On the Performance of Transparent MPI Piggyback Messages

Martin Schulz, Greg Bronevetsky and Bronis R. de Supinski
{*schulzm,bronevetsky1,bronis*}*@llnl.gov*

Center for Applied Scientific Computing
Lawrence Livermore National Laboratory
PO Box 808, L-560, Livermore, CA 94551, USA

Abstract. Many tools, including performance analysis tools, tracing libraries and application level checkpointers, add piggyback data to messages. However, transparently implementing this functionality on top of MPI is not trivial and can severely reduce application performance. We study three transparent piggyback implementations on multiple production platforms and demonstrate that all are inefficient for some application scenarios. Overall, our results show that efficient piggyback support requires mechanisms within the MPI implementation and, thus, the interface should be extended to support them.

1 Motivation

Tools and support layers often must send additional information along with every message initiated by the main application. In most cases, tools must uniquely associate this additional information, often called piggyback data, with a specific message in order to capture the correct context and to avoid additional communication paths. A wide range of software systems piggyback data onto messages for diverse purposes; we detail a few here. Tracing libraries correlate send and receive events by sending vector clock information [4]. Performance analysis tools attach timing information to detect and analyze critical paths [2] or to compensate for instrumentation perturbation [7]. Application level checkpoint layers transmit epoch identifiers to synchronize global checkpoints [5].

Unfortunately, the MPI standard does not include a transparent piggyback mechanism. Instead, each system must provide its own, often ad-hoc, implementation. While a generic piggyback service could be added to an infrastructure like P^NMPI [6], the optimal solution depends on the specific usage scenario.

In this paper, we study the overhead and tradeoffs of three methods to support piggyback data:

– manual packing and unpacking the piggyback data and application payload into the same buffer;
– using datatypes with absolute addresses to attach piggyback data to the application payload; and

This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under contract DE-AC52-07NA27344 (LLNL-CONF-402937).

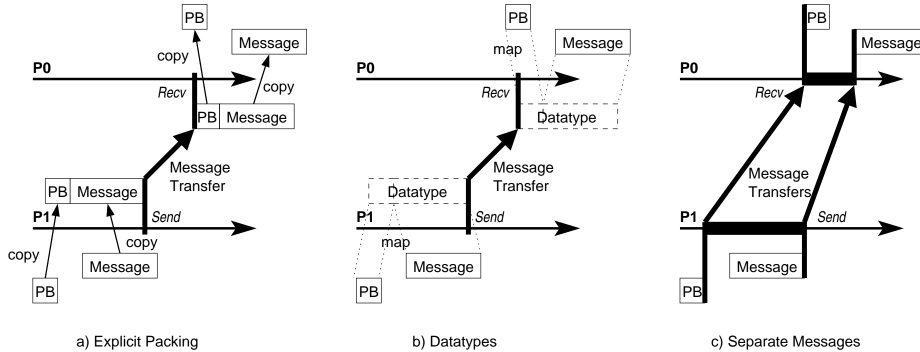

Fig. 1. Three different methods for transparent MPI piggyback messaging.

– using separate messages for the piggyback data and application payload.

All three methods are implemented as transparent PMPI modules.

Using these three mechanisms we discuss the impact on message latency and bandwidth, as well as application performance across three production platforms. Our results show that piggyback mechanisms implemented on top of MPI significantly reduce performance. In particular, communication intensive applications incur overhead that makes layered piggyback techniques inappropriate for performance critical scenarios, which significantly impedes the implementation of transparent tools. Thus, we need other mechanisms, potentially within the MPI implementation, that support this critical functionality.

2 MPI Piggyback Techniques

In the following we present three approaches to associate piggyback data with MPI messages transparently: explicitly packing piggyback data; using datatypes with absolute addresses; and sending separate piggyback messages. We focus on all combinations of MPI point-to-point messages including asynchronous and combined send/receive calls. We do not explore associating piggyback data with collective operations, which requires separate collectives tailored to the specific piggyback semantics (e.g., whether or not the data must be aggregated).

2.1 Explicit Pack Operations

Our first approach, shown in Figure 1a, uses *MPI_Pack* to pack the message payload and the piggyback data into a newly allocated buffer and transfers this buffer using the original send primitive. We receive the data into a local buffer and unpack the piggyback data and the message payload into their destinations. This approach is the easiest to implement and does not alter the original communication pattern. However, we must create additional buffers of varying sizes on both the send and the receiver side. Further, it adds a memory copy of the entire message payload on both sides and prevents the use of scatter/gather hardware available in many NICs.

2.2 Datatypes

Our second approach (Figure 1b) uses datatypes to combine the message contents with the piggyback data. During each send and receive operation we create a new datatype with *MPI_Type_struct* that combines a global pointer to the piggyback data with the original, possibly user-defined, datatype description of the message payload. We then communicate the payload that includes the piggyback data through the new datatype with the original send/receive primitives. This approach avoids additional explicit memory copies but must construct a special datatype for each communication operation. While it is possible to reduce the number of individual datatypes created, a new datatype must be defined for each message buffer location since it must use absolute addresses. Further, many MPI implementations do not provide efficient implementations of complex datatypes and may lose the memory copy savings.

2.3 Separate Messages

Our third approach does not change the actual message. Instead, it transmits the piggyback data in a separate message, as shown in Figure 1c. This approach duplicates all send and receive calls, transmitting the original, unchanged message in one message and the piggyback data in the other. "Wildcard" receives, however, require special treatment. If the original receive operation does not specify an explicit source node or tag but instead uses `MPI_ANY_SOURCE` and/or `MPI_ANY_TAG`, we must first complete the receive with the wildcards, determine all message parameters, including the sender ID and tag, and then post the second receive without wildcards. In the case of asynchronous receives, this means that we can not post the second receive until after the test or wait operation has completed for the first receive.

3 Experimental Setup

For the following experiments we implement our three piggyback methods as separate, application transparent modules, using the PMPI profiling layer and we explore both sending or packing piggyback data before or after the message payload. To reduce complexity, to enable code reuse, and to simplify experimentation we use $P^N MPI$ [6] for common services (e.g., request tracking) and to load the different modules dynamically. We use an additional driver module that emulates a tool or library using the piggyback functionality. This driver requests predefined piggyback sizes and produces and consumes the piggyback data.

We use three different machines for our experiments: *Thunder*, a 1024 node Linux cluster with 1.4 GHz 4-way Itianium-2 nodes and a Quadrics QSNetII (elan4) network running Quadrics MPI; *Atlas*, a 1152 node Linux cluster with 2.4 GHz 4-way dual-core Opteron nodes and an Infiniband network running MVAPICH; and *uP*, a 108 node AIX cluster with 1.9 GHz 8-way Power5 nodes and a Federation Switch network running IBM's MPI implementation.

4 Results

We first evaluate the impact of our piggyback methods on point-to-point bandwidth and latency using a simple ping-pong test between two tasks on different nodes sending arrays of integers. Figure 2 shows the results for varying message sizes and a constant piggyback size of one four byte integer. This represents a typical scenario since most tools use small piggyback messages.

The results show that explicit packing always leads to the worst performance while sending two separate messages has the best. In particular, using separate piggyback messages usually incurs only a negligible bandwidth reduction. The other methods, packing or using datatypes, incur a similar, larger bandwidth reduction on both commodity platforms (*Thunder* and *Atlas*), suggesting that their respective datatype implementations perform internal packing similar to our explicit packing approach. In contrast, it appears that IBM's MPI implementation on *uP* optimizes the transfer of custom datatypes, resulting in higher bandwidth compared to the packing approach. All approaches incur significant latency penalties of up to 200%. On *Thunder*, all approaches show similar latency overhead, while on *Atlas* and *uP* the datatype method has higher latency overhead compared to the other two approaches, which suggests a higher relative cost for the repeated creation and destruction of the custom datatypes. Sending or packing the piggyback data before or after the actual payload has little or no impact on raw latency and bandwidth in general.

The remainder of our analysis focuses on *Thunder* and *uP* since the *Atlas* and *Thunder* results are similar. Figure 3 presents latency (using 4 byte messages) and bandwidth (using 512 Kbyte messages) results with varying piggyback sizes. We observe an almost constant bandwidth reduction independent of the size of the piggyback data since performance is dominated by the significantly larger message payload. Latency, on the other hand, directly depends on the piggyback data size and is further influenced by changes in the underlying message protocol triggered by the additional payload (as indicated by the changing slopes).

However, impact on bandwidth and latency does not necessarily translate into application overhead. Thus, we study the performance of two scientific applications: Sweep3D, a computation-bound 3D neutron transport code from the ASCI Blue benchmark suite [1], and SMG2000, a communication-intensive semi-coarsening multigrid solver from the ASC Purple benchmark suite [3]. We execute both codes on 16 processors using a global working set size of 120x120x120 for Sweep3D and a local working set size of 70x70x70 for SMG2000. We run each configuration five times and report the lowest execution time. We report on two versions of each code: one that uses wildcard receives; and one that precisely specifies all receive parameters. These versions allow us to investigate the impact of the separate message method on wildcard receives, where the second receive must be postponed, as described in Section 2.3. This can have a notable effect on the performance of asynchronous wildcard communication.

The results, shown in Figure 4, reveal a negligible overhead for any piggyback implementation for the computation-bound Sweep3D. On *Thunder* the overhead is practically within the limits of measurement accuracy, while we see

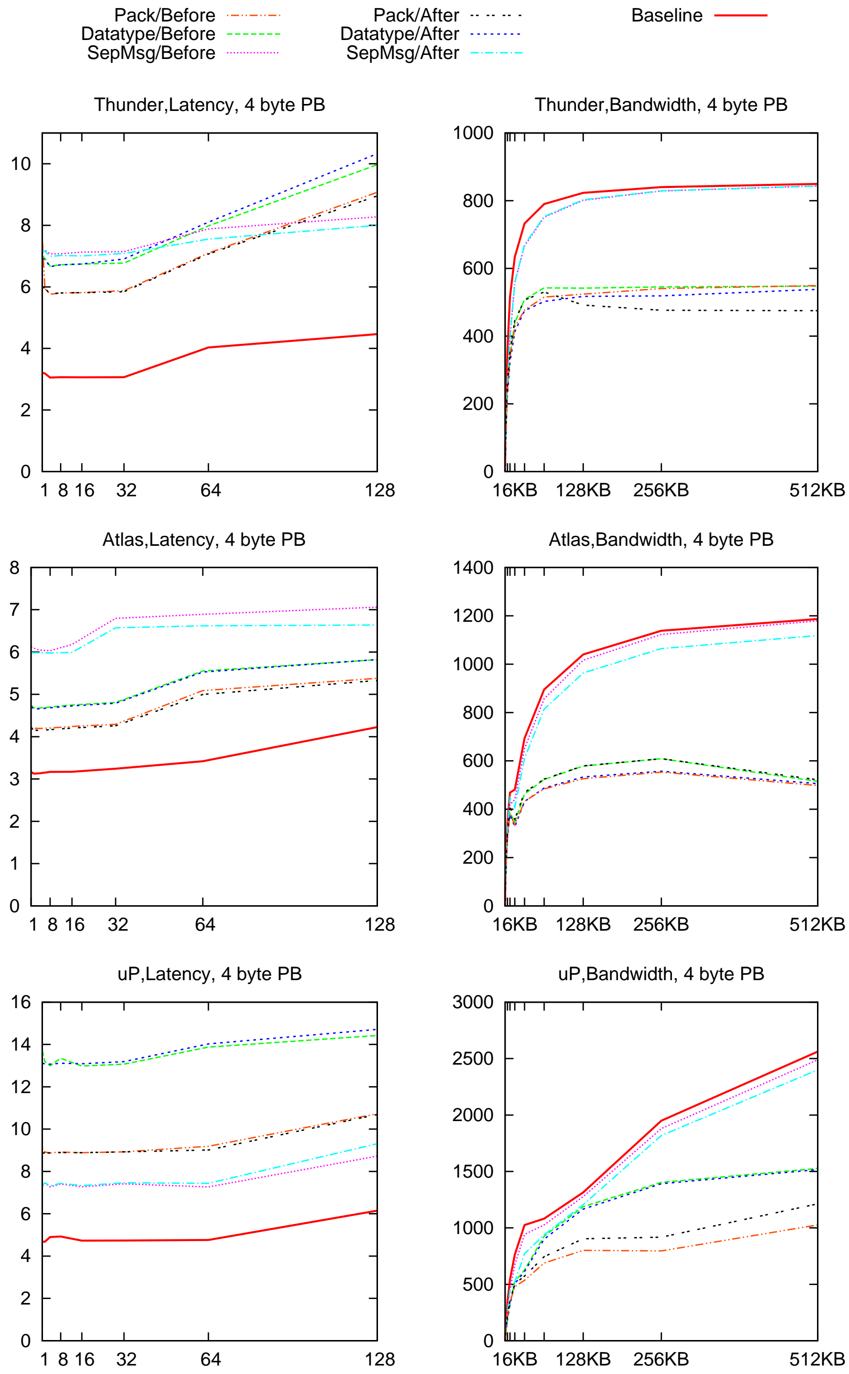

Fig. 2. Point to point transfer time in μs (left) and bandwidth in MB/s (right) with 4 byte piggyback data for varying message sizes shown on the x-axis.

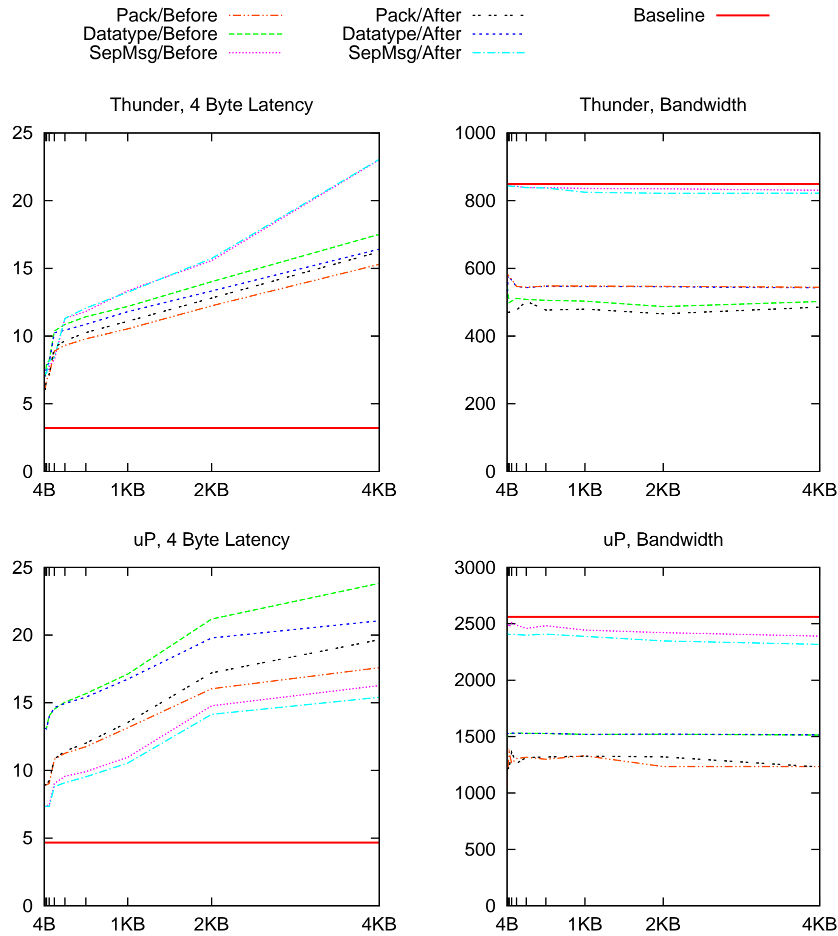

Fig. 3. Point to point latency in μs (left) and bandwidth in MB/s (right), varying piggyback size from 4 bytes to 4 Kbyte, as shown on the x-axis.

small overheads on *uP* (about 1% to 2%). The communication-bound SMG2000, on the other hand, presents a different picture: the average overhead on *Thunder* is around 10%, with the best performance being achieved by either packing the piggyback data after the message or using datatypes that prepend the piggyback data. Since packing the piggyback data first or using a datatype to place the piggyback after the message payload does not alter the overall message size or communication, the differences are probably due to cache effects based on the traversal direction of the MPI datatype and the corresponding data structures in the MPI implementation. Further, separate piggyback messages lead to the highest overhead for SMG2000, due to the added latency of the duplicated mes-

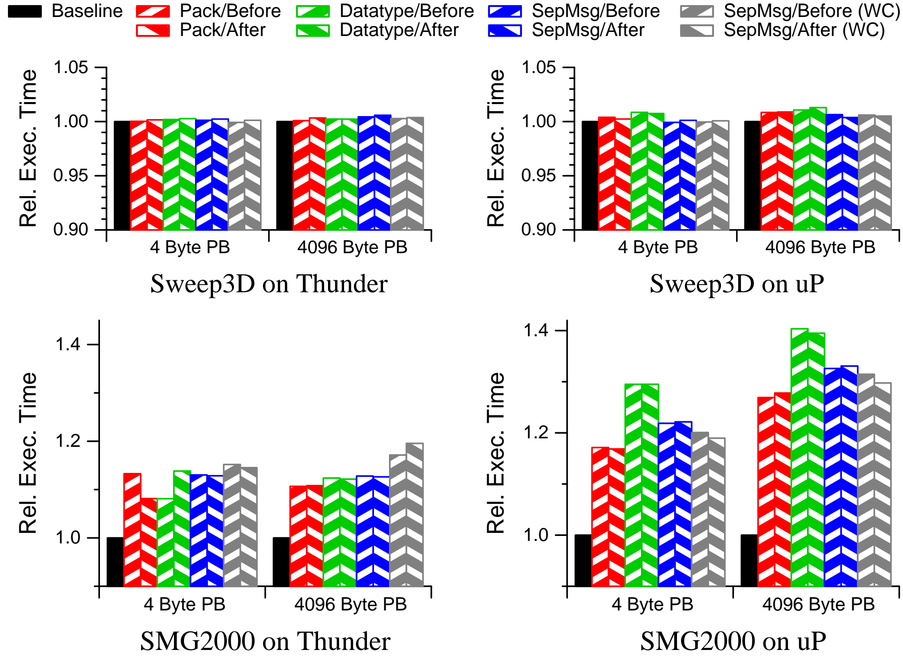

Fig. 4. Application overheads, i.e., execution time relative to sending no piggyback, for SMG2000 and Sweep3D on Thunder and uP using 4 and 4096 Bytes piggybacks.

sage traffic and the complex completion semantics during *MPI_Waitall*, which SMG2000 uses extensively. Using wildcard receives (marked as WC in the figure) further increases overhead to almost 20% for larger piggybacks.

We see even larger overheads on *uP* (20% for small piggybacks and 30% for large) with explicit packing achieving the best performance. In contrast to *Thunder*, using separate messages does not have the highest overhead compared to the other methods and using wildcard receives actually reduces overhead slightly. The latter most likely stems from the smaller number of outstanding requests the MPI implementation has to deal with since piggyback receives are postponed. Using datatypes exhibits the worst performance, most likely due to SMG2000's extensive use of user defined datatypes, as opposed to the simple integer arrays in the ping-pong test. Wrapping SMG200's already complex datatypes into the new piggyback datatypes prevents some of the previously observed optimizations for datatypes under IBM's MPI implementation.

5 Conclusions

Associating extra data with individual messages is an essential parallel tool technique, with uses in performance analysis, debugging and checkpointing. In this paper we have studied the performance of three different piggyback approaches — packing piggyback data and the original message into a unified buffer, using datatypes to transmit piggyback and message data in a single message, and using separate messages for piggyback and message data — and contrasted them to uninstrumented baselines.

We show that the choice of piggyback method strongly depends on the target application: simply examining the impact on bandwidth and latency can mispredict the real impact. In addition, performance depends on the MPI implementation and its optimization of advanced mechanisms such as custom datatypes or message coalescing. However, in communication intensive codes, like SMG2000, the use of piggyback data, independent of the implementation choice, leads to unacceptably high overhead for most tools.

In summary, our results show that we cannot layer a fully general piggyback solution on top of MPI without significantly harming performance for some application scenarios. However, efficient, low perturbation tools require exactly such an implementation. We therefore advocate extending the MPI standard to include a piggyback mechanism and are working with the MPI Forum towards this goal. Such extensions would allow the MPI implementation to optimize piggyback transfers, e.g., by including the data into a configurable header, and provide a truly portable and generally efficient piggyback mechanism.

References

1. Accelerated Strategic Computing Initiative. The ASCI sweep3d benchmark code. http://www.llnl.gov/asci_benchmarks/asci/limited/sweep3d/, Dec. 1995.
2. B. Barnes, B. Rountree, D. Lowenthal, J. Reeves, B. R. de Supinski, and M. Schulz. A Regression-Based Approach to Scalability Prediction. In *Proceedings of the International Conference on Supercomputing*, June 2008.
3. R. Falgout and U. Yang. hypre: a Library of High Performance Preconditioners. In *Proceedings of the International Conference on Computational Science (ICCS), Part III, LNCS vol. 2331*, pages 632–641, Apr. 2002.
4. D. Kranzlmüller and J. Volkert. NOPE: A Nondeterministic Program Evaluator. In *Proceedings of the 4th International ACPC Conference Including Special Tracks on Parallel Numerics and Parallel Computing in Image Processing, Video Processing, and Multimedia. Lecture Notes In Computer Science; Vol. 1557.*, pages 490–499, 1999.
5. M. Schulz, G. Bronevetsky, R. Fernandes, D. Marques, K. Pingali, and P. Stodghill. Implementation and Evaluation of a Scalable Application-level Checkpoint-Recovery Scheme for MPI Programs. In *Supercomputing 2004 (SC'04)*, Nov. 2004.
6. M. Schulz and B. R. de Supinski. P^NMPI tools a whole lot greater than the sum of their parts. In *Supercomputing 2007 (SC'07)*, 2007.
7. S. Shende, A. D. Malony, A. Morris, and F. Wolf. Performance Profiling Overhead Compensation for MPI Programs. In *Proceedings of the 12th European PVM/MPI Users' Group Meeting*, pages 359–367, Sept. 2005.