Exploring the Capabilities of the New MPI_T Interface

Tanzima Islam, Kathryn Mohror, Martin Schulz
Lawrence Livermore National Laboratory
Livermore, CA USA
{islam3,kathryn,schulzm}@llnl.gov

ABSTRACT

The latest version of the MPI Standard, MPI 3.0, includes a new interface for tools, the MPI Tools Information Interface (MPI_T). In this paper, we focus on the new functionality and insights that users can gain from MPI_T. For this purpose, we present two new tools that are the first that exploit the new interface. 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.

1. INTRODUCTION

From its inception, the MPI Standard [1] provided portable support for tools through the MPI profiling interface, also known as PMPI. PMPI offers a portable mechanism for tools to intercept MPI calls and to wrap the actual execution of MPI routines with profiling code. Many tools have been built using this interface, including: profilers like mpiP [11]; trace collectors like OTFtrace [9] and Score-P [6]; and correctness tools like MUST [8]. Additionally, the interface has uses beyond profiling, e.g., to implement virtualization of resources by transparently partitioning `MPI_COMM_WORLD` or to remap MPI processes at startup.

Overall, PMPI 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 [7], 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 [2]. 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. 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 *variables*, typed buffers maintained and updated by the MPI library, which can be read and in some cases written. MPI_T offers two classes of such 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 and allow users to both document the exact runtime environment of their codes and to adjust configuration settings, e.g., for auto-tuning purposes.

In this paper, we describe two publicly-available tools, which are the first to exploit the MPI_T interface [5]. 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:

- An introduction to the recently ratified MPI Tool Information Interface, MPI_T, from the user perspective;
- 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 the use 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; and Section 6 concludes with a discussion of future options for tools built on top of MPI_T.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
EuroMPI/ASIA '14, September 9-12 2014, Kyoto, Japan
Copyright 2014 ACM 978-1-4503-2875-3/14/09 ...$15.00.
http://dx.doi.org/10.1145/2642769.2642781.

```
==============================
Control Variables
==============================

Found 1026 control variables
Found 1026 control variables with verbosity <= D/A-9

Variable                                              VRB   Type   Bind     Scope    Value
-------------------------------------------------------------------------------------------------
...
mpi_ddt_unpack_debug                                  U/A-3 INT    n/a      LOCAL    false
mpi_ddt_pack_debug                                    U/A-3 INT    n/a      LOCAL    false
mpi_ddt_position_debug                                U/A-3 INT    n/a      LOCAL    false
mpi_ddt_copy_debug                                    U/A-3 INT    n/a      LOCAL    false
dss_buffer_type                                       D/D-8 INT    n/a      ALL      described
dss_buffer_initial_size                               D/D-8 INT    n/a      ALL      128
dss_buffer_threshold_size                             D/D-8 INT    n/a      ALL      1024
event                                                 U/D-2 CHAR   n/a      ALL
event_base_verbose                                    D/D-8 INT    n/a      LOCAL    0
event_libevent2021_event_include                      U/A-3 CHAR   n/a      LOCAL    poll
opal_event_include                                    U/A-3 CHAR   n/a      LOCAL    poll
event_libevent2021_major_version                      D/A-9 INT    n/a      UNKNOWN  1
event_libevent2021_minor_version                      D/A-9 INT    n/a      UNKNOWN  9
event_libevent2021_release_version                    D/A-9 INT    n/a      UNKNOWN  0
mpi_param_check                                       D/A-9 INT    n/a      READONLY true
mpi_yield_when_idle                                   D/A-9 INT    n/a      READONLY false
mpi_event_tick_rate                                   D/A-9 INT    n/a      READONLY -1
mpi_show_handle_leaks                                 D/A-9 INT    n/a      READONLY true
mpi_no_free_handles                                   D/A-9 INT    n/a      READONLY false
mpi_show_mpi_alloc_mem_leaks                          D/A-9 INT    n/a      READONLY 0
mpi_show_mca_params                                   D/A-9 CHAR   n/a      READONLY
mpi_show_mca_params_file                              D/A-9 CHAR   n/a      READONLY
mpi_abort_delay                                       D/A-9 INT    n/a      READONLY 0
mpi_abort_print_stack                                 D/A-9 INT    n/a      READONLY true
...
```

(a) OpenMPI-1.9a1r31420

```
==============================
Performance Variables
==============================

Found 25 performance variables
Found 25 performance variables with verbosity <= D/A-9

Variable                                   VRB   Class   Type   Bind     R/O CNT ATM
------------------------------------------------------------------------------------
posted_recvq_length                        U/D-2 LEVEL   UINT   n/a      YES YES  NO
unexpected_recvq_length                    U/D-2 LEVEL   UINT   n/a      YES YES  NO
posted_recvq_match_attempts                U/D-2 COUNTER UNKNOW n/a       NO YES  NO
unexpected_recvq_match_attempts            U/D-2 COUNTER UNKNOW n/a       NO YES  NO
time_failed_matching_postedq               U/D-2 TIMER   DOUBLE n/a       NO YES  NO
time_matching_unexpectedq                  U/D-2 TIMER   DOUBLE n/a       NO YES  NO
unexpected_recvq_buffer_size               U/D-2 LEVEL   UNKNOW n/a      YES YES  NO
mem_allocated                              U/B-1 LEVEL   ULLONG n/a      YES YES  NO
mem_allocated                              U/B-1 HIGHWAT ULLONG n/a      YES YES  NO
mv2_progress_poll_count                    D/B-7 COUNTER ULONG  n/a       NO  NO  NO
coll_bcast_binomial                        U/B-1 COUNTER ULLONG n/a      YES YES  NO
coll_bcast_scatter_doubling_allgather      U/B-1 COUNTER ULLONG n/a      YES YES  NO
coll_bcast_scatter_ring_allgather          U/B-1 COUNTER ULLONG n/a      YES YES  NO
mv2_num_2level_comm_requests               U/D-2 COUNTER ULONG  n/a      YES YES  NO
mv2_num_2level_comm_success                U/D-2 COUNTER ULONG  n/a      YES YES  NO
mv2_num_shmem_coll_calls                   T/B-4 COUNTER ULONG  n/a      YES YES  NO
mv2_coll_bcast_binomial                    T/B-4 COUNTER ULLONG n/a      YES YES  NO
mv2_coll_bcast_scatter_doubling_allgather  T/B-4 COUNTER ULLONG n/a      YES YES  NO
mv2_coll_bcast_scatter_ring_allgather      T/B-4 COUNTER ULLONG n/a      YES YES  NO
mv2_coll_bcast_scatter_ring_allgather_shm  T/B-4 COUNTER ULLONG n/a      YES YES  NO
mv2_coll_bcast_shmem                       T/B-4 COUNTER ULLONG n/a      YES YES  NO
mv2_coll_bcast_knomial_internode           T/B-4 COUNTER ULLONG n/a      YES YES  NO
mv2_coll_bcast_knomial_intranode           T/B-4 COUNTER ULLONG n/a      YES YES  NO
mv2_coll_bcast_mcast_internode             T/B-4 COUNTER ULLONG n/a      YES YES  NO
mv2_coll_bcast_pipelined                   T/B-4 COUNTER ULLONG n/a      YES YES  NO
------------------------------------------------------------------------------------
```

(b) MVAPICH2-2.0a

Figure 1: Output from VARLIST for two different MPI implementations. Figure 1a shows a partial list of 1026 control variables found in OpenMPI-1.9a1r31420, and Figure 1b presents the complete list of performance variables found in MVAPICH2-2.0a.

2. THE MPI_T INTERFACE

The existing 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. While this capability has been used successfully by many tools, it does not provide the ability to gather performance information from within the MPI library. Further, it does not provide a standardized mechanism to read, document, and change control information. The newly defined MPI tools information interface fills both of these gaps.

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 RMA window, which allows users to specialize a variable to a particular object.

2.2 Control Variables

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.

2.3 Performance Variables

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 the easy implementation of calipers. This functionality is similar to the well known PAPI interface [10] used for hardware counters.

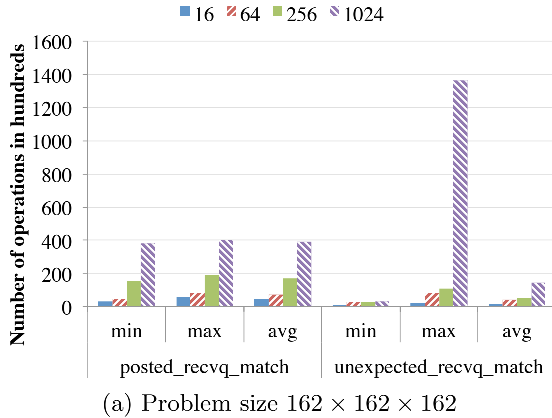
2.4 Categories and Verbosities

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 an hierarchical grouping of variables with the same meaning (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.

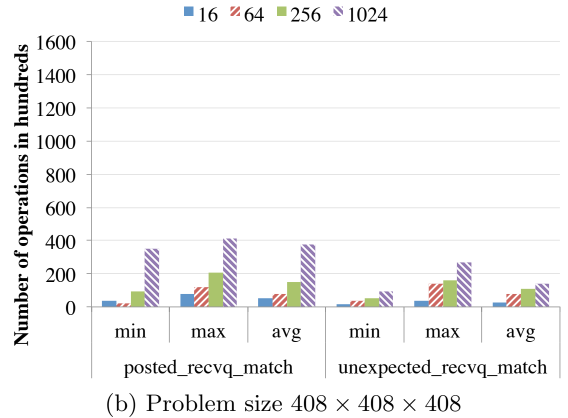
3. QUERYING MPI_T WITH VARLIST

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.

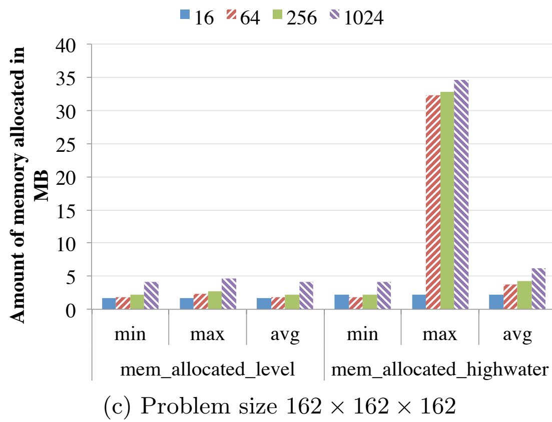
Our tool VARLIST fulfills this requirement. It lists all avail-



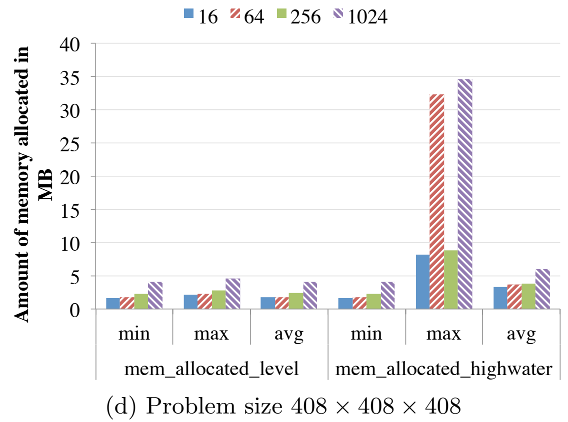
(a) Problem size $162 \times 162 \times 162$



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



(c) Problem size $162 \times 162 \times 162$



(d) Problem size $408 \times 408 \times 408$

Figure 2: Gyan results for BT using MVAPICH2-2.0a with increasing processor counts.

Options	Description
-c	List only Control Variables
-p	List only Performance Variables
-v <VL>	List up to verbosity level=[1,9]
-l	Long list with all information, including descriptions
-m	Do not call `MPI_Init` before listing variables

Table 1: Usage of Varlist

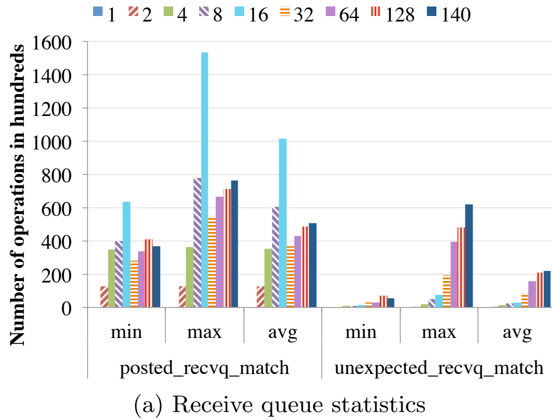
able control and performance variables offered by a particular MPI library. Users can request a short overview of variables, a list with all metadata and complete descriptions, a list limited to variables of a particular verbosity level, or a list of variables offered before or after `MPI_Init` (which may influence variable availability as well as writability). This tool is intended to give users a quick overview of the MPI_T capabilities of a particular MPI library. More importantly, by listing all control variables and their current values, users can document the runtime settings for MPI for each run, allowing them to identify possible problems or to improve experiment and execution repeatability.

Table 1 lists all options available through the Varlist tool. Figure 1a shows the output of Varlist for control variables with OpenMPI-1.9a1r31420. Values of these variables are useful in understanding how the behavior of this particular implementation of MPI is controlled and power users can influence the behavior by setting these variables. Figure 1b presents all performance variables currently implemented in MVAPICH2-2.0a.

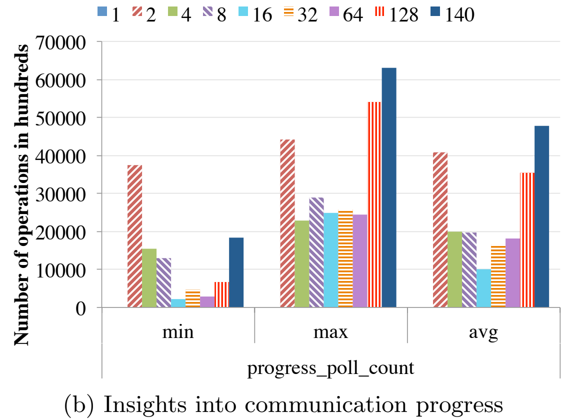
4. GYAN: PROFILING USING MPI_T

Gyan is the Sanskrit word for "knowledge". We developed Gyan to offer knowledge about the internal performance of an MPI implementation. Gyan profiles 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. The latter alleviates the potential for mistakenly setting a performance variable that is not exposed by that particular MPI implementation, but comes with a possibly larger overhead.

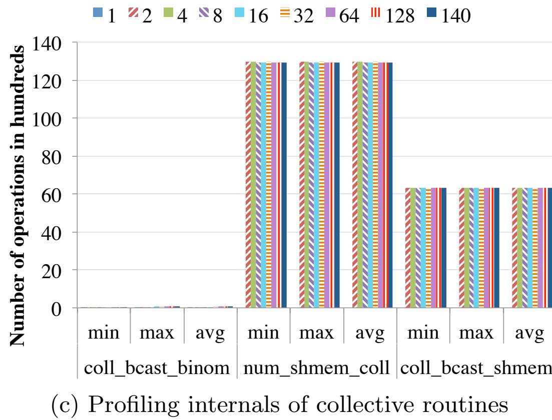
Gyan uses the PMPI interface to intercept the call to `MPI_Init`, and if no individual performance variable was specified in `MPIT_VAR_TO_TRACE`, starts monitoring all variables. Then, it intercepts the call to `MPI_Finalize`, reads in the current status of all monitored performance variables, and presents the difference between starting and ending values for the variable(s) in an easily understandable format.



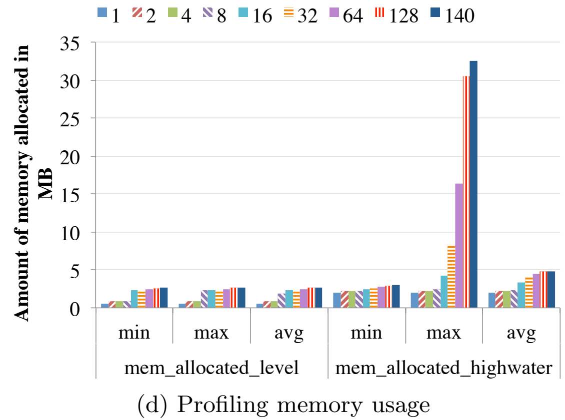
(a) Receive queue statistics



(b) Insights into communication progress



(c) Profiling internals of collective routines



(d) Profiling memory usage

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

Since GYAN has no dependence on any other library or package, the usage of GYAN is straightforward. As any other PMPI tool, it can be either preloaded via `LD_PRELOAD` or linked with an application.

5. EXPERIMENTS

Here, 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 these experiments, we ran the BT benchmark from the NAS Parallel Benchmark suite [3], and NEK5000 [4]. 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 the runtime of an application with no MPI_T monitoring to that with GYAN monitoring all MPI_T performance variables. The average run times for these two cases were at most 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.

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 the QDR Infiniband interconnect and have dual socket 2.6 GHz Intel Sandy Bridge processor, and 32 GB of memory each. 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 2 presents names and descriptions of a subset of performance variables implemented in MVAPICH2-2.0a and discussed in Sections 5.2 and 5.3[1].

```
#!/bin/bash
varlist -c -v 9
srun -n 512 -N 32 nek5000
```

Figure 4: Sample job script to automatically document MPI's runtime environment

5.1 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 directly impact application performance: e.g., to decide when an implementation should switch from eager to rendezvous protocols. However, typi-

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

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 2: Performance variables in MVAPICH2-2.0a that are presented in Section 5.2 and 5.3.

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 3: Impact of runtime configurations on the performance of NEK5000.

cal MPI implementations provide users with many configuration options (often with little documentation), making it hard for users to know a) which configuration options even exist in a particular implementation, b) what they mean, 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, as illustrated in Figure 4, to automatically document all settings for a given run and study the impact of these changes during postmortem analysis.

To demonstrate the principle and the impact changes of control variables can have, 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 3 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 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 draw the impact of different settings on the performance of applications.

5.2 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 2a and 2b 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. The first observation is that 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. The second observation, shown in Figure 2a, is that increasing the number of MPI processes to 1024 may have caused at least one MPI process to check the unexpected receive queue significantly more than other processes, 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. The third observation is that, with the increase in problem size, the average number of times the unexpected receive queue is used to receive messages increases for the same number of processes.

Figures 2c and 2d present performance variables that count how many bytes of memory were allocated by the library for two different problem sizes (providing both current values and high water marks). The first observation 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.

5.3 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 that have nonzero values. We use a moderately big and fixed problem size, that has practical use in the nuclear reactor community. It uses strong scaling with a total 140 elements. Hence, the particular problem has a fixed scaling limit of 140 processes.

Figure 3 presents performance variables pertaining to four different categories – the receive queue, communication progress, collective calls, and the amount of memory allocated by the library. 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.

From Figure 3b, we can observe that the number of times the progress of communications is polled is significantly high. This value is directly proportional to the amount of CPU time spent in polling the progress of communications. A high value indicates that other processes running on the same node were not able to utilize the CPU as effectively. MPI implementations often provide configuration variables to enable blocking mode of communication (in the case of MVAPICH2-2.0a, it is `MV2_USE_BLOCKING`). Our experiments (not presented in this paper) show that the blocking mode of communication increases the runtime of NEK5000 by 28% for the 64 process case. However, the blocking mode of communication makes CPU cycles available for other processes to use. An application can use GYAN to read these internal performance variables in order to make smart tradeoff decisions, such as being "energy-aware" by saving CPU cycles as opposed to saving runtime.

Figure 3c shows that the behavior of MPI collective calls remain consistent as the application scales up. Figure 3d 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 explanation for the phenomenon in the MVAPICH2-2.0a library itself.

6. CONCLUSION

The MPI Tool Information Interface, MPI_T, introduced with MPI 3.0, offers 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 as well as to configuration variables. It is based on the concept of typed variables that can be queried, read and 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.

7. REFERENCES

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

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

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

[4] Nekbone: A Suite of Proxy Application for NEK5000.

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

[6] 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*, pages 85–97. Springer, 2012.

[7] 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.

[8] 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*, pages 53–66. Springer, 2010.

[9] A. Knüpfer, R. Brendel, H. Brunst, H. Mix, and W. E. Nagel. Introducing the Open Trace Format (OTF). In *Computational Science–ICCS 2006*, pages 526–533. Springer, 2006.

[10] 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.

[11] J. Vetter and C. Chambreau. mpiP: Lightweight, Scalable MPI Profiling.

Acknowledgements

The MPI_T interface has been designed, prototyped, and implemented by a large group of people, most of them involved in the Working Group on Tools within the MPI forum; we thank them for their effort. This material is based upon work supported by the U.S. DOE, Office of Science, ASCR Program, and was performed by LLNL under Contract DE-AC52-07NA27344. LLNL-CONF-654091.