The Parallel Universe

intel Software

The Message Passage Interface:

A Standard for Distributed Memory Systems

MPI-3 Shared Memory Programming

Estimating Memory Consumption

Intel® MPI Library Conditional Reproducibility

Issue
21
2015

CONTENTS

The Parallel Universe is a free quarterly magazine. Click here to sign up for future issue alerts and to share the magazine with friends.

LETTER FROM THE EDITOR

James Reinders, Director of Parallel Programming Evangelism at Intel Corporation, coeditor of an exciting new book *High Performance Programming Pearls*. His other book credits include *Multithreading for Visual Effects* (2014), *Intel® Xeon Phi™ Coprocessor High Performance Programming* (2013), *Structured Parallel Programming* (2012), *Intel® Threading Building Blocks: Outfitting C++ for Multicore Processor Parallelism* (2007), and *VTune™ Performance Analyzer Essentials* (2005).

Happy Birthday, MPI

We recently passed the birthday of the Message Passage Interface (MPI) standard, which like many achievements in the world of computing, was borne out of the gathering of great minds. On April 29, 1992, dozens of programmers and computer industry specialists from 40 organizations in government, academia, and the private sector gathered in Williamsburg, Virginia, at the Workshop on Standards for Message Passing in a Distributed Memory Environment. There they discussed and developed the genesis of what is today the widely used MPI standard for writing message-passing programs.

MPI offers parallel programmers many advantages, including practicality, portability, efficiency, and ease of use. The latest MPI standard, MPI-3, increases flexibility as discussed in this issue's feature article, "An Introduction to MPI-3 Shared Memory Programming." The authors share how to transform common MPI send/receive patterns using Shared Memory Programming, with resulting performance advantages.

But improvements often come with trade-offs. The other articles in this issue offer ways to keep the MPI standard working in tandem with all other aspects of **parallel programming**. For example, floating-point operations in numerical codes may introduce differences that can increase with each iteration. "Intel® **MPI Library** Conditional Reproducibility" uses a simple example to demonstrate Intel MPI Library's collective operations that can be used for reproducible results when certain required and reasonable conditions are met.

High performance computing applications tend to use most of the available memory on a node, and estimating the memory consumption of MPI libraries can be difficult. "Intel MPI Memory Consumption" takes a closer look at estimating the memory consumption of the Intel MPI Library and how users can fine-tune their settings to reduce their memory footprint.

The MPI standard has long offered a great foundation to build upon. In addition to the high value of coding to a standard you can depend on for your application, a benefit of having a widely accepted standard is the innovation that can occur through additions to the standard and tools support. In this edition of Parallel Universe, we are happy to share three examples of this value that the longstanding MPI standard has while continuing to evolve.

James Reinders
May 2015

An Introduction to MPI-3 Shared Memory Programming

An All-MPI Alternative to MPI/OpenMP* Programming Worth Considering

By **Mikhail Brinskiy**, *Software Development Engineer, and* **Mark Lubin**, *Technical Consulting Engineer, Intel Corporation*

Abstract

The Message Passing Interface (MPI) standard is a widely used programming interface for distributed memory systems. Hybrid parallel programming on many-core systems most often combines MPI with OpenMP*. This MPI/OpenMP approach uses an MPI model for communicating between nodes while utilizing groups of threads running on each computing node in order to take advantage of multicore/many-core architectures such as Intel® Xeon® processors and Intel® Xeon Phi™ coprocessors.

The MPI-3 standard introduces another approach to hybrid programming that uses the new MPI Shared Memory (SHM) model.[1] The MPI SHM model, supported by Intel® MPI Library Version 5.0.2[2] enables changes to existing MPI codes incrementally in order to accelerate communication between processes on the shared-memory nodes.[3]

In this article, we present a tutorial on how to start using MPI SHM on multinode systems using Intel Xeon with Intel Xeon Phi. The article uses a 1-D ring application as an example and includes code snippets to describe how to transform common MPI send/receive patterns to utilize the MPI SHM interface. The MPI functions that are necessary for internode and intranode communications will be described. A modified MPPTEST benchmark has been used to illustrate performance of the MPI SHM model with different synchronization mechanisms on Intel Xeon and Intel Xeon Phi based clusters. With the help of Intel MPI Library Version 5.0.2, which implements the MPI-3 standard, we show that the shared memory approach produces significant performance advantages compared to the MPI send/receive model.

1-D Ring: From Standard MPI Point-to-Point to MPI SHM

We approach the semantics of the MPI SHM API by modifying a well-known 1-D ring example, where each MPI rank can exchange MPI-1 nonblocking messages with its left and right neighbors.[4]


```
MPI_Irecv (&buf[0],…, prev,…, MPI_COMM_WORLD, &reqs[0]);
MPI_Irecv (&buf[1],…, next,…, MPI_COMM_WORLD, &reqs[1]);
MPI_Isend (&rank,…, prev,…, MPI_COMM_WORLD, &reqs[2]);
MPI_Isend (&rank,…, next,…, MPI_COMM_WORLD, &reqs[3]);
      {do some work}
MPI_Waitall (4, reqs, stats);
```

1 Figure 1. Nearest neighbor exchange in a 1-D ring topology and corresponding MPI-1 code

We intend to run our code on multiple multicore nodes with all MPI ranks sharing memory on each node. The function `MPI_Comm_split_type` enables programmers to determine the maximum groups of MPI ranks that allow such memory sharing. This function has a powerful capability to create "islands" of processes on each node that belong to the output communicator `shmcomm`:

```
MPI_Comm shmcomm;
MPI_Comm_split_type (MPI_COMM_WORLD, MPI_COMM_TYPE_SHARED,0, MPI_INFO_NULL,
&shmcomm);
```

The companion collective function then allocates MPI-3 remote memory access (RMA) type memory windows on each node. They are called windows because MPI restricts what part of a process's memory will be made available to other processes:

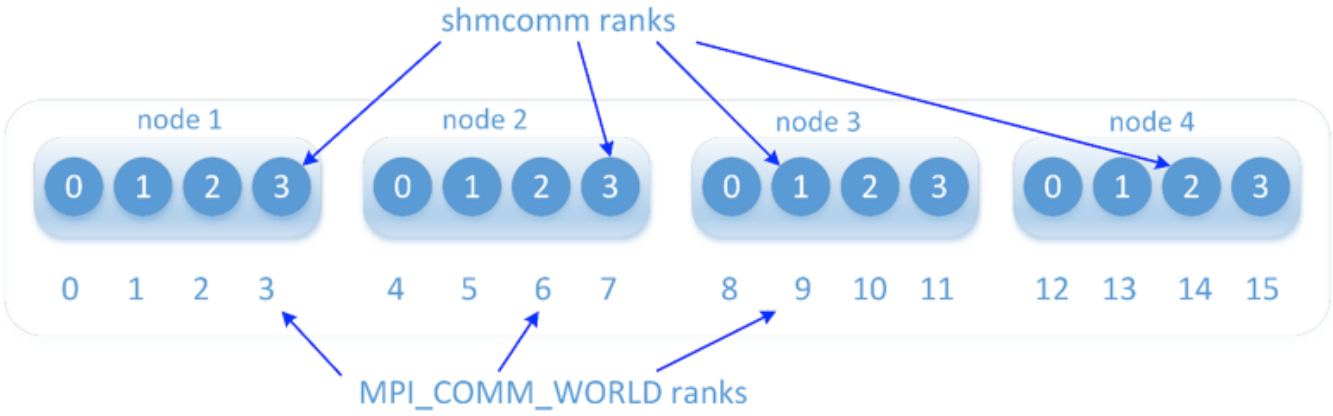
```
MPI_Win_allocate_shared (alloc_length, 1,info, shmcomm, &mem, &win);
```

To execute MPI send/receive point-to-point operations between the nodes (as in the original example) and execute MPI SHM functions within each node, we need a mechanism to distinguish between ranks that fit into the same node versus ranks belonging to different nodes. To accomplish this, we separate MPI groups from the global communicator and shared memory communicator `shmcomm`:

```
MPI_Comm_group (MPI_COMM_WORLD, &world_group);
MPI_Comm_group (shmcomm, &shared_group);
```

Then we can map global rank numbers onto the `shmcomm` ranks numbers and store this mapping into the `partners_map` array (**Figure 2**).

```
MPI_Group_translate_ranks (world_group, n_partners, partners, shared_group,
partners_map);
```



2 Mapping of global ranks to shmcomm ranks. If some of the neighboring ranks are residing on a different node, their mapping in the resulting array partners_map will be a predefined constant, MPI_UNDEFINED.

The `MPI_Win_shared_query` API can be used to find out the process-local addresses for shared memory segments using a conditional test, `partners_map[j]!= MPI_UNDEFINED`, which is true when the current rank and its communication partners reside on the same node and therefore share common memory. The returned memory pointers array, `partners_ptrs`, can be used for simple loads and stores, replacing costly MPI send/receive functions within the shared memory domain (**Figure 3**).

```
for (j=0; j<n_partners; j++)
{
  if (partners_map[j] != MPI_UNDEFINED)
      MPI_Win_shared_query (win, partners_map[j],…, &partners_ptrs[j]);
}
```

3 MPI_Win_shared_query can return different process-local addresses for the same physical memory on different processes

Unlike the point-to-point message-passing model, the MPI SHM interface assumes explicit use of synchronizations to ensure memory consistency and assumes that the changes in memory are visible to the other processes. In some cases, it enables higher performance at the cost of more complex code that each developer needs to understand and maintain. Therefore, in this article, we focus on the semantics of these new synchronizations and their effect on performance.

The MPI SHM model, supported by Intel® MPI Library Version 5.0.2, enables changes to existing MPI codes incrementally in order to accelerate communication between processes on the shared-memory nodes.

The so-called passive target MPI RMA synchronization, defined by the pair of `MPI_Win_lock_all` and `MPI_Win_unlock_all` functions for all processes sharing an RMA window, was chosen as one of the most performance-efficient.[5] The term “lock” here does not have the same connotation familiar to shared memory programmers such as with mutexes. The pair of `MPI_Win_lock_all` and `MPI_Win_unlock_all` simply denotes the time interval, called an RMA access epoch, when remote memory operations are allowed to occur. In this case, the `MPI_Win_sync` function has to be used to ensure completion of memory updates and `MPI_Barrier` to synchronize all processes on the node in time (**Figure 4**).

```
//Start passive RMA epoch
MPI_Win_lock_all (MPI_MODE_NOCHECK, win);

// write into mem array hello_world info
mem[0] = rank;
mem[1] = numtasks;
memcpy(mem+2, name, namelen);

MPI_Win_sync (win);     // memory fence - sync node exchanges
MPI_Barrier (shmcomm); //time barrier
```

4 Passive RMA synchronizations are needed for MPI SHM updates. The performance assertion MPI_MODE_NOCHECK hints that the epoch can begin immediately at the target. Note that on some platforms one more MPI_Win_sync would be needed after the MPI_Barrier to ensure memory consistency at the reader side.

Calling `MPI_Win_lock` for each particular neighbor is a valid approach as well, and sometimes it can provide performance advantages, but it requires more lines of code. Alternatively, one could employ the active target MPI RMA communication mode that relies on a pair of `MPI_Win_fence` operations surrounding memory updates. The `MPI_Win_fence` method is less verbose compared to lock/unlock epochs since it already includes barrier synchronizations, but it produced slower results in our experiments.

With correct synchronizations in place, all processes can retrieve their neighbors' information either via shared memory or using standard point-to-point communications if neighbors are on the different nodes (**Figure 5**).

```
for (j=0; j<n_partners; j++){
  if (partners_map[j] != MPI_UNDEFINED)
  {
    i0 = partners_ptrs[j][0]; //load ops from MPI SHM!
    i1 = partners_ptrs[j][1];
    i2 = partners_ptrs[j]+2;

  } else { // inter-node non-blocking MPI
    MPI_Irecv (&rbuf[j],…, partners[j], 1 , MPI_COMM_WORLD, rq++);
    MPI_Isend (&rank,…, partners[j], 1 , MPI_COMM_WORLD, rq++);

  }
}
```

5 Halo exchanges using MPI SHM on the node and standard nonblocking MPI send/receive for internode communications

After completion of MPI SHM communications, we can close the access epoch using `MPI_Win_unlock_all`. The internode communications are synced with `MPI_Waitall` as usual.

The resulting code is available for **download**.

Modifying MPPTEST Halo Exchange to Include MPI SHM

To evaluate the performance of the MPI SHM available in Intel MPI Library Version 5.0.2 on clusters based on Intel Xeon processors and Intel Xeon Phi coprocessors, we modified the halo exchange algorithm from the MPPTEST benchmark[6] using as a prototype the 1-D ring example. Although the MPPTEST halo test does not have the computational kernels present in many real applications, it provides an unhindered view of how different-order halo exchanges, message sizes, and MPI synchronizations may affect performance.

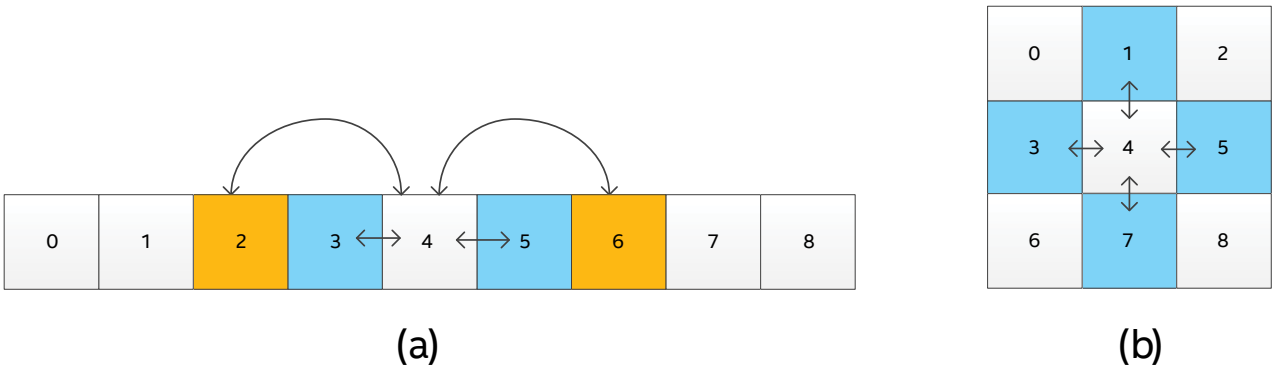
It is known that the MPI SHM model provides performance benefits by avoiding regular send/receive memory copy operations, MPI stack latencies, and tag matching.[7] The replacement of these traditional MPI mechanisms with fast intranode communications, such as memory copy operations, exposes in turn the effect of the remaining major contribution to overall intranode performance, the different available MPI SHM synchronizations briefly described in the last section.

We implemented three new halo patterns for the MPPTEST suite—`mpi3shm_lockall`, `mpi3shm_lock`, and `mpi3shm_fence`—that can be used as new MPPTEST configuration parameters. All of them use the same MPI SHM communication scheme, but they employ different shared memory synchronization primitives:

- **mpi3shm_lockall.** This relies on `MPI_Win_lock_all` and `MPI_Win_unlock_all` to open and close an access epoch and relies on `MPI_Barrier` and `MPI_Win_sync` for process synchronization (memory and time).
- **mpi3shm_lock.** This is the same as `mp3shm_lockall` but uses separate `MPI_Win_lock` and `MPI_Win_unlock` calls for each neighbor in the halo exchange.
- **mpi3shm_fence.** A pair of successive `MPI_Win_Fence` calls ensures that any local stores to the shared memory executed between them are consistent, and thus there is no need for any other synchronization primitives.

To investigate different processes topologies in halo exchanges, we introduced a new configuration parameter into the MPPTEST halo benchmarks: `-dimension`. This parameter instructs MPPTEST to use one of two available process decompositions, 1-D or 2-D, with the latter used by default. If the specified number of partners is more than enough for nearest

neighbors' exchanges, the decomposition with deeper density is used. An example based on nine processes and four partners is shown in Figure 6. In the case of 1-D decomposition, the rank 4 partners are ranks 2, 3, 5 and 6, while in the 2-D case its neighbors are ranks 1, 3, 5 and 7.



6 Process decomposition: (a) 1-D with four neighbors; (b) 2-D with four neighbors

In our experiments, 1-D process decompositions produced up to a 20 percent advantage using MPI SHM versus point-to-point communications, depending on message size.

Finally, we modified the reported timing by adjusting it to the timing for a process with the biggest execution time. The current MPPTEST approach reports overall timing as a timing of a Rank 0, which might not be representative, especially in nonperiodic cases where Rank 0 typically has fewer neighbors than other processes.

Evaluation Environment and Results

In our performance studies, we used the Intel® Endeavor cluster, in which each node is equipped with dual Intel® Xeon® E5-2697 processors, one Intel® Xeon Phi™ 7120P-C0 coprocessor, and one Mellanox Connectx-3 InfiniBand* adapter connected to the same socket. The cluster was running Red Hat 6.5 Linux* OS, Intel® MPSS 3.3.30726, and OFED* 1.5.4.1. We used Intel MPI Library Version 5.0.2, Intel® C++ Compiler Version 15.0.1, and the MPPTEST benchmark with the modifications described in the previous section.

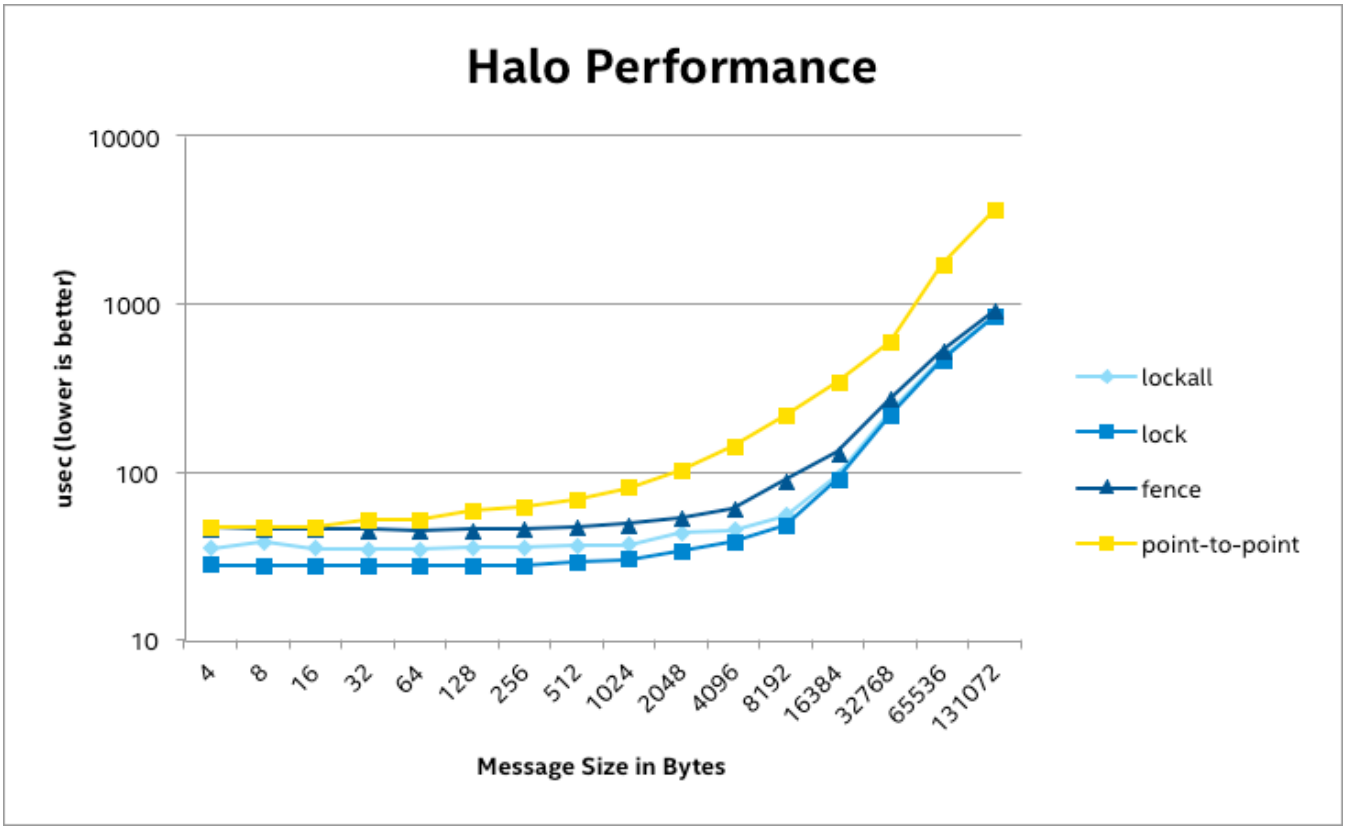
The following command line was used to obtain the performance data:

```
mpirun -n 64 -machinefile hostfile ./mpptest -halo -waitall -logscale -n_avg
1000  -npartner 8 -dimension 2
```

where the argument after `—halo` specifies the particular communication pattern for ghost cell exchanges (i.e., `—waitall` is used in the case of point-to-point messages; `—logscale` indicates

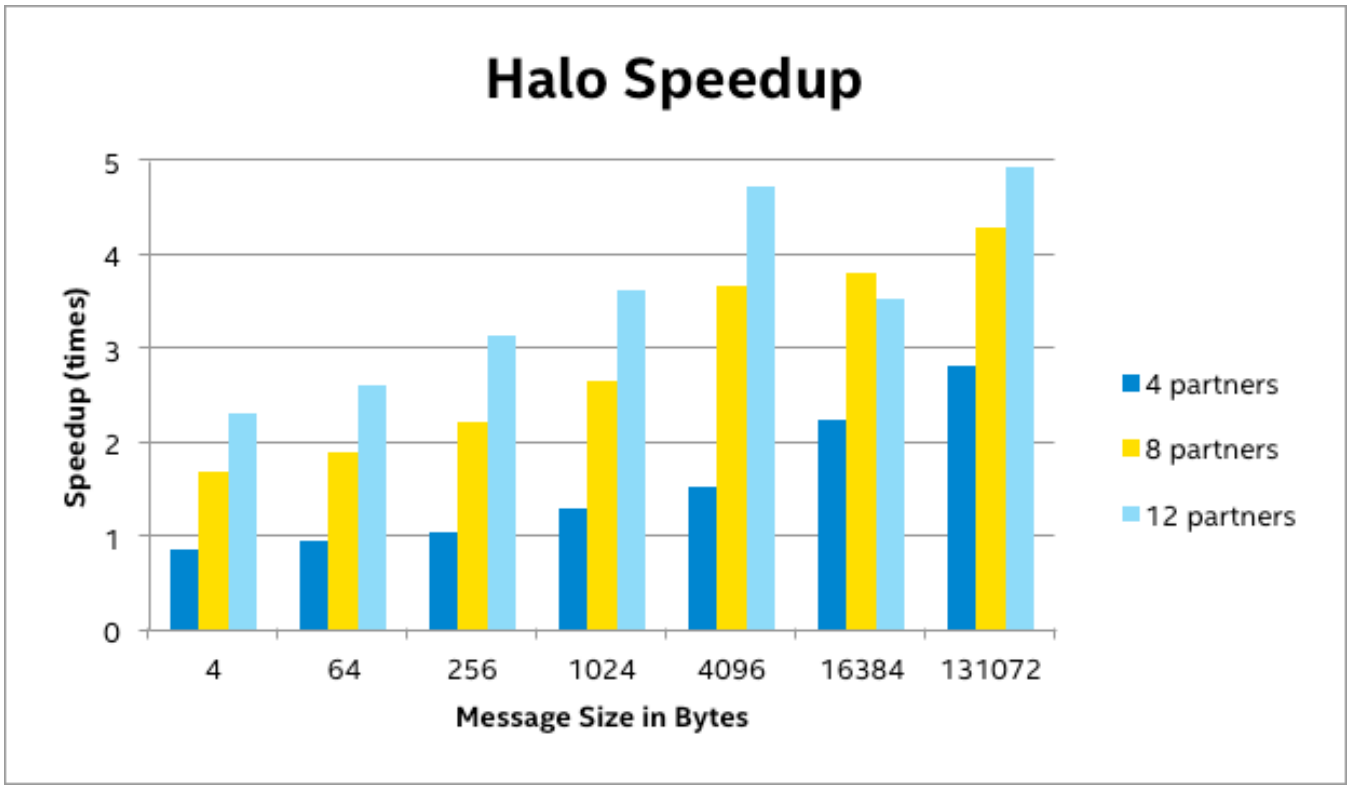
that we want to run the powers of two message sizes tests, starting from 4 bytes up to 128KB; `–n_avg` specifies the number of iterations to be used; and `–npartner` determines the number of neighbors per process). As described in the previous section, we introduced three new parameters corresponding to our new benchmarks (`–mpi3shm_lock`, `–mpi3shm_lockall` and `–mpi3shm_fence`) that can be used in place of `–waitall`. The `–dimension` parameter is optional (the default dimension is 2); this was also described in the last section.

Figure 7 shows the results obtained on one coprocessor with 32 processes and eight partners. In this case, the MPI SHM feature noticeably outperforms the regular point-to-point pattern regardless of synchronization type (please note the logarithmic scale of the y-axis). However, we should note that with a relatively small amount of updates (i.e., iterations in MPPTEST) the synchronization overhead based on locks might become crucial. This is because we do locking once per test, thus its contribution to the overall time is inverse to the number of iterations. Another observation is that using separate locks provides better performance than locking all the processes. This may become especially significant when the number of node neighbors to exchange the data with is significantly less than the number of processes bound to the interested window (thus, calling `MPI_Win_lock_all`/`MPI_Win_unlock_all` may lead to unnecessary communication with all the processes rather than to the neighbors only). Also, we see that using `MPI_Win_fence` gives the worst result of the sync primitives selected for this comparison.



7 Different halo patterns on one coprocessor with 32 processes and eight partners

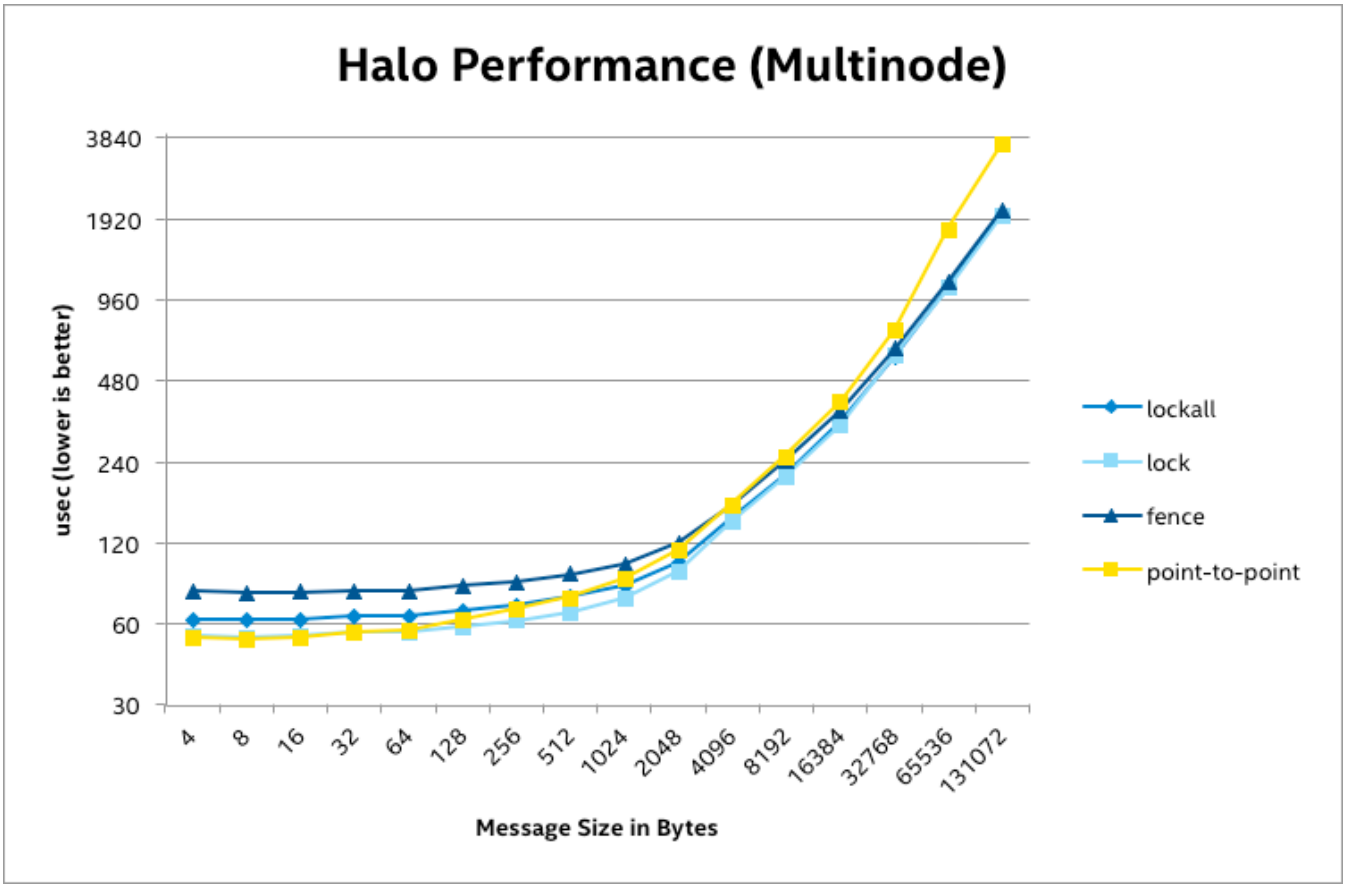
Then we analyzed how the number of neighbors in halo exchanges impacts overall performance. **Figure 8** shows the speedup of MPI SHM with lock synchronization in comparison to the common `MPI_Isend`/`MPI_Irecv` approach. We see that the performance advantage of our approach grows with the number of processes partners. This is expected because the relative cost of MPI SHM synchronizations stays the same regardless of the number of partners, while the performance advantage of simple memory copies compared to point-to-point operations grows with every other exchange. With 12 partners per process, we get up to 2.6x improvement with small message sizes and as much as 4.9x with relatively large message sizes.



8 Speedup of MPI SHM approach compared to the point-to-point based (measured on one coprocessor with 32 processes)

We repeated the measurements on two Intel Xeon Phi coprocessors connected to different nodes. We used 64 processes, 32 per coprocessor. The results depicted in **Figure 9** show lesser speedup than we observed on a single node. This is because some exchanges are done via the network, and the cost of intranode communication is just a part of the overall cost. We see that a personal lock-based shared memory approach is the best for almost all message sizes

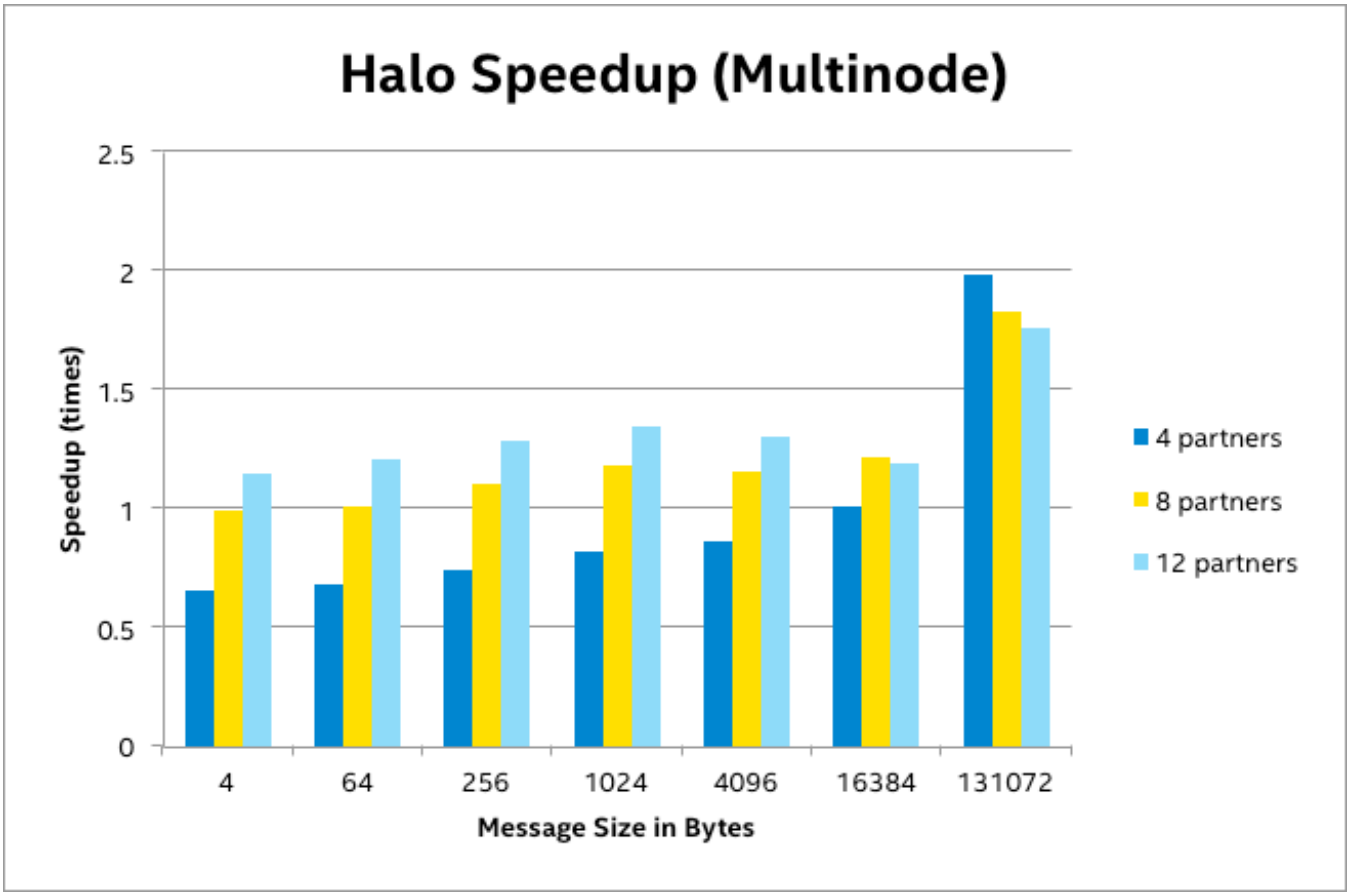
except very small messages, where the standard point-to-point scheme performs better. The experiments described so far have been done with default 2-D neighbors' topology. Using 1-D process topology, the personal locks-based MPI SHM approach also outperforms all other approaches at small message sizes. Also, starting from 4KiB messages, all shared memory-bound patterns outperform the point-to-point based ones.



9 Different halo patterns on two Intel® Xeon Phi™ coprocessors with 64 processes (32 per card) and eight partners

The speedup of the lock-based approach compared to the reference point-to-point one with different numbers of neighbors is shown in **Figure 10**. We see that with four partners, our approach is beneficial only above medium-sized messages. However, as it was with the one-node case, the performance benefit becomes more significant with a growing number of neighbors. With eight and 12 partners' processes, we get up to 1.2x improvement on small message sizes and 1.8x on big ones.

The preliminary studies with four and eight nodes using both Intel Xeon processors and Intel Xeon Phi coprocessors have shown similar results. Scaling with higher numbers of nodes and comparing hybrid MPI and OpenMP codes are left for future studies.



10 Speedup of MPI SHM approach compared to the point-to-point based ones (measured on two coprocessors with 64 processes)

Conclusion

In this article, we described the shared memory capabilities introduced in the MPI-3 standard. Because using this feature requires application modification, we demonstrated how to cope with it based on a simple 1-D ring "Hello World" example and extended it for several node runs. Using a modified MPPTEST benchmark, we managed to get up to 4.7x improvement over a standard point-to-point approach on one Intel Xeon Phi coprocessor. Moreover, we showed that the proposed approach may benefit halo exchanges even for multinode cases, and we obtained up to 1.8x improvement with two Intel Xeon Phi coprocessors.

Finally, our analysis indicates that it might be beneficial to use MPI SHM for ghost cell exchange-based applications, especially when there are larger numbers of halo exchange neighbors.

Acknowledgments

Thanks to Charles Archer for contributing a solution on how to apply MPI Groups to the multinode MPI SHM code, to Jim Dinan for many useful discussions, and to Robert Reed and Steve Healey for reviewing a draft of this article.

References

1. T. Hoefler et al., "Leveraging MPI's One-Sided Communication Interface for Shared-Memory Programming: Recent Advances in the Message Passing Interface," Proceedings of the 19th European MPI Users' Group Meeting (EuroMPI 2012), Vienna, Austria, Vol. 7490, Sept. 23–26, 2012.
2. T. Hoefler et al., "MPI+MPI: A New Hybrid Approach to Parallel Programming with MPI Plus Shared Memory," Computing (2013), Vol. 95, No. 12, p. 1,121.
3. M. Brinskiy et al., **"Mastering Performance Challenges with the new MPI-3 Standard,"** Parallel Universe Magazine Issue 18
4. B. Barney, **"Message Passing Interface Tutorial: Non-Blocking Message Passing Routines."**
5. W. D. Gropp et al., "Using Advanced MPI. Modern features of the Message-Passing Interface," MIT Press, November 2014.
6. W. D. Gropp and Rajeev Thakur, "Revealing the Performance of MPI RMA Implementations, PVM/MPI'07," Proceedings of the 14th European Conference on Recent Advances in Parallel Virtual Machine and Message Passing Interface, pp. 272–280.
7. Message Passing Interface Forum, **MPI: A Message-Passing Interface Standard Version 3.0**, University of Tennessee (Knoxville), Sept. 21, 2012.

For more information regarding performance and optimization choices in Intel® Software Products, visit **http://software.intel.com/en-us/articles/optimization-notice**.

Intel® MPI Library Conditional Reproducibility

By **Michael Steyer**, *Technical Consulting Engineer, Software and Services Group, Developer Products Division, Intel Corporation*

Introduction

High performance computing (HPC) users running numerical codes may experience cases where floating-point operations create slightly different results. Usually this would not be considered a problem, but due to the nature of such applications, differences can quickly propagate forward through the iterations and combine into larger differences.

In order to address these variations, the Intel® Compiler has several switches that manipulate floating-point precision, while the Intel® Math Kernel Library (Intel® MKL) Conditional Numerical Reproducibility (CNR) feature[1] provides functions for obtaining reproducible floating-point results. Also, deterministic reduction algorithms are available for Intel® OpenMP and **Intel® Threading Building Blocks** (Intel® TBB) runtimes. Some of the collective operations

of the **Intel® MPI Library**, however, might also lead to slight differences in their results. This article will address methods that can be used to gather conditionally reproducible results from collective operations of the Intel MPI Library.

Motivation

Let's have a look at a simple example with 64 MPI ranks calling an `MPI_Reduce` operation where double precision values are accumulated.

Figure 1 shows the Fortran code that calls an `MPI_Reduce` operation. Each MPI rank writes a very small number (2^-60) to its `local_value` variable—except where Rank #16 (Index 15) writes 1.0 and Rank #17 (Index 16) writes -1.0. All `local_value` fields from the different ranks will then be accumulated to a global sum using `MPI_Reduce`. After the reduction operation, Rank 0 will write out `global_sum` with up to 20 digits after the decimal point.

```
program rep
  use mpi
  implicit none
  integer :: n_ranks,rank,errc
  real*8 :: global_sum,local_value

  call MPI_Init(errc)
  call MPI_Comm_size(MPI_COMM_WORLD, n_ranks, errc)
  call MPI_Comm_rank(MPI_COMM_WORLD, rank, errc)

  local_value = 2.0 ** -60

  if(rank.eq.15) local_value= +1.0
  if(rank.eq.16) local_value= -1.0

  call MPI_Reduce(local_value,global_sum,1,MPI_DOUBLE_PRECISION, &
         MPI_SUM,0,MPI_COMM_WORLD, errc)

  if(rank.eq.0) write(*,'(f22.20)') global_sum

  call MPI_Finalize(errc)
end program rep
```

1 Fortran 90 accumulation example

Assume we have four nodes available, in which each system has 32 processor cores. Since we can run our application with only two systems, let's consider two different distributions schemas of MPI ranks:

A) 64 ranks across all four nodes => 16 ranks per node

B) 64 ranks on only two nodes => 32 ranks per node

Due to its highly optimized nature, Intel MPI Library will try to leverage distributed and shared memory resources as efficiently as possible. Depending on the size of the run (#MPI ranks) and the message sizes that have to be exchanged, the library can choose among different algorithms available for each collective operation. Choosing the topologically aware algorithm for the reduce operation may result in a different order of operation for cases A and B.

To reduce load on the cluster interconnect, the algorithm would accumulate local (per node) operations first and then send these results only once through the cluster network in order to accumulate the final result.

A) Reduce(Reduce(#1 – #16) + Reduce(#17 – #32) + Reduce(#33 – #48) + Reduce(#49 – #64))

B) Reduce(Reduce(#1 – #32) + Reduce(#33 – #64))

The associative law "(a + b) + c = a + (b + c)" assumes exact computations and effectively unlimited precision; therefore, it does not apply when using limited precision representations. Since floating-point numbers are approximated by a limited number of bits representing the value, operations on these values will frequently introduce rounding errors. For a sequence of floating-point operations, the total rounding error can depend on the order in which these operations are executed.[2]

The Intel MPI Library offers algorithms to gather conditionally reproducible results, even when the MPI rank distribution environment differs from run to run.

As a result of the different order of operations in cases A and B, the final Reduce could generate slightly different values.

While the results could be slightly different, they are still valid according to the IEEE 754 floating-point standard.[3] Let's break down the distribution of ranks for cases A and B from a pure floating-point perspective. This will provide a clearer picture of the actual problem:

A) ((... + 2^-60 + (+1)) + ((-1) + 2^-60 + ...) + ...

B) ((... + 2^-60 + (+1) + (-1) + 2^-60 + ...) + ...

In case A, +1 and -1 have to be accumulated with the very small 2^-60 values. In case B, +1 and -1 will be eliminated since they're calculated in the same step.

Depending on the Intel MPI Library runtime configuration (shown in **Table 1**), this can result in the output in **Figure 2**.

```
$ cat ${machinefile_A}
ehk248:16
ehs146:16
ehs231:16
ehs145:16
$ cat ${machinefile_B}
ehk248:32
ehs146:32
ehs231:0
ehs145:0
$ mpiifort -fp-model strict –o ./rep.x ./rep.f90
$ export I_MPI_ADJUST_REDUCE=3
$ mpirun -n 64 -machinefile ${machinefile_A} ./rep.x
0.00000000000000000000
$ mpirun -n 64 -machinefile ${machinefile_B} ./rep.x
0.00000000000000004163
```

2 Getting diverse floating-point results

Preparation

Before addressing Intel MPI Library reproducibility, we should make sure that all other parts of the application produce numerically stable results.

For example, the OpenMP standard, as a frequently used hybrid threading extension to MPI, does not specify the order in which partial sums should be combined. Therefore, the outcome of a reduction operation in OpenMP can vary from run to run depending on the runtime parameters. The Intel OpenMP runtime provides the environment variable `KMP_DETERMINISTIC_REDUCTION`, which can be used to control the runtime behavior.[4] Also, the Intel® TBB Library does support deterministic reductions using the "`parallel_deterministic_reduce`" function.[5]

Read more about using both the Intel Compiler and Intel MKL in the article "Using the Intel Math Kernel Library and Intel Compilers to Obtain Run-to-Run Numerical Reproducible Results."[6]

Reproducibility

To explicitly set the expectations, we need to differentiate the terms *reproducible* and *repeatable*. Furthermore, when we use the term *reproducible*, we always mean conditional reproducibility.

Repeatable	Provides consistent results if the application is launched under exactly the same conditions—repeating the run on the same machine and configuration.
Reproducible (conditional)	Provides consistent results even if the distribution of ranks differs, while the number of ranks (and #threads for hybrid applications) involved has to be stable. Also, the runtime including the microarchitecture has to be consistent.[7]

All Intel MPI Library operations guarantee repeatable results.

The reproducibility of Intel MPI Library operations is guaranteed under the following conditions:

1. Do not use topologically aware algorithms inside the collective reduction operations.
2. Avoid the recursive doubling algorithm for the `MPI_Allreduce` operation.
3. Avoid `MPI_Reduce_scatter_block`—as well as the MPI-3 nonblocking-collective operations.

The **first condition** for reproducibility can be met by explicitly setting the corresponding collective reduction operation algorithm using the `I_MPI_ADJUST_` environment variables. A detailed documentation can be found in the *Intel MPI Library Reference Manual*[8] in the "Collective Operation Control" chapter. The information provided in the document clearly states which algorithms are topologically aware and should be avoided.

Table 1 shows the five collective operations, which use reductions, and the corresponding Intel MPI Library environment variables. Set these accordingly in order to leverage the nontopologically aware algorithms (fulfilling the first condition above):

Table 1

Collective MPI Operation Using Reductions	Intel MPI Collective Operation Control Environment	Nontopologically Aware Algorithms
MPI_Allreduce	I_MPI_ADJUST_ALLREDUCE	(1)[a], 2 , 3 , 5 , 7 , 8, 9[b]
MPI_Exscan	I_MPI_ADJUST_EXSCAN	1
MPI_Reduce_scatter	I_MPI_ADJUST_REDUCE_SCATTER	1 , 2 , 3 , 4
MPI_Reduce	I_MPI_ADJUST_REDUCE	1 , 2 , 5, 7[a]
MPI_Scan	I_MPI_ADJUST_SCAN	1

a Keep in mind that while the first algorithm of `MPI_Allreduce` is not topologically aware, it does not guarantee conditionally reproducible results—see the second condition for details.

b The Knomial algorithm (IMPI ≥ 5.0.2) provides reproducible results, only if the `I_MPI_ADJUST_<COLLECTIVE-OP-NAME>_KN_RADIX` environment is kept stable or unmodified.

To see which algorithms are currently selected, set the environment variable `I_MPI_DEBUG=6` and review the output. The default algorithms for collective operations can differ, depending on the size of the run (#ranks) as well as the transfer message sizes. **Figure 3** shows the debug output for the collective operations used in the simple MPI reduce application introduced earlier.

```
…
[0] MPI startup(): Reduce_scatter: 4: 0-2147483647 & 257-512
[0] MPI startup(): Reduce_scatter: 4: 0-5 & 513-2147483647
[0] MPI startup(): Reduce_scatter: 5: 5-307 & 513-2147483647
[0] MPI startup(): Reduce_scatter: 1: 307-1963 & 513-2147483647
[0] MPI startup(): Reduce_scatter: 3: 1963-2380781 & 513-2147483647
[0] MPI startup(): Reduce_scatter: 4: 0-2147483647 & 513-2147483647
[0] MPI startup(): Reduce: 1: 0-2147483647 & 0-2147483647
[0] MPI startup(): Scan: 0: 0-2147483647 & 0-2147483647
[0] MPI startup(): Scatter: 1: 1-494 & 0-32
[0] MPI startup(): Scatter: 2: 495-546 & 0-32
[0] MPI startup(): Scatter: 1: 547-1117 & 0-32
[0] MPI startup(): Scatter: 3: 0-2147483647 & 0-32
[0] MPI startup(): Scatter: 1: 1-155 & 33-2147483647
…
```

3 Example of selected collective operations

One can see that for the `MPI_Reduce` collective operation, the first algorithm is being selected across all message sizes (0–2147483647) and ranges of MPI ranks (0–2147483647) by default. This is why it was necessary to select a different topology-aware algorithm (3) for the example above in order to get differing results for the MPI reduction (`I_MPI_ADJUST_REDUCE=3`).

The **second condition** can be met by avoiding the recursive doubling algorithm for the `MPI_Allreduce` operation (`I_MPI_ADJUST_ALLREDUCE=1`). While the order of MPI ranks is guaranteed to be stable, the order of operands inside each MPI rank can differ due to the applied optimizations.

If, however, the operation is covered by the commutative law "a + b = b + a," even the recursive doubling algorithm can be used to achieve reproducible results.

The **third condition** is necessary since the `MPI_Reduce_scatter_block`—as well as the new MPI-3[9] nonblocking-collective operations—is implemented by using topology-aware algorithms. These collective operations cannot be adjusted by the Intel MPI Library user (as of Version 5.0.2), as they are only determined at runtime based on certain operation parameters.

In **Figure 4**, we show how to achieve reproducible results for the simple reduction example used in the Motivation section of this article. Therefore, we will apply a nontopology-aware collective operation algorithm in the Intel MPI Library environment.

As we have seen in **Figure 3**, the first algorithm was already the default case. Another option here was not specifying any `I_MPI_ADJUST_REDUCE` environment at all and leaving the default settings intact.

```
$ cat ${machinefile_A}
ehk248:16
ehs146:16
ehs231:16
ehs145:16
$ cat ${machinefile_B}
ehk248:32
ehs146:32
ehs231:0
ehs145:0
$ mpiifort -fp-model strict –o ./rep.x ./rep.f90
$ export I_MPI_ADJUST_REDUCE=1
$ mpirun -n 64 -machinefile ${machinefile_A} ./rep.x
0.00000000000000004163
$ mpirun -n 64 -machinefile ${machinefile_B} ./rep.x
0.00000000000000004163
```

4 Getting reproducible floating-point results

Keep in mind that while the distribution of MPI ranks along the nodes changed, all other parameters, such as the number of ranks and the architecture used, have been kept stable. This is necessary, as according to the definition of conditional reproducibility, the runtime environment has to be the same.

Intel® Xeon Phi™ Coprocessor

When discussing conditional reproducibility for the Intel MPI Library, there is no difference between treatment for an Intel® Xeon® processor and an Intel® Xeon Phi™ coprocessor. The same considerations we discussed apply to both. This allows the user to transparently integrate the Intel Xeon Phi coprocessor into HPC solutions.

Remember, however, that different microarchitectures/instruction sets also come with different hardware-rounding support, which can lead to different results between the two microarchitectures. Also, as defined in the Reproducibility section of this article, the conditions have to be the same and, therefore, the number of threads and MPI ranks have to be stable.

Summary

In this article, we have shown several methods to enable the Intel MPI Library to use algorithms that guarantee deterministic reductions for the different collective MPI operations.

We also demonstrated the impact of such algorithms, using a simple example of an MPI reduce operation moving from a repeatable to a conditionally reproducible outcome. This has been achieved without any need to modify the application's source code.

The Intel MPI Library offers algorithms to gather conditionally reproducible results, even when the MPI rank distribution environment differs from run to run. It is important to understand that all other parameters, like the number of ranks or the microarchitecture, have to be equal from run to run. This is necessary in order to fulfill the requirements for conditionally reproducible results.

End Notes

1. T. Rosenquist, "**Introduction to Conditional Numerical Reproducibility (CNR)**," Intel Corporation, 2012.
2. D. Goldberg, "What Every Computer Scientist Should Know About Floating-Point Arithmetic," Association for Computing Machinery, Inc., 1991.
3. *IEEE Standard for Binary Floating-Point Arithmetic,* The Institute of Electrical and Electronics Engineers, Inc., 1985.
4. M.J. Corden and D. Kreitzer, "**Consistency of Floating-Point Results using the Intel® Compiler**," Intel Corporation, 2012.

BLOG HIGHLIGHTS

Tuning Tips for Compute Offload to Intel® Processor Graphics

BY ANOOP MADHUSOODHANAN PRABHA »

Below are some tuning tips, which will help the programmer tune his kernel to get better performance from processor graphics:

- Offloaded loop nests must have enough iterations for all hardware threads available on Processor Graphics. Using perfectly nested parallel _Cilk_for loops allows parallelization in the dimensions of the parallel loop nest.
- Pragmas and code restructuring can be employed to get offloaded code vectorized.
- Using __restrict and __assume_aligned keywords may help vectorization too.
- Using the pin clause of the offload pragma will eliminate data copying to/from the GPU.
- Scalar memory accesses are much less efficient than vector accesses. Using Intel® Cilk™ Plus array notation for memory accesses may help vectorize computation. A single memory access can handle up to 128 bytes. Gather/scatter operations of 4-byte elements are quite efficient, but with 2-byte elements are slower. Gather/scatter operations may result from array sections with non-unit strides.

Read more ›

5. A. Katranov, "**Deterministic Reduction: A New Community Preview Feature in Intel® Threading Building Blocks**," 2012.

6. T. Rosenquist and S. Story, "**Using the Intel Math Kernel Library and Intel Compilers to Obtain Run-to-Run Numerical Reproducible Results**," Intel® Parallel Universe Magazine, 2012.

7. Even if the target application is compiled for one single vector instruction set such as AVX, running it on different microarchitectures such as Sandy Bridge or Haswell might trigger libraries to utilize different vector instruction sets based on the available microarchitecture. See "Consistency of Floating-Point Results using the Intel® Compiler"[3] for more information.

8. **Intel® MPI Library—Documentation**, Intel Corporation, 2015.

9. "**MPI: A Message-Passing Interface Standard—Version 3.0**," Message-Passing Interface Forum, 2012.

Intel® MPI Memory Consumption

By **Dmitry Durnov**, *Senior Software Engineer,* and **Michael Steyer**, *Technical Consulting Engineer, Software and Services Group, Intel Corporation Developer Products Division*

Introduction

High performance computing (HPC) applications tend to consume most of the available system memory on a node; therefore, it is useful to deal with the limited memory resources on a cluster thoughtfully. However, in order to provide the maximum amount of dedicated memory to an application, other memory-consuming parts on a cluster must be taken into account. In particular, the operating system and libraries that are used need to be understood. As the memory consumption of the message passage interface (MPI) library grows with the job size, along with the number of MPI ranks, estimating the memory footprint becomes rather complex.

This article will serve as an orientation about the estimated memory consumption for the **Intel® MPI Library**, using different fabrics. (The authors cannot offer a byte-accurate prediction model because actual memory consumption depends on the operating system environment.) This article will also help users fine-tune the Intel MPI Library settings for a reduced memory footprint.

The Memory of a Library

Memory consumption analysis is a complex task, as there are many sources that may influence the memory footprint. Even though it is possible to predict how much virtual memory will be allocated, the amount of physical memory used will depend on the characteristics of the application that is utilizing the library and the operating system configuration.

The memory-consuming parts of the Intel MPI Library can be split into two categories: parts that scale with the number of MPI processes involved, and parts that have a fixed memory footprint. The largest fraction of memory comes from needs that scale consumption with the job size.

BLOG HIGHLIGHTS

Intel® Math Kernel Library Inspector-Executor Sparse BLAS Routines

BY ZHANG Z »

Intel® Math Kernel Library (Intel® MKL) 11.3 Beta, released April 2015, offers the inspector-executor API for Sparse BLAS (SpMV 2). This API divides operations into two steps. During an initial analysis stage, the API inspects the matrix sparsity pattern and applies matrix structure changes. In subsequent routine calls, this information is reused in order to improve performance.

This inspector-executor API supports key Sparse BLAS operations for iterative sparse solvers, and covers all the functionality available in the classic Sparse BLAS implementation available in Intel MKL:

- Sparse matrix-vector multiplication
- Sparse matrix-matrix multiplication with sparse or dense result
- Triangular system solution
- Sparse matrix addition

Read more ›

There are several reasons the prediction for a library's memory consumption depends on the platform (operating system and architecture). One factor is the operating system's choice of page sizes. The page size will influence the alignment and size of buffers that are used for sending and receiving messages among different MPI processes. Another factor is use-diverse middleware. Some middleware libraries are used at a very low level and are therefore part of the memory consumers that scale with job size. Different versions of such libraries may have different buffer sizes for each connection. This can also lead to a varying memory footprint for the Intel MPI Library.

To best address these challenges of the memory-consumption prediction, this article will focus on the worst-case scenario: an all-to-all connection. In this scenario, each MPI rank has an active connection to every other MPI rank, and therefore **n^2** active connections overall. Furthermore, the message sizes have been set close to the maximum internal MPI eager buffers, while the memory footprint of larger message transfers will be reflected by the application. This way, almost all allocated memory will be used. If a buffer in virtual memory is not filled completely, it might not require an equal amount of physical memory in a machine.

All the graphs in this article focus on the worst-case Intel MPI Library memory consumption per rank. The amount of memory consumed per node will therefore depend on the number of ranks being used on each node. Each graph includes a dotted line indicating the memory consumption of 64GB per node, as an orientation for a system with 28 MPI ranks per node, such as a Haswell EP. The memory consumption estimates are based on the internals of Intel® MPI Library Version 5.0.3.048.

Please note that in real-life applications that do not need all-to-all connection patterns, the Intel MPI memory footprint will be much smaller. Therefore, the estimations provided in this article can be considered worst-case assumptions. In most applications, the dynamic connections establishment logic of the Intel MPI Library, which is present in almost all transport methods supported, will hold only a minimal number of necessary connections per rank.

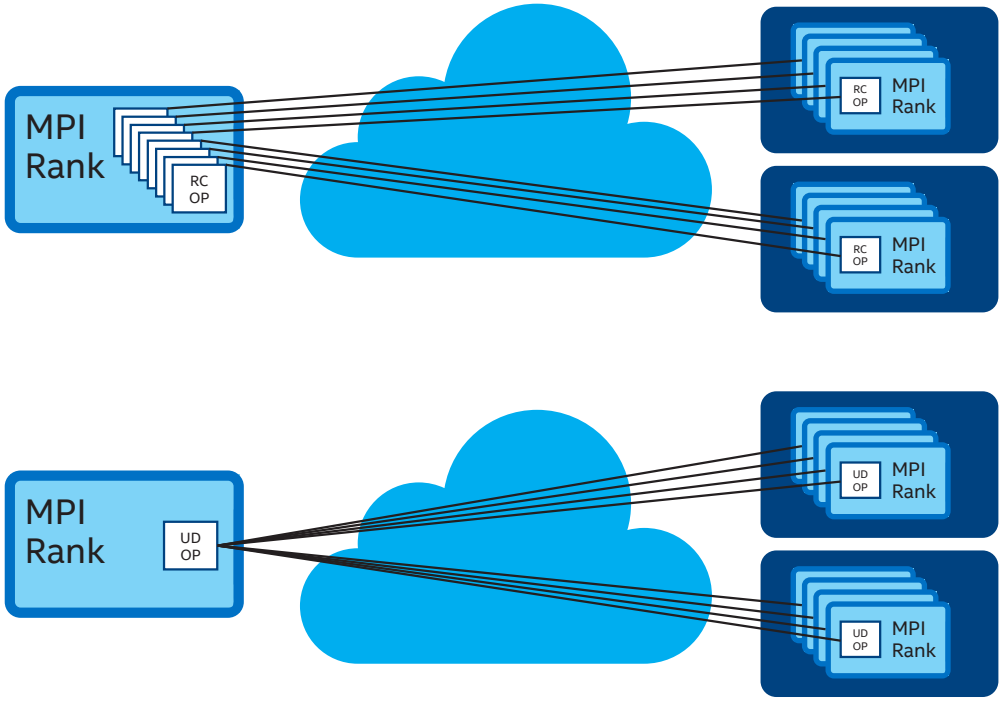
Intel® MPI Fabrics and Memory

The Intel MPI Library supports the following low-level transport mechanisms:

- **Shared Memory (SHM).** Used for intra-node message transport. It provides several configurations for memory consumption/latency trade-offs and supports architecture-specific optimizations for the latest processors.
- **Direct Access Programming Library (DAPL) (Reliable Connection [RC] and User Datagram [UD]).** Based on the User Direct Access Programming Library (uDAPL). uDAPL provides a high level of flexibility for the hardware utilization of different vendors. Vendor- and technology-specific features are transparent to the uDAPL level.
- **OFA.** This transport layer is based on the direct usage of IB-verbs RC.
- **Tag Matching Interface (TMI).** Based on the Performance Scaled Messaging (PSM) API, which is the main API for Intel® True Scale Fabric hardware.
- **TCP.** Based on TCP sockets and also applicable to solutions such as IPoIB.

Additionally, the **Intel MPI Library** supports combinations of these fabrics, which can be used to separate a fabric for intra- as well as inter-node communication.

When it comes to Intel MPI Library memory consumption, however, these fabrics behave very differently. Due to the nature of DAPL RC, where each rank holds a QP (Queue Pair) buffer and transport buffers for each other rank (n), the per-rank memory consumption for DAPL RC grows linearly to the number of MPI ranks (n * number of buffers per connection). DAPL UD, in contrast, performs better at scale, while it only utilizes one QP as well as one common pool of buffers for all ranks (n * buffer pool size). **Figure 1** illustrates the differences between DAPL RC and DAPL UD.

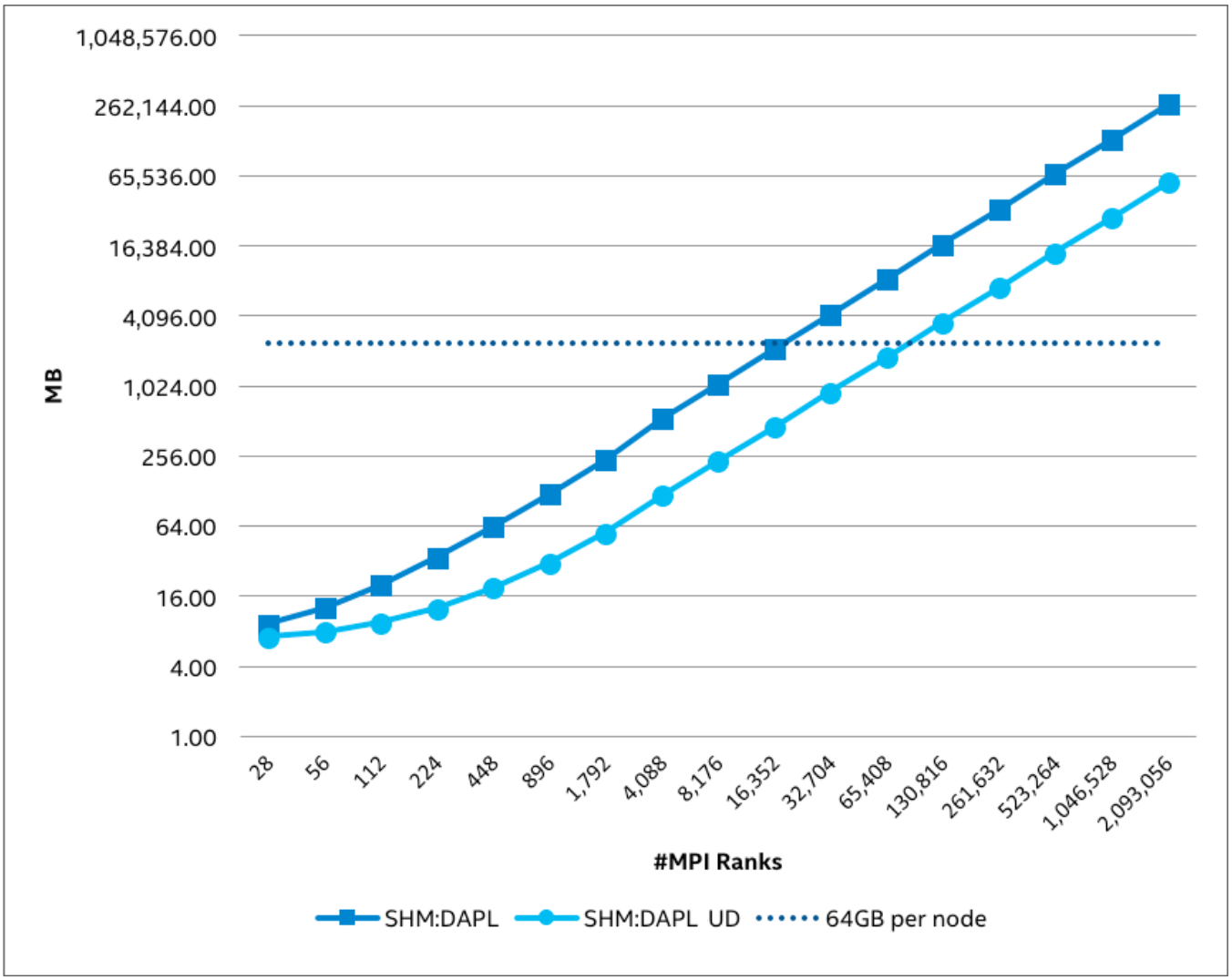


1 DAPL RC (top) vs. DAPL UD (bottom)

The use of connectionless communication in DAPL UD also has disadvantages, such as the lack of RDMA support (although it is available in DAPL UD + DAPL RC mixed mode). Also, the potentially longer delay in transfers, due to the segmentation and reassembly of each message, can be a disadvantage. However, the large memory savings, compared to DAPL RC, show its performance benefits on a large scale of MPI ranks.

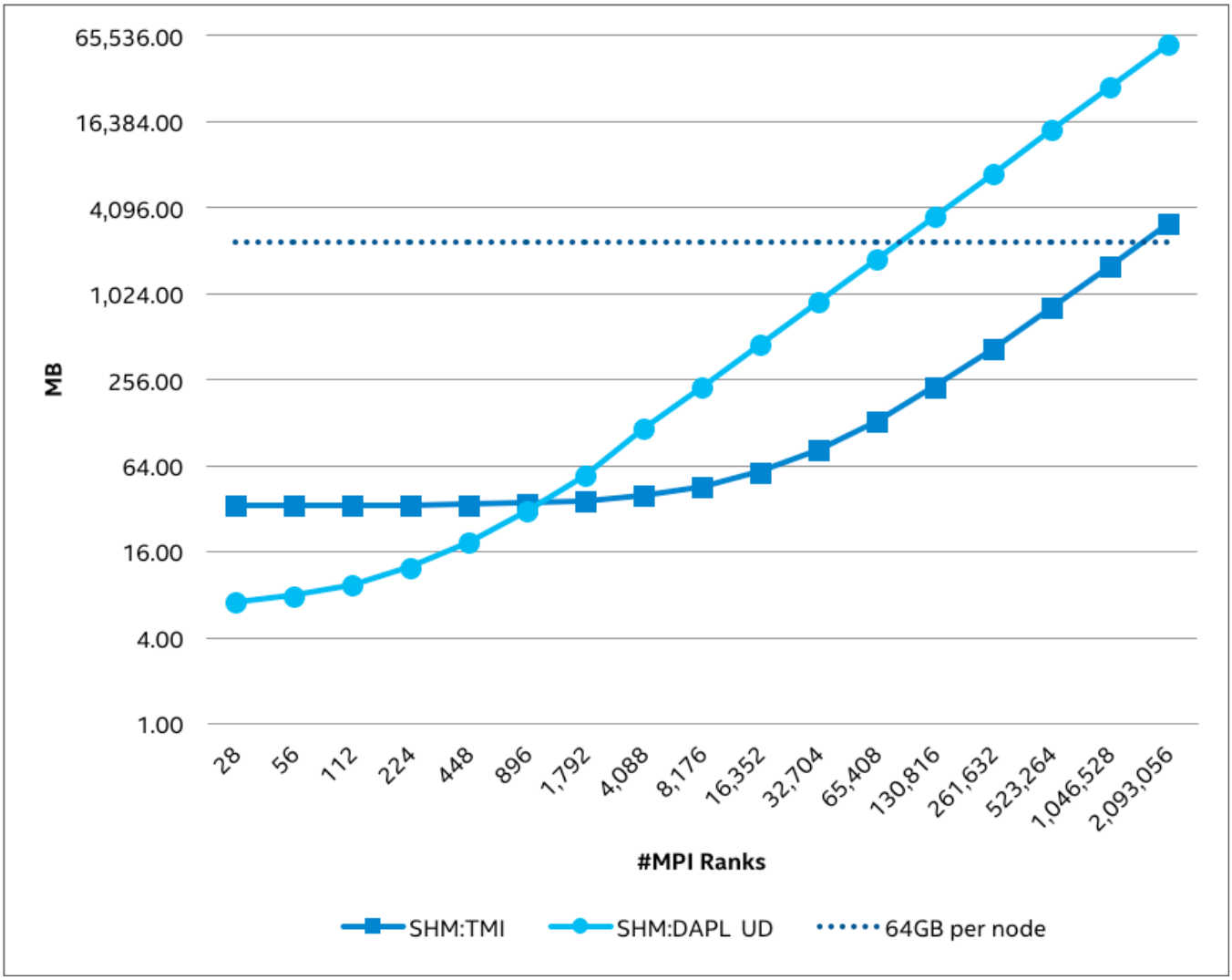
Figure 2 shows the Intel MPI Library's memory consumption difference between DAPL RC and DAPL UD in the all-to-all connections scenario. Please keep in mind that these estimations are unlikely due to the worst-case representation. Therefore, these numbers can be considered an upper limit rather than an exact memory consumption model of the Intel MPI Library.

For our validations, we forced the `MPI_Alltoall` operation and utilized the pairwise exchange algorithm in order to get the highest memory footprint by setting the `I_MPI_ADJUST_ALLTOALL` environment to "3."



2 Memory consumption of DAPL RC vs. DAPL UD (lower is better)

TMI is an API used by Intel MPI Library to get benefits from low-level transports, which provide their own messages matching logic. The main technology currently used below TMI is PSM, available on Intel® True Scale Fabric and other adapters of the Intel® Omni-Path Architecture family. This technology bypasses the IB-verbs layer and has a fixed-memory footprint for transport buffers. The amount of memory required per connection was further reduced in the latest Intel MPI Library versions in order to align with PSM as a scalable fabric solution.



3 Memory consumption of DAPL UD vs. TMI (lower is better)

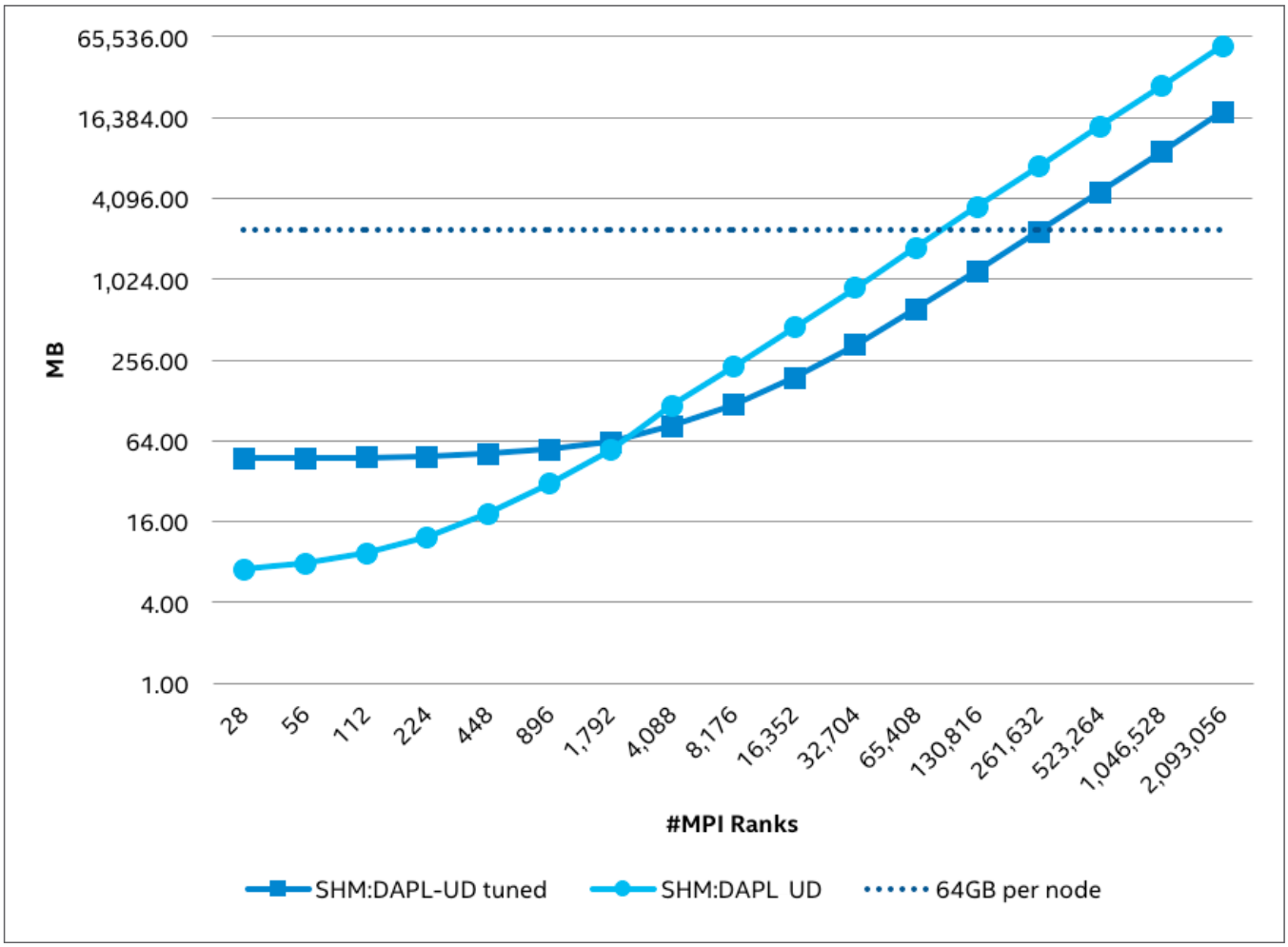
Large-Scale Memory Tuning

While DAPL UD already works in a highly memory-conservative fashion, one can further tune the fabric for memory efficiency on large-scale MPI runs. **Table 1** shows some default environment settings versus their tuned versions.

Table 1. DAPL UD default vs. tuned environment variables

Environment Variable	Default Value	Tuned Value
I_MPI_DAPL_UD_SEND_BUFFER_NUM	Runtime dependent	8208
I_MPI_DAPL_UD_RECV_BUFFER_NUM	Runtime dependent	8208
I_MPI_DAPL_UD_ACK_SEND_POOL_SIZE	256	8704
I_MPI_DAPL_UD_ACK_RECV_POOL_SIZE	Runtime dependent	8704
I_MPI_DAPL_UD_RNDV_EP_NUM	4	2

These environment settings set the number of internal DAPL UD buffers to a fixed value, while the default size of DAPL UD buffer pools scales along with the number of MPI ranks. At the same time, the tuning parameters fix the memory required for lower-level QP-related buffers. The impact of these settings is especially interesting for large-scale MPI runs, as one can see in **Figure 3**.



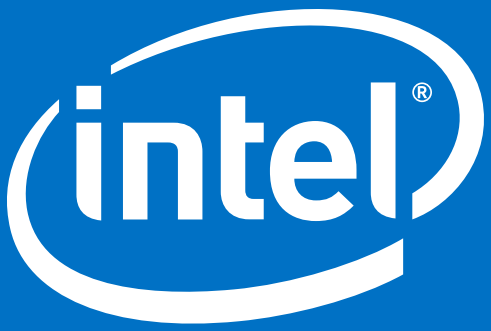
4 Memory consumption DAPL UD default vs. DAPL UD tuned (lower is better)

Summary

In this article, we have shown the worst-case memory-consumption of the Intel MPI Library on different fabrics. This information can serve the user as an upper-bound memory-consumption estimation despite not being a byte-accurate prediction. With this information, users will be able to determine how much memory will be left at a certain scale for their HPC applications.

Users with Mellanox interconnects on their cluster should focus on the DAPL UD fabric to reduce the Intel MPI memory footprint for large-scale runs. While this article did not consider the memory consumption using the OFA fabric, it would still provide a good memory scalability and end up somewhere in between DAPL RC and DAPL UD for memory.

Users also can further reduce the memory consumption of the DAPL UD fabric by modifying the Intel MPI internal buffer structure. The best memory-consumption scalability of the Intel MPI Library can be observed using the TMI fabric on Intel True Scale Fabric and future Intel Omni-Path interconnects.



Software

The Parallel Universe