What’s New In MPI

Jon Gibson
NAG

Technical Talk
ECMWF
2nd April, 2019

Experts in numerical software and
High Performance Computing

Talk Overview

- A Brief History of MPI
- Non-Blocking Collectives
- One-Sided Communication
- Fortran 2008 Bindings
- Other New Features
- Current Implementations and MPI-4.0

A Brief History of MPI

A New Hope

- January, 1993 – The MPI Forum first met
- May, 1994 – Version 1.0 of the standard published
- June, 1995 – Version 1.1 corrected and clarified the above
- July, 1997 – MPI-2 published, specifying **extensions** to MPI. Chapter 3 of this document contained corrections/clarifications to MPI-1.1, hence specifying MPI-1.2
- MPI standard development then stalled…

The MPI Strikes Back

- …to be picked up again a decade later.
- May 2008 – MPI-1.3 published, combining the MPI-1.1 document, the MPI-1.2 chapter of the MPI-2 document and additional errata
- June 2008 – MPI-2.1 published, combining the MPI-1.3 and MPI-2 documents and adding errata and clarifications.
- September 2009 – MPI-2.2 published, adding a few extensions but mostly corrections and clarifications of MPI-2.1.

Return of the MPI

- September 2012 – MPI-3.0 published, a major update of the standard
- June 2015 – MPI-3.1 published, adding some new functionality but mostly corrections and clarifications of MPI-3.0. This is the current standard.
- The MPI-4.0 standard is currently under development.

Changes made at MPI-3

- New functionality
  - Non-blocking collectives
  - New one-sided communication operations
  - Fortran 2008 bindings
  - Neighbourhood collectives
  - Tools interface
  - Plus several minor additions
- Some previously-deprecated functionality now removed
  - Including the C++ bindings

Non-Blocking Collectives

Non-blocking Collective Operations

- Non-blocking versions of all collective communication functions have been added.
- As with point-to-point,
  - the names are differentiated from the blocking calls by the addition of an **`I`** (for **`I`**mmediate return) after the **`MPI_`**
    - e.g. **`MPI_Ibcast`**, **`MPI_Ireduce`**, **`MPI_Iallgather`** and, of course, **`MPI_Ibarrier`**
  - the calls return an MPI_Request object, which is later used in a call to **`MPI_Wait`**/**`MPI_Test`** to complete the operation.

A Non-blocking Broadcast

```
INTEGER, DIMENSION(100) :: array1, array2
INTEGER :: root=0
INTEGER :: req, ierr
...
CALL MPI_IBCAST(array1, 100, MPI_INTEGER, &
root, MPI_COMM_WORLD, req, ierr)
! Computation that doesn't require array1
CALL compute(array2, 100)
CALL MPI_WAIT(req, MPI_STATUS_IGNORE, ierr)
```

A Note on the Matching of Collective Operations

- Non-blocking collective operations do not match blocking collective operations
  - unlike point-to-point

This Example is Wrong!

```
MPI_Request req;

switch(rank) {
     case 0:
          MPI_Ialltoall(sbuf, scnt, stype, rbuf,
          rcnt, rtype, comm, &req);

          MPI_Wait(&req, MPI_STATUS_IGNORE);
          break;
     case 1:
          MPI_Alltoall(sbuf, scnt, stype, rbuf,
          rcnt, rtype, comm);
          break;
}
```

Non-blocking Collective Operations

- Multiple non-blocking collectives may be outstanding on a single communicator.

```
MPI_Request reqs[3];

compute(buf1);
MPI_Ibcast(buf1, count, type, 0, comm, &reqs[0]);
compute(buf2);
MPI_Ibcast(buf2, count, type, 0, comm, &reqs[1]);
compute(buf3);
MPI_Ibcast(buf3, count, type, 0, comm, &reqs[2]);
MPI_Waitall(3, reqs, MPI_STATUSES_IGNORE);
```

Non-blocking Collective Operations

- All collective operations (blocking and non-blocking) in a given communicator must be called in the same order on all processes.

This Example is Wrong…

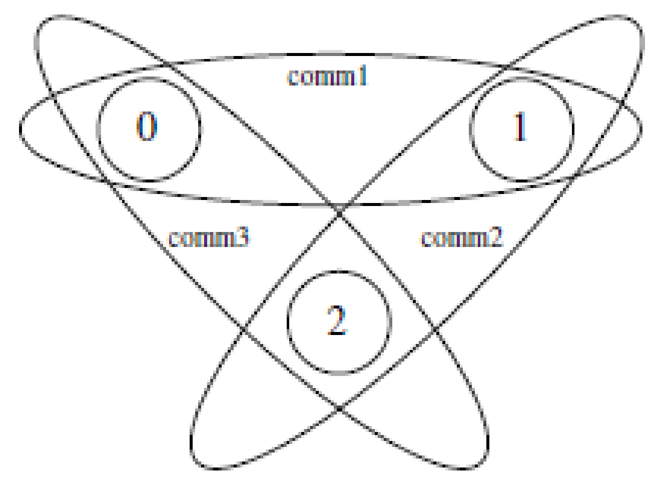
```
MPI_Request req;
switch(rank) {
     case 0:
          MPI_Ibarrier(comm, &req);
          MPI_Bcast(buf1, count, type, 0, comm);
          MPI_Wait(&req, MPI_STATUS_IGNORE);
          break;
     case 1:
          MPI_Bcast(buf1, count, type, 0, comm);
          MPI_Ibarrier(comm, &req);
          MPI_Wait(&req, MPI_STATUS_IGNORE);
          break;
}
```

…but could be re-written if we really wanted to do this.

```
MPI_Request req;
MPI_Comm dupcomm;
MPI_Comm_dup(comm, &dupcomm);
switch(rank) {
     case 0:
           MPI_Ibarrier(comm, &req);
           MPI_Bcast(buf1, count, type, 0, dupcomm);
           MPI_Wait(&req, MPI_STATUS_IGNORE);
           break;
     case 1:
           MPI_Bcast(buf1, count, type, 0, dupcomm);
           MPI_Ibarrier(comm, &req);
           MPI_Wait(&req, MPI_STATUS_IGNORE);
           break;
}
```

Overlapping Communicators

- Non-blocking collective operations can be used to enable simultaneous collective operations on multiple overlapping communicators.



Hold on a minute, did you say Non-blocking Barrier?

- Yes, that's right – a barrier without all that tedious waiting around! Let's check what it actually does and then look at a "real" example.

```
INTEGER :: req, ierr
...
CALL MPI_IBARRIER(MPI_COMM_WORLD, req, ierr)
...
CALL MPI_WAIT(req, MPI_STATUS_IGNORE, ierr)
```

A “Real” Example: Dynamic Sparse Data Exchange (DSDE)

- We have an N-body code with the physical domain distributed across different processes.
- Computation is divided into 2 phases:
  - 1) Calculation of forces and 2) movement of particles
- Particles may move from one process to another.
- Only the originating process knows which particles are leaving and where they are going.
- The destination processes typically won’t know how much they will receive, if anything, from the other processes.

Standard Ways of Implementing DSDE

- The “obvious” solution is to
  - First exchange the data sizes with **`MPI_Alltoall`**
  - And to use these to actually exchange the data in a call to **`MPI_Alltoallv`**
- This sends p^2 data items for a communicator of size p
  - Not ideal for sparse communication around the neighbourhood of each process.
- An alternative approach would be to use **`MPI_Reduce_scatter`** so that each process knows how many messages it has to receive, and then to receive them using **`MPI_Probe`** with **`MPI_ANY_SOURCE`**.
  - Still communicating p^2 items of metadata.

DSDE with Non-blocking Barrier

```
Each process sends each of its messages using
MPI_Issend

barrier_active = barrier_completed = 0

While (!barrier_completed)
  Check for incoming data using MPI_Iprobe with MPI_ANY_SOURCE
  If there is any incoming data, then receive it
  If (!barrier_active)
    Call MPI_Testall with the request handles from the MPI_Issend calls
    If all the MPI_Issend messages have completed,
        then call MPI_Ibarrier and set barrier_active = 1
  Else
    Call MPI_Test with the request handle returned by MPI_Ibarrier
    If completed, set barrier_completed = 1
```

One-Sided Communication

One-Sided Communication at MPI-2

- One-sided communication was introduced at MPI-2 with the aim of decoupling data transfer and process synchronisation.
- Each process would expose part of its memory, called a *window*, to other processes in the communicator via a call to **`MPI_Win_create`**.
- Routines were specified for controlling when windows can be accessed, *epochs*.
- Three routines were defined for transferring data
  - **`MPI_Put`**, **`MPI_Get`** and **`MPI_Accumulate`**

Additions at MPI-3

- New window creation routines
- New atomic read-modify-write operations
- New request-based RMA operations
- A new “unified memory model”

New Window Creation Routines

- **`MPI_Win_allocate`**
  - MPI allocates the memory associated with the window
  - Instead of the user passing allocated memory
- **`MPI_Win_create_dynamic`**
  - Creates a window without memory attached
  - The user can dynamically attach and detach memory to/from the window by calling **`MPI_Win_attach`** and **`MPI_Win_detach`**
- **`MPI_Win_allocate_shared`**
  - Creates a window of shared memory (within a node) that can be accessed by direct load/store accesses as well as RMA operations.

New Atomic Read-Modify-Write Operations

- **`MPI_Get_accumulate`**
  - The remote data is returned to the caller before the sent data is accumulated into the remote data.
- **`MPI_Fetch_and_op`**
  - Performs an **`MPI_Get_accumulate`** operation on single elements of data. This allows for a faster implementation.
- **`MPI_compare_and_swap`**
  - A single value at the origin is compared to a value at the target. The value at the target is replaced by a third value if the values at the origin and target are equal. The original value at the target is returned.

Request-based RMA Communication Operations

- **`MPI_Rput`**, **`MPI_Rget`**, **`MPI_Raccumulate`** and **`MPI_Rget_accumulate`**
- These routines return a request handle, which can later be used in one of the **`MPI_Test`**/**`MPI_Wait`** family of routines to test/wait for completion.
- Only valid within a passive target epoch.
  - i.e. only the origin process is explicitly involved in the transfer

The Unified Memory Model

- A new “unified memory model” had been added, in addition to the model assumed at MPI-2, now referred to as the “separate memory model”.
- The unified memory model assumes coherent memory (i.e. caches and explicit communication buffers) and the separate memory model does not.
  - Hence, the unified memory model allows the user to omit some synchronisation calls and potentially improve performance.
- The memory model of a window can be determined by accessing the attribute **`MPI_WIN_MODEL`**.

Fortran 2008 Bindings

Fortran 2008 Bindings

- An additional set of Fortran bindings
- Supports full and better quality argument checking with individual handles
- Support for choice arguments
  - Similar to `(void *)` in C
- Enables passing array subsections to non-blocking functions
- Optional *ierror* argument
- Fixes many issues with existing Fortran bindings

Fortran 2008 Bindings

- There are now three methods of Fortran support
  - **`USE mpi_f08`** – the only method consistent with the Fortran standard (Fortran 2008 + TS29113; or Fortran 2018)
  - **`USE mpi`** – the standard states “its use is not recommended”
  - **`INCLUDE ‘mpif.h’`** – its use is “strongly discouraged”

Err, remind me what TS29113 is?

- Technical Specification on “Further Interoperability of Fortran and C”
  - https://wg5-fortran.org/N1901-N1950/N1942.pdf
  - Now incorporated into the Fortran 2018 standard.
- The relevant additional language features are *assumed-type*, *assumed-rank* and an extension to the **`ASYNCHRONOUS`** attribute.
  - An assumed-type object is declared as **`TYPE(*)`**
  - An assumed-rank object is declared with **`DIMENSION(..)`**
  - **`ASYNCHRONOUS`** attribute extended to apply to variables used for asynchronous communication

How does that affect the bindings?

- In the original Fortran binding, we have

```
MPI_ISEND(BUF, COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR)
<type> BUF(*)
INTEGER COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR
```

- In the shiny new Fortran 2008 binding, we have

```
MPI_Isend(buf, count, datatype, dest, tag, comm, request, ierror)
TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: buf
INTEGER, INTENT(IN) :: count, dest, tag
TYPE(MPI_Datatype), INTENT(IN) :: datatype
TYPE(MPI_Comm), INTENT(IN) :: comm
TYPE(MPI_Request), INTENT(OUT) :: request
INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

Other New Features

Neighbourhood Collectives

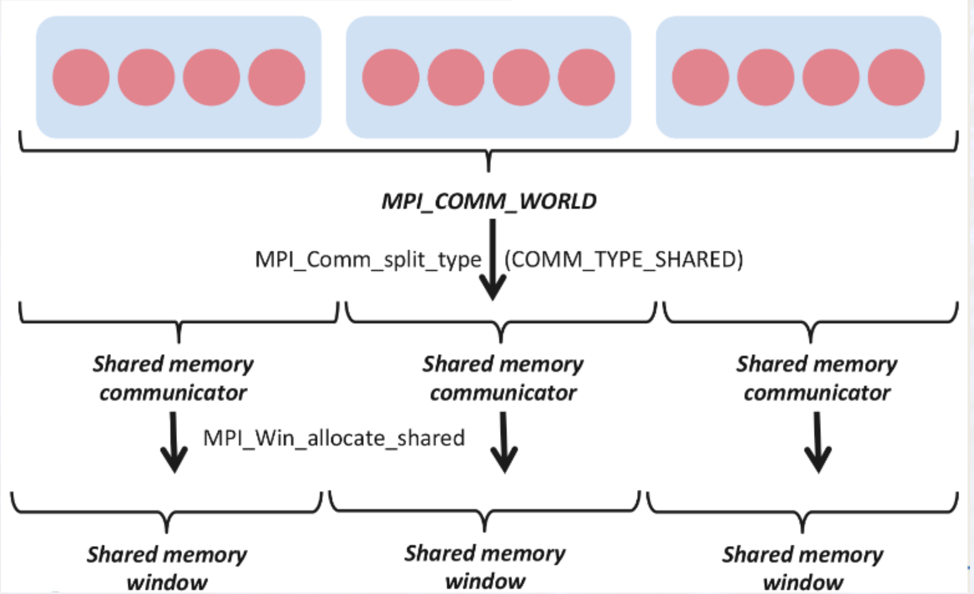
- Neighbourhood collectives perform collective communication between nearest neighbours in an MPI Cartesian, Graph or Distributed Graph topology.
  - Useful for stencil computations that require nearest-neighbour exchanges
- The new routines are:
  - **`MPI_Neighbor_allgather`** and **`MPI_Neighbor_allgatherv`**
  - **`MPI_Neighbor_alltoall`**, **`MPI_Neighbor_alltoallv`** and **`MPI_Neighbor_alltoallw`**
  - And the non-blocking counterparts to the above routines.

The MPI Tool Information Interface

- Beyond the PMPI profiling interface
- An extensive interface to allow tools (debuggers, performance analysers, etc) to extract information about MPI processes
- Note that each implementation defines its own performance and control variables; MPI does not define them.

`MPI_Comm_split_type`

- Splits the group associated with an existing communicator into subgroups of the same *split_type* and associates a new communicator with each.
- The *split_type* **`MPI_COMM_TYPE_SHARED`** is predefined.



Matching Probe and Recv

- `MPI_Probe` and `MPI_Iprobe` check for incoming messages without receiving them but since the list of incoming messages is global amongst the threads of an MPI process, these calls can be problematic in multithreaded environments.
- The new calls `MPI_Mprobe` and `MPI_Improbe`, used with the new matched receive calls, `MPI_Mrecv` and `MPI_Imrecv` avoid this problem.
  - The matched probe calls return a handle to the message, which can then be used in the matched receive call to actually receive that message.

“const” correct C bindings

- For example, the C binding for **`MPI_Send`** is now

```
int MPI_Send(const void* buf, int count,
MPI_Datatype datatype, int dest, int tag, MPI_Comm comm)
```

Other new features

- Non-collective communicator creation routine
- Non-blocking **`MPI_Comm_dup`**
- **`MPI_Type_create_hindexed_block`** routine

Current Implementations and MPI 4.0

Status of MPI-3.1 Implementations at June 2018

	MPICH	MVAPICH	Open MPI	Cray	Tianhe	Intel MPI	IBM			HPE	Fujitsu	MS	MPC	NEC	Sunway	RIKEN	AMPI
							BG/Q (legacy) [1]	PE (legacy) [2]	Spectrum								
NBC	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔
Nbr. Coll.	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✘	✔	✔	✔	✔	✔
RMA	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	(*)
Shr. mem	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	*
MPI_T	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	*	✔	✔	✔	✔	Q1 '19
Comm-create group	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	*	✔	✔	✔	✔	✔
F08 Bindings	✔	✔	✔	✔	✔	✔	✔	✘	✔	✔	*	✘	✔	✔	✔	✔	Q2 '19
New Dtypes	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔
Large Counts	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔
MProbe	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔
NBC I/O	✔	✔	✔	✔	✘	✔	✘	✘	✔	✔	*	✘	*	✔	✘	✔	Q2'19

[1] Open Source but unsupported [2] No MPI_T variables exposed * Under development (*) Partly done

Slide Updated 11/6/2018

Release dates are estimated and are subject to change at any time

✘ indicates no publicly announced plan to implement/support that feature

Thanks to Pavan Balaji

MPI-4.0

- Extensions to better support hybrid programming models
  - Each thread would have its own “rank”, which would make MPI messages from multiple threads faster.
- Support for fault tolerance in MPI applications
- Persistent collectives
- Performance assertions and hints

Want a say in the future of MPI?

- Please complete this short international survey of MPI users for the MPI Forum by 15th April, 2019:

https://docs.google.com/forms/d/e/1FAIpQLSd1bDppVODc8nB0BjlXdqSCO_MuEuNAAbBixl4onTchwSQFwg/viewform

Any questions?