Implementation and Usage of the PERUSE-Interface in Open MPI

Rainer Keller[1], George Bosilca[2], Graham Fagg[2], Michael Resch[1], and Jack J. Dongarra[2]

[1] High-Performance Computing Center, University of Stuttgart,
{keller,resch}@hlrs.de
[2] Innovative Computing Laboratory, University of Tennessee
{bosilca,fagg,dongarra}@cs.utk.edu

Abstract. This paper describes the implementation, usage and experience with the MPI performance revealing extension interface (Peruse) into the Open MPI implementation. While the PMPI-interface allows timing MPI-functions through wrappers, it can not provide MPI-internal information on MPI-states and lower-level network performance. We introduce the general design criteria of the interface implementation and analyze the overhead generated by this functionality. To support performance evaluation of large-scale applications, tools for visualization are imperative. We extend the tracing library of the Paraver-toolkit to support tracing Peruse-events and show how this helps detecting performance bottlenecks. A test-suite and a real-world application are traced and visualized using Paraver.

1 Introduction

The Message Passing Interface (MPI) [7, 8] is the standard for distributed memory parallelization. Many scientific and industrial applications have been parallelized and ported on top of this parallel paradigm. From the very beginning the MPI standard offered a way for performance evaluation of all provided functions including the communication routines with the so-called Profiling-Interface (PMPI). Thereby all MPI-function calls are accessible through the prefix `PMPI_`, allowing wrapper-functions, which mark the time at entry and exit. The tracing libraries of performance analysis tools, such as Vampir[3], Paraver[5] and Tau[9] are build upon the PMPI-Interface. However, the information gathered using this interface has a limited impact, as it can only provide high level details about any communications (such as starting and ending time), rather than more interesting internal implementation and networking activities triggered by the MPI calls.

In order to know the internals of how the communication between two processes proceeds and where possible bottlenecks are located, a more in-depth and finer-grained knowledge is required than is available from the PMPI-interface level. The Peruse-interface [2], a multi-institution effort driven by LLNL which

gained larger audience at a BoF at SC2002, proposes a standard way for applications and libraries to gather this information from a Peruse-enabled MPI-library. Especially with more diverse hardware, such as multi-core chips using shared-memory and many hierarchies in large-scale clusters, this performance evaluation becomes essential for in-depth analysis.

This paper introduces an implementation of Peruse in the Open MPI [4] implementation. In section 2 we describe the general design and implementation of the Peruse-interface within Open MPI, and state the impact on communication performance degradation, while section 3 shows the performance metrics gathered. Section 4 illustrates a possible method to evaluate the communication performance by extending the `mpitrace`-library of the Paraver-toolkit. In section 5 a real-world application is traced and visualized with Paraver. Finally, the last section gives a conclusion and an outlook on future developments.

2 Design and Implementation

The Open MPI implementation uses the so-called modular component architecture (MCA) to support several component implementations offering a specific functionality [10]. In this paper, we will consider only the frameworks and components used for communication purposes, i.e., the Point-to-Point management layer (PML), the recursively named BML management layer (BML) and the Bit-transport layer (BTL). These frameworks are stacked, as may be seen in figure 1. MPI communication calls are passed on to the PML, which uses the BML to select the best possible BTL, and then passes the message (possibly in multiple fragments depending on length) to the BTL for transmission.

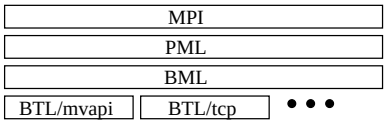

Fig. 1. Open MPI stack of frameworks and modules for communication.

The Peruse interface allows an application or performance measurement library to gather information on state-changes within the MPI library. For this, user-level callbacks have to be registered with the Peruse interface, which are subsequently invoked upon the triggering of corresponding events. The interface allows a single callback function to be registered for multiple events, as well as multiple callback functions for one event (which covers the rare instance of an application and one or more libraries wanting to gather statistics on a single event simultaneously). Peruse does not impose any particular message passing method and recommends not supporting a particular event, if this would burden or slow down the MPI implementation. The interface is portable in design, by allowing applications or performance tracing libraries to query for supported events using

defined ASCII strings. The tracing library may then register for an event, supplying a callback function, which is invoked upon triggering a particular event, e. g. `PERUSE_COMM_REQ_XFER_BEGIN` when the first data transfer of a request is scheduled. Registration then returns an event-handle. Events implemented in Open MPI are presented in sec. 3.

Prior experience with the implementation of Peruse-functionality was gained with PACX-MPI [6]. Special care was taken not to slow down the critical fast path of the Open MPI library. The actual test for an active handle and the immediate invocation of the callback function is implemented as a macro, which the preprocessor optimizes away in a default build of the library. When building with the configure parameter `--enable-peruse`, the actual test for an active handle involves at most two additional `if`-statements: whether any handles are set on this communicator and whether the particular one is set and active.

Although most of the events are pertaining to messages being sent and received, the actual calls to the callback functions are performed in the PML-layer, as it has all the necessary information regarding requests and fragments being sent. Currently, only one major PML-module exists (`ob1`), in contrast to the six major BTLs (`sm`, `tcp`, `mvapi`, `openib`, `gm`, `mx`), which would have each required modifications for every possible Peruse event.

Additionally, this initial implementation only allows a single callback function per event. As handles are stored per communicator (`PERUSE_PER_COMM`) as array of `ompi_peruse_handle_t`-pointer, allowing more callbacks per event or worse case multiple handles (instances) per event would have required iterating over all the registered and active handles in the communicator-storage, greatly increasing the overall overhead.

Cluster	cacau	strider
Processor	Dual Intel Xeon EM64T, 3.2 GHz	Dual AMD Opteron 246, 2GHz
Interconnect	Infiniband	Myrinet 2000
Interface	`mvapi-4.1.0`	`gm-2.0.8`
Compiler	Intel compiler 9.0	PGI compiler 6.1.3
Open MPI	no debug, static build	no debug, dynamic build
Native MPI	Voltaire MPIch-1.2.6	MPIch-1.2.6

Table 1. Configuration of clusters for the Peruse overhead evaluation.

For performance comparison with and without the Peruse-interface implementation, several measurements were conducted on the clusters given in table 1. We compare the latency induced by the additional overhead by using a build without any Peruse-support and two versions with Peruse-support: one without any callbacks and one with callbacks attached for all possible events. Additionally this is compared to the latency of the native MPI-implementation provided on each cluster.

Table 2 shows the measurements done with the Intel-MPI Benchmark using the zero Byte PingPong-test. The `IMB_settings.h` was changed to perform

each test for 10000 iterations with ten warm-up phases. For the native MPI, the optimized vendor's version on the cluster was used as listed in table 1.

	cacau			strider		
	native	mvapi	sm	native	gm	sm
No Peruse	4.13	4.69	1.02	7.16	7.16	1.33
Peruse, no callbacks		4.67	1.06		7.26	1.71
Peruse, no-op callbacks		4.77	1.19		7.49	1.84

Table 2. Latency (in μs) of zero-byte messages using IMB-2.3 with `PingPong`.

In comparison with the cluster's native MPI, the Open MPI's BTLs `mvapi` and `gm` only show marginal difference in latency being 1.7% and 4.6% respectively. Therefore, a much more sensitive test using the shared-memory BTL `sm` was performed. Here, one experiences a degradation in latency – but even with all 16 communication events registered, the increase is 16% and 38% respectively for the two target systems. For larger message sizes, the overhead compared to the bandwidth without any Peruse-support is shown in Fig. 2.

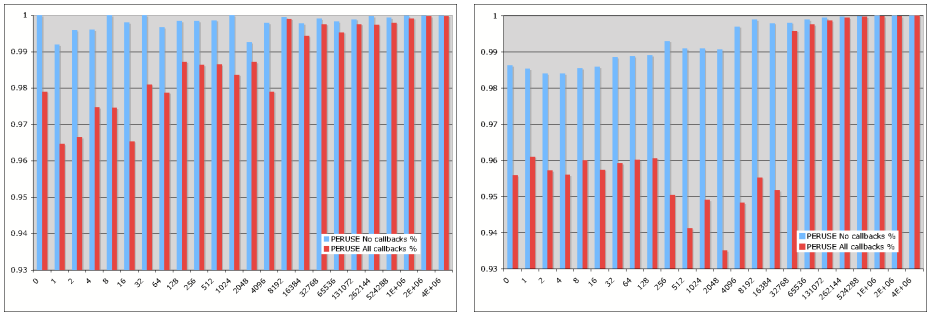

Fig. 2. Percentage of achieved bandwidth on cacau (left) and strider (right) compared to Open MPI without Peruse.

3 Performance Metrics gathered

The current implementation in Open MPI supports all events stated in the current Peruse-2.0 specification [2]. Orthogonal to the `PERUSE_COMM_REQ_XFER_BEGIN/_END` Open MPI implements the `PERUSE_COMM_REQ_XFER_CONTINUE` notifying of new fragments arriving for this request. This event is only issued in the case of long messages not using the `eager` protocol. The sequence of callbacks that may be generated on the way for sending / receiving a message are given in Fig. 3.

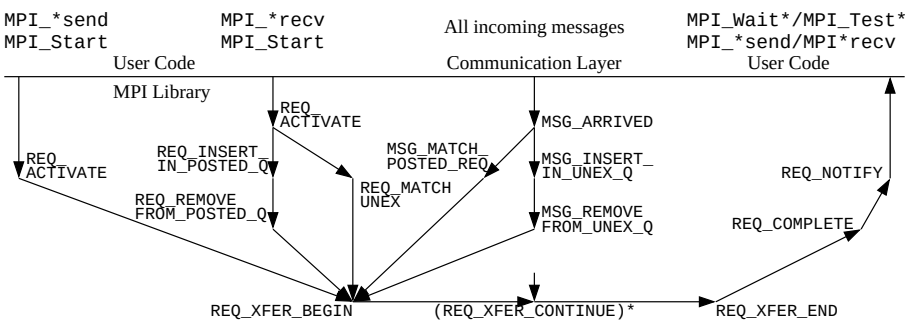

Fig. 3. Sequence of Peruse events implemented in Open MPI.

The following example shows the callback sequence when sending a message from rank zero to rank one, which nicely corresponds to figure 3. Here, we have imposed an early receiver by delaying the sender by one second. One may note the early activation of the request, searching in the unexpected receive queue, insertion into the expected receive queue, and finally the arriving message with the subsequent start of transfer of messages events (edited):

```
PERUSE_COMM_REQ_ACTIVATE at 0.00229096 count:10000 ddt:MPI_INT
PERUSE_COMM_SEARCH_UNEX_Q_BEGIN at 0.00229597 count:10000 ddt:MPI_INT
PERUSE_COMM_SEARCH_UNEX_Q_END at 0.00230002 count:10000 ddt:MPI_INT
PERUSE_COMM_REQ_INSERT_IN_POSTED_Q at 0.00230312 count:10000 ddt:MPI_INT
PERUSE_COMM_MSG_ARRIVED at 1.00425 count:0 ddt:0x4012bbc0
PERUSE_COMM_SEARCH_POSTED_Q_BEGIN at 1.00426 count:0 ddt:0x4012bbc0
PERUSE_COMM_SEARCH_POSTED_Q_END at 1.00426 count:0 ddt:0x4012bbc0
PERUSE_COMM_MSG_MATCH_POSTED_REQ at 1.00426 count:10000 ddt:MPI_INT
PERUSE_COMM_REQ_XFER_BEGIN at 1.00427 count:10000 ddt:MPI_INT
PERUSE_COMM_REQ_XFER_CONTINUE at 1.0043 count:10000 ddt:MPI_INT
   -- subsequent XFER_CONTINUEs deleted --
PERUSE_COMM_REQ_XFER_CONTINUE at 1.00452 count:10000 ddt:MPI_INT
PERUSE_COMM_REQ_XFER_END at 1.00452 count:10000 ddt:MPI_INT
PERUSE_COMM_REQ_COMPLETE at 1.00453 count:10000 ddt:MPI_INT
PERUSE_COMM_REQ_NOTIFY at 1.00453 count:10000 ddt:MPI_INT
```

Collecting the output of the callbacks from a late sender:

```
PERUSE_COMM_REQ_ACTIVATE at 1.00298 count:10000 ddt:MPI_INT
PERUSE_COMM_REQ_XFER_BEGIN at 1.0031 count:10000 ddt:MPI_INT
PERUSE_COMM_REQ_XFER_CONTINUE at 1.00313 count:10000 ddt:MPI_INT
   -- subsequent XFER_CONTINUEs deleted --
PERUSE_COMM_REQ_XFER_CONTINUE at 1.00322 count:10000 ddt:MPI_INT
PERUSE_COMM_REQ_XFER_END at 1.00327 count:10000 ddt:MPI_INT
PERUSE_COMM_REQ_COMPLETE at 1.00328 count:10000 ddt:MPI_INT
PERUSE_COMM_REQ_NOTIFY at 1.00328 count:10000 ddt:MPI_INT
```

4 Trace-File generation

To cope with the information provided by Peruse's functionality, one needs tools to visualize the output generated. We have ported the `mpitrace`-library of the Paraver-toolkit [1] to Open MPI. Paraver is a powerful performance analysis and visualization tool developed at CEPBA/BSC. Similar to Vampir, a trace is a time-dependant function of values for each process. Through filtering and

combination of several functions, meaningful investigations may be deduced even within large traces, e. g. searching and highlighting of parts of the trace with a GFlop-rate below a specified value.

Several points had to be addressed when porting `mpitrace` to Open MPI: removing assumptions on opaque MPI-objects (pointers to Open MPI internal structures) being integer values and separating helper functions into C- and Fortran-versions to avoid passing C-Datatypes to the Fortran PMPI-Interface. The port was tested on the Cacau-Cluster (having 64-bit pointers and 32-bit integers) with the `mpi_test_suite`, which employs combinations of simple functionality to stretch tests to the boundaries of the MPI-standard's definition.

For tracing, an application needs re-linking with the Peruse-enabled `mpitrace`-library. Peruse-events to be tested for are specified by the environment variable `MPITRACE_PERUSE_EVENTS`, separated by colons. Figure 4 shows the Paraver-window of an exemplary trace of ten sends, each of 10MB-size messages from rank zero to rank one with four Peruse-events attached[3]. Clearly, the initialization of the buffer on rank zero is visible as running time, while rank one awaits the message in the first `MPI_Recv`. Only with the Peruse-events (shown in gray), can the actual transfer be seen as the small green flags for each transmitted data fragment. By clicking into the trace-window, one may get further information on the Peruse-Events of the trace.

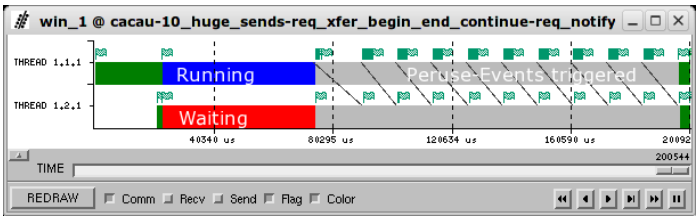

Fig. 4. Paraver visualization of 10 large msgs sent from rank zero to one (edited).

5 Application measurement

To demonstrate the suitability of Peruse-events tracing with the Paraver-toolkit, we show the tracing of the large molecular-dynamics package IMD with a benchmark test (`bench_cu3au_1048k.param`). The overall trace with 32 processes on cacau is shown in Fig. 5. One may note the long data distribution done using a linear send, followed by a collective routine during the initialization at the

[3] `MPITRACE_PERUSE_EVENTS=PERUSE_COMM_REQ_XFER_BEGIN:PERUSE_COMM_REQ_XFER_END:PERUSE_COMM_REQ_XFER_CONTINUE:PERUSE_COMM_REQ_NOTIFY`

beginning of the execution. The overall run shows ten iterations and a final collection phase. The right-hand window of Fig. 5 shows the achieved bandwidth, here ranging from 101 to 612 MB/s.

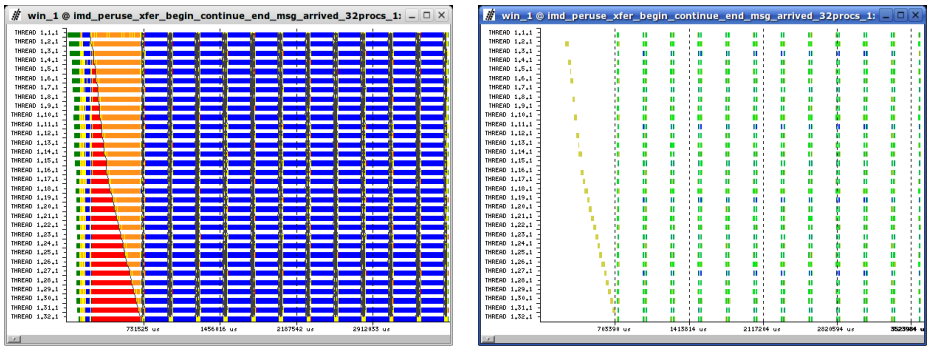

Fig. 5. Trace of IMD with 32 processes – overall run (left) and bandwidth (right).

Figure 6 zooms into one communication step of the run (left) with the corresponding bandwidth graph on the right hand side with up to 611 MB/s using on average 524 kB-sized messages. In order to appreciate the additional information Peruse-events give to the performance analyst, the actual time between message fragments arriving with the `PERUSE_COMM_REQ_XFER_CONTINUE` event are shown at the top-right of Fig. 6. Here, one may see how the in-flow rate of messages changes over time, ranging from $46\mu s$ to $224\mu s$ between fragments, corresponding to 4464 fragments/s up to 21739 fragments/s.

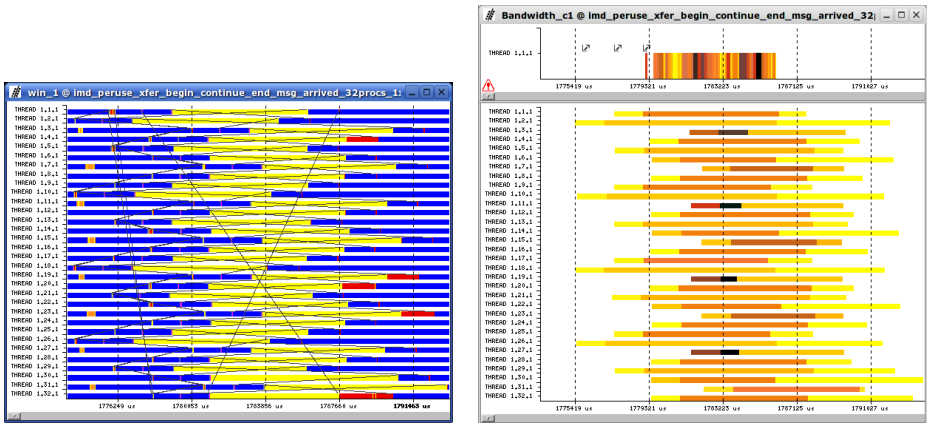

Fig. 6. Zoom into one communication step; bandwidth(right) and interval between fragments(right-top).

While with PMPI-based tracing it is possible to detect performance problems, such as "Late Sender", or "Late Receiver", the actual transferral of the message can not be seen. Particularly, for eager sends (small message) sends, the actual logical transferral of the message is far longer than the physical. This may be detected only with a corresponding `PERUSE_COMM_MSG_ARRIVED`-event on the receiver side.

Similarly, "Late Wait" situations of non-blocking communication cannot be detected through PMPI, as the communication will only be considered finished upon the corresponding `MPI_Wait`/`MPI_Wait`; here the `PERUSE_COMM_REQ_COMPLETE`-event notifies of the completion. Figure 7 shows a trace of such a situation. Process zero again sends a small message with eager protocol to process one, using non-blocking send and receive, respectively. The recv's `MPI_Wait` however is delayed by roughly 1.6ms. While the PMPI-based tracing considers the logical communication to finish within the `MPI_Wait` only, with Peruse one receives the early `PERUSE_COMM_REQ_COMPLETE`.

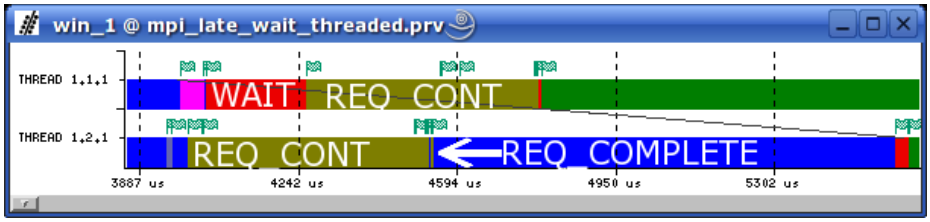

Fig. 7. Detection of late wait situation with Peruse on Open MPI (edited).

Furthermore, together with PMPI-wrapper, the tracing-library may additionally uncover book-keeping work commonly done by MPI-implementations before returning to the application, e. g. running event-handlers to progress other communication.

Additionally to message send/arrival times, Peruse allows information on the timing of internal traversal of message queues, which may be used to distinguish low network performance from slow queue management. Finally, with the introduction of the `PERUSE_COMM_REQ_XFER_CONTINUE`-event one may uncover fluctuations of the stream of fragments in case of network congestion.

6 Conclusion

In this paper we have described the implementation of the Peruse-interface into the Open MPI library. The integration into Open MPI was straightforward due to the modular design and the target platform. The authors are however aware, that for other implementations, the current design of the Peruse interface may

not be feasible due to MPI running in a different context, not allowing callbacks or due to the overhead introduced.

In the future, the authors would like to extend the Open MPI Peruse system with additional events yet to be defined in the current Peruse specification, e.g., collective routines and/or one-sided operations. Additionally the functionality for very low level events such as those defined within networking devices is also envisaged.

We would like to thank BSC for making `mpitrace` available. This work was made possible by funding of the EU-project HPC-Europa (Contract No. 506079), and also by the "Los Alamos Computer Science Institute (LACSI)", funded by Rice University Subcontract No. R7B127 under Regents of the University Subcontract No. 12783-001-05 49.

References

1. Paraver Homepage. WWW, May 2006. `http://www.cepba.upc.es/paraver`.
2. Peruse specification. WWW, May 2006. `http://www.mpi-peruse.org`.
3. Holger Brunst, Manuela Winkler, Wolfgang E. Nagel, and Hans-Christian Hoppe. Performance Optimization for Large Scale Computing: The scalable VAMPIR approach. In V.N. Alexandrov et al., editors, *Computational Science (ICCS'01)*, volume 2, pages 751–760. Springer, May 2001.
4. Edgar Gabriel, Graham E. Fagg, George Bosilca, Thara Angskun, Jack J. Dongarra, and Jeffrey M. Squyres et al. Open MPI: Goals, Concept, and Design of a Next Generation MPI Implementation. In D. Kranzlmüller, P. Kacsuk, and J.J. Dongarra, editors, *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, volume 3241, pages 97–104, Budapest, Hungary, September 2004. Springer.
5. Gabriele Jost, Haoquian Jin, Jesus Labarta, Judit Gimenez, and Jordi Caubet. Performance analysis of multilevel parallel applications on shared memory architectures. In *International Parallel and Distributed Processing Symposium (IPDPS 2003)*, volume 00, page 80b, April 2003. Nice, France.
6. Rainer Keller, Edgar Gabriel, Bettina Krammer, Matthias S. Müller, and Michael M. Resch. Towards efficient execution of MPI applications on the Grid: Porting and Optimization issues. *Journal of Grid Computing*, 1(2):133–149, 2003.
7. Message Passing Interface Forum. *MPI: A Message Passing Interface Standard*, June 1995. `http://www.mpi-forum.org`.
8. Message Passing Interface Forum. *MPI-2: Extensions to the Message-Passing Interface*, July 1997. `http://www.mpi-forum.org`.
9. Sameer Shende and Allen D. Malony. TAU: The TAU Parallel Performance System. 2005.
10. T.S. Woodall, R.L. Graham, R.H. Castain, D.J. Daniel, M.W. Sukalski, G.E. Fagg, E. Gabriel, G. Bosilca, T. Angskun, J.J. Dongarra, J.M. Squyres, V. Sahay, P. Kambadur, B. Barrett, and A. Lumsdaine. Open MPI's TEG Point-to-Point Communications Methodology: Comparison to Existing Implementations. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, volume 3241, pages 105–111, Budapest, Hungary, September 2004. Springer.