Exploring the MPI Tool Information Interface: Features and Capabilities

Abstract

The latest version of the MPI Standard, MPI 3.0, includes a new interface, the MPI Tools Information Interface (MPI_T), which provides tools with access to MPI internal performance and configuration information. In combination with the complementary and widely used profiling interface, PMPI, it gives tools access to a wide range of information in an MPI implementation independent way.

In this paper, we focus on the new functionality offered by MPI_T and present two new tools to exploit this new interface by providing users with new insights about the execution behavior of their codes: VARLIST allows users to query and document the MPI environment and GYAN provides profiling information using internal MPI performance variables. Together, these tools provide users with new capabilities in a highly portable way that previously required in-depth knowledge of individual MPI implementations, and demonstrate the advantages of MPI_T. In our case studies, we demonstrate how MPI_T enables both MPI library and application developers to study the impact of an MPI library's runtime settings and implementation specific behaviors on the performance of applications.

Index Terms

MPI, MPI_T, tools interface, performance counters

1 INTRODUCTION

From its inception, the Message Passing Interface (MPI) Standard [1] provided portable support for tools through the MPI profiling interface, PMPI. PMPI offers a portable mechanism for tools to intercept MPI calls and to wrap the actual execution of MPI routines with profiling code. This approach was widely successful and has been used to implement a wide range of both profiling tools, like mpiP [2] or IPM [3], and tracing tools, like OTFtrace [4] and Score-P [5]. Additionally, the interface found a wide use beyond the original intended purpose of performance profiling, e.g., in correctness tools like MUST [6] or DAMPI [7] or for application support, e.g., by implementing virtualization of resources by transparently partitioning `MPI_COMM_WORLD` [8] or to remap MPI processes at startup.

Overall, the definition of PMPI in the standard has been the fundamental enabler for a rich set of portable tools, unlike with any other parallel programming model. However, PMPI focuses solely on capturing the interaction between the application and the MPI library; it does not allow insights into what is happening inside the MPI library, which can have performance critical implications. While previous approaches, such as PERUSE [9], [10], have attempted to address this shortcoming, they have failed in the standardization process, leaving users to rely on implementation specific hacks to extract such information.

However, in recent years, the MPI Forum developed a new interface for standardizing access to internal MPI library information, the MPI Tool Information Interface (MPI_T), which was adopted into the MPI 3.0 Standard [11]. One of the key differences between MPI_T and previous approaches is that MPI_T does not make any explicit assumptions about what information an implementation will provide nor demands certain information to be delivered since it may not be available in a particular implementation. Instead, it allows each implementation to decide what information to expose and then provides an interface for users to query what information is available.

MPI_T exposes internal MPI library information in the form of *variables*, typed buffers maintained and updated by the MPI library, which can be read and in some cases written. MPI_T

offers two groups of variables: performance variables, which provide information about internal MPI performance information analogously to hardware counters for processor performance; and control variables, which expose MPI configuration information for users to both document the exact runtime environment of their codes and to adjust configuration settings, e.g., for performance tuning purposes.

In this paper, we describe two publicly-available tools, which are, to the best of our knowledge, the first to exploit the MPI_T interface [12]. VARLIST queries the set of available performance and control variables and can be used to automatically extract and store configuration information. GYAN is a light-weight profiler that captures the performance information exposed by the performance interface of MPI_T. In particular, we make the following contributions:

- A user's introduction to the recently ratified MPI Tool Information Interface, MPI_T;
- VARLIST, a tool that helps users document their runtime environment;
- GYAN, the first profiler exploiting MPI_T information;
- Case studies showing information MPI_T can capture and how the information can be used.

Together, these two tools show the flexibility and versatility of the new interface and demonstrate the new opportunities users can gain from tools built on top of MPI_T.

The remainder of the paper is organized as follows: Section 2 introduces the MPI_T interface and provides a quick tutorial for users; Sections 3 and 4 present the usage of VARLIST and the design of the GYAN profiler; Section 5 shows case studies using VARLIST and GYAN on benchmarks and selected real applications; Section 6 provides our experience with working with the MPI_T interface; Section 7 details related work; and Section 8 concludes with a discussion of future options for tools built on top of MPI_T.

2 THE MPI_T INTERFACE

The PMPI interface enables standardized and platform independent access for tools; however, it limits users to observing only the interactions between the application and the MPI implementation. The newly defined MPI tools information interface addresses this shortcoming by providing an interface to access and control the internals of an MPI implementation.

2.1 Basic Concepts

All information within MPI_T is managed through named performance and control variables, each representing a typed buffer that contains internal MPI information. Since MPI_T does not explicitly state any required variables, the interface offers a query mechanism to discover the variables available in the MPI library. Users of the interface can first query the number of available performance or control variables N, and then iterate over the complete space of variables referenced by indices 0 to $N-1$. Using this index, the user can query comprehensive metadata information, including the name and the type of a variable together with an optional textual description. Once a user has identified a variable of interest, he or she can use that index to generate a handle to the variable, and then use it to read from and (in some cases) write to the variable. During the handle creation process, variables can also be bound to a particular MPI object, such as a communicator or Remote Memory Access (RMA) window, which allows users to specialize a variable to a particular object.

Control variables can be used to discover and define the behavior of MPI implementations. Examples include the definition of protocols, specification of eager limits, or selections of collective communication algorithms. The interface offers both read and write access (if implemented by the MPI library). The write functionality can be used by applications to configure themselves or even by auto-tuning systems [13].

Performance variables represent performance critical information from within the library, such as Unexpected Message Queue (UMQ) lengths or memory consumption of the MPI library. The functionality is similar to that of control variables, but performance variables require an additional

indirection: each variable needs to be part of a session and performance data will be relative to this session only. This enables the use of multiple simultaneous tools by providing proper isolation. Once a handle is allocated, users can start and stop as well as read and reset performance variables, which allows for the easy implementation of calipers. This functionality is similar to the well known PAPI interface [14] used for hardware counters.

Depending on the MPI implementation, MPI_T potentially exposes a large number of variables and users need the ability to categorize them. MPI_T offers two concepts for this purpose: *verbosity* and *categories*. The verbosity is an integer value that is returned by the metadata query call mentioned above and describes the "importance" of a variable (ranging from "this is an important variable designed for the end user" to "this is a detail variable intended only for MPI library developers"). Beyond this, categories enable a hierarchical grouping of variables referring to similar concepts, resources or performance problems (e.g., all variables related to communication or all variables related to communication protocol configuration). Categories, as with variables, are not predetermined, but offered by the MPI implementation and can be queried by the user.

2.2 Comparison with PERUSE

Before the inception of MPI_T, there was another concerted effort to extract information about the internals of MPI implementations: PERUSE [9], [10]. PERUSE was an international endeavor, developed by parties in government laboratories, universities, and industry. Although it was never adopted into the MPI Standard, the partially completed specification serves as a wealth of documentation of the kinds of information helpful for understanding application behavior and performance.

In PERUSE, MPI implementations use callbacks to notify registered tools or applications when certain internal MPI events occur. The interface defines events of interest and also allows MPI implementations to register their own implementation-specific events. Examples of events defined in PERUSE include "message activation" and "message transfer initiation" that refer to the times when MPI starts processing a message request and when it actually begins the data transfer, respectively. PERUSE defines a large number of events related to message transfers and message queues, and before the effort was ended, participants had begun to develop event definitions for collective communications, MPI I/O, one-sided communication, MPI object naming, and dynamic process creation. As part of the PERUSE specification, PERUSE-compliant MPI implementations are required to support all PERUSE functions and data types, but if a particular defined event does not directly correlate to internal MPI implementation mechanisms or would incur undue overhead to support, implementors are encouraged to simply not supply that event.

Despite the efforts in defining the PERUSE specification, it was not adopted into the MPI Standard. The main argument against PERUSE was the definition of MPI internal events that could be a potential mismatch for some MPI implementations. Although implementations were free to omit support for variables at will, there was concern that in the quest to be competitive for procurement bids, the MPI implementations would need to support events that didn't make sense or incurred high overhead. Additionally, there were arguments against the callback-based design. Some felt that the use of callbacks essentially dictated PERUSE implementation choices, while others held concerns about compatibility of callbacks with Fortran programs.

MPI_T was designed in response to the criticism against PERUSE. Tool developers and performance analysts still desired internal information from MPI libraries [15], [16], but realized defining events that satisfied all MPI implementations was a herculean task. Thus, the MPI_T interface does not define any variables, but allows each implementation to provide only what makes sense for that particular implementation. While this does pose a burden on tool developers and analysts, the general feeling is that it is better to have information that comes with challenges in interpretation than to have no information at all.

The primary differences between PERUSE and MPI_T are in the definition of what internal MPI information is provided and the mechanism for providing the information. As stated before, PERUSE defines events to represent MPI internal processes in its specification, while MPI_T leaves

the definition of the exposed information entirely up to the MPI implementation. The mechanisms for providing the information for each interface differs. In PERUSE, tools are notified of event occurrences via callbacks. However, in MPI_T tools explicitly query the MPI implementation for information on-demand.

Despite these differences, PERUSE and MPI_T have commonalities. For one, both interfaces support layering of performance tools, meaning multiple tools can utilize the interface concurrently in the same application. This functionality is important since typically each tool only extracts a particular subset of the available information. By combining information from multiple tools, analysts can acquire a full picture of application behavior [17]. Secondly, both MPI_T and PERUSE provide a query interface to determine what information will be provided by the MPI implementation at runtime. This is required for both interfaces since with MPI_T a tool cannot know *a priori* what variables will be provided, and with PERUSE, a tool won't know which of the events are supported.

3 QUERYING MPI_T WITH VARLIST

The MPI_T information interface specification does not prescribe any particular sets of variables. Instead, it offers users a mechanism to query all existing variables, performance or control, together with metadata and descriptions. Users therefore need a way to list all available variables in order to extract what variables can be used on a particular implementation.

Our tool VARLIST fulfills this requirement. It lists all available control and performance variables offered by a particular MPI library and thereby basically creates an automatic self-documentation feature for the MPI_T capabilities of a particular MPI library implementation. Users can request a short overview of variables, a list with all metadata and complete descriptions, a list limited to variables of a particular verbosity level, or a list of variables offered before or after `MPI_Init` (which may influence variable availability as well as writability). With the VARLIST tool, a user can list either control and performance variables separately at different verbosity levels, or can list all information along with long descriptions provided by the particular MPI implementation.

One particular use case for this tool, aside from helping users to decide what they can profile or extract with tools like GYAN, is to help in documenting the environment and runtime settings for a particular run. By running VARLIST at job start and dumping all control variables and their current values, users can get a snapshot of the MPI runtime settings for that run. This enables them to identify possible problems, help with debugging of runtime problems, or to improve experiment and execution reproducibility.

4 GYAN: PROFILING USING MPI_T

GYAN is Sanskrit for "knowledge". We developed GYAN to offer knowledge about the internal performance of an MPI implementation. GYAN uses the PMPI interface to profile MPI_T performance variables over the course of an MPI job execution. Since performance variables will vary in name and in number depending on the MPI implementation used and possibly across different versions of the same MPI library, GYAN provides two ways to select which performance variables to monitor. The user of GYAN can select a specific performance variable using the environment variable `MPIT_VAR_TO_TRACE`, or simply let the tool monitor all performance variables exposed by the MPI implementation being used that are not tied to a specific MPI object. The latter alleviates the potential for mistakenly setting a performance variable that is not exposed by that particular MPI implementation or for setting the wrong one that does not provide useful information for the targeted performance problem, but comes with a possibly larger overhead.

Algorithm 1 presents the steps taken by GYAN in accessing and reporting the status of performance variables after an MPI application is run:

- A user sets the variable `MPIT_VAR_TO_TRACE` with specific performance variable names to monitor; alternatively, if this variable is left unset, GYAN reports the status of all performance variables internal to the MPI implementation;

Algorithm 1 Algorithmic steps of GYAN.

Input: Set `MPIT_VAR_TO_TRACE` (optional)

function MPI_INIT ▷ Intercepts `MPI_Init` call of an application
  Call `PMPI_Init` and `MPI_T_init_thread`
  Get number of available variables using `MPI_T_pvar_get_num`
  Create a session using `MPI_T_pvar_session_create`
  Allocate handles for all performance variables
  if `MPIT_VAR_TO_TRACE` is not set **then**
    Set all performance variables to the watch list
  else
    Set variables in `MPIT_VAR_TO_TRACE` to the watch list
  for each variable in the watch list **do**
    Initialize data structures for reading values
    Start session using `MPI_T_pvar_start`
function PVAR_READ_ALL ▷ Reads current status of all watched variables
  for each variable in the watch list **do**
    Read the current status of this variable's current session using `MPI_T_pvar_read`
    Copy values to a pre-allocated buffer
function MPI_FINALIZE ▷ Intercepts `MPI_Finalize` call of an application
  Read current values of all watched variables using `pvar_read_all`
  Compute sum, minimum, and maximum of all variables across all ranks
  Rank 0 reports the output
  Stop watching variables
  Cleanup all data structures initialized for reading values
  Call `PMPI_Barrier` to ensure all processes reached this point
  Close the MPI_T library with a call to `MPI_T_finalize`
  Finalize the MPI library with a call to `PMPI_Finalize`

- Once an MPI application starts running, GYAN intercepts the call to `MPI_Init` and initializes the library with a call to `PMPI_Init`;
- GYAN then creates a session, filters all variables that are not tied to a particular MPI object, attaches performance variables within the MPI_T interface to this session, reads the variables' initial values using `MPI_T_pvar_read` and starts monitoring the selected variables by calling `MPI_T_pvar_start` for each;
- At the end of the application's execution, GYAN intercepts `MPI_Finalize`, reads the values of all monitored performance variables again by calling `MPI_T_pvar_read`, reports the difference between the values and the corresponding values read during initialization in an easily readable format; and cleans up the library before finalizing MPI.

GYAN continuously monitors programs from start to finish to report a cumulative status of all performance variables. This is partly because the two MPI implementations we worked with only supported a single session at the time of our experiments. Once MPI implementations provide multi-session support, GYAN can be extended to report more advanced and fine-grained performance measures. The usage of GYAN is straightforward since it has no dependence on any other library or package. As any other PMPI tool, it can be either preloaded via `LD_PRELOAD` or linked with an application, and can collect data without perturbing the performance of an application.

5 EXPERIMENTS

In the following, we present three case studies showing the kind of information VARLIST and GYAN can provide and how this can help users in application execution and optimization. In the first case

VARIABLE	DESCRIPTION
posted_recvq_match	Counts how many times the queue for receiving expected messages is read.
unexpected_recvq_match	Counts how many times the queue for receiving unexpected messages is read.
progress_poll_count	Counts how many times the application polls the progress of a communication. The higher the value, the more CPU time is spent in polling.
mem_allocated_level	Gives the instantaneous memory usage by the library in bytes.
mem_allocated_highwater	Gives the maximum number of bytes ever allocated by the MPI library at a given process for the duration of the application.
coll_bcast_binom	Counts how many of the MPI broadcast collective calls use the Binomial algorithm during an application run.
num_shmem_coll	Counts how many of the collective communication calls are using shared memory.
coll_bcast_shmem	Counts how many of the MPI broadcast communication calls are shared memory based collectives.

TABLE 1: Performance variables in MVAPICH2-2.0a that are presented in Section 5.3 and 5.4.

study, we demonstrate how VARLIST can be used to document relevant settings for a given MPI application run. In the second and the third case study, we use GYAN with two MPI applications to report the status of available performance variables as we scale problem size and process count.

5.1 Experimental Setup

We executed our experiments on Cab, one of the Tri-Lab Capacity Clusters (TLCC) at Lawrence Livermore National Laboratory with $1,296$ nodes and a peak performance of around 0.5 PFlop/s. Nodes in this Linux cluster are connected using a QDR Infiniband interconnect and each have dual socket 2.6 GHz Intel Sandy Bridge processors with a total of 16 cores as well as 32 GB of memory. We compiled both codes with the GNU compiler $4.4.7$ at optimization level -O2. We used both OpenMPI-1.9a1r31420 and MVAPICH2-2.0a for the experiments as noted. Table 1 presents names and descriptions of a subset of performance variables implemented in MVAPICH2-2.0a and discussed in Sections 5.3 and 5.4[1].

In our experiments, we ran the BT benchmark from the NAS Parallel Benchmark suite [18], and NEK5000 [19]. BT solves a synthetic system of nonlinear partial differential equations using the block tridiagonal algorithm. NEK5000 is a Gordon Bell prize-winning computational fluid dynamics code that captures the thermal-hydraulics phase inside nuclear reactors by simulating unsteady incompressible fluid flow with thermal and passive scalar transport.

We compared application runtime with no MPI_T monitoring to that with GYAN monitoring all MPI_T performance variables. The average run times for these two cases were within 0.5% from each other, which is well below the noise threshold. Hence, our experiments show that monitoring MPI_T performance variables with GYAN does not incur significant overhead. In addition, VARLIST takes only 1 second to list all control and performance variables of MVAPICH2-2.0a.

5.2 Case Study I: Use of Varlist

Most MPI implementations provide a set of control variables that can be set through either command line or environment variables to alter the runtime behavior of the MPI library. These variables can control aspects of the MPI library that directly impact application performance: e.g., to decide when an implementation should switch from eager to rendezvous protocols. However, typical MPI implementations provide users with many configuration options (often with little to no documentation), making it hard for users to know: a) which configuration options even exist in a particular implementation; b) what they mean, what behavior they influence and how; and c) what settings are used for a particular option for a given run. Using MPI_T, VARLIST can provide the documentation for questions a) and b) automatically by running `varlist -c -l`. This prints all available configuration variables including any description offered by the MPI library. Furthermore, for c), users can include VARLIST into their job scripts, to automatically document all settings for

1. Note that variable names have been shortened to make the table more reader friendly.

VARLIST Output		RUNTIME
VAR. NAME	VALUE	
btl_self_eager_limit btl_sm_eager_limit	131072 4096	5.06
btl_self_eager_limit btl_sm_eager_limit	10 4096	5.08
btl_self_eager_limit btl_sm_eager_limit	10 10	5.12

TABLE 2: Impact of runtime configurations on the performance of NEK5000.

a given run and study the impact of these changes during postmortem analysis. The latter is particularly useful when done systematically over many/all runs. In case of performance changes, the output of VARLIST can be used to detect changes in the behavior of the MPI library to previous executions, which is normally hard to detect.

To demonstrate how changes of control variables can impact the performance of an application, we select two control variables from OpenMPI-1.9a1r31420 that control eager limits for short messages. Since OpenMPI-1.9a1r31420 does not currently support writing to control variables programmatically, we change these variables by setting the corresponding environment variables. The variable `btl_self_eager_limit` in OpenMPI-1.9a1r31420 sets the maximum size of short messages, and `btl_sm_eager_limit` sets the same for eager protocols using shared memory. We randomly change their values to 10, and use VARLIST to verify these changes, and then record the runtime of NEK5000. Practically, good settings for these control variables will depend on the configuration of the machine and the communication behavior of the application in question. Decreasing these values to 10 indicate that MPI will use rendezvous protocols for messages that are greater than 10 bytes in size.

Table 2 presents the control variables that are modified by setting environment variables, the output from VARLIST after environment variables were updated, and runtimes of the corresponding application runs. The first row presents the runtime of NEK5000 with the default setting of eager limits. After setting control variables, we again run VARLIST to document the current status of control variables, and record the runtime. Similarly, using VARLIST for multiple configurations, users can see and understand the impact of different settings on the performance of applications.

5.3 Case Study II: BT from NPB

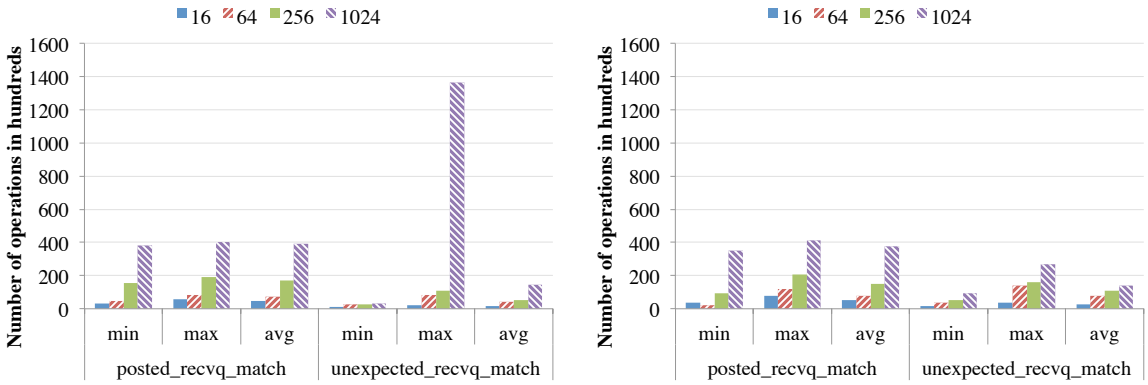
We use GYAN on the BT benchmark with problem sizes "C" and "D". The problem size "C" uses $162 \times 162 \times 162$ elements and "D" uses $408 \times 408 \times 408$ elements. The objective of this experiment is to compare the MPI_T performance variables at scale, and across different problem sizes.

Figures 1a and 1b present performance variables that count how many times incoming messages were matched with entries in the posted receive queue or were taken from the unexpected message queue when `MPI_Recv` was called. The observations that can be made from these two figures are:

First, for the same problem size, increasing the number of MPI processes increases the volume of communication among processes, as shown by the increasing number of total receives processed in either queue. This is expected and a typical scaling behavior.

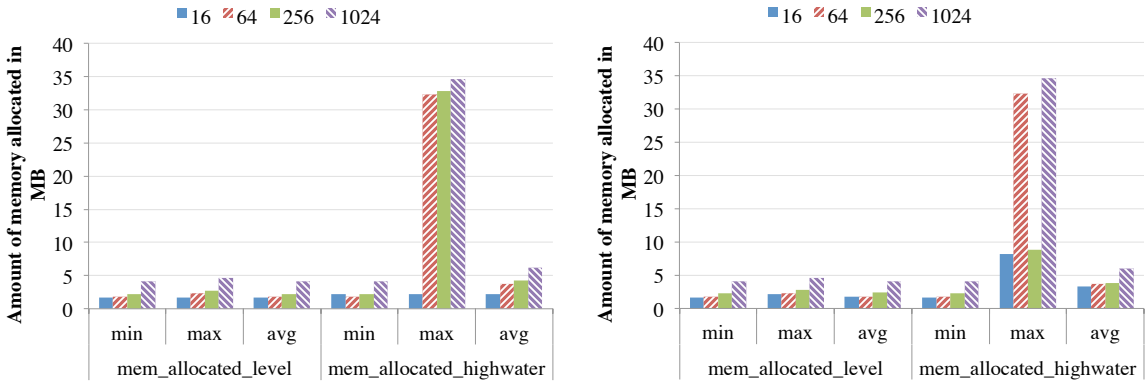
Second, Figure 1a shows that increasing the number of MPI processes to 1024 caused at least one (or a small number of) MPI process to complete operations through the unexpected receive queue significantly more than others, since the average value for this variable is much smaller than the maximum value. Such an observation can instigate further investigation in cases of performance imbalance in applications to identify the (small) set of processes exhibiting this behavior.

Third, the average number of times the unexpected receive queue is used to receive messages increases with the increase in problem size for the same number of processes, while the number of receives completed through the posted receives queue stays the same. This indicates that the increased traffic caused by scaling the problem no longer arrives at the target nodes in time for a receive to be posted already, which has the potential to cause more overhead. By tracing this back



(a) Problem size $162 \times 162 \times 162$ (b) Problem size $408 \times 408 \times 408$

Fig. 1: GYAN reports the number of times incoming messages were matched by the MVAPICH2-2.0a library while running BT for two problem sizes with increasing processor counts.



(a) Problem size $162 \times 162 \times 162$ (b) Problem size $408 \times 408 \times 408$

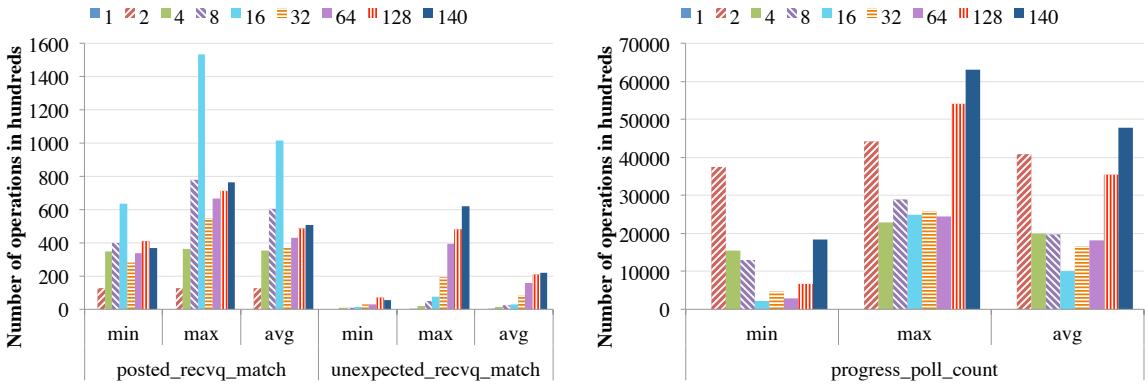
Fig. 2: GYAN reports the number of bytes of memory allocated by the MVAPICH2-2.0a library while running BT for two problem sizes with increasing processor counts.

to the MPI operations in the source code, we can now identify the operations that are responsible for this observed behavior and match them up with the intended message patterns with the goal of rearranging the messaging schedule.

Further, this information can expose a potential bottleneck in the `MPI_Recv` implementation. The unexpected queue must be searched for a matching message each time an `MPI_Recv` is posted. If no match is found, it is then enqueued in the posted receive queue. Since MPI implementations need to set aside memory for buffering unexpected messages, applications with a high number of such messages can quickly exhaust this buffer space and cause performance loss for applications.

Note, that this is a good example of how the information from MPI_T exposed by GYAN can be helpful. GYAN extract information from MPI_T to not only alert users of situations like the bottleneck described above, but can also be deployed by MPI implementors. For implementors, GYAN can be used to list the utilization of the relevant variables for a new implementation version, e.g., to compare how optimization changed the message queue utilization, and with that enable implementors to reason about their code's performance.

Figures 2a and 2b present performance variables that count the bytes of memory allocated by the



(a) Receive queue statistics

(b) Insights into communication progress

Fig. 3: GYAN results for NEK5000 using MVAPICH2-2.0a with increasing processor counts.

library for two different problem sizes (providing both current values and high water marks). The observation that can be made from these pictures is that since GYAN reads the value of memory currently allocated when `MPI_Init` and `MPI_Finalize` are called, this value is the same across both of these problem sizes, showing the minimal footprint the MPI library is using. For the high water mark, though, there is significant disparity between the maximum and the average memory footprint across processes ($5\times$ for 1024 processes). Such imbalance in memory usage across processes may lead to poor performance of the entire application if it results in memory thrashing, as well as to reduced problem sizes that can be computed if applications are designed to allocate the same amount of application data for each process and hence are limited by the process with the least amount of available memory. Further investigation needs to be done in order to identify whether this problem is application dependent or pertains solely to the MPI implementation.

5.4 Case Study III: NEK5000

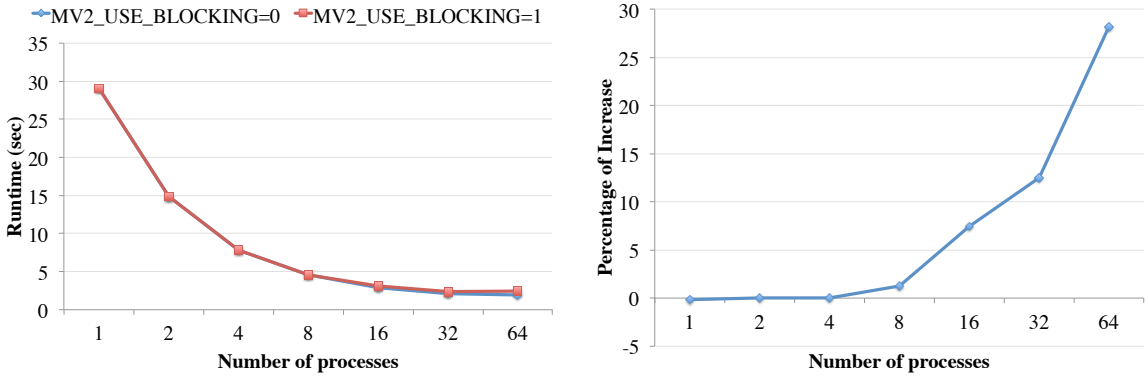
In this experiment, we collect all 25 performance variables exposed by MVAPICH2-2.0a for the production application NEK5000 and present those variables that have nonzero values. The problem size is moderately large with 140 elements in total simulating the thermal hydraulics phase of a nuclear reactor. This is a practical problem size used in the nuclear reactor community.

Figure 3 presents performance variables pertaining to four different categories – the receive queues, communication progress, collective calls, and the amount of memory allocated by the library. Our observations are presented below.

First, from Figure 3a we can observe that the number of times messages are matched with receives on the posted message queue increases significantly until 16 processes, and then it drops. Since each node on the machine has 16 cores, for cases $1-16$, all processes are packed on the same node and communicate only using shared memory, which apparently increases the likelihood that receive operations are executed and posted in time before the corresponding message arrives.

Second, from Figure 3b we can observe that the number of times the progress of communications being polled is high. This value is directly proportional to the amount of CPU time spent in polling the progress of communications. A high value therefore indicates significant wait times and with that potential load imbalances, especially if there is a large gap between min and max values. We can see the latter behavior for executions of more than 4 processes, which should trigger a deeper analysis of load balance properties, in particular for codes that almost zero poll times.

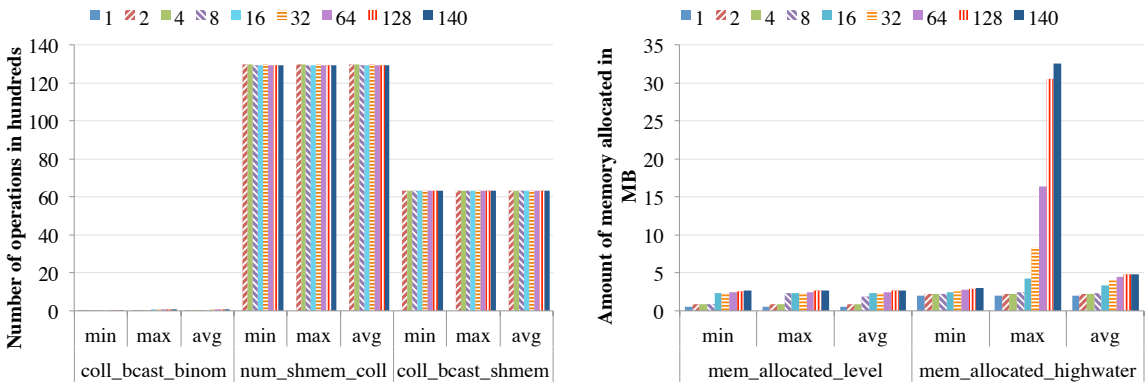
MPI implementations often provide configuration variables to enable blocking mode of communication (in the case of MVAPICH2-2.0a, it is `MV2_USE_BLOCKING`). This can reduce the number of cycles wasted polling for messages and can open up performance improvements for other processes,



(a) Absolute runtimes

(b) Execution time increase using blocking receives

Fig. 4: Comparing the performance of blocking and non-blocking receives.



(a) Profiling internals of collective routines

(b) Profiling memory usage

Fig. 5: GYAN results for NEK5000 using MVAPICH2-2.0a with increasing processor counts.

e.g., in oversubscribed nodes or if communication is handled in multithreaded scenarios, or help reduce power and energy consumption. However, our experiment (presented in Figures 4a, and 4b) shows that the blocking mode of communication comes at the price of a increased runtime of NEK5000 by 28% for the 64 process case. An application can use GYAN to read these internal performance variables in order to make smart trade-off decisions.

Third, Figure 5a shows that the behavior of MPI collective calls remains consistent as the application scales up. This indicates that, as NEK5000 scales up, the number of binomial and shared memory broadcasts, and the number of shared memory collectives in general, stays stable and at a 2:1 ratio. This information reported by GYAN shows a global consistent use of collectives that should allow the application to scale.

Fourth, Figure 5b shows that memory utilization, both instantaneous and highwater marks, increases with scale. Also, the maximum amount of memory allocated by any process for the duration of the application is significantly higher than the average value. Since this value is high for both BT and NEK5000, it may be worth investigating if there is any systematic explanation for the phenomenon in the MVAPICH2-2.0a library itself.

6 DISCUSSION

In the previous section, we showed how GYAN and VARLIST can extract internal MPI information and expose it to users, and how this information can be useful for documenting and understanding application performance. The standardized interface of MPI_T made the actual development of these tools straightforward. However, as we stated previously, MPI_T does not prescribe any variables that should be exposed by MPI implementations; the variables, their definitions, and number are completely determined by the MPI implementors. This factor complicates tool development because tools cannot rely on the existence of a particular variable across different MPI implementations, across implementation versions, or even across different runs, e.g., in the case that different hardware is used. This means that it is extremely challenging for tools to automatically reason about the collected information, and human interpretation will be needed in many cases. Nonetheless, despite this complexity, tool developers were in full support of standardizing MPI_T because it provides the only standardized mechanism for accessing MPI internal data.

In our study, we found that the performance variables exposed by early adopters of MPI_T are among the most important variables from the literature [15], [20], [21], and include variables that describe message queues and memory usage. We imagine that in the future, application and tool developers can suggest more performance variables based on their experience in using the MPI_T interface that can be used to pinpoint the cause of performance differences due to the internal implementation of an MPI library. It is unclear whether there will ever be a common set of defined variables that satisfy all or even most MPI implementations; thus we expect tool developers to continue to need to write flexible software when using MPI_T.

7 RELATED WORK

Over the years, there have been several efforts to expose internal MPI information to tools and application developers. The MPI implementors have done this primarily by adopting early versions of tools interfaces for this purpose, namely PERUSE and MPI_T. The Open MPI implementation includes support for both PERUSE [22] and MPI_T, while the implementations based on MPICH currently support MPI_T. The MVAPICH team recently published work showing details of their support for MPI_T, and discussed and evaluated their design decisions and how it impacted measurement overheads [20].

Tools developers have worked to gather internal MPI information by directly modifying MPI implementations and related libraries. This was done because there was no officially adopted interface at the time of the work. For example, Brightwell et al. instrumented the MPICH library to gather message queue statistics [15], [21]. Other researchers uncovered internal information about MPI-I/O operations by instrumenting MPICH2, ROMIO, and PVFS [16]. They were able to show underlying low-level details about collective MPI-I/O operations that explained their performance.

With the development of interfaces like PERUSE and MPI_T, tools adopted their use to extract MPI internal information. Some tools were developed to access PERUSE via the support in Open MPI [23], [24]. These were focused on accessing message queue information to provide details like message queue visualizations and metrics including message queue lengths and message queue search times. In the Periscope Tuning Framework, the developers are using MPI_T to change control variables in order to automatically tune applications and recommend best topologies [25]. However, as of this writing, this is a work in progress and not yet available.

8 CONCLUSION

The MPI Tool Information Interface, MPI_T, introduced in MPI 3.0, is a new standardized mechanism for tools to gather information about MPI applications. It complements the existing MPI profiling interface, PMPI, and offers access to both internal performance information and configuration variables. It is based on the concept of typed variables that can be queried, read, and set.

In this paper, we present a first set of tools using MPI_T for accessing, listing, and monitoring both the control and the performance variables in any MPI implementation. The VARLIST tool

currently reads names of all control and performance variables exposed by any MPI implementation. The GYAN tool monitors performance variables during the execution phase of an application, and generates statistics for each measured variable. Together the tools provide simplified and MPI implementation independent access to the internal states of MPI implementations, and enable expert users to further enhance the performance of their applications by understanding and ultimately controlling the MPI runtime behavior. In fact, our case studies showed examples of the implications of changing internal MPI settings and how application execution characteristics can change internal MPI functioning. For instance, we found that changes to the "eager limit" setting could affect overall runtime, and that the size of the "unexpected message queue" can change with the number of processes in the job.

We expect the MPI_T interface to be as successful as the existing PMPI interface and the two tools presented here to be only the first in a long line of performance, debugging, and correctness tools exploiting MPI_T.

REFERENCES

[1] "MPI Standard 1.0," http://www.mpi-forum.org/docs/docs.html.

[2] J. Vetter and C. Chambreau, "mpiP: Lightweight, Scalable MPI Profiling." [Online]. Available: http://mpip.sourceforge.net

[3] D. Skinner, "Performance monitoring of parallel scientific applications," Lawrence Berkeley National Laboratory, Tech. Rep. LBNL-5503, 2005.

[4] A. Knüpfer, R. Brendel, H. Brunst, H. Mix, and W. E. Nagel, "Introducing the Open Trace Format (OTF)," in *Computational Science–ICCS 2006*. Springer, 2006, pp. 526–533. [Online]. Available: http://mpip.sourceforge.net

[5] D. an Mey, S. Biersdorf, C. Bischof, K. Diethelm, D. Eschweiler, M. Gerndt, A. Knüpfer, D. Lorenz, A. Malony, W. E. Nagel *et al.*, "Score-P: A Unified Performance Measurement System for Petascale Applications," in *Competence in High Performance Computing 2010*. Springer, 2012, pp. 85–97.

[6] T. Hilbrich, M. Schulz, B. R. de Supinski, and M. S. Müller, "MUST: A Scalable Approach to Runtime Error Detection in MPI Programs," in *Tools for High Performance Computing 2009*. Springer, 2010, pp. 53–66.

[7] A. Vo, S. Aananthakrishnan, G. Gopalakrishnan, B. R. de Supinski, M. Schulz, and G. Bronevetsky, "A Scalable and Distributed Dynamic Formal Verifier for MPI Programs," in *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*. IEEE, 2010, pp. 1–10.

[8] T. Gamblin, "CRAM: A tool to run many small MPI jobs inside of one large MPI job," https://github.com/scalability-llnl/cram.

[9] R. Dimitrov, A. Skjellum, T. Jones, B. de Supinski, R. Brightwell, C. Janssen, and M. Nochumson, "PERUSE: An MPI Performance Revealing Extensions Interface," *Sixth IBM System Scientific Computing User Group*, 2002.

[10] T. Jones, R. Dimitrov *et al.*, "MPI PERUSE: An MPI Extension for Revealing Unexposed Implementation Information," 2006.

[11] "MPI Standard 3.0," http://www.mpi-forum.org/docs/docs.html.

[12] "Scalability Team MPI Tools," https://github.com/scalability-llnl/mpi-tools.

[13] I. Compres, "On-line Application-specific Tuning with the Periscope Tuning Framework and the MPI Tools Interface," Presentation at the 2014 Petascale Tools Workshop, Madison, WI, August 2014.

[14] P. J. Mucci, S. Browne, C. Deane, and G. Ho, "PAPI: A Portable Interface to Hardware Performance Counters," in *Proc. Department of Defense HPCMP User Group Conference*, June 1999, pp. 7–10.

[15] R. Brightwell, S. Goudy, and K. Underwood, "A Preliminary Analysis of the MPI Queue Characteristics of Several Applications," in *Proceedings of the 2005 International Conference on Parallel Processing*, ser. ICPP '05. IEEE, 2005, pp. 175–183.

[16] J. Kunkel, Y. Tsujita, O. Mordvinova, and T. Ludwig, "Tracing Internal Communication in MPI and MPI-I/O," in *International Conference on Parallel and Distributed Computing, Applications and Technologies*. IEEE, Dec 2009, pp. 280–286.

[17] M. Schulz and B. R. de Supinski, "PNMPI Tools: A Whole Lot Greater Than the Sum of Their Parts," in *Proceedings of the 2007 ACM/IEEE Conference on Supercomputing (SC '07)*. ACM, Nov 2007, p. 30.

[18] "NAS Parallel Benchmarks (NPB)," https://www.nas.nasa.gov/publications/npb.html.

[19] "Nekbone: A suite of proxy application for nek5000." [Online]. Available: https://cesar.mcs.anl.gov/content/software/thermal_hydraulics

[20] R. Rajachandrasekar, J. Perkins, K. Hamidouche, M. Arnold, and D. K. Panda, "Understanding the Memory-Utilization of MPI Libraries: Challenges and Designs in Implementing the MPI_T Interface," in *Proceedings of the 21st European MPI Users' Group Meeting (EuroMPI/Asia'14)*. ACM, September 2014, p. 97.

[21] R. Brightwell, K. Pedretti, and K. Ferreira, "Instrumentation and Analysis of MPI Queue Times on the SeaStar High-Performance Network," in *Computer Communications and Networks (ICCCN '08)*, Aug 2008, pp. 1–7.

[22] R. Keller, G. Bosilca, G. Fagg, M. Resch, and J. J. Dongarra, "Implementation and Usage of the PERUSE-Interface in Open MPI," in *Proceedings, 13th European PVM/MPI Users' Group Meeting*, ser. Lecture Notes in Computer Science. Bonn, Germany: Springer-Verlag, September 2006.

[23] "Sun HPC ClusterTools User's Guide," https://docs.oracle.com/cd/E19708-01/821-1319-10/index.html.

[24] R. Keller and R. Graham, "MPI Queue Characteristics of Large-Scale Applications," in *Cray User's Group Meeitng (CUG 2010)*, 2010.

[25] M. Gerndt, K. Fürlinger, and E. Kereku, "Periscope: Advanced Techniques for Performance Analysis," in *Parallel Computing: Current & Future Issues of High-End Computing (Proceedings of ParCo)*, vol. 33, 2005, pp. 15–26.