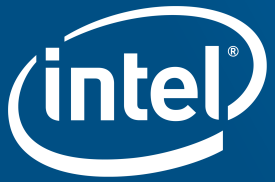


MPI: AS EASY AS 2, 3, 4

WHY MOST OF WHAT YOU THINK YOU KNOW ABOUT MPI IS WRONG

Jeff Hammond
Parallel Computing Lab
Intel Corporation

Legal Disclaimer & Optimization Notice

INFORMATION IN THIS DOCUMENT IS PROVIDED “AS IS”. NO LICENSE, EXPRESS OR IMPLIED, BY ESTOPPEL OR OTHERWISE, TO ANY INTELLECTUAL PROPERTY RIGHTS IS GRANTED BY THIS DOCUMENT. INTEL ASSUMES NO LIABILITY WHATSOEVER AND INTEL DISCLAIMS ANY EXPRESS OR IMPLIED WARRANTY, RELATING TO THIS INFORMATION INCLUDING LIABILITY OR WARRANTIES RELATING TO FITNESS FOR A PARTICULAR PURPOSE, MERCHANTABILITY, OR INFRINGEMENT OF ANY PATENT, COPYRIGHT OR OTHER INTELLECTUAL PROPERTY RIGHT.

Software and workloads used in performance tests may have been optimized for performance only on Intel microprocessors. Performance tests, such as SYSmark and MobileMark, are measured using specific computer systems, components, software, operations and functions. Any change to any of those factors may cause the results to vary. You should consult other information and performance tests to assist you in fully evaluating your contemplated purchases, including the performance of that product when combined with other products.

Optimization Notice

Intel’s compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804

Extreme Scalability Group Disclaimer

- I work in Intel Labs and therefore don't know anything about Intel products.
- I am not an official spokesman for Intel.
- I do not speak for my collaborators, whether they be inside or outside Intel.
- You may or may not be able to reproduce any performance numbers I report.
- Hanlon's Razor (blame stupidity, not malice).

Motivation

- Many programmers have better things to do than debug *runtime system issues*. HPC ubiquity requires things to just work.
- *Complex software* needs a language-agnostic* and language-interoperable programming model / runtime.
- Application performance dependence on communication runtime varies.

"I don't always compute in parallel, but when I do, I use MPI."

* **C (C++), Fortran (3x), Python**, Java, C#, D, Go, Perl, Ruby, Rust, Julia, Ocaml, Haskell, Pascal, Ada, …but apparently not COBOL.

MPI-1

- *Standardized a bunch of existing messaging libraries.*
- Developed when CPU much faster than network and before multicore.
- Send-Recv couple synchronization and data motion. Collectives were synchronous.
- Send-Recv requires either copy from eager buffer or partial rendezvous (to setup RDMA) due to lack of complete information about transaction on either size.
- MPI communicators are amazing for hierarchy, topology, libraries, etc.
- **Good match for a lot of numerical apps…**

Why people used MPI-1

Parallel programmers were using Intel NX, IBM MPL, P4, PARMACS, PVM, TCGMSG, Thinking Machines CMMD, Zipcode, etc. already.

Message passing a good match for:

- Dense linear algebra.
- Domain decomposition and boundary exchange.
- Numerical solvers e.g. Krylov.
- Monte Carlo

CSP is intuitive and easy to debug.

Good semantic match for *inexpensive* networks (via TCP/IP).

Incomplete programming models are incomplete

"You have put, get and atomics, but why should I re-implement collectives and messaging?"

- If someone offers you a new programming model that doesn't support what MPI 1.0 did in 1997, run away screaming.
- Collective **algorithms matter**. The alpha term is not zero.
- Broadcast/Reduction algorithms warrant specialization, even if we pretend scatter/gather and alltoall do not.

MPI-2 (sequel not better than the original)

First awareness of threads. Unfortunately, no one implemented THREAD_MULTIPLE efficiently until Blue Gene/Q. As a result, most applications rely upon THREAD_FUNNELED and fork-join threading.

Dynamic processes adopted in order to make PVM to disappear. This was arguably the only useful purpose of this feature until people started to think about resilience.

One-sided communication forced into horrible semantic corner by the existence of one strange but unfortunately #1 system (EarthSim), which was not cache-coherent.

Really dropped the ball on atomics.

*18 years later, we still do not have good **implementations** of some of these features…*

MPI-2 and Threads

```
MPI_Init_thread(.., FUNNELED);
#omp parallel
{
  for (..) { Compute(..); }
  #omp master
  { MPI_Bar(..); }
}
MPI_Foo(..);
```

```
MPI_Init_thread(.., SERIALIZE);
#omp parallel
{
  for (..) {
    Compute(..);
    #omp critical
    { MPI_Bar(..); }
  }
}
MPI_Foo(..);
```

MPI-2 and Threads

```
MPI_Init_thread(.., MULTIPLE);
#omp parallel
{
  Compute(..);
  MPI_Bar(..);
}
MPI_Foo(..);
```

This is the ONLY method that works reliably with more than one threading model!

Common

```
int MPI_Bar(..)
{
  if (MULTIPLE) Lock(Mutex);
  rc = MPID_Bar(..);
  if (MULTIPLE) Unlock(Mutex);
  return rc;
}
```

Optimized

```
int MPI_Bar(..)
{
  return MPID_Bar(..);
  /* ^ fine-grain locking
       inside of this call... */
}
```

Open-MPI does not support MPI_THREAD_MULTIPLE correctly yet. Please complain to them and use M(VA)PICH (Intel/Cray MPI) instead.

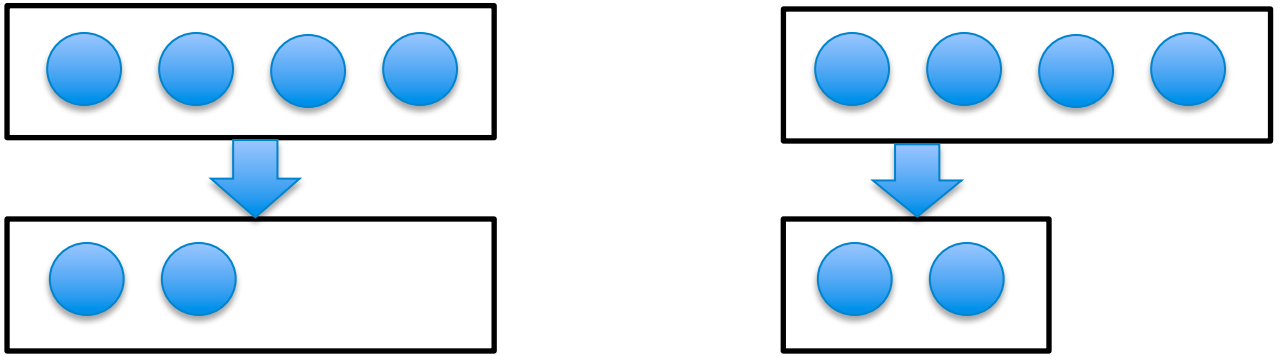
Lockless MPI_THREAD_MULTIPLE

K. Vaidyanathan, D. Kalamkar, K. Pamnany, J. Hammond, P. Balaji, D. Das, J. Park, and B. Joo. SC15.
``Improving concurrency and asynchrony in multithreaded MPI applications using software offloading.”
http://dx.doi.org/10.1145/2807591.2807602

MPI-3

- Nonblocking collectives! Implementations must get better and reward users for avoiding synchronization.
- Thread-safe Probe (Mprobe) – the right way to do active-messages in MPI-3.
- Topology is a first-class object with distributed graph communicators and neighborhood collectives.
- One-sided (RMA) communication is fixed. Supporting PGAS programming models like Global Arrays, UPC, and OpenSHMEM was an explicit goal.
- POSIX/Sys5 shared memory rolled into RMA.
- Better (non-collective) subcommunicator creation.

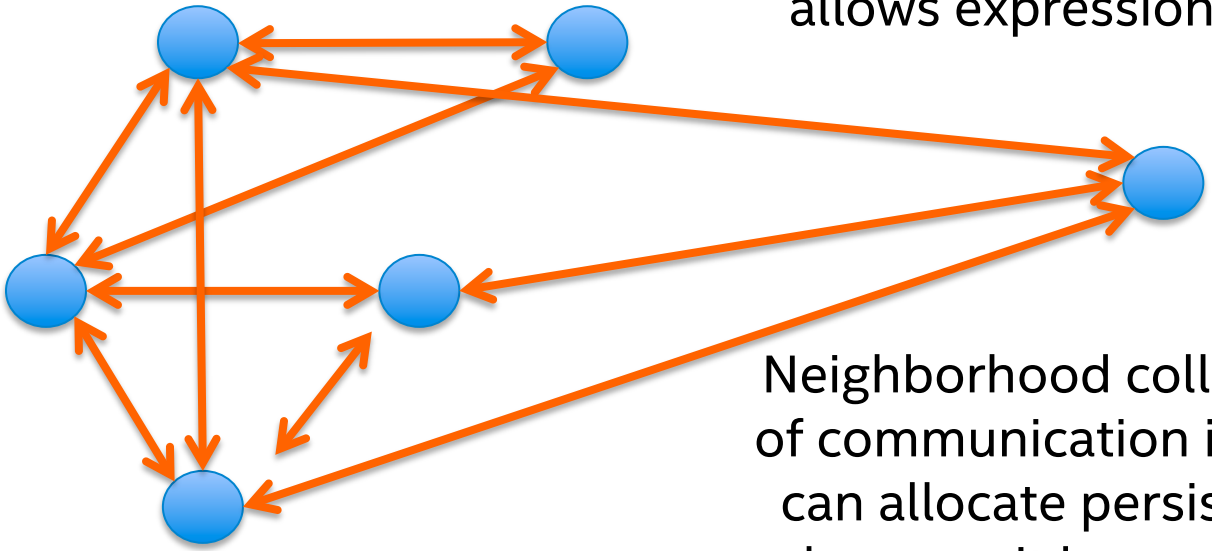
MPI-2 vs MPI-3 subcomm creation



If you have divergence in your control flow, you can't synchronize over the parent group. Form ad hoc subcomms and still use collectives with **MPI_Comm_create_group**.

Inspired by Global Arrays groups, this is also required for OpenSHMEM to use MPI collectives for anything other than (start=0,log_stride=0,size=npes).

Topology



Cartesian communicators were just the beginning – distributed graph topology allows expression of any communication pattern to the runtime.

Neighborhood collectives express O(pairs) of communication in a single call. Runtime can allocate persistent network resources because it knows the pattern in advance.

Boundary element exchange as N isend-irecv + waitall is perhaps the most common MPI pattern.

MPI-3 SHARED MEMORY

Application motivations for shared-memory

Storage bottlenecks:

- Large, lookup (WORM) tables, e.g. Quantum Monte Carlo.
- Replicated data structures that scale with job size.
- Eliminate O(ppn) halo buffers.

Communication bottlenecks:

- Load-store is (usually) faster than Send-Recv within a node.
- Complex data structures when dereferencing through indirection.
- Aggregation of small messages or I/O writes.

Threads versus processes…

Threads:

- Automatic variables (i.e. stack) all **shared by default**.
- Per-thread **privatization upon request** (OpenMP, C11, C++11,…).
- Dealing with NUMA requires OS interactions (e.g. page-faulting).
- All library calls must use mutual exclusion for shared state.

Processes:

- Automatic variables (i.e. stack) all **private by default**.
- Interprocess **sharing upon request** (Sys5, POSIX, MPI-3, XPMEM, …).
- NUMA placement done by MPI, private data naturally local.
- Mutual exclusion required only for *explicitly* shared state.

PE = Processing Element = Thread or Process

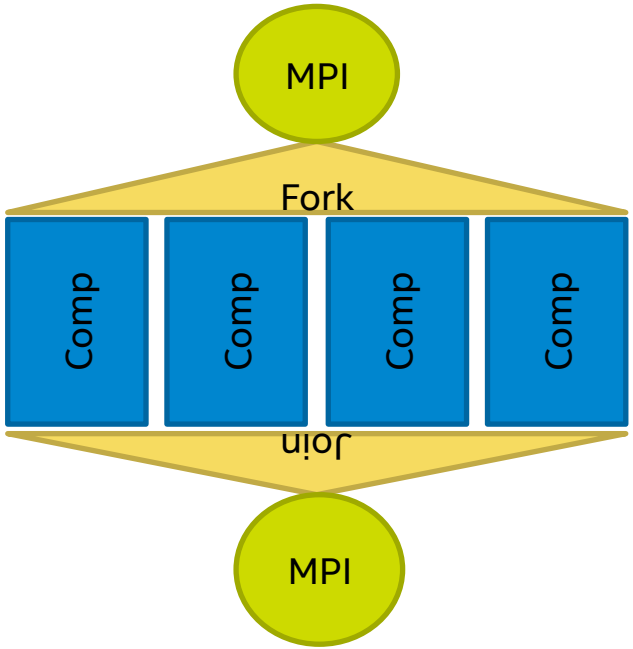
Design choices

Choose Threads:

- Data sharing: free everywhere.
- Race conditions: fork-join or mutex them all.
- Compute sharing: must parallelize extensively or Amdahl will get you.

Choose Processes:

- Data sharing: wherever necessary.
- Race conditions: only on shared data.
- Compute sharing: already done, up to MPI scalability.



Lack of libraries that exploit interprocess shared-memory is unfortunate, but compare ScaLAPACK to threaded LAPACK...

MPI-3 Shared memory

```
/* NUMA optimization */
MPI_Info_set(sheap_info, "alloc_shared_noncontig", "true");

double * my_base_ptr;
MPI_Win_allocate_shared(per_proc_shm_size, sizeof(double), sheap_info,
   node_comm, &my_base_ptr, &shm_win); /* collective ☹ */

double** all_base_ptrs= malloc( node_comm_size * sizeof(double*) );
for (int rank=0; rank<node_comm_size ; rank++) {
   MPI_Aint size;
   int disp;
   MPI_Win_shared_query(shm_win, rank, &size, &disp, &all_base_ptrs[rank]);
}
```

Exascale Computing without Threads*

A White Paper Submitted to the
DOE High Performance Computing Operational Review (HPCOR)
on Scientific Software Architecture for Portability and Performance
August 2015

Matthew G. Knepley[1], Jed Brown[2], Barry Smith[2], Karl Rupp[3], and Mark Adams[4]
[1]Rice University, [2]Argonne National Laboratory, [3]TU Wien, [4]Lawrence Berkeley National Laboratory
knepley@rice.edu, [jedbrown,bsmith]@mcs.anl.gov, rupp@iue.tuwien.ac.at, mfadams@lbl.gov

http://www.orau.gov/hpcor2015/whitepapers/Exascale_Computing_without_Threads-Barry_Smith.pdf

MPI-3 ONE-SIDED COMMUNICATION

MPI-3 window constructor options

Window ctor	Buffer	Restrictions	T/S*
Alloc_mem, Win_create	input	static, coll.	B
Win_allocate	output	static, coll.	A
Win_allocate_shared	output	ld/st domain	A+
Alloc_mem, Win_{create_dynamic,attach}	input	-	?

- Win_create cannot use symmetric memory, likely will not allocate shm or registered buffers without info keys.
- Dynamic windows require not-yet-standard info keys to cache RDMA metadata, in addition to the restrictions of Win_create.
- Win_allocate_shared hopefully deprecated (into Win_allocate) in MPI-4.

* Time/space, i.e. performance and metadata scalability. Details available elsewhere.

MPI RMA memory allocation

- All RMA operations act on windows, which are handles to opaque objects that describe memory on which RMA can act.
- MPI-2 had one way to construct a window. MPI-3 added 2.5 new ways. All of them are formally collective (more on this later).
- Most PGAS models require a suballocator, compiler and/or OS hooks for memory management in general…

The purpose of multiple window constructors is to make the tradeoffs between flexibility and performance explicit. MPI is nothing if not explicit.

Synchronization epochs

```
MPI_Win w;
/* construct window */
MPI_Win_lock_all(MPI_MODE_NOCHECK,w); /* “PGAS mode” */
{
      …
      MPI_Put(..,pe,w); /* all RMA communications are nonblocking */
      MPI_Win_flush_local(pe,w); /* local completion */
      MPI_Win_flush (pe,w); /* remote completion = global visibility */
      …
}
MPI_Win_unlock_all(w);
MPI_Win_free(w);
```

This is the **only** synchronization motif ~~PGAS~~ programmers should ever use.

Direct local access

```
int * ptr; MPI_Win w;
MPI_Win_{allocate_shared,shared_query}(&ptr,&w);
...
if (pe==0) {
      MPI_Put(..,pe=1,w); /* Write */
      MPI_Win_flush (pe=1,w); /* Release */
      MPI_Send(..,pe=1); /* Send */
} else if (pe==1) {
      MPI_Recv(..,pe=0); /* Recv */
      MPI_Win_sync(w); /* Acquire*/
      int tmp = *ptr; /* Read */
}
```

This approach to memory consistency is consistent with OpenMP “flush”…

Direct local access

```
#include <stdatomic.h>;
...
if (pe==0) {
   *ptr = 0x86; /* Write */
   atomic_thread_fence(...release); /* Release */
   MPI_Send(..,pe=1); /* Send */
} else if (pe==1) {
   MPI_Recv(..,pe=0); /* Recv */
   atomic_thread_fence(...acquire); /* Acquire */
   int tmp = *ptr; /* Read */
}
```

- Shared-memory is equivalent to threads.
- Threads cannot be implemented as a library.
- MPI is a library.

→ Use language (C11 or C++11) features instead of MPI_WIN_SYNC*.

Warning: This code is meant to be illustative. I did not verify correctness.

* This is Bill Gropp's current position. Fortran needs a memory model.

Direct local access

```
#include <stdatomic.h>;
atomic_flag *flag; MPI_Win wf;
MPI_Win_{allocate_shared,shared_query}(&flag,&wf);
ATOMIC_FLAG_INIT(*flag);
...
if (pe==0) {
   atomic_store_explicit(ptr,0x86,release); /* Write + Release*/
   atomic_store_explicit(flag,1,release); /* Send */
} else if (pe==1) {
   while (!atomic_load_explicit(flag,acquire)); /* Recv */
   int tmp = atomic_load_explicit(ptr,acquire); /* Acquire + Read */
}
```

Warning: This code is meant to be illustative. I did not verify correctness.

Here MPI is just a portable wrapper around shared memory.

MPI-3 RMA COMMUNICATION OPERATIONS

Ordering and atomicity (memory model)

- Ordered memory operations easy on programmers, hard on networks.
- Atomic memory operations easy on programmers, hard on hardware.
- MPI-3 provides both:
  - Put/Get neither atomic nor ordered.
  - Accumulate/Get_accumulate always *element-wise* atomic, ordered* by default. User requests relaxation of ordering (RAR & RAW & WAR & WAW).

AtomicPut = MPI_Accumulate(operation=MPI_REPLACE,..)
AtomicGet = MPI_Get_accumulate(operation=MPI_NO_OP,..)

* Pairwise ordering only.

Atomic operations

- Atomicity is per window+operation+datatype *within an epoch*.
- MPI-3 default of `same_op_no_op` is quite restrictive. E.g. Fetch, Set and Inc is a reasonable usage that MPI-3 doesn't support (FIXME).
- OpenSHMEM currently allows all operation+datatype atomicity (~~FIXME~~).
- MPI-3 semantics restrictive because of operations like MPI_PROD, which no one implements in hardware. Need fine-grain subset control (FIXME).
- Can emulate ops w/o HW using Fetch+Op+CAS (may suffer from starvation).
- Multi-element atomicity requires exclusive or higher-level mutual exclusion.

Implementation, implementation, implementation!

SC13 Concludes with Awards for Outstanding Achievements in HPC

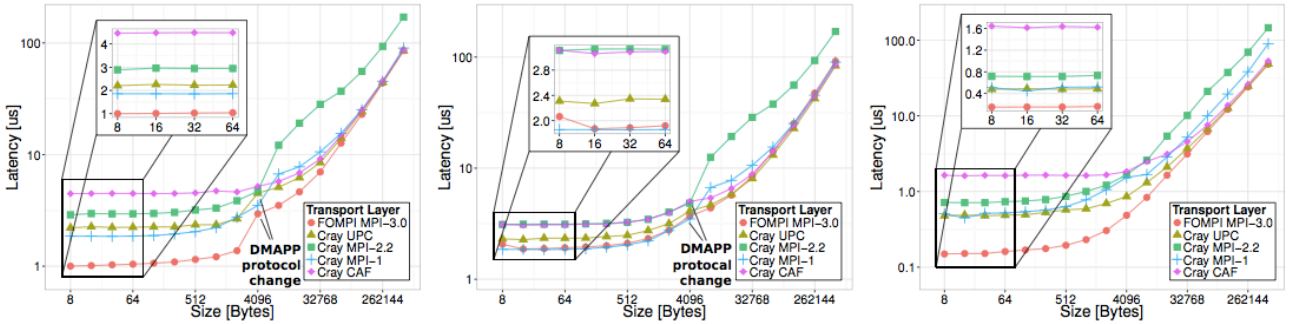
DENVER, CO.—SC13, the 25th anniversary conference of high performance computing, networking, storage and analysis, celebrated the contributions of researchers, from those just starting their careers to those whose contributions have made lasting impacts, in a special awards session on Thursday, Nov. 21.

The conference drew 10,613 registered attendees who attended a technical program spanning six days and viewed the offerings of 335 exhibitors in the Colorado Convention Center.

The following awards were presented at the conference:

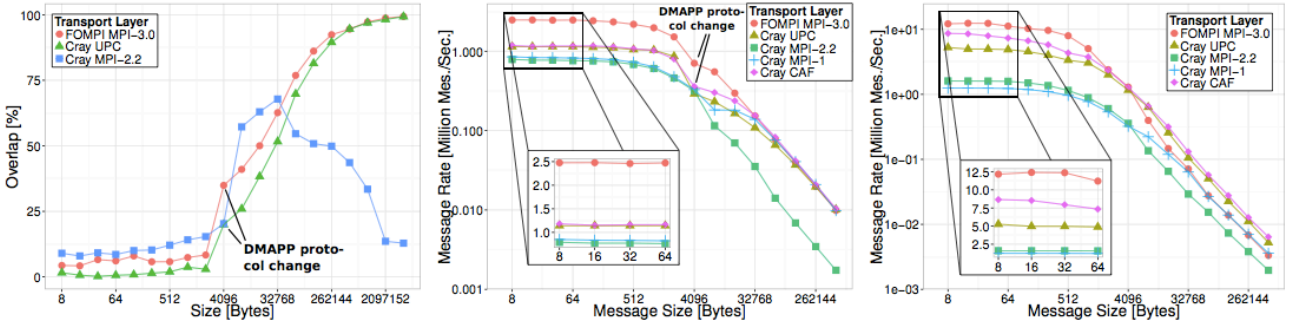
The **Best Paper Award** went to *"Enabling Highly-Scalable Remote Memory Access Programming with MPI-3 One Sided,"* written by Robert Gerstenberger, University of Illinois at Urbana-Champaign, and Maciej Besta and Torsten Hoefler, both of ETH Zurich.

The Best Paper Award went to "Enabling Highly-Scalable Remote Memory Access Programming with MPI-3 One Sided," written by Robert Gerstenberger, University of Illinois at Urbana-Champaign, and Maciej Besta and Torsten Hoefler, both of ETH Zurich.



(a) Latency inter-node Put (b) Latency inter-node Get (c) Latency intra-node Put/Get

Figure 4: Latency comparison for remote put/get for DMAPP and XPMEM (shared memory) communication. Note that MPI-1 Send/Recv implies remote synchronization while UPC, Fortran Coarrays and MPI-2.2/3.0 only guarantee consistency.



(a) Overlap inter-node (b) Message Rate inter-node (c) Message Rate intra-node

Figure 5: Communication/computation overlap for Put over DMAPP, Cray MPI-2.2 has much higher latency up to 64 kB (cf. Figure 4a), thus allows higher overlap. XPMEM implementations do not support overlap due to the shared memory copies. Figures (b) and (c) show message rates for put communication for all transports.

This figure was taken from the Besta and Hoefler SC13 paper.

STATUS OF PGAS MODELS IN MPI-3

Status

Model	Project	Team	Status
Global Arrays	ARMCI-MPI	*Argonne/Intel*	In production for NWChem
OpenSHMEM	OSHMPI	*Argonne/Intel*	Useful for research, SMPs
Fortran 2008	OpenCoarrays	Sourcery Institute, *et al.*	GCC 5+
Fortran 2008	Intel Fortran	Intel	Intel 13+ (?) - get latest
Fortran coarrays	CAF 2.0	Rice/Argonne	Published
UPC	GUPC	Intrepid, *et al.*	Evaluating

http://git.mpich.org/armci-mpi.git/

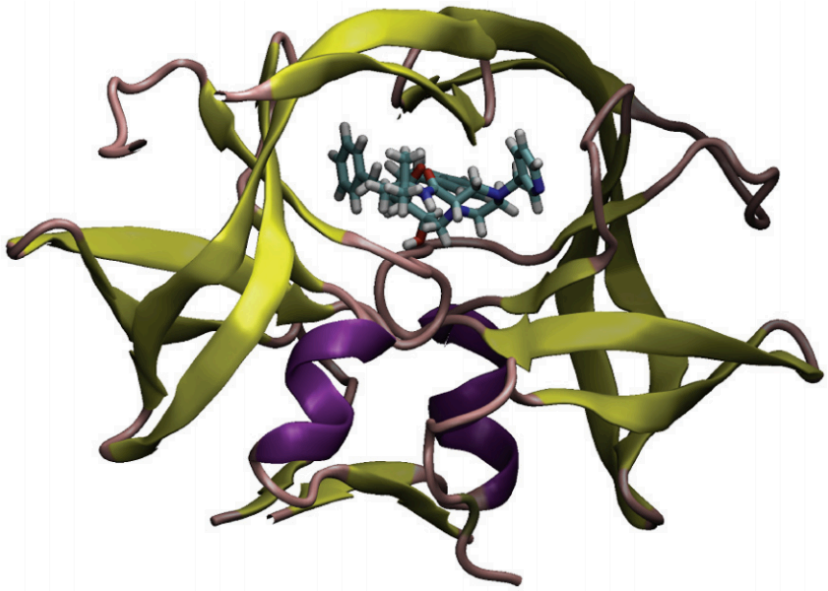
https://github.com/jeffhammond/oshmpi

http://www.opencoarrays.org/

https://goo.gl/6cQuN3 (Intel)

NWChem Evaluation

- 1hsg_28 benchmark system from Chow et al. (see below)
- 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
(aside: GTFock used GA/ARMCI-MPI and MPICH-Glex for these petascale runs.)

NWChem SCF performance (old)

NWChem 6.3/ARMCI-MPI3/Casper

iter	energy	time
1	-2830.4366669992	69.6
2	-2831.3734512508	78.8
3	-2831.5712563433	86.9
4	-2831.5727802438	96.1
5	-2831.5727956882	110.0
6	-2831.5727956978	**127.8**

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

iter	energy	time
1	-2830.4366670018	67.6
2	-2831.3734512526	85.5
3	-2831.5713109544	105.4
4	-2831.5727856636	126.6
5	-2831.5727956992	161.7
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

iter	energy	time
1	-2830.4366669990	**69.3**
2	-2831.3734512499	77.1
3	-2831.5712604368	84.6
4	-2831.5727804428	93.0
5	-2831.5727956927	107.3
6	-2831.5727956977	**128.0**

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

iter	energy	time
1	-2830.4366669999	**61.4**
2	-2831.3734512509	69.3
3	-2831.5713109521	77.8
4	-2831.5727856618	87.3
5	-2831.5727956974	103.9
6	-2831.5727956980	**125.7**

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

NWChem SCF performance (new)

NWChem 6.3/ARMCI-MPI3/Casper

iter	energy	time
1	-2830.4366670122	23.7
2	-2831.3734512625	27.4
3	-2831.5713109936	31.3
4	-2831.5727856739	35.4
5	-2831.5727957093	42.6
6	-2831.5727957100	**51.6**

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

iter	energy	time
1	-2830.4366670122	23.5
2	-2831.3734512625	27.0
3	-2831.5713109936	30.9
4	-2831.5727856739	34.7
5	-2831.5727957093	41.4
6	-2831.5727957100	**49.7**

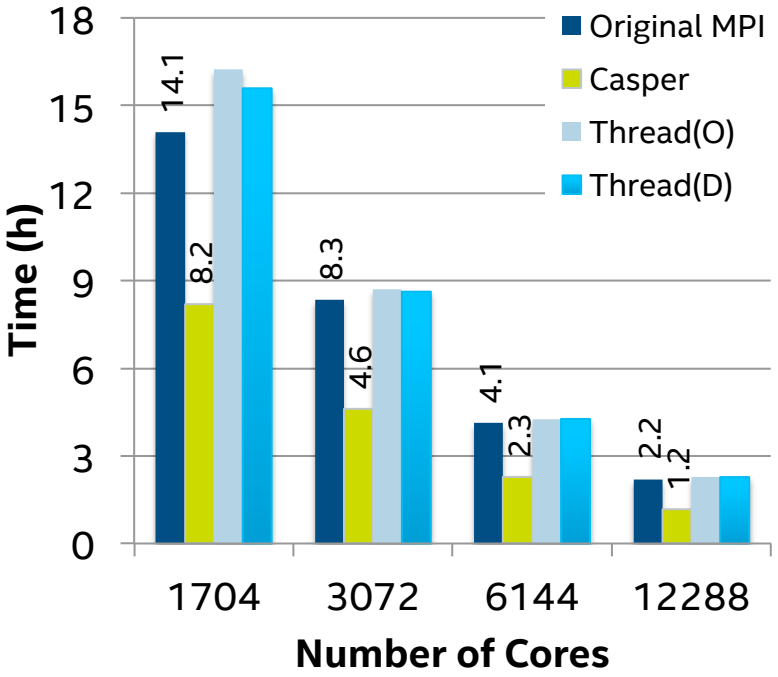
Running on 20 nodes with 24 ppn. **Both** use 2 ppn for comm.

ARMCI-MPI

MPI-2: contort ARMCI to RMA semantics; nonblocking wasn't.

MPI-3: +atomics, +nonblocking; ~async.

Casper: MPI-3 +async.



Application usage

- NWChem on 40K+ cores of Cray XC30.
- Performance-competitive w/ ARMCI@DMAPP when using Casper for *latency-sensitive* NWChem AO-DFT code.
- Bandwidth-limited CCSD(T) for $(H_2O)_{21}$.

M. Si, A. J. Pena, J. Hammond, P. Balaji, M. Takagi, Y. Ishikawa. IPDPS15.
"Casper: An Asynchronous Progress Model for MPI RMA on Many-Core Architectures."

MPI WITH LARGE COUNTS

A Simple Problem

- int MPI_Send(const void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm)
- It is not MPI's problem that a C int is 32 bits everywhere. If you use ISO C or Fortran incorrectly and overflow that type, your code is WRONG.
- It is MPI's fault that it kept int instead of switching to size_t like almost everyone else.
- ABI (Application Binary Interface) compatibility is a big deal in the MPI world for a variety of reasons. The MPI Forum did not want to break ABI compatibility with MPI-3, given the relatively limited use case for large-count operations and known workarounds for the common cases.

BigMPI: You can haz moar counts!

To INT_MAX...and beyond!
Exploring large-count support in MPI

Jeff R. Hammond
Parallel Computing Lab
Intel Corp.
jeff_hammond@acm.org

Andreas Schäfer
Friedrich-Alexander-Universität
Erlangen-Nürnberg
andreas.schaefer@fau.de

Rob Latham
Mathmatics and Computer Science Division
Argonne National Lab.
robl@mcs.anl.gov

https://github.com/jeffhammond/BigMPI
https://github.com/jeffhammond/BigMPI-paper

Large counts in MPI-4

1. Add select new functions with _x suffix.
2. Static polymorphism via Fortran 2008 interfaces, C++ overloading and C11 `_Generic` to hide new interfaces.
3. Do nothing and continue to frustrate users.

MPI-4 FAULT TOLERANCE

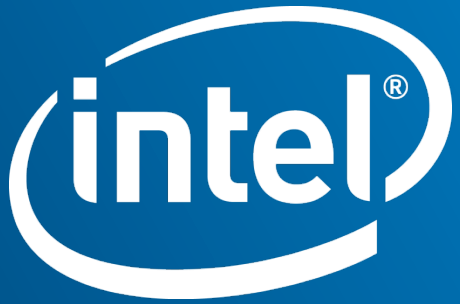
Can we do exascale without this???

With great functionality, comes great responsibility

- Supporting fault-tolerance (FT) in a point-to-point, connect-based communication interface is easy.
- Supporting fault-tolerance in a latency-agnostic (i.e. slow) way is easy.
- Connectionless, communicators, collectives – these are what make MPI FT hard, and what make MPI great.
- If fault-tolerance vs performance+functionality trade-offs were favorable for HPC users, we would see adoption of TCP/IP.
- ULFM: https://github.com/mpi-forum/mpi-issues/issues/20 (21, 22)
- FENIX (http://dx.doi.org/10.1109/SC.2014.78) looks like a good client…

Conclusions

- MPI is not just about message-passing anymore. It is the lingua franca for distributed-memory HPC.
- MPI continues to be portable, composable, language-agnostic, and orthogonal (use only what you need).
- Performance-oriented changes, fault-tolerance and large-count support should be coming.


intel®