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.