VSS Challenge Problem: Verifying the Correctness of AllReduce Algorithms in the MPICH Implementation of MPI

Paul D. Hovland
Mathematics and Computer Science Division
Argonne National Laboratory
Lemont, IL, USA
hovland@anl.gov

We describe a challenge problem for verification based on the MPICH implementation of MPI. The MPICH implementation includes several algorithms for allreduce, all of which should be functionally equivalent to reduce followed by broadcast. We created standalone versions of three algorithms and verified two of them using CIVL.

1 Introduction

The message passing interface (MPI) standard includes several forms of collective communication, including broadcast, allgather, reduce, and allreduce [2]. Implementations of MPI such as MPICH and OpenMPI often include multiple algorithms for each collective communication operation, with the particular algorithm selected at runtime based on hardware characteristics such as network topology and latencies and application characteristics such as process count and array lengths [3]. All of the algorithms for a given operation should be functionally equivalent; the only differences should be in performance and, perhaps, restrictions on the domain of applicability (e.g., only power-of-two process counts or only for builtin (not user-defined) datatypes). Consequently, the correctness of the implementations of the various algorithms can be confirmed by verifying their equivalence to one another and to the behavior specified in the MPI standard.

As a concrete example, we consider the implementation of `MPI_Allreduce` in the MPICH library. This function performs a global reduction operation such as product or sum and distributes the result to all processes. It can be viewed as equivalent to `MPI_Reduce` followed by `MPI_Broadcast`. MPICH version 4.3.0 implements 7 different algorithms for (blocking) allreduce (not including special cases for intercommunicator reductions and shared memory parallelism): recursive doubling, recursive exchange, recursive multiplying, reducescatter-allgather, k-way reducescatter-allgather, ring exchange, and tree exchange. In addition, MPICH supports calling one of the nonblocking algorithms and waiting on the result. Some of these algorithms are described in [3].

The MPICH implementations of allreduce algorithms typically rely on internal communication routines such as `MPIC_Send` and `MPIR_Reduce_local` as well as MPICH-specific data structures. To facilitate verification of individual algorithms, we created a header file that replaces MPICH's `mpiimpl.h` and redefines MPICH subroutines and macros using standard MPI functions. This enables one to, for example, compile the file `allreduce_intra_recursive_doubling.c` with no modifictions and without any additional dependencies and use `MPIR_Allreduce_intra_recursive_doubling` as a replacement for `MPI_Allreduce`. A limitation of this approach is that we assume that the reduction operator is a builtin reduction operator (maximum, sum, product, etc.) and the datatype is a builtin datatype (integer, float,

S.F. Siegel and G. Gopalakrishnan (Eds.):
International Workshop on Verification of
Scientific Software (VSS 2025)
EPTCS 432, 2025, pp. 90–97, doi:10.4204/EPTCS.432.10

double, etc.). It may be possible to remove the first restriction by making fewer assumptions about commutativity but the latter assumption will be difficult to overcome, in part because one would need to expose or replicate much of the complex mechanisms inside MPICH to deal with derived datatypes.

Currently, our challenge problem[1] consists of the following files:

`allreduce_intra_recursive_doubling.c`: verbatim copy of the MPICH v4.3.0 file, also available at `https://github.com/pmodels/mpich/blob/main/src/mpi/coll/allreduce/allreduce_intra_recursive_doubling.c` and included with light editing as Appendix A.

`allreduce_intra_recursive_multiplying.c`: slightly modified version of the file from MPICH v.4.3.0 (modifications include replacing direct access to fields within MPICH internal data structures with accessor macros and allocating memory for nonblocking communication requests).

`allreduce_intra_reduce_scatter_allgather.c`: slightly modified version of the file from MPICH v.4.3.0 (replace direct access to communicator internals and alloc/free temporary data structures).

`mpiimpl.h`: wrapper functions and macros to replace MPICH-specific functions and data structures (included as Appendix B).

`mpir_threadcomm.h`: empty file to satisfy one of the include dependencies.

`rd_allreduce_driver.c`: A driver that calls `MPIR_Allreduce_intra_recursive_doubling` and `MPI_Allreduce` and tests that the results are identical (included as Appendix C).

`rm_allreduce_driver.c`: A driver that calls `MPIR_Allreduce_intra_recursive_multiplying` and `MPI_Allreduce` and tests that the results are identical.

`rsag_allreduce_driver.c`: A driver that calls `MPIR_Allreduce_intra_reduce_scatter_allgather` and `MPI_Allreduce` and tests that the results are identical.

We have verified the correctness of the MPICH implementation of recursive doubling by using CIVL [1] to prove the equivalence of MPIR_Allreduce_intra_recursive_doubling and `MPI_Allreduce` for arbitrary inputs (up to a maximum array length of 10 for `MPI_SUM` and `MPI_PROD` and 5 for `MPI_MIN` and `MPI_MAX`) and arbitrary process counts (up to 10 and 5, respectively). In doing so, we leveraged the fact that CIVL has a builtin model of the semantics of `MPI_Allreduce`. We needed to provide CIVL with definitions of `MPI_Type_size` and `MPI_Reduce_local` since those functions are not included in the builtin model of MPI semantics. In addition, we needed to provide a definition of the `MPIR_Localcopy` that did not use `MPI_Sendrecv` (alternatively, we could have removed the `const` modifier from the send buffer parameter). Finally, we needed to modify the memory allocation for a temporary buffer to use the `datatype` parameter to cast the pointer to a pointer of appropriate type (CIVL requires that all memory allocations be cast to a non-void type). We were also able to verify the MPICH implementation of reduce_scatter-allgather following similar modifications. We do not expect to be able to use CIVL to verify the correctness of the MPICH implementation of recursive multiplying, as the latter relies on nonblocking sends and receives and CIVL cannot currently model these MPI operations.

Acknowledgements

This material is based upon work supported by the U.S. Department of Energy, Office of Science, Office of Advanced Scientific Computing Research, under the RAPIDS Institute within the SciDAC program, under contract DE-AC02-06CH11357. We thank Hui Zhou, Ken Raffenetti, and Mike Wilkins for their insights into the MPICH implementation of allreduce.

[1] Available at `https://github.com/pmodels/civl-verification`

References

[1] Stephen F. Siegel, Manchun Zheng, Ziqing Luo, Timothy K. Zirkel, Andre V. Marianiello, John G. Edenhofner, Matthew B. Dwyer & Michael S. Rogers (2015): *CIVL: the concurrency intermediate verification language*. In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '15, Association for Computing Machinery, New York, NY, USA, doi:10.1145/2807591.2807635.

[2] Marc Snir, Steve Otto, Steven Huss-Lederman, David Walker & Jack Dongarra (1998): *MPI-The Complete Reference, Volume 1: The MPI Core*, 2nd. (revised) edition. MIT Press, Cambridge, MA, USA.

[3] Rajeev Thakur, Rolf Rabenseifner & William Gropp (2005): *Optimization of collective communication operations in MPICH*. *The International Journal of High Performance Computing Applications* 19(1), pp. 49–66, doi:10.1177/1094342005051521.

A Example algorithm

The implementation in `MPIR_Allreduce_intra_recursive_doubling` includes a prologue and an epilogue to handle cases where the number of processes is not a power of 2. The prologue code begins at line 45. After ensuring that the number of processes participating in the reduction is a power of two, the recursive doubling algorithm begins at line 67. Finally, an epilogue at line 97 distributes the result to processes not participating in the reduction.

```
1  /* Copyright (C) by Argonne National Laboratory */
2  #include "mpiimpl.h"
3  #include "mpir_threadcomm.h"
4
5  int MPIR_Allreduce_intra_recursive_doubling(const void *sendbuf,
6          void *recvbuf, MPI_Aint count, MPI_Datatype datatype,
7          MPI_Op op, MPIR_Comm * comm_ptr, MPIR_Errflag_t errflag)
8  {
9      int comm_size, rank;
10     int mpi_errno = MPI_SUCCESS;
11     int mask, dst, is_commutative, pof2, newrank, rem, newdst;
12     MPI_Aint true_extent, true_lb, extent;
13     void *tmp_buf;
14
15     MPIR_THREADCOMM_RANK_SIZE(comm_ptr, rank, comm_size);
16
17     is_commutative = MPIR_Op_is_commutative(op);
18
19     /* need to allocate temporary buffer to store incoming data */
20     MPIR_Type_get_true_extent_impl(datatype, &true_lb, &true_extent);
21     MPIR_Datatype_get_extent_macro(datatype, extent);
22
23     MPIR_CHKLMEM_MALLOC(tmp_buf, count*(MPL_MAX(extent, true_extent)));
24
25     /* adjust for potential negative lower bound in datatype */
26     tmp_buf = (void *) ((char *) tmp_buf - true_lb);
27
28     /* copy local data into recvbuf */
29     if (sendbuf != MPI_IN_PLACE) {
30         mpi_errno = MPIR_Localcopy(sendbuf, count, datatype, recvbuf,
31             count, datatype);
32     }
33
34     /* get nearest power-of-two less than or equal to comm_size */
35     pof2 = MPL_pof2(comm_size);
36
37     rem = comm_size - pof2;
38
39     /* In the non-power-of-two case, all even-numbered
40      * processes of rank < 2*rem send their data to
41      * (rank+1). These even-numbered processes no longer
42      * participate in the algorithm until the very end. The
43      * remaining processes form a nice power-of-two. */
44
```

```
45      if (rank < 2 * rem) {
46          if (rank % 2 == 0) {    /* even */
47              mpi_errno = MPIC_Send(recvbuf, count, datatype, rank + 1,
48                  MPIR_ALLREDUCE_TAG, comm_ptr, errflag);
49
50              /* temporarily set the rank to -1 so that this process
51               * does not pariticipate in recursive doubling */
52              newrank = -1;
53          } else {        /* odd */
54              mpi_errno = MPIC_Recv(tmp_buf, count, datatype, rank - 1,
55                  MPIR_ALLREDUCE_TAG, comm_ptr, MPI_STATUS_IGNORE);
56
57              /* do the reduction on received data. since the
58               * ordering is right, it doesn't matter whether
59               * the operation is commutative or not. */
60              mpi_errno = MPIR_Reduce_local(tmp_buf, recvbuf, count,
61                  datatype, op);
62              /* change the rank */
63              newrank = rank / 2;
64          }
65      } else      /* rank >= 2*rem */
66          newrank = rank - rem;
67      if (newrank != -1) {
68          mask = 0x1;
69          while (mask < pof2) {
70              newdst = newrank ^ mask;
71              /* find real rank of dest */
72              dst = (newdst < rem) ? newdst * 2 + 1 : newdst + rem;
73
74              /* Send the most current data, which is in recvbuf. Recv
75               * into tmp_buf */
76              mpi_errno = MPIC_Sendrecv(recvbuf, count, datatype, dst,
77                 MPIR_ALLREDUCE_TAG, tmp_buf, count, datatype, dst,
78                 MPIR_ALLREDUCE_TAG, comm_ptr, MPI_STATUS_IGNORE, errflag);
79
80              /* tmp_buf contains data received in this step.
81               * recvbuf contains data accumulated so far */
82              if (is_commutative || (dst < rank)) {
83                  /* op is commutative OR the order is already right */
84                  mpi_errno = MPIR_Reduce_local(tmp_buf, recvbuf, count,
85                      datatype, op);
86              } else {
87                  /* op is noncommutative and the order is not right */
88                  mpi_errno = MPIR_Reduce_local(recvbuf, tmp_buf, count,
89                      datatype, op);
90                  /* copy result back into recvbuf */
91                  mpi_errno = MPIR_Localcopy(tmp_buf, count, datatype,
92                      recvbuf, count, datatype);
93              }
94              mask <<= 1;
95          }
96      }
```

```
97      /* In the non-power-of-two case, all odd-numbered
98       * processes of rank < 2*rem send the result to
99       * (rank-1), the ranks who didn't participate above. */
100     if (rank < 2 * rem) {
101         if (rank % 2)   /* odd */
102             mpi_errno = MPIC_Send(recvbuf, count, datatype, rank - 1,
103                 MPIR_ALLREDUCE_TAG, comm_ptr, errflag);
104         else    /* even */
105             mpi_errno = MPIC_Recv(recvbuf, count, datatype, rank + 1,
106                 MPIR_ALLREDUCE_TAG, comm_ptr, MPI_STATUS_IGNORE);
107     }
108   fn_exit:
109     return mpi_errno;
110 }
```

B Wrapper functions and macros

The following wrapper functions and macros map MPICH internal functions to standard MPI functions, enabling the MPICH allreduce implementations to be compiled as standalone functions.

```
1  #include "mpi.h"
2  #include <stdlib.h>
3  #include <stdio.h>
4
5  // Macros and functions to make standalone
6
7  #define MPIC_Sendrecv(A,B,C,D,E,F,G,H,I,J,K,L,M) \
8          MPI_Sendrecv(A,B,C,D,E,F,G,H,I,J,(*K),L)
9  #define MPIC_Send(A,B,C,D,E,F,G) MPI_Send(A,B,C,D,E,(*F))
10 #define MPIC_Recv(A,B,C,D,E,F,G) MPI_Recv(A,B,C,D,E,(*F),G)
11 #define MPIC_Isend(A,B,C,D,E,F,G,H) MPI_Isend(A,B,C,D,E,(*F),(*G))
12 #define MPIC_Irecv(A,B,C,D,E,F,G) MPI_Irecv(A,B,C,D,E,(*F),(*G))
13 #define MPIC_Waitall(A,B,C) MPI_Waitall(A,(*B),C)
14 #define MPIR_Request MPI_Request
15 #define MPIR_Reduce_local MPI_Reduce_local
16 #define MPIR_ERR_CHECK(X)
17 #define MPIR_THREADCOMM_RANK_SIZE(A,B,C) \
18         MPI_Comm_rank(*A, &B); MPI_Comm_size(*A, &C);
19 #define MPL_MIN(X, Y)  ((X) < (Y) ? (X) : (Y))
20 #define MPL_MAX(X, Y)  ((X) > (Y) ? (X) : (Y))
21 #define MPIR_CHKLMEM_MALLOC(A,B) A = malloc(B)
22 #define MPIR_ALLREDUCE_TAG            14
23 #define LOCALCOPY_TAG                 2153
24 #define MPIR_Comm MPI_Comm
25 #define MPIR_CHKLMEM_DECL(X)
26 #define MPIR_Op_is_commutative(X) 1
27 #define MPIR_CHKLMEM_FREEALL(X)
28 #define MPIR_Assert(X)
29 #define MPIR_Errflag_t MPI_Status*
30 #define MPIR_Datatype_get_extent_macro(A,B) \
31         MPIR_Datatype_get_extent(A,&B)
```

```
32 #define MPIR_Localcopy(sbuf, scount, sdatatype, rbuf, rcount, rdatatype)\
33     MPI_Sendrecv(sbuf, scount, sdatatype, 0, LOCALCOPY_TAG, rbuf, \
34     rcount, rdatatype, 0, LOCALCOPY_TAG, MPI_COMM_SELF, \
35     MPI_STATUS_IGNORE)
36 #define MAX_ALIGNMENT 16
37
38 void MPIR_Type_get_true_extent_impl(MPI_Datatype dtype, MPI_Aint *lbptr,
39                                     MPI_Aint *extentptr){
40    int myextent;
41
42    *lbptr = 0;
43    MPI_Type_size(dtype, &myextent);
44    *extentptr = myextent;
45 }
46
47 void MPIR_Datatype_get_extent(MPI_Datatype dtype, MPI_Aint *extentptr){
48    int myextent;
49
50    MPI_Type_size(dtype, &myextent);
51    *extentptr = myextent;
52 }
53
54 /* Returns int(log2(number)) */
55 static inline int MPL_log2(int number)
56 {
57     int p = 0;
58
59     while (number > 0) {
60         number >>= 1;
61         p++;
62     }
63     return p - 1;
64 }
65
66 static inline int MPL_pof2(int number)
67 {
68     if (number > 0) {
69         return 1 << MPL_log2(number);
70     } else {
71         return 0;
72     }
73 }
```

C Example driver

The driver for testing the correctness of `MPIR_Allreduce_intra_recursive_doubling` using concrete array values is as follows.

```
1 #include <mpi.h>
2 #include <stdio.h>
3 #include <string.h>
```

```
4  #define AR_OP MPI_SUM
5  #define N 10
6
7  int MPIR_Allreduce_intra_recursive_doubling(const void *, void *,
8          MPI_Aint, MPI_Datatype, MPI_Op, MPI_Comm *, MPI_Status *);
9
10 int main(int argc, char *argv[]) {
11     int correct = 1, rank, size;
12     double x[N], allreduce_result[N], my_allreduce_result[N];
13     MPI_Comm mycomm;
14
15     MPI_Init(&argc, &argv);
16     MPI_Comm_dup(MPI_COMM_WORLD, &mycomm);
17     MPI_Comm_rank(mycomm, &rank);
18     MPI_Comm_size(mycomm, &size);
19
20     for (int i = 0; i < N; i++)
21         x[i] = (double)rank + i*1.0;
22
23     MPI_Allreduce(x, allreduce_result, N, MPI_DOUBLE, AR_OP, mycomm);
24     MPIR_Allreduce_intra_recursive_doubling(x, my_allreduce_result,
25         N, MPI_DOUBLE, AR_OP, &mycomm, MPI_STATUS_IGNORE);
26
27     for (int i = 0; i < N; i++) {
28         if (allreduce_result[i] != my_allreduce_result[i]) {
29             correct = 0;
30             break;
31         }
32     }
33
34     if (correct)
35         printf("Rank %d: Results match!\n", rank);
36     else
37         printf("Rank %d: Results do NOT match!\n", rank);
38
39     MPI_Finalize();
40     return 0;
41 }
```

The CIVL driver is similar, replacing `#define N 10` on line 5 with

```
#ifndef NMAX
#define NMAX 10
#endif
$input int N;
$assume (1 <= N && N <= NMAX);
$input double x_val[N][16];
```

and the initialization of `x[i]` on line 21 with

```
        x[i] = x_val[i][rank];
```

to treat `x[i]` as symbolic values. Finally, an assertion is added before line 28:

```
        $assert(allreduce_result[i] == my_allreduce_result[i]);
```