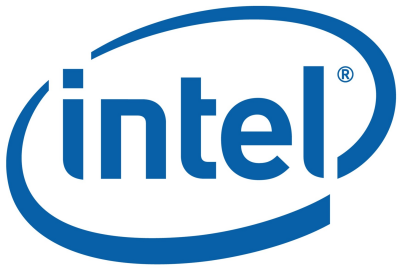
# The MPI+MPI programming model and why we need shared-memory MPI libraries

Jeff Hammond

Extreme Scalability Group & Parallel Computing Lab
Intel Corporation (Portland, OR)

26 September 2014



# Legal Disclaimer & Optimization Notice

INFORMATION IN THIS DOCUMENT IS PROVIDED IN CONNECTION WITH INTEL PRODUCTS. NO LICENSE, EXPRESS OR IMPLIED, BY ESTOPPEL OR OTHERWISE, TO ANY INTELLECTUAL PROPERTY RIGHTS IS GRANTED BY THIS DOCUMENT. EXCEPT AS PROVIDED IN INTEL'S TERMS AND CONDITIONS OF SALE FOR SUCH PRODUCTS, INTEL ASSUMES NO LIABILITY WHATSOEVER AND INTEL DISCLAIMS ANY EXPRESS OR IMPLIED WARRANTY, RELATING TO SALE AND/OR USE OF INTEL PRODUCTS 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 work for Intel, but I am not an official spokesman for Intel. Hence anything I say are my words, not Intel's. Furthermore, 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.
- Performance numbers for non-Intel platforms were obtained by non-Intel people.
- Hanlon's Razor.

# Abstract (for posterity)

The MPI-3 standard provides a portable interface to interprocess shared-memory through the RMA functionality. This allow applications to leverage shared-memory programming within a strictly MPI paradigm, which mitigates some of the challenges of MPI+X programming using threads associated with shared-by-default behavior and race conditions, NUMA and Amdahl's law. I will describe the MPI shared-memory capability and how it might be targeted by existing multithreaded libraries.

# MPI-3

# Quiz

What is **MPI**?
(A) A bulky, bulk-synchronous model.
(B) The programing model of Send-Recv.
(C) An explicit, CSP-like, private-address-space programming model.
(D) An industry-standard runtime API encapsulating 1-, 2- and *N*-sided blocking and nonblocking communication and a whole bunch of utility functions for library development.
(E) The assembling language of parallel computing!!

# The MPI You Know

```
MPI_Init(..);
MPI_Comm_size(..); MPI_Comm_rank(..);
MPI_Barrier(..); MPI_Bcast(..);
MPI_Reduce(..); MPI_Allreduce(..);
MPI_Gather(..); MPI_Allgather(..);
MPI_Scatter(..); MPI_Alltoall(..);
MPI_Reduce_scatter(..); MPI_Reduce_scatter_block(..);
MPI_Send(..); MPI_Recv(..); /* [b,nb] x [r,s,b] */
...
MPI_Finalize();
```

# The MPI You Have Heard of But Don't Use

```
MPI_Ibarrier(..); MPI_Ibcast(..);
MPI_Ireduce(..); MPI_Iallreduce(..);
MPI_Igather(..); MPI_Iallgather(..);
MPI_Iscatter(..); MPI_Ialltoall(..);
MPI_Ireduce_scatter(..);
MPI_Ireduce_scatter_block(..);
```

Go forth a write bulk-asynchronous code!

# The MPI You Don't Know But Should

```
MPI_Comm_create_group(..);
MPI_Icomm_dup(..);
...
MPI_Dist_graph_create_adjacent(..);
MPI_Neighborhood_allgather(..);
MPI_Neighborhood_allgatherv(..);
MPI_Neighborhood_alltoall(..);
```

Virtual topologies corresponding to algorithmic topology; additional semantic information enables MPI to optimize.

# The MPI You Don't Know and Might Not Want To

```
Win_create(..); Win_allocate(..);
Win_allocate_shared(..); Win_shared_query(..);
Win_create_dynamic(..); Win_attach(..);
Win_detach(..);
Put(..); Get(..); Accumulate(..);
Fetch_and_op(..); Compare_and_swap(..);
Win_lock(..); Win_lock_all(..);
Win_flush(_local)(_all)(..); Win_sync(..);
...
```

MPI-3 is a superset of ARMCI and OpenSHMEM...

http://wiki.mpich.org/armci-mpi/
https://github.com/jeffhammond/oshmpi/

# Shared Memory implementations

What is MPI_Win_allocate_shared(..)?

Historically, SysV shared memory used, but painfully.

POSIX shared memory good, but Windows, BSD/Mach...

In HPC, we have XPMEM (Cray and SGI). And BGQ...

MPI processes can be threads, in which case, all is shared.

The purpose of MPI is to standardize best practice!

Shared-memory is a "best practice."

# MPI-3 Shared Memory

Limitations:

- Only defined for cache-coherent systems (WIN_MODEL=UNIFIED).
- Allocated collectively.
- Memory allocated contiguously *by default*.

Features:

- It's SHARED MEMORY: what don't you love?
- Works together with RMA ops (e.g. atomics).
- Noncontiguous allocation upon request (hint).

# MPI+X

# The future is MPI+X (supposedly)

- MPI+OpenMP is too often fork-join.
- Pthreads scare people; can't be used from Fortran (easily).
- Intel® has Cilk® and TBB.
- OpenCL is not a good model for application programmers and has no magic for portable performance (since such magic does not exist).
- CUDA® is an X for only one type of hardware (ignoring Ocelot).

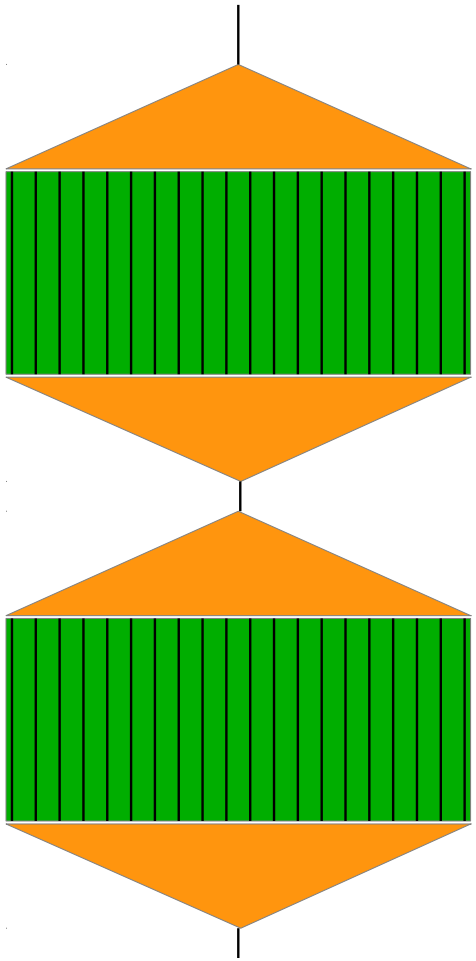
Never confuse portability with portable performance!

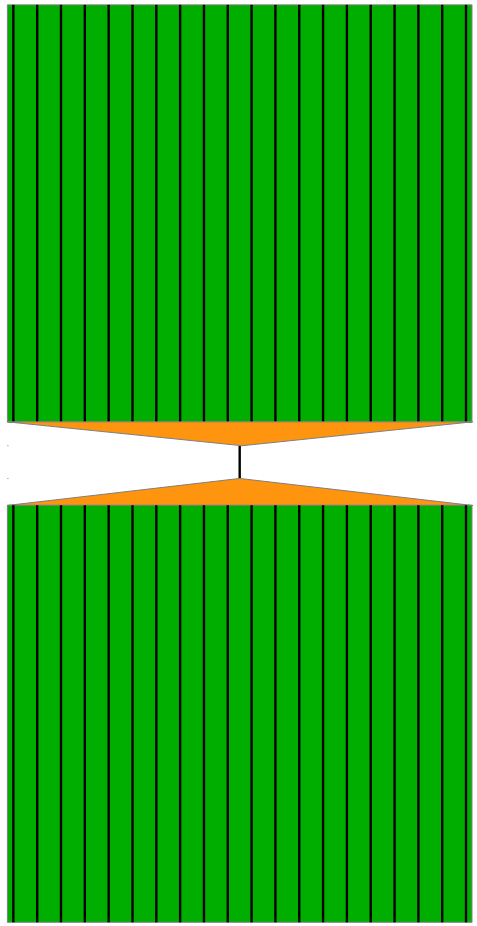
# Using MPI+OpenMP effectively

- Private data should behave like MPI but with load-store for comm.
- Shared data leads to cache reuse but also false sharing.
- NUMA is going to eat you alive. BG is a rare exception.
- OpenMP offers little to no solution for NUMA.
- If you do everything else right, Amdahl is going to get you.

Intranode Amdahl and NUMA are giving OpenMP a bad name; fully rewritten hybrid codes that exploit affinity behave very different from MPI codes evolved into MPI+OpenMP codes.

# Fork-Join vs. Parallel-Serialize





# Fork-Join vs. Parallel-Serialize

```
#pragma omp parallel
{
/* thread-safe */

#pragma omp single
/* thread-unsafe */

#pragma omp parallel for
/* threaded loops */

#pragma omp sections
/* threaded work */
}
```

```
/* thread-unsafe */

#pragma omp parallel for
{
/* threaded loops */
}

/* thread-unsafe */

#pragma omp parallel for
{
/* threaded loops */
}

/* thread-unsafe work */
```

# NUMA

This is a toy DAXPY-like test I wrote for an ALCF tutorial...

```
> for n in 1e6 1e7 1e8 1e9 ; do ./numa.x $n ; done
n = 1000000    a: 0.009927 b: 0.009947
n = 10000000   a: 0.018938 b: 0.011763
n = 100000000  a: 0.123872 b: 0.072453
n = 1000000000 a: 0.915020 b: 0.811122
```

The first-order effect requires a multi-socket system.

For more complicated data access patterns, you may see this even with parallel initialization.

MPI⊗X

- Threads are independent, long-lived tasks in a shared address space.
- Threads all access MPI like they own it.
- `MPI_THREAD_MULTIPLE` is non-trivial overhead.
- Be sure you have a communicator per thread with collectives...
- If your low-level network stack is not thread-safe...
- God help you if you want to mix more than one threading model![1]

[1] See `https://www.ieeetcsc.org/activities/blog/challenges_for_interoperability_of_runtime_systems_in_scientific_applications`

# MPI+MPI

# Best of all worlds?

MPI-1 between nodes; MPI-Shm within the node...

- Private by default; shared by request. Safe.
- Memory affinity to each core; NUMA issues should be rare.
- No fork-join - end-to-end parallel execution, just a question of replicated or distributed (GA-like).
- No need to reimplement *any* collectives.
- Easily supports both task- and data-parallelism.
- Hierarchy via MPI communicators.
- One runtime to rule them all. No interop BS.
- `MPI_THREAD_SINGLE` sufficient.

# Why not MPI+MPI?

- MPI shm allocation collective.
- MPI shm allocator not `malloc`.
- No cure for data races.
- Data races not cured.
- cured. races not Data
- All the intranode libraries use threads!!!

# MPI-Shm libraries 1

**BLIS** should be the first MPI-Shm library:

- BLIS thread communicator maps perfectly to MPI communicator.
- Need to put BLIS communicator outside of API calls, but that's the only major change I can see.
- Tyler's implementation with OpenMP is trivially mapped to MPI calls.
- API refactoring for this is incredibly useful in threading models for task-parallelism and batching.

# MPI-Shm libraries 2

**Elemental** should be the second MPI-Shm library:

- Does OpenMP really meet the needs of Elemental within a node?
- Lots of people don't want to think about hybrid, just MPI-only.
- Elemental with MPI-Shm within node could compete with threaded libraries and might beat them because of well-known fork-join issues in LAPACK.
- DistMatrix object hides all of the allocation issues internally, as it's already collective.
- We have an MPI-3 RMA AXPY implementation as a related proof-of-concept.

# MPI is dead. Long live MPI!

# Acknowledgements

Tyler Smith and Jed Brown, for explaining and discussing thread communicators at length.

Jack Poulson, for Elemental discussions over the years.

NUMA and Amdahl's Law, for holding OpenMP back and keeping MPI-only competitive in spite of the ridiculous cost of Send-Recv within a shared-memory domain.