication/computation overlap called "early-bird communication". Figure 1 provides an illustration of how early-bird communication for partitioned communication in our model might work.

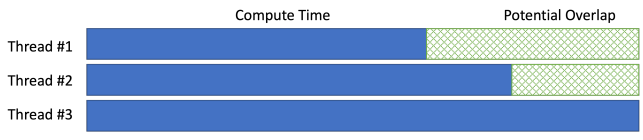


Figure 2: Potential for computation-communication overlap.

This leaves the question: how much do thread arrival times vary? If the thread arrival times are too similar, we expect applications to see a negative performance impact from moving to partitioned communication. However, as thread arrival times increase in variability the time for early-bird communication increases. Distribution likely has an impact as well; if an application is waiting on a single laggard thread (such as in OS noise [19]) it may be able to complete the transmission of all but the data produced by the laggard thread before the laggard thread finishes its computation. Figure 2 illustrates this with green boxes representing the time available for potential overlap.

3 INSTRUMENTATION AND EXPERIMENTAL DESIGN

3.1 Proxy Applications

We instrumented three proxy applications, MiniMD, MiniFE, and MiniQMC [12] in order to characterize the distribution of thread arrival times in a threaded compute section. In particular, we measure the times at which each thread enters and exits various OpenMP parallel-for regions for each iteration on each process. Timing data was collected using `clock_gettime()` with `CLOCK_MONOTONIC`, as defined in the IEEE POSIX.1-2017 standard [22]. The standard guarantees that the returned value is the time in nanoseconds since an undefined event in the past and that on a given core the time returned by `clock_gettime()` is never earlier than a previous call to the same function.

The POSIX standard does not provide this ordering guarantee across an entire multithreaded process spread across multiple cores and sockets. The necessary synchronization is not standard, and its existence is indicated by the `tsc_reliable` CPU flag. This feature is not available on many systems, including our test platform. In order to convert the returned value into a form that is comparable outside of a given compute core, we instead calculate a derived data point, *compute time*, as shown in Figure 2. Compute time for each thread is elapsed time in nanoseconds, and is calculated as the difference between the time the thread exits the parallel region and the time it enters. This subtraction cancels out any divergence in the results and allows for the comparison of thread timings across cores, sockets, and nodes.

```
#pragma omp parallel
{
  int t = omp_get_thread_num();
  #pragma omp barrier
  clock_gettime(CLOCK_MONOTONIC, &t_start[i][t]);
  #pragma omp for nowait
  for(int n = 0; n < nsteps; n++) {
    // do work
  }
  clock_gettime(CLOCK_MONOTONIC, &t_end[i][t]);
  #pragma omp barrier
}
```

Listing 1: Example of code instrumentation for data collection. The variable `i` is the application iteration number.

As illustrated in Listing 1, in each measured compute region, we nested the main `#pragma omp for` loop (adding the nowait flag) of each compute section inside of a `#pragma omp parallel` region. This allowed us to efficiently collect start and stop times in a threaded context without consideration of internal loop indexing. As we are using elapsed time as an estimate of thread arrival time,

we add a `#pragma omp barrier` before the collection of thread start times to synchronize the threads.

3.2 Experimental System and Application Configuration

Data was collected on the Manzano cluster. Each node has two 24-core Intel Cascade Lake CPUs running at 2.90 GHz and 192 GB of RAM. The machine uses the RHEL7 operating system and runs on an Intel Omni-Path network. Data collected on this system used OpenMPI 4.1.1 and all executables were compiled with GCC version 10.2.1. Each application was run for ten trials on eight nodes with one process per node. Each process used all 48 available hardware thread contexts, and was configured to run for two hundred iterations. For each process and each iteration, the application gathers timing data for each of its 48 threads.

For MiniMD [12], a parallel molecular dynamics proxy application based on LAMMPS [29], we timed all threaded compute sections in the application. Data shown is from the Lennard-Jones forcing function, the most computationally intensive section of the application. Data was collected with a compute volume of 128^3. For MiniFE, an unstructured mesh finite element solver, we timed the matrix vector product: the linear algebra function of highest order. Data was collected with a compute volume of 200^3 matrix elements per process. For MiniQMC, a quantum Monte Carlo proxy application based on QMCPACK [13], we timed the entirety of the computation for the individual threaded "movers". Although MiniQMC does not do meaningful inter-process communication, the class of applications it represents often perform considerable inter-node communication and MiniQMC serves as proxy the threading behavior we measure in this paper.

4 RESULTS

This section presents analysis of thread arrival times for each application. In order to evaluate the the sources of potential communication computation overlap put forward in Section 1, we break our analysis into two sections. First, an analysis of the potential normality of thread arival times as aggregated at three scales, and second, an analysis of laggard thread arrivals and characterization of classes of thread arrival distribution.

4.1 Evaluation of Thread Arrival Times for Normality

This section explores three logical groupings for understanding thread arrival time distributions: (1) At the level of an entire application across all trials and processes; (2) thread timings aggregated at the level of application iteration (the iteration count used by an application for a time step across all processes); and (3) aggregating at the level of individual processes on a single iteration (one processes' thread pool for a parallel compute region). We will refer to these as *application level aggregation*, *application iteration level aggregation*, and *process iteration level aggregation*, respectively. Existing work in the study of early-bird communication often assumes that thread arrival can be modeled based on a single distribution. We test this assertion for our three selected groupings.

Application level aggregation is explored for the possibility that thread arrival times can be described by a single normal distribution. In order to determine the normality of our thread arrival time distributions, we performed three tests for each application: D'Agostino [3], Shapiro-Wilk [24], and Anderson-Darling [26]. Data presented for Anderson-Darling is for a significance level of 5%. Each test assumes a null hypothesis that the data is normally distributed. Each test was run on the complete data set of application level thread arrival times for a total of 768000 samples per application. Results for all three of our applications led to rejecting the null hypothesis that the data is from a normal distribution. This strongly suggests that using a single, normal distribution of thread arrival times for every rank, trial, and iteration in each application is not valid model.

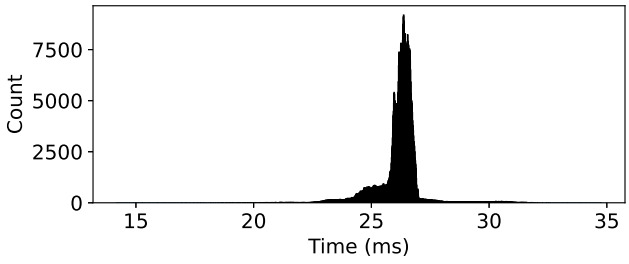
Thread timing data allows determining how application behavior can vary over the course of program execution. The thread arrival times for individual application iterations can be tested to see if they can be described by a normal distribution on an iteration by iteration basis. Running the same tests for each of the 200 application iterations that contain 3840 samples from each application resulted in identical results for MiniFE and MiniMD: thread arrival times for individual application iterations are not normally distributed. For MiniQMC however, there were eight application iterations for which D'Agostino's test failed to reject the null hypothesis. These same eight iterations did reject the null hypothesis for both Shapiro-Wilk or Anderson-Darling. It does not appear valid to assume that thread arrival times for an individual application iteration are normally distributed.

Test	MiniFE	MiniMD	MiniQMC
D'Agostino	3%	77%	95%
Shapiro-Wilk	$< 1\%$	74%	96%
Anderson-Darling	$< 1\%$	76%	96%

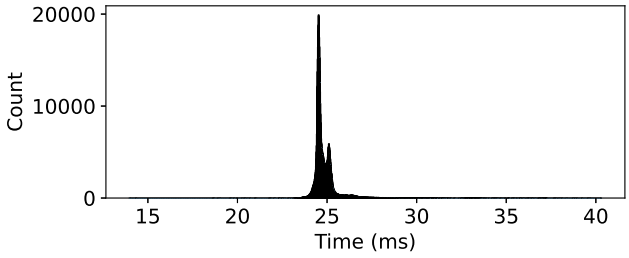
Table 1: Process iteration normality test results. Each cell is the percentage of aggregated process iterations that passed the normality test (i.e. failed to reject the null hypothesis).

Finally, testing at the finest explored level of aggregation, the process iteration, shows how threads join in an individual process' parallel compute context. Tests are run for each application on each for the 16000 process iteration level sets that contain 48 thread arrival samples. Table 1 presents the results of these tests. For the majority of MiniMD and MiniQMC process iterations, arrival times were normally distributed. For MiniFE less than 3% of process iterations were normally distributed. We see that the normality of process iteration arrival times varies depending on the application, with our three applications demonstrating the classes of nearly completely normal, nearly completely non-normal, and a mix of normal and non-normal. Process iteration arrival times for some applications can be modeled with a normal distribution, but it is not a constant.

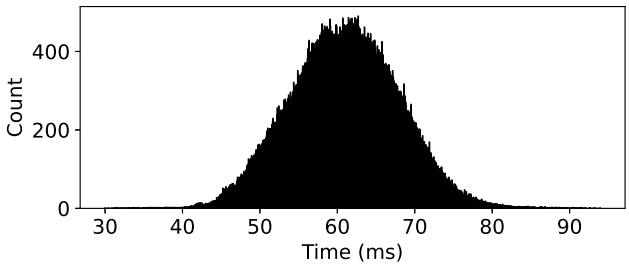
Even for applications where individual process iterations can be modeled with normal distributions, their aggregation in application level thread arrival time behavior remains complex. We have observed large variations both within and between applications. This



(a) MiniFE



(b) MiniMD



(c) MiniQMC

Figure 3: Application thread arrival time histograms for each of our three applications. Each has a bin width of 10 microseconds.

indicates that no single distribution is representative for all applications at any our examined aggregation levels. Figure 3 presents histograms of cumulative application level thread arrival times for each of our three applications with a bin width of 10 microseconds. In the Section 4.2 we characterize the particulars of each these application in greater detail.

4.2 Analysis of Laggard Thread Arrivals and Reclaimable Time

In this section we present the thread arrival distributions of the three applications, with a focus laggard thread arrivals and high-level trends across application iteration. Percentile plots display all thread arrival timings across each process and trial for a total of 3840 samples per iteration. Histograms in this section provide examples of arrival patterns observed within a process. The data presented in each histogram is only from a single iteration, process, and trial and is used to describe the time available for early-bird communication. The histograms are used to typify patterns seen in multiple iterations. Times shown are in milliseconds and represent time spent in the measured parallel compute region not the overall time spent in that iteration. Reclaimable time was determined by the summing the difference between the latest thread in that process iteration and each preceding thread. This paper presents two metrics. The first is the average amount of reclaimable time per iteration as averaged over the entire data set. The second is the average proportion of the time spent idle that iteration which is computed as the ratio between the cumulative time spent idle by all threads that iteration and the latest arrival time that iteration multiplied by number of threads. We will refer to these as *average reclaimable time* and the *ratio of time spent idle*, respectively.

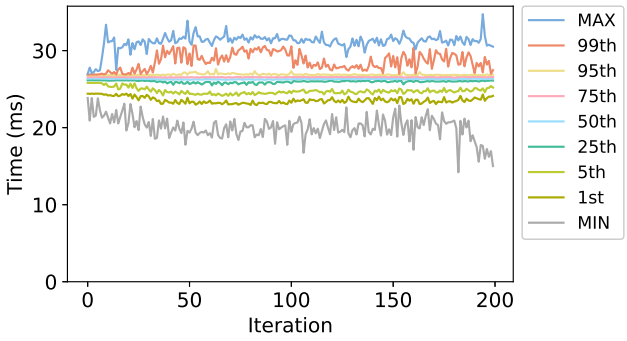
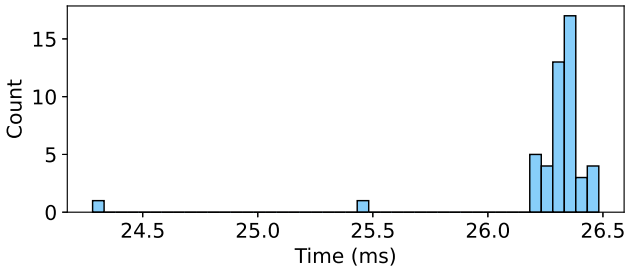


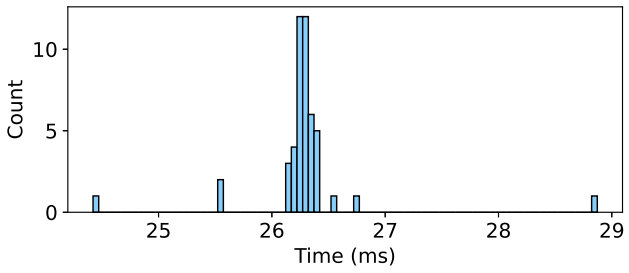
Figure 4: MiniFE: Time spent in computing matrix-vector product. Legend values correspond to percentiles of the collected thread execution times.

4.2.1 MiniFE. Figure 4 presents the thread arrival times for MiniFE as a percentile plot. The inter-quartile range has an average value of 0.18*ms* and a maximum value of 4.24*ms*. Based on Figure 3a and that the 5th and 25th percentiles are generally further from the median than the 95th and 75th percentiles, we can see that early arrival is significantly more common than late arrival for this application. The early threads are potentially due to work distribution imbalance; an outer loop iterates over 200 planes of the problem space and are distributed to 48 threads.

Figure 5 shows a pair of histograms presenting thread completion times taken from our MiniFE data. Each bin has a width of 50μs. In order to determine the reclaimable time that could potentially be used for early-bird communication, we examined the difference between the arrival of each thread and the last thread to arrive during that iteration on that process. We observed two patterns, Figure 5a shows a pattern without a laggard, and Figure 5b has a clear laggard thread. To identify how many of the observed iterations had a laggard, we found the difference between the median and maximum thread time and compared that to a threshold of 1*ms*. This value was chosen in order to determine if a thread arrival was approximately 5% slower than the mean median thread. We determined that only in 22.4% of iterations was the latest thread to arrive more that 1*ms* slower than the median thread. Regardless of whether a laggard was present, we can see a very tight distribution of thread arrivals. The mean median thread arrival time is 26.30*ms*. This corresponds to the peak in Figure 3a. Previous experiments



(a) 77.6% **of recorded iterations contain no laggard thread**



(b) 22.4% **of recorded iterations contain a laggard thread.**

Figure 5: Example histograms of MiniFE thread arrival distribution classes.

reveal that the distributions in Figure 5 are not normally distributed, but barring outliers they are symmetric and unimodal. The average reclaimable time was 42.82*ms* with a 0.1928 ratio of time spent idle.

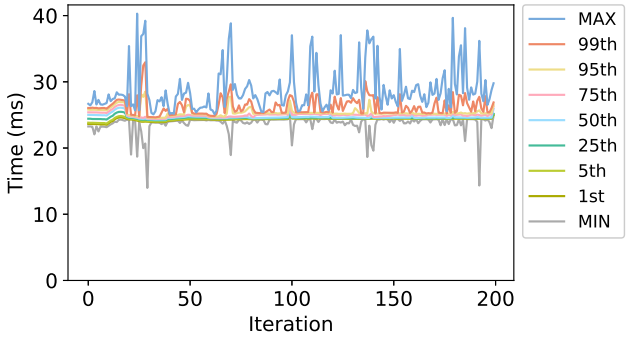
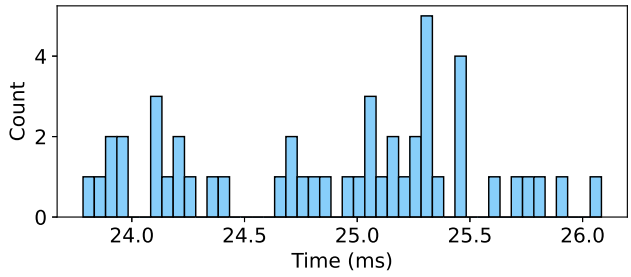
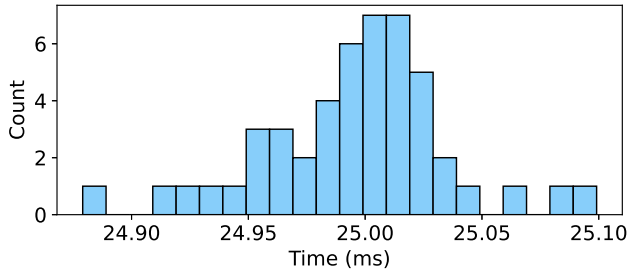


Figure 6: MiniMD: Time spent in Lennard-Jones forcing function. Legend values correspond to percentiles of the collected thread execution times.

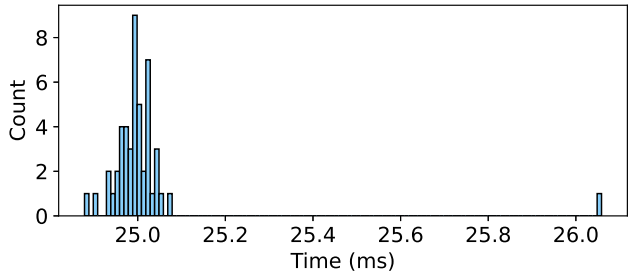
4.2.2 MiniMD. Figure 6 presents the thread arrival times for MiniMD as a percentile plot. These results show that two very different thread arrival distribution behaviors occur across application iteration. For the first nineteen iterations there is a significantly wider distribution of thread arrivals, which differs from the remainder of the application. This initial section appears to have consistent distribution of arrival times and few outliers of significant magnitude. This is followed by a section with sporadic laggard threads and



(a) Initial behavior (iterations one through nineteen).



(b) 95.2% **of recorded iterations contain no laggard thread.**



(c) 4.8% **of recorded iterations contain a laggard thread.**

Figure 7: Example histograms of MiniMD thread arrival distribution classes.

extremely few early arrivals. The inter-quartile range for the first section has an average value of 0.93*ms* and a maximum value of 1.45*ms* while the inter-quartile range for the second section has a much lower average value of 0.15*ms* and a much higher maximum value of 7.43*ms*.

Figure 7 shows three histograms, each representative of a subset of the observed distributions. Figure 7a gives an example of a distribution from the first nineteen iterations. Each bin has a width of 50μs. We found that the spread of times seen in the percentile plots is not a result of variation in process or trial but is instead present in individual iterations. The observed distributions were highly consistent, with a range of just over 2*ms* a median of between 25*ms* and 26*ms*. Figures 7b and 7c provide examples of the remainder of computation. Each bin has a width of 10μs. We determined that only in 4.8% of iterations was the latest tread to arrive more that 1*ms* slower than the median thread. Regardless of whether a laggard was present, we can see a very tight, normal distribution of thread arrivals. The mean median thread arrival time is 24.74*ms*.

This corresponds to the peak in Figure 3b. The average reclaimable time was 17.61ms with a 0.5012 ratio of time spent idle.

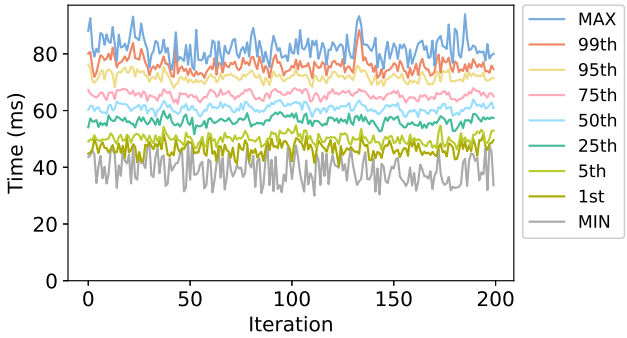


Figure 8: MiniQMC: Time spent computing movers. Legend values correspond to percentiles of the collected thread execution times.

4.2.3 MiniQMC. Figure 8 presents the thread arrival times for MiniQMC as a percentile plot. These results show that MiniQMC has the most uniform thread arrival distribution of the applications tested. Previous experiments reveal that these distributions are normally distributed. There is little variation across iterations. It is also notable for having the highest magnitude of variation among thread arrival times during each iteration with the inter-quartile range having a maximum value of 15.61ms and a mean value of 9.05ms.

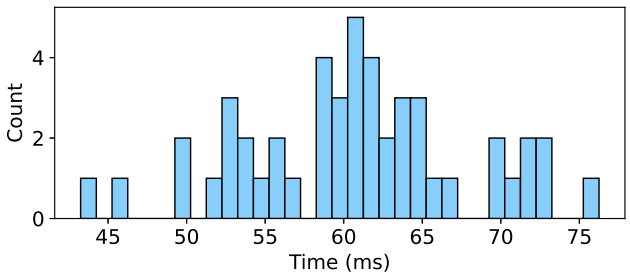


Figure 9: Example histogram of MiniQMC thread arrival distribution.

In order to determine whether the individual iterations have such a wide range of run-times or if the broad distributions are from the aggregation of the 80 process trial pairs, Figure 9 presents an example distribution from our MiniQMC data. Each bin has a width of 1ms. Our data showed that the breadth of over 40ms in the observed arrival times present in Figure 8 are the result of variation of thread arrivals in each iteration. The mean median thread arrival time is 60.91ms. This corresponds to the peak in Figure 3c. The average reclaimable time was 708.03ms with a 0.5033 ratio of time spent idle.

5 DISCUSSION AND FUTURE WORK

The reclaimable time results in the previous section demonstrate significant fork/join idle times that could be leveraged by early-bird communication in a restructured application to improve application performance. First, our analysis shows distributions of thread arrival times that have a large variance and significant numbers of laggard threads that suggest that early-bird transmission may improve communication performance. While these features are not present in every application or with perfect consistency, each application evaluated exhibited at least one of these two features frequently. Specifically, MiniMD and MiniQMC show significant thread idle times (50.12% and 50.33%, respectively) due to laggard threads (MiniMD), and the large variance of thread arrival times (MiniQMC). While MiniFE exhibited lower idle times (19.2791%), 22% of MiniFE iterations included laggard threads that could be potentially exploited by early-bird communication.

This highlights the opportunity to significantly improve application performance by taking advantage of this idle time for early-bird communication. However, successfully doing so would likely require significant changes to the applications, for example fusing multiple existing fork/join loops that precede communications or changing the overall communication plans of the applications. Current applications are not structured to do so, which is why our approach has focused on measuring extant fork/join idle times in the applications to understand the scale of this opportunity. Given the sample of proxy applications in this paper we see abundant opportunity for early-bird communication and reaffirm the assumptions of the existing literature.

This data also provides insight into potential directions for best implementing early-bird communication in applications, an active area of research. MiniFE shows a fairly consistent distribution of thread arrival times for the majority of executions; the main opportunity for reclaiming idle time are due to early completion in the one fifth of the time when laggard threads exist. In this case, system periodically transmits all available unsent data with a timeout based on this data would enable threads that were previously idle to efficiently transmit data in these idle times.

Similarly, the large variance of thread arrival times in MiniQMC and applications similar to it also include significant opportunities for leveraging early-bird communication. Because this arrival distribution results in 50% of cores consistently being idle, full applications with workloads similar to MiniQMC (e.g. QMCPACK) would significantly benefit from both a traditional binning model for aggregating data for early-bird communication and from fine-grain early-bird communication that does not leverage aggregation.

In contrast, the data shows that MiniMD would require a more sophisticated approach to successfully leverage early-bird communication. The first section of MiniMD would support a similar timeout or binning-based aggregation approach, but MiniMD also includes a second section where this model of overlap is unlikely to succeed. Specifically, most of the threads executed in this section have very similar arrival times, and laggard threads happen in only 4.8% of our observed iterations. When they do exist, they have high magnitude compared to median run time. Because of this, a more sophisticated approach would need to be used to leverage these relatively rare opportunities for early-bird communication.

6 RELATED WORK

In this paper, we examine thread timing measurements in the context of determining the extent to which early-bird communication in partitioned communication may yield faster message delivery. Thread timing measurements have been used in many existing research efforts, principally as a means of identifying and diagnosing performance issues.

Profiling Tools. Several existing tools have been developed to characterize the performance of individual threads in a multithreaded environment. Mohson et al. [18] provides an overview of these tools. Existing tools for characterizing multithreaded performance use several different approaches for collecting thread execution times, including using the OpenMP profiling interface [4, 8, 17], specialized processor counters [2], and information provided by the runtime [20]. Similarly, Gamblin et al. [9, 10] rely on the MPI profiling interface (PMPI) to characterize per-process execution times in MPI applications. In this paper, we manually instrument the code in our target workloads to precisely measure per-thread computation time within specific code blocks of interest, *see* Section 3.1.

Performance Analysis. A common motivation for collecting detailed measurements of thread execution times is to evaluate application performance and to diagnose performance problems (e.g., load imbalance, poor parallelization). The Performance Optimisation and Productivity (POP) Centre of Excellence defines a metric based on process execution times for characterizing the extent to which work is evenly distributed across processes. The Load Balance metric is defined as the ratio of the average process execution time to the maximum process execution time. Orland and Terboven [23] extend this metric to threads and examine load imbalance among OpenMP threads in GMRES. Muddukrishna et al. [21] use application characteristics, including thread execution times, to visualize application execution for the purpose of identifying performance bottlenecks. Liu et al. [14] break thread execution time into categories that enable them to identify periods of idleness that are indicative of poor performance. In this paper, we do not explicitly seek to understand the performance of current workloads. Rather, we use measurements of OpenMP thread execution time to characterize the opportunity to exploit early-bird communication in partitioned communication to deliver message contents earlier.

Fine-Grained Communication. There have been many other works that have looked fine-grained communication, particularly with an aim to support this form of communication-computation overlap. The gap in knowledge that this paper addresses is how to analyze application behavior to evaluate these techniques. The original partitioned communication paper [11] assumes a single laggard thread in the analysis. A notable extension is an evaluation by Temucin et al. [27] who evaluate the performance of partitioned communication under different distributions including a normal distribution. Wombat [16] was focused on adapting a single application to multithreaded communication. Other works that explore fine-grained communication have focused on evaluating the performance of their approach rather than developing a fine-grained application. This includes optimized message matching [7] and RMA-MT [6]. Finally, existing work on software offloading [30] and MPI Endpoints [25] adapted two computational kernels (QCD Dslash and FFT) to use their fine-grained communication schemes, but they did not evaluate thread timings.

7 CONCLUSION

Fine-grained, thread-safe communication, such as MPI partitioned communication, is a promising approach for enabling efficient communication in multithreaded HPC applications. In this paper, we instrumented several HPC proxy applications to understand the extent to which early-bird communication may allow for earlier message delivery. While additional investigation is necessary, the data we presented in this paper suggests that there may be a meaningful opportunity in scientific HPC applications to achieve better overall communication performance by using early-bird communication.

ACKNOWLEDGMENTS

This award was supported in part by the U.S. Department of Energy's National Nuclear Security Administration (NNSA) under the Predictive Science Academic Alliance Program (PSAAP-III), Award DE-NA0003966

This article has been authored by an employee of National Technology Engineering Solutions of Sandia, LLC under Contract No. DE-NA0003525 with the U.S. Department of Energy (DOE). The employee owns all right, title and interest in and to the article and is solely responsible for its contents. The United States Government retains and the publisher, by accepting the article for publication, acknowledges that the United States Government retains a non-exclusive, paid-up, irrevocable, world-wide license to publish or reproduce the published form of this article or allow others to do so, for United States Government purposes. The DOE will provide public access to these results of federally sponsored research in accordance with the DOE Public Access Plan https://www.energy.gov/downloads/doe-public-access-plan.

REFERENCES

[1] David E Bernholdt, Swen Boehm, George Bosilca, Manjunath Gorentla Venkata, Ryan E Grant, Thomas Naughton, Howard P Pritchard, Martin Schulz, and Geoffroy R Vallee. A survey of MPI usage in the US exascale computing project. *Concurrency and Computation: Practice and Experience*, 32(3):e4851, 2020.

[2] Intel Corporation. Intel vtune profiler user guide. https://www.intel.com/content/www/us/en/develop/documentation/vtune-help/top.html?wapkw=intel%20vtune%20profiler. Accessed: 2022-08-04.

[3] RALPH B. D'AGOSTINO. An omnibus test of normality for moderate and large size samples. *Biometrika*, 58(2):341–348, 1971.

[4] Luiz DeRose, Bernd Mohr, and Seetharami Seelam. Profiling and tracing OpenMP applications with POMP based monitoring libraries. In *European Conference on Parallel Processing*, pages 39–46. Springer, 2004.

[5] James Dinan, Pavan Balaji, David Goodell, Douglas Miller, Marc Snir, and Rajeev Thakur. Enabling MPI interoperability through flexible communication endpoints. In *Proceedings of the 20th European MPI Users' Group Meeting*, pages 13–18, 2013.

[6] Matthew GF Dosanjh, Taylor Groves, Ryan E Grant, Ron Brightwell, and Patrick G Bridges. RMA-MT: a benchmark suite for assessing MPI multi-threaded RMA performance. In *2016 16th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*, pages 550–559. IEEE, 2016.

[7] Mario Flajslik, James Dinan, and Keith D Underwood. Mitigating MPI message matching misery. In *International conference on high performance computing*, pages 281–299. Springer, 2016.

[8] Karl Fürlinger and Michael Gerndt. ompP: A profiling tool for OpenMP. In *International Workshop on OpenMP*, pages 15–23. Springer, 2005.

[9] Todd Gamblin. Scalable load balance analysis. https://github.com/tgamblin/libra. Accessed: 2022-08-03.

[10] Todd Gamblin, Bronis R De Supinski, Martin Schulz, Rob Fowler, and Daniel A Reed. Scalable load-balance measurement for SPMD codes. In *SC'08: Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, pages 1–12. IEEE, 2008.

[11] Ryan E Grant, Matthew GF Dosanjh, Michael J Levenhagen, Ron Brightwell, and Anthony Skjellum. Finepoints: Partitioned multithreaded MPI communication. In *International Conference on High Performance Computing*, pages 330–350. Springer, 2019.

[12] Michael Heroux and Richard Barrett. Mantevo project, 2019.

[13] Jeongnim Kim, Andrew D Baczewski, Todd D Beaudet, Anouar Benali, M Chandler Bennett, Mark A Berrill, Nick S Blunt, Edgar Josué Landinez Borda, Michele Casula, David M Ceperley, Simone Chiesa, Bryan K Clark, Raymond C Clay, Kris T Delaney, Mark Dewing, Kenneth P Esler, Hongxia Hao, Olle Heinonen, Paul R C Kent, Jaron T Krogel, Ilkka Kylänpää, Ying Wai Li, M Graham Lopez, Ye Luo, Fionn D Malone, Richard M Martin, Amrita Mathuriya, Jeremy McMinis, Cody A Melton, Lubos Mitas, Miguel A Morales, Eric Neuscamman, William D Parker, Sergio D Pineda Flores, Nichols A Romero, Brenda M Rubenstein, Jacqueline A R Shea, Hyeondeok Shin, Luke Shulenburger, Andreas F Tillack, Joshua P Townsend, Norm M Tubman, Brett Van Der Goetz, Jordan E Vincent, D ChangMo Yang, Yubo Yang, Shuai Zhang, and Luning Zhao. ttQMCPACK/tt: an open sourceiab initio/iquantum monte carlo package for the electronic structure of atoms, molecules and solids. *Journal of Physics: Condensed Matter*, 30(19):195901, April 2018.

[14] Xu Liu, John Mellor-Crummey, and Michael Fagan. A new approach for performance analysis of OpenMP programs. In *Proceedings of the 27th international ACM conference on International conference on supercomputing*, pages 69–80, 2013.

[15] W Pepper Marts, Matthew GF Dosanjh, Scott Levy, Whit Schonbein, Ryan E Grant, and Patrick G Bridges. MiniMod: A modular miniapplication benchmarking framework for HPC. In *2021 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 12–22. IEEE, 2021.

[16] PJ Mendygral, Nick Radcliffe, Krishna Kandalla, David Porter, Brian J O'Neill, Chris Nolting, Paul Edmon, Julius MF Donnert, and Thomas W Jones. WOMBAT: A scalable and high-performance astrophysical magnetohydrodynamics code. *The Astrophysical Journal Supplement Series*, 228(2):23, 2017.

[17] Bernd Mohr, Allen D Malony, Sameer Shende, and Felix Wolf. Design and prototype of a performance tool interface for OpenMP. *The Journal of Supercomputing*, 23(1):105–128, 2002.

[18] Mubrak S Mohsen, Rosni Abdullah, and Yong M Teo. A survey on performance tools for OpenMP. *World Academy of Science, Engineering and Technology*, 49:754–765, 2009.

[19] Alessandro Morari, Roberto Gioiosa, Robert W Wisniewski, Francisco J Cazorla, and Mateo Valero. A quantitative analysis of OS noise. In *2011 IEEE International Parallel & Distributed Processing Symposium*, pages 852–863. IEEE, 2011.

[20] Ananya Muddukrishna, Peter A Jonsson, and Mats Brorsson. Characterizing task-based OpenMP programs. *PloS one*, 10(4):e0123545, 2015.

[21] Ananya Muddukrishna, Peter A Jonsson, Artur Podobas, and Mats Brorsson. Grain graphs: OpenMP performance analysis made easy. In *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 1–13, 2016.

[22] Institute of Electrical and Electronics Engineers. International organization for standardization. information technology–portable operating system interface (posix). https://pubs.opengroup.org/onlinepubs/9699919799.2018edition/. Accessed: 2022-08-04.

[23] Fabian Orland and Christian Terboven. A case study on addressing complex load imbalance in OpenMP. In *International Workshop on OpenMP*, pages 130–145. Springer, 2020.

[24] S. S. SHAPIRO and M. B. WILK. An analysis of variance test for normality (complete samples). *Biometrika*, 52(3-4):591–611, dec 1965.

[25] Srinivas Sridharan, James Dinan, and Dhiraj D Kalamkar. Enabling efficient multithreaded MPI communication through a library-based implementation of MPI endpoints. In *SC'14: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 487–498. IEEE, 2014.

[26] M. A. Stephens. EDF statistics for goodness of fit and some comparisons. *Journal of the American Statistical Association*, 69(347):730–737, September 1974.

[27] Yiltan Hassan Temucin, Ryan Grant, and Ahmad Afsahi. Micro-benchmarking MPI partitioned point-to-point communication. In *2022 International Conference on Parallel Processing (ICPP*. ACM, 2022.

[28] Rajeev Thakur and William Gropp. Test suite for evaluating performance of multithreaded MPI communication. *Parallel Computing*, 35(12):608–617, 2009.

[29] A. P. Thompson, H. M. Aktulga, R. Berger, D. S. Bolintineanu, W. M. Brown, P. S. Crozier, P. J. in 't Veld, A. Kohlmeyer, S. G. Moore, T. D. Nguyen, R. Shan, M. J. Stevens, J. Tranchida, C. Trott, and S. J. Plimpton. LAMMPS - a flexible simulation tool for particle-based materials modeling at the atomic, meso, and continuum scales. *Comp. Phys. Comm.*, 271:108171, 2022.

[30] Karthikeyan Vaidyanathan, Dhiraj D Kalamkar, Kiran Pamnany, Jeff R Hammond, Pavan Balaji, Dipankar Das, Jongsoo Park, and Bálint Joó. Improving concurrency and asynchrony in multithreaded MPI applications using software offloading. In *SC'15: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–12. IEEE, 2015.