Characterization of MPI Usage on a Production Supercomputer

Sudheer Chunduri, Scott Parker, Pavan Balaji, Kevin Harms and Kalyan Kumaran
Argonne National Laboratory,
{*sudheer, sparker, balaji, kharms, kumaran*}*@anl.gov*

***Abstract*—MPI is the most prominent programming model used in scientific computing today. Despite the importance of MPI, however, how scientific computing applications use it in production is not well understood. This lack of understanding is attributed primarily to the fact that production systems are often wary of incorporating automatic profiling tools that perform such analysis because of concerns about potential performance overheads. In this study, we used a lightweight profiling tool, called Autoperf, to log the MPI usage characteristics of production applications on a large IBM BG/Q supercomputing system (Mira) and its corresponding development system (Cetus). Autoperf limits the amount of information that it records, in order to keep the overhead to a minimum while still storing enough data to derive useful insights. MPI usage statistics have been collected for over 100K jobs that were run within a two-year period and are analyzed in this paper. The analysis is intended to provide useful insights for MPI developers and network hardware developers for their next generation of improvements and for supercomputing center operators for their next system procurements.**

***Index Terms*—MPI, monitoring, Autoperf, core-hours**

I. Introduction

Optimizing scientific computing applications to run on large supercomputing systems is a complicated process. Effective utilization of communication resources is vital for application performance and scaling efficiency. MPI [1] is the predominant parallel programming model for scientific computing today, making it a key technology to be optimized so that scientific computing applications can take full advantage of the supercomputing system that they use. Optimization requires a detailed understanding of the usage characteristics of applications on production supercomputing systems. Unfortunately, such a usage characterization does not exist today, at least not on large production systems. This situation is attributed primarily to the fact that production systems are often wary of incorporating automatic profiling tools that perform such analysis, because of concerns about potential performance overheads.

In this paper, we analyze the MPI usage characteristics of applications on production supercomputers. To this end, we first present a lightweight profiling tool, called Autoperf, that we developed as a mechanism for automatically profiling the MPI usage of applications executing on a large supercomputing system. Autoperf is a PMPI-based tool that transparently traps MPI calls and gathers various statistics on the MPI calls. It restricts the data gathered to simple summaries of the statistics, rather than detailed traces, in order to keep the overhead—in terms of both cycle count and memory/cache footprint—to a minimum.

Using Autoperf, we have captured the MPI usage characteristics of production applications on a large IBM BG/Q supercomputing system (Mira) and its corresponding development system (Cetus). Mira is a 786,432-core (10 petaflop) supercomputer that is ranked 11th in the November 2017 Top500 ranking. Cetus is a 65,536-core supercomputer. Having a development system corresponding to a production supercomputer has become increasingly common in the past decade. These development systems are smaller-scale versions of the larger production machine, typically with an identical hardware and software infrastructure. They are meant to be used as early development or performance-tuning platforms for applications that are eventually intended to be executed on the full production machine. A side-by-side comparison of both these systems over the same period gives an indication of the MPI usage characteristics of the applications during their *development time* as well as during their *production runs*.

MPI usage statistics have been collected for over 100K jobs that were run within a two-year period and are analyzed in this paper. The bulk of our analysis filters out jobs that are known MPI microbenchmarks, jobs that are known test applications (meant for development or profiling), and jobs that are not long enough in terms of their runtime to be considered real applications. This filtering was done in order to avoid diluting our results with such jobs. It leaves us with only those jobs that we consider are solving real science problems and are thus the true intent for the usage of the supercomputer.

We present a detailed analysis of the MPI usage logs that were gathered using Autoperf. In particular, some of the key (and surprising) insights that we gathered are listed below.

1) The fraction of time used by the MPI library is much larger than what was previously assumed by most supercomputing centers. While most centers realize the importance of MPI, the general assumption has been that most production applications tend to spend less than a quarter of their time in MPI. Our analysis shows that this is not true even for large production applications. In fact, a reasonably high number of applications spend more than half their time in MPI.
2) MPI collectives take a significantly larger fraction of time compared with point-to-point (send, receive) operations, in terms of both the number of calls and the total time spent. In fact, the few applications that are

SC18, November 11-16, 2018, Dallas, Texas, USA
978-1-5386-8384-2/18/$31.00

dominated by point-to-point operations are those that perform structured nearest-neighbor communication. Private discussions with these application developers indicate that the general goal is to replace the point-to-point operations with neighborhood collective operations that were introduced in MPI-3. This replacement was not done on Mira and Cetus, however, because the MPI implementation on those machines supports only MPI-2.1. Nevertheless, this trend makes point-to-point operations even less critical in the future.

3) Hybrid MPI+OpenMP (or pthreads) applications are more widely used than we expected. In particular, we were surprised to see that approximately 30% of the profiled jobs used `MPI_THREAD_MULTIPLE` mode where multiple threads were issuing MPI calls simultaneously. Although this trend might be exaggerated by the fact that IBM BG/Q is one of the few machines that provide an efficient `MPI_THREAD_MULTIPLE` implementation, it does point to a general trend in what applications would like to use if it were sufficiently optimized.
4) Small message ($\leq$ 256 bytes) `MPI_Allreduce` operations are, by far, the most heavily used part of MPI, in terms of both the number of times the reduction function is called and the total amount of time spent in it. This is a reasonably well-understood fact. The surprising part, however, is that nearly 20% of the jobs use very large message ($\geq$ 512-Kbyte) `MPI_Allreduce` operations.

The intent of this analysis is multifold. For MPI developers, these insights are valuable for better aligning their feature development roadmaps. For network hardware developers, the insights point to the usage characteristics of the network subsystem and help estimate the network efficiency of the systems. For operators of supercomputer centers, these insights can provide critical help in administering the current systems and, more important, in making optimal system design decisions for future procurements.

The insights presented from this study would be similar to any other system that is of same large scale as BG/Q. At this scale, the applications tend to use more communication and parallelism. While the insights at a finer level vary from Mira to Cetus owing to the way these machines operate and their scales, at a high-level, the key take aways from the MPI usage patterns remain the same between these two systems. On a different system with a different application workload set, we could potentially notice a different set of insights. However, we expect that the large collection of data analyzed in this study repesents the diversity of workload set used by the scientific community, and thus the insights are of general relevance.

The rest of the paper is organized as follows. The design of the Autoperf tool along with an assessment on its overhead is provided in Section II. Section III provides an overview of all the jobs run on Mira and Cetus in the two-year time frame considered in this study. Section IV discusses the MPI usage across all the Autoperf jobs. Other literature related to this paper is presented in Section V, and our concluding remarks are presented in Section VI.

II. Autoperf: Lightweight MPI Profiling

Autoperf is a lightweight profiling library for the automatic collection of hardware performance counters and MPI statistics. The library transparently collects performance data from a running job, using PMPI redirection for MPI and hardware counters for processor usage data, and stores this information as log files at job completion. Autoperf output is in plain text format and includes MPI usage and performance information indicating which MPI routines were called, how many times each routine was called, the time spent in each routine, and the number of bytes sent or received, where applicable. Data from the hardware performance counters such as instruction counts, flop rates, and memory usage is also collected and recorded in the log.

To collect performance data and generate performance data files, the program must use MPI, call `MPI_Init` and `MPI_Finalize`, and terminate without error. Logs are generated at `MPI_Finalize` so applications that fail to reach `MPI_Finalize` will not record a log.

Autoperf records MPI usage statistics from all MPI processes. However, in order to reduce memory footprint and compute overhead, data from only four processes is written to the log file. These four processes are the process with MPI rank zero, the process that takes maximum MPI time, the process that takes minimum MPI time and the process that takes MPI time close to the average MPI time across all the ranks. While the data from these four ranks can be used to approximate the load imbalance in the total MPI communication, the tool does not capture latency distribution across the different calls for a specific collective, hence, we can't differentiate the collectives actual latency vs. synchronization overhead due to load imbalance. Unless otherwise noted, the data reported by the process with average MPI time is used for further analysis in this study.

A. Overhead of Autoperf

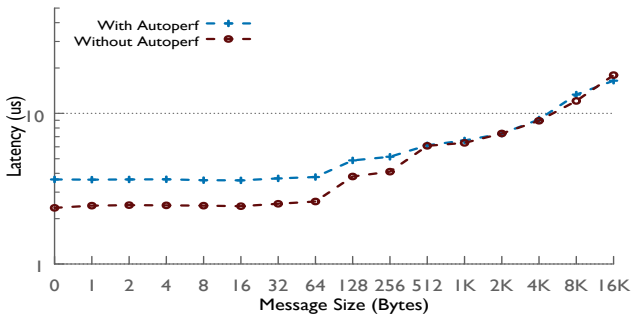

Fig. 1: Ping-pong latency with and without using Autoperf

An important aspect of any profiling tool is that it should not deviate significantly from the nonprofiled execution. Thus, the overhead added by Autoperf—in terms of both performance and memory/cache usage—is important. Before we analyze this overhead, we briefly describe the operations involved in a typical Autoperf execution. Autoperf's MPI function wrapper performs the following operations in each call:

Log Data	Job Count	Core-Hours	Executables
RAW data (All jobs)	682255 (100%)	11.6 Bi (100%)	1450
INCITE+ALCC jobs	505648 (74.1%)	11.1 Bi (95.6%)	819
Jobs after Filtering (Filtered jobs)	405100 (59.3%)	10.6 Bi (91.1%)	110
Filtered Jobs with exit status 0 or $>$ 255	337942 (49.5%)	6.4 Bi (54.8%)	105
Filtered Jobs with exit status $\geq$ 1 & $\leq$ 255	67158 (9.8%)	4.2 Bi (36.2%)	110
Filtered Jobs with Autoperf log (Autoperf jobs)	86490 (12.6%)	2.6 Bi (23.0%)	64

TABLE I: Overview of jobs on Mira for two-year period

Log Data	Job Count	Core-Hours	Executables
RAW data (All jobs)	914419 (100%)	736 Mi (100%)	4352
INCITE+ALCC jobs	148494 (16.2%)	191 Mi (25.9%)	651
Jobs after Filtering (Filtered jobs)	100442 (10.9%)	172 Mi (23.3%)	28
Filtered Jobs with exit status 0 or $>$ 255	81839 (8.9%)	83 Mi (11.2%)	28
Filtered Jobs with exit status $\geq$ 1 & $\leq$ 255	18603 (2.0%)	88 Mi (12.0%)	28
Filtered Jobs with Autoperf log (Autoperf jobs)	34012 (3%)	35 Mi (4.7%)	15

TABLE II: Overview of jobs on Cetus for two-year period

1) Obtain the time stamp at the start of the call.
2) Convert the MPI data type to a size in bytes.
3) Add the resulting number to the accumulated byte total for the MPI function.
4) Increment the call counter for the MPI function.
5) Get the time stamp at the end of the call.
6) Subtract from the start time, and add the result to accumulated time for the MPI function.

To understand the overhead associated with the above sequence of steps, we first performed a simple ping-pong benchmark to measure the overhead caused by Autoperf. The results in Figure 1 show that Autoperf adds less than 0.2 us overhead on the performance for latency-sensitive small message sizes, that is, around 300 processor cycles per MPI call. When the message size is 512 bytes or larger, the overhead from Autoperf is not noticeable. We also analyzed the overhead of Autoperf with several real applications, where the overhead was negligible. For instance, with the Nek5000 and VSVB applications and the Nekbone miniapp, autoperf adds less than 0.05% overhead when running on 256 nodes. Apart from the overhead coming the PMPI instrumentation, the only other overhead comes from the statistics aggregation phase (`MPI_Reduce`) onto selected ranks. Irrespective of the number of ranks used in the application, the overhead from this phase is negligible.

With respect to its memory and cache footprint, Autoperf adds around 200 KB of data per process and collects only profile data. Profile size is fixed and is not dependent on the characteristics of the process. This includes storage for the hardware performance counters as well as that for MPI functions. The active memory footprint, which is the amount of data that is fetched and processed during most of the execution, is just three data entities, timestamp (64-bit int), call_count (64-bit int) and total_bytes (double), for each monitored MPI function. All data should fit into a single cache line, thus, minimizing cache pollution and allowing for efficient access.

B. Limitations of Autoperf

Although Autoperf is commissioned in production to capture MPI usage data for all the jobs, not all the jobs would have an Autoperf log. To determine the reasons for missing coverage of Autoperf for some executables, we used multiple sources, including surveying the users with a questionnaire and parsing through the logs of Tracklibs (a tool linked by default with all executables that logs the set of libraries linked). The reasons gathered are provided in the appendix A.

III. Overview of Jobs

In this section we first describe the characteristics of all the jobs run on Mira and Cetus over the past two years. We also demonstrate later that the jobs monitored by Autoperf are representative of all jobs of significance on the respective systems.

A. Control System

The BG/Q nodes are stateless with no embedded read-only memories or resident basic input/output system. When the node hardware is reset, the control system server [2] loads the OS into the memory of each compute node and boots the nodes. The control system server, encompassing multiple software components for managing the hardware resources, is run on the service node and provides an interface to the hardware-level system components. Users access the control system primarily through a job scheduler, whereas system administrators can log all the requests processed by the control server. The BG/Q job schedulers use the runjob [2] interface for job submission. The control system logs includes all the runjob commands processed on the system. We use the control system log data pertaining to the two-year time frame and process these logs to determine the jobs that were run successfully and the jobs that were aborted. If a job fails to start or exits because of any signal, the nonzero exit status will be between 1 and 255 inclusive [2]. A zero exit status or an exit status of more than 255 corresponds to a normally exited job, and these jobs should potentially go through the

Autoperf summarization step in `MPI_Finalize` and thus should ideally have an Autoperf log.

B. Summary of Jobs

Table I summarizes all the jobs run on Mira in the two-year time frame (years 2016 and 2017). The total core-hours consumed on the system for these two years account for around 11.6 billion core-hours spread across 1,450 unique executables. These jobs come from the different compute time allocations on Mira including major scientific compute allocations such INCITE [3] and ALCC [4] as well as smaller experimental allocations. The INCITE and ALCC allocations cater to grand challenge investigations in science and technology that have the potential for impact at the national and global scale. To ensure that our observations and insights in this study are based on real scientific codes and not on benchmark codes, we first filter out the jobs belonging to allocations other than INCITE and ALCC as they generally do not represent scientific production simulations. By applying this filtering, although we lose 25.9% of the jobs in terms of job count, only 4.4% of the core-hours are lost. However, we still have around 819 unique executables, which possibly include executables corresponding test and debug runs. Hence, we apply another filtering criterion, removing jobs belonging to executables that take less than 0.1% of the total (11.6 Bi) core-hours. With this filtering, we are left with around 405K jobs accounting for around 10.6 Bi core-hours. We now have only 110 executables, and these executables belong to various science domains, with lattice QCD (18), molecular dynamics (11), computational fluid dynamics (11) and quantum chemistry (9) being the prominent domains. While 47 of these 110 executables have no Autoperf coverage, around 30 executables have more than 50% of their core-hours logged by Autoperf. The reason for this missing coverage from Autoperf was discussed in Section II-B.

Table I categorizes these jobs based on their exit status. Not all the jobs that exit cleanly have an Autoperf coverage, with only 23% of the total core-hours covered by Autoperf. A high-level summary of jobs on Cetus is provided in Table II. Owing to its mode of usage as a test and development resource, filtered jobs (applying the same filtering criteria as applied earlier) on Cetus come from a smaller number (15) of applications.

The following terminology is used for the different sets of jobs analyzed in this study: **All jobs:** all the jobs logged by the Control system; **Filtered jobs**: jobs after removing the those that do not meet the filtering criteria (jobs corresponding to applications that take less than 0.1% of the total core-hours and belonging to an ALCC or INCITE allocation); and **Filtered Autoperf jobs**: filtered jobs that have an Autoperf log.

C. Core-Hours and Runtimes of the Jobs

A more detailed analysis of the jobs in terms of their core-hour and runtime distributions is provided here. Figure 2 shows the distribution of core-hours across the different allocation sizes for All jobs, Filtered jobs, and Autoperf jobs on Mira. It shows that around 10% of the total core-hours is accounted for by jobs that use around 48K nodes (whole system) allocation. All jobs, Filtered jobs, and Autoperf jobs have roughly similar distributions except for 4K jobs, where Autoperf jobs consume significantly more core-hours. Even though of 90% (in job count) of the All jobs use 2K or fewer nodes, they consume only 50% of the core-hours. The rest of the 10% jobs use more than 2K and up to 48K nodes and consume the remaining 50% of the total core-hours. Autoperf has good coverage for these large node jobs. The core-hour distribution for jobs on Cetus is shown in Figure 3.

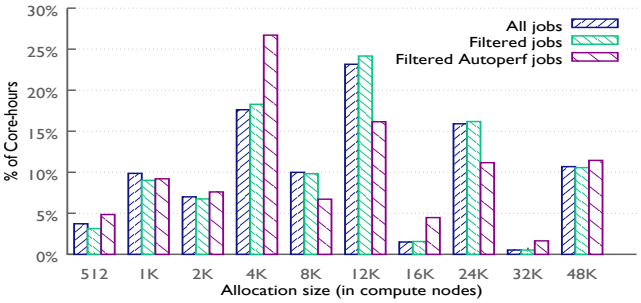

Fig. 2: Job sizes and the corresponding core-hours on Mira

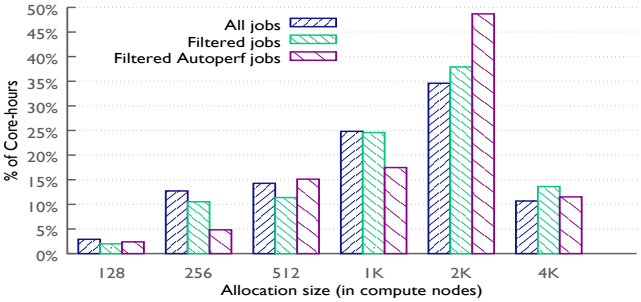

Fig. 3: Job sizes and the corresponding core-hours on Cetus

Figure 4 shows the cumulative distribution of runtimes across the jobs on Mira sorted by the runtimes for All jobs, Filtered jobs and Autoperf jobs. Although the aim of the filtering criteria is to filter out test and benchmark executables, which typically have smaller runtimes, no explicit filtering scheme based on runtimes is used. Hence, some runs of filtered applications may still have short runtimes; however, these jobs would not have a meaningful Autoperf log and hence do not appear in the Autoperf jobs. The range of Autoperf runtimes starts at a higher value than that of the Filtered jobs, with 80% of the core-hours corresponding to jobs that take more than 1,000 seconds. More importantly, the distribution of runtimes in Autoperf jobs is representative of the runtimes for the Filtered jobs.

Similarly, the runtime distributions on Cetus are shown in Figure 5.

Since, Autoperf jobs cover the spread of job sizes possible on the systems and their runtime and core-hour distributions are representative of the Filtered jobs, we argue that the observations we make in the following sections based on

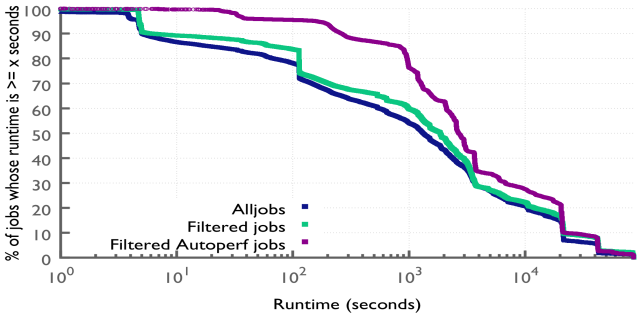

Fig. 4: Runtime distribution across the jobs on Mira

the Autoperf log data have significant merit because they are representive of the entire systems workload.

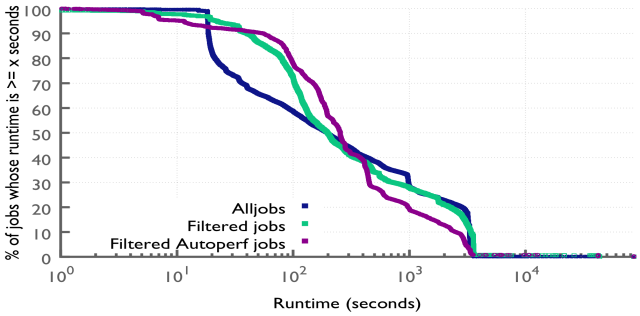

Fig. 5: Runtime distribution across the jobs on Cetus

IV. MPI Usage across all the Applications on Mira & Cetus

This section provides a high-level overview of the MPI usage across all the Filtered Autoperf jobs and the corresponding applications. In this section, the term total jobs refers to the Filtered Autoperf jobs.

A. Overview of MPI Usage of Autoperf Jobs

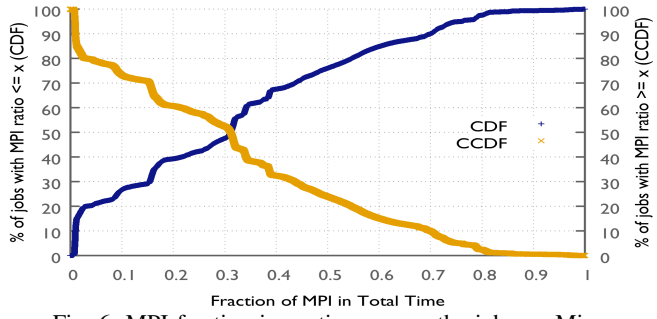

Fig. 6: MPI fraction in runtime across the jobs on Mira

An Autoperf log records the time spent in MPI and the total runtime of the job. The ratio of these two times gives the fraction of time spent in the communication (MPI) phases of the application. Figure 6 shows the MPI fraction across all the jobs on Mira. The CDF (cumulative distribution function) and the CCDF (complementary cumulative distribution function)

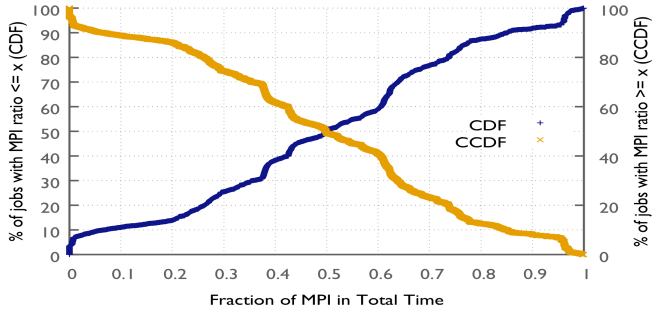

Fig. 7: MPI fraction in runtime across the jobs on Cetus

are shown in Figure 6. It shows that around half of the jobs have an MPI fraction of 0.3 or more (i.e., 30% of the total time is spent within MPI). Also, around 20% of the jobs spend almost no time in MPI, having a MPI fraction of close to zero. We attribute this behavior to the Monte Carlo applications such as QMCPACK [5], QWalk [6], and topmon [7]. Further, around 15% of the jobs have spent more than 60% of the total time in MPI. While the Figure 6 shows the MPI fraction in the total time, the constituents parts of the MPI time in terms of collectives and point-to-point times is shown in the Figures 14 and 24 respectively.

Figure 7 shows the MPI fraction across all the jobs on Cetus. Around half of the jobs have an MPI fraction of 0.5 or more. Further, around 10% of the jobs have spent more than 80% of the total time in MPI.

Roughly 34% of core-hours on Mira are expended in MPI.

In summary, a significant portion of the resource usage (represented in terms of core-hours) is accounted for by the time spent in communication and synchronization. One cannot always decompose this time into constituent parts such as time spent in the MPI software stack, time spent in the network interface layer (such as SPI on BG/Q), time delay on the interconnect network and the time spent waiting due to application load imbalance. At a high level this suggests that optimizing MPI could improve the performance of applications to the extent that the time in MPI is bound by communication latency, i.e., the MPI time is bound by the actual communication latency (number of instructions needed to instruct the hardware to perform communication) rather than the synchronization time (due to application load imbalance).

B. MPI Collectives and Point-to-Point Operations

Presented here are the usage patterns of collective and point-to-point operations summarized across all the runs in the filtered Autoperf logs. The collective operations account for 66% of the total MPI time, whereas the rest is accounted for by the point-to-point operations.

Collectives are used prominently than point-to-point operations.

1) MPI Collectives: Figures 8 and 9 show the aggregated call counts and times (represented in terms of core-hours along

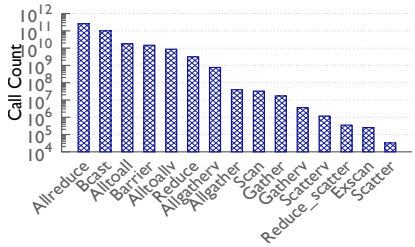

Fig. 8: MPI collectives call count on Mira

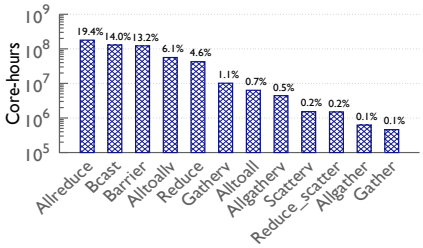

Fig. 9: MPI collectives time (in terms of core-hours) on Mira

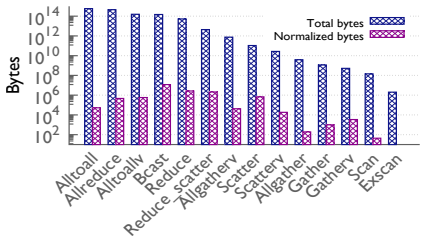

Fig. 10: MPI collectives total bytes (normalized bytes) used on Mira

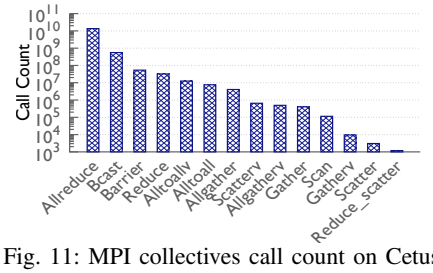

Fig. 11: MPI collectives call count on Cetus

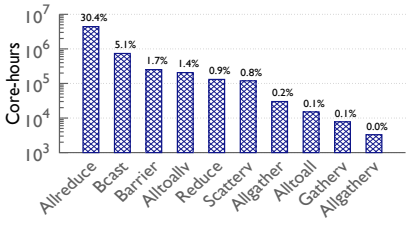

Fig. 12: MPI collectives time (in terms of core-hours) on Cetus

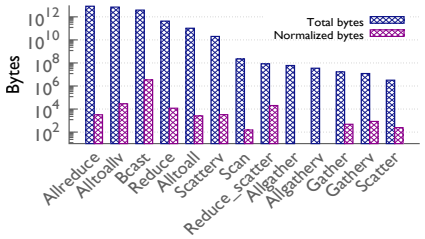

Fig. 13: MPI collectives total bytes (normalized bytes) used on Cetus

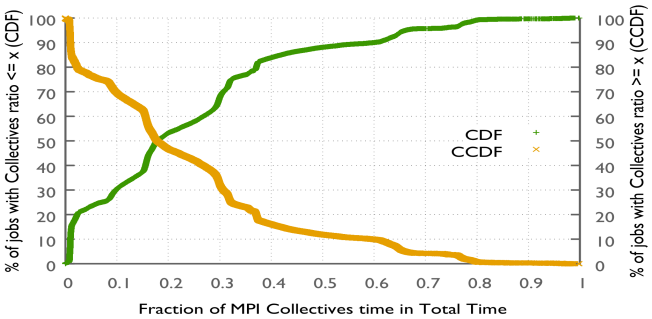

Fig. 14: Collectives fraction in total job runtime across the jobs on Mira

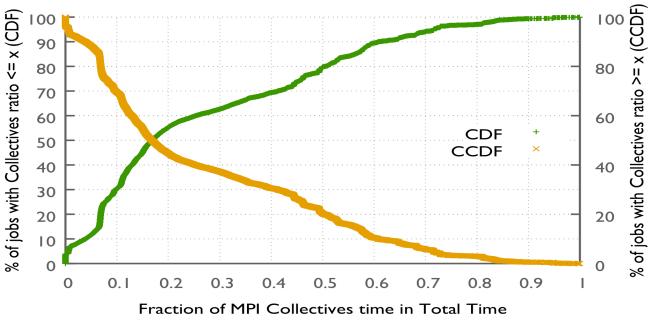

Fig. 15: Collectives fraction in total job runtime across the jobs on Cetus

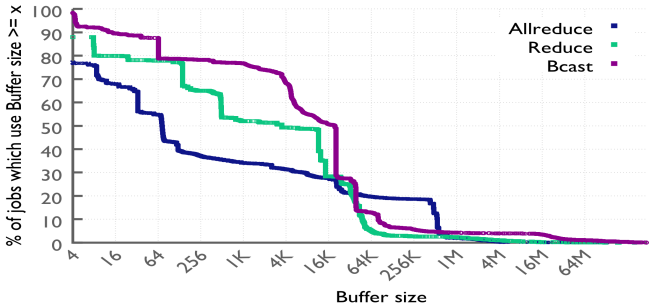

Fig. 16: MPI collectives total accumulated bytes on Mira

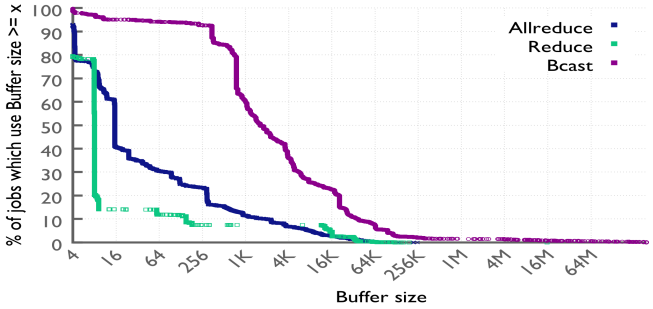

Fig. 17: MPI collectives total accumulated bytes on Cetus

with their ratios in total MPI core-hours), respectively, for the collectives on Mira.

`MPI_Allreduce` accounts for 19.4% of the total core-hours in MPI (aggregated across all the jobs) on Mira. Overall, `MPI_Allreduce`, `MPI_Bcast`, `MPI_Barrier`, `MPI_Alltoallv`, and `MPI_Reduce` are the significant collectives in terms of time (or core-hours) across all the jobs. The collectives `MPI_Exscan`, and `MPI_Scatter` are not used as often as the rest of the collectives.

`MPI_Allreduce` is the most significant collective in terms of usage and time (MPI core-hours).

Figures 11 and 12 provide the call counts and times (represented in terms of core-hours along with their ratios in total MPI core-hours) respectively for the collectives on Cetus. Here again, `MPI_Allreduce` is the most prominent collective in

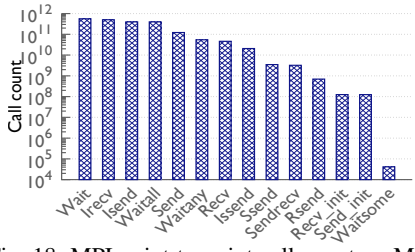

Fig. 18: MPI point-to-point call count on Mira

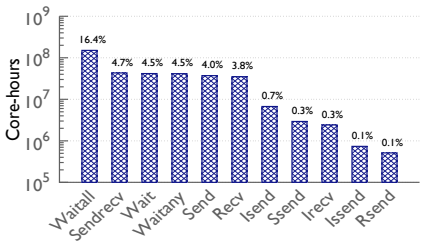

Fig. 19: MPI point-to-point time (in terms of core-hours) on Mira

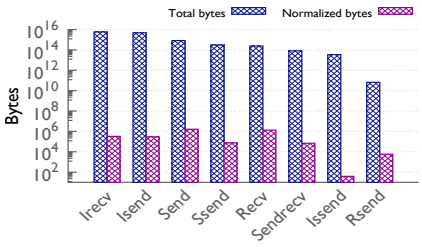

Fig. 20: MPI point-to-point total bytes (normalized bytes) used on Mira

terms of call count and time. It accounts for 30.4% of the total core-hours in MPI. The usage of other collectives is similar to that observed on Mira.

Autoperf reports the total bytes communicated through all of the monitored MPI primitives (collectives and point-to-point operations). Figure 10 shows the total bytes (accumulated across all the jobs) for both the nonvector and the vector collectives on Mira. Because of the range of sizes used, a log scale is used in the figure. While these sizes are not an exact representation of the eventual data volume on the physical network due to a collective, this should nevertheless be helpful in providing a rough estimate. The total bytes used in the individual collectives across all the Autoperf jobs can be summed up, and from that a rough estimate on the bytes used per call (total bytes divided by the call count) for each collective can be calculated. While `MPI_Bcast` and `MPI_Reduce` potentially use larger buffer sizes compared with the rest of the nonvector collectives, `MPI_Alltoallv` and `MPI_Allgatherv` potentially use larger buffer sizes among the vector collectives. Overall, `MPI_Alltoall` and `MPI_Allreduce` are responsible for high data volume on the network compared to the rest of the collectives. Figure 13 shows the total bytes used for the collectives on Cetus indicating a similar trend to that observed on Mira.

The time spent in collective operations across the jobs ordered by their respective collectives fraction with respect to the total application runtime is shown in Figures 14 and 15 for Mira and Cetus respectively. Around 10% of the jobs spend a large portion of their total runtime within the collective operations.

Autoperf records four stats (call count, total cycles, total bytes, total time) for the MPI collective and point-to-point interfaces. The total bytes recorded for a collective is the cumulative sum of the buffer sizes used across the different calls for that collective on a process. While the tool does not capture the distribution of buffer sizes used across all the calls of a collective, the potential buffer size used can be approximated using the total bytes used and the call count. While this normalization indirectly includes the job length attribute, the job size attribute is not considered, thus, the data can only be used as a representative estimate. Figure 16 shows the CCDF plot of the normalized bytes (total bytes divided by call count) used across all the jobs. The figure shows that around 40% of the jobs use MPI reduction operations with small messages (possibly buffer sizes less than 256 bytes), thus emphasizing the importance of small message collectives. Medium-sized (<=16K) reduction operations are used by 70% of the jobs. Only 5% of the jobs use `MPI_Bcast` with messages of 16M bytes or larger. Figure 17 shows the normalized bytes for the jobs on Cetus. The prominence of small MPI reductions is even greater on Cetus, with around 85% of the jobs using these. This quantitative data can be used for choosing the optimal MPI runtime configurable parameters and for configuring the collective algorithm choices.

Small sized reduction performance is most important.

2) MPI Point-to-Point Operations: MPI supports blocking, nonblocking, and persistent point-to-point operations. The point-to-point operation call counts and times (in MPI core-hours) on Mira are shown in Figures 18 and 19, respectively. The nonblocking mode of communication (`MPI_Isend`, `MPI_Irecv`, and `MPI_Wait` operations) is used more prominently than the blocking mode of communication (`MPI_Send` and `MPI_Recv`). The persistent mode of communication (`MPI_Send_init` and `MPI_Recv_init`) is used as well by some applications. The Wait operations dominate the total time in the point-to-point operations, indicating that most of the applications are programmed to potentially exploit the computation and communication overlap. The operation call counts and times (in MPI core-hours) on Cetus shown in Figures 21 and 22, respectively, show a similar behavior.

Nonblocking point-to-point operations are used more frequently than blocking or persistent operations

The bytes communicated per call (normalized bytes) with the different point-to-point operations for jobs on Mira and Cetus are shown respectively in Figures 20 and 23. While these sizes are much lower than those with the collective operations, these are significant when the corresponding data volume on the network is considered.

Figure 24 shows the time spent in point-to-point operations across the Mira jobs ordered by their respective point-to-point fraction with respect to the total time. Almost 50% of the jobs do not use point-to-point operations. When compared to Figure 14, the CCDF in Figure 24 drops more rapidly, thus, indicating that applications use more time on collectives.

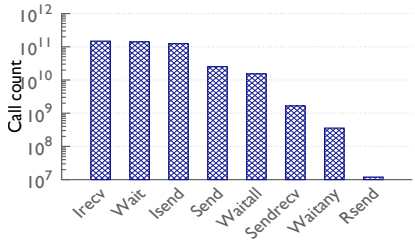

Fig. 21: MPI point-to-point call count on Cetus

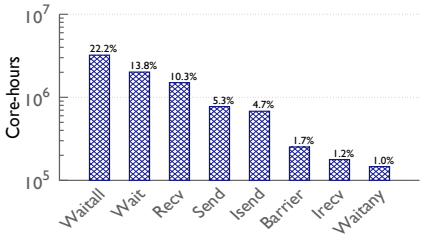

Fig. 22: MPI point-to-point time (in terms of core-hours) on Cetus

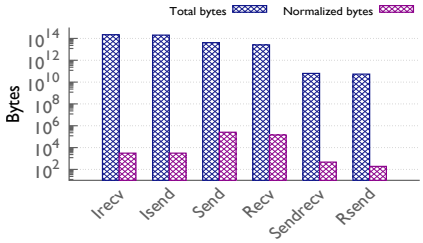

Fig. 23: MPI point-to-point total bytes (normalized bytes) used on Cetus

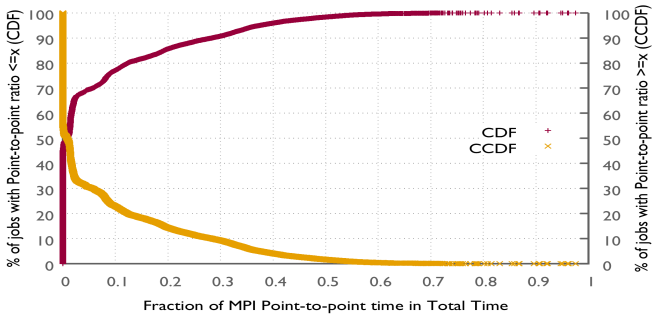

Fig. 24: Point-to-point fraction in runtime across the jobs on Mira

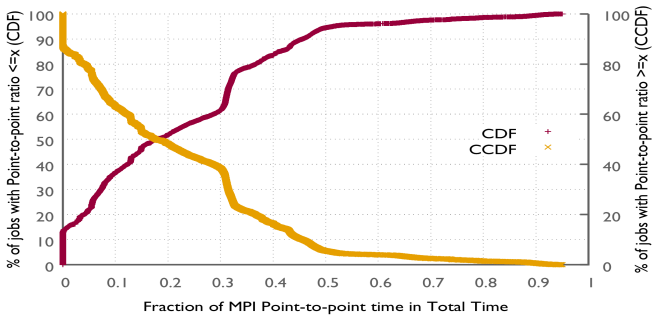

Fig. 25: Point-to-point fraction in runtime across the jobs on Cetus

The point-to-point fraction for Cetus jobs is shown in Figure 25. Here we can observe a distinctively different behavior from that noticed on Mira.

C. Applications and Their MPI Usage

Here we present an overview of the MPI usage of the applications (or unique executables). Table I shows 64 unique executables on Mira within the filtered Autoperf logs. The total 86K Autoperf logs essentially correspond to the different executions of these 64 executables. The aggregated MPI fraction across the different jobs corresponding to the 64 executables is shown in Figure 26. The multiple executions of an application could possibly have differing communication characteristics owing to the change in the run parameters. The data is arranged as per the median (across all the runs of an application) MPI fraction value in decreasing order. The range bars indicate the IQR (interquartile range) in the MPI fraction value across the different runs of an application. This range may be due to the differences in the execution parameters such as the input data set, runtime parameters and the problem size. An application may show different communication characteristics at different scales. We note that 15 of the 64 executables have an MPI fraction above 0.6, meaning they spend 60% of the runtime in MPI communication. At least half of the executables have an MPI fraction higher than 0.4. Figure 26 shows the applications labeled with their science domains. While no clear correlation patterns exist between the science domain and the MPI ratio, we can see that QMC codes have a relatively low MPI portion whereas the materials (SPH_Meso) code QDPD [8] and a quantum chemistry code VSVB [9] spend 80% of the time in MPI.

An application could have differing communication characteristics depending on the problem size, input data and execution parameters. The multi-modal distribution of runtimes (and the MPI ratios) across the runs is possible due to any of these changes, however, of these Autoperf only captures the node size parameter which by itself is insufficient to characterize the multi-modality.

Table II shows 15 unique executables within the Autoperf logs on Cetus. (Note that these are a subset of the 64 executables on Mira. However, Cetus is smaller than Mira and is used differently from Mira; hence, data from Cetus is also presented.) The total 34K Autoperf logs essentially correspond to the different executions of these 15 executables. Figure 27 shows the aggregated MPI fraction across the all Autoperf jobs on Cetus corresponding to the 15 executables along with their science domains.

Figure 28 shows the same data as in Figure 26; however, the data is now sorted based on the core-hour consumption by the executables. The highest core-hour consumer executable has an MPI fraction of 0.5. While no obvious patterns exist, we note that some of the top core-hour consumer executables have higher (greater than 0.5) MPI fraction value. Similarly, the data for Cetus applications shown in Figure A.1 also indicates that top core-hour consumer applications are dominated by the communication.

Some of the top core-hour consuming applications have high MPI time fractions.

1) Collectives and Point-to-Point Operations: Analyzed in this section are the usage patterns of the collective and point-to-point operations in the 64 applications on Mira. Figure 29 shows the breakdown of total runtime in terms of non-MPI or compute portion and communication portion (MPI collective

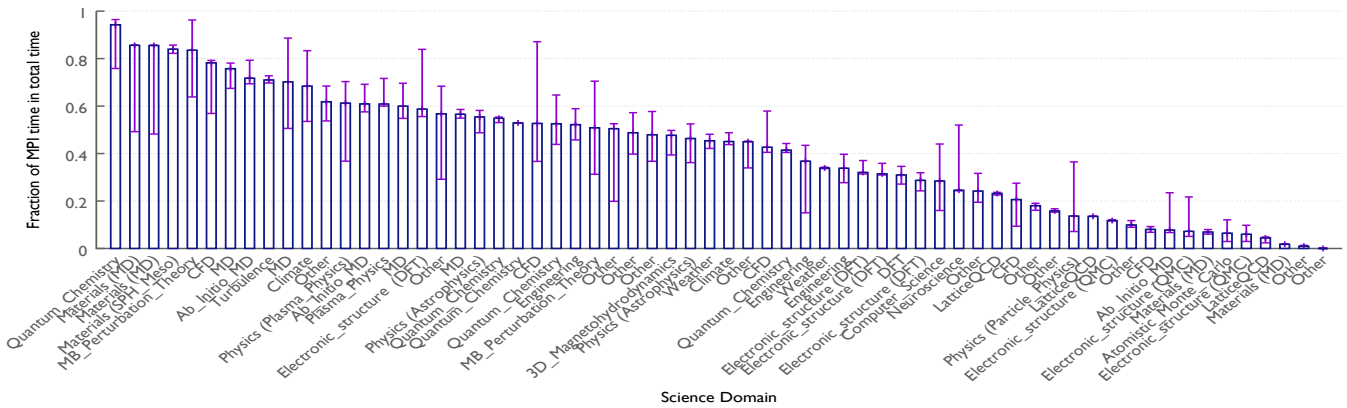

Fig. 26: MPI fraction of the applications (labeled with the corresponding science domain) on Mira

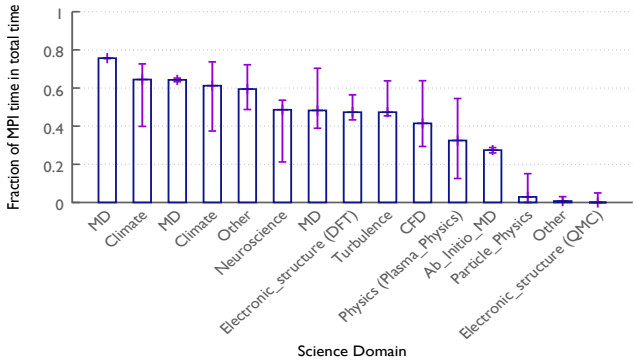

Fig. 27: MPI fraction of the applications (labeled with the corresponding science domain) on Cetus

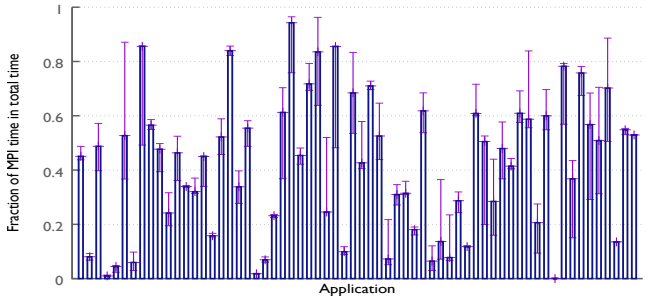

Fig. 28: MPI fraction of the applications (sorted by the core-hours recorded in Autoperf) on Mira

and point-to-point operations) for each application. The data presented in the figure is a representative breakdown of the time, considering the range of potentially differing communication characteristics across the runs of an application, it is not feasible to depict an exact characterization of all applications. However, given that all the time components referred in the figure are normalized with respect to same quantity (total application time), the data presented here is beleived to be a reasonable representation of the applications. While the usage of collectives is dominant compared with the point-to-point operations, few applications have their MPI time spent only within the collective communication. Figure 30 shows the same data with applications ordered by their compute fractions. Five applications have spent their MPI time predominantly in point-to-point operations, while more than half of the 64 applications have spent a major portion of their communication time in collectives.

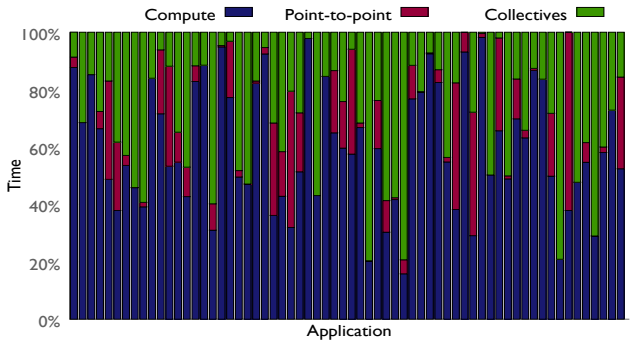

Fig. 29: MPI primitive time breakdown in communication time of the applications (ordered by core-hours) on Mira

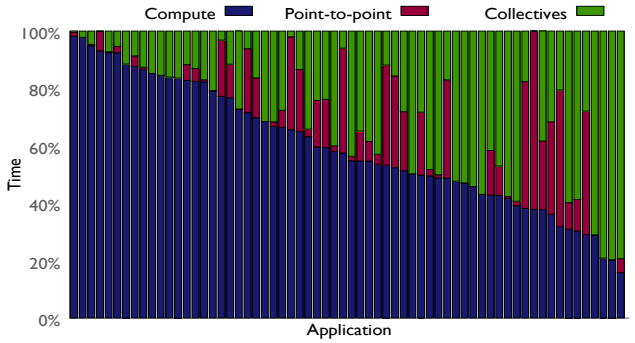

Fig. 30: MPI primitive time breakdown in communication time of the applications (in decreasing order of the compute-to-communication ratio) on Mira

Half of the total applications have spent a majority of their MPI time in collectives.

The time breakdown in compute and communication portions for applications on Cetus is provided in the appendix.

Similar to that observed on the Mira applications, the usage of collectives dominate compared to the usage of point-to-point operations.

V. Related Work

MPI profiling tools collect summary information (e.g. time, call count, message volume) for a specified set of MPI interface routines and/or MPI call stacks from a defined set of processes (some or all MPI processes). Profiling tools such as mpiP [10], IPM tool [11]–[13] and Darshan [14] provide performance information of a single MPI application execution and are mostly used by developers to optimize the code.

MPI tracing tools collect execution traces of events of interest, they typically record data for every call for each MPI function of interest. Vampir [15], DUMPI [16] and the Intel Trace Analyzer and Collector (ITAC) [17] are the commonly used tracing tools. Tracing tools provide a more accurate view of the performance at the expense of higher overhead compared to the profiling tools. They can be used to identify hotspots, load imbalances and MPI communication patterns etc. Traces can be analyzed after a run in order to infer the application behavior for understanding the computation and communication phases. The mpiP [10] tool uses a statistical analysis approach for understanding the scalability of MPI applications. Rather than capturing the MPI traces, it summarizes statistics at a per process level at runtime and aggregates the statistics at the completion of the job.

Tools such as Tau [18] and CrayPAT [19] can be configured to do both tracing and profiling. Other tools such as HPC-Toolkit [20], Score-P [21], Extrae [22] also be used for tracing and profiling. PMPI is an MPI standard profiling interface that provides an interface for profiling tools to intercept MPI calls. Many of the tools mentioned above use the PMPI interface [23].

The Autoperf tool used in this study can be viewed as a light-weight MPI profiling tool, or more precisely, referred as an MPI monitoring tool as it is meant to monitor the MPI usage of all jobs on a system and not typically used as a profiler for a specific execution. IPM tool [11]–[13] allows for MPI application profiling and workload characterization, IPM supports two modes of reporting, detailed mode (full) and concise mode (terse). The logs generated by IPM can be analyzed after a run in order to infer the application behavior for understanding the computation and communication phases. Similar to Autoperf, IPM and EZM [24] have been used in production, however, there is no known detailed report on the log analysis using these tools. IBM's HPC Toolkit (HPCT) [25] has the same set of MPI wrapper coverage and implementation as the Autoperf tool. The use of monitoring tools for automated collection of MPI usage patterns on production systems was presented in some early efforts [26], [27] and [28].

Tools have also been developed to aid in debugging complex MPI applications, Florez et al. [29] and Whalen et al. [30] presented methods to verify the correct execution of an MPI parallel program. They implemented lightweight monitoring for anomaly detection but made no attempt to accumulate log-type data for the whole application. Although the tools discussed thus far are MPI based and focused on monitoring or profiling MPI codes, monitoring tools such as LDMS (Lightweight Distributed Metric Service) [31] and HOPSA [32] have been developed to obtain systemwide resource utilization information on production HPC systems. While thes system-scale tools help monitor global events such as network congestion, however, they do not have access to the application context and thus cannot correlate application effects with network effects. The INAM [33] tool can monitor and correlate the impact of particular MPI jobs on InfiniBand networks. An overview of the application I/O behavior can be monitored by using Darshan [14].

As opposed to the standard PMPI interface approach where the MPI calls are intercepted, OpenMPI provides a low-level monitoring tool using an API based on the MPI tool standard [34]. This tool can trace the actual point-to-point communications that are issued by OpenMPI collective operations. While this tool provides information at different granularities, either as communication patterns or as message size distributions, it has not yet been evaluated in production system settings.

In summary, Autoperf provides lightweight monitoring and has been used in production for few years without any reported issues from the users. Also, it provides a sufficient coverage of the leadership-quality application set and production use of MPI on a leadership scale resource. To the best of our knowledge, ours is a first reported effort on analyzing the production application MPI usage patterns of large scale.

VI. concluding Remarks

In this paper, we study the usage characteristics of MPI on a production supercomputing system, Mira, and its corresponding development system, Cetus. We first present a lightweight profiling tool, called Autoperf, that profiles and logs summarized statistics of MPI usage for each application. Using Autoperf, we collected MPI usage profiles for around 100K jobs over a two-year period. We then present an analysis that uses these MPI usage logs and provide several key—previously unknown—insights into how scientific computing applications use MPI in production.

Acknowledgement

This research used resources of the Argonne Leadership Computing Facility, which is a U.S. Department of Energy Office of Science User Facility operated under contract DE-AC02- 06CH11357. This research was supported by the Exascale Computing Project (17-SC-20-SC), a collaborative effort of the U.S. Department of Energy Office of Science and the National Nuclear Security Administration, and by the U.S. Department of Energy, Office of Science, under Contract DE-AC02-06CH11357. We gratefully acknowledge the assistance provided by the Argonne Computational Scientists and the Darshan team.

REFERENCES

[1] "MPI: A Message-Passing Interface Standard," http://mpi-forum.org/docs/mpi-3.1/mpi31-report.pdf.

[2] G. Lakner, B. Knudson *et al.*, *IBM system Blue Gene solution: Blue Gene/Q system administration*. IBM Redbooks, 2013.

[3] "INCITE," http://www.doeleadershipcomputing.org/.

[4] "ALCC," https://science.energy.gov/ascr/facilities/accessing-ascr-facilities/alcc/.

[5] K. P. Esler, J. Kim, D. M. Ceperley, W. Purwanto, E. J. Walter, H. Krakauer, S. Zhang, P. R. C. Kent, R. G. Hennig, C. Umrigar, M. Bajdich, J. Kolorenč, L. Mitas, and A. Srinivasan, "Quantum Monte Carlo algorithms for electronic structure at the petascale: the Endstation project," *Journal of Physics: Conference Series*, vol. 125, no. 1, p. 012057, 2008. [Online]. Available: http://stacks.iop.org/1742-6596/125/i=1/a=012057

[6] L. K. Wagner, M. Bajdich, and L. Mitas, "QWalk: A quantum Monte Carlo program for electronic structure," *Journal of Computational Physics*, vol. 228, no. 9, pp. 3390–3404, 2009. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0021999109000424

[7] M. G. Martin, "MCCCS Towhee: a tool for Monte Carlo molecular simulation," *Molecular Simulation*, vol. 39, no. 14-15, pp. 1212–1222, 2013.

[8] J. S. Sims and N. Martys, "Simulation of sheared suspensions with a parallel implementation of QDPD," *Journal of Research of the National Institute of Standards and Technology*, pp. 267–277, 2014.

[9] G. Fletcher, "The variational subspace valence bond method," *The Journal of Chemical Physics*, vol. 142, p. 134112, 04 2015.

[10] J. S. Vetter and M. O. McCracken, "Statistical scalability analysis of communication operations in distributed applications," in *Proceedings of the Eighth ACM SIGPLAN Symposium on Principles and Practices of Parallel Programming*, ser. PPoPP '01. New York, NY, USA: ACM, 2001, pp. 123–132. [Online]. Available: http://doi.acm.org/10.1145/379539.379590

[11] W. N. J, W. Pfeiffer, and A. Snavely, "Characterizing parallel scaling of scientific applications using IPM," in *The 10th LCI International Conference on High-Performance Clustered Computing*. Citeseer, 2009, pp. 10–12.

[12] K. Fürlinger, N. J. Wright, D. Skinner, C. Klausecker, and D. Kranzlmüller, "Effective holistic performance measurement at petascale using IPM," in *Competence in High Performance Computing 2010*, C. Bischof, H.-G. Hegering, W. E. Nagel, and G. Wittum, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 15–26.

[13] X. Aguilar, E. Laure, and K. Fürlinger, "Online performance data introspection with IPM," in *2013 IEEE 10th International Conference on High Performance Computing and Communications and 2013 IEEE International Conference on Embedded and Ubiquitous Computing*, Nov. 2013, pp. 728–734.

[14] P. Carns, R. Latham, R. Ross, K. Iskra, S. Lang, and K. Riley, "24/7 characterization of petascale I/O workloads," in *2009 IEEE International Conference on Cluster Computing and Workshops*, Aug. 2009, pp. 1–10.

[15] A. Knüpfer, H. Brunst, J. Doleschal, M. Jurenz, M. Lieber, H. Mickler, M. S. Müller, and W. E. Nagel, "The Vampir performance analysis toolset," in *Tools for High Performance Computing*, M. Resch, R. Keller, V. Himmler, B. Krammer, and A. Schulz, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 139–155.

[16] H. Adalsteinsson, S. Cranford, D. A. Evensky, J. P. Kenny, J. Mayo, A. Pinar, and C. L. Janssen, "A simulator for large-scale parallel computer architectures," *Int. J. Distrib. Syst. Technol.*, vol. 1, no. 2, pp. 57–73, Apr. 2010. [Online]. Available: http://dx.doi.org/10.4018/jdst.2010040104

[17] R. Asbury and M. Wrinn, "MPI tuning with Intel/spl copy/ Trace Analyzer and Intel/spl copy/ Trace Collector," in *2004 IEEE International Conference on Cluster Computing (IEEE Cat. No.04EX935)*, Sep. 2004, pp. 4–.

[18] S. S. Shende and A. D. Malony, "The tau parallel performance system," *Int. J. High Perform. Comput. Appl.*, vol. 20, no. 2, pp. 287–311, May 2006. [Online]. Available: http://dx.doi.org/10.1177/1094342006064482

[19] *Using Cray Performance Measurement and Analysis Tools*. Cray, 2014, p. no. S–2376–622. [Online]. Available: http://docs.cray.com/books/S-2376-622/S-2376-622.pdf

[20] A. L., B. S., F. M., K. M., M. G., M. J., and T. N. R., "HPCTOOLKIT: tools for performance analysis of optimized parallel programs," *Concurrency and Computation: Practice and Experience*, vol. 22, no. 6, pp. 685–701.

[21] A. Knüpfer, C. Rössel, D. an Mey, S. Biersdorff, K. Diethelm, D. Eschweiler, M. Geimer, M. Gerndt, D. Lorenz, A. Malony, W. E. Nagel, Y. Oleynik, P. Philippen, P. Saviankou, D. Schmidl, S. Shende, R. Tschüter, M. Wagner, B. Wesarg, and F. Wolf, "Score-P: A joint performance measurement run-time infrastructure for periscope,scalasca, TAU, and vampir," in *Tools for High Performance Computing 2011*, H. Brunst, M. S. Müller, W. E. Nagel, and M. M. Resch, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 79–91.

[22] H. Gelabert and G. Sánchez, "Extrae user guide manual for version 2.2. 0," *Barcelona Supercomputing Center (B. Sc.)*, 2011.

[23] B. Mohr, *PMPI Tools*. Boston, MA: Springer US, 2011, pp. 1570–1575.

[24] R. A. Ballance and J. Cook, "Monitoring MPI programs for performance characterization and management control," in *Proceedings of the 2010 ACM Symposium on Applied Computing*, ser. SAC '10. New York, NY, USA: ACM, 2010, pp. 2305–2310. [Online]. Available: http://doi.acm.org/10.1145/1774088.1774566

[25] IBM, "IBM HPC toolkit (HPCT)," https://researcher.watson.ibm.com/researcher/files/us-hfwen/mpt_manual.pdf, 2012.

[26] R. Rabenseifner, *Automatic MPI Counter Profiling of All Users: First Results on a CRAY T3E 900-512*, Atlanta, GA, USA, 1999, pp. 77–85.

[27] R. Rabenseifner, P. Gottschling, W. E. Nagel, and S. Seidl, *Effective Performance Problem Detection of MPI Programs on MPP Systems: from the Global View to the Details*. Delft, Netherlands: Published by Imperial College Press and Distributed by World Scientific Publishing Co., 1999, pp. 647–655.

[28] R. Rabenseifner, "Automatic profiling of MPI applications with hardware performance counters," in *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, J. Dongarra, E. Luque, and T. Margalef, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 1999, pp. 35–42.

[29] G. Florez, Z. Liu, S. M. Bridges, A. Skjellum, and R. B. Vaughn, "Lightweight monitoring of MPI programs in real time: research articles," *Concurr. Comput. : Pract. Exper.*, vol. 17, no. 13, pp. 1547–1578, Nov. 2005. [Online]. Available: http://dx.doi.org/10.1002/cpe.v17:13

[30] S. Whalen, S. Peisert, and M. Bishop, "Multiclass classification of distributed memory parallel computations," *Pattern Recognition Letters*, vol. 34, no. 3, pp. 322–329, 2013. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0167865512003376

[31] A. Agelastos, B. Allan, J. Brandt, A. Gentile, S. Lefantzi, S. Monk, J. Ogden, M. Rajan, and J. Stevenson, "Toward rapid understanding of production HPC applications and systems," in *2015 IEEE International Conference on Cluster Computing*, Sep. 2015, pp. 464–473.

[32] B. Mohr, V. V. Voevodin, J. Giménez, E. Hagersten, A. Knüpfer, D. A. Nikitenko, M. Nilsson, H. Servat, A. A. Shah, F. Winkler, F. Wolf, and I. Zhukov, "The HOPSA workflow and tools," in *Parallel Tools Workshop*, 2012.

[33] H. Subramoni, A. M. Augustine, M. Arnold, J. Perkins, X. Lu, K. Hamidouche, and D. K. Panda, "INAM2: InfiniBand network analysis and monitoring with MPI," in *High Performance Computing*, J. M. Kunkel, P. Balaji, and J. Dongarra, Eds. Cham: Springer International Publishing, 2016, pp. 300–320.

[34] G. Bosilca, C. Foyer, E. Jeannot, G. Mercier, and G. Papauré, "Online Dynamic Monitoring of MPI Communications: Scientific User and Developper Guide," Inria Bordeaux Sud-Ouest, Research Report RR-9038, Mar. 2017. [Online]. Available: https://hal.inria.fr/hal-01485243

APPENDIX A
ADDENDUM: PERFORMANCE CHARACTERIZATION OF MPI ON A PRODUCTION SUPERCOMPUTER

We provide additional details on the monitoring tool in this appendix. Also, we provide analysis on the other aspects of MPI usage such as MPI-IO and the hybrid MPI.

A. Limitations of Autoperf

Although Autoperf is commissioned in production to capture MPI usage data for all the jobs, not all the jobs would have an Autoperf log. To determine the reasons for missing coverage of Autoperf for some executables, we used multiple sources, including surveying the users with a questionnaire and parsing through the logs of Tracklibs (a tool linked by default with all executables that logs the set of libraries linked). The reasons gathered are provided below.

- Codes not using MPI. Some applications, such as those corresponding to lattice QCD (such as MILC, CPS, and Chroma) use the BG/Q SPI low level communication API directly for efficient communication, instead of relying on MPI.
- Executables built with other conflicting profiling libraries. For example, BGPM, TAU, and HPCTW would not have Autoperf linked with them, since autoperf profiles the hardware counter-based data by using BGPM.
- Executables that are not linked with autoperf: autoperf is linked by default only for the XL and GCC compiler wrappers. Executables build with CLANG compilers wrappers, ex. some builds of QMCPACK, do not have autoperf linked automatically.
- Executables that do not call `MPI_Finalize`. These might not necessarily be erroneous applications. In some cases, applications simply write occasional checkpoints and continue execution until their job terminates, and the next job simply continues from the last checkpoint. An example application that falls in this category is Nek5000.

B. MPI Usage

1) MPI Thread Execution Environment on Mira: The MPI execution environment is initialized by using `MPI_Init` or `MPI_Init_Thread`. The valid MPI thread environment options are `MPI_THREAD_SINGLE`, `MPI_THREAD_FUNNELED`, `MPI_THREAD_SERIALIZED`, and `MPI_THREAD_MULTIPLE`. Table A.1 shows the use of these modes across the jobs and the applications. `MPI_THREAD_SINGLE` is the predominantly used option, with 62% of the jobs using this option. In terms of core-hour percentage, `MPI_THREAD_FUNNELED` is the next prodomminently used option. While around 30% of jobs use the `MPI_THREAD_MULTIPLE` option, they essentially belong to the three specific executables. Although this data is representative, we note that users may have incorrectly indicated `MPI_THREAD_SINGLE` when they actually meant `MPI_THREAD_FUNNELED` or `MPI_THREAD_SERIALIZED`.

TABLE A.1: MPI thread modes used in MPI communicator creation for jobs on Mira

MPI Thread Mode	Job Count	Core-Hours	Executables
`MPI_THREAD_SINGLE`	62.2%	69.7%	51
`MPI_THREAD_FUNNELED`	6.2%	15.7%	10
`MPI_THREAD_SERIALIZED`	0.6%	7.1%	4
`MPI_THREAD_MULTIPLE`	30.8%	7.3%	3

2) MPI Node-Level Parallelism on Mira: The compute nodes on Mira have a PowerPC A2 1600 MHz processor containing 16 cores, each capable of 4-way hardware threading. Hence, potentially 64 MPI processes can run on a compute node. Table A.2 shows usage of on-node MPI parallelism across all the Autoperf jobs on Mira. Whereas 60% of the jobs use one or two processes per node, the rest of the jobs primarily use either 4-way or 16-way MPI parallelism. This information can be helpful to center operators in defining the most suitable default environment settings that would be applicable broadly to a large fraction of the jobs.

TABLE A.2: MPI processes per node for jobs on Mira

MPI Processes per Node	Job Count	Core-Hours
1	38.1%	22.5%
2	22.6%	16.9%
4	16 %	9.2%
8	2.4%	20.0%
16	17.5%	22.7%
32	1 %	7.7%
64	1.9%	0.7%

3) MPI-IO usage on Mira: The Darshan tool [14] which records the IO interface usage patterns is enabled by default on Mira and has been designed to work in conjunction with Autoperf. Darshan captures MPI-IO routines using the PMPI interface.

The Table A.3 shows the MPI-IO usage statistics across all the Filtered Autoperf jobs that have a Darshan log file. Around 83.7% of the Filtered Autoperf jobs have a Darshan log. Collective MPI file write is the most prominently used MPI-IO interface. Darshan does not distinguish between the file pointer based (ex. `MPI_File_read`) and the explicit offset based (ex. `MPI_File_read_at`) file operations.

TABLE A.3: MPI-IO interface usage counts across all Filtered jobs on Mira

MPI-IO Interface	Usage Count
`MPI_File_iread[_at]`	0
`MPI_File_iwrite[_at]`	0
`MPI_File_open` *[Collective]*	5e+09
`MPI_File_open` *[Independent]*	4e+05
`MPI_File_read[_at]`	2e+09
`MPI_File_read[_at]_all`	4e+05
`MPI_File_read[_at]_all_begin`	0
`MPI_File_set_view`	9e+10
`MPI_File_write[_at]`	1e+08
`MPI_File_write[_at]_all`	7e+10
`MPI_File_write[_at]_all_begin`	0

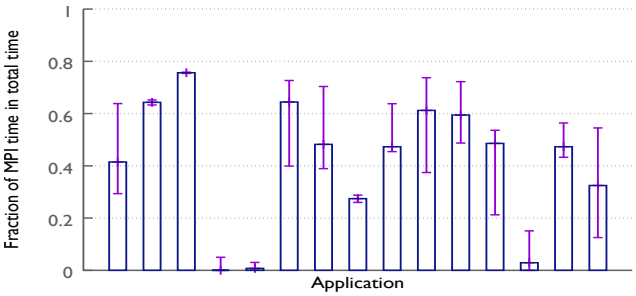

Fig. A.1: MPI fraction of the applications (sorted by the core-hours recorded in Autoperf) on Cetus

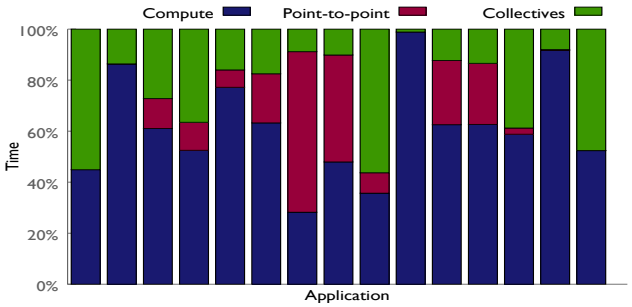

Fig. A.2: MPI primitive time breakdown in communication time of the applications (ordered by core-hours) on Cetus

4) Applications and Their MPI Usage on Cetus: Figure A.2 shows the time breakdown in compute and communication (collective and point-to-point operations) portions on Cetus. Similar to that observed on the Mira applications, the usage of collectives dominate compared to the usage of point-to-point operations.

APPENDIX B
ARTIFACT DESCRIPTION: PERFORMANCE CHARACTERIZATION OF MPI ON A PRODUCTION SUPERCOMPUTER

A. Abstract

This artifact contains the source code for the Autoperf tool that is used in this study as a lightweight MPI monitoring tool, a sample log file format as recorded by Autoperf, and the SPARK-based parallel analysis tool used to efficiently process the huge (100K) set of log files. The instructions and the tools provided here would be helpful to commission Autoperf on any production system and to make similar observations that were presented in this paper.

B. Description

1) Checklist (artifact metainformation):

- **Tools:** Autoperf, tool to process the control system log files, SPARK-based parallel tool to process the Autoperf logs
- **Compilation:** MPI monitoring portion of Autoperf, which can be built with any MPI-based compiler
- **Software:** SPARK
- **Compilation:** MPI compiler
- **Runtime environment:** Linux/Unix-based OS
- **Hardware:** BG/Q system Mira; Autoperf is based on PMPI and hence is portable to any Linux/Unix-based system with MPI
- **Execution:** See respective run scripts
- **Output:** Log files from Autoperf, summary statistics from the Analysis tool
- **Experiment workflow:** Download source code, compile source code, commission the tool on a production system using either modules or softenv type of package managers or make it part of the default MPI compiler wrapper, parse the output files, and analyze them using SPARK-based tool.
- **Publicly available?:** Yes

2) How delivered: The Autoperf tool and various other processing scripts used in this study are accessible at https://repocafe.cels.anl.gov/repos/autoperf.

3) Hardware dependencies: Autoperf log data is collected on Argonne's Mira and Cetus systems; both systems are IBM Blue Gene/Q machines using the IBM's CNK OS on the compute nodes. The control system data that is used in this study is specific to a BG/Q system. Similar infrastructure for the Cray machines is the Cray ALPS (Application Level Placement Scheduler). The Autoperf used on Mira and Cetus in production monitors both the hardware performance counters and the MPI usage data. While this version is not portable to a machine other than BG/Q, the MPI monitoring portion of the Autoperf is based on PMPI interface and hence is portable to other systems.

4) Software dependencies: The Autoperf library tool was built on Mira and Cetus using the BG/Q default compiler suite. Since Autoperf's hardware counters monitoring is performed using the BGPM (BG/Q performance monitoring) interface, Autoperf cannot be linked with executables that link BGPM for profiling and debugging reasons. Also, since the MPI monitoring portion is based on PMPI interface, executables that are linked with any PMPI-based profiling tool cannot link the Autoperf library.

5) Datasets: The control system log data used in this study is accessible at https://reports.alcf.anl.gov/data/index.html, specifically the "TASK_HISTORY" and "DIM_JOB_COMPOSITE" datasets for Mira for the two years 2016 and 2017 are used in this study. These files contain the details on all the control system (runjobs) that were run during the two-year time period. An anonymized version of the Autoperf log data is accessible at https://reports.alcf.anl.gov/data/index.html. An Autoperf log file is a single simple text file containing MPI usage data from four processes from a run. The size of a log file is on the order of 36 kilobytes, and whole log data for the two years (2016 and 2017) is in the order of 113 gigabytes. The SPARK-based tool was able to process this whole data on an Intel Xeon system in 15 minutes.

An Autoperf log file contains the four stats (call count, total cycles, total bytes, total time) for the following MPI functions: `MPI_Comm_size`, `MPI_Comm_rank`, `MPI_Send`, `MPI_Ssend`, `MPI_Rsend`, `MPI_Bsend`, `MPI_Isend`, `MPI_Issend`, `MPI_Irsend`, `MPI_Ibsend`, `MPI_Send_init`, `MPI_Ssend_init`, `MPI_Rsend_init`, `MPI_Bsend_init`, `MPI_Recv_init`, `MPI_Recv`, `MPI_Irecv`, `MPI_Sendrecv`, `MPI_Sendrecv_replace`, `MPI_Buffer_attach`, `MPI_Buffer_detach`, `MPI_Probe`, `MPI_Iprobe`, `MPI_Test`, `MPI_Testany`, `MPI_Testall`, `MPI_Testsome`, `MPI_Wait`, `MPI_Waitany`, `MPI_Waitall`, `MPI_Waitsome`, `MPI_Start`, `MPI_Startall`, `MPI_Bcast`, `MPI_Barrier`, `MPI_Gather`, `MPI_Gatherv`, `MPI_Scatter`, `MPI_Scatterv`, `MPI_Scan`, `MPI_Allgather`, `MPI_Allgatherv`, `MPI_Reduce`, `MPI_Allreduce`, `MPI_Reduce_scatter`, `MPI_Alltoall`, `MPI_Alltoallv`, `MPI_AlltoallW` and `MPI_Exscan`.

The MPI topology related calls were not monitored in this version of the tool, however, the future version of the tool is being designed to capture all the new MPI interfaces as well. The methodology used by the tool is portable and can be adapted to other platforms, we are now deploying an improved version of Autoperf on Theta.

Similar to IPM, Autoperf also records HPM (processor) related performance data as well, though those portions of the logs are not presented here.

As mentioned, Autoperf does not capture codes such as Nek5000, GFMC that use a checkpoint-restart mode of execution, however, we have ideas (using mmap or coredump) to cover these codes in the future.

C. Installation

Build instructions for building Autoperf and the SPARK-based analysis tool will be provided at the github site. Since the tools are tested primarily on Mira and Cetus, which are BG/Q systems, the build commands provided give examples that should be adapted if they are to be used on other systems.

D. Experiment Workflow

- Build Autoperf on a given production system to monitor MPI usage.
- Choose the file system path where the Autoperf log files would be written, and set the proper file permissions.
- Commission Autoperf in production to capture log data from the jobs.
- Process the log data using SPARK-based tool to generate statistics that can be used to gain insights from the data.

E. Evaluation and Expected Results

The expected results from Autoperf are the generation of log files containing the MPI usage summary. The logs can be then be processed in the analysis phase to gain insights as presented in this paper. All in the log data and the Analysis would help provide insights for the following:

- MPI developers, to better align future feature roadmaps considering the needs of production applications
- Supercomputing facility operators, to optimize resource usage by employing better job coscheduling strategies
- Procurement team, to influence the future system design and enable better match between applications and the system characteristics
- Users, to look for opportunities to optimize their parallel codes