Combining Performance and Portability

Jeff Hammond (jhammond@anl.gov)

Argonne Leadership Computing Facility

https://wiki.alcf.anl.gov/parts/index.php/User:Jhammond

Outline

- Portable performance 101
- Superficial but necessary portability in MPI
- Communication abstractions
- Distributed data-structures and methods
- Challenges with MPI+Threads
- Globalization of tasks and data

Print

The Story of a Man Who Outsourced His Work to China so He Could Watch Cat Videos All Day

BUSINESS INSIDER By Megan Rose Dickey | Business Insider – Wed, Jan 16, 2013 8:57 AM EST

Email Recommend 25.5k Tweet Share +1 Print

The Story of a Postdoc Who Outsourced Her Programming to Libraries so She Could Do Science All Day

BUSINESS INSIDER By Megan Rose Dickey | Business Insider – Wed, Jan 16, 2013 8:57 AM EST

Portable performance 101

Software monoliths are huge barriers to performance and portability.

- There's probably a library for that.
- If not, write a library for that (and make it OSS).
- Trade performance for portability most of the time.
- Mitigate risk by encapsulating non-portable elements.

"The best performance improvement is the transition from the nonworking state to the working state." – John Osterhout

Portable MPI Communication

Portable MPI

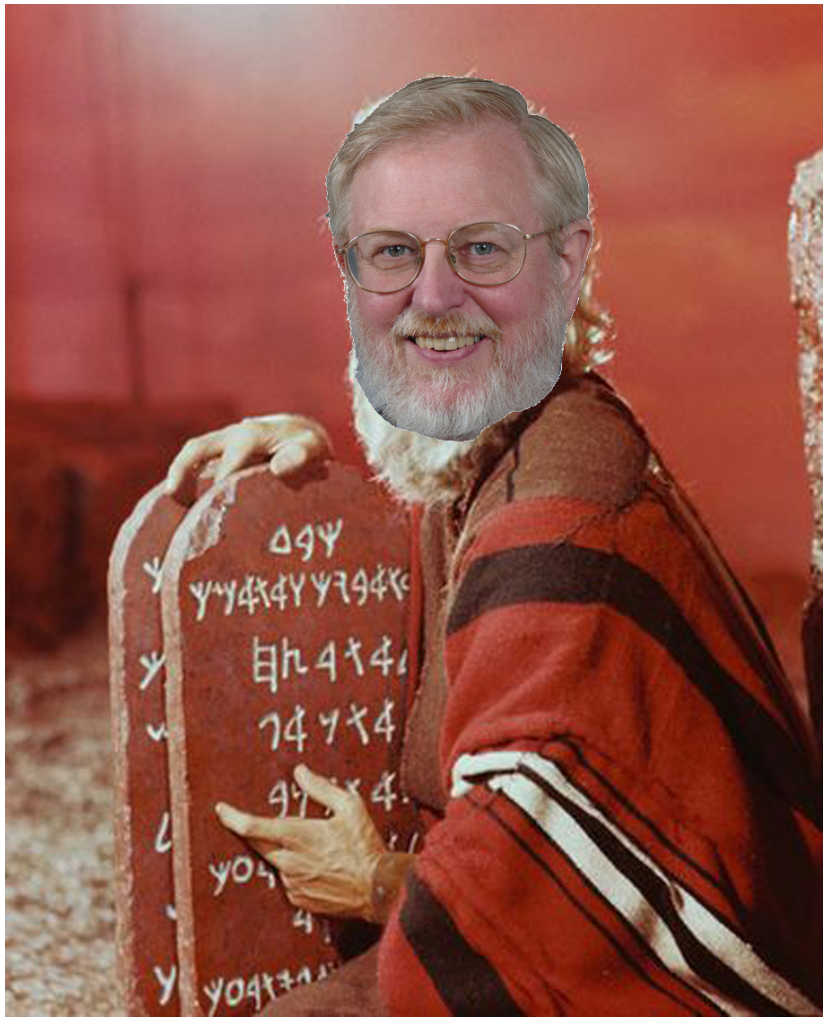
"But MPI *is* portable. WTF is portable MPI?!?!?"

- The MPI standard is perfect.
- Implementations are not perfect.
- Hardware is never ideal.

We have to deal with:

- Lack of latest features.
- Broken features.
- Performance quirks.
- Ambiguity in the standard.

Wrapping MPI costs cycles but has a huge payoff in many contexts.



Examples

```
#ifdef WORKAROUND_BGQ_BUG
   int junk[1];
   MPI_Isend(junk,1,MPI_INT,rank,tag,comm,req);
#else
   MPI_Isend(NULL,0,MPI_INT,rank,tag,comm,req);
#endif
```

Optimization of ssend created bug for count=0 on BGQ. It was fixed in a matter of days but GFMC folks don't like to lose days.

I only had to instantiate this workaround 27 times in their code...

Here we deal with features, syntax, performance and memory.

```
#if defined(__bg__)
# if defined(__bgq__)
  MPI_Barrier(...);   /* yeah... */
# endif
  MPI_Allreduce(...); /* faster than MPI_Reduce  */
  memcpy(...);        /* faster than MPI_Scatter */
#elif (MPI_VERSION > 3)
  MPI_Reduce_scatter_block(const void *sendbuf, ...);
#elif (MPI_VERSION == 2) && (MPI_SUBVERSION == 2)
  MPI_Reduce_scatter_block(void *sendbuf, ...);
#elif defined(AVOID_UNNECESSARY_VECTOR_ARGS)
  MPI_Reduce(...);  /* loss of fusion could hurt perf */
  MPI_Scatter(...); /* avoids vector arg */
#else
  MPI_Reduce_scatter(...); /* MPI_Scatterv-like args */
#endif
```

From https://github.com/elemental/Elemental/blob/master/src/core/imports/mpi.cpp:

```
template<typename R>
void Send( const R* buf, int count, int to, int tag,
           Comm comm )
{
    MpiMap<R> map;
    SafeMpi( MPI_Send( const_cast<R*>(buf), count,
                       map.type, to, tag, comm ) );
}
```

MpiMap is C++ magic for MPI type inference. Even before they were deleted, the MPI C++ bindings didn't do this.

Summary

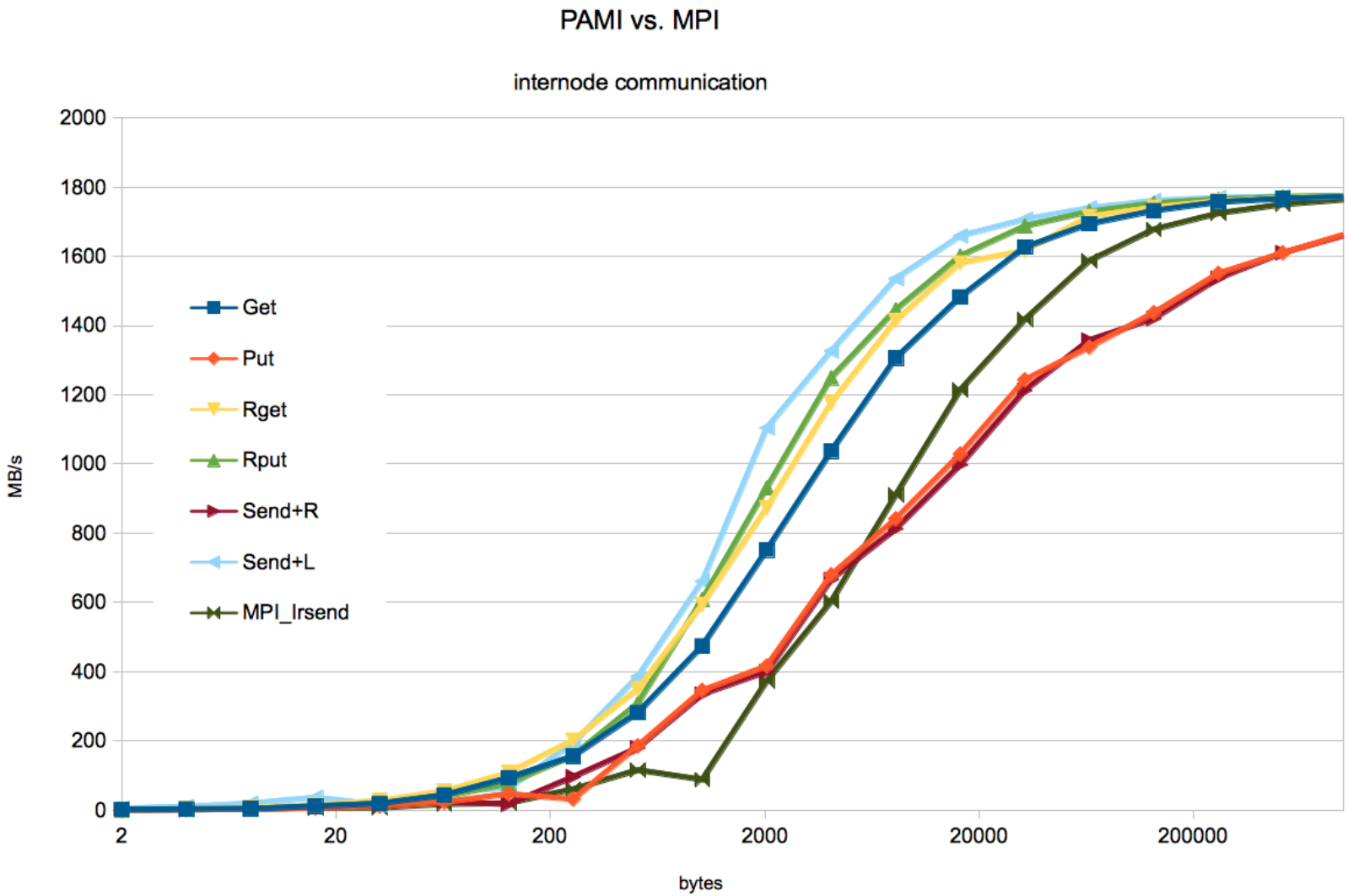
Wrapping MPI allows you to:

- Work around bugs and performance quirks.
- Deal with different MPI standards and implementations.
- Write your own language bindings (C++ and Fortran 200X are both compelling cases).
- Add your own performance instrumentation as $O(1)$ not $O(N)$ LOC.
- Parallel debugging, e.g. replace Send with Ssend to identify unsafe assumptions about buffering.

Beyond MPI

- I am not suggesting you stop using MPI!!!
- IBM (PAMI) and Cray (DMAPP) both provide non-MPI communication libraries that exploit their hardware in ways that MPI cannot or does not (in some case due to shortcomings in their own MPI libraries).
- In some cases, you can replace MPI calls with non-portable ones (inside of your communication wrappers, of course) and see better performance.
- For the most part, MPI-3 renders this unnecessary w.r.t. features since nonblocking collectives and remote atomics are now present.
- In the latency-sensitive regime, software overhead matters and non-portable APIs can lead to a significant speedup.

MPI vs. PAMI on Blue Gene/Q



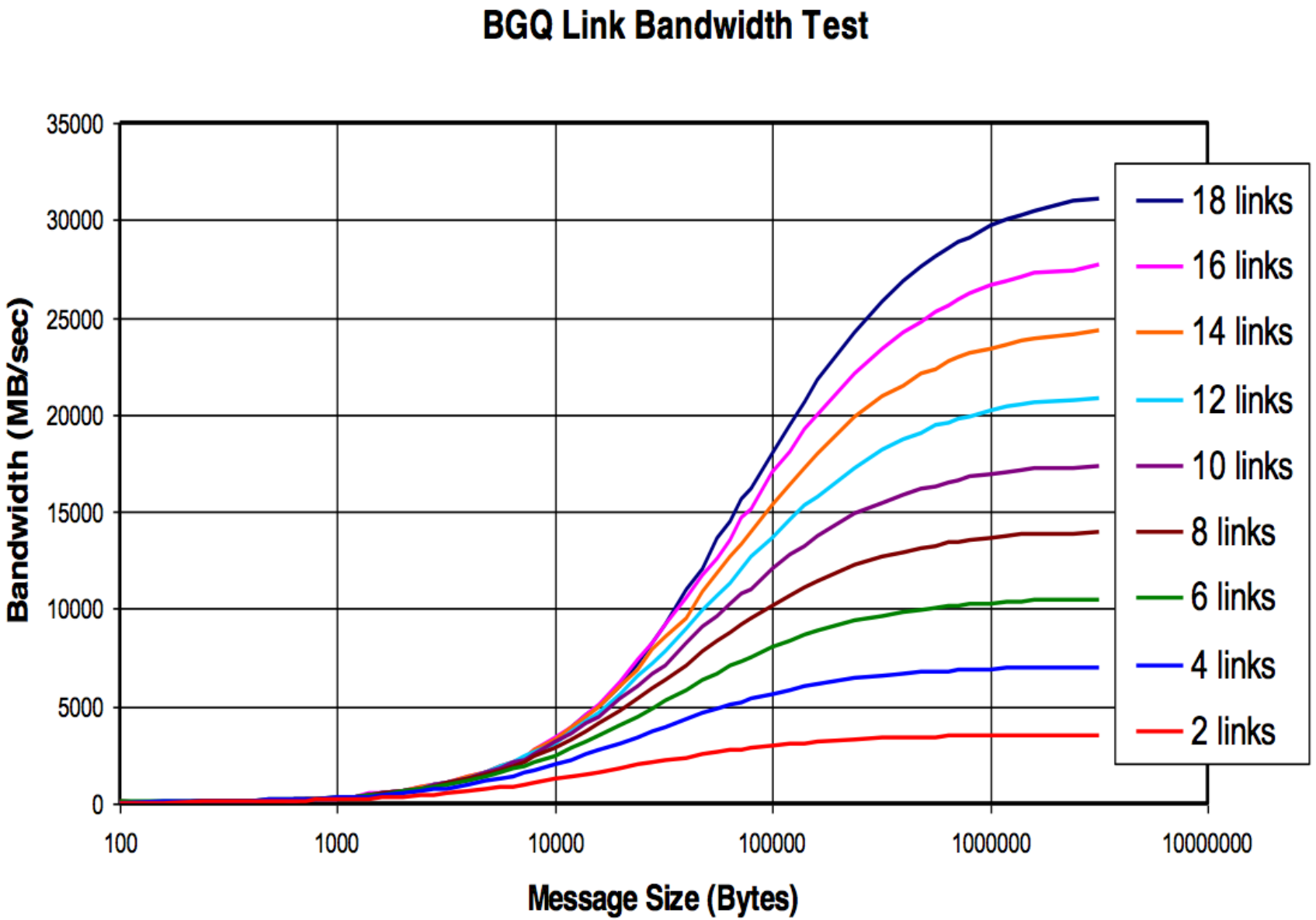
What not to do

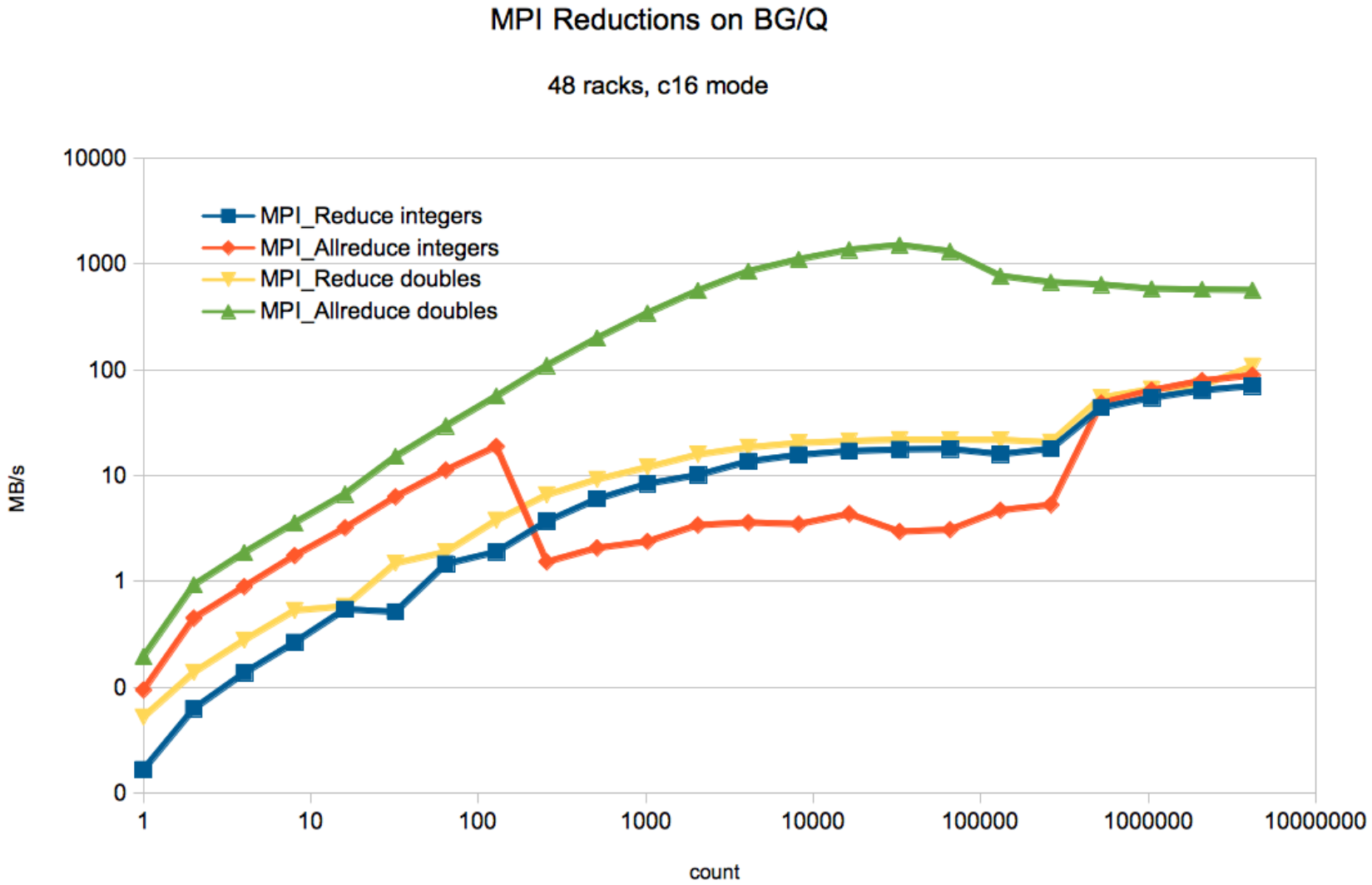
From http://www.sns.ias.edu/~adler/talks/memo.txt:

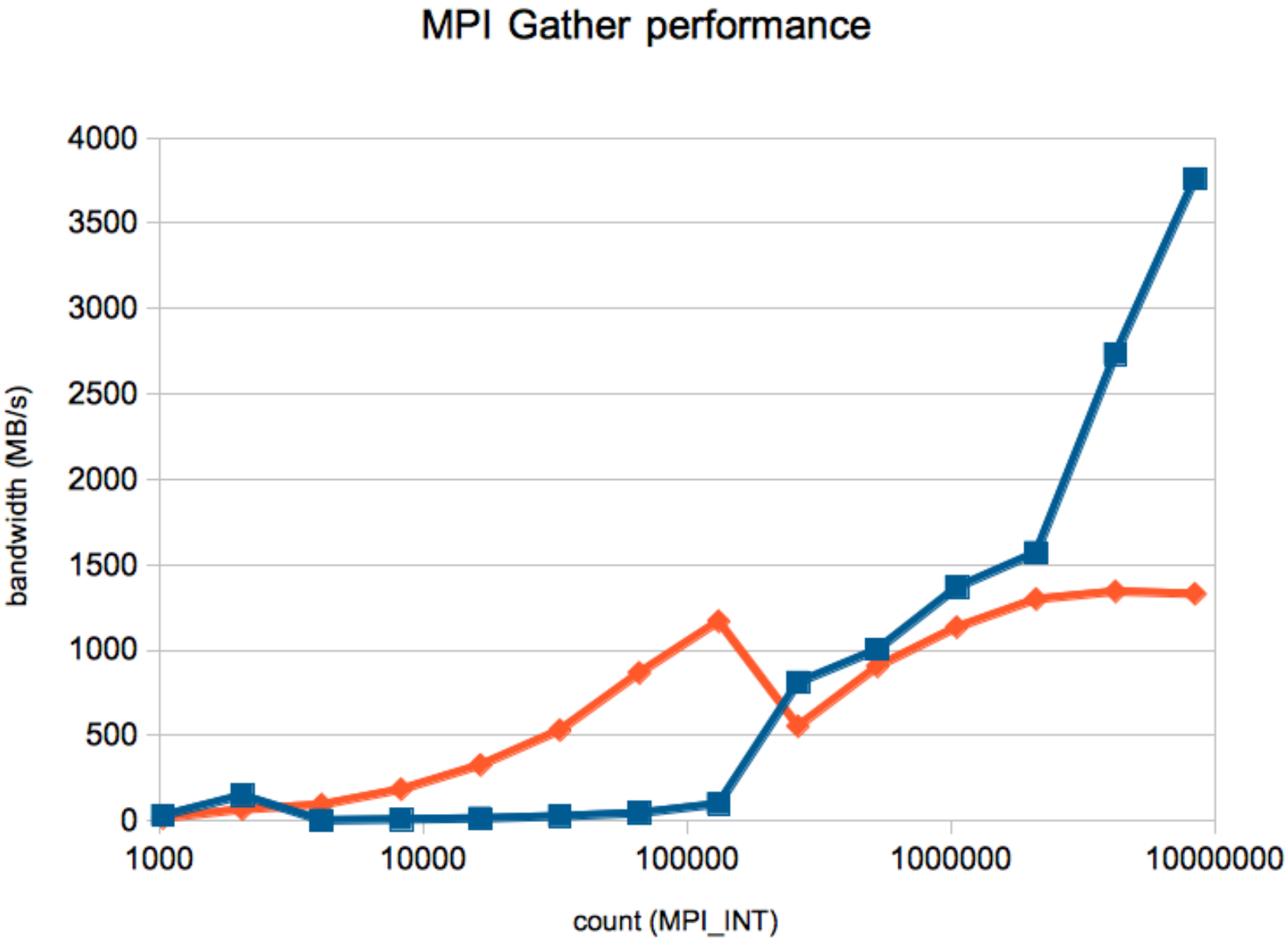
(1) If your program uses subroutines, put all MPI statements (MPI_Send, MPI_Recv, etc.) in your main program, not in the subroutines. You can always do this by transferring information from the subroutines to the main program through the subroutine arguments.

Performance Characteristics of MPI

Thanks to Bob Walkup at IBM for the first slide.


BGQ Link Bandwidth Test
Bandwidth (MB/sec)
35000
30000
25000
20000
15000
10000
5000
0
100
1000
10000
100000
1000000
10000000
Message Size (Bytes)
18 links
16 links
14 links
12 links
10 links
8 links
6 links
4 links
2 links


MPI Reductions on BG/Q
48 racks, c16 mode
MPI_Reduce integers
MPI_Allreduce integers
MPI_Reduce doubles
MPI_Allreduce doubles
MB/s
10000
1000
100
10
1
0
0
1
10
100
1000
10000
100000
1000000
10000000
count



(Total count i.e. count argument times nproc)

MPI Performance Artifacts

- Not all networks saturate at the same rate.
- Topology effects are huge. (Google for papers)
- You get what you pay for w.r.t. optimizations.
- Unexpected type-dependent performance.
- Protocol cutoffs have dramatic effects on performance.
- Many supercomputers benefit from barrier before other collectives; implementation degradation from non-ideal usage can be 10-1000x.

Deadlock as a canary for protocol effects

See ./code/deadlock.c.

```
#define MCW MPI_COMM_WORLD
{
  int dst = (rank+1)%size;
  int src = (rank-1)%size;
#if DEADLOCK
  MPI_Send(s, n, MPI_INT, dst, 0, MCW);
  MPI_Recv(r, n, MPI_INT, src, 0, MCW, MPI_STATUS_IGNORE);
#else
  MPI_Request req[2];
  MPI_Isend(s, n, MPI_INT, dst, 0, MCW, &(req[0]) );
  MPI_Irecv(r, n, MPI_INT, src, 0, MCW, &(req[1]) );
  MPI_Waitall(2, req, MPI_STATUSES_IGNORE);
#endif
}
```

Deadlock Exercise

How does the behavior of the deadlock version of the program change with (1) process count and (2) message size?

Example invocation:

```
./deadlock.x
mpiexec -n 1 ./deadlock.x 1000
mpiexec -n 2 ./deadlock.x 1000000
```

Nobs

A good MPI implementation will expose nobs for tuning.

- Lots of assumptions go into the defaults:
  (1) vendor wants to get paid so acceptance tests have to pass with defaults;
  (2) MPI-1 usage is common in applications.
- Eager-rendezvous cutoff is all about space-time trade-offs.
- Be wary of flow-control effects on irregular applications.
- Asynchronous progress is usually disabled by default; see `https://wiki.alcf.anl.gov/parts/index.php/MPI#Performance_Considerations` for details.

Homework

- Run OSU or other OSS MPI benchmarks on different machines.
- Write your own halo-exchange simulator and see how many different cartesian dimensions. are required to saturate the total node bandwidth.
- `https://code.google.com/p/mpi-qoit/` (just look at collectives).
- `http://www.mcs.anl.gov/events/workshops/p2s2/2012/slides/Morozov-P2S2-MPI_benchmark.pdf` has additional examples.

Case Study: Elemental

Jack Poulson is the lead author and PI of Elemental.

Elemental Background

Home page:

`http://www.libelemental.org/`

Documentation:

`http://poulson.github.io/Elemental/`

There's lots of information about Elemental on the internet...

Porting Elemental to New Platforms

Blue Gene/P: Jack was working at ALCF and the design was in-flux. C++ compiler bugs and lack of MPI-2.2 were the only real issues (that I remember).

Mac: Jeff learns about "-framework Accelerate" and CMake. Port takes 5 minutes.

Blue Gene/Q: It took an entire day to port CMake, at which point Elemental worked immediately. In 2011.

Cray XC30: Dealt with CMake problem related to shared libraries, then Elemental worked immediately.

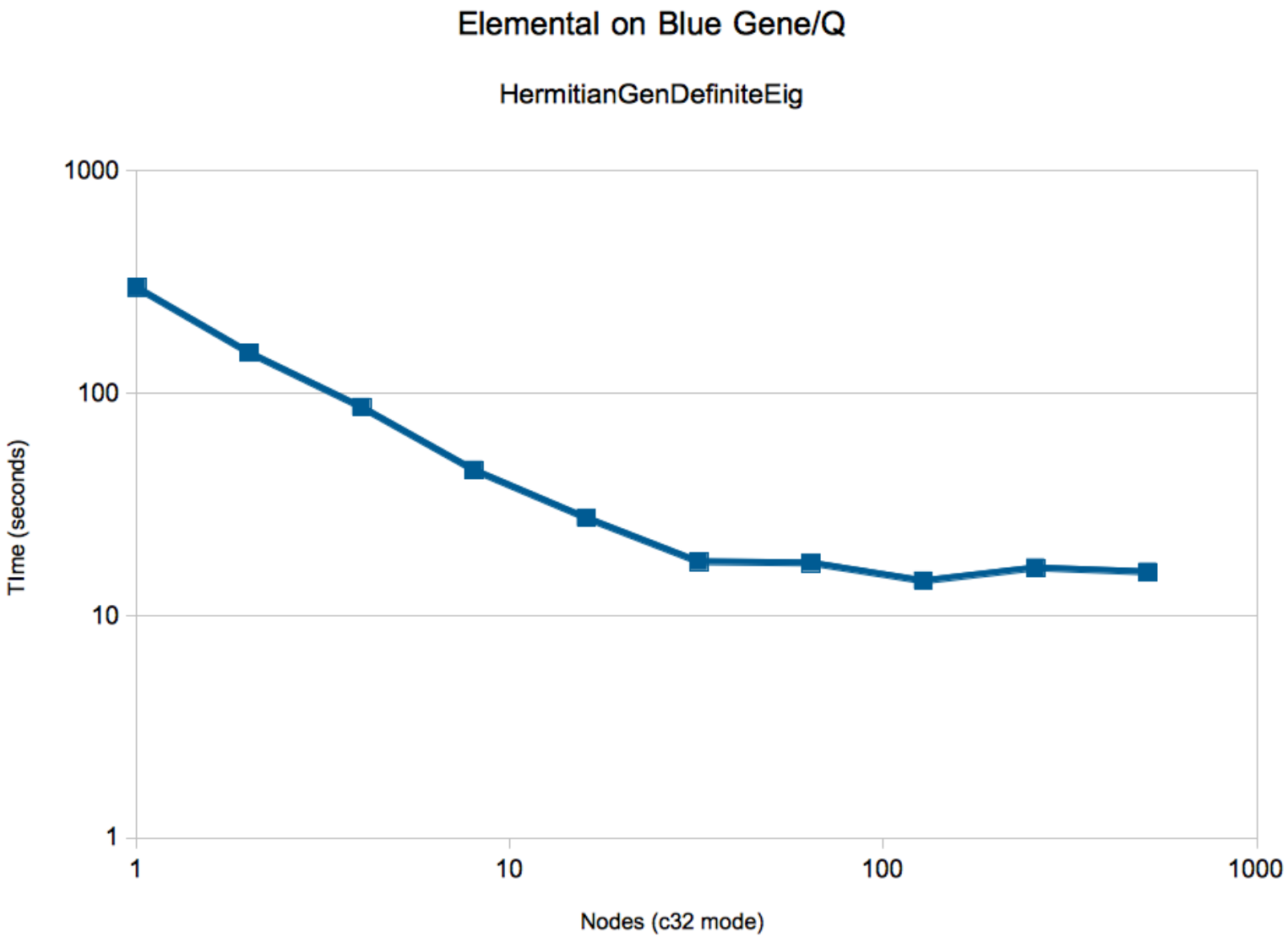
(immediately = rate-limited by login node environment.)

Why is Elemental so easy to port?

- Despite the annoyances, CMake captures machine-specific details effectively. For supercomputers, toolchain files have correct defaults.
- Restrained use of C++ templates (they are statically instantiated). No Fortran. No premature optimization.
- BLAS, LAPACK and MPI are all wrapped.
- Handling all known MPI portability issues inside of the wrapper once-and-for-all.

Robust build systems and conscientious developers are critical if you want portability in HPC. If you build a good library, good people will line up to help you.

Elemental on Blue Gene/Q



(rank 10,000 matrix)

Portable performance of Elemental

- Elemental uses the best known algorithms just like ScaLAPACK. You should not underestimate the effort that goes into this. Algorithms trump software.
- Essentially all of the flops happen in BLAS, which is tuned by someone else.
- Using MPI collectives whenever possible. This is very BG-friendly but generally punts the communication problem to someone else.
- Exploiting subcommunicators in a topology-*friendly* way.
- Tuning parameters are runtime options.

Programmer productivity

```
/* generic setup */
const int blocksize = 128;
SetBlocksize(blocksize);
Grid G(MPI_COMM_WORLD);

/* problem-specific data */
DistMatrix<T>  A( n, n, G ), B( n, n, G );
DistMatrix<double> X( n, n, G ); // eigenvectors
DistMatrix<double,VR,STAR> w( n, n, G); // eigenvalues

/* solve problem */
HermitianGenDefiniteEigType eigType = AXBX; // Ax=wBx
UpperOrLower uplo = CharToUpperOrLower('U');
HermitianGenDefiniteEig( eigType, uplo, A, B, w, X );
```

Case Study: NWChem

NWChem Background

Home page: `http://www.nwchem-sw.org/`

- Began at the dawn of the MPP age, before MPI.
- Attempted to reuse existing code; most of this was a waste of time.
- Designed to be object-oriented but constrained by Fortran 77 (i.e. non-viability of C++ at the time).
- Global Arrays programming model abstracted away explicit communication.
- Most of the gross bits of non-portability (outside of ARMCI) live in `src/util`.
- Uses its own memory allocator, IO layer, runtime database, hooks resource managers, low-level timers, etc.

Porting NWChem to New Platforms

Some of this is historical...

1. Port ARMCI to low-level network interface. This is **hard**.
2. Workaround Fortran compiler bugs; detect them in classic Make build system.
3. Example: XLF doesn't preprocess the normal way and the workaround is nasty.
4. Workaround operating system quirks (lots of Unixen prior to Linux era).
5. All sorts of Fortran integer crap that shortens my life to even think about.

Summary: The only thing that is still hard about porting NWChem is ARMCI, but more on that later...

NWChem Portability

Supported: Cray SV1, YMP, X1, XT/XE/XK/XC;
Intel Delta; KSR; NEC, Fujitsu;
Linux; Unix; Windows; Cygwin; Mac;
x86, PPC, Itanium, etc.;
Ethernet; Myrinet; Infiniband, etc.;
IBM POWER, Blue Gene/L, P, Q;
NVIDIA GPGPU (partial).

Unsupported: SiCortex; iPhone.

Summary: If NWChem doesn't run on it, there's a 50-50 change you'll go out of business :-)

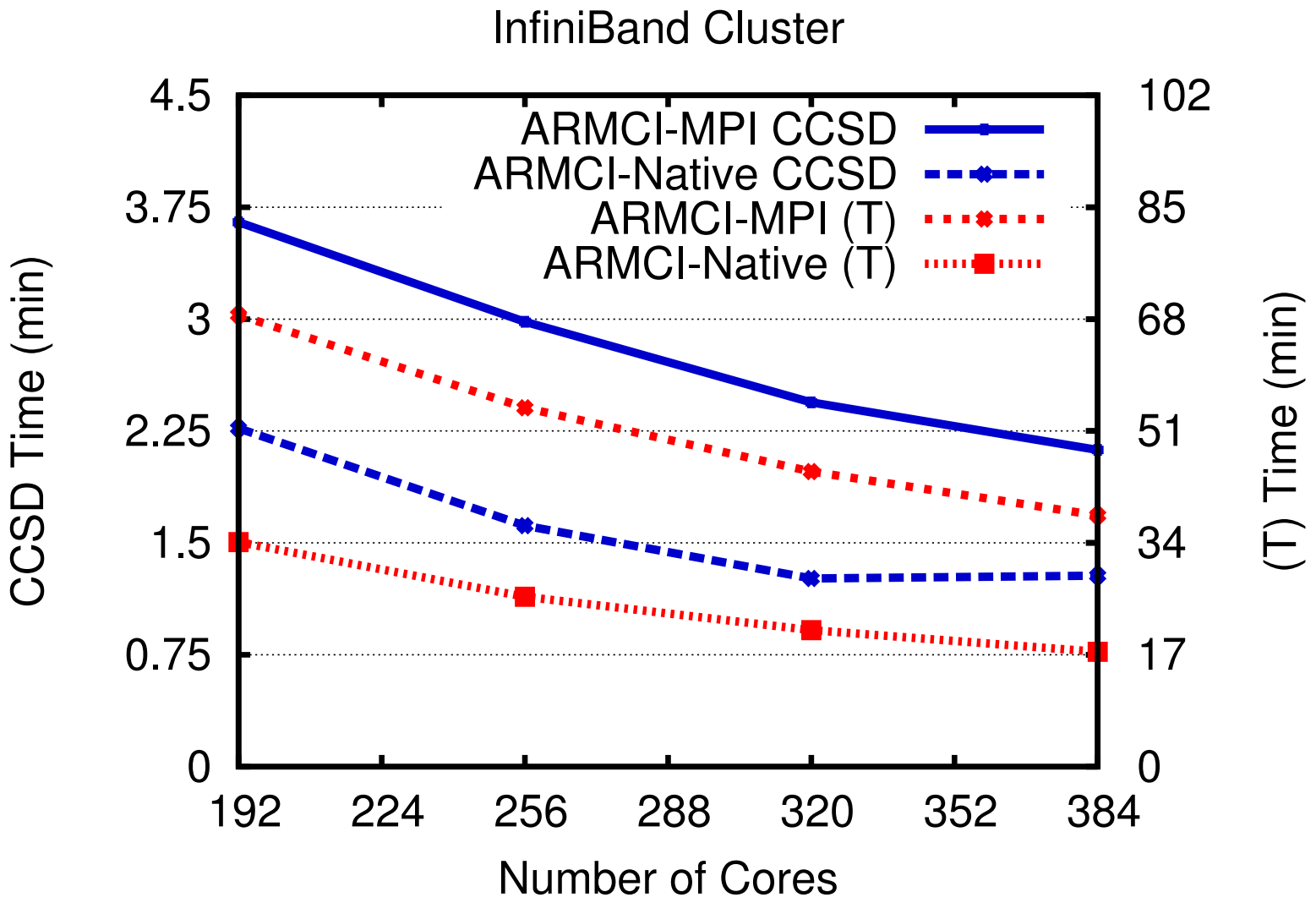
Solving the ARMCI problem

Attempts at portability:

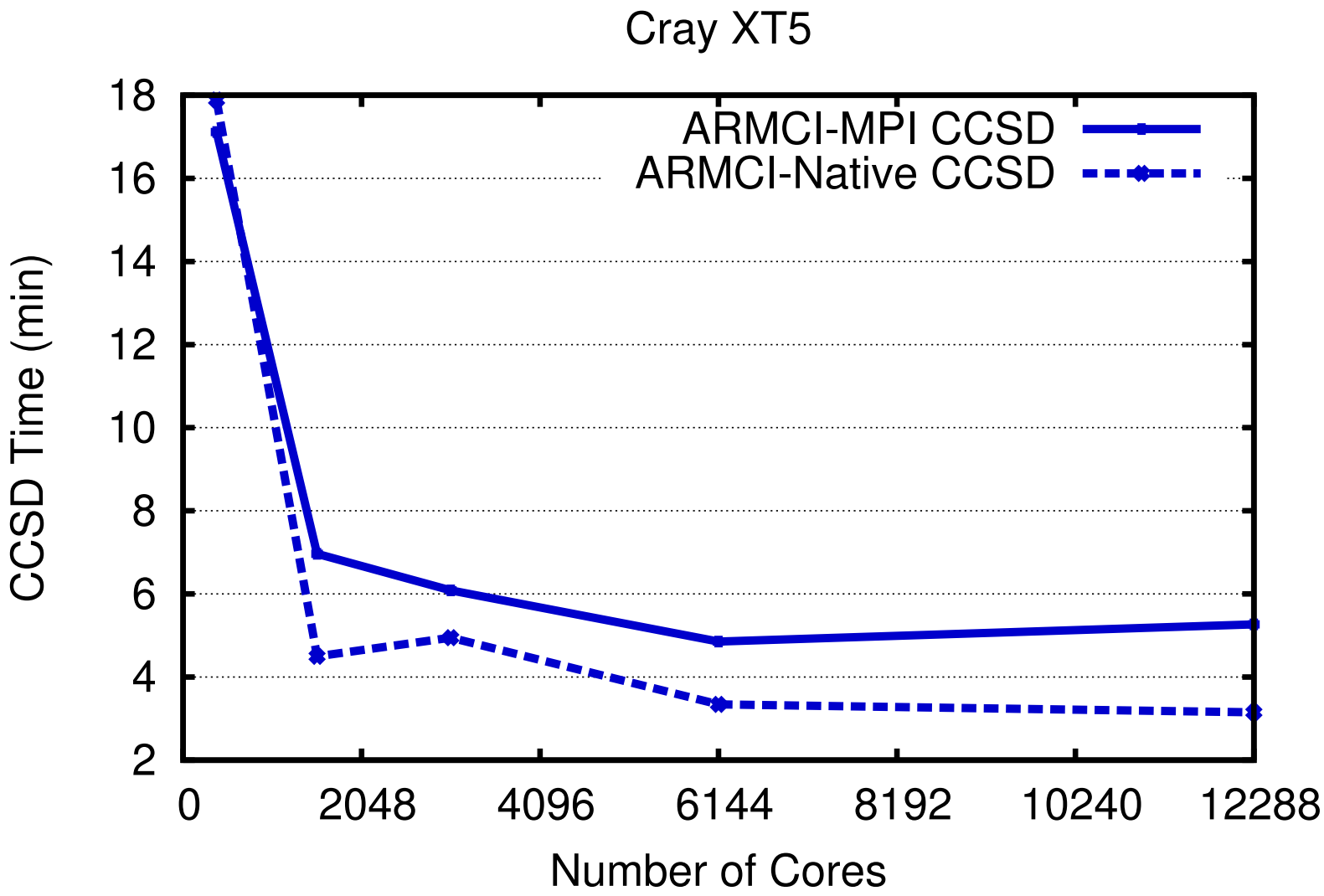
- TCP/IP performs poorly and isn't available on some supercomputers.
- Cray-oriented MPI Send+Spawn implementation of ARMCI.
- Cluster-oriented MPI Send+Threads implementation of ARMCI.
- ARMCI-MPI (from Argonne) is the first implementation using MPI one-sided.

ARMCI-MPI is fundamentally limited by the underlying RMA implementation. Historically, these have been lacking. Also, MPI-2 lacks atomics and other essential features.

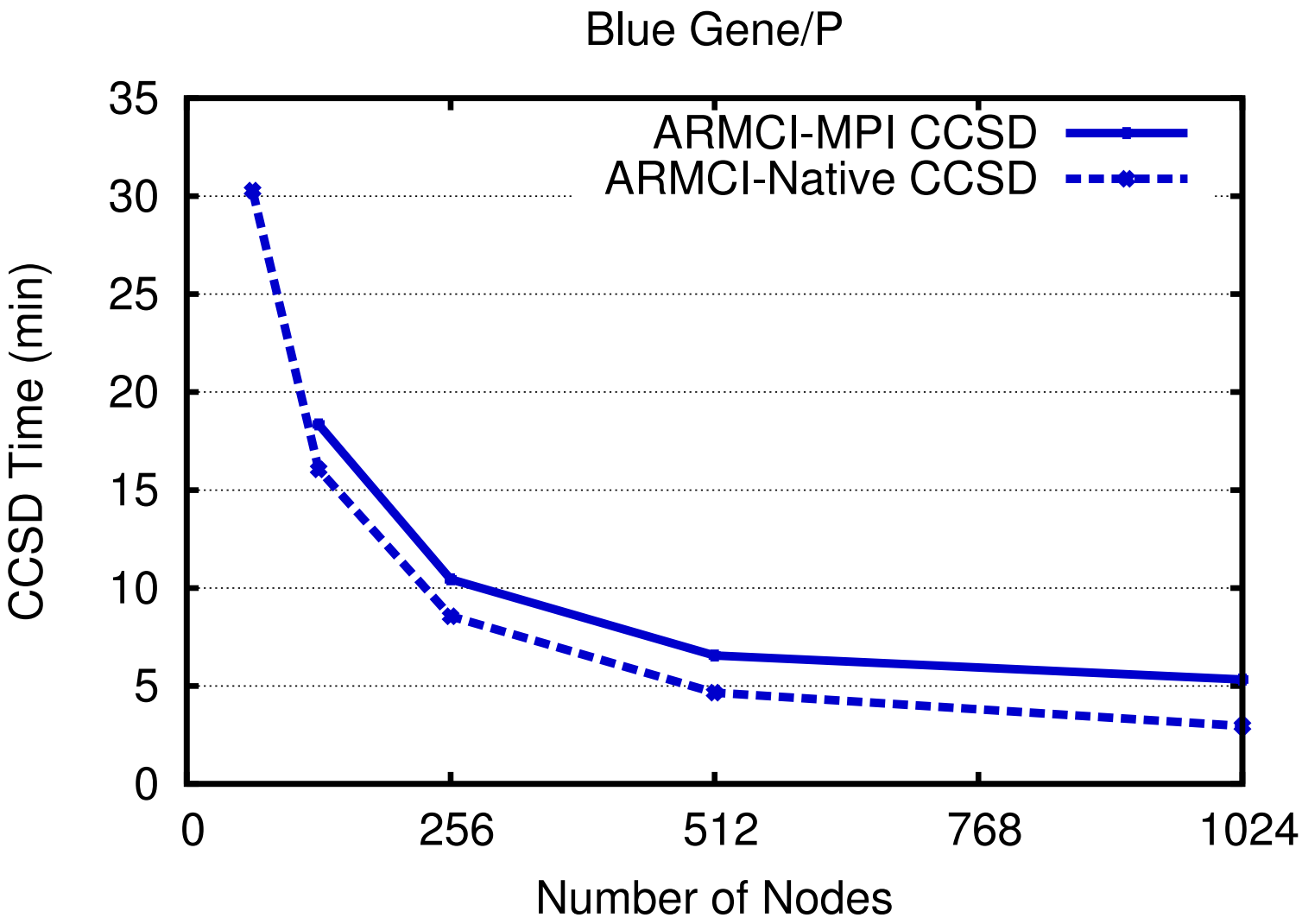
NWChem with ARMCI-MPI



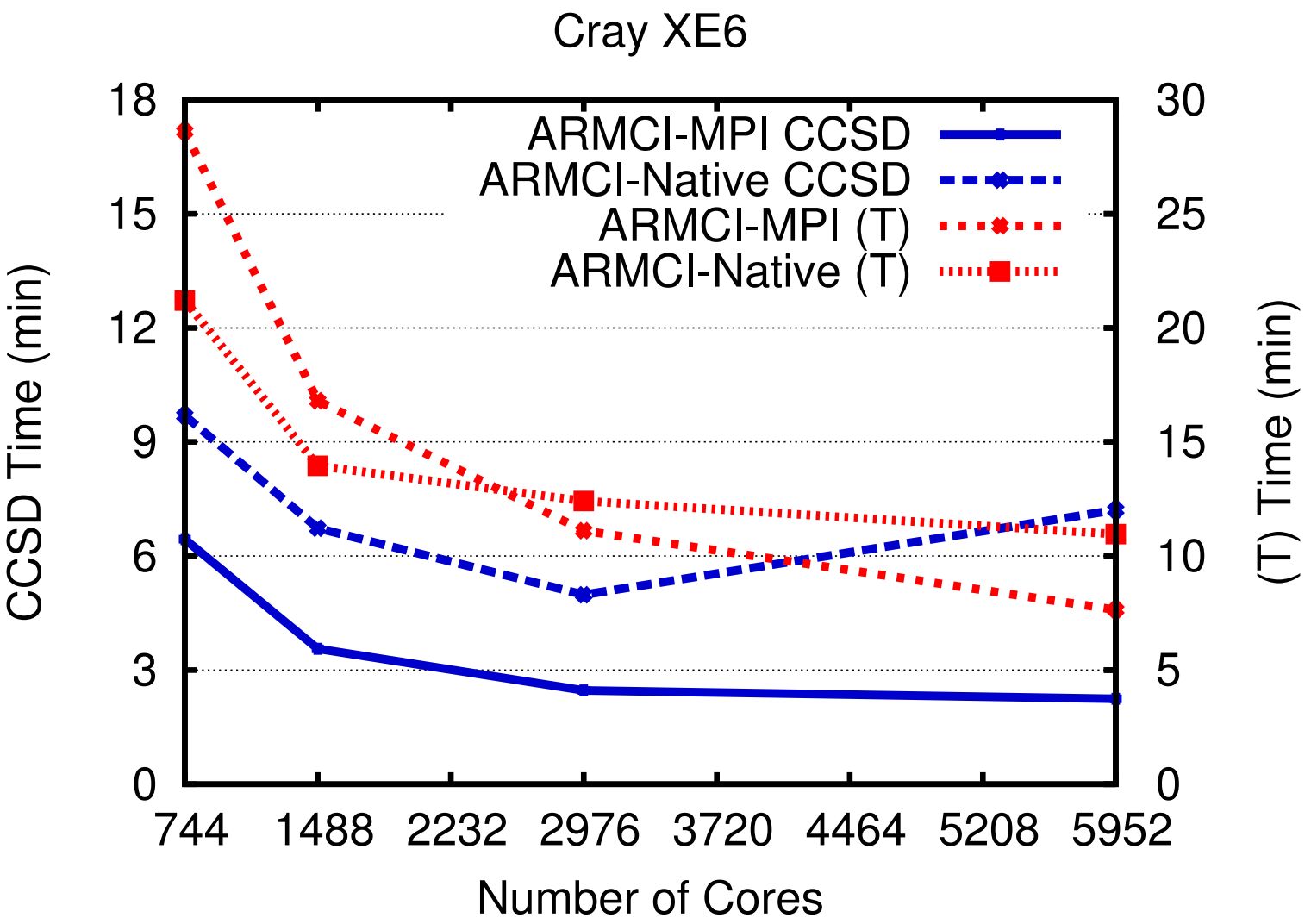
NWChem with ARMCI-MPI



NWChem with ARMCI-MPI



NWChem with ARMCI-MPI



Porting NWChem to Blue Gene/Q

What a difference MPI makes...

1. Global Arrays ran immediately with ARMCI-MPI at SC11.
2. NWChem trunk was ported sans ESSL, PeIGS and ScaLAPACK in 3 hours at workshop.
3. Voodoo bug in C kernel fixed by restoring original Fortran.

Remaining issues:

- NWChem is frequently memory-limited ∴ need fewer ppn.
- GA/ARMCI is not thread-safe ∴ fork-join threading.
- F77 common blocks are not thread-safe ∴ careful OpenMP.
- MPI RMA performance is lacking; working on ARMCI-PAMI.

Optimizing NWChem for modern architectures

- Quantum chemistry is flop-rich; offload to GPU/MIC is mostly fine.
- Critical to exploit vectorization and fine-grain parallelism.
- DGEMM and loop-heavy CCSD(T) now runs better in hybrid mode.
- MIC kernels are scaling to more than 100 threads.
- GPU vs. MIC? Rewrite in CUDA vs. refactor and add pragmas...

Homework

- Google or measure offload bandwidth and DGEMM flop-rate; determine for what matrix size offload is worthwhile.
- Repeat first exercise for something more interesting (e.g. your code).
- Compare OpenMP scaling of simple loop kernels on Blue Gene vs. Intel and AMD, particularly multi-socket nodes (i.e. heavy NUMA).
- Write a simple vectorizable kernel in F77, F95 (with colon and/or array notation), and C/C++; does the compiler auto-vectorize for you? When?

Challenges with MPI+X

The future is MPI+X

- MPI+OpenMP is too often fork-join.
- Pthreads scare people; can't be used from Fortran (easily).
- TBB and Cilk come from Intel (FYI: TBB now runs on BGQ).
- OpenCL is an eye chart and has no abstraction for performance variability.
- CUDA is an X for only one type of hardware (ignoring Ocelot).

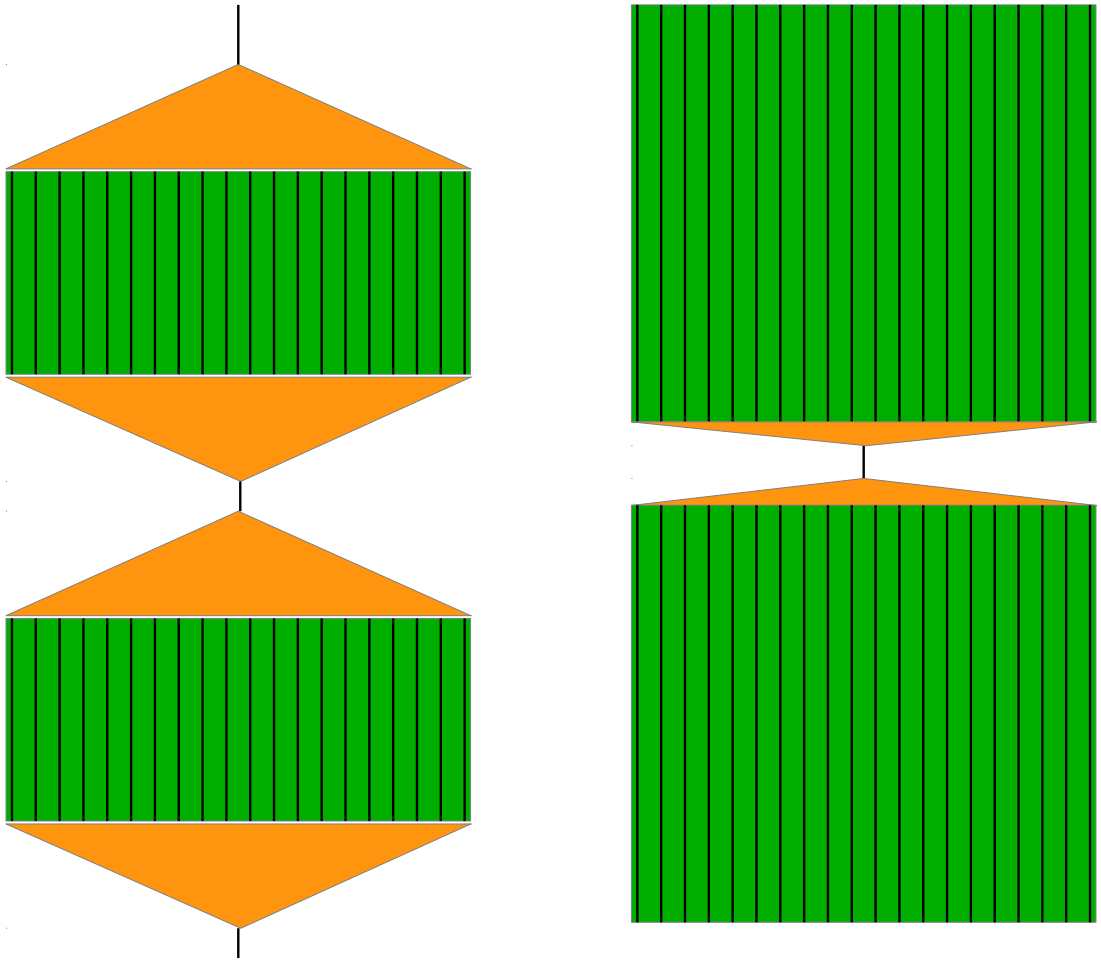
Never confuse portability with portable performance!

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

See `./src/omp/numa.c`

```
> 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 so you will not see this on your laptop. Run on an AMD Magny Cours for the "best" effect.

For more complicated data access patterns, you may see this even with parallel initialization. In this case, consider (1) hiding latency, (2) not being bandwidth bound, and (3) task parallelism.

MPI+Y

- If you use OpenMP libraries built with multiple compilers, you may get multiple thread pools.
- OpenMP, TBB, etc. all use Pthreads. So do many apps and libraries. Oversubscribe much?
- `MPI_THREAD_MULTIPLE` adds overhead; some apps use their own mutex but internal mutexes are invisible to other MPI clients.

The stark reality is that general MPI+Y – i.e. MPI+X for X≠OpenMP – is heavily dependent upon an MPI implementation that is designed to be used in a truly multithreaded way. Today, only Blue Gene/Q as this.

Based on `https://www.ieeetcsc.org/activities/blog/challenges_for_interoperability_of_runtime_systems_in_scientific_applications`

Acknowledgments

ALCF, Pavan Balaji, Jim Dinan, Robert Harrison, Karol Kowalski, Jack Poulson, Robert van de Geijn, and many others.