# 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
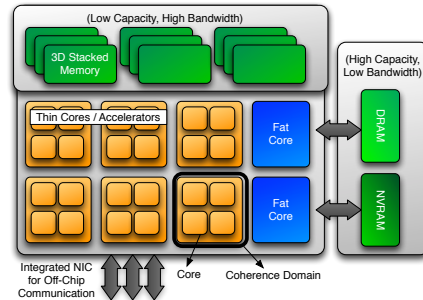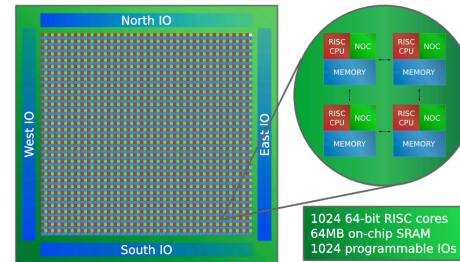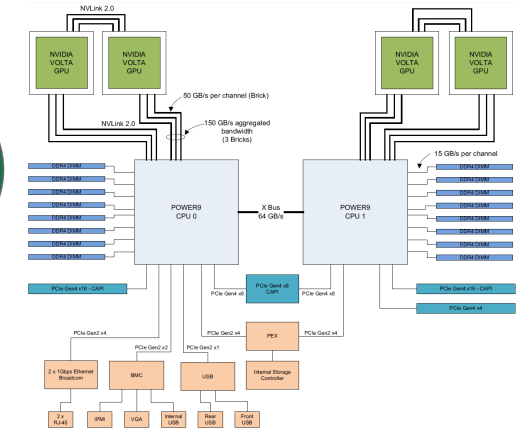


Figure 2.1: Abstract Machine Model of an exascale Node Architecture

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

NCSA

# 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"!

NCSA

# 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 assumes 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?

NCSA

# Rates Per MPI Process
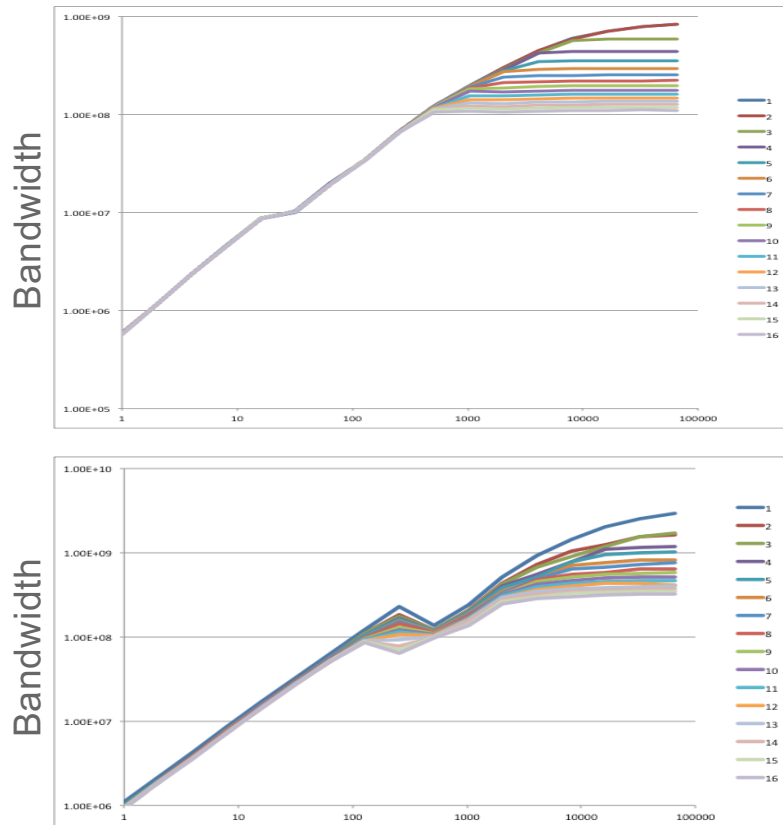


- Ping-pong between 2 nodes using 1-16 cores on each node
- Top is BG/Q, bottom Cray XE6
- "Classic" model predicts a single curve – rates independent of the number of communicating processes

NCSA

# 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?
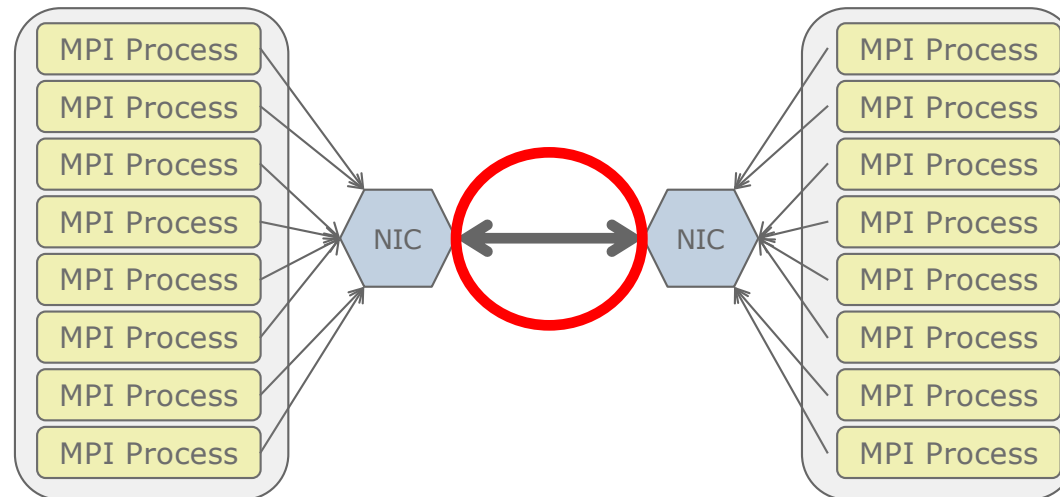
# A Slightly Better Model

- For k processes sending messages, the sustained rate is
    - $\min(R_{NIC-NIC}, k\, R_{CORE-NIC})$
- Thus
    - $T = s + k\, n/\min(R_{NIC-NIC}, k\, R_{CORE-NIC})$
- Note if $R_{NIC-NIC}$ is very large (very fast network), this reduces to
    - $T = s + k\, n/(k\, R_{CORE-NIC}) = s + n/R_{CORE-NIC}$
- 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
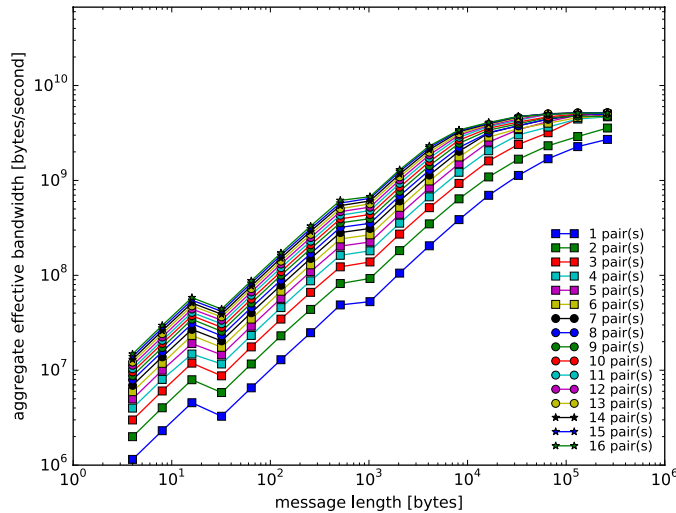


Measured Data
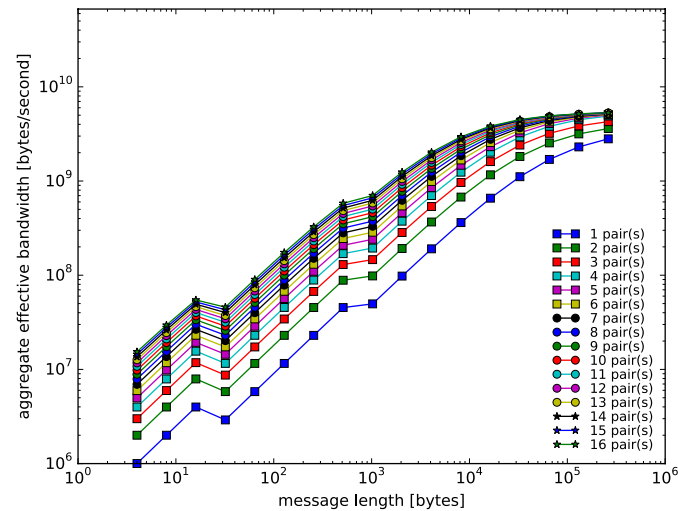
Max-Rate Model

# 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: 4 Nodes, 4 Cores/Node



Desired (Expected?)
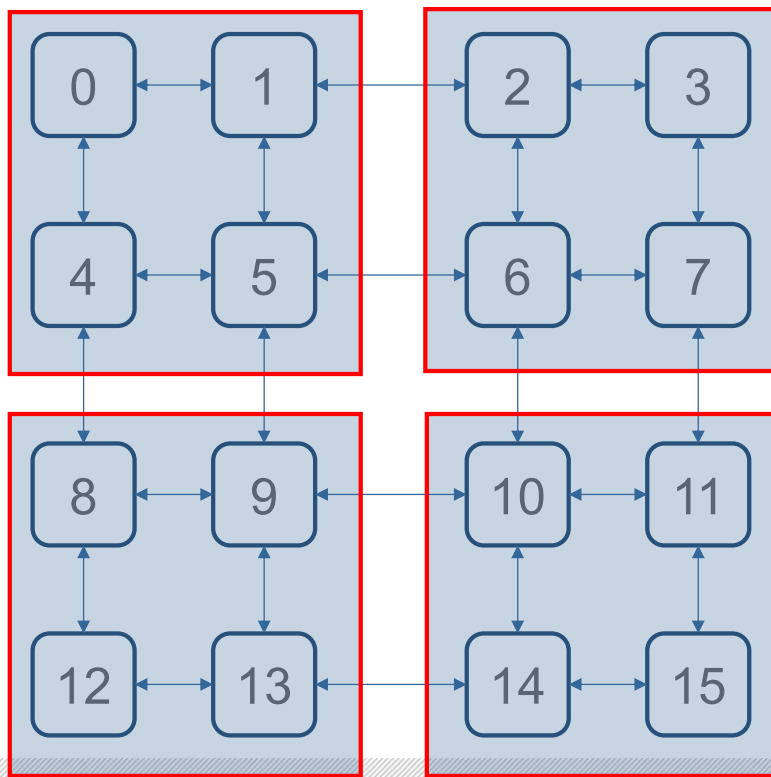
Typical Process Mapping
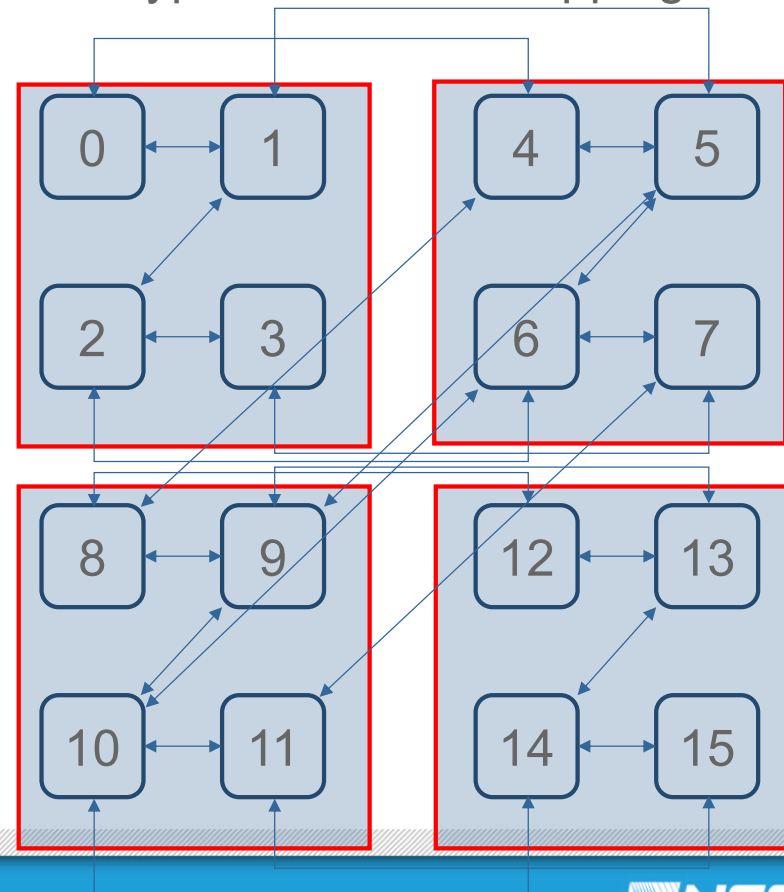
# Can We Do Better?

- 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

NCSA

# Algorithm

- Find the nodes
  - MPI provides a way to split a communicator based on a characteristic; MPI_COMM_TYPE_SHARED works on all systems
- Create communicators of (a) all processes on the same node (nodecomm) and (b) the $0^{th}$ process from each node (leadercomm)
  - All processes now know number of processes on each node and the number of nodes
- Form a 2 (or 3) level decomposition of the process mesh
  - Factor dimensions and find consistent pair in each dimension
- From rank in nodecom and leadercomm, compute coordinates in node and among nodes. Gives new coordinate in mesh and hence new rank
- Use MPI_Comm_split on this rank to form new Cartesian communicator

# Testing the Hypothesis: The Systems

- Blue Waters at Illinois
  - Cray XE6/XK7
  - 3D mesh (Gemini); service nodes embedded in mesh
  - 22,636 XE6 nodes, each with 2 AMD Interlagos (and 4228 XK7 nodes)

- Theta at Argonne
  - Cray XC40
  - Dragonfly (Aires) interconnect
  - 4392 Intel KNL nodes

- Piz Daint at Swiss National Supercomputing Center
  - Cray XC50/XC40
  - Dragonfly (Aires) interconnect
  - 5320 XC50 and 1813 XC40 nodes
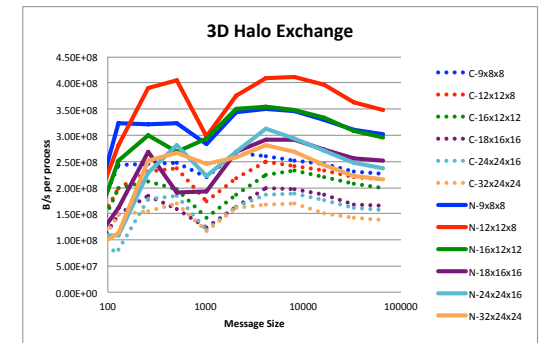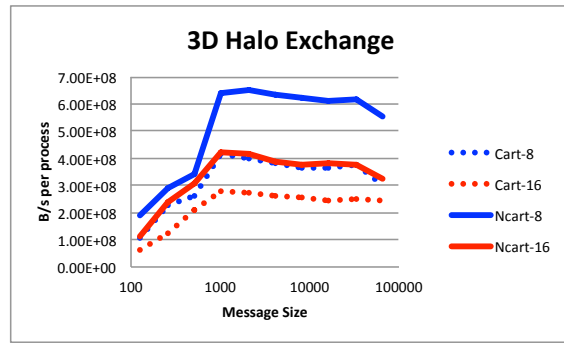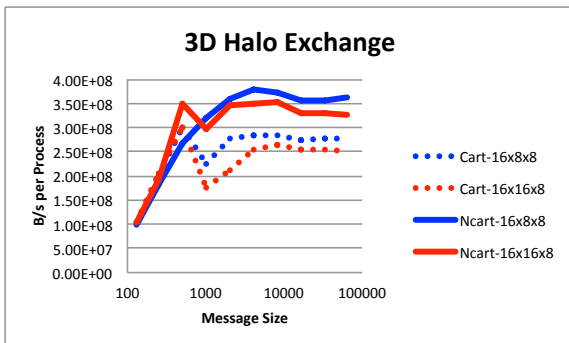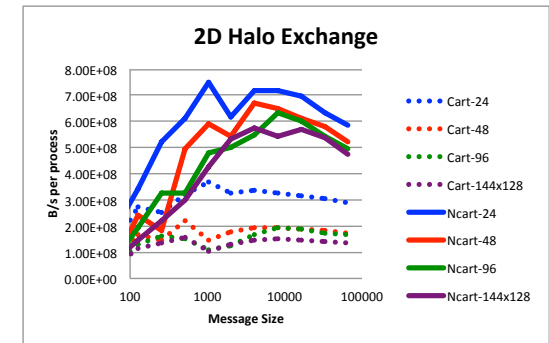
# Comparing Halo Exchanges
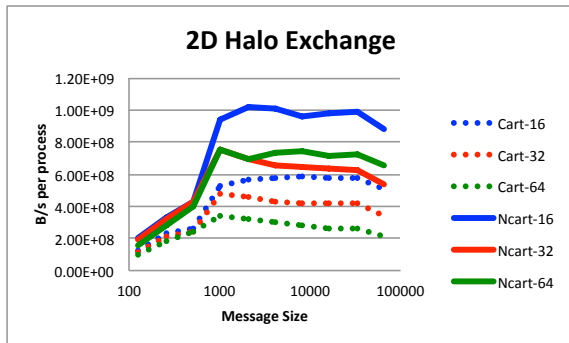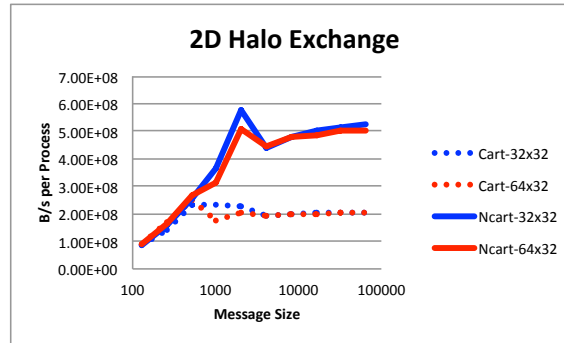
# How Important is Network Topology?

- No answer yet, but…

- 432 nodes, 3D halo exchange on Blue Waters
  - Requested a cube of nodes, used non-standard routines to implement mapping for network topology

- Part of study into scalable Krylov methods (looking to avoid the blocking MPI_Allreduce)

- Nodecart version provides most of the benefit with no need for network topology information

- Some (nontrivial) further benefit possible by taking network topology into account

- But the largest contribution comes from node-awareness

- Thanks to Paul Eller for these results



**Topology-aware Halo**

Legend: Original, Node-aware, Topology-aware

X-axis: Message Size

Y-axis: Runtime (s)



**Topology-aware Halo Congestion**

Legend: Original, Node-aware, Topology-aware

X-axis: Message Size

Y-axis: Stalls per Byte

NCSA
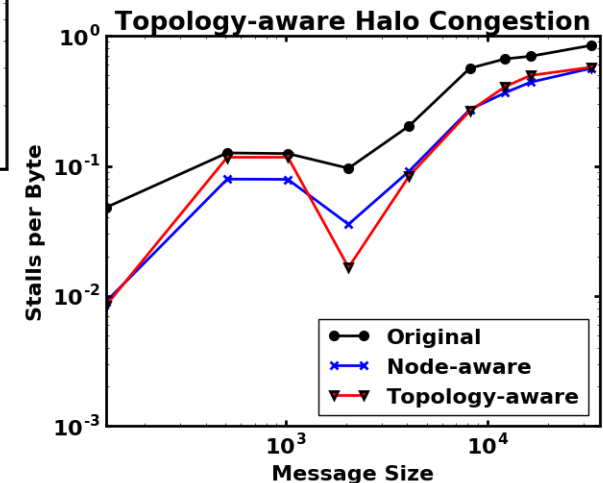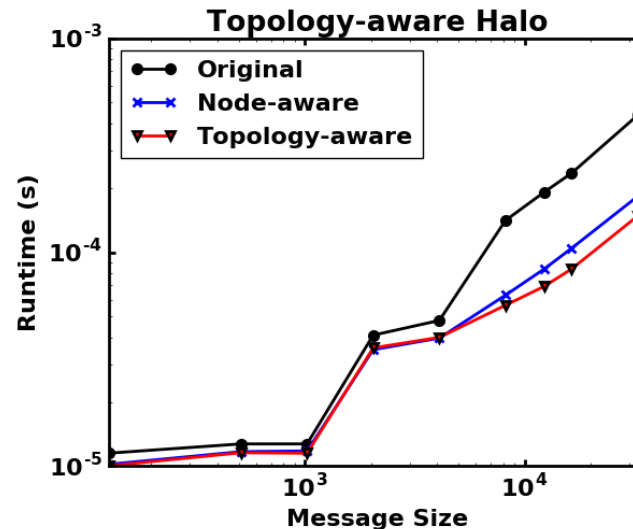
# Further Refining the Model: SpMV for Algebraic Multigrid
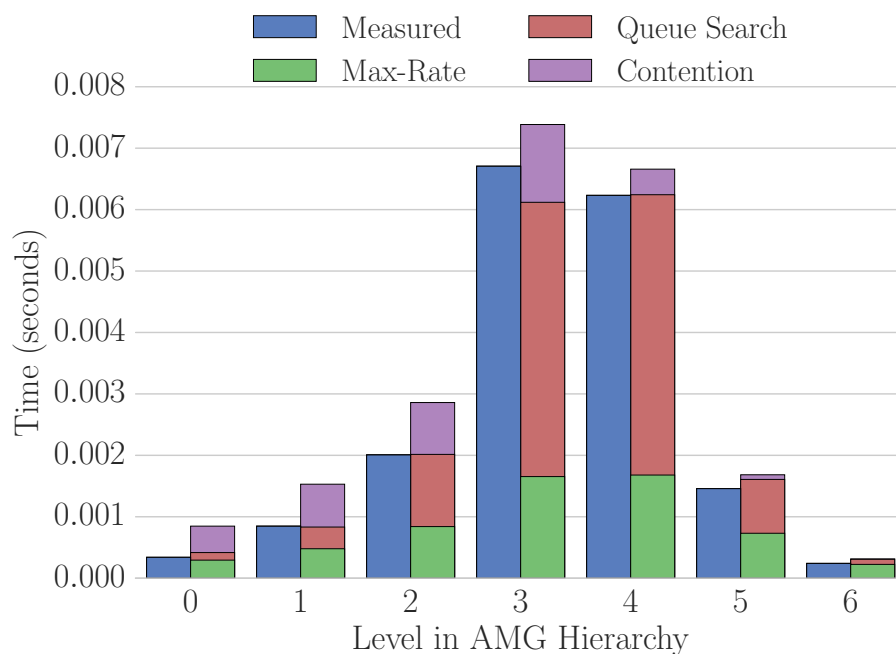


- Intermediate levels if AMG Coarse Grid problem require many messages
- Model greatly improved with queue search time and contention parameters
- Queue search time dominates cost on coarse levels
- Leads to new algorithm that improves performance
- Work of Amanda Bienz et al https://arxiv.org/abs/1806.02030

# Impact of Node-Aware Communication

Cost of Ruge-Stuben (RS) and Smoothed Aggregation (SA) AMD compared to Hypre
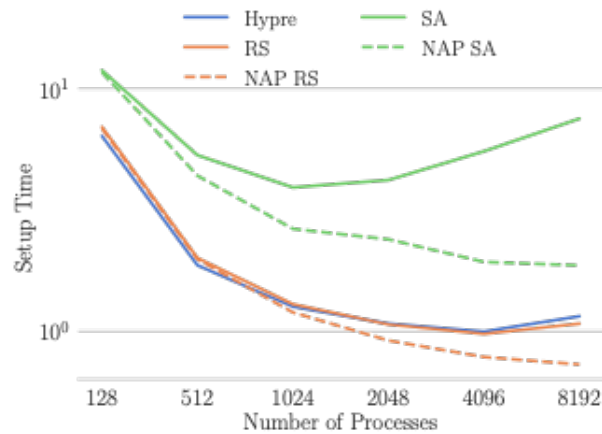


DG Diffusion
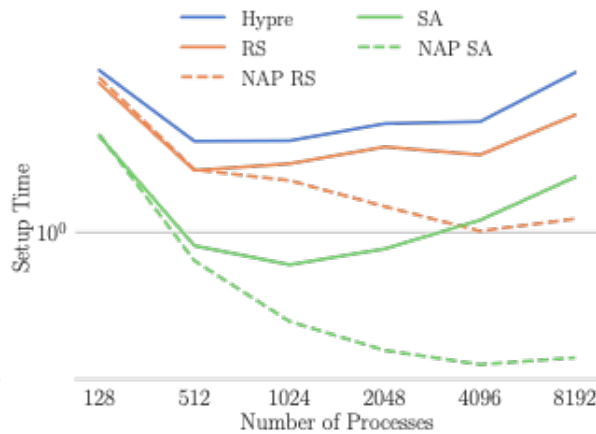
Grad-Div
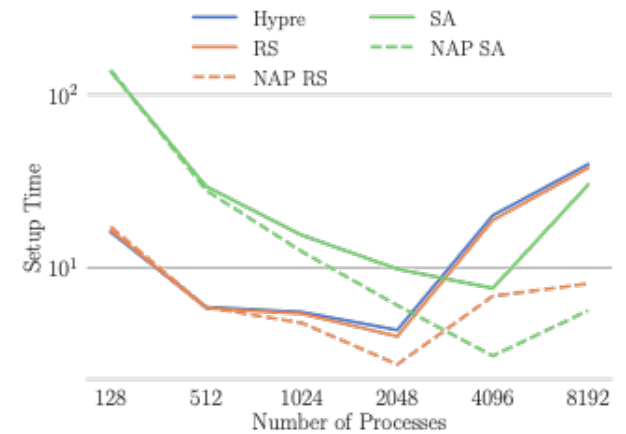
Laplacian

Work of Amanda Bienz and Luke Olson

# 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?

NCSA

# 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
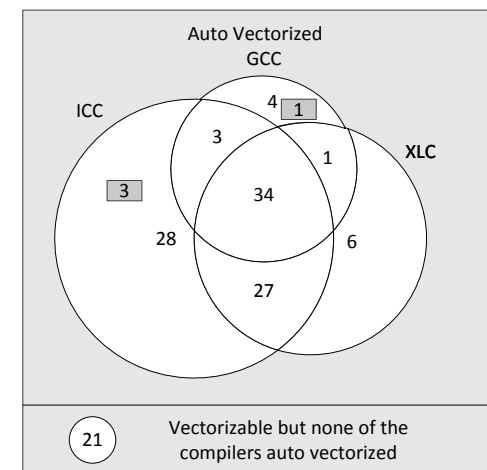
# Can We Pick One Approach?

| 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

**NCSA**

# A Simple Example: Dense Matrix Transpose

- Lets look at one of the simplest operations for a single core, dense matrix transpose
  - Only a double loop (fewer options to consider)

- do j=1,n
    do i=1,n
      b(i,j) = a(j,i)
    enddo
  enddo

- No temporal locality (data used once)

- Spatial locality only if (words/cacheline) * n fits in cache



Perf limit based on STREAM
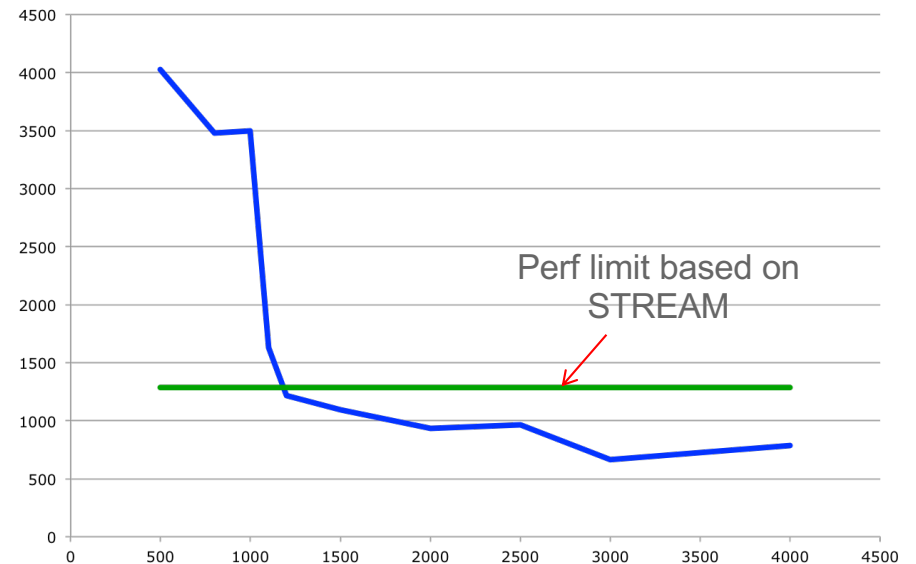
- Performance plummets when matrices no longer fit in cache

NCSA

# Blocking for cache helps

- do jj=1,n,stridej
      do ii=1,n,stridei
          do j=jj,min(n,jj+stridej-1)
              do i=ii,min(n,ii+stridei-1)
                  b(i,j) = a(j,i)

- Good choices of stridei and stridej can improve performance by a significant factor

- How sensitive is the performance to the choices of stridei and stridej?

Simple, unblocked code compiled with O3 – 709MB/s



1500-2000
1000-1500
500-1000
0-500



1800-2000
1600-1800
1400-1600
1200-1400
1000-1200
800-1000
600-800
400-600
200-400
0-200

NCSA
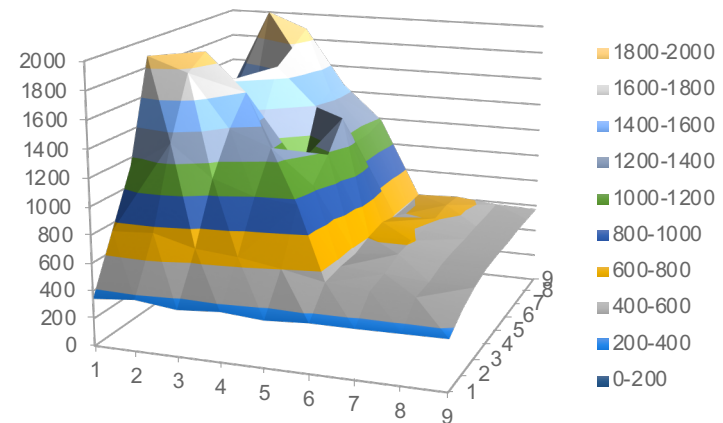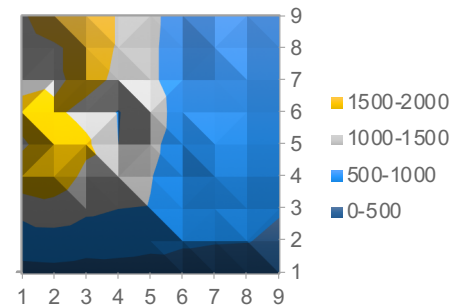
# Real Codes Include Performance Workarounds

- Code excerpt from VecMDot_Seq in PETSc
- Code is unrolled to provide performance
  - Decision was made once (and verified as worth the effort *at the time*)
  - Remains part of the code forevermore
  - Unroll by 4 *probably* good for vectorization
    - But not necessarily best for performance
    - Does not address alignment
- If we can't have the dream, what do we really need?

```
switch (j_rem=j&0x3) {
case 3:
  x2    = x[2];
  sum0 += x2*yy0[2]; sum1 += x2*yy1[2];
  sum2 += x2*yy2[2];
case 2:
  x1    = x[1];
  sum0 += x1*yy0[1]; sum1 += x1*yy1[1];
  sum2 += x1*yy2[1];
case 1:
  x0    = x[0];
  sum0 += x0*yy0[0]; sum1 += x0*yy1[0];
  sum2 += x0*yy2[0];
case 0:
  x   += j_rem;
  yy0 += j_rem;
  yy1 += j_rem;
  yy2 += j_rem;
  j   -= j_rem;
  break;
}
while (j>0) {
  x0 = x[0];
  x1 = x[1];
  x2 = x[2];
  x3 = x[3];
  x += 4;

  sum0 += x0*yy0[0] + x1*yy0[1] + x2*yy0[2] + x3*yy0[3]; yy0+=4;
  sum1 += x0*yy1[0] + x1*yy1[1] + x2*yy1[2] + x3*yy1[3]; yy1+=4;
  sum2 += x0*yy2[0] + x1*yy2[1] + x2*yy2[2] + x3*yy2[3]; yy2+=4;
  j    -= 4;
}
z[0] = sum0;
z[1] = sum1;
z[2] = sum2;
```

NCSA

# 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.
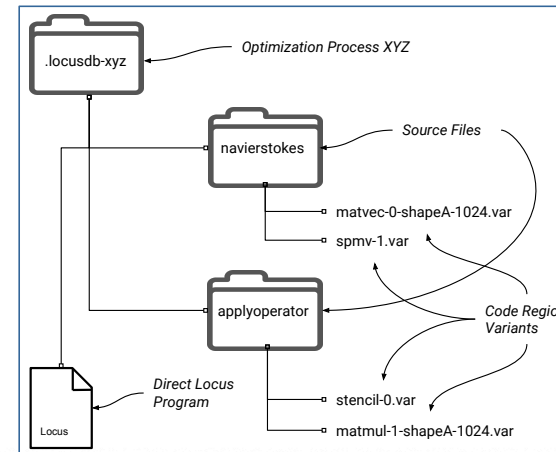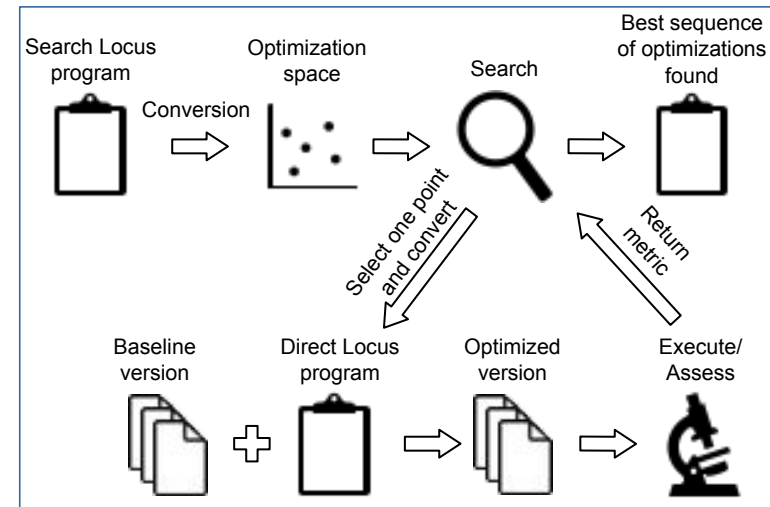
# Design Implications

- 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", IJHPCA, 2019
https://doi.org/10.1177%2F1094342019865606



Search Locus program → Conversion → Optimization space → Search → Best sequence of optimizations found

Select one point and convert → Baseline version + Direct Locus program → Optimized version → Execute/Assess → Return metric



.locusdb-xyz — Optimization Process XYZ

navierstokes — Source Files
matvec-0-shapeA-1024.var
spmv-1.var

applyoperator — Code Region Variants

Locus — Direct Locus Program
stencil-0.var
matmul-1-shapeA-1024.var

# 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;
  }
}
```
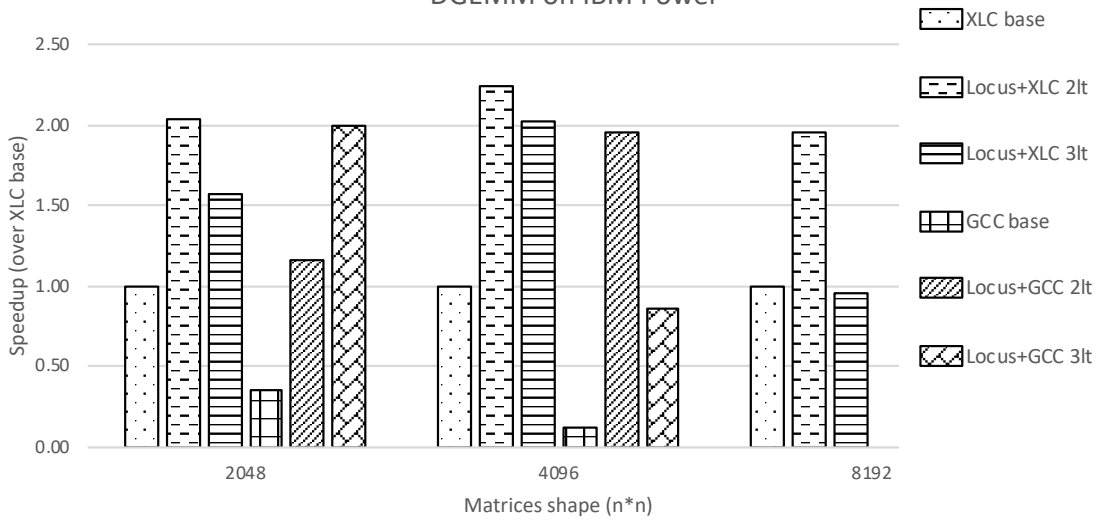
# Locus Generated Code
# (for specific platform/size)

- **#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



DGEMM on IBM Power

XLC base
Locus+XLC 2lt
Locus+XLC 3lt
GCC base
Locus+GCC 2lt
Locus+GCC 3lt

Speedup (over XLC base)
Matrices shape (n*n)

DGEMM on Intel x86

ICC base
Locus+ICC 2lt
Locus+ICC 3lt
Intel MKL

Speedup (over ICC base)
Matrices Shape (n*n)
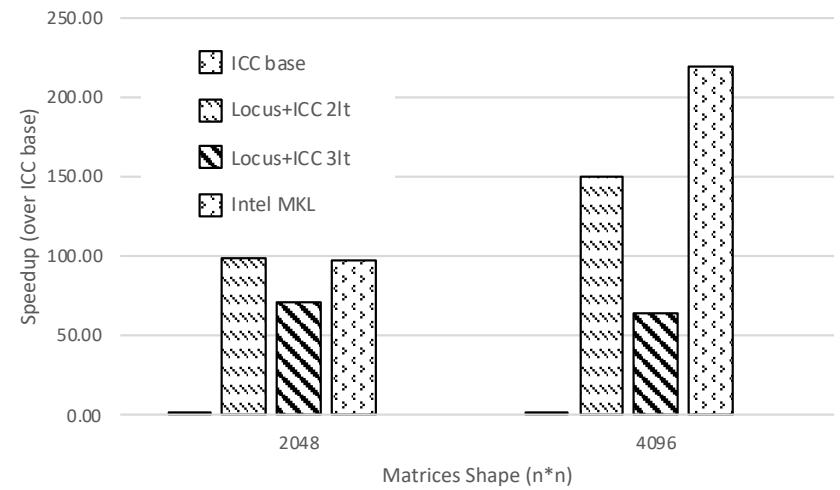
# Tuning Must be in a Representative Environment

- 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

- Common operation for PDE solvers
  - Structured are often "matrix free"
  - Unstructured and structured mesh stencils have low "computational intensity" – number of floating point operations per bytes moved
- Conventional wisdom is that cache blocking and similar optimizations are ineffective
  - For example, "Optimization and Performance Modeling of Stencil Computations on Modern Microprocessors" argues this, and provides experimental data to support it
  - https://epubs.siam.org/doi/10.1137/070693199 (accepted 2007, published 2009)
- But the analysis and experiments are usually based on one core per chip/socket
  - And the number of cores has grown substantially since 2007
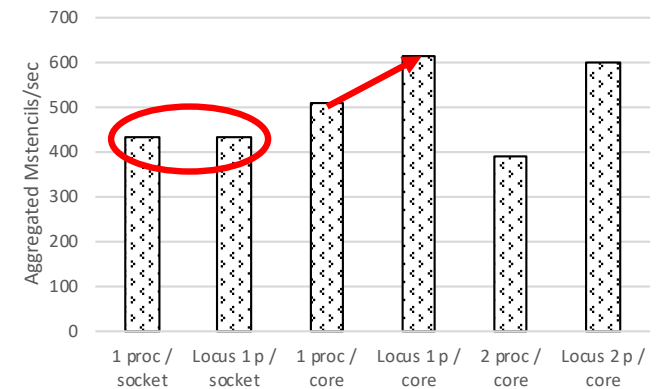  - What if every core is executing a stencil sweep?

# 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]; } } } }
}
```



3D Heat on IBM Power
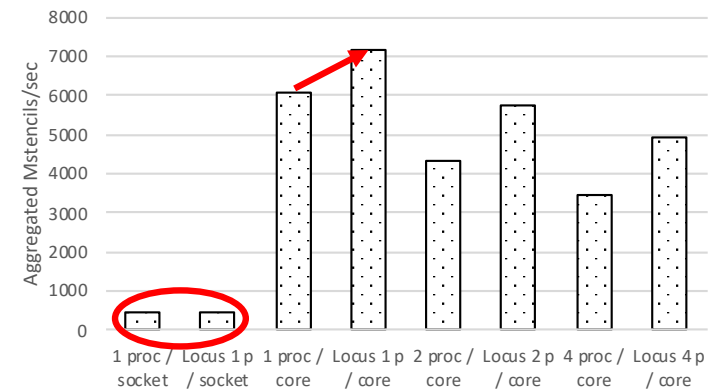


3D Heat on Intel x86

# 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?



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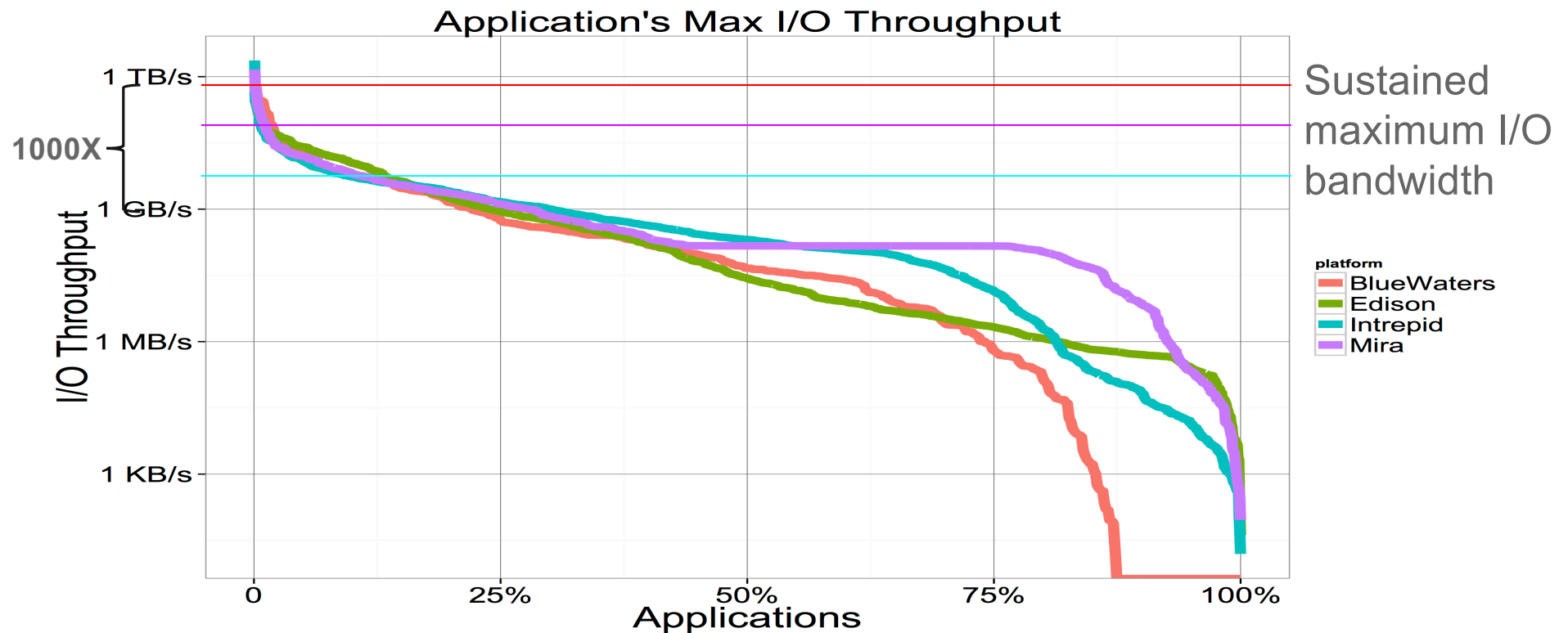
"*A Multiplatform Study of I/O Behavior on Petascale Supercomputers*," Huong Luu, Marianne Winslett, William Gropp, Robert Ross, Philip Carns, Kevin Harms, Prabhat, Suren Byna, and Yushu Yao, proceedings of HPDC'15.

# What Are Some of the Problems?

- POSIX I/O has a strong 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 ?

# But POSIX Works (Or We Can Fix It)

- "Our file system is stable"
  - Sometimes (Often?) due to operating in a subset of POSIX semantics
  - One National Lab told me this, but I also know that they pushed one of our students off the system because that student kept causing the file system to go down – and that student was running a correct, POSIX-compliant program
- In some cases, systems turn off POSIX correctness to provide better performance
  - But applications that rely on concurrent writes then mail fail, even though those applications are correct
- Burst buffers will not fix these problems
  - Hard to get effective use without changing the semantics of the operations

# What Options Are There?

- Instead of ignoring inconvenient parts of the POSIX specification, why not consider more modern high performance I/O designs?

- "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

# Remember

- POSIX is not just "open, close, read, and write" (and seek …)
  - That's (mostly) syntax
- POSIX includes strong semantics about concurrent accesses
  - Even if such accesses never occur
- POSIX also requires consistent metadata
  - Access and update times, size, …

# No Science Application Code Needs POSIX I/O

- 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
  - Standard requires unique open, enabling correct and aggressive client and/or server-side caching
- MPI-IO would be better off without POSIX
  - Does not and never has required POSIX

**NCSA**

# The really hard part – Combining internode and intranode programming systems

- Most common approach likely to be MPI + X

- What To Use as X in MPI + X?
  - Threads and Tasks
    - OpenMP, pthreads, TBB, OmpSs, StarPU, …
  - Streams (esp for accelerators)
    - OpenCL, OpenACC, CUDA, …
  - Alternative distributed memory system
    - UPC, CAF, Global Arrays, GASPI/GPI
  - MPI shared memory

# What are the Issues?

- Isn't the beauty of MPI + X that MPI and X can be learned (by users) and implemented (by developers) independently?
  - Yes (sort of) for users
  - No for developers
- MPI and X must either partition or share resources
  - User must not blindly oversubscribe
  - Developers must negotiate
- What can you do now?
  - Systems are providing more control of allocation of processes and threads to nodes, sockets, and cores. Unfortunately, each system is different.
  - Be aware of all uses of resources – don't forget the OS, runtime systems, monitoring demons, etc.

NCSA

# More Effort needed on the "+"

- MPI+X won't be enough for Exascale if the work for "+" is not done very well
  - Some of this may be language specification:
    - User-provided guidance on resource allocation, e.g., MPI_Info hints; thread-based endpoints, new APIs
  - Some is developer-level standardization
    - A simple example is the MPI ABI specification – users should ignore but benefit from developers supporting

# 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

NCSA

# Taking Advantage of Intranode Communication

- The "flat" execution model (all cores the same regardless of location) is no longer a good guide for algorithm design or application development
- Many examples where node-aware methods provide an advantage
  - Cartesian topology – better implementation of MPI_Cart_create
  - Node-aware Algebraic MultiGrid (AMG) – Raptor library provides significant speedup over Hypre
    - Uses streamline library to simplify using node-aware communication in place of direct use of MPI isend/irecv/wait
  - Faster allreduce – SMP-aware algorithms for MPI collectives reduce to a master for the node. Node-aware algorithms are more balanced, faster for shorter (e.g., 1 to a few doubles) operations
  - Graph partitioning – What is the cost model used in choosing cuts? Most current methods based on a simple, flat cost model.

# More Details and Software

- **github.com/cedar-framework/cedar**
  - *Scaling Structured Multigrid to 500K+ Cores through Coarse-Grid Redistribution* Reisner, Olson, Moulton, SISC, 2018

- **github.com/raptor-library/raptor**
  - *Node-Aware Sparse Matrix-Vector Communication* Bienz, Gropp, Olson, JPDC, 2019
  - *Improving Performance Models for Irregular Point-to-Point Communication* Bienz, Gropp, Olson, EuroMPI, 2018. https://dl.acm.org/citation.cfm?doid=3236367.3236368
  - *Reducing Communication in Algebraic Multigrid with Multi-step Node Aware Communication,* https://arxiv.org/abs/1904.05838

- **github.com/bienz2/Node_Aware_MPI**

- **github.com/bienz2/streamline**
  - Node-aware communication library

# Thanks!

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

NCSA