



Modern MPI and what you need to know about it

Jeff Hammond
Principal Engineer
GPU Communication Software

Outline

1. MPI Application Binary Interface (ABI)
2. MPI Large-Count Support
3. MPI and Fortran
4. Using MPI RMA Effectively
5. MPI GPU Futures



ABI

Jeff Hammond, Lisandro Dalcin, Erik Schnetter, Marc PéRache, Jean-Baptiste Besnard, Jed Brown, Gonzalo Brito Gadeschi, Simon Byrne, Joseph Schuchart, and Hui Zhou. 2023. MPI Application Binary Interface Standardization. In *Proceedings of EuroMPI2023: the 30th European MPI Users' Group Meeting (EUROMPI '23)*, September 11-13, 2023, Bristol, United Kingdom. ACM, New York, NY, USA. <https://doi.org/10.1145/3615318.3615319>

API versus ABI

API

```
int MPI_Bcast(void * b, int n, MPI_Datatype d, int r, MPI_Comm c);
```

MPI_Datatype and MPI_Comm are unspecified types

ABI

```
typedef struct ompi_datatype_t * MPI_Datatype; // Open MPI family
```

```
typedef int MPI_Datatype; // MPICH family
```

Lots of other stuff like constants, SO names, SO versioning, calling convention, etc.

ABI Status Quo

MPI has been an **API** standard, which defines the source code behavior in C (C++) and Fortran. The **compiled** representation of MPI features is implementation-defined.

If you **compile** with one of the following MPI families, you **MUST run** with the same.

1. MPICH / Intel MPI / MVAPICH / Cray MPI
2. Open MPI / NVIDIA HPC-X / Amazon MPI / IBM Spectrum MPI

Family 1 exists because there was a demand for interoperability with Intel MPI due to the prevalence of usage in ISV codes.

Family 2 is not guaranteed to be consistent, especially across major versions.

Why standardize an MPI ABI?

Modern software use cases:

- Third-party **language** support, e.g. Python, Julia, Rust, etc.
- **Package** distribution, e.g. Apt, EasyBuild, Spack, etc.
- **Tools** become implementation-agnostic
- **Containers**
- More efficient **testing**

We can:

- Architectural reasons not to are gone
- Two platform ABIs cover >90% of HPC platforms

ABI Design

Integer Constants

Requirements:

- Position sequences: 0..n (MPI_SUCCESS..MPI_ERR_LASTCODE)
- XOR-able, i.e., 2^k (e.g. MPI_MODE_NOCHECK)
- Negative (MPI_ANY_SOURCE)
- Sizes (e.g. MPI_BSEND_OVERHEAD)
- Ordered subsets (e.g. MPI_THREAD_*)
- Arbitrary (e.g. MPI_ORDER_FORTRAN)

Except for error codes, array sizes and XOR-ables, all integer constants are unique. Error messages can easily tell user what they passed as it appears in the source code.

Handles

```
typedef struct MPI_ABI_Comm * MPI_Comm;  
typedef struct MPI_ABI_Request * MPI_Request;  
...
```

Satisfies existing requirements (= comparison, fits into a pointer because attributes).

Supports type-safety. Compilers know that MPI_Comm is not MPI_Group.

Downside: conversions to/from Fortran are not free like MPICH (at least with LP64).

The current 103 predefined handles are compile-time constants less than 1024 defined by a Huffman code. They can be translated to/from Fortran trivially.

Integer Types

Surprisingly, this was the hardest part...

- MPI_Aint is intptr_t because that satisfies all of the requirements
- MPI_Offset is int64_t because that will be sufficient for ~30 years
- MPI_Count is int64_t because $\max(\text{MPI_Aint}, \text{MPI_Offset})$
- *MPI_Fint is not part of the ABI*
 - f2c/c2f are replaced by fromint/toint

It is our intent specify an ABI for 32b and 64b systems since those are what we understand. 128b ABIs (e.g. CHERI) can be added in the future when appropriate.

Packaging

- The header is `mpi.h`
 - `#include <mpi.h>` still works - no code changes required to adopt ABI.
 - The Forum will distribute a standard header for convenience.
- The library is `libmpi_abi.ext` (or `mpi_abi.dll`)
 - Implementations are instructed to use platform-specific SO versioning conventions.
 - The Forum will distribute a standard SO for convenience.
- The ABI is versioned
 - Starts with 1.0
 - Backwards-compatible changes (e.g. new handle type or procedures) increment the minor version, which will happen for every new release of the standard.
 - Backwards-incompatible changes increment the major version.

Fortran ABI

Platform ABIs

The MPI ABI depends on the platform ABI, which is a function of:

1. The operating system and C compiler
2. The filesystem (offset size, but only weakly)
3. The Fortran compiler
 - a. **INTEGER and REAL** ← Visible to the MPI C ABI via handle constants and `MPI_Type_size`.
 - b. string passing
 - c. `CFI_cdesc_t` ← Constant for a given Fortran compiler usage (assuming no weird flags).
 - d. Module format

MPI Fortran ABI

- The MPI ABI for Fortran is incomplete, because we cannot specify it without specifying the implementation of MPI Fortran modules.
- **We defined an ABI that allows C to work with Fortran code, and for MPI Fortran to be implementable on top of the MPI C ABI.**
- Both VAPAA and MPICH are decoupling the Fortran and C implementations.

```
MPI_Abi_set_fortran_info, MPI_Abi_get_fortran_info, ...
```

The following keys are predefined for this object:

```
"mpi_integer_size": The size in bytes of the Fortran default INTEGER kind.
```

```
...
```

<https://github.com/jeffhammond/vapaa>

FAQ

- Final vote in June 2025, right before ISC.
- Launchers are not part of the ABI. There are at least two options:
 - Slurm and PBS launchers are supported by all the major MPIs already.
 - mpirun can set the shared library to use, in which case the launcher and library will match.
- Wrapper scripts (e.g. mpicc) are not standard but the ecosystem will probably have mpicc_abi or mpicc -abi.
- MPICH and Open MPI will continue to support their existing ABIs (for now).



Large-count support

J. R. Hammond, A. Schafer and R. Latham, "To INT_MAX... and Beyond!
Exploring Large-Count Support in MPI," in 2014 Workshop on Exascale
MPI at Supercomputing Conference (ExaMPI), New Orleans, LA, USA,
2014. <https://doi.ieeecomputersociety.org/10.1109/ExaMPI.2014.5>

Support for large-counts

C API

```
int MPI_Bcast(void *buffer, int count, MPI_Datatype  
datatype, int root, MPI_Comm comm)
```

```
int MPI_Bcast_c(void *buffer, MPI_Count count, MPI_Datatype  
datatype, int root, MPI_Comm comm)
```

With one exception, all MPI functions that take a count argument have two APIs, one for each count integer type.

Support for large-counts

Fortran API in mpi_f08

```
MPI_Bcast(buffer, count, datatype, root, comm, ierror)  
  TYPE(*), DIMENSION(..) :: buffer  
  INTEGER, INTENT(IN) :: count, root  
  TYPE(MPI_Datatype), INTENT(IN) :: datatype  
  TYPE(MPI_Comm), INTENT(IN) :: comm  
  INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

```
MPI_Bcast(buffer, count, datatype, root, comm, ierror) !(_c)  
  TYPE(*), DIMENSION(..) :: buffer  
  INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count  
  TYPE(MPI_Datatype), INTENT(IN) :: datatype  
  INTEGER, INTENT(IN) :: root  
  TYPE(MPI_Comm), INTENT(IN) :: comm  
  INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

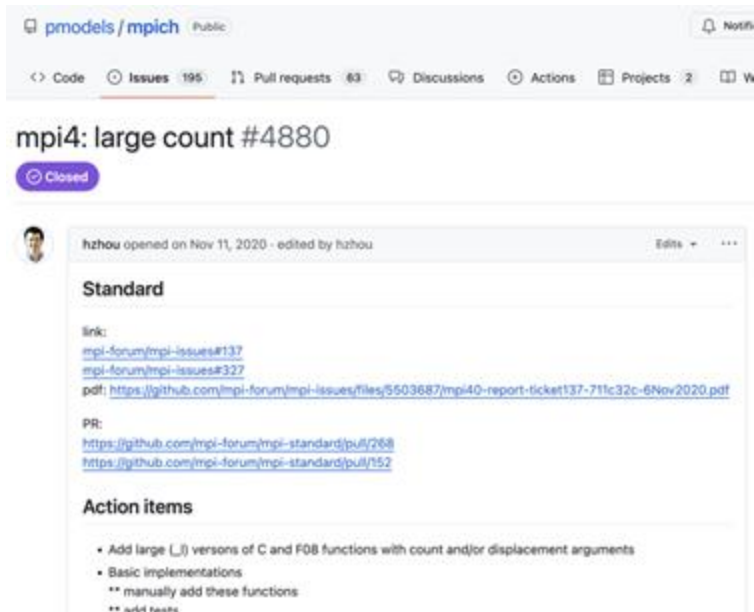
Support for large-counts

Fortran API in mpi module and mpif.h

```
MPI_BCAST(BUFFER, COUNT, DATATYPE, ROOT, COMM, IERROR)  
  <type> BUFFER(*)  
  INTEGER COUNT, DATATYPE, ROOT, COMM, IERROR
```

NO LARGE-COUNT SUPPORT IN PRE-MODERN FORTRAN SUPPORT

Implementation Status



pmodels / mpich Public

<> Code Issues 195 Pull requests 83 Discussions Actions Projects 2

mpi4: large count #4880

Closed

hzhou opened on Nov 11, 2020 · edited by hzhou

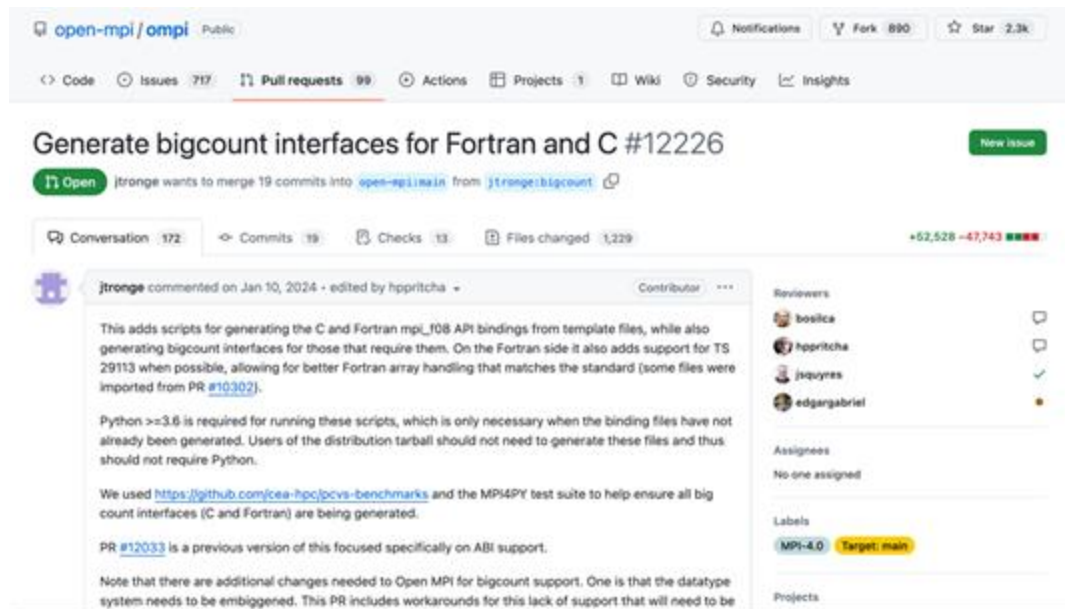
Standard

link:
<https://github.com/mpich/mpich/issues/137>
<https://github.com/mpich/mpich/issues/327>
pdf: <https://github.com/mpich/mpich/files/5503687/mpi4-report-ticket137-711c32c-6Nov2020.pdf>

PR:
<https://github.com/mpich/mpich/pull/268>
<https://github.com/mpich/mpich/pull/152>

Action items

- Add large (L) versions of C and Fortran functions with count and/or displacement arguments
- Basic implementations
 - ** manually add these functions
 - ** add tests



open-mpi / ompi Public

Notifications Fork 890 Star 2.3k

<> Code Issues 717 Pull requests 99 Actions Projects 3 Wiki Security Insights

Generate bigcount interfaces for Fortran and C #12226

Open jtronge wants to merge 19 commits into open-mpi:main from jtronge:bigcount

Conversation 172 Commits 18 Checks 13 Files changed 1,229 +62,526 -47,743

jtronge commented on Jan 10, 2024 · edited by hpritcha

This adds scripts for generating the C and Fortran mpi_f08 API bindings from template files, while also generating bigcount interfaces for those that require them. On the Fortran side it also adds support for TS 29113 when possible, allowing for better Fortran array handling that matches the standard (some files were imported from PR #10302).

Python >=3.6 is required for running these scripts, which is only necessary when the binding files have not already been generated. Users of the distribution tarball should not need to generate these files and thus should not require Python.

We used <https://github.com/cea-hpc/pcvs-benchmarks> and the MPI4PY test suite to help ensure all big count interfaces (C and Fortran) are being generated.

PR #12033 is a previous version of this focused specifically on ABI support.

Note that there are additional changes needed to Open MPI for bigcount support. One is that the datatype system needs to be embiggened. This PR includes workarounds for this lack of support that will need to be

Reviewers

- bosilca
- hpritcha
- jquyres
- edgargabriel

Assignees

No one assigned

Labels

- MPI-4.0 Target: main

Projects

<https://github.com/pmodels/mpich/issues/4880>
<https://github.com/open-mpi/ompi/pull/12226>



Fortran

“MPI and Modern Fortran: Better Together”

<https://pasc24.pasc-conference.org/presentation/?id=msa277&sess=sess129>

<https://drive.google.com/file/d/1--poinTx7bETmU-tnUu--Wj3eGHUFJKp/view>

MPI Language Support

1. MPI C API - used by all languages except Fortran
- ~~2. MPI C++ API deleted in MPI 3.0 (2012)~~
3. MPI **mpif.h** (falsely known as “F77” bindings)
4. MPI **mpi** module (falsely known as “F90” bindings)
5. MPI **mpi_f08** module (the good stuff)

MPI Language Support

1. MPI C API - used by all languages except Fortran
- ~~2. MPI C++ API deleted in MPI 3.0 (2012)~~
- ~~3. MPI `mpif.h` deprecated in MPI 4.1 (2023)~~
4. MPI **mpi** module (falsely known as “F90” bindings)
5. MPI **mpi_f08** module (the good stuff)

MPI Fortran legacy API

```
MPI_BCAST (BUFFER, COUNT, DATATYPE, ROOT, COMM, IERROR)
```

```
!$PRAGMA IGNORE_TKR
```

```
<type> :: BUFFER (*)
```

```
INTEGER :: COUNT, DATATYPE, ROOT, COMM, IERROR
```

Until Fortran 2008, there is no standard mechanism for type-agnostic buffers equivalent to C void*. Implementations rely on compiler-specific extensions (e.g., as shown above) or lack of enforcement of type safety to compile. There is also no way to use Fortran array properties, including subarrays.

This matters more now, because GCC warnings are increasingly hostile to type abuse.

MPI Fortran legacy API

```
MPI_BCAST(BUFFER, COUNT, DATATYPE, ROOT, COMM, IERROR)
```

```
!$PRAGMA IGNORE_TKR
```

```
<type> :: BUFFER(*)
```

```
INTEGER :: COUNT, DATATYPE, ROOT, COMM, IERROR
```

Datatypes and communicators are MPI object handles, not generic integers. Without proper types, compilers cannot identify user errors, so they manifest in unpleasant ways at runtime.

MPI Fortran modern API

```
MPI_Bcast(buffer, count, datatype, root, comm, ierror)
  TYPE(*), DIMENSION(..) :: buffer
  INTEGER, INTENT(IN) :: count, root
  TYPE(MPI_Datatype), INTENT(IN) :: datatype
  TYPE(MPI_Comm), INTENT(IN) :: comm
  INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

Buffers are assumed-type, assumed-rank arguments. MPI implementations can - but are not required to - support non-contiguous subarrays.

MPI Fortran modern API

```
MPI_Bcast(buffer, count, datatype, root, comm, ierror)
  TYPE(*), DIMENSION(..) :: buffer
  INTEGER, INTENT(IN) :: count, root
  TYPE(MPI_Datatype), INTENT(IN) :: datatype
  TYPE(MPI_Comm), INTENT(IN) :: comm
  INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

MPI object handles are properly typed and thus compilers will not accept erroneous usage. At the same time, MPI object handle types are interoperable with the old method, because the type contains the integer handle as its only member.

MPI Fortran modern API

```
MPI_Bcast(buffer, count, datatype, root, comm, ierror)
  TYPE(*), DIMENSION(..) :: buffer
  INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count
  TYPE(MPI_Datatype), INTENT(IN) :: datatype
  INTEGER, INTENT(IN) :: root
  TYPE(MPI_Comm), INTENT(IN) :: comm
  INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

All MPI procedures that take a count argument use polymorphic interfaces to support both INTEGER (usually 32b) and large-count (i.e. 64b) variants. This aspect of the MPI Fortran API is superior to both C and C++.

MPI Fortran modern API

```
MPI_Irecv(buf, count, datatype, source, tag, comm,  
          request, ierror)
```

```
TYPE(*), DIMENSION(..), ASYNCHRONOUS :: buf
```

```
INTEGER, INTENT(IN) :: count, source, tag
```

```
TYPE(MPI_Datatype), INTENT(IN) :: datatype
```

```
TYPE(MPI_Comm), INTENT(IN) :: comm
```

```
TYPE(MPI_Request), INTENT(OUT) :: request
```

```
INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

Non-blocking procedures require ASYNCHRONOUS buffer attribute to prohibit (unlikely) compiler optimizations and (likely) temporary copies that break correctness.

Asynchronous Procedures

`MPI_SUBARRAYS_SUPPORTED = .FALSE.`

`MPI_ASYNC_PROTECTS_NONBLOCKING = .FALSE.`

```
integer :: buf(1000)
```

```
call MPI_Irecv(buf(1:1000:2), .., req)
```

```
! integer :: temp(500) = buf(1:1000:2)
```

```
! address of temp passed to network API
```

```
! temp is deallocated when MPI_Irecv returns
```

```
call MPI_Wait(req)
```

```
! network writes to temp, which no longer exists
```

```
! segmentation fault
```

<= This is uncommon usage.

Asynchronous Procedures

`MPI_SUBARRAYS_SUPPORTED = .TRUE.`

`MPI_ASYNC_PROTECTS_NONBLOCKING = .TRUE.`

```
void CFI_MPI_Irecv(CFI_cdesc_t * desc, int count, int datatype_f, int
source, int tag, int comm_f, int * request_f, int * ierror)
{
    ! translate datatype and communication handles from Fortran to C
    ! create MPI datatype from desc, count and datatype: temp_type
    *ierror = PMPI_Irecv(desc->base_addr, 1, temp_type, source, tag,
comm, &request);
    ! translate request handle from C to Fortran
}
```

Implementation Status

Today:

- Fortran compiler should support Fortran 2018 CFI features, i.e. `CFI_cdesc_t`.
 - NVHPC Fortran was the last major compiler to support this.
- MPI implementation should support assumed-rank, assumed-type arguments in the interface and `CFI_cdesc_t` buffer arguments in the implementation.
 - MPICH has this. Open MPI doesn't. Both implementations are valid.
- The MPI library is compiled once for every Fortran compiler.
 - Regular users don't care but sysadmins hate this.

Tomorrow:

- MPI Fortran features depend only on the C ABI and may be built separately.
 - VAPAA is a prototype of this. MPICH may also get there.



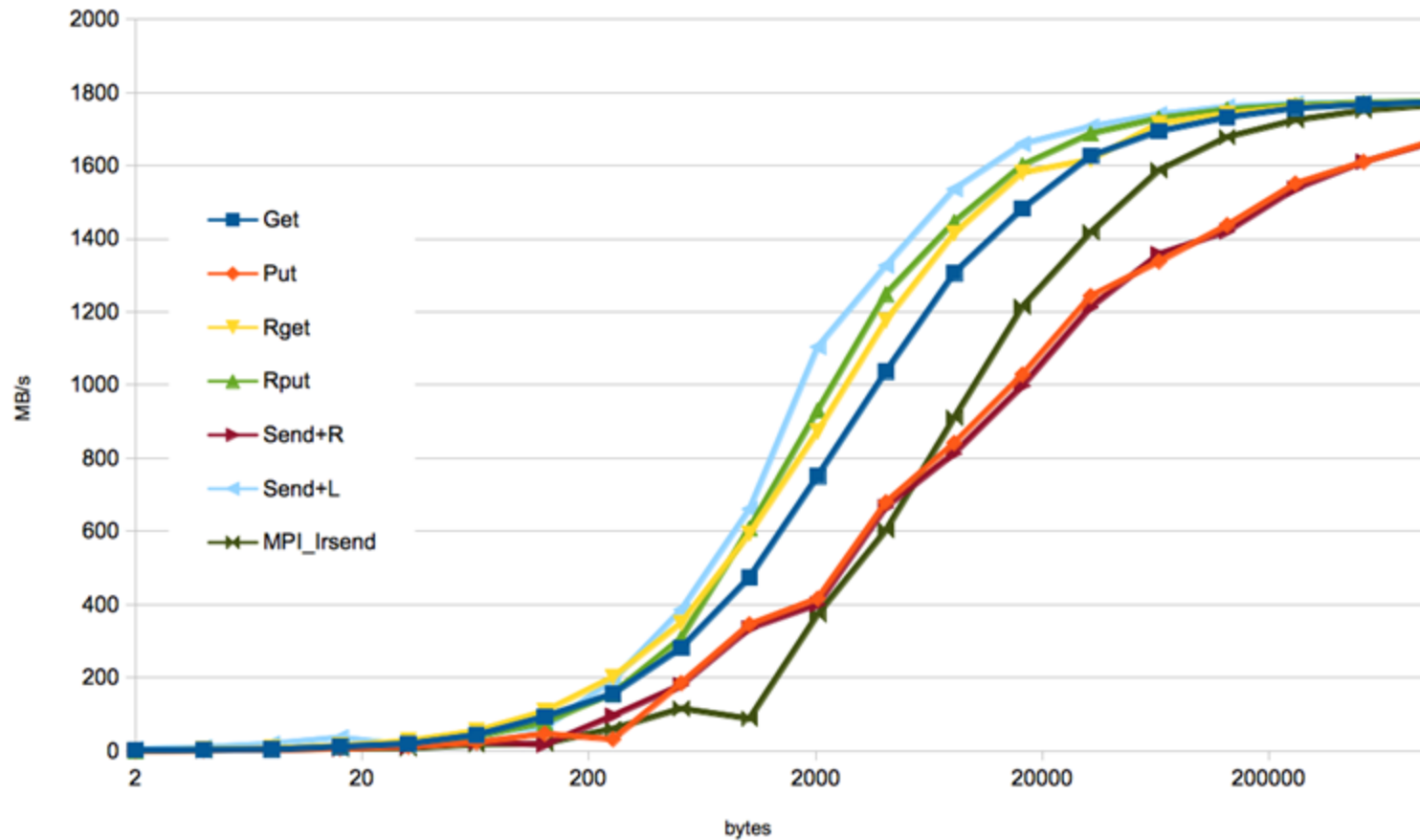
RMA (aka one-sided)

Why one-sided communication?

- Separate data movement from synchronization. Theoretically more scalable.
 - Send-Recv matches in order and often requires a handshake.
 - RMA allows control over ordering.
- Unstructured communication. Random access reads and/or updates.
 - NWChem distributed arrays are accessed quasi-randomly.
 - Sorting can be written as scattering across the system based on local info.
- Better match for modern HPC hardware.
 - RDMA read and write = MPI Get and Put (in theory)
 - RDMA atomics = MPI atomics (in theory)
 - Registered memory = MPI window allocation

PAMI vs. MPI

internode communication



MPI RMA Ecosystem Examples



Intel Fortran
GCC Fortran (OpenCoarrays)



NWCHEM

HIGH-PERFORMANCE COMPUTATIONAL
CHEMISTRY SOFTWARE

Global Arrays → ARMCI-MPI



OSHMPI

The only standardized PGAS language, the original one-sided library, and the most widely used application based on one-sided are all demonstrated to use MPI-3 RMA.

Overview of RMA

Window Types

1. Created collectively from user buffer
2. Created dynamically from user buffers
3. Allocated by MPI
4. Allocated by MPI from shared-memory

One-sided operations

1. Put/Get: non-atomic RMA
2. Accumulate: atomic RMA
 - a. Acc/Get_acc w/ math
 - b. w/ NOOP & REPLACE
 - c. F&Op, CAS for scalars

Synchronization Motifs

1. Fence (true BSP)
2. Post-Start-Complete-Wait (PSCW)
3. Passive-target (Lock/Unlock)
 - a. Exclusive lock
 - b. Shared lock
 - c. Flush local and remote
 - d. Request-based
4. Pure shared-memory

Overview of RMA

Window Types

1. Created collectively from user buffer
2. Created dynamically from user buffers
3. Allocated by MPI
4. Allocated by MPI from shared-memory

One-sided operations

1. Put/Get: non-atomic RMA
2. Accumulate: atomic RMA
 - a. Acc/Get_acc w/ math
 - b. w/ NOOP & REPLACE
 - c. F&Op, CAS for scalars

Synchronization Motifs

1. Fence (true BSP)
2. Post-Start-Complete-Wait (PSCW)
3. Passive-target (Lock/Unlock)
 - a. Exclusive lock
 - b. Shared lock
 - c. Flush local and remote
 - d. Request-based
4. Pure shared-memory

The key to using RMA is to pick a small subset of things you actually need and ignore the rest!

Choosing a Window Type

Do I need to be able to attach multiple user buffers asynchronously?
You must use **MPI_Win_create_dynamic** +
MPI_Win_attach.

Do I need to be able to attach a user buffer?
You must use **MPI_Win_create**.

Do I need to know for sure that I can use shared-memory on this window?
You must use **MPI_Win_allocate_shared**.

For all other purposes, use **MPI_Win_allocate**.

Performance Improves

Choosing a Synchronization Motif

Am I weirdly opposed to using Send-Recv and want to gamble on performance?
Fine, use Post-Start-Complete-Wait (**PSCW**).

Am I weirdly opposed to using collectives like MPI_Alltoallv?
Fine, use **Fence** synchronization.

Am I using MPI as an easy alternative to POSIX shm_open and **nothing** else?
Okay, use language/processor-specific **atomic operations**
and sync.

Use **passive-target** synchronization...

Choosing a Synchronization Motif, Part 2

Do I want MPI to provide transaction semantics for a group of RMA operations?
MPI_Win_lock(MPI_LOCK_EXCLUSIVE,..) provides this.
(Why?)

Is your communication limited to a small set of processes?
MPI_Win_lock(MPI_LOCK_SHARED,..) *may* be more efficient.

Use **MPI_Win_lock_all**, which is ~free with MPI_MODE_NOCHECK.

Summary: Windows and Synchronization

1. Use `MPI_Win_allocate` whenever possible.
2. Just use passive-target synchronization with `MPI_Win_lock_all`.

```
void Allocate<T>(MPI_Comm comm, MPI_Aint n, T* *ptr, MPI_Win *win)
{
    MPI_Win_allocate(n, sizeof(T), MPI_INFO_NULL, comm, ptr, win);
    MPI_Win_lock_all(MPI_MODE_NOCHECK, *win);
}
```

RMA Communication

```
void Blocking_Put(const void *buf, int count, MPI_Datatype type,
                 int target, MPI_Aint disp, MPI_Win win)
{
    MPI_Put(buf, count, type, target, disp, count, type, win);
    MPI_Win_flush_local(target, win);
}

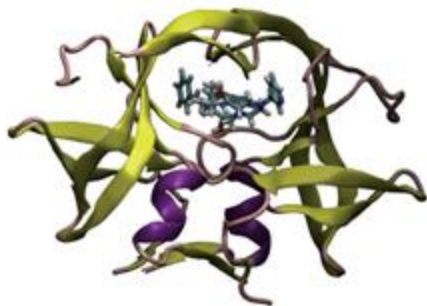
void Global_Sync(MPI_Win win, MPI_Comm comm)
{
    MPI_Win_flush_all(MPI_Win win);
    MPI_Barrier(comm);
}
```



RMA Results

NWChem Evaluation

- 1hsg_28 benchmark system
- 122 atoms, 1159 basis functions
- H,C,N,O w/ cc-pVDZ basis set
- Semidirect algorithm
- Closed shell (RHF)



E. Chow, X. Liu, S. Misra, M. Dukhan, M. Smelyanskiy, J. R. Hammond, Y. Du, X.-K. Liao and P. Dubey. *International Journal of High Performance Computing Applications*. "Scaling up Hartree-Fock Calculations on Tianhe-2." <http://dx.doi.org/10.1177/1094342015592960>
(GTFock used GA/ARMCI-MPI and MPICH-Glex for these petascale runs.)

The primary motivation for ARMCI-MPI is that it does not crash and requires no effort to port to a modern HPC system, but it often performs well, too.

NWChem SCF performance (old)

NWChem 6.3/ARMCI-MPI3/Casper

NWChem 6.5/ARMCI-DMAPP
(built by NERSC, Nov. 2014)

| iter | energy | time | iter | energy | time |
|------|------------------|--------------|------|------------------|--------------|
| 1 | -2830.4366669992 | 69.6 | 1 | -2830.4366670018 | 67.6 |
| 2 | -2831.3734512508 | 78.8 | 2 | -2831.3734512526 | 85.5 |
| 3 | -2831.5712563433 | 86.9 | 3 | -2831.5713109544 | 105.4 |
| 4 | -2831.5727802438 | 96.1 | 4 | -2831.5727856636 | 126.6 |
| 5 | -2831.5727956882 | 110.0 | 5 | -2831.5727956992 | 161.7 |
| 6 | -2831.5727956978 | 127.8 | 6 | -2831.5727956998 | 190.9 |

Running on 8 nodes with 24 ppn. Casper uses 2 ppn for comm.

NWChem SCF performance (new)

NWChem 6.3/ARMCI-MPI3/Casper

NWChem Dev/ARMCI-MPIPR
(built by NERSC, Sept. 2015)

| iter | energy | time | iter | energy | time |
|------|------------------|--------------|------|------------------|--------------|
| 1 | -2830.4366669990 | 69.3 | 1 | -2830.4366669999 | 61.4 |
| 2 | -2831.3734512499 | 77.1 | 2 | -2831.3734512509 | 69.3 |
| 3 | -2831.5712604368 | 84.6 | 3 | -2831.5713109521 | 77.8 |
| 4 | -2831.5727804428 | 93.0 | 4 | -2831.5727856618 | 87.3 |
| 5 | -2831.5727956927 | 107.3 | 5 | -2831.5727956974 | 103.9 |
| 6 | -2831.5727956977 | 128.0 | 6 | -2831.5727956980 | 125.7 |

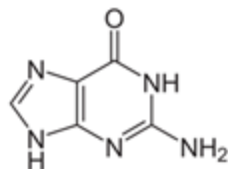
Running on 8 nodes with 24 ppn. **Both** use 2 ppn for comm.

Cluster A

- Dual Socket Intel Platinum 8280 CPU @2.70GHz
- NVIDIA ConnectX-6 HDR100 InfiniBand adapter
- NVIDIA Quantum HDR InfiniBand Switch QM7800
- Memory: 192GB DDR4 2666MHz RDIMMs per nox
- Lustre Storage, NFS

Software

- OS: Rocky Linux 8.6 , MLNX_OFED 5.6.1
- MPI: OpenMPI 4.1, OpenMPI 5
- Global Arrays configured with ARMCI-MPI, which uses MPI-3 RMA.
- NWChem 7.0.1
- Input: nsf_rccsd_cc-pvdz_energy.nw



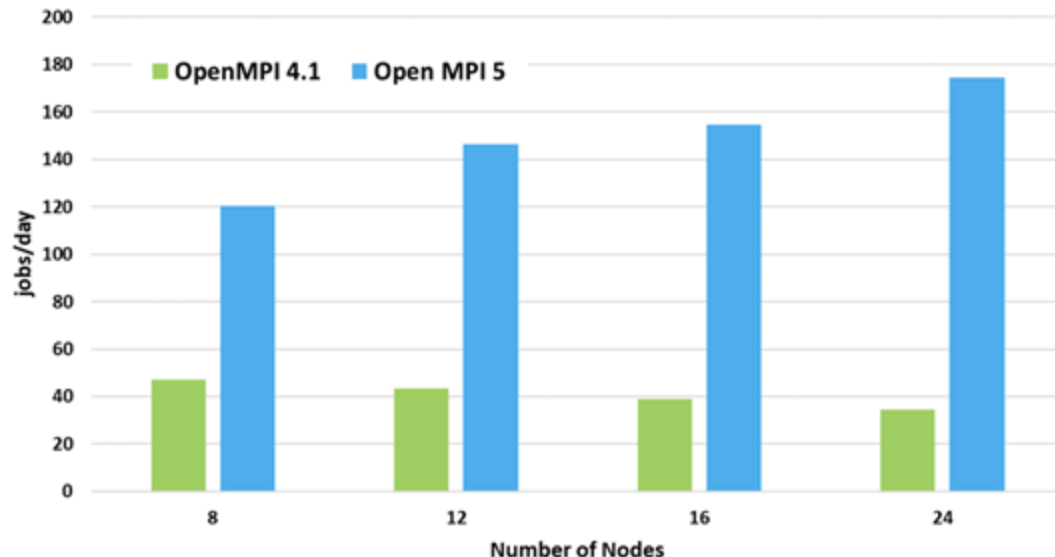
(1 guanine)

Benchmark details

- Based on the NSF Blue Waters SPP benchmark: <https://bluewaters.ncsa.illinois.edu/spp-benchmarks>
- Evaluates the communication-intensive CCSD phase of the benchmark, which only ran on 1 processor per CPU on Blue Waters due to network saturation.

NWChem

(nsf_rccsd_cc-pvdz_energy)



| Parameter | OpenMPI 4.1 | OpenMPI 5.0 | Percentage Change |
|------------------------------|-------------|-------------|-------------------|
| RMW max (Tail latency) | 7.4s | 0.011s | 654X |
| RMW max average | 0.327580s | 0.000395s | 829X |
| Barrier max (Tail latency) | 44s | 16s | 2.7X |
| Barrier average | 0.05s | 0.012 | 4.2X |
| Wait time max (Tail latency) | 11.6s | 0.025s | 8.6X |
| Fence average | 5usec | 3usec | 1.66X |
| NbGet max (Tail latency) | 11.9s | 0.8s | 13.7X |
| NbGet average | 0.0046s | 0.00055s | 8.5X |

Summary of MPI RMA in Practice

- MPI RMA is complicated and hard to understand.
Everyone involved feels bad about this.
 - All the easy fixes break backwards-compatibility.
- Abstraction layers like Fortran coarrays, OpenSHMEM and ARMCI-MPI hide the complexity of MPI RMA semantics.
- MPI RMA is properly implemented on essentially all platforms.
 - Open MPI 5, MPICH, Intel MPI and MVAPICH all work well.
- Asynchronous progress is critical for some applications.
 - Casper (from Argonne) is an implementation-agnostic solution.
 - Open MPI 5 on IB is really good.

Reading List

- Dinan, J., Balaji, P., Buntinas, D., Goodell, D., Gropp, W., and Thakur, R. (2016) *An implementation and evaluation of the MPI 3.0 one-sided communication interface*. Concurrency Computat.: Pract. Exper., 28: 4385–4404. <https://doi.org/10.1002/cpe.3758>
- Robert Gerstenberger, Maciej Besta, and Torsten Hoefler. 2013. *Enabling highly-scalable remote memory access programming with MPI-3 one sided*. In Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (SC '13). Association for Computing Machinery, New York, NY, USA, Article 53, 1–12. <https://doi.org/10.1145/2503210.2503286>
- Hammond, J.R., Ghosh, S., Chapman, B.M. (2014). *Implementing OpenSHMEM Using MPI-3 One-Sided Communication*. In: Poole, S., Hernandez, O., Shamis, P. (eds) OpenSHMEM and Related Technologies. Experiences, Implementations, and Tools. OpenSHMEM 2014. Lecture Notes in Computer Science, vol 8356. Springer, Cham. https://doi.org/10.1007/978-3-319-05215-1_4
- Torsten Hoefler, James Dinan, Rajeev Thakur, Brian Barrett, Pavan Balaji, William Gropp, and Keith Underwood. 2015. *Remote Memory Access Programming in MPI-3*. ACM Trans. Parallel Comput. 2, 2, Article 9 (July 2015), 26 pages. <https://doi.org/10.1145/2780584>
- Tutorial: https://hpc.inf.ethz.ch/publications/img/MPI_RMA_and_advanced_MPI.pdf
- <https://github.com/pmodels/armci-mpi> and <https://github.com/jeffhammond/oshmpi> demonstrate all of the aforementioned techniques in detail.



GPUs

Why MPI on GPUs is Hard

- Is this a CUDA GPU or an OpenCL GPU?
 - Forward-progress guarantees are important.
 - Blocking, synchronization and ordering are all performance hazards on GPUs. RMA is a good model for GPUs...
 - MPI support for NUMA doesn't easily extend to GPUs.
- NCCL is MPI for GPUs
 - Stream semantics in everything.
 - Only implements patterns that make sense.
 - Supports GPU endpoints...
- Hopefully, MPI-6 will catch up to NCCL...