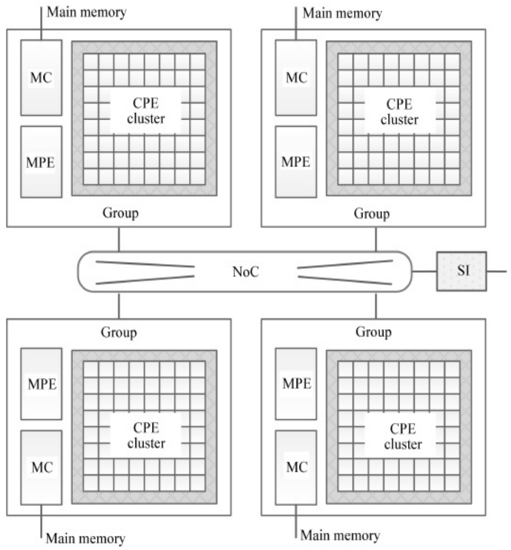
Challenges in Programming Extreme Scale Systems

William Gropp
wgropp.cs.illinois.edu

Some Likely Exascale Architectures



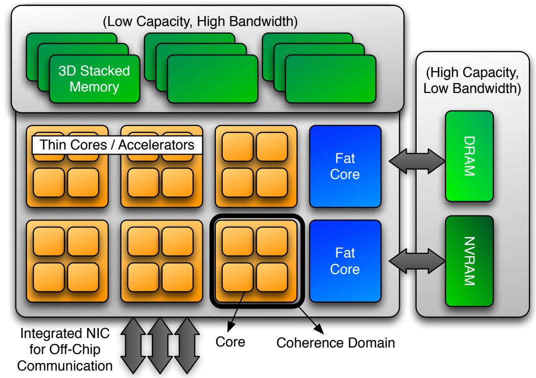
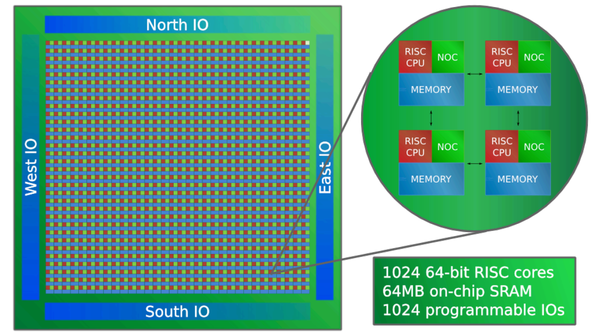


Figure 2.1: Abstract Machine Model of an exascale Node Architecture



Sunway TaihuLight

- Heterogeneous processors (MPE, CPE)
- No data cache
- Tianhe2a has some data cache

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)

New Applications Will Be As Varied and Demanding

- Wide range of applications today
  - More than CFD, Structural Mechanics, Molecular dynamics, QCD
  - Include image processing, event-driven simulations, graph analytics
- Rising importance of machine learning and ***Imitation Intelligence***
  - The appearance of intelligence without anything behind it
  - Still incredibly powerful and useful, but …
  - Not ***Artificial intelligence***
    - Intelligence achieved through artificial means
  - Training required for each "behavior" (one reason this is II, not AI)
  - Current methods require large amounts of data and compute to train; application of the trained system is not (relatively speaking) computationally intensive
- Workflows involving all of the above
  - One example:
    - Use Einstein Toolkit to compute gravity waves from cataclysmic events
      - This is classic time-dependent PDE solution
    - Use waveforms to train a machine learning system
    - Use that system to provide (near) real time detection of gravity waves from aLIGO
  - Many workflow-related events at SC

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

Modern MPI

- MPI is much more than message passing
  - I prefer to call MPI a programming *system*
    - 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

There are challenges

- Implementations not always as efficient as they could / should be
- One sided notification still limited (and under discussion)
- A standard moves slowly (and it should)
  - But a drawback when architectural innovation is fast
  - We need examples that go past MPI
    - But they don't need to *replace* MPI

MPI (The Standard) Can Scale Beyond Exascale

- MPI implementations already supporting more than 1M processes
  - Several systems (including Blue Waters) with over 0.5M independent cores
- Many Exascale designs have a similar number of nodes as today's systems
  - MPI as the <u>internode</u> programming system seems likely
- There are challenges
  - Connection management
  - Buffer management
  - Memory footprint
  - Fast collective operations
  - …
  - And no implementation is as good as it needs to be, but
  - There are no intractable problems here – MPI implementations can be engineered to support Exascale systems, even in the MPI-everywhere approach

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?

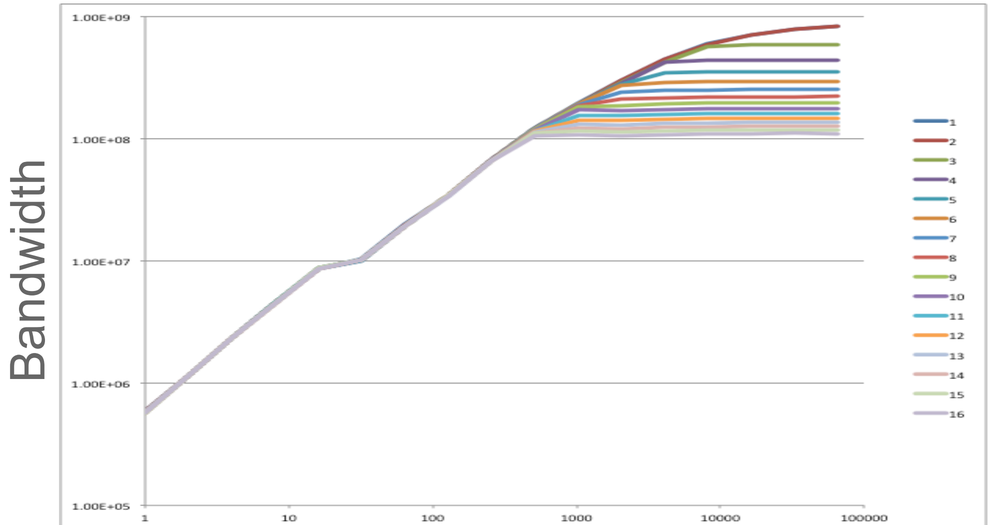
MPI is not a BSP system

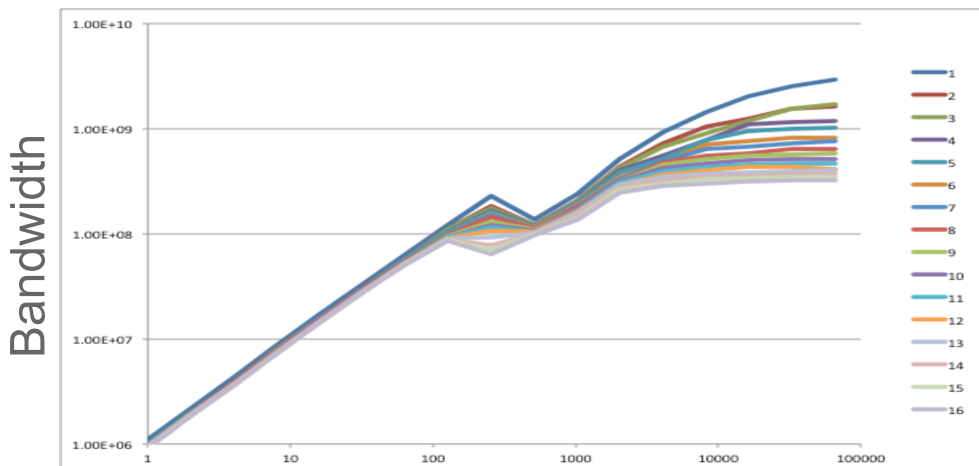
- BSP = Bulk Synchronous Programming
  - Programmers **like** the BSP model, adopting it even when not necessary (see “A Formal Approach to Detect Functionally Irrelevant Barriers in MPI Programs”)
  - Unlike most programming models, *designed* with a **performance model** to encourage *quantitative* design in programs
- MPI makes it easy to emulate a BSP system
  - Rich set of collectives, barriers, blocking operations
- MPI (even MPI-1) sufficient for dynamic adaptive programming
  - The main issues are performance and “progress”
  - Improving implementations and better HW support for integrated CPU/NIC coordination the answer

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

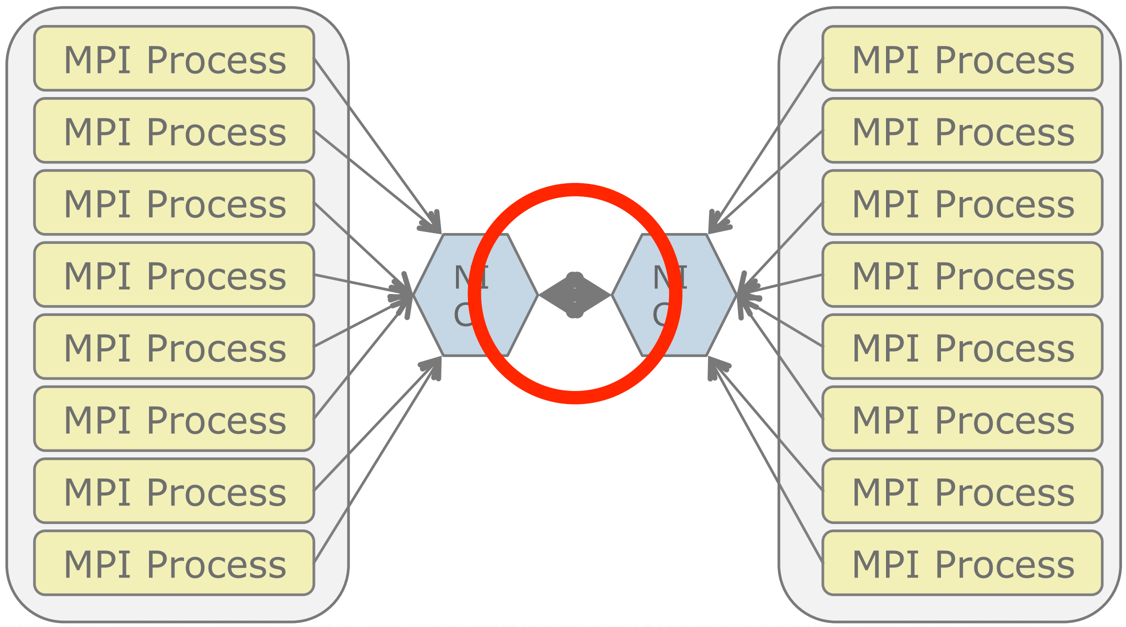




- Ping-pong between 2 nodes using 1-16 cores on each node
- Top is BG/Q, bottom Cray XE6
- “Classic” model predicts a single curve – rates independent of the number of communicating processes

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?

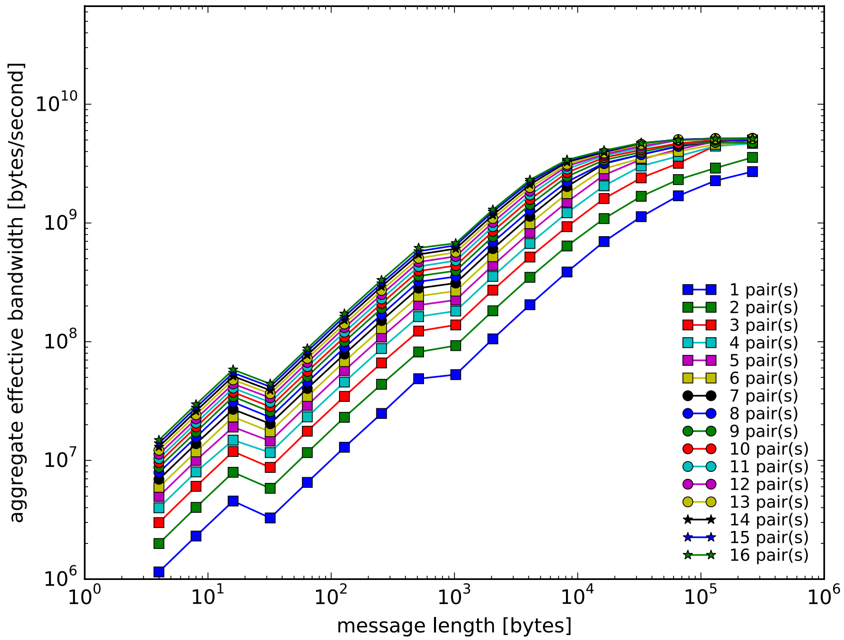


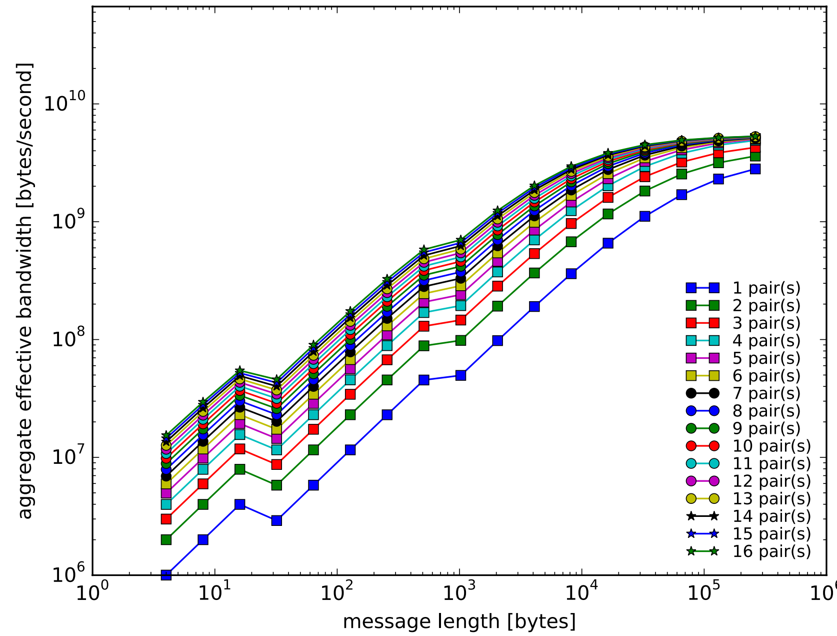
A Slightly Better Model

- For k processes sending messages, the sustained rate is
  - $\min(R_{NIC-NIC}, k\ R_{CORE-NIC})$
- 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
  - $T = s + k\ n/(k\ R_{CORE-NIC}) = s + n/R_{CORE-NIC}$

- KNL may need a similar term for s: $s+\max(0,(k-k_0)s_i)$, representing an incremental additional cost once more than k_0 concurrently communicating processes

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

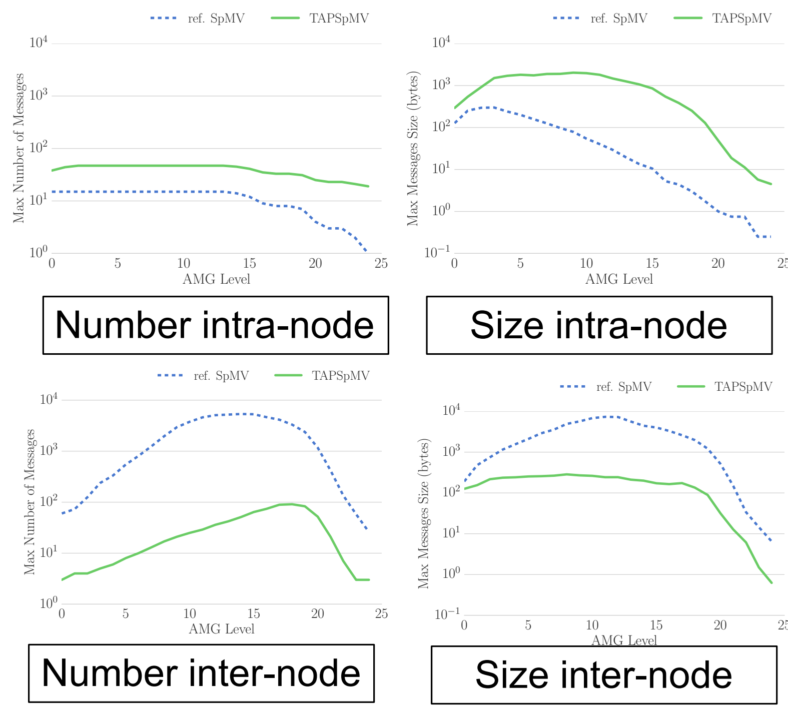
More Challenges For Extreme Scale Systems

- Simple MPI everywhere models hide important performance issues
  - Impacts algorithms – ex SpMV
- MPI implementations don't take nodes into account
  - Impacts memory overhead, data sharing
  - Process topology – Dims_create (for Cart_create) wrong API – ex nodecart
- File I/O bottlenecks
  - Metadata operations impact scaling, even for file/process (or should it be file per node?)
  - Need to monitor performance; avoid imposing too much order on operations – ex MeshIO
- Communication synchronization
  - Common "bogeyman" for extreme scale
  - But some of the best algorithms use, e.g., Allreduce
  - Reorder operations to reduce communication cost; permit overlap
  - Ex scalable CG algorithms and implementations

Node-Aware Sparse Matrix-Vector Product

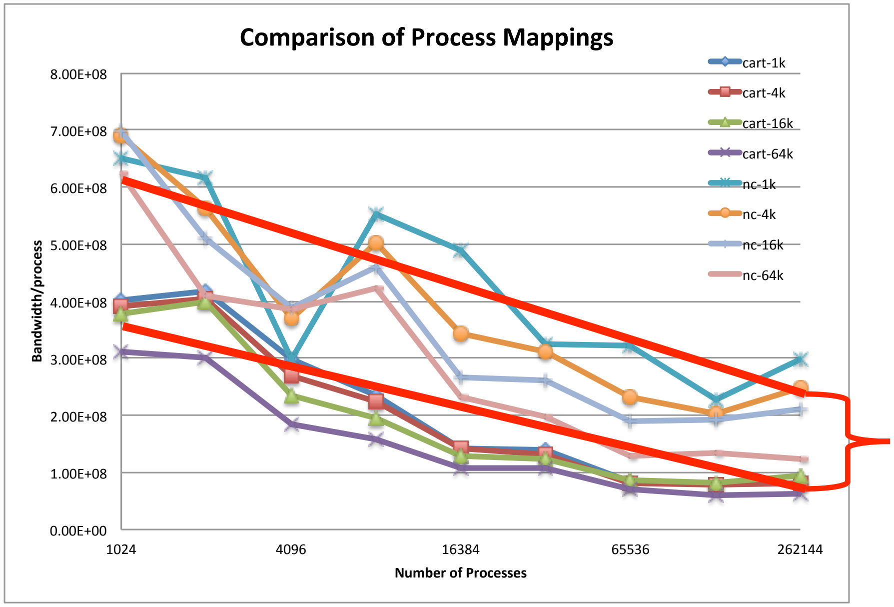
- Sparse matrix-vector products the core to many algorithms
  - E.g., in Krylov methods and in stencil application
- "Good" mappings of processes to nodes for locality also mean that the same data may be needed for different processes on the same node
- Can significantly improve performance by trading intra-node for internode communication…
- Work of Amand Bienz and Luke Olson

TAPSpMV Communication



MPI Process Topology: The Reality

- MPI provides a rich set of routines to allow the MPI implementation to map processes to physical hardware
- But in practice, behaves poorly or ignored (allowed by the standard)
- Halo exchange illustrates
  - Cart uses MPI_Cart_create
  - Nc is a user-implemented version that taeks noes into account
  - Nc is about 2x as fast
  - Note both have scaling problems (the network topology)



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

Scalable Preconditioned Conjugate Gradient Methods

- Reformulations of CG trade computation for the ability to overlap communication
- Hide communication costs and absorb noise to produce more consistent runtimes
- Must overlap allreduce with more matrix kernels as work per core decreases and communication costs increase
- Faster, more consistent runtimes in noisy environments
- Effective for simpler preconditioners and shows some speedups for more complex preconditioners without modifications
- Work of Paul Eller, “Scalable Non-blocking Preconditioned Conjugate Gradient Methods”, SC16 http://ieeexplore.ieee.org/document/7877096/

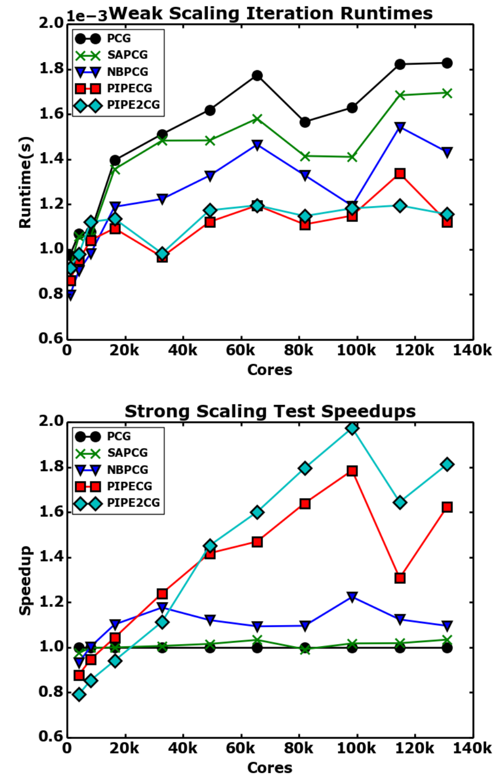


Figure: 27-point Poisson matrices with 4k rows per core (top) and 512^3 rows (bottom)

The hard part – Intranode perfomrnace

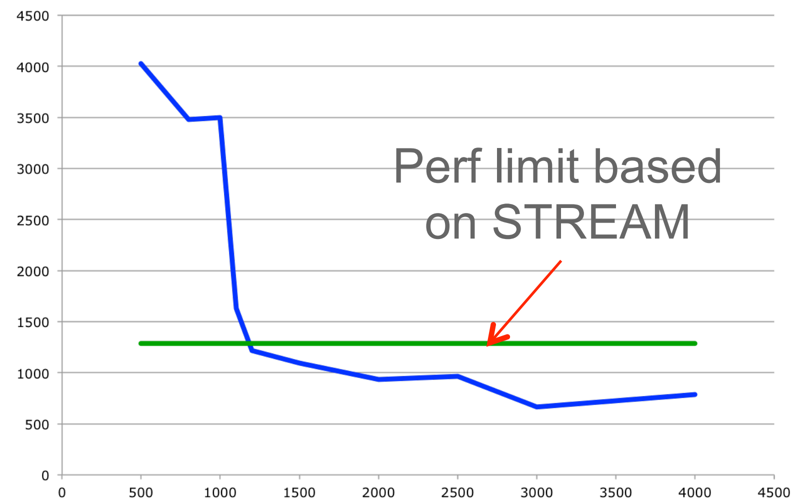
- This has alw*ays been* the hard part
  - In 1999, we achieved a 7x (!) improvement in performance for a scalable CFD code
    - This was *all in* the intranode performance
    - "Achieving high sustained performance in an unstructured mesh CFD application" https://dl.acm.org/citation.cfm?id=331600 , 1999; early analysis of memory limit to performance, key to GB award
- It is harder now
  - Good performance requires effective use of
    - Vector and other instructions
    - Cache and TLB
- Upcoming systems have
  - More complex memory systems
  - More and wider vector
  - Inter-thread synchronization
- And the community has mostly been in denial about this
  - Emphasis on fantasy solutions that provide magic performance
- For example…

Let The Compiler Do It

- This is the right answer …
  - If only the compiler *could* do it
- Lets look at one of the simplest operations for a single core, dense matrix transpose
  - Transpose involves only data motion; no floating point order to respect
  - Only a double loop (fewer options to consider)

A Simple Example: Dense Matrix Transpose

- 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

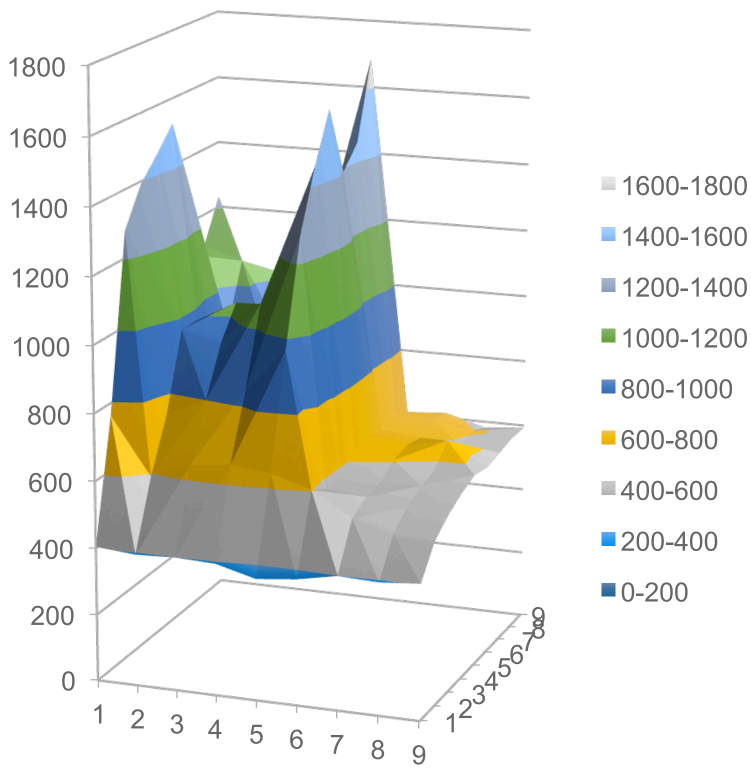


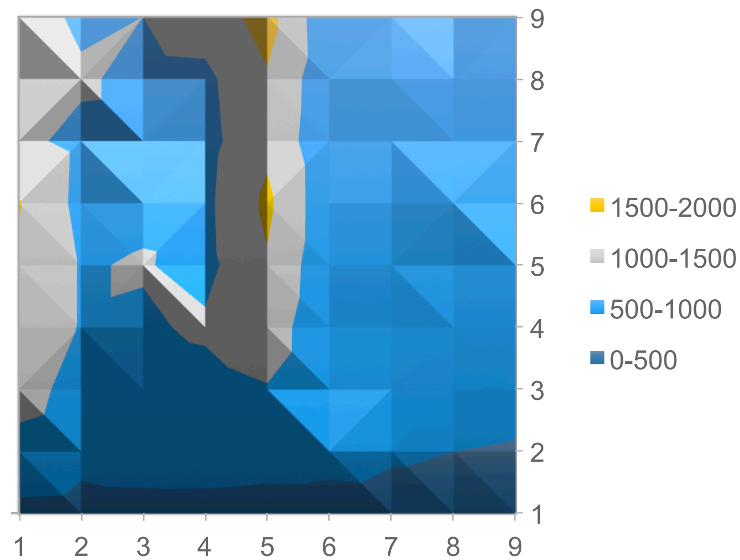
- Performance plummets when matrices no longer fit in cache

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 factor of 5 or more
- But what are the choices of stridei and stridej?
  ```

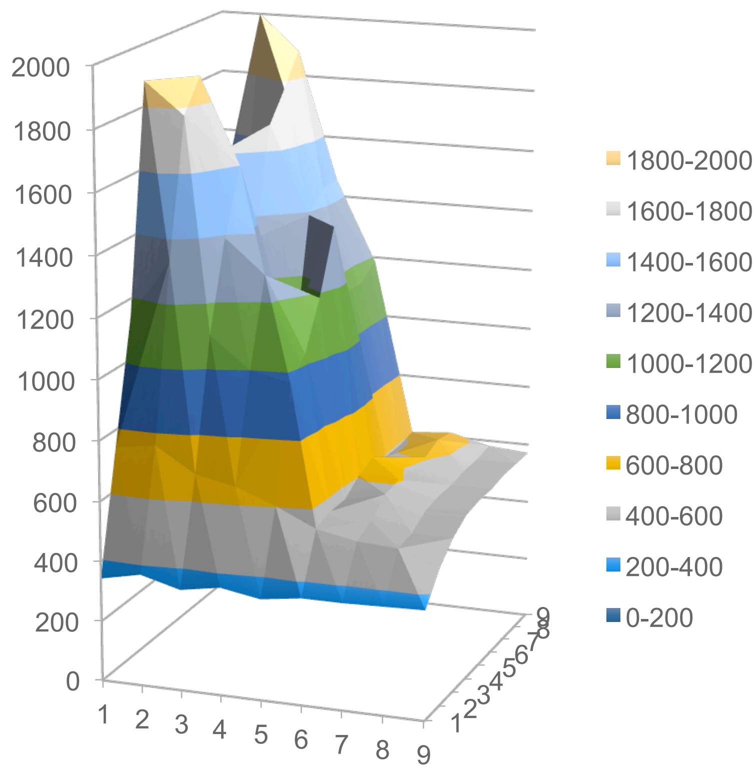
Results: Blue Waters O1

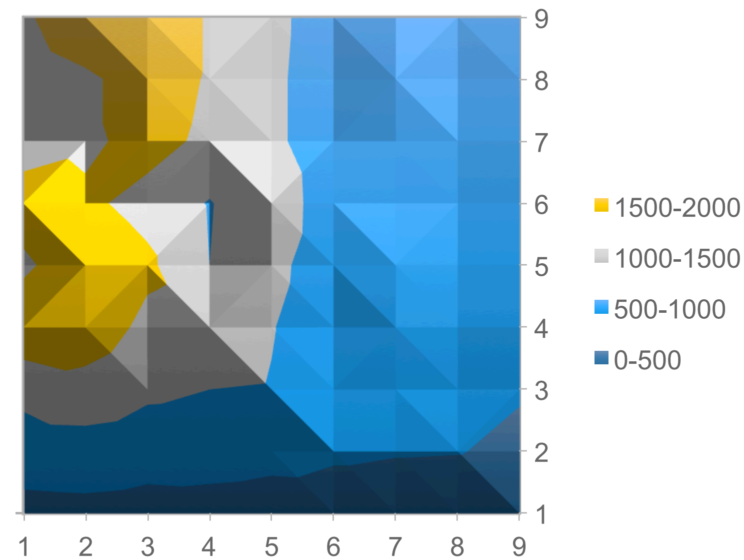




Results: Blue Waters O3

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





Some Different Approaches to Performance Portability

- Language based
  - Existing languages, possibly with additional information
    - Info from pragmas (e.g., align) or compile flags (assume associative)
  - Extensions, especially for parallelism
    - Directives + runtimes, e.g., OpenMP/OpenCL/OpenACC
    - May also relax constraints, e.g., for operation order, bitwise reproducibility
  - New languages, especially targeted at
    - Specific data structures and operations
    - Specific problem domains
- Library based (define mathematical operators and implement those efficiently)
  - Specific data structure/operations (e.g., DGEMM)
  - Specific operations with families of data structures (e.g., PETSc)
    - This is likely the most practical way to include data-structure and even algorithm choice
    - At the cost of pushing the performance portability problem onto the library developers

Some Different Approaches to Performance Portability

- Tools based
  - Recognize that the user can always write poorly-performing code
  - Support programming in finding and fixing performance problems
  - Example: Early vectorizing compilers gave feedback about missed vectorization opportunities; trained programmer to write “better” code
- Programmer support and solution components
  - Work with programmer to develop code
  - Source-to-source tools to transform and to generate code under programmer guidance
  - Autotuning to select from families of code
  - Database systems to manage architecture and/or system-specific derivatives
- Magic
  - Any sufficiently advanced technology is indistinguishable from magic. (Clarke’s 3rd law)
  - Any sufficiently advanced technology is indistinguishable from a rigged demo.
- Note these approaches are not orthogonal
  - Successful performance portability requires many approaches, working together
- For example…

An Example: Stencil Code from a Real Application

- Stencil for CFD code
- Supports 2D and 3D
- Supports different stencil widths
- Matches computational scientists' view of the mathematics

```
! @ICE block=StrainRate
do i = 1,ND
  do k = 1,ND ! diagonal components first
    do ii = 1, Nc
      StrnRt(ii,i) = StrnRt(ii,i) + &
        MT1(ii,i+k*ND-2) * VelGradInt(ii,i+k*ND-2)
    end do
  end do ! k
  do j = i+1,ND ! upper-half part of strain-rate tensor due to symmetry
    do k = 1,ND
      do ii = 1, Nc
        StrnRt(ii,i+j+ND-2) = StrnRt(ii,i+j+ND-2) + &
          MT1(ii,k+j+ND-2) * VelGradInt(ii,i+k*ND-2) + &
          MT1(ii,k+i+ND-2) * VelGradInt(ii,j+k*ND-2)
      end do
    end do ! k
    do ii = 1, Nc
      StrnRt(ii,i+j+ND-2) = 0.5_rfreal * StrnRt(ii,i+j+ND-2)
    end do
  end do ! j
end do ! i
do k = 1,size(StrnRt,2)
  do ii = 1, Nc
    StrnRt(ii,k) = JAC(ii) * StrnRt(ii,k)
  end do
end do ! k
! @ICE endblock
```

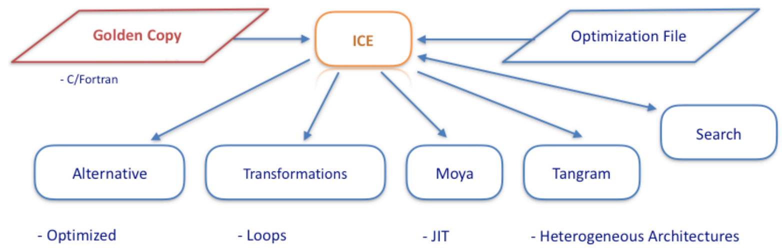
Another Version of the Same Code

- This version is 4X as fast as the simpler, easier to read code
- Less general code (subset to stencil, problem dimension)
- Same algorithm, data structure, and operations, but transformed to aid compiler in generating fast (and vectorized) code

```
if( ND == 2 ) then
  do ii = 1, Nc
    ! diagonal components first
    StrnRt(ii,1) = JAC(ii) * (                    &
      MT1(ii,1) * VelGradist(ii,1)                &
      + MT1(ii,2) * VelGradist(ii,3) )
    StrnRt(ii,2) = JAC(ii) * (                    &
      MT1(ii,3) * VelGradist(ii,2)                &
      + MT1(ii,4) * VelGradist(ii,4) )
    StrnRt(ii,3) = JAC(ii) * 0.5_rfreal * (       &
      MT1(ii,3) * VelGradist(ii,1)                &
      + MT1(ii,1) * VelGradist(ii,2)              &
      + MT1(ii,4) * VelGradist(ii,3)              &
      + MT1(ii,2) * VelGradist(ii,4) )
  end do
else if( ND == 3 ) then
  do ii = 1, Nc
    ! diagonal components first
    StrnRt(ii,1) = JAC(ii)   * (                  &
      MT1(ii,1) * VelGradist(ii,1)                &
      + MT1(ii,2) * VelGradist(ii,4)              &
      + MT1(ii,3) * VelGradist(ii,7) )
    StrnRt(ii,4) = JAC(ii) * 0.5_rfreal * (       &
      MT1(ii,4) * VelGradist(ii,1)                &
      + MT1(ii,1) * VelGradist(ii,2)              &
      + MT1(ii,5) * VelGradist(ii,4)              &
      + MT1(ii,2) * VelGradist(ii,5)              &
      + MT1(ii,6) * VelGradist(ii,7)              &
      + MT1(ii,3) * VelGradist(ii,8) )
  end do
  do ii = 1, Nc
    StrnRt(ii,2) = JAC(ii) * (                    &
      MT1(ii,4) * VelGradist(ii,2)                &
      + MT1(ii,5) * VelGradist(ii,5)              &
      + MT1(ii,6) * VelGradist(ii,8) )
    StrnRt(ii,6) = JAC(ii) * 0.5_rfreal * (       &
      MT1(ii,7) * VelGradist(ii,2)                &
      + MT1(ii,4) * VelGradist(ii,3)              &
      + MT1(ii,8) * VelGradist(ii,5)              &
      + MT1(ii,5) * VelGradist(ii,6)              &
      + MT1(ii,9) * VelGradist(ii,8)              &
      + MT1(ii,6) * VelGradist(ii,9) )
  end do
  do ii = 1, Nc
    StrnRt(ii,3) = JAC(ii) * (                    &
      MT1(ii,7) * VelGradist(ii,3)                &
      + MT1(ii,8) * VelGradist(ii,6)              &
      + MT1(ii,9) * VelGradist(ii,9) )
    StrnRt(ii,5) = JAC(ii) * 0.5_rfreal * (       &
      MT1(ii,7) * VelGradist(ii,1)                &
      + MT1(ii,1) * VelGradist(ii,3)              &
      + MT1(ii,8) * VelGradist(ii,4)              &
      + MT1(ii,2) * VelGradist(ii,6)              &
      + MT1(ii,9) * VelGradist(ii,7)              &
      + MT1(ii,3) * VelGradist(ii,9) )
  end do
end if
```

Illinois Coding Environment (ICE)

- One pragmatic approach
- Assumptions
  - Fast code requires some expert intervention
  - Can't all be done at compile time
  - Original code (in standard language) is maintained as reference
  - Can add information about computation to code
- Center for Exascale Simulation of Plasma-Coupled Combustion
  - http://xpacc.illinois.edu



- Approach
  - Annotations provide additional descriptive information
    - Block name, expected loop sizes, etc.
  - Source-to-source transformations used to create code for compiler
    - Exploit tool ecosystem – interface to existing tools
    - Original "Golden Copy" used for development, correctness checks
  - Database used to manage platform-specific versions; detect changes that invalidate transformed versions

Example: Dense Matrix Multiply

- Matrix Multiplication

```
#pragma @ICE loop=matmul
  for (i = 0; i < n; i++)
    for (j = 0; j < n; j++)
      for (k = 0; k < n; k++)
         mC[i][j] += mA[i][k] * mB[k][j];
#pragma @ICE endloop
```

+

```
---
#Compilation command before tests
buildcmd: make realclean;  make CC={compiler} COPT={params}

search:
  tool: opentuner
  time-limit: 30000
  variants-limit: 1000

buildoptions:
   gcc:
        params:{'-O':{'default': 3, 'min': 0, 'max': 3}}

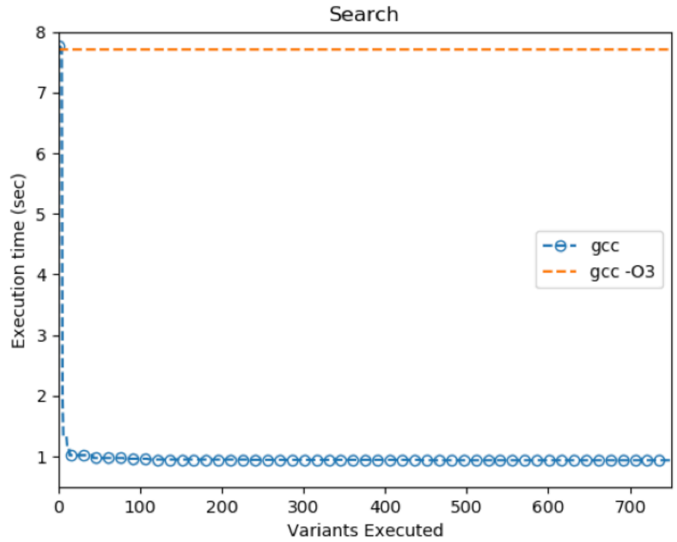
#Command call for each test
runcmd: ./mmc

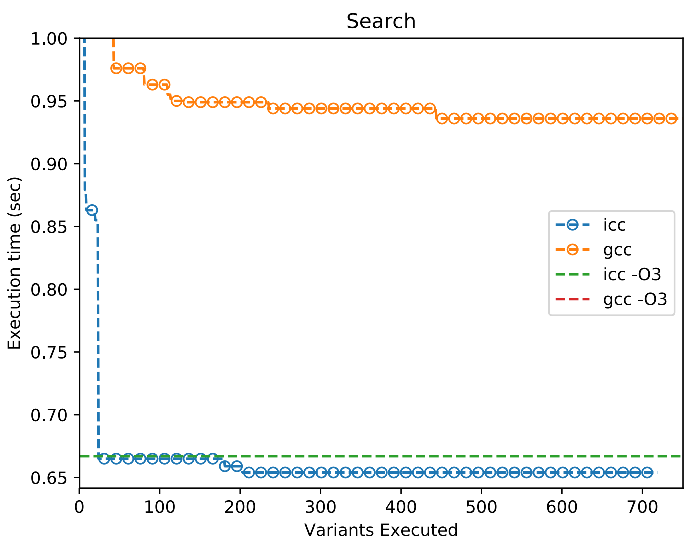
tuning: on

matmul:
      rose_uiuc:
          - stripmine+:
              loop: 3
              factor: 2..36
          - stripmine+:
              loop: 2
              factor: 2..48
          - interchange+:
              order: 1,3,0,2,4
          - unroll*:
              loop: 5
              factor: 2..24
…
```

Performance Results

- Dense matrix-matrix multiply
  - 302,680 total variants
  - Subset evaluated (based on results-so-far)
  - 8.2x speedup over gcc compiler with optimization
  - Small but consistent speedup over icc –O3
- Different parameters can be selected/remembered for each platform
  - Within the constraints of the performance parameters considered





Stencil 3D

```
#pragma @ICE loop=stencil
for(i = 1; i < x-1; i++) {
  for(j = 1; j < y-1; j++) {
    for(k = 1; k < z-1; k++) {
      B[i][j][k] = C0 * A[i][j][k] + C1 * (
                   A[i+1][j][k] + A[i-1][j][k] +
                   A[i][j+1][k] + A[i][j-1][k] +
                   A[i][j][k+1] + A[i][j][k-1]);
    }
  }
}
#pragma @ICE endloop
```

+

```
---
#Built command before compilation
prebuildcmd:

#Compilation command before tests
buildcmd:
    make realclean;  make CC={compiler} COPT={params}

buildoptions:
   gcc:
        params:{'-O':{'default': 3,'min': 0,'max': 3}}
   icc:
        params:{'-O':{'default': 3,'min': 0,'max': 3}}

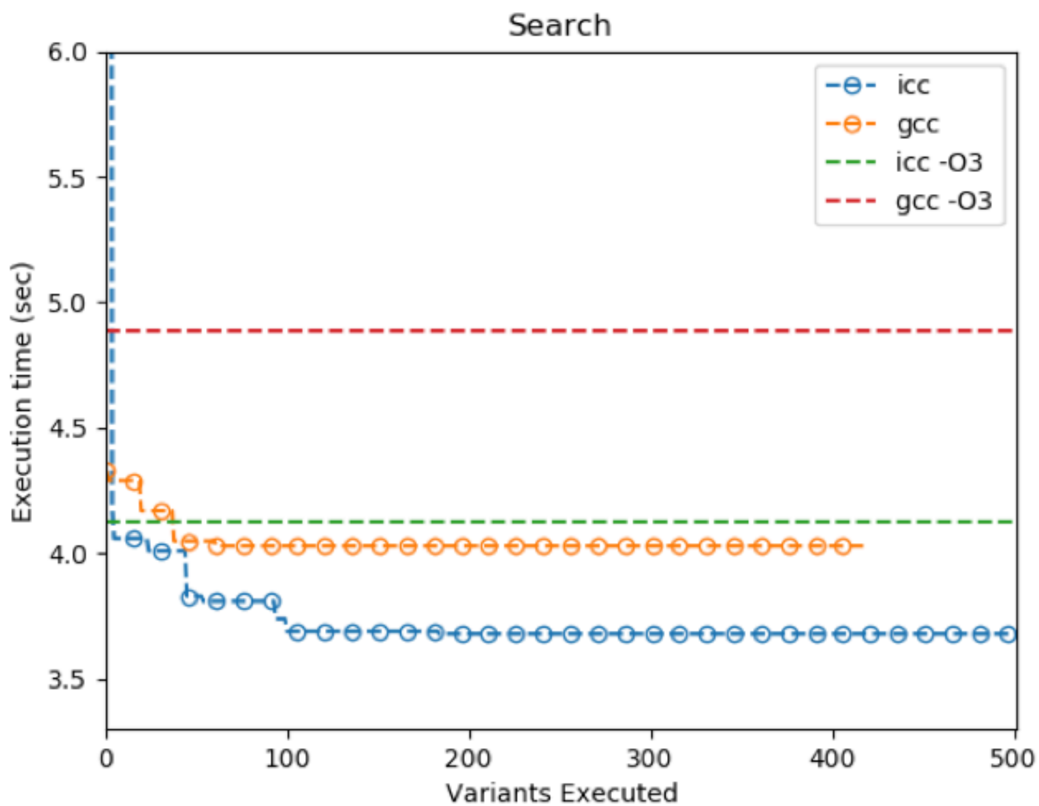
#Command call for each test
runcmd: ./sten3d 1024 20

tuning: on

stencil:
       rose_uiuc:
          - stripmine+:
              loop: 4
              factor: 16..1024
                type: poweroftwo
          - stripmine+:
              loop: 3
              factor: 16..1024
                type: poweroftwo
          - stripmine+:
              loop: 2
              factor: 16..1024
                type: poweroftwo
          - interchange+:
              order:0,1,3,5,2,4,6
...
```

Performance Results

- 3-D Stencil
  - 11,664 variants
  - Max 12.6 sec
  - Min 3.68 sec
  - Speedup over simple code
    - icc: 1.12x
    - gcc: 1.21x



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

X = MPI (or X = ϕ)

- MPI 3.1 features esp. important for Exascale
  - Generalize collectives to encourage post BSP (Bulk Synchronous Programming) approach:
    - Nonblocking collectives
    - Neighbor – including nonblocking – collectives
  - Enhanced one-sided
    - Precisely specified (see “Remote Memory Access Programming in MPI-3,” Hoefler et at, in ACM TOPC)
    - http://dl.acm.org/citation.cfm?doid=2780584
    - Many more operations including RMW
  - Enhanced thread safety

X = Programming with Threads

- Many choices, different user targets and performance goals
  - Libraries: Pthreads, TBB
  - Languages: OpenMP 4, C11/C++11
- C11 provides an adequate (and thus complex) memory model to write portable thread code
  - Also needed for MPI-3 shared memory; see “Threads cannot be implemented as a library”, http://www.hpl.hp.com/techreports/2004/HPL-2004-209.html
  - Also see “You don’t know Jack about Shared Variables or Memory Models”, CACM Vol 55#2, Feb 2012

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

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

Some Resources to Negotiate

- CPU resources
  - Threads and contexts
  - Cores (incl placement)
  - Cache
- Memory resources
  - HBM, NVRAM
  - Prefetch, outstanding load/ stores
  - Pinned pages or equivalent NIC needs
  - Transactional memory regions
  - Memory use (buffers)
- NIC resources
  - Collective groups
  - Routes
  - Power
- OS resources
  - Synchronization hardware
  - Scheduling
  - Virtual memory
  - Cores (dark silicon)

Two Viewpoints on Programming Systems

- Single Unified System
  - Examples
    - UPC, Python, Fortran (with CoArrays), Chapel
  - Pro
    - Can be simpler for user
      - Single set of concepts applies to everything
    - System has complete control – all productivity and performance optimizations enabled
  - Con
    - May be limited to problem types (e.g., structured grids)
    - Gap between promise and delivery in performance due to complexity
- Composed system
  - Examples
    - MPI+OpenMP, Python+C, PETSc + C
  - Pro
    - Can be simpler for user
      - Concepts match each component's domain
    - Implementation simplicity – each piece smaller, more limited domain
  - Con
    - Hard to impossible to integrate across components
    - Limits optimization opportunities

Summary

- Challenges for Exascale 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 system
  - But some useful features still missing, especially WRT notification, and implementations don’t realize available performance
- 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 (e.g., MPI+X) is a practical approach
  - But requires close attention to “+”

Thanks!

- Philipp Samfass, Ed Karrels, Amanda Bienz, Paul Eller, Thiago Teixeira
- Luke Olson, David Padua
- Department of Energy, National Nuclear Security Administration, under Award Number DE-NA0002374
- ExxonMobile 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