Performance of MPI sends of non-contiguous data

Victor Eijkhout

2018

Abstract

We present an experimental investigation of the performance of MPI derived datatypes. For messages up to the megabyte range most schemes perform comparably to each other and to manual copying into a regular send buffer. However, for large messages the internal buffering of MPI causes differences in efficiency. The optimal scheme is a combination of packing and derived types.

1 Introduction

Most Message Passing Interface (MPI) benchmarks use the send of a contiguous buffer to measure how close the MPI software gets to the theoretical speed of the hardware. However, in practice we often need to send non-contiguous data, such as the real parts of a complex array, every other element of a grid during multigrid coarsening, or irregularly spaced elements in a Finite Element Method (FEM) boundary transfer.

To support this, MPI has so-called 'derived datatypes' routines, that describe non-contiguous data. While they offer an elegant interface, programmers always worry about any performance implications of using these derived types. In this report we perform an experimental investigation of various schemes for sending non-contiguous data, comparing their performances both to each other, as well as to the performance of a contiguous send, which we will consider as the optimal, reference, rate.

Specifically, we investigate the following non-orthogonal issues:

1. Manually copying irregular data to a contiguous buffer and sending the latter, versus using MPI derived datatypes to send irregular data;
2. using both two-sided and one-sided point-to-point; and
3. using buffered and packed sends to counteract adverse effects of MPI's internal buffering.

We discuss our schemes in section 2 and our experimental setup in section 3. Results are graphed and discussed in section 4. We give our overall conclusion in section 5.

2 Send schemes

We describe the various send schemes that we tested. Any performance discussion will be postponed to section 4.

2.1 Contiguous send

To establish a baseline we send a contiguous buffer, and accept the result as the attainable performance of the hardware/software combination.

The cost of this scheme is

1. loading N elements from memory to the processor, and
2. sending N elements through the network.

The two are probably overlapped, and to first order of approximation we equate memory bandwidth and network transmission speed, so we assign a proportionality constant of 1 to this scheme.

2.2 Manual copying

Since we are sending non-contiguous data, the minimal solution is to copy the data between a user array and the send buffer. We allocate the send buffer outside the timing loop, and reuse it.

The cost for this is

1. the cost of the copying loop, and
2. the cost of the contiguous send.

The copying loop loads $2N$ elements from memory and writes N. The latter writes can be interleaved with the loads, so most likely only the loads contribute to the measured time. Next, the building of the send buffer has to be fully finished before the `MPI_Send` can be issued, so with the assumptions made before, we expect a slowdown of a factor of 3 over the reference speed.

A further point to note is that the copying loop is of necessity in user space, so any offload of sends to the Network Interface Card (NIC) is of limited value, reducing the proportionality constant to 2.

2.3 Derived datatypes

MPI has routines such as `MPI_Type_create_vector` and `MPI_Type_create_subarray` to offer a convenient interface to non-contiguous data. In a naive implementation, these routines copy data to an internal MPI buffer, which is subsequently send. In this case we expect a performance similar to the manual copying scheme; any degradation indicates a true performance penalty.

However, with enough support of the NIC and its firmware, it would be possible for this scheme to pipeline the reads and sends similarly to the reference case. That this is possible in principle was shown in [2]. In practice we don't see this performance, and our observed performance numbers are completely in accordance with an assumption of non-pipelining.

2.4 Buffered sends

At larger message sizes, it is reasonable to assume that MPI can run out of internal buffer space, and send the message in packets, giving a performance degradation. We have investigated whether any benefits are accrued from using buffered sends. Here a user buffer is attached with `MPI_Buffer_attach` and the send is replaced by `MPI_Bsend`.

2.5 One-sided transfer

We explored using one-sided point-to-point calls. There are two options:

1. Transfering the data elements in a loop. However, this requires a function call per elements so we did not pursue this.
2. Transfering a single derived type.

While one-sided messages have the potential to be more efficient, since they dispense with a rendez-vous protocol, they still require some sort of synchronization. We use `MPI_Win_fence`.

2.6 Packing

Finally, we considered the use of the `MPI_PACKED` data type, used two different ways.

- We use a separate `MPI_Pack` call for each element. Since this uses a function call for each element sent, we expect a low performance.
- We use a single `MPI_Pack` call on a derived (vector) datatype.

The `MPI_Pack` command packs data in an explicitly allocated send buffer in userspace. This means that we no longer rely on MPI for buffer management. If the implementation of the pack command is sufficiently efficient, we expect this scheme to track the manual copying scheme.

3 Experimental setup

3.1 Hardware

We use the following clusters at TACC:

- Lonestar5: a Cray XC40 with Intel compilers and Cray mpich7.3.
- Stampede2-knl: a Dell cluster with Knightslanding nodes connected with Omnipath, using Intel compilers and Intel MPI.
- Stampede2-skx: a Dell cluster with dual Skylake nodes connected with Omnipath, using Intel compilers and both Intel MPI and mvapich.

3.2 Performance measurement

We measured the time for 20 ping-pongs, where:

- The 'ping' was the non-contiguous send. This used an ordinary `MPI_Send` for all two-sided versions except `MPI_Bsend` for the buffered protocol, and `MPI_Put` for the one-sided version.
- In all two-sided schemes the target process executed an `MPI_Recv` on a contiguous buffer.
- In the two-sided schemes, there was a zero byte 'pong' return message. In the one-sided scheme we surrounded the transfer with active target synchronization fences; the timers surrounded these fences.

Time reported in the figures below is the total time divided by the number of ping-pongs. Every ping-pong was timed individually. We used `MPI_Wtime`, which on our platforms had a resolution of $1 \cdot 10^{-6}$ seconds, and the minimum measurement ever (for the very smallest message) was around $6 \cdot 10^{-6}$ seconds. Our

code is set up to dismiss measurements that are more than one standard deviation from the average, but in practice this test is never needed.

All buffers were allocated outside the timing loop, using 64 byte alignment. Page instantiation was kept outside the timing loop by setting all arrays explicitly to zero.

In between every two ping-pongs an array of size 50M is rewritten. This is enough to flush the caches on our systems.
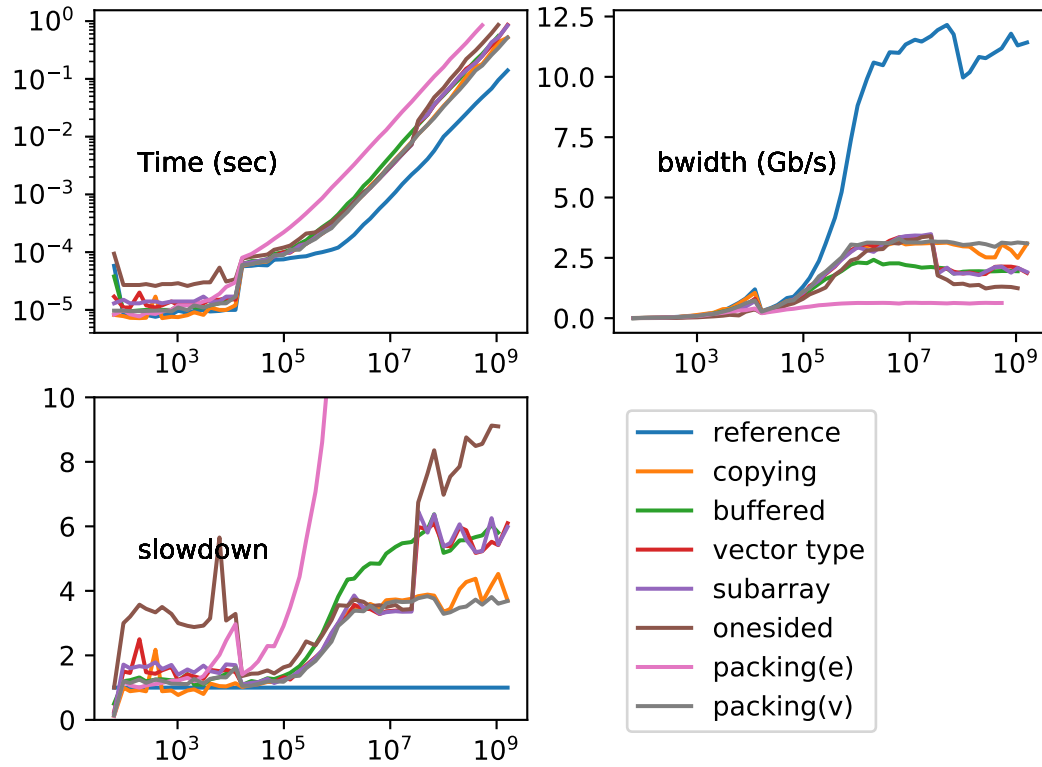
4   Results and discussion

Figure 1: Time and bandwdith on Stampede2-skx using Intel MPI

We report on the performance with Skylake processors with Intel MPI (figure 1), Skylake with MVAPICH2 (figure 2), Cray with Cray MPICH (figure 3), and Knights Landing processors with Intel MPI (figure 4). The legend on the figures is self-explanatory, except that `packing(e)` stands for 'packing by element' and `packing(v)` for 'packing a vector data type'.
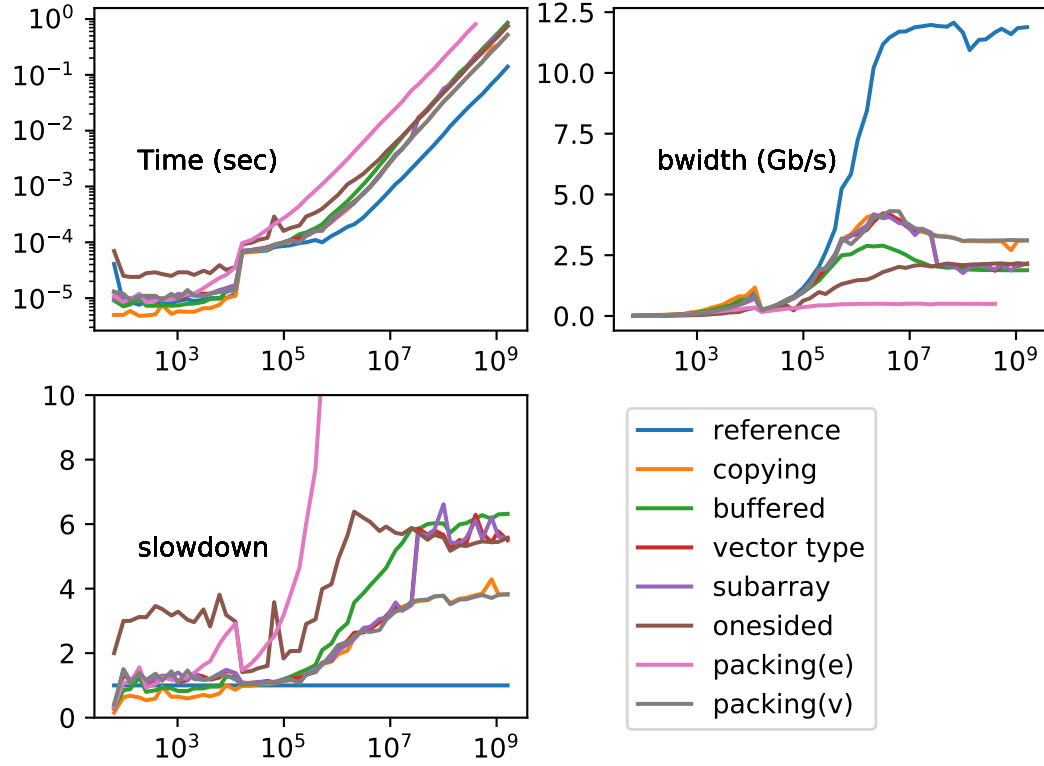
Figure 2: Time and bandwdith on Stampede2-skx nodes using mpivach2

The graphs show for each installation:

- The measured time of one ping-pong, as function of message size in bytes;
- the effective bandwidth; and
- the slowdown with respect to the contiguous send. The value of this plot is mostly for smaller messages, where differences between schemes are not apparent in the first two plots. Note that occasional blips in the reference curve are reflected in the slowdown numbers.

We start by discussing general tendencies, concluding with the relatively minor differences between the various installations.

4.1 Derived datatypes

We see that in most cases the time for sending a derived datatype tracks the time for sending a manually copied buffer very well. This can best be explained by assuming that sending a derived datatype is done by copying the data internally to contiguous storage, which is then sent.
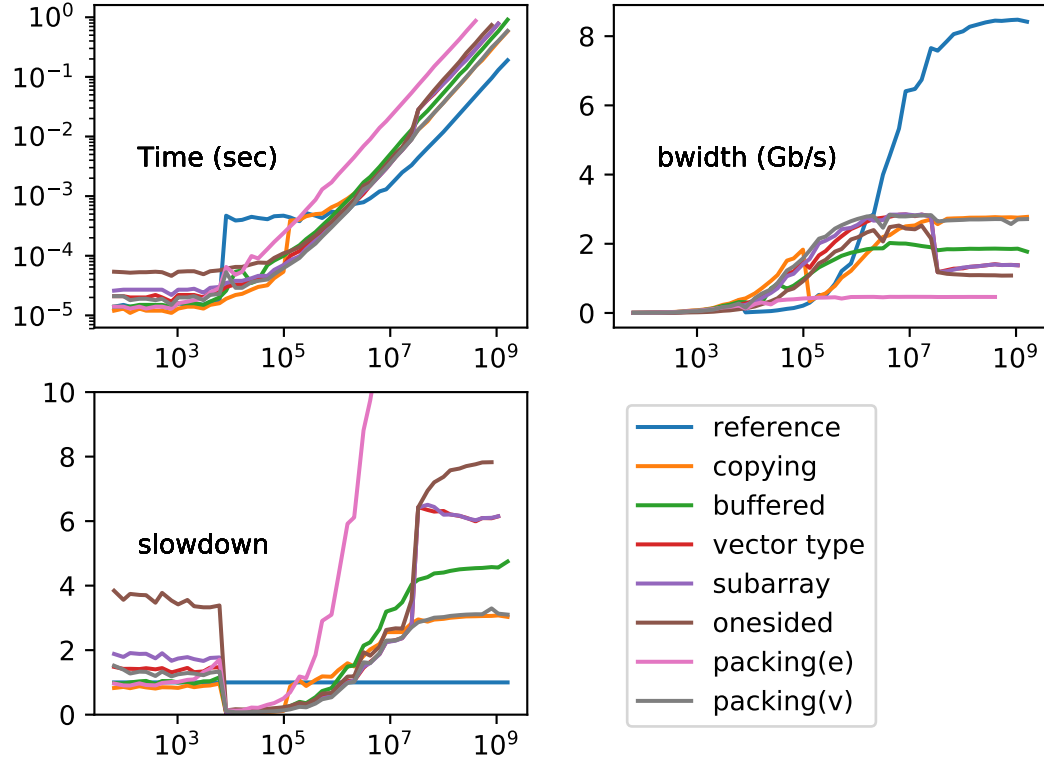
Figure 3: Time and bandwdith on a Cray XC40 using the native MPI

However, we see a drop in performance for messages beyond a few tens of megabytes. We assume that for such relatively large messages the internal buffer bookkeeping of MPI becomes complicated, and incurs extra overhead.

Since many MPI implementations are closed source, and the behaviour depends on the precise scheme, we can not further analyze this. (In fact, even for the open MPICH implementation we found the buffer management code too inscrutable for analysis.)

4.2 Buffered sends

MPI buffered sends (that is, passing a userspace buffer with `MPI_Buffer_attach` for MPI to use internally, and calling `MPI_Bsend`) are supposed to alleviate limitations of MPI's internal buffering. Strangely, having a fully allocated userspace buffer does not help the slowdown for large messages. In fact, in most MPI implementations it performs worse, even for intermediate message sizes.
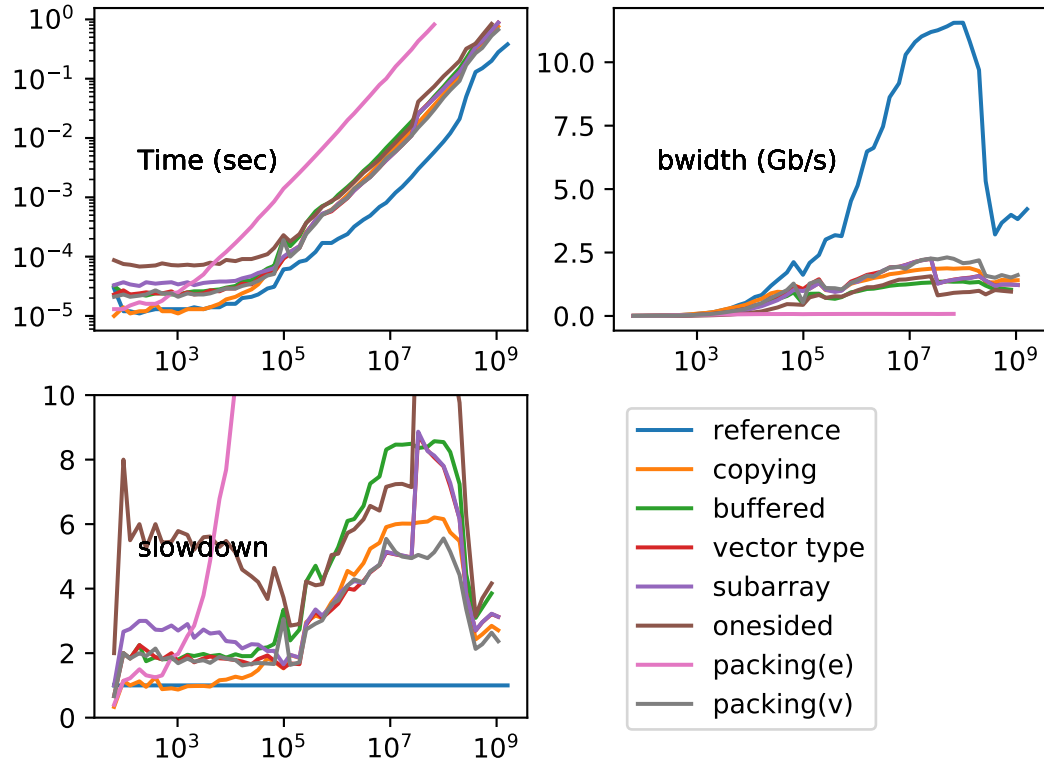
Figure 4: Time and bandwdith on Stampede2-knl using Intel MPI

4.3 Packed buffers

The elementwise packing scheme performs predictably very badly. However, packing a derived type gives essentially the same performance as manual copying. This proves two points:

- The `MPI_Pack` command is as efficient as a user-coded copying loop.
- There are no intrinsic inefficiencies in the derived datatype code.

The fact that this scheme performs better than using a derived type directly also indicates that there are inefficiencies in the internal buffer management of MPI that are obviated by having the buffer totally in user space.

4.4 One-sided communication

The performance of one-sided communication does not track the previous schemes. Its behaviour seems dependent on the message size.

1. For small messages, one-sided transfer is slow, probably because of the more complicated synchronization mechanism of `MPI_Win_fence`, which imposes a large overhead.
2. For intermediate size messages, one-sided transfer is competitive with other schemes, except for the mvapich implementation where it is several factors slower.
3. For large messages the range of behaviours is even larger. However, there are few cases where one-sided communication is truly competitive.

4.5   Eager limit

Most MPI implementations have a switch-over between the 'eager protocol', where messages are send without handshake, and a 'rendezvous' protocol that does involve acknowledgement. From its nature, the eager protocol relies on sufficient buffer space, so there is an 'eager limit'. One typically observes that messages just over the eager limit can perform worse (at least per byte) than once just under.

The effects of the eager limit are visible in most plots, with the following remarks:

- As is to be expected, we mostly see a performance drop at the eager limit for all send schemes. For one-sided puts it is less pronounced, and for the packing scheme it largely drowns in the overhead.
- On Cray-mpich we see the performance drop for the reference speed, and at double the data sizes for the packing scheme. For the other schemes not much of a drop is visible. The reason for this is unclear.

We have tested setting the eager limit over the maximum message size, but this did not appreciably change the results for large messages.

4.6   Cache flushing

In tests not reported here we dispensed with flushing the cache in between sends. This had a clear positive effect on intermediate size messages.

4.7   Further tests

This paper is of course not a full exploration of all possible scenarios in which derived datatypes can be used. In fact, it only examines the very simplest case of a derived type.

1. Types with less regular spacing may give worse performance due to decreased use of prefetch streams in reading data.
2. Types with larger block sizes may perform better due to higher cache line utilization in the read.

A more interesting objection to these tests is that we used exactly one communicating process per node. This is a realistic model for certain hybrid cases, but not for 'pure MPI' codes. However, a limited test, not reported in graph form here, shows that no performance degradation results from having all processes on a node communicate. (In fact, sometimes a slightly higher bandwidth results.)

The interested reader can find the code in an open repository and adapt it to their own purposes [1].

4.8 Differences between installations

We give the results on Skylake nodes with OmniPath fabric and Intel MPI (henceforth: `impi`) as representative in figure 1. Switching to `mvapich2` gives largely the same results; figure 2.

The native Cray MPI also has similar performance, with the exception that one-sided performance for large sizes is on par with the derived types, unlike on Stampede2 where for all sizes it shows a relative degradataion; figure 3.

Finally, Stampede2-knl shows the same peak network performance, but the performance of our non-contiguous tests is hampered by the core performance in constructing the send buffer; figure 4.

5 Conclusion

Schemes for sending non-contiguous data between two processes are considerably slower than sending contiguous data. The slowdown of at least a factor of three can mostly be explained by multiple memory reads and a lack of overlap of memory and network traffic. Attaining such overlap for non-contiguous data depends on advanced functionality of the network interface [2].

For any but large (over 10^8 bytes) messages the various schemes perform fairly similarly, so there should be no reason not to use derived datatypes, these being the most user-friendly. Depending on the architecture, one-sided transfers may behave worse, and buffered sends are at a disadvantage, both even at intermediate sizes (10^4–10^8 bytes).

The scheme that consistently performs best applies `MPI_Pack` to a derived datatype. This scheme in all cases gives the same performance as a manual gather copy of the data.

References

[1] Victor Eijkhout. Parallel performance studies repository. https://bitbucket.org/VictorEijkhout/parallel-performance-studies/src.

[2] M. Li, H. Subramoni, K. Hamidouche, X. Lu, and D. K. Panda. High performance mpi datatype support with user-mode memory registration: Challenges, designs, and benefits. In *2015 IEEE International Conference on Cluster Computing*, pages 226–235, Sept 2015.