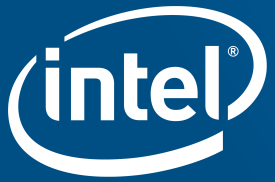


THREADS: THE WRONG ABSTRACTION AND THE WRONG SEMANTIC

Jeff Hammond
Parallel Computing Lab
Intel Corporation

Abstract

MPI+OpenMP is frequently proposed as the right evolutionary programming model for exascale. Unfortunately, the evolutionary introduction of OpenMP into existing MPI-only codes is fraught with difficulty. We will describe "The Right Way" to do MPI+OpenMP and ultimately conclude that MPI+MPI is a more effective alternative for legacy codes.

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

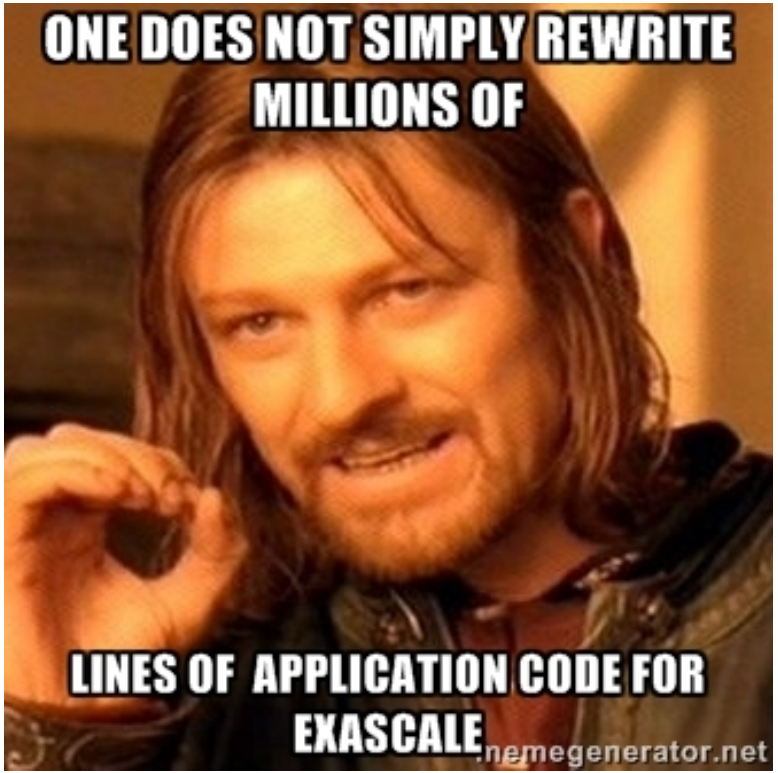
Extreme Scalability Group Disclaimer

- 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).

HPC software design challenges

- To MPI or not to MPI…
- One-sided vs. two-sided?
- Does your MPI/PGAS need a +X?
- Static vs. dynamic execution model?
- What synchronization motifs maximize performance across scales?

Application programmers can afford to rewrite/redesign applications zero to one times every 20 years…



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

MPI+THREADS

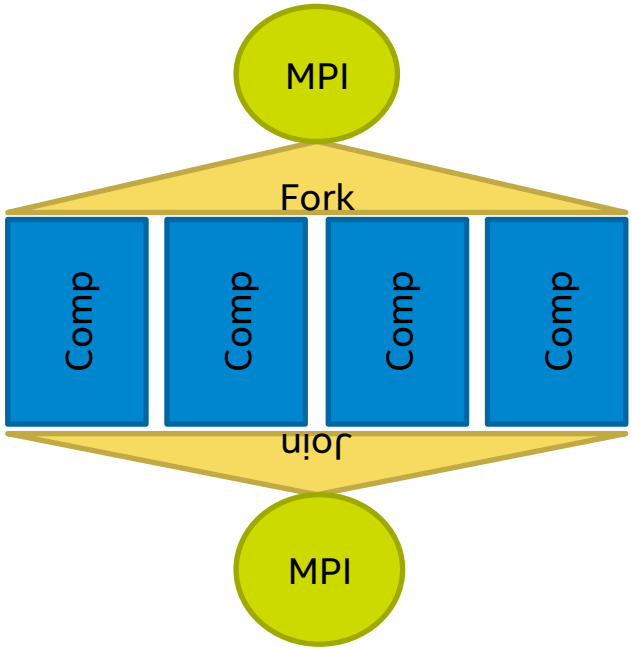
Design choices

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-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.

Optimization work on threaded MPI

- P. Balaji, D. Buntinas, D. Goodell, W. D. Gropp, and R. Thakur. 2010. Fine-Grained Multi- threading Support for Hybrid Threaded MPI Programming. Int. J. High Perform. Comput. Appl. 24 (Feb. 2010), 49–57.
- D. Goodell, P. Balaji, D. Buntinas, G. Dozsa, W. Gropp, S. Kumar, B. R. de Supinski, and R. Thakur. 2010. Minimizing MPI Resource Contention in Multithreaded Multi- core Environments. In Proceedings of the 2010 IEEE International Conference on Cluster Computing (CLUSTER '10). IEEE Computer Society, Washington, DC, USA, 1–8.
- G. Dozsa, S. Kumar, P. Balaji, D. Buntinas, D. Goodell, W. Gropp, J. Ratterman, and R. Thakur. 2010. Enabling Concurrent Multithreaded MPI Communication on Multicore Petascale Systems. In Proceedings of the 17th European MPI Users' Group Meeting Conference on Re- cent Advances in the Message Passing Interface (EuroMPI'10). Springer-Verlag, Berlin, Heidelberg, 11–20.
- A. Amer, H. Lu, Y. Wei, P. Balaji, and S. Matsuoka. 2015. MPI+Threads: runtime contention and remedies. In Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming. ACM, 239–248.
- 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

Software offloading for MPI_THREAD_MULTIPLE

- Application code is consistent with MPI_THREAD_MULTIPLE; implementation only requires MPI_THREAD_FUNNELED.
- Assumes agent thread can:
  - keep up with application
  - drive network
- Good for common use of NB p2p with bolt-on OpenMP.
- Side-effect: asynchronous progress

```
MPI_Isend(ARGS)
{
 /* insert uses atomics to be
    thread-safe without locking */
 insert(&queue,ARGS);
}
/* agent runs in a polling thread */
agent_function()
{
 ARGS = remove(&queue);
 PMPI_Send(ARGS);
}
```

PRK

Programming model evaluation

Standard methods

- NAS Parallel Benchmarks
- Mini Applications
  (e.g. Mantevo, LULESH)
- HPC Challenge

There are numerous examples of these on record, covering a wide range of programming models, but is source available and curated*?

What is measured?

- Productivity (?), elegance (?)
- Implementation quality
  (runtime or application)
- Asynchrony/overlap
- Semantics:
  - Automatic load-balancing (AMR)
  - Atomics (GUPS)
  - Two-sided vs. one-sided, collectives

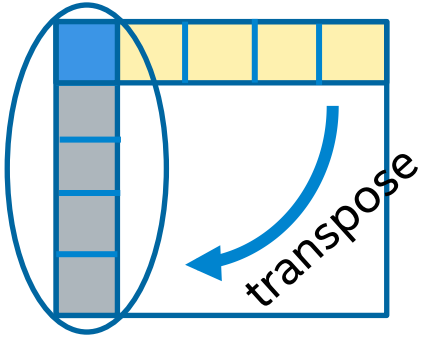
* PRK curation is TBD.

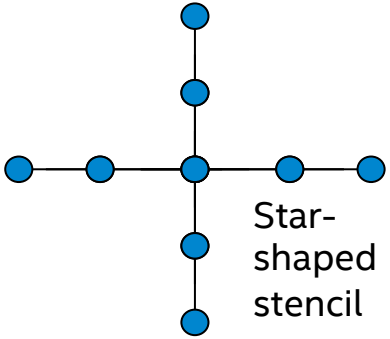
Goals of the Parallel Research Kernels

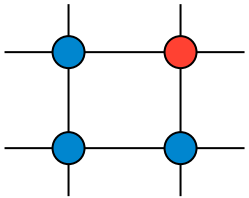
1. Universality: Cover broad range of performance critical application patterns.
2. Simplicity: Concise pencil-and-paper definition and transparent C reference implementation. *No domain knowledge required.*
3. Portability: Should be implementable in any sufficiently general programming model.
4. Extensibility: Parameterized to run at any scale. Other knobs to adjust problem or algorithm included.
5. Verifiability: Automated correctness checking and built-in performance metric evaluation.
6. ~~Hardware benchmark~~: No! Use HPCChallenge, Xyz500, etc. for this.

Outline of PRK Suite

- ***Dense matrix transpose***
- Synchronization: global
- ***Synchronization: point to point***
- Scaled vector addition
- Atomic reference counting
- Vector reduction
- Sparse matrix-vector multiplication
- Random access update
- ***Stencil computation***
- Dense matrix-matrix multiplication
- Branch
- Particle-in-cell







$$A_{i,j} = A_{i-1,j} + A_{i,j-1} - A_{i-1,j-1}$$

PRK implementations

- Serial
- OpenMP
- MPI1 – MPI two-sided
  - FG-MPI – MPI1 using Fine Grain MPI from UBC
  - AMPI – MPI1 using Adaptive MPI from
- MPIOMP – MPI two-sided with local OpenMP
- MPISHM – MPI two-sided with MPI-3 shared-memory
- MPIRMA – MPI one-sided communication (multiple flavors)
- SHMEM
- UPC
- Fortran 2008 (serial, OpenMP, coarrays, intrinsics)
- Python (simple and Numpy)
- Grappa (C++)
- Charm++ (C++)

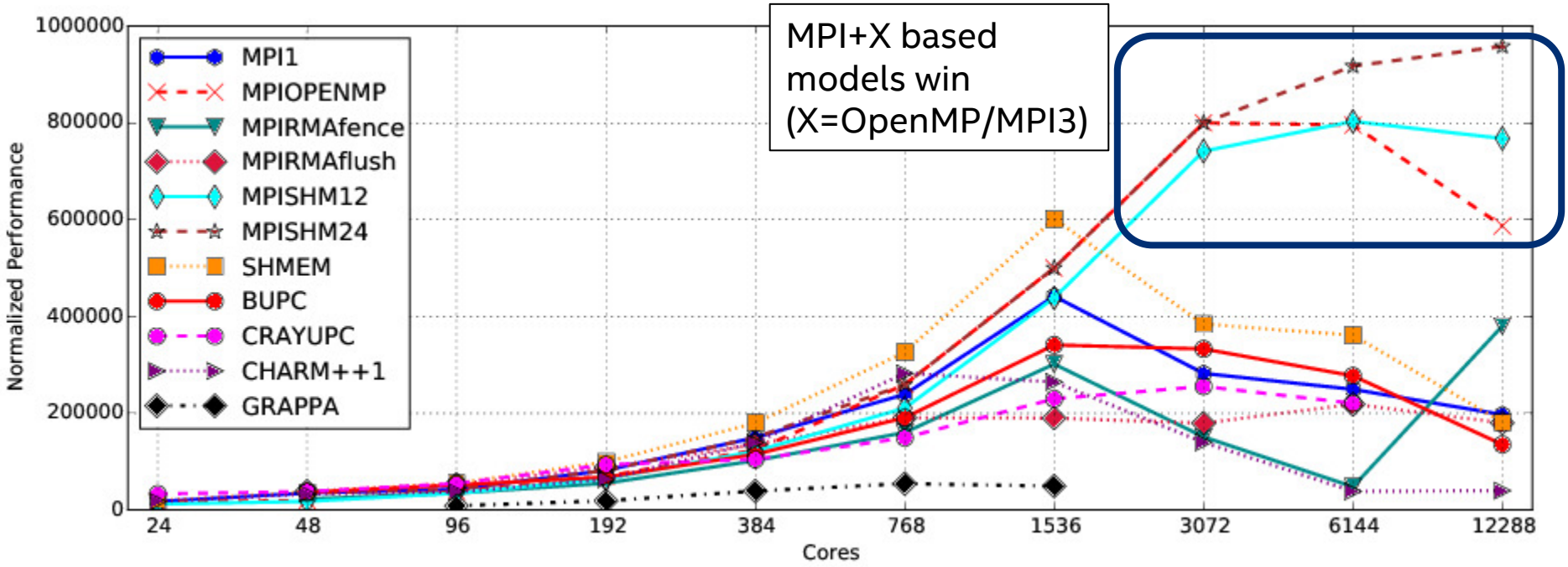
OpenMP, Serial and MPI support most of the PRKs. **Synch_p2p**, **Stencil** and **Transpose** are primary targets for distributed-memory evaluation.

In progress:
Legion (Stanford)
HPX (LSU & IU)
OCR (Rice/Intel)
Chapel (Cray)

Experimental apparatus

See paper for details.

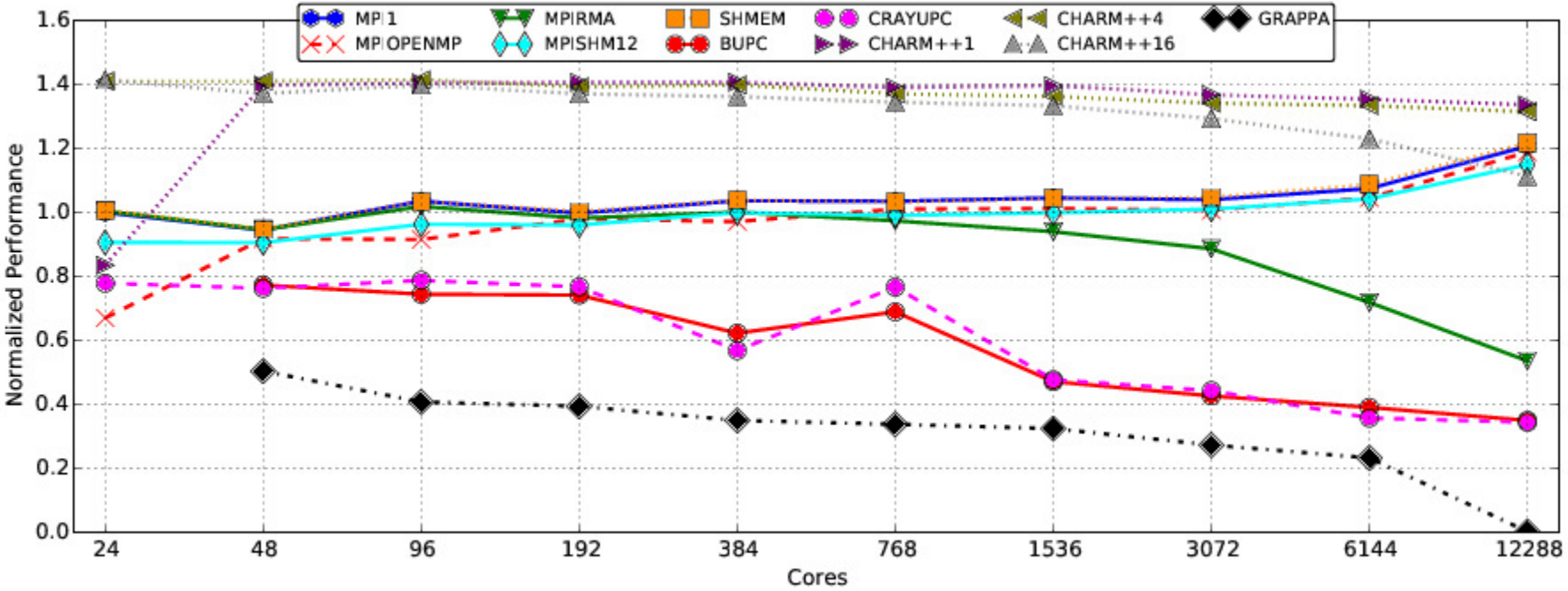
Transpose, strong scaled (49152x49152*)



Aggregate performance MB/s

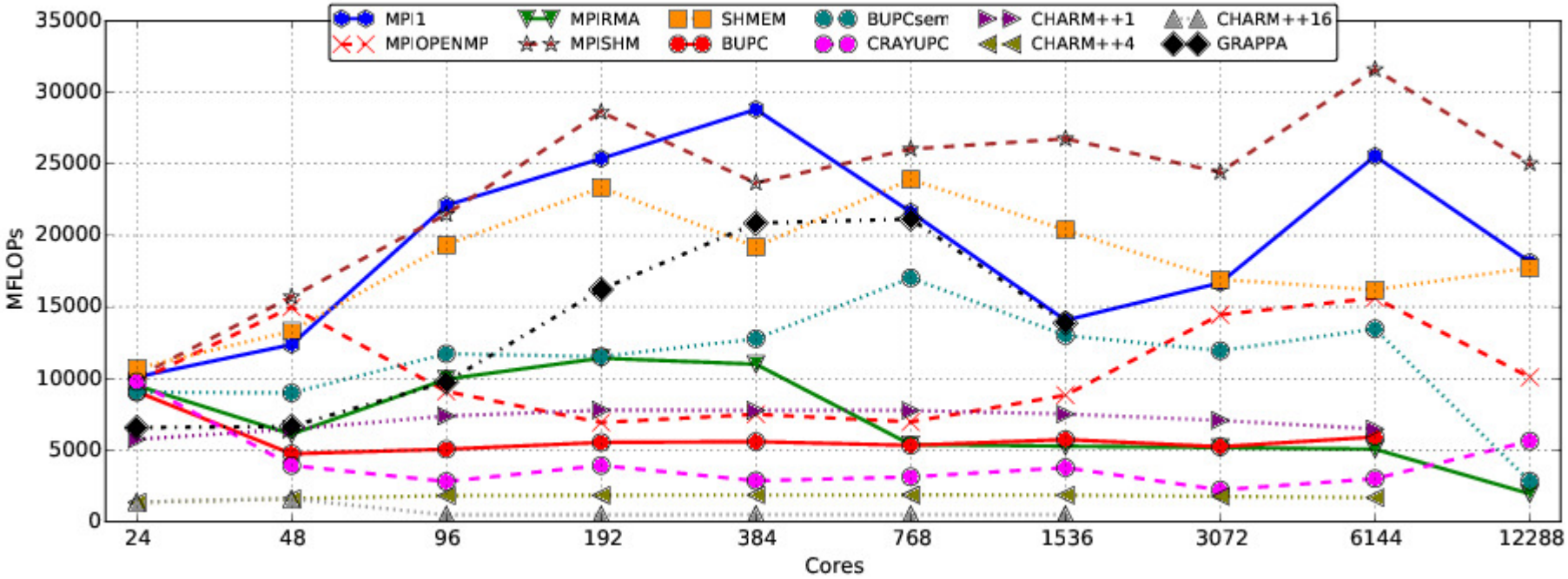
* Charm++: (47104x47104)

Stencil, strong scaled (49152x49152*)



Normalized performance (Mflops/#nodes)/Mflops_single_node_MPI1

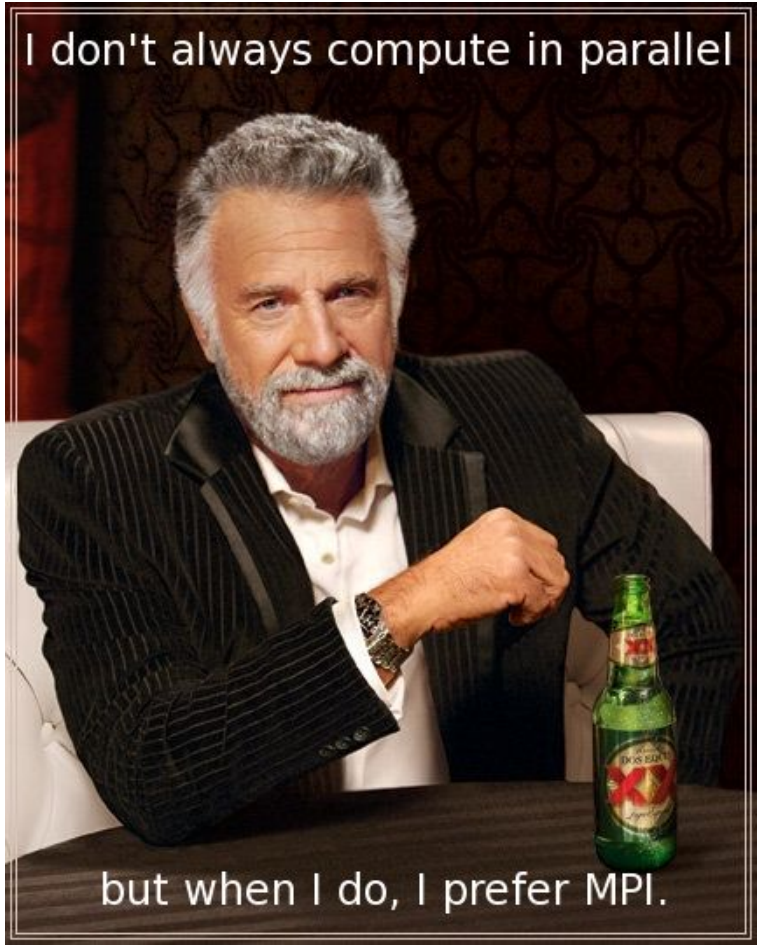
Synch_p2p, strong scaled (49152x49152*)



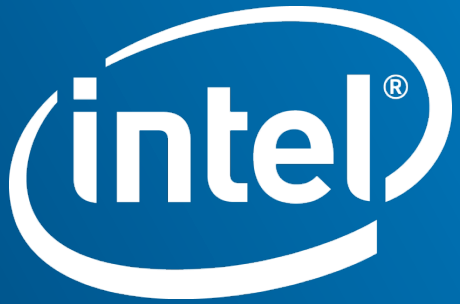
Aggregate performance MFlops

Conclusions

- Private data is the right default, both for applications and for system software.
- Good OpenMP looks like MPI:
  - Fork threads once.
  - Very little data sharing.
- MPI+OpenMP usually entails bad OpenMP, especially when threaded libraries are involved.
- Good MPI+OpenMP is MPI+MPI.



Jonathan Goldsmith, Dos Equis and Heineken International had nothing to do with this slide.


intel®

MPI-3 RMA

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*.

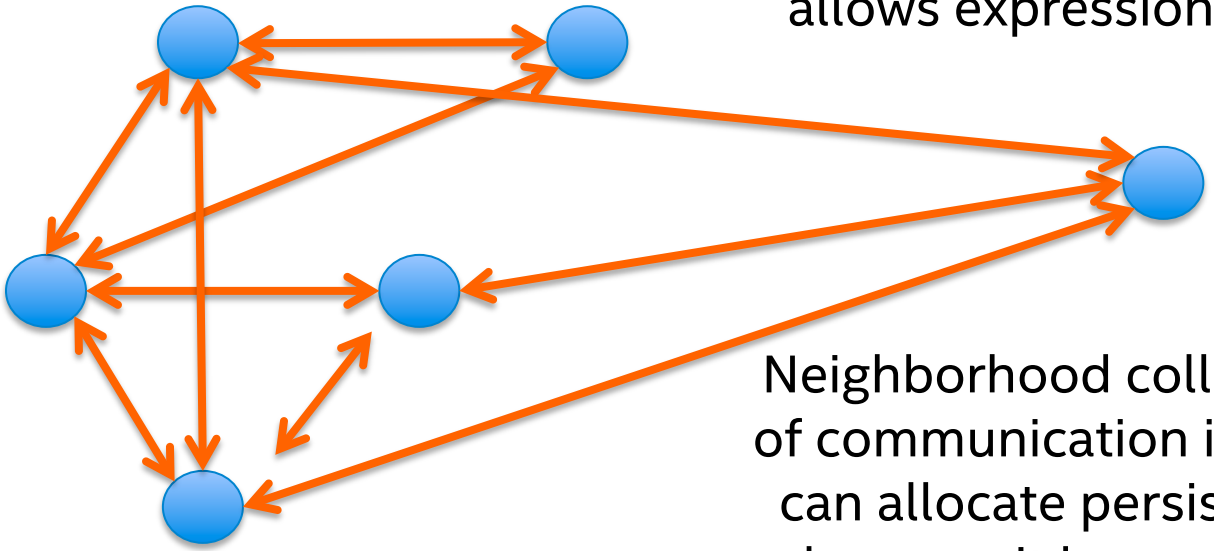
* This is Bill Gropp's current position. Fortran needs a memory model.

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 */
}
```


Here MPI is just a portable wrapper around shared memory.

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

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