Challenges in Programming the Next Generation of HPC Systems

William Gropp

wgropp.cs.illinois.edu

Department of Computer Science
and
National Center for Supercomputing Applications
University of Illinois at Urbana-Champaign

NCSA

Towards Exascale Architectures

Next Generation Systems in China: All Heterogeneous Increasing diversity in accelerator choices

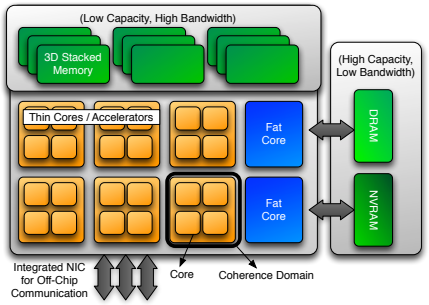
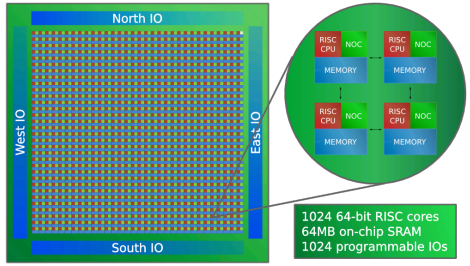
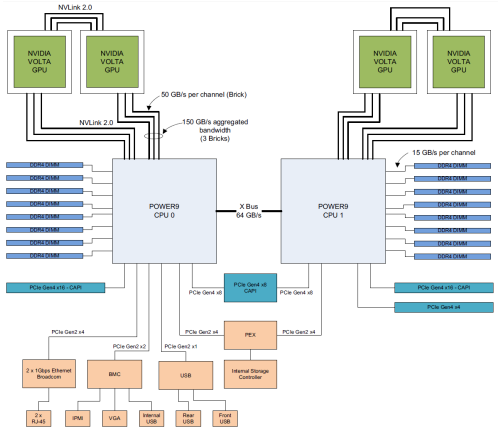


Figure 2.1: Abstract Machine Model of an exascale Node Architecture





From “Abstract Machine Models and Proxy Architectures for Exascale Computing Rev 1.1,” J Ang et al

Adapteva Epiphany-V

- 1024 RISC processors
- 32x32 mesh
- Very high power efficiency (70GF/W)

DOE Sierra

- Power 9 with 4 NVIDA Volta GPU
- 4320 nodes

NCSA Deep Learning System
16 nodes of Power 9 with 4 NVIDIA Volta GPU + FPGA

Where are the real problems in using HPC Systems?

- HPC Focus is typically on scale
  - “How will we program a million (or a billion) cores?
  - “What can use use to program these machines?”
- This talk focuses on some of the overlooked issues
  - Performance models still (mostly) process to process and single core
    - Node bottlenecks missed; impacts design from hardware to algorithms
  - Dream of “Performance Portability” stands in the way of practical solutions to “transportable” performance
  - HPC I/O requirements impede performance, hurt reliability
- This talk does *not* talk about the need for different algorithms for different architectures – there is no magic fix
  - But some ideas and approaches here can help

Programming Models and Systems

- In past, often a tight connection between the execution model and the programming approach
  - Fortran: FORmula TRANslation to von Neumann machine
  - C: e.g., “register”, ++ operator match PDP-11 capabilities, needs
- Over time, execution models and reality changed but programming models rarely reflected those changes
  - Rely on compiler to “hide” those changes from the user – e.g., auto-vectorization for SSE(n)
- Consequence: Mismatch between users’ expectation and system abilities.
  - Can’t fully exploit system because user’s mental model of execution does not match real hardware
  - Decades of compiler research have shown this problem is extremely hard – can’t expect system to do everything for you.

The Easy Part – Internode communication

- Often focus on the “scale” in Exascale as the hard part
  - How to deal with a million or a billion processes?
  - But really not too hard
    - Many applications have large regions of regular parallelism
  - Or nearly impossible
    - If there isn’t enough independent parallelism
  - Challenge is in handling definition and operation on distributed data structures
  - Many solutions for the internode programming piece
  - The dominant one in technical computing is the Message Passing Interface (MPI)

Modern MPI

- MPI is much more than message passing
  - I prefer to call MPI a programming *system* rather than a programming *model*
    - Because it implements several programming *models*
- Major features of MPI include
  - Rich message passing, with nonblocking, thread safe, and persistent versions
  - Rich collective communication methods
  - Full-featured one-sided operations
    - Many new capabilities over MPI-2
    - Include remote atomic update
  - Portable access to shared memory on nodes
    - Process-based alternative to sharing via threads
    - (Relatively) precise semantics
  - Effective parallel I/O that is not restricted by POSIX semantics
    - But see implementation issues …
  - Perhaps most important
    - Designed to support “programming in the large” – creation of libraries and tools
- MPI continues to evolve – MPI “next” Draft released at SC in Dallas last November

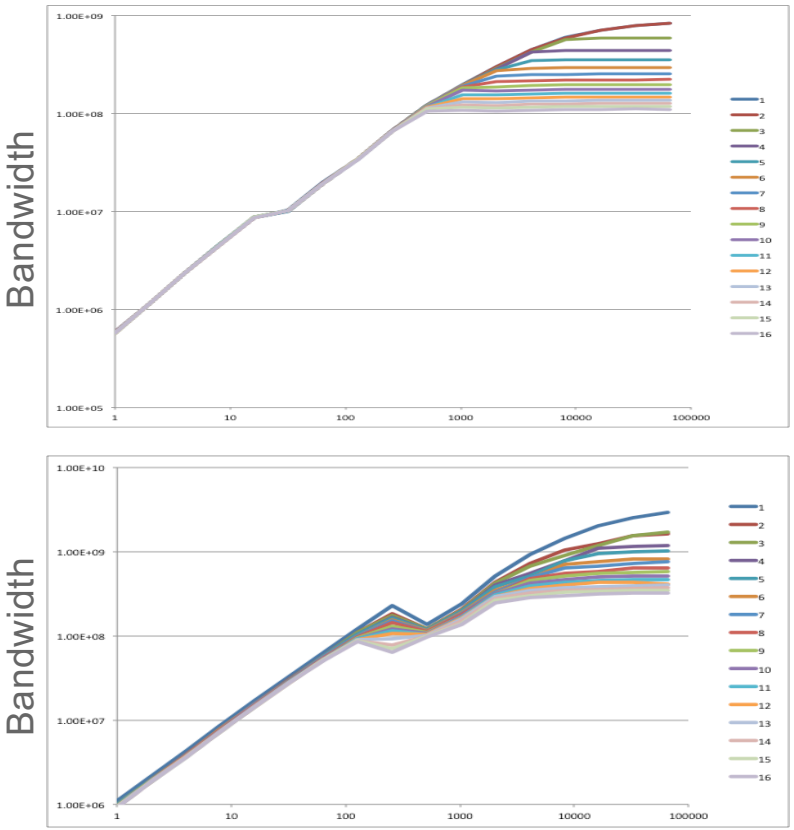
Applications Still Mostly MPI-Everywhere

- "the larger jobs (> 4096 nodes) mostly use message passing with no threading." – Blue Waters Workload study, https://arxiv.org/ftp/arxiv/papers/1703/1703.00924.pdf
- Benefit of programmer-managed locality
  - Memory performance nearly stagnant (will HBM save us?)
  - Parallelism for performance implies locality must be managed effectively
- Benefit of a single programming system
  - Often stated as desirable but with little evidence
  - Common to mix Fortran, C, Python, etc.
  - But…Interface between systems must work well, and often don't
    - E.g., for MPI+OpenMP, who manages the cores and how is that negotiated?
    - Don't forget the "+" in "MPI + X"!

MPI On Multicore Nodes

- MPI Everywhere (single core/single thread MPI processes) still common
  - Easy to think about
  - We have good performance models (or do we?)
- In reality, there are issues
  - Memory per core declining
    - Need to avoid large regions for data copies, e.g., halo cells
    - MPI implementations could share internal table, data structures
      - May only be important for extreme scale systems
  - MPI Everywhere implicitly assume uniform communication cost model
    - Limits algorithms explored, communication optimizations used
- Even here, there is much to do for
  - Algorithm designers
  - Application implementers
  - MPI implementation developers
- One example: Can we use the single core performance model for MPI?

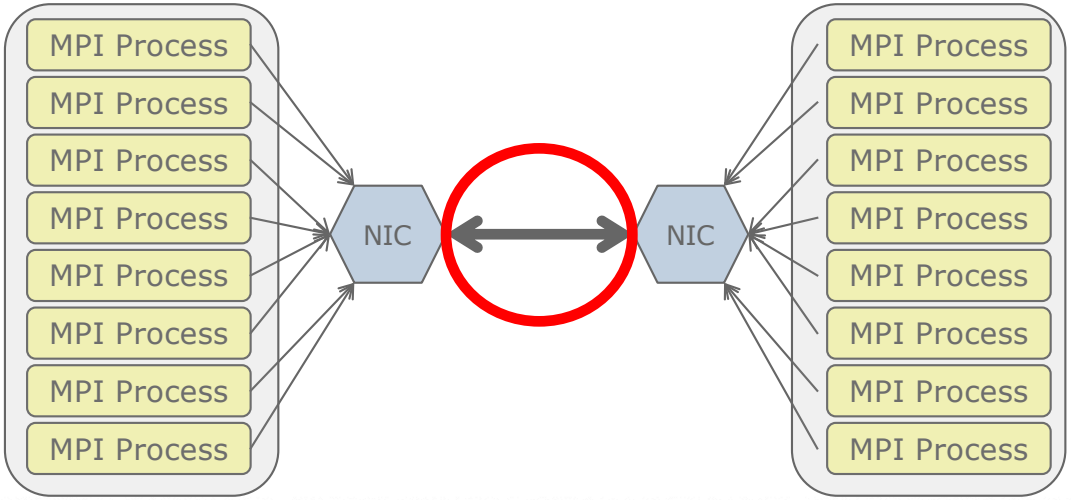
Rates Per MPI Process



- Ping-pong between 2 nodes using 1-16 cores on each node
- Top is BG/Q, bottom Cray XE6
- "Classic" model predicts a single curve – rates independent of the number of communicating processes

Why this Behavior?

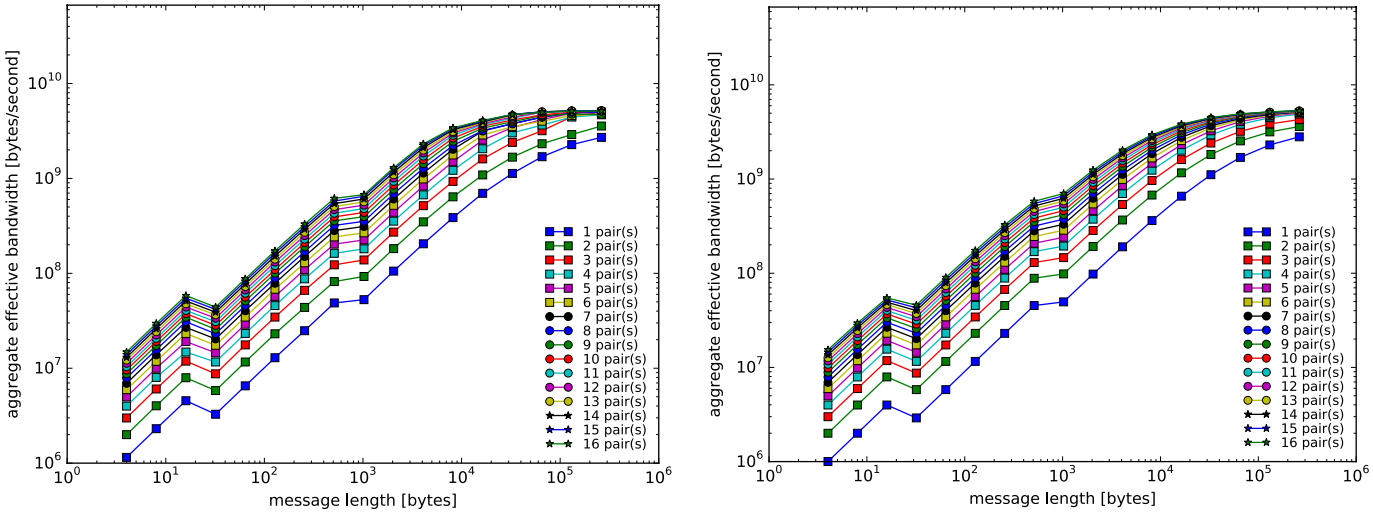
- The T = s + r n model predicts the *same* performance independent of the number of communicating processes
  - What is going on?
  - How should we model the time for communication?



A Slightly Better Model

- For k processes sending messages, the sustained rate is
  - $\min(R_{NIC\text{-}NIC}, k\ R_{CORE\text{-}NIC})$
- Thus
  - $T = s + k\ n/\min(R_{NIC\text{-}NIC}, k\ R_{CORE\text{-}NIC})$
- Note if $R_{NIC\text{-}NIC}$ is very large (very fast network), this reduces to
  - $T = s + k\ n/(k\ R_{CORE\text{-}NIC}) = s + n/R_{CORE\text{-}NIC}$
- This model is approximate; additional terms needed to capture effect of shared data paths in node, contention for shared resources
- But this new term is by far the dominant one

Comparison on Cray XE6



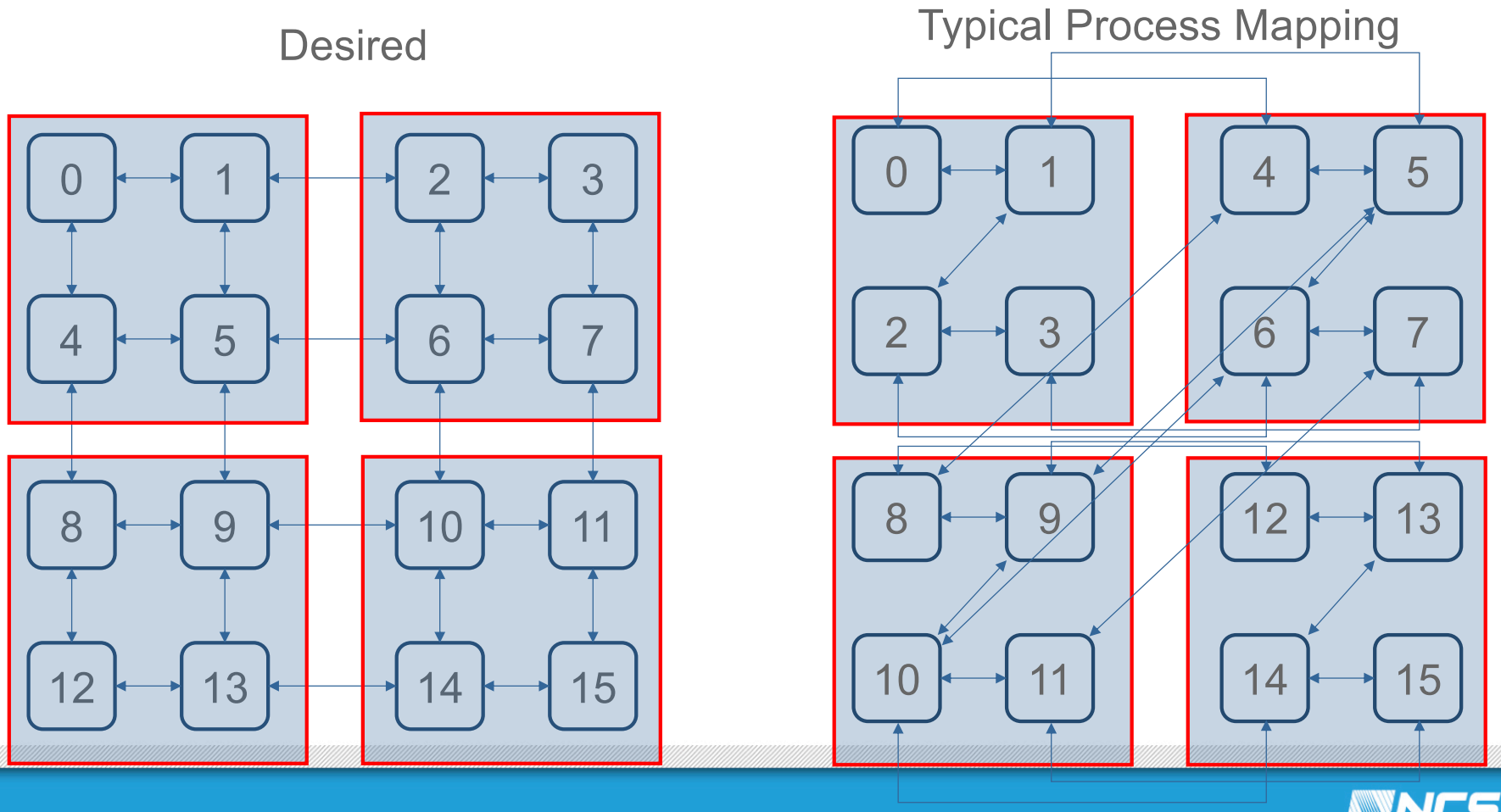
Measured Data Max-Rate Model

Modeling MPI Communication Performance on SMP Nodes: Is it Time to Retire the Ping Pong Test, W Gropp, L Olson, P Samfass, Proceedings of EuroMPI 16, https://doi.org/10.1145/2966884.2966919

MPI Virtual Process Topologies

- Lets user describe some common communication patterns
- Promises
  - Better performance (with “reorder” flag true)
  - Convenience in describing communication (at least with Cartesian process topologies)
- Reality
  - “Reorder” for performance rarely implemented
    - Few examples include NEC SX series and IBM BlueGene/L
  - Challenge to implement in general
    - Perfect mapping complex to achieve except in special cases
      - And perfect is only WRT the abstraction, not the real system
- Rarely used in benchmarks/applications, so does not perform well, so is rarely used in benchmarks/applications

Example Cartesian Process Mesh – Four Nodes

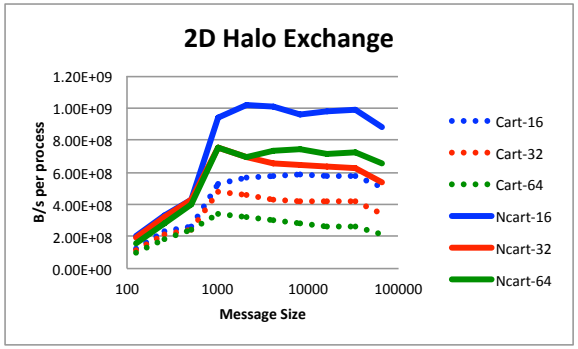


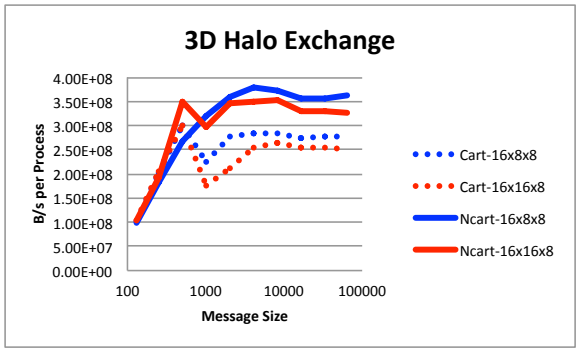
Can We Do Better?

- Hypothesis: A better process mapping **within** a node will provide significant benefits
  - **Ignore** the internode network topology
    - Vendors have argued that their network is fast enough that process mapping isn't necessary
    - They may be (almost) right – once data enters the network
- Idea for Cartesian Process Topologies
  - Identify nodes (see MPI_Comm_split_type)
  - Map processes *within* a node to minimize **inter**node communication
    - Trading **intra**node for **inter**node communication
    - *Using Node Information to Implement MPI Cartesian Topologies*, Gropp, William D., Proceedings of the 25th European MPI Users' Group Meeting, 18:1–18:9, 2018 https://dl.acm.org/citation.cfm?id=3236377
    - *Using Node and Socket Information to Implement MPI Cartesian Topologies*, Parallel Computing, 2019 https://doi.org/10.1016/j.parco.2019.01.001

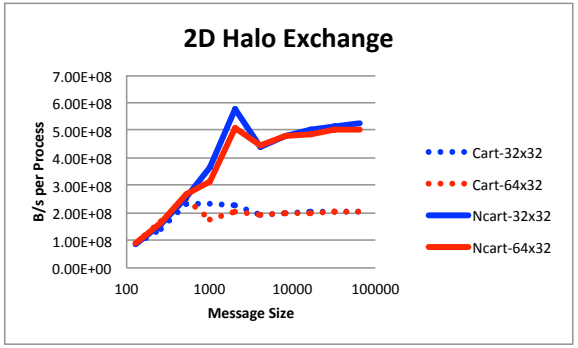
Comparing Halo Exchanges

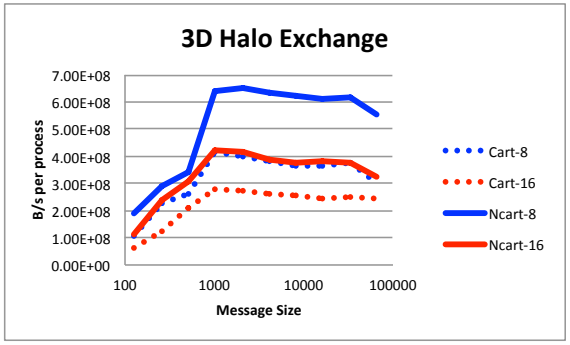
Blue Waters



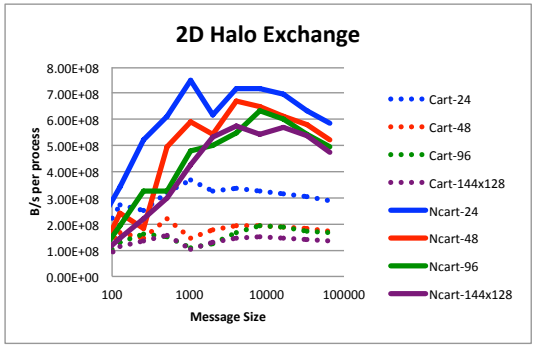


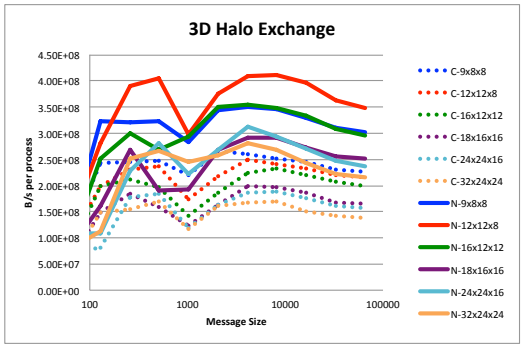
Theta





Piz Daint



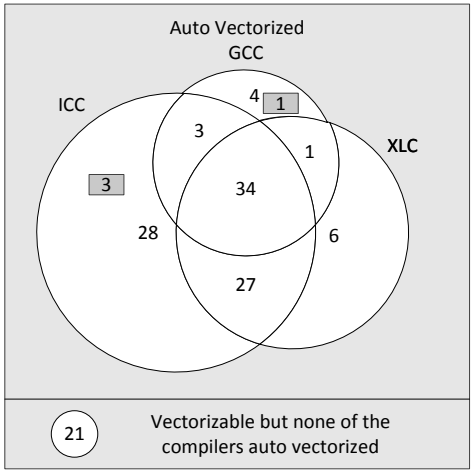


Dreams and Reality

- For codes that demand performance (and parallelism almost always implies that performance is important enough to justify the cost and complexity of parallelism), the dream is performance portability
- The reality is that most codes require specialized code to achieve high performance, even for non-parallel codes
- A typical refrain is “Let The Compiler Do It”
  - This is the right answer …
    - If only the compiler *could* do it
  - We have lots of evidence that this problem is unsolved – consider one of the most studied kernels – dense matrix-matrix multiply (DGEMM)
  - And what about vectorization?

A Simple (?) Problem: Generating Fast Code for Loops

- Long history of tools and techniques to produce fast code for loops
  - Vectorization, streams, etc., dating back nearly 40 years (Cray-1) or more
- Many tools for optimizing loops for both CPUs and GPUs
  - Compiler (auto) vectorization, explicit programmer use of directives (e.g., OpenMP or OpenACC), lower level expressions (e.g., CUDA, vector intrinsics)
- Is there a clear choice?
  - Not for vectorizing compilers (e.g., see S. Maleki, Y. Gao, T. Wong, M. Garzarán, and D. Padua, *An Evaluation of Vectorizing Compilers*. PACT 2011)
  - Probably not for the others
    - OpenACC preliminary examples follow
  - Vector tests part of baseenv; OpenACC and OpenMP vectorization tests under development (and some OpenACC examples follow)
- Need to separate description of semantics and operations from particular programming system choices



Can We Pick One Approach?

Loop Performance range in GF	Single Core Vectorized	OpenACC multicore	OpenACC tesla (loop)	OpenACC tesla (kernel)
Single Precision	2.6-16.3	1.1-3.3	394-1420	1.6-1710
Double Precision	1.3-8.2	--	320-826	1.4-731

- Test system node
  - 2 x Power9 (20 cores each) with 4 NVIDIA Tesla V100 GPU; Only 1 GPU used in tests
- Caveats
  - Only basic tuning performed (e.g., -O3, -fast)
  - Defaults used (almost certainly not full # cores for OpenACC multicore)
  - Data resident on GPU for all tests
  - Only 6 simple vector loop tests
  - Test time variations not included
- Take-aways
  - No absolute winner (though explicit OpenACC *for these loops* is close for GPU – but poor for CPU)
  - Can abstract memory domains
  - There are common abstractions but no one system is perfect
- If we can't have the dream, what do we really need?

Design Requirements

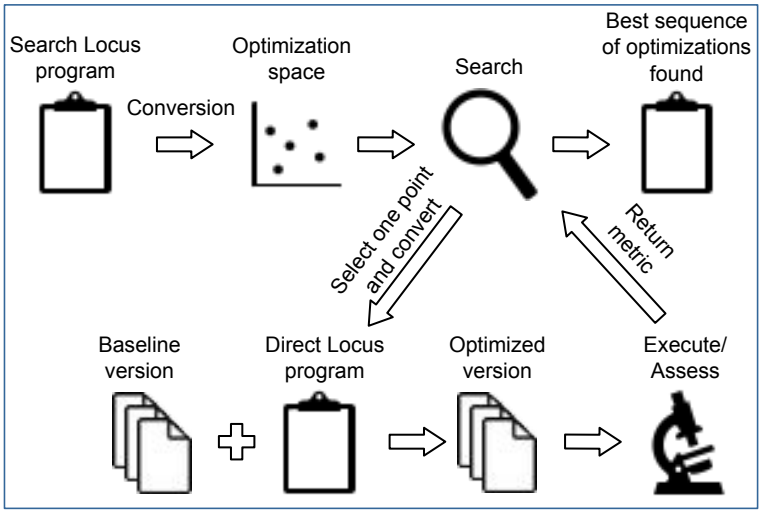
1. A clean version of the code for the developers. This is the *baseline* code.
2. The code should run in the absence of any tool, so that the developers are comfortable that their code will run.
3. A clean way to provide extra semantic information.
4. Code must run with good performance on multiple platforms and architectures.
5. A performance expert must be able to provide additional, possibly target-specific, information about optimizations.
6. The system must reuse the results of the autotuning step(s) whenever possible.
7. Changes to the baseline code should ensure that “stale” versions of the optimized code are not used and preferably replaced by updated versions.
8. Hand-tuned optimizations should be allowed.
9. Using (as opposed to creating) the optimized code *must not* require installing the code generation and autotuning frameworks.
10. The system should make it possible to gather performance data from a remote system.

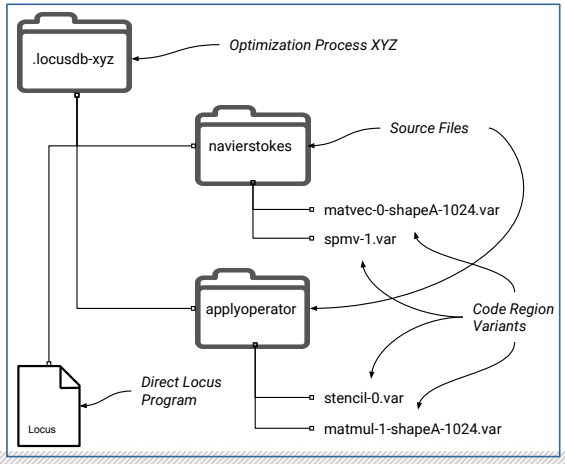
Design Implications

- Our system uses annotated code, written in C, C++, or Fortran, with high-level information that marks regions of code for optimization (addresses 1 and 2).
- The annotations only cover high-level, platform- independent information (addresses 3).
- Platform and tool-dependent information (e.g., loop-unroll depth) is maintained in a separate *optimization file* (addresses 5).
- We maintain a database of optimized code, organized by target platform and other parameters (addresses 4 and 6).
- The database maintains a hash of the relevant parts of the code for each transformed section (addresses 7).
- Hand-tuned versions of code may be inserted into the database (addresses 8 and 5).
- The system separates the steps of determining optimized code and populating the database from extracting code from the database to replace labeled code regions in the baseline version (addresses 9).
- The system provides some support for running tests on a remote system; especially important when the target is a supercomputer (addresses 9 and 10).
- Allow hand-optimized version as the default code, with clean baseline in database as source for transformations (addresses 2).

Locus

- Source code is annotated to define code regions
- Optimization file notation orchestrates the use of the optimization tools on the code regions defined
- Interface provides operations on the source code to invoke optimizations through:
  - Adding pragmas
  - Adding labels
  - Replacing code regions
- These operations are used by the interface to plug-in optimization tools
- Most tools are source-to-source
  - tools must understand output of previous tools
- Joint work with Thiago Teixeira and David Padua, “Managing Code Transformations for Better Performance Portability”, submitted to IJHPCA, 2018





Matrix Multiply Example

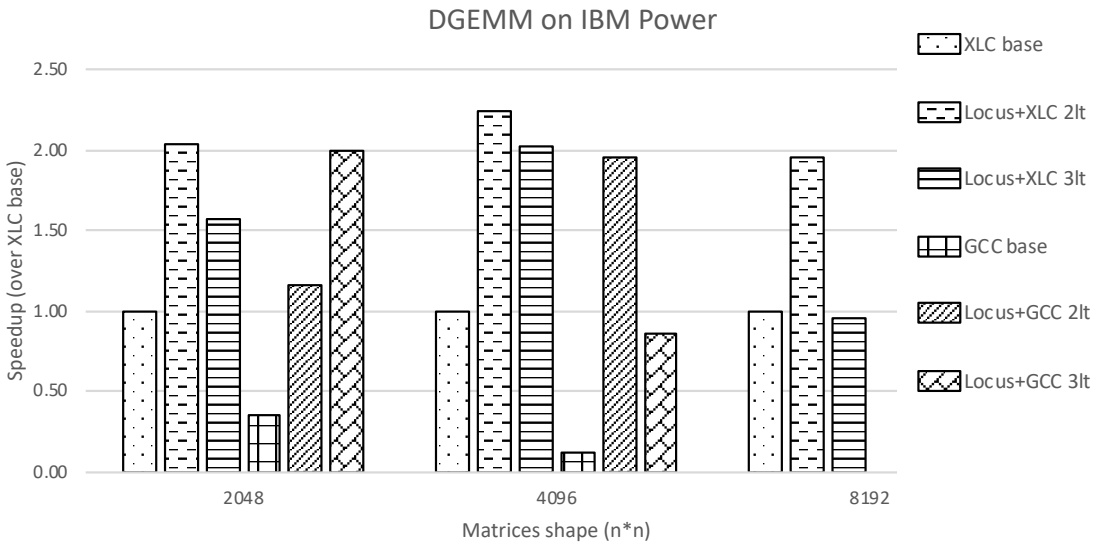
- **#pragma** @LOCUS loop=matmul
  for(i=0; i<M; i++)
    for(j=0; j<N; j++)
      for(k=0; k<K; k++)
        C[i][j] = beta*C[i][j] + alpha*A[i][k] * B[k][j];

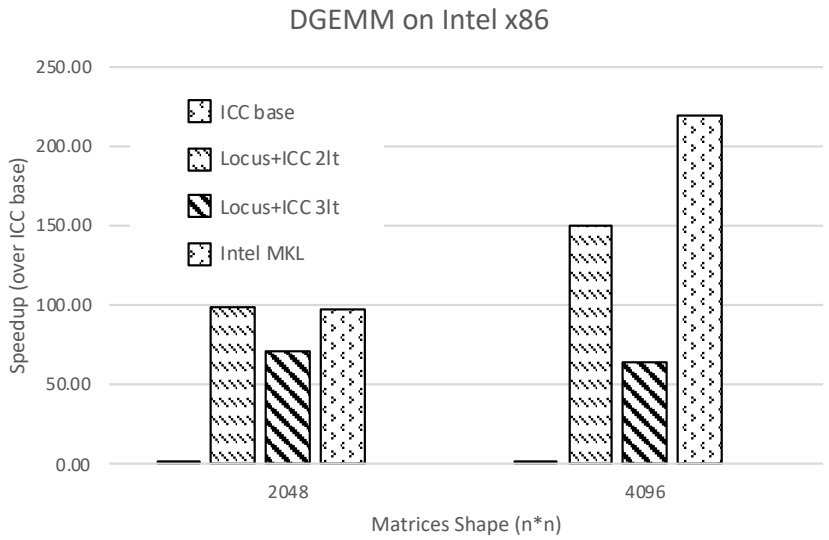
```
dim=4096;
Search {
  buildcmd = "make clean all";
  runcmd = "./matmul";
}
CodeReg matmul {
  RoseLocus.Interchange(order=[0,2,1]);
  tileI = poweroftwo(2..dim);
  tileK = poweroftwo(2..dim);
  tileJ = poweroftwo(2..dim);
  Pips.Tiling(loop="0", factor=[tileI, tileK, tileJ]);
  tileI_2 = poweroftwo(2..tileI);
  tileK_2 = poweroftwo(2..tileK);
  tileJ_2 = poweroftwo(2..tileJ);
  Pips.Tiling(loop="0.0.0.0",
              factor=[tileI_2, tileK_2, tileJ_2]);
  {
    tileI_3 = poweroftwo(2..tileI_2);
    tileK_3 = poweroftwo(2..tileK_2);
    tileJ_3 = poweroftwo(2..tileJ_2);
    Pips.Tiling(loop="0.0.0.0.0.0.0",
                factor=[tileI_3, tileK_3, tileJ_3]);
  } OR {
    None;
  }
}
```

Locus Generated Code (for specific platform/size)

- **#pragma** @LOCUS loop=matmul
  for(i_t = 0; i_t <= 7; i_t += 1)
  for(k_t = 0; k_t <= 3; k_t += 1)
  for(j_t = 0; j_t <= 1; j_t += 1)
  for(i_t_t = 8 * i_t; i_t_t <= ((8 * i_t) + 7); i_t_t += 1)
  for(k_t_t = 256 * k_t; k_t_t <= ((256 * k_t) + 255); k_t_t += 1)
  for(j_t_t = 32 * j_t; j_t_t <= ((32 * j_t) + 31); j_t_t += 1)
  for(i = 64 * i_t_t; i <= ((64 * i_t_t) + 63); i += 1)
  for(k = 4 * k_t_t; k <= ((4 * k_t_t) + 3); k += 1)
  for(j = 64 * j_t_t;j <= ((64 * j_t_t) + 63); j += 1)
    C[i][j] = beta*C[i][j] + alpha*A[i][k]*B[k][j];

DGEMM by Matrix Size



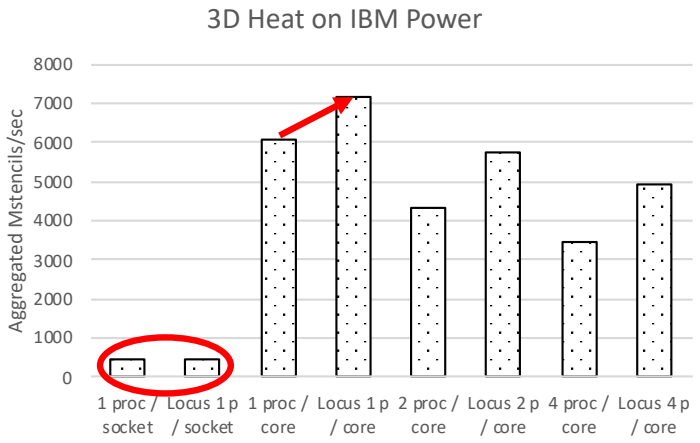


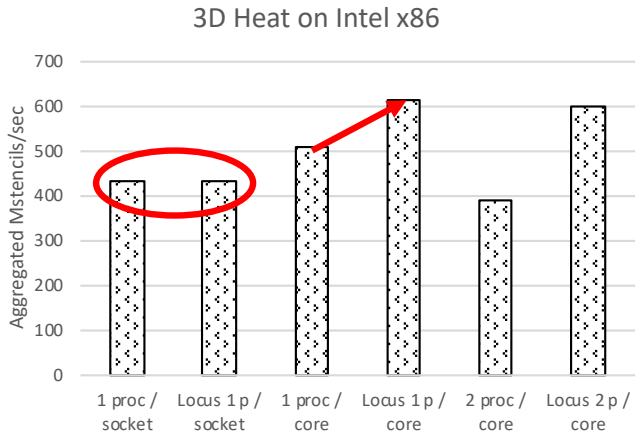
Tuning Must be in a Representative Environment

- For most processors and regular (e.g., vectorizable) computations
  - Memory bandwidth for a *chip* is much larger than needed by a single *core*
  - *Share of* memory bandwidth for a *core* (with all cores accessing memory) is much smaller than needed to avoid waiting on memory
- Performance tests on a single core can be very misleading
  - Example follows
  - Can use simple MPI tools to explore dependence on using one to all cores
    - See baseenv package
  - Ask this question when you review papers ☺

Stencil Sweeps

```
void heat3d(double A[2][N+2][N+2][N+2]) {
int i, j, t, k;
#pragma @LOCUS loop=heat3d
for(t = 0; t < T-1; t++) {
for(i = 1; i < N+1; i++) {
for(j = 1; j < N+1; j++) {
for (k = 1; k < N+1; k++) {
A[(t+1)%2][i][j][k] = 0.125 * (A[t%2][i+1][j][k] -
2.0 * A[t%2][i][j][k] + A[t%2][i-1][j][k]) + 0.125 * (A[t%2][i][j+1][k]
- 2.0 * A[t%2][i][j][k] + A[t%2][i][j-1][k]) + 0.125 * (A[t%2][i][j][k-
1] - 2.0 * A[t%2][i][j][k] + A[t%2][i][j][k+1]) + A[t%2][i][j][k]; } } } }
}
```





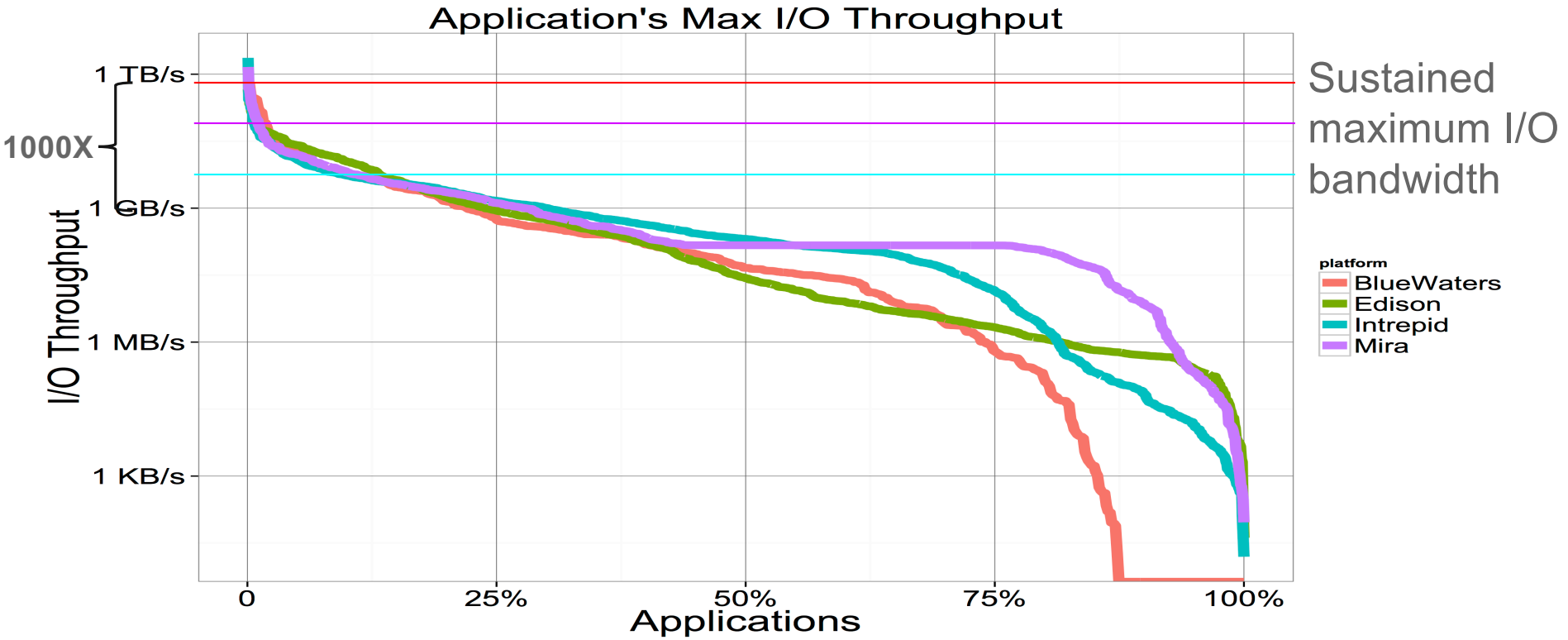
Often Overlooked – IO Performance Often Terrible

- Applications just assume I/O is awful and can't be fixed
- Even simple patterns not handled well
- Example: read or write a submesh of an N-dim mesh at an arbitrary offset in file
- Needed to read input mesh in PlasComCM. Total I/O time less than 10% for long science runs (that is < 15 hours)
  - But long init phase makes debugging, development hard

	Original	Meshio	Speedup
PlasComCM	4500	1	4500
MILC	750	15.6	48

- Meshio library built to match application needs
- Replaces many lines in app with a single *collective* I/O call
- Meshio https://github.com/oshkosher/meshio
- Work of Ed Karrels

Just how bad Is current I/O performance?



Sustained maximum I/O bandwidth

"*A Multiplatform Study of I/O Behavior on Petascale Supercomputers*," Huong Luu, Marianne Winslett, William Gropp, Robert Ross, Philip Carns, Kevin Harms, Prabhat, Suren Byna, and Yushu Yao, proceedings of HPDC'15.

What Are Some of the Problems?

- POSIX I/O has a strong and required consistency model
  - Hard to cache effectively
  - Applications need to transfer block-aligned and sized data to achieve performance
  - Complexity adds to fragility of file system, the major cause of failures on large scale HPC systems
- Files as I/O objects add metadata "choke points"
  - Serialize operations, even with "independent" files
  - Do you know about O_NOATIME ?
- Burst buffers will *not* fix these problems – must change the semantics of the operations
  - Typical approach is to falsely claim POSIX while violating POSIX semantics, causing some applications to fail
- "Big Data" file systems have very different consistency models and metadata structures, designed for their application needs
  - Why doesn't HPC?
    - There have been some efforts, such as PVFS, but the **requirement** for POSIX has **held up** progress
- Real problem for HPC – user's "execution model" for I/O far from reality

No Science Application Code Needs POSIX I/O
(precisely, no app needs POSIX consistency semantics)

- Many are single reader or single writer
  - Eventual consistency is fine
- Some are disjoint reader or writer
  - Eventual consistency is fine, but must correctly handle non-block-aligned writes
- Some applications use the file system as a simple data base
  - Use a data base – we know how to make these fast and reliable
- Some applications use the file system to implement interprocess mutex
  - Use a mutex service – even MPI point-to-point
- A few use the file system as a bulletin board
  - May be better off using RDMA (available in MPI)
  - Only need release or eventual consistency
- *Correct* Fortran codes do not require POSIX (in any form)
  - Standard requires unique open, enabling correct and aggressive client and/or server-side caching
- MPI-IO would be better off without POSIX (in any form)
  - Does not and never has required POSIX

Summary

- Challenges for HPC programming are not just in scale
  - Need to achieve extreme power and cost efficiencies puts large demands on the effectiveness of single core (whatever that means) and single node performance
- MPI remains the most viable internode programming *system*
  - Supports a multiple parallel programming models, including one-sided and shared memory
  - Contains features for “programming in the large” (tools, libraries, frameworks) that make it particularly appropriate for the internode programming system
- Intranode programming for performance still an unsolved problem
  - Lots of possibilities, but adoption remains a problem
    - That points to unsolved problems, particularly in integration with large, multilingual codes
  - Composition of tools (rather than a single does-everything compiler) a promising approach
- Parallel I/O increasingly important
  - But HPC centers need to change their approach and embrace the “big data” view

Thanks!

- Philipp Samfass, Ed Karrels, Amanda Bienz, Paul Eller, Thiago Teixeira
- Luke Olson, David Padua
- Rajeev Thakur for runs on Theta
- Torsten Hoefler and Timo Schneider for runs on Piz Daint

- Department of Energy, National Nuclear Security Administration, under Award Number DE-NA0002374
- National Science Foundation Major Research Instrumentation program, grant #1725729,
- ExxonMobil Upstream Research
- Blue Waters Sustained Petascale Project, supported by the National Science Foundation (award number OCI 07–25070) and the state of Illinois.
- Argonne Leadership Computing Facility