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]

Sign up for future issues | Share with a friend

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);
```

Sign up for future issues │ Share with a friend

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);
```
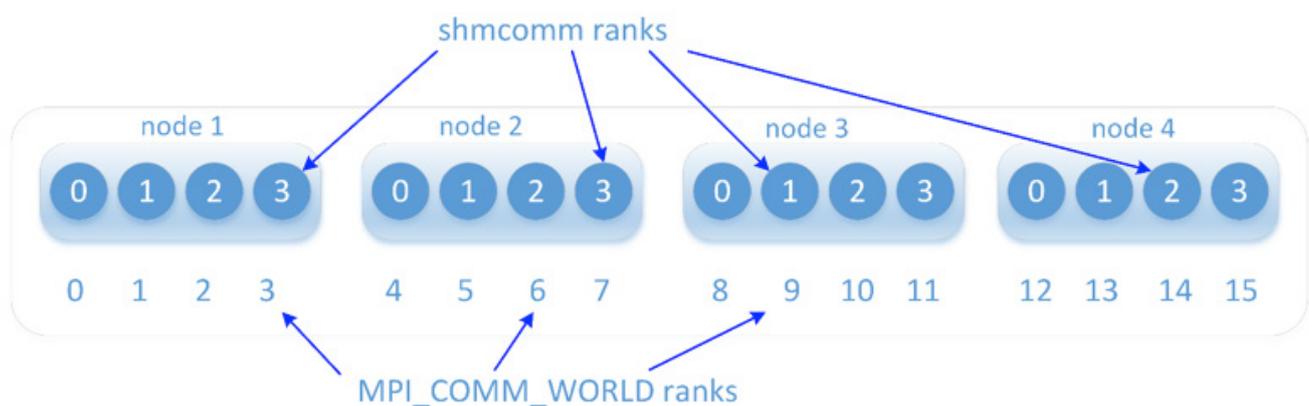

2 Mapping of global ranks to shmcomm ranks. If some of the neighboring ranks are residing on a different node, their mapping in the resulting array partners_map will be a predefined constant, MPI_UNDEFINED.

Sign up for future issues | Share with a friend

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**).

Sign up for future issues | Share with a friend

```
//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

Sign up for future issues | Share with a friend

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

Sign up for future issues | Share with a friend

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.

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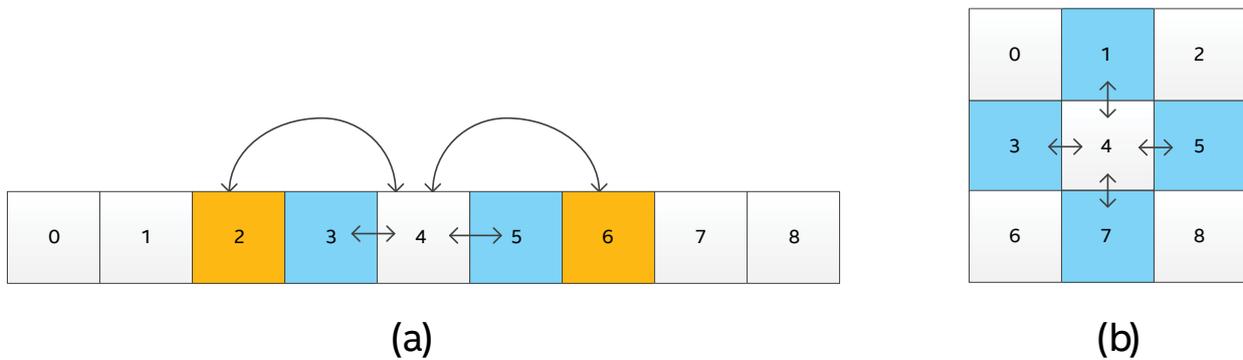
6 Process decomposition: (a) 1-D with four neighbors; (b) 2-D with four neighbors

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

Sign up for future issues | Share with a friend

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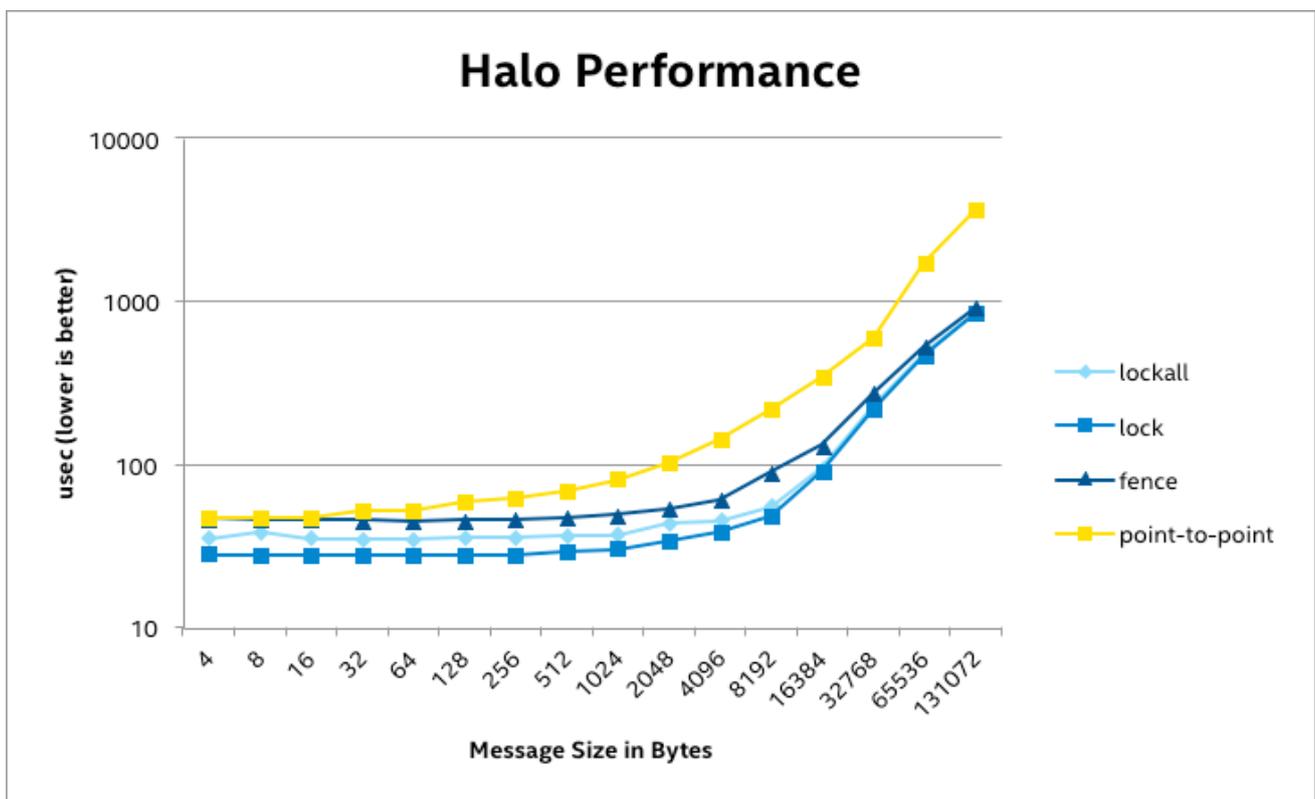
7 Different halo patterns on one coprocessor with 32 processes and eight partners

Sign up for future issues | Share with a friend

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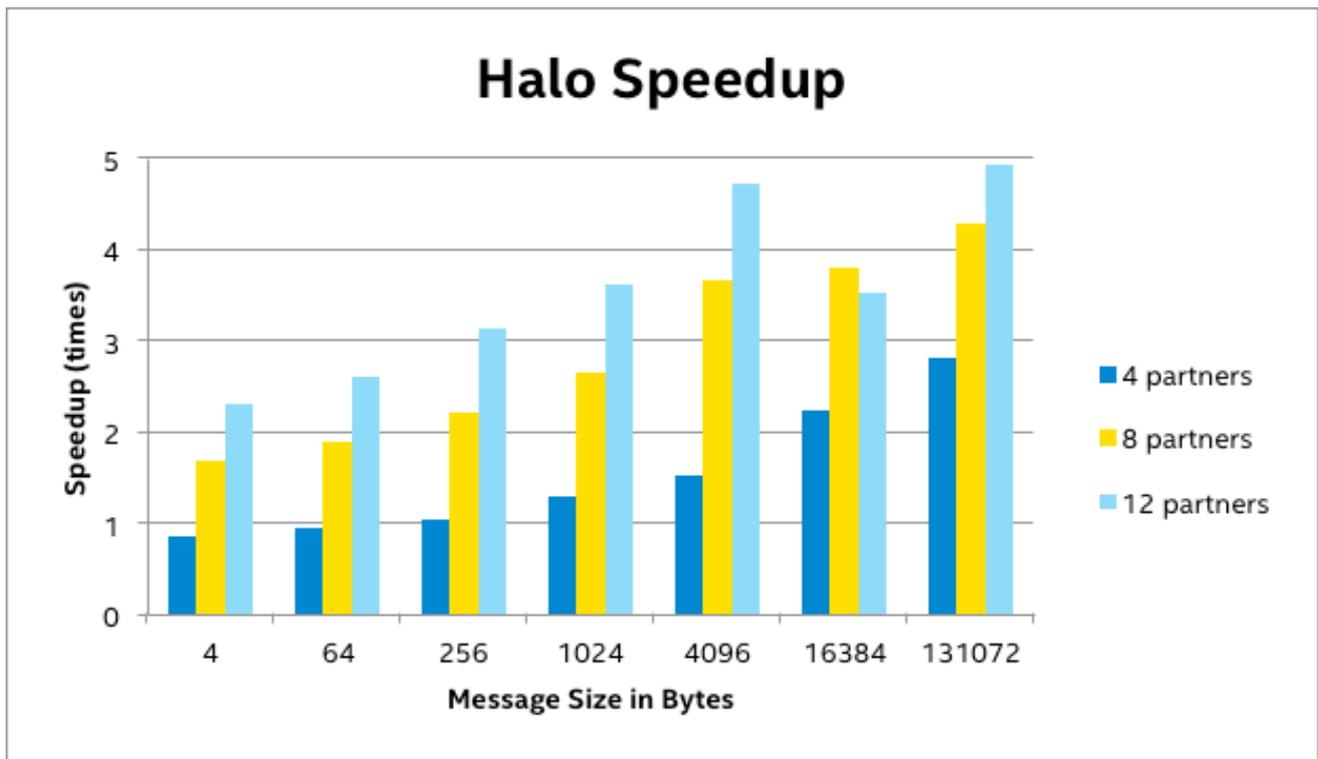
8   Speedup of MPI SHM approach compared to the point-to-point based (measured on one coprocessor with 32 processes)

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

Sign up for future issues | Share with a friend

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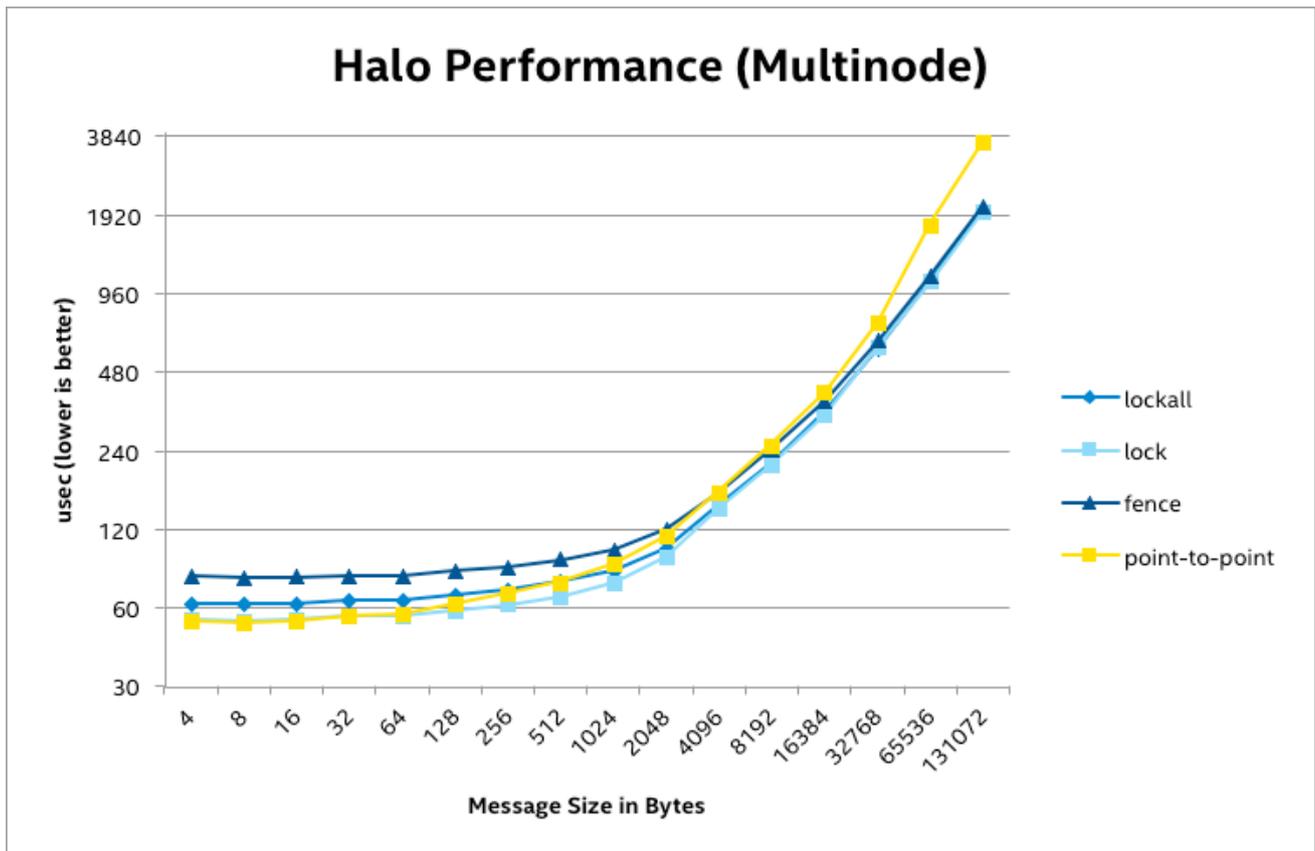
9   Different halo patterns on two Intel® Xeon Phi™ coprocessors with 64 processes (32 per card) and eight partners

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.

Sign up for future issues | Share with a friend
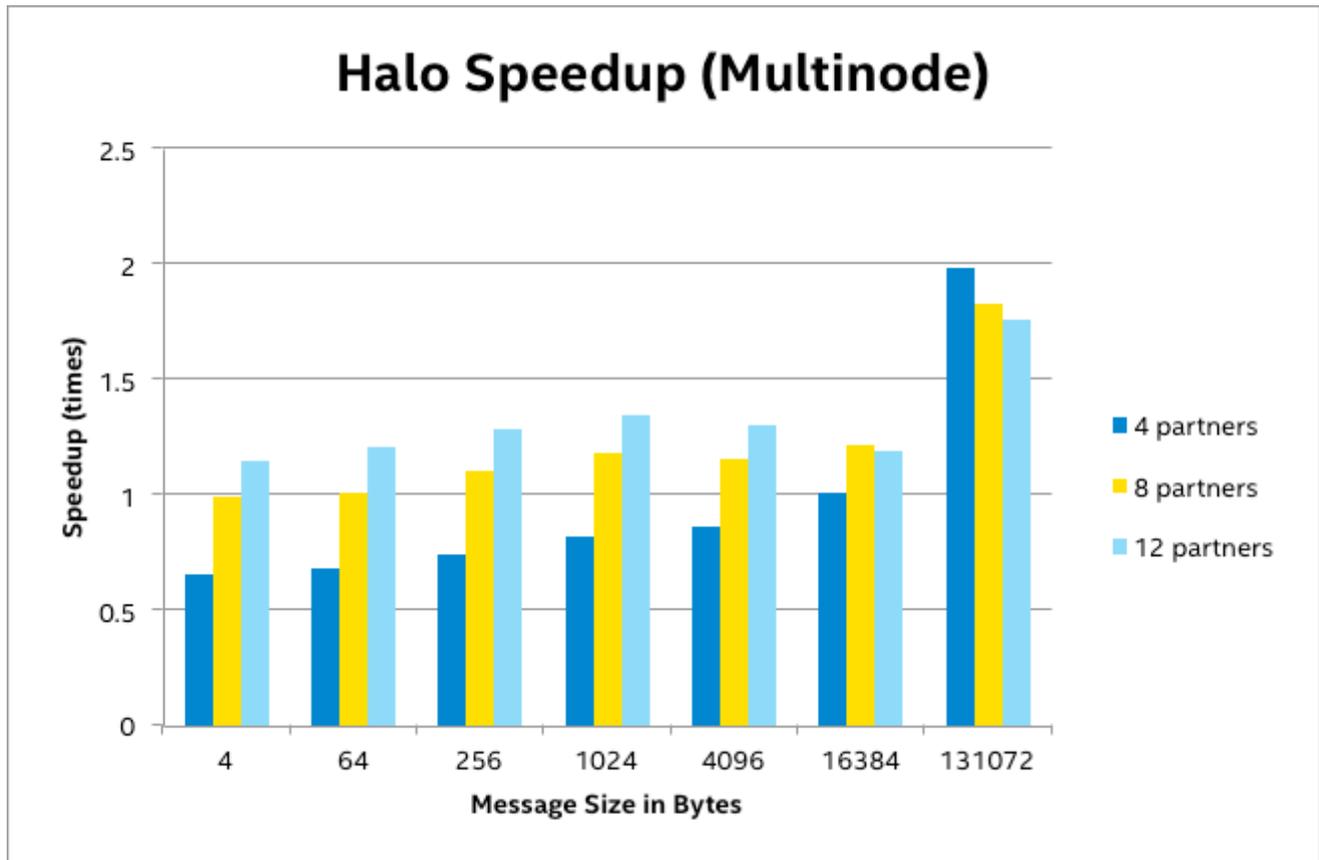
10 Speedup of MPI SHM approach compared to the point-to-point based ones
(measured on two coprocessors with 64 processes)

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.

Sign up for future issues | Share with a friend

Acknowledgments

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

Sign up for future issues | Share with a friend

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**.